
Linux Networking Documentation

Release 6.8.0

The kernel development community

Jan 16, 2026

CONTENTS

1	AF_XDP	3
1.1	Overview	3
1.2	Concepts	4
1.3	Libbpf	6
1.4	XSKMAP / BPF_MAP_TYPE_XSKMAP	6
1.5	Configuration Flags and Socket Options	7
1.6	Multi-Buffer Support	10
1.7	Sample application	15
1.8	FAQ	16
1.9	Credits	17
2	Bare UDP Tunnelling Module Documentation	19
2.1	Special Handling	19
2.2	Usage	19
3	batman-adv	21
3.1	Configuration	21
3.2	Usage	22
3.3	Logging/Debugging	23
3.4	batctl	23
3.5	Contact	23
4	SocketCAN - Controller Area Network	25
4.1	Overview / What is SocketCAN	25
4.2	Motivation / Why Using the Socket API	25
4.3	SocketCAN Concept	26
4.4	How to use SocketCAN	28
4.5	SocketCAN Core Module	41
4.6	CAN Network Drivers	42
4.7	SocketCAN Resources	49
4.8	Credits	49
5	The UCAN Protocol	51
5.1	USB Endpoints	51
5.2	CONTROL Messages	51
5.3	IN Message Format	54
5.4	OUT Message Format	55
5.5	CAN Error Handling	56
5.6	Example Conversation	57

6	Hardware Device Drivers	59
6.1	Asynchronous Transfer Mode (ATM) Device Drivers	59
6.2	Cable Modem Device Drivers	66
6.3	Controller Area Network (CAN) Device Drivers	69
6.4	Cellular Modem Device Drivers	93
6.5	Ethernet Device Drivers	96
6.6	Fiber Distributed Data Interface (FDDI) Device Drivers	372
6.7	Amateur Radio Device Drivers	377
6.8	Wi-Fi Device Drivers	391
6.9	WWAN Device Drivers	404
7	Distributed Switch Architecture	409
7.1	Architecture	409
7.2	Broadcom RoboSwitch Ethernet switch driver	428
7.3	Broadcom Starfighter 2 Ethernet switch driver	431
7.4	LAN9303 Ethernet switch driver	433
7.5	NXP SJA1105 switch driver	434
7.6	DSA switch configuration from userspace	442
8	Linux Devlink Documentation	451
8.1	Locking	451
8.2	Nested instances	451
8.3	Interface documentation	452
8.4	Driver-specific documentation	491
9	CAIF	519
9.1	Linux CAIF	519
9.2	Using Linux CAIF	522
10	Netlink interface for ethtool	525
10.1	Basic information	525
10.2	Conventions	525
10.3	Request header	526
10.4	Bit sets	526
10.5	List of message types	528
10.6	STRSET_GET	530
10.7	LINKINFO_GET	531
10.8	LINKINFO_SET	532
10.9	LINKMODES_GET	532
10.10	LINKMODES_SET	533
10.11	LINKSTATE_GET	533
10.12	DEBUG_GET	536
10.13	DEBUG_SET	536
10.14	WOL_GET	536
10.15	WOL_SET	537
10.16	FEATURES_GET	537
10.17	FEATURES_SET	538
10.18	PRIVFLAGS_GET	538
10.19	PRIVFLAGS_SET	539
10.20	RINGS_GET	539
10.21	RINGS_SET	540
10.22	CHANNELS_GET	541

10.23 CHANNELS_SET	541
10.24 COALESCE_GET	541
10.25 COALESCE_SET	543
10.26 PAUSE_GET	544
10.27 PAUSE_SET	545
10.28 EEE_GET	545
10.29 EEE_SET	546
10.30 TSINFO_GET	546
10.31 CABLE_TEST	547
10.32 CABLE_TEST TDR	547
10.33 TUNNEL_INFO	549
10.34 FEC_GET	550
10.35 FEC_SET	552
10.36 MODULE_EEPROM_GET	552
10.37 STATS_GET	553
10.38 PHC_VCLOCKS_GET	554
10.39 MODULE_GET	554
10.40 MODULE_SET	555
10.41 PSE_GET	556
10.42 PSE_SET	557
10.43 RSS_GET	557
10.44 PLCA_GET_CFG	558
10.45 PLCA_SET_CFG	559
10.46 PLCA_GET_STATUS	560
10.47 MM_GET	560
10.48 MM_SET	563
10.49 Request translation	564
11 IEEE 802.15.4 Developer's Guide	567
11.1 Introduction	567
11.2 Socket API	567
11.3 6LoWPAN Linux implementation	568
11.4 Drivers	568
11.5 Device drivers API	569
12 J1939 Documentation	571
12.1 Overview / What Is J1939	571
12.2 Motivation	571
12.3 J1939 concepts	572
12.4 How to Use J1939	573
13 Linux Networking and Network Devices APIs	579
13.1 Linux Networking	579
13.2 Network device support	699
14 MSG_ZEROCOPY	873
14.1 Intro	873
14.2 Interface	874
14.3 Implementation	876
14.4 Testing	877
15 FAILOVER	879

15.1	Overview	879
16	Net DIM - Generic Network Dynamic Interrupt Moderation	881
16.1	Assumptions	881
16.2	Introduction	881
16.3	Net DIM Algorithm	882
16.4	Registering a Network Device to DIM	883
16.5	Example	883
16.6	Dynamic Interrupt Moderation (DIM) library API	884
17	NET_FAILOVER	893
17.1	Overview	893
17.2	virtio-net accelerated datapath: STANDBY mode	893
17.3	Live Migration of a VM with SR-IOV VF & virtio-net in STANDBY mode	895
18	Page Pool API	897
18.1	Architecture overview	898
18.2	Monitoring	898
18.3	API interface	898
18.4	Coding examples	905
19	PHY Abstraction Layer	909
19.1	Purpose	909
19.2	The MDIO bus	909
19.3	(RG)MII/electrical interface considerations	910
19.4	Connecting to a PHY	911
19.5	Letting the PHY Abstraction Layer do Everything	912
19.6	PHY interface modes	913
19.7	Pause frames / flow control	914
19.8	Keeping Close Tabs on the PAL	914
19.9	Doing it all yourself	915
19.10	PHY Device Drivers	916
19.11	Board Fixups	917
19.12	Standards	917
20	phylink	919
20.1	Overview	919
20.2	Modes of operation	919
20.3	Rough guide to converting a network driver to sfp/phylink	920
21	IP-Aliasing	925
21.1	Alias creation	925
21.2	Alias deletion	925
21.3	Alias (re-)configuring	925
21.4	Relationship with main device	926
22	Ethernet Bridging	927
22.1	Introduction	927
22.2	Bridge kAPI	927
22.3	Bridge uAPI	929
22.4	STP	937
22.5	VLAN	939
22.6	Multicast	940

22.7	Switchdev	941
22.8	Netfilter	941
22.9	Other Features	942
22.10	FAQ	942
22.11	Contact Info	942
22.12	External Links	942
23	SNMP counter	943
23.1	General IPv4 counters	943
23.2	ICMP counters	944
23.3	General TCP counters	946
23.4	TCP Fast Open	948
23.5	TCP Fast Path	948
23.6	TCP abort	949
23.7	TCP Hybrid Slow Start	950
23.8	TCP retransmission and congestion control	950
23.9	DSACK	952
23.10	invalid SACK and DSACK	952
23.11	SACK shift	953
23.12	TCP out of order	953
23.13	TCP PAWS	953
23.14	TCP ACK skip	954
23.15	TCP receive window	955
23.16	Delayed ACK	955
23.17	Tail Loss Probe (TLP)	955
23.18	TCP Fast Open description	956
23.19	SYN cookies	956
23.20	Challenge ACK	957
23.21	prune	957
23.22	examples	958
24	Checksum Offloads	973
24.1	Introduction	973
24.2	TX Checksum Offload	973
24.3	LCO: Local Checksum Offload	974
24.4	RCO: Remote Checksum Offload	975
25	Segmentation Offloads	977
25.1	Introduction	977
25.2	TCP Segmentation Offload	977
25.3	UDP Fragmentation Offload	978
25.4	IPIP, SIT, GRE, UDP Tunnel, and Remote Checksum Offloads	978
25.5	Generic Segmentation Offload	979
25.6	Generic Receive Offload	979
25.7	Partial Generic Segmentation Offload	979
25.8	SCTP acceleration with GSO	979
26	Scaling in the Linux Networking Stack	981
26.1	Introduction	981
26.2	RSS: Receive Side Scaling	981
26.3	RPS: Receive Packet Steering	983
26.4	RFS: Receive Flow Steering	986

26.5	Accelerated RFS	987
26.6	XPS: Transmit Packet Steering	988
26.7	Per TX Queue rate limitation	990
26.8	Further Information	990
27	Kernel TLS	991
27.1	Overview	991
27.2	User interface	991
27.3	Statistics	995
28	Kernel TLS offload	997
28.1	Kernel TLS operation	997
28.2	Device configuration	999
28.3	Normal operation	1000
28.4	Resync handling	1001
28.5	Error handling	1003
28.6	Performance metrics	1004
28.7	Statistics	1004
28.8	Notable corner cases, exceptions and additional requirements	1005
29	In-Kernel TLS Handshake	1007
29.1	Overview	1007
29.2	User handshake agent	1007
29.3	Kernel Handshake API	1007
29.4	Handshake Completion	1009
30	Linux NFC subsystem	1011
30.1	Architecture overview	1011
30.2	Device Driver Interface	1012
30.3	Userspace interface	1012
31	Netdev private dataroom for 6lowpan interfaces	1015
32	6pack Protocol	1017
32.1	1. What is 6pack, and what are the advantages to KISS?	1017
32.2	2. Who has developed the 6pack protocol?	1018
32.3	3. Where can I get the latest version of 6pack for LinuX?	1018
32.4	4. Preparing the TNC for 6pack operation	1018
32.5	5. Building and installing the 6pack driver	1018
32.6	How to turn on 6pack support:	1019
32.7	6. Known problems	1020
33	ARCnet Hardware	1021
33.1	Introduction to ARCnet	1021
33.2	Cabling ARCnet Networks	1022
33.3	Setting the Jumpers	1025
33.4	Unclassified Stuff	1028
33.5	Standard Microsystems Corp (SMC)	1029
33.6	Possibly SMC	1039
33.7	PureData Corp	1041
33.8	CNet Technology Inc. (8-bit cards)	1044
33.9	CNet Technology Inc. (16-bit cards)	1048
33.10	Lantech	1051

33.11 Acer	1053
33.12 Datapoint?	1056
33.13 Topware	1059
33.14 Thomas-Conrad	1061
33.15 Waterloo Microsystems Inc. ??	1063
33.16 No Name	1065
33.17 Tiara	1078
33.18 Other Cards	1079
34 ARCnet	1081
34.1 Where do I discuss these drivers?	1082
34.2 Other Drivers and Info	1082
34.3 Installing the Driver	1082
34.4 Loadable Module Support	1084
34.5 Using the Driver	1084
34.6 Multiple Cards in One Computer	1085
34.7 How do I get it to work with...?	1085
34.8 Using Multiprotocol ARCnet	1086
34.9 It works: what now?	1090
34.10 It doesn't work: what now?	1090
34.11 I want to send money: what now?	1091
35 ATM	1093
36 AX.25	1095
37 Linux Ethernet Bonding Driver HOWTO	1097
37.1 Introduction	1097
37.2 1. Bonding Driver Installation	1098
37.3 2. Bonding Driver Options	1098
37.4 3. Configuring Bonding Devices	1111
37.5 4 Querying Bonding Configuration	1124
37.6 5. Switch Configuration	1125
37.7 6. 802.1q VLAN Support	1126
37.8 7. Link Monitoring	1126
37.9 8. Potential Sources of Trouble	1128
37.10 9. SNMP agents	1129
37.11 10. Promiscuous mode	1130
37.12 11. Configuring Bonding for High Availability	1131
37.13 12. Configuring Bonding for Maximum Throughput	1132
37.14 13. Switch Behavior Issues	1137
37.15 14. Hardware Specific Considerations	1138
37.16 15. Frequently Asked Questions	1140
37.17 16. Resources and Links	1142
38 cdc_mbim - Driver for CDC MBIM Mobile Broadband modems	1143
38.1 Command Line Parameters	1143
38.2 Basic usage	1144
38.3 MBIM control channel userspace ABI	1144
38.4 MBIM data channel userspace ABI	1146
38.5 References	1148

39 DCCP protocol	1151
39.1 Introduction	1151
39.2 Missing features	1151
39.3 Socket options	1151
39.4 Sysctl variables	1153
39.5 IOCTLS	1154
39.6 Other tunables	1154
39.7 Notes	1154
40 DCTCP (DataCenter TCP)	1155
41 DNS Resolver Module	1157
41.1 Overview	1157
41.2 Compilation	1157
41.3 Setting up	1157
41.4 Usage	1158
41.5 Reading DNS Keys from Userspace	1159
41.6 Mechanism	1159
41.7 Debugging	1159
42 Softnet Driver Issues	1161
42.1 Probing guidelines	1161
42.2 Close/stop guidelines	1161
42.3 Transmit path guidelines	1161
43 EQL Driver: Serial IP Load Balancing HOWTO	1165
43.1 1. Introduction	1165
43.2 2. Kernel Configuration	1166
43.3 3. Network Configuration	1166
43.4 4. About the Slave Scheduler Algorithm	1168
43.5 5. Testers' Reports	1169
44 LC-trie implementation notes	1173
44.1 Node types	1173
44.2 A few concepts explained	1173
44.3 Comments	1174
44.4 Locking	1175
44.5 Main lookup mechanism	1175
45 Linux Socket Filtering aka Berkeley Packet Filter (BPF)	1177
45.1 Notice	1177
45.2 Introduction	1177
45.3 Structure	1178
45.4 Example	1178
45.5 BPF engine and instruction set	1180
45.6 JIT compiler	1186
45.7 BPF kernel internals	1188
45.8 Testing	1188
45.9 Misc	1189
45.10 Written by	1189
46 Generic HDLC layer	1191
46.1 Board-specific issues	1193

47 Generic Netlink	1195
48 Generic networking statistics for netlink users	1197
48.1 Collecting:	1197
48.2 Export to userspace (Dump):	1197
48.3 TCA_STATS/TCA_XSTATS backward compatibility:	1198
48.4 Locking:	1198
48.5 Rate Estimator:	1198
48.6 Authors:	1199
49 The Linux kernel GTP tunneling module	1201
49.1 What is GTP	1201
49.2 The Linux GTP tunnelling module	1202
49.3 Userspace Programs with Linux Kernel GTP-U support	1202
49.4 Userspace Library / Command Line Utilities	1202
49.5 Protocol Versions	1202
49.6 IPv6	1203
49.7 Mailing List	1203
49.8 Issue Tracker	1203
49.9 History / Acknowledgements	1203
49.10 Architectural Details	1203
49.11 APN vs. Network Device	1204
50 Identifier Locator Addressing (ILA)	1207
50.1 Introduction	1207
50.2 ILA terminology	1207
50.3 Operation	1208
50.4 Transport checksum handling	1209
50.5 Identifier types	1209
50.6 Identifier formats	1210
50.7 Configuration	1211
50.8 Some examples	1211
51 IOAM6 Sysfs variables	1213
51.1 /proc/sys/net/conf/<iface>/ioam6_* variables:	1213
52 IP dynamic address hack-port v0.03	1215
53 IPsec	1217
54 IP Sysctl	1219
54.1 /proc/sys/net/ipv4/* Variables	1219
54.2 INET peer storage	1223
54.3 TCP variables	1224
54.4 UDP variables	1237
54.5 RAW variables	1238
54.6 CIPSOv4 Variables	1238
54.7 IP Variables	1239
54.8 /proc/sys/net/ipv6/* Variables	1249
54.9 icmp/*:	1261
54.10 /proc/sys/net/bridge/* Variables:	1262
54.11 proc/sys/net/sctp/* Variables:	1263
54.12 /proc/sys/net/core/*	1269

54.13 /proc/sys/net/unix/*	1269
55 IPv6	1271
56 IPVLAN Driver HOWTO	1273
56.1 1. Introduction:	1273
56.2 2. Building and Installation:	1273
56.3 3. Configuration:	1273
56.4 4. Operating modes:	1274
56.5 5. Mode flags:	1274
56.6 6. What to choose (macvlan vs. ipvlan)?	1275
56.7 6. Example configuration:	1275
57 IPvs-sysctl	1277
57.1 /proc/sys/net/ipv4/vs/* Variables:	1277
58 Kernel Connection Multiplexor	1283
58.1 KCM sockets	1284
58.2 Multiplexor	1284
58.3 TCP sockets & Psocks	1284
58.4 Connected mode semantics	1284
58.5 Socket types	1284
58.6 User interface	1285
58.7 Use in applications	1287
59 L2TP	1289
59.1 Overview	1289
59.2 L2TP APIs	1290
59.3 Internal Implementation	1297
59.4 Miscellaneous	1299
60 The Linux LAPB Module Interface	1301
60.1 Structures	1301
60.2 LAPB Initialisation Structure	1301
60.3 LAPB Parameter Structure	1302
60.4 Functions	1303
60.5 Callbacks	1304
61 How to use packet injection with mac80211	1307
62 Management Component Transport Protocol (MCTP)	1309
62.1 Structure: interfaces & networks	1309
62.2 Sockets API	1309
62.3 Kernel internals	1313
63 MPLS Sysfs variables	1315
63.1 /proc/sys/net/mpls/* Variables:	1315
64 MPTCP Sysfs variables	1317
64.1 /proc/sys/net/mptcp/* Variables	1317
65 HOWTO for multiqueue network device support	1319
65.1 Section 1: Base driver requirements for implementing multiqueue support	1319

65.2	Section 2: Qdisc support for multiqueue devices	1319
65.3	Section 3: Brief howto using MULTIQ for multiqueue devices	1320
66	NAPI	1321
66.1	Driver API	1321
66.2	User API	1324
67	Common Networking Struct Cachelines	1327
67.1	inet_connection_sock struct fast path usage breakdown	1327
67.2	inet_sock struct fast path usage breakdown	1328
67.3	net_device struct fast path usage breakdown	1328
67.4	netns_ipv4 struct fast path usage breakdown	1330
67.5	netns_ipv4 enum fast path usage breakdown	1331
67.6	tcp_sock struct fast path usage breakdown	1333
68	Netconsole	1337
68.1	Introduction:	1337
68.2	Sender and receiver configuration:	1337
68.3	Dynamic reconfiguration:	1338
68.4	Extended console:	1340
68.5	Miscellaneous notes:	1340
69	Netdev features mess and how to get out from it alive	1343
69.1	Part I: Feature sets	1343
69.2	Part II: Controlling enabled features	1343
69.3	Part III: Implementation hints	1344
69.4	Part IV: Features	1344
70	Network Devices, the Kernel, and You!	1347
70.1	Introduction	1347
70.2	struct net_device lifetime rules	1347
70.3	MTU	1350
70.4	struct net_device synchronization rules	1350
70.5	struct napi_struct synchronization rules	1352
71	Netfilter Sysfs variables	1353
71.1	/proc/sys/net/netfilter/* Variables:	1353
72	NETIF Msg Level	1355
72.1	History	1355
73	Resilient Next-hop Groups	1357
73.1	Algorithm	1358
73.2	Offloading & Driver Feedback	1359
73.3	Netlink UAPI	1359
73.4	Usage	1361
73.5	Netdevsim	1361
74	Netfilter Conntrack Sysfs variables	1363
74.1	/proc/sys/net/netfilter/nf_conntrack_* Variables:	1363
75	Netfilter's flowtable infrastructure	1369
75.1	Overview	1369

75.2	Example configuration	1370
75.3	Layer 2 encapsulation	1371
75.4	Bridge and IP forwarding	1371
75.5	Counters	1372
75.6	Hardware offload	1372
75.7	Limitations	1372
75.8	More reading	1373
76	Open vSwitch datapath developer documentation	1375
76.1	Flow key compatibility	1375
76.2	Flow key format	1376
76.3	Wildcarded flow key format	1376
76.4	Unique flow identifiers	1377
76.5	Basic rule for evolving flow keys	1377
76.6	Handling malformed packets	1378
76.7	Other rules	1379
77	Operational States	1381
77.1	1. Introduction	1381
77.2	2. Querying from userspace	1381
77.3	3. Kernel driver API	1382
77.4	4. Setting from userspace	1383
78	Packet MMAP	1385
78.1	Abstract	1385
78.2	Why use PACKET_MMAP	1385
78.3	How to use mmap() to improve capture process	1386
78.4	How to use mmap() directly to improve capture process	1386
78.5	How to use mmap() directly to improve transmission process	1387
78.6	PACKET_MMAP settings	1388
78.7	PACKET_MMAP setting constraints	1389
78.8	PACKET_MMAP buffer size calculator	1391
78.9	What TPACKET versions are available and when to use them?	1395
78.10	AF_PACKET fanout mode	1396
78.11	AF_PACKET TPACKET_V3 example	1399
78.12	PACKET_QDISC_BYPASS	1404
78.13	PACKET_TIMESTAMP	1404
78.14	Miscellaneous bits	1405
78.15	THANKS	1405
79	Linux Phonet protocol family	1407
79.1	Introduction	1407
79.2	Packets format	1407
79.3	Link layer	1408
79.4	Network layer	1408
79.5	Low-level datagram protocol	1408
79.6	Resource subscription	1409
79.7	Phonet Pipe protocol	1409
79.8	Authors	1410
80	HOWTO for the linux packet generator	1411
80.1	Tuning NIC for max performance	1411

80.2	Kernel threads	1412
80.3	Viewing devices	1412
80.4	Configuring devices	1413
80.5	Sample scripts	1415
80.6	Interrupt affinity	1416
80.7	Enable IPsec	1416
80.8	Disable shared SKB	1416
80.9	Current commands and configuration options	1417
81	PLIP: The Parallel Line Internet Protocol Device	1421
81.1	PLIP Introduction	1421
81.2	PLIP driver details	1422
81.3	PLIP hardware interconnection	1423
82	PPP Generic Driver and Channel Interface	1425
82.1	PPP channel API	1426
82.2	Buffering and flow control	1427
82.3	SMP safety	1428
82.4	Interface to pppd	1429
83	The proc/net/tcp and proc/net/tcp6 variables	1433
84	How to use radiotap headers	1435
84.1	Pointer to the radiotap include file	1435
84.2	Structure of the header	1435
84.3	Requirements for arguments	1435
84.4	Example valid radiotap header	1436
84.5	Using the Radiotap Parser	1436
85	RDS	1439
85.1	Overview	1439
85.2	RDS Architecture	1439
85.3	Socket Interface	1440
85.4	RDMA for RDS	1442
85.5	Congestion Notifications	1442
85.6	RDS Protocol	1442
85.7	RDS Transport Layer	1443
85.8	RDS Kernel Structures	1444
85.9	Connection management	1444
85.10	The send path	1444
85.11	The recv path	1445
85.12	Multipath RDS (mprds)	1446
86	Linux wireless regulatory documentation	1449
86.1	Keeping regulatory domains in userspace	1449
86.2	How to get regulatory domains to the kernel	1449
86.3	How to get regulatory domains to the kernel (old CRDA solution)	1449
86.4	Who asks for regulatory domains?	1450
86.5	Example code - drivers hinting an alpha2:	1450
86.6	Example code - drivers providing a built in regulatory domain:	1451
86.7	Statically compiled regulatory database	1452
87	Network Function Representors	1453

87.1	Motivation	1453
87.2	Definitions	1453
87.3	What does a representor do?	1454
87.4	What functions should have a representor?	1454
87.5	How are representors created?	1455
87.6	How are representors identified?	1455
87.7	How do representors interact with TC rules?	1456
87.8	Configuring the representee's MAC	1457
88	RxRPC Network Protocol	1459
88.1	Overview	1459
88.2	RxRPC Protocol Summary	1460
88.3	AF_RXRPC Driver Model	1461
88.4	Control Messages	1464
89	SOCKET OPTIONS	1467
90	SECURITY	1469
91	EXAMPLE CLIENT USAGE	1471
91.1	Example Server Usage	1472
91.2	AF_RXRPC Kernel Interface	1475
91.3	Configurable Parameters	1480
92	Linux Kernel SCTP	1483
92.1	Caveats	1483
93	LSM/SeLinux secid	1485
94	Seg6 Sysfs variables	1487
94.1	/proc/sys/net/conf/<iface>/seg6_* variables:	1487
95	struct sk_buff	1489
95.1	Basic sk_buff geometry	1489
95.2	Shared skbs and skb clones	1490
95.3	dataref and headerless skbs	1490
95.4	Checksum information	1490
96	SMC Sysctl	1495
96.1	/proc/sys/net/smc/* Variables	1495
97	Interface statistics	1497
97.1	Overview	1497
97.2	uAPIs	1499
97.3	struct rtnl_link_stats64	1500
97.4	Notes for driver authors	1504
98	Stream Parser (strparser)	1505
98.1	Introduction	1505
98.2	Interface	1505
98.3	Functions	1505
98.4	Callbacks	1506
98.5	Statistics	1508

98.6 Message assembly limits	1508
98.7 Author	1508
99 Ethernet switch device driver model (switchdev)	1509
99.1 Include Files	1510
99.2 Configuration	1510
99.3 Switch Ports	1510
99.4 L2 Forwarding Offload	1512
99.5 L3 Routing Offload	1515
99.6 Device driver expected behavior	1516
100 Sysfs tagging	1519
101 TC Actions - Environmental Rules	1521
102 TC queue based filtering	1523
103 TCP Authentication Option Linux implementation (RFC5925)	1525
103.1 1. Introduction	1525
103.2 2. In-kernel MKTs database vs database in userspace	1529
103.3 3. uAPI	1530
103.4 4. setsockopt() vs accept() race	1531
103.5 5. Interaction with TCP-MD5	1531
103.6 6. SNE Linux implementation	1531
103.7 7. Links	1532
104 Thin-streams and TCP	1533
104.1 References	1533
105 Team	1535
106 Timestamping	1537
106.1 1. Control Interfaces	1537
106.2 2 Data Interfaces	1543
106.3 3. Hardware Timestamping configuration: SIOCSHWTSTAMP and SIOCGHWTSTAMP	1545
107 Linux Kernel TIPC	1551
107.1 Introduction	1551
107.2 Implementation	1553
108 Transparent proxy support	1625
108.1 1. Making non-local sockets work	1625
108.2 2. Redirecting traffic	1626
108.3 3. Iptables and nf_tables extensions	1626
108.4 4. Application support	1627
109 Universal TUN/TAP device driver	1629
109.1 1. Description	1629
109.2 2. Configuration	1630
109.3 3. Program interface	1630
109.4 Universal TUN/TAP device driver Frequently Asked Question	1633

110 The UDP-Lite protocol (RFC 3828)	1635
110.1 1. Applications	1635
110.2 2. Programming API	1635
110.3 3. Header Files	1636
110.4 4. Kernel Behaviour with Regards to the Various Socket Options	1637
110.5 5. UDP-Lite Runtime Statistics and their Meaning	1639
110.6 6. IPTables	1639
110.7 7. Maintainer Address	1640
111 Virtual Routing and Forwarding (VRF)	1641
111.1 The VRF Device	1641
111.2 Using iproute2 for VRFs	1644
112 Virtual eXtensible Local Area Networking documentation	1651
113 Linux X.25 Project	1653
114 X.25 Device Driver Interface	1655
114.1 Packet Layer to Device Driver	1655
114.2 Device Driver to Packet Layer	1656
114.3 Requirements for the device driver	1656
115 XFRM device - offloading the IPsec computations	1657
115.1 Overview	1657
115.2 Callbacks to implement	1658
115.3 Flow	1658
116 XFRM proc - /proc/net/xfrm_* files	1661
116.1 Transformation Statistics	1661
117 XFRM	1663
117.1 1) Message Structure	1663
117.2 2) TLVS reflect the different parameters:	1664
117.3 3) Default configurations for the parameters:	1664
117.4 4) Message types	1665
117.5 Exceptions to threshold settings	1666
118 XFRM Syscall	1667
118.1 /proc/sys/net/core/xfrm_* Variables:	1667
119 XDP RX Metadata	1669
119.1 General Design	1669
119.2 AF_XDP	1671
119.3 XDP_PASS	1671
119.4 bpf_redirect_map	1672
119.5 bpf_tail_call	1672
119.6 Supported Devices	1672
119.7 Example	1672
120 AF_XDP TX Metadata	1673
120.1 General Design	1673
120.2 Software TX Checksum	1674
120.3 Querying Device Capabilities	1674

120.4 Example	1674
Index	1675

Refer to netdev-FAQ for a guide on netdev development process specifics.

Contents:

AF_XDP

1.1 Overview

AF_XDP is an address family that is optimized for high performance packet processing.

This document assumes that the reader is familiar with BPF and XDP. If not, the Cilium project has an excellent reference guide at <http://cilium.readthedocs.io/en/latest/bpf/>.

Using the XDP_REDIRECT action from an XDP program, the program can redirect ingress frames to other XDP enabled netdevs, using the `bpf_redirect_map()` function. AF_XDP sockets enable the possibility for XDP programs to redirect frames to a memory buffer in a user-space application.

An AF_XDP socket (XSK) is created with the normal `socket()` syscall. Associated with each XSK are two rings: the RX ring and the TX ring. A socket can receive packets on the RX ring and it can send packets on the TX ring. These rings are registered and sized with the `setsockopt XDP_RX_RING` and `XDP_TX_RING`, respectively. It is mandatory to have at least one of these rings for each socket. An RX or TX descriptor ring points to a data buffer in a memory area called a UMEM. RX and TX can share the same UMEM so that a packet does not have to be copied between RX and TX. Moreover, if a packet needs to be kept for a while due to a possible retransmit, the descriptor that points to that packet can be changed to point to another and reused right away. This again avoids copying data.

The UMEM consists of a number of equally sized chunks. A descriptor in one of the rings references a frame by referencing its `addr`. The `addr` is simply an offset within the entire UMEM region. The user space allocates memory for this UMEM using whatever means it feels is most appropriate (`malloc`, `mmap`, huge pages, etc). This memory area is then registered with the kernel using the new `setsockopt XDP_UMEM_REG`. The UMEM also has two rings: the FILL ring and the COMPLETION ring. The FILL ring is used by the application to send down `addr` for the kernel to fill in with RX packet data. References to these frames will then appear in the RX ring once each packet has been received. The COMPLETION ring, on the other hand, contains frame `addr` that the kernel has transmitted completely and can now be used again by user space, for either TX or RX. Thus, the frame `addrs` appearing in the COMPLETION ring are `addrs` that were previously transmitted using the TX ring. In summary, the RX and FILL rings are used for the RX path and the TX and COMPLETION rings are used for the TX path.

The socket is then finally bound with a `bind()` call to a device and a specific queue id on that device, and it is not until `bind` is completed that traffic starts to flow.

The UMEM can be shared between processes, if desired. If a process wants to do this, it simply skips the registration of the UMEM and its corresponding two rings, sets the `XDP_SHARED_UMEM` flag in the `bind` call and submits the XSK of the process it would like to share UMEM with as well as its own newly created XSK socket. The new process will then

receive frame addr references in its own RX ring that point to this shared UMEM. Note that since the ring structures are single-consumer / single-producer (for performance reasons), the new process has to create its own socket with associated RX and TX rings, since it cannot share this with the other process. This is also the reason that there is only one set of FILL and COMPLETION rings per UMEM. It is the responsibility of a single process to handle the UMEM.

How is then packets distributed from an XDP program to the XSKs? There is a BPF map called XSKMAP (or BPF_MAP_TYPE_XSKMAP in full). The user-space application can place an XSK at an arbitrary place in this map. The XDP program can then redirect a packet to a specific index in this map and at this point XDP validates that the XSK in that map was indeed bound to that device and ring number. If not, the packet is dropped. If the map is empty at that index, the packet is also dropped. This also means that it is currently mandatory to have an XDP program loaded (and one XSK in the XSKMAP) to be able to get any traffic to user space through the XSK.

AF_XDP can operate in two different modes: XDP_SKB and XDP_DRV. If the driver does not have support for XDP, or XDP_SKB is explicitly chosen when loading the XDP program, XDP_SKB mode is employed that uses SKBs together with the generic XDP support and copies out the data to user space. A fallback mode that works for any network device. On the other hand, if the driver has support for XDP, it will be used by the AF_XDP code to provide better performance, but there is still a copy of the data into user space.

1.2 Concepts

In order to use an AF_XDP socket, a number of associated objects need to be setup. These objects and their options are explained in the following sections.

For an overview on how AF_XDP works, you can also take a look at the Linux Plumbers paper from 2018 on the subject: http://vger.kernel.org/lpc_net2018_talks/lpc18_paper_af_xdp_perf-v2.pdf. Do NOT consult the paper from 2017 on "AF_PACKET v4", the first attempt at AF_XDP. Nearly everything changed since then. Jonathan Corbet has also written an excellent article on LWN, "Accelerating networking with AF_XDP". It can be found at <https://lwn.net/Articles/750845/>.

1.2.1 UMEM

UMEM is a region of virtual contiguous memory, divided into equal-sized frames. An UMEM is associated to a netdev and a specific queue id of that netdev. It is created and configured (chunk size, headroom, start address and size) by using the XDP_UMEM_REG setsockopt system call. A UMEM is bound to a netdev and queue id, via the bind() system call.

An AF_XDP is socket linked to a single UMEM, but one UMEM can have multiple AF_XDP sockets. To share an UMEM created via one socket A, the next socket B can do this by setting the XDP_SHARED_UMEM flag in struct sockaddr_xdp member sxdp_flags, and passing the file descriptor of A to struct sockaddr_xdp member sxdp_shared_umem_fd.

The UMEM has two single-producer/single-consumer rings that are used to transfer ownership of UMEM frames between the kernel and the user-space application.

1.2.2 Rings

There are four different kind of rings: FILL, COMPLETION, RX and TX. All rings are single-producer/single-consumer, so the user-space application need explicit synchronization of multiple processes/threads are reading/writing to them.

The UMEM uses two rings: FILL and COMPLETION. Each socket associated with the UMEM must have an RX queue, TX queue or both. Say, that there is a setup with four sockets (all doing TX and RX). Then there will be one FILL ring, one COMPLETION ring, four TX rings and four RX rings.

The rings are head(producer)/tail(consumer) based rings. A producer writes the data ring at the index pointed out by struct xdp_ring producer member, and increasing the producer index. A consumer reads the data ring at the index pointed out by struct xdp_ring consumer member, and increasing the consumer index.

The rings are configured and created via the _RING setsockopt system calls and mmapped to user-space using the appropriate offset to mmap() (XDP_PGOFF_RX_RING, XDP_PGOFF_TX_RING, XDP_UMEM_PGOFF_FILL_RING and XDP_UMEM_PGOFF_COMPLETION_RING).

The size of the rings need to be of size power of two.

UMEM Fill Ring

The FILL ring is used to transfer ownership of UMEM frames from user-space to kernel-space. The UMEM addrs are passed in the ring. As an example, if the UMEM is 64k and each chunk is 4k, then the UMEM has 16 chunks and can pass addrs between 0 and 64k.

Frames passed to the kernel are used for the ingress path (RX rings).

The user application produces UMEM addrs to this ring. Note that, if running the application with aligned chunk mode, the kernel will mask the incoming addr. E.g. for a chunk size of 2k, the log2(2048) LSB of the addr will be masked off, meaning that 2048, 2050 and 3000 refers to the same chunk. If the user application is run in the unaligned chunks mode, then the incoming addr will be left untouched.

UMEM Completion Ring

The COMPLETION Ring is used transfer ownership of UMEM frames from kernel-space to user-space. Just like the FILL ring, UMEM indices are used.

Frames passed from the kernel to user-space are frames that has been sent (TX ring) and can be used by user-space again.

The user application consumes UMEM addrs from this ring.

RX Ring

The RX ring is the receiving side of a socket. Each entry in the ring is a struct xdp_desc descriptor. The descriptor contains UMEM offset (addr) and the length of the data (len).

If no frames have been passed to kernel via the FILL ring, no descriptors will (or can) appear on the RX ring.

The user application consumes struct xdp_desc descriptors from this ring.

TX Ring

The TX ring is used to send frames. The struct xdp_desc descriptor is filled (index, length and offset) and passed into the ring.

To start the transfer a sendmsg() system call is required. This might be relaxed in the future.

The user application produces struct xdp_desc descriptors to this ring.

1.3 Libbpf

Libbpf is a helper library for eBPF and XDP that makes using these technologies a lot simpler. It also contains specific helper functions in tools/lib/bpf/xsk.h for facilitating the use of AF_XDP. It contains two types of functions: those that can be used to make the setup of AF_XDP socket easier and ones that can be used in the data plane to access the rings safely and quickly. To see an example on how to use this API, please take a look at the sample application in samples/bpf/xdpsock_usr.c which uses libbpf for both setup and data plane operations.

We recommend that you use this library unless you have become a power user. It will make your program a lot simpler.

1.4 XSKMAP / BPF_MAP_TYPE_XSKMAP

On XDP side there is a BPF map type BPF_MAP_TYPE_XSKMAP (XSKMAP) that is used in conjunction with bpf_redirect_map() to pass the ingress frame to a socket.

The user application inserts the socket into the map, via the bpf() system call.

Note that if an XDP program tries to redirect to a socket that does not match the queue configuration and netdev, the frame will be dropped. E.g. an AF_XDP socket is bound to netdev eth0 and queue 17. Only the XDP program executing for eth0 and queue 17 will successfully pass data to the socket. Please refer to the sample application (samples/bpf/) in for an example.

1.5 Configuration Flags and Socket Options

These are the various configuration flags that can be used to control and monitor the behavior of AF_XDP sockets.

1.5.1 XDP_COPY and XDP_ZEROCOPY bind flags

When you bind to a socket, the kernel will first try to use zero-copy copy. If zero-copy is not supported, it will fall back on using copy mode, i.e. copying all packets out to user space. But if you would like to force a certain mode, you can use the following flags. If you pass the XDP_COPY flag to the bind call, the kernel will force the socket into copy mode. If it cannot use copy mode, the bind call will fail with an error. Conversely, the XDP_ZEROCOPY flag will force the socket into zero-copy mode or fail.

1.5.2 XDP_SHARED_UMEM bind flag

This flag enables you to bind multiple sockets to the same UMEM. It works on the same queue id, between queue ids and between netdevs/devices. In this mode, each socket has their own RX and TX rings as usual, but you are going to have one or more FILL and COMPLETION ring pairs. You have to create one of these pairs per unique netdev and queue id tuple that you bind to.

Starting with the case were we would like to share a UMEM between sockets bound to the same netdev and queue id. The UMEM (tied to the fist socket created) will only have a single FILL ring and a single COMPLETION ring as there is only on unique netdev,queue_id tuple that we have bound to. To use this mode, create the first socket and bind it in the normal way. Create a second socket and create an RX and a TX ring, or at least one of them, but no FILL or COMPLETION rings as the ones from the first socket will be used. In the bind call, set he XDP_SHARED_UMEM option and provide the initial socket's fd in the sxdp_shared_umem_fd field. You can attach an arbitrary number of extra sockets this way.

What socket will then a packet arrive on? This is decided by the XDP program. Put all the sockets in the XSK_MAP and just indicate which index in the array you would like to send each packet to. A simple round-robin example of distributing packets is shown below:

```
#include <linux/bpf.h>
#include "bpf_helpers.h"

#define MAX SOCKS 16

struct {
    __uint(type, BPF_MAP_TYPE_XSKMAP);
    __uint(max_entries, MAX SOCKS);
    __uint(key_size, sizeof(int));
    __uint(value_size, sizeof(int));
} xsks_map SEC(".maps");

static unsigned int rr;

SEC("xdp_sock") int xdp_sock_prog(struct xdp_md *ctx)
```

```
{
    rr = (rr + 1) & (MAX_SOCKS - 1);

    return bpf_redirect_map(&xskks_map, rr, XDP_DROP);
}
```

Note, that since there is only a single set of FILL and COMPLETION rings, and they are single producer, single consumer rings, you need to make sure that multiple processes or threads do not use these rings concurrently. There are no synchronization primitives in the libbpf code that protects multiple users at this point in time.

Libbpf uses this mode if you create more than one socket tied to the same UMEM. However, note that you need to supply the XSK_LIBBPF_FLAGS_INHIBIT_PROG_LOAD libbpf_flag with the xsk_socket_create calls and load your own XDP program as there is no built in one in libbpf that will route the traffic for you.

The second case is when you share a UMEM between sockets that are bound to different queue ids and/or netdevs. In this case you have to create one FILL ring and one COMPLETION ring for each unique netdev,queue_id pair. Let us say you want to create two sockets bound to two different queue ids on the same netdev. Create the first socket and bind it in the normal way. Create a second socket and create an RX and a TX ring, or at least one of them, and then one FILL and COMPLETION ring for this socket. Then in the bind call, set the XDP_SHARED_UMEM option and provide the initial socket's fd in the sxdp_shared_umem_fd field as you registered the UMEM on that socket. These two sockets will now share one and the same UMEM.

There is no need to supply an XDP program like the one in the previous case where sockets were bound to the same queue id and device. Instead, use the NIC's packet steering capabilities to steer the packets to the right queue. In the previous example, there is only one queue shared among sockets, so the NIC cannot do this steering. It can only steer between queues.

In libbpf, you need to use the xsk_socket_create_shared() API as it takes a reference to a FILL ring and a COMPLETION ring that will be created for you and bound to the shared UMEM. You can use this function for all the sockets you create, or you can use it for the second and following ones and use xsk_socket_create() for the first one. Both methods yield the same result.

Note that a UMEM can be shared between sockets on the same queue id and device, as well as between queues on the same device and between devices at the same time.

1.5.3 XDP_USE_NEED_WAKEUP bind flag

This option adds support for a new flag called need_wakeup that is present in the FILL ring and the TX ring, the rings for which user space is a producer. When this option is set in the bind call, the need_wakeup flag will be set if the kernel needs to be explicitly woken up by a syscall to continue processing packets. If the flag is zero, no syscall is needed.

If the flag is set on the FILL ring, the application needs to call poll() to be able to continue to receive packets on the RX ring. This can happen, for example, when the kernel has detected that there are no more buffers on the FILL ring and no buffers left on the RX HW ring of the NIC. In this case, interrupts are turned off as the NIC cannot receive any packets (as there are no buffers to put them in), and the need_wakeup flag is set so that user space can put buffers on the FILL ring and then call poll() so that the kernel driver can put these buffers on the HW ring and start to receive packets.

If the flag is set for the TX ring, it means that the application needs to explicitly notify the kernel to send any packets put on the TX ring. This can be accomplished either by a poll() call, as in the RX path, or by calling sendto().

An example of how to use this flag can be found in samples/bpf/xdpsock_user.c. An example with the use of libbpf helpers would look like this for the TX path:

```
if (xsk_ring_prod_needs_wakeup(&my_tx_ring))
    sendto(xsk_socket_fd(xsk_handle), NULL, 0, MSG_DONTWAIT, NULL, 0);
```

I.e., only use the syscall if the flag is set.

We recommend that you always enable this mode as it usually leads to better performance especially if you run the application and the driver on the same core, but also if you use different cores for the application and the kernel driver, as it reduces the number of syscalls needed for the TX path.

1.5.4 XDP_{RX|TX|UMEM_FILL|UMEM_COMPLETION}_RING setsockopt

These setsockopt sets the number of descriptors that the RX, TX, FILL, and COMPLETION rings respectively should have. It is mandatory to set the size of at least one of the RX and TX rings. If you set both, you will be able to both receive and send traffic from your application, but if you only want to do one of them, you can save resources by only setting up one of them. Both the FILL ring and the COMPLETION ring are mandatory as you need to have a UMEM tied to your socket. But if the XDP_SHARED_UMEM flag is used, any socket after the first one does not have a UMEM and should in that case not have any FILL or COMPLETION rings created as the ones from the shared UMEM will be used. Note, that the rings are single-producer single-consumer, so do not try to access them from multiple processes at the same time. See the XDP_SHARED_UMEM section.

In libbpf, you can create Rx-only and Tx-only sockets by supplying NULL to the rx and tx arguments, respectively, to the xsk_socket_create function.

If you create a Tx-only socket, we recommend that you do not put any packets on the fill ring. If you do this, drivers might think you are going to receive something when you in fact will not, and this can negatively impact performance.

1.5.5 XDP_UMEM_REG setsockopt

This setsockopt registers a UMEM to a socket. This is the area that contain all the buffers that packet can reside in. The call takes a pointer to the beginning of this area and the size of it. Moreover, it also has parameter called chunk_size that is the size that the UMEM is divided into. It can only be 2K or 4K at the moment. If you have an UMEM area that is 128K and a chunk size of 2K, this means that you will be able to hold a maximum of $128K / 2K = 64$ packets in your UMEM area and that your largest packet size can be 2K.

There is also an option to set the headroom of each single buffer in the UMEM. If you set this to N bytes, it means that the packet will start N bytes into the buffer leaving the first N bytes for the application to use. The final option is the flags field, but it will be dealt with in separate sections for each UMEM flag.

1.5.6 SO_BINDTODEVICE setsockopt

This is a generic SOL_SOCKET option that can be used to tie AF_XDP socket to a particular network interface. It is useful when a socket is created by a privileged process and passed to a non-privileged one. Once the option is set, kernel will refuse attempts to bind that socket to a different interface. Updating the value requires CAP_NET_RAW.

1.5.7 XDP_STATISTICS getsockopt

Gets drop statistics of a socket that can be useful for debug purposes. The supported statistics are shown below:

```
struct xdp_statistics {
    __u64 rx_dropped; /* Dropped for reasons other than invalid desc */
    __u64 rx_invalid_descs; /* Dropped due to invalid descriptor */
    __u64 tx_invalid_descs; /* Dropped due to invalid descriptor */
};
```

1.5.8 XDP_OPTIONS getsockopt

Gets options from an XDP socket. The only one supported so far is XDP_OPTIONS_ZEROCOPY which tells you if zero-copy is on or not.

1.6 Multi-Buffer Support

With multi-buffer support, programs using AF_XDP sockets can receive and transmit packets consisting of multiple buffers both in copy and zero-copy mode. For example, a packet can consist of two frames/buffers, one with the header and the other one with the data, or a 9K Ethernet jumbo frame can be constructed by chaining together three 4K frames.

Some definitions:

- A packet consists of one or more frames
- A descriptor in one of the AF_XDP rings always refers to a single frame. In the case the packet consists of a single frame, the descriptor refers to the whole packet.

To enable multi-buffer support for an AF_XDP socket, use the new bind flag XDP_USE_SG. If this is not provided, all multi-buffer packets will be dropped just as before. Note that the XDP program loaded also needs to be in multi-buffer mode. This can be accomplished by using "xdp.frags" as the section name of the XDP program used.

To represent a packet consisting of multiple frames, a new flag called XDP_PKT_CONTD is introduced in the options field of the Rx and Tx descriptors. If it is true (1) the packet continues with the next descriptor and if it is false (0) it means this is the last descriptor of the packet. Why the reverse logic of end-of-packet (eop) flag found in many NICs? Just to preserve compatibility with non-multi-buffer applications that have this bit set to false for all packets on Rx, and the apps set the options field to zero for Tx, as anything else will be treated as an invalid descriptor.

These are the semantics for producing packets onto AF_XDP Tx ring consisting of multiple frames:

- When an invalid descriptor is found, all the other descriptors/frames of this packet are marked as invalid and not completed. The next descriptor is treated as the start of a new packet, even if this was not the intent (because we cannot guess the intent). As before, if your program is producing invalid descriptors you have a bug that must be fixed.
- Zero length descriptors are treated as invalid descriptors.
- For copy mode, the maximum supported number of frames in a packet is equal to CONFIG_MAX_SKB_FRAGS + 1. If it is exceeded, all descriptors accumulated so far are dropped and treated as invalid. To produce an application that will work on any system regardless of this config setting, limit the number of frags to 18, as the minimum value of the config is 17.
- For zero-copy mode, the limit is up to what the NIC HW supports. Usually at least five on the NICs we have checked. We consciously chose to not enforce a rigid limit (such as CONFIG_MAX_SKB_FRAGS + 1) for zero-copy mode, as it would have resulted in copy actions under the hood to fit into what limit the NIC supports. Kind of defeats the purpose of zero-copy mode. How to probe for this limit is explained in the "probe for multi-buffer support" section.

On the Rx path in copy-mode, the xsk core copies the XDP data into multiple descriptors, if needed, and sets the XDP_PKT_CONTD flag as detailed before. Zero-copy mode works the same, though the data is not copied. When the application gets a descriptor with the XDP_PKT_CONTD flag set to one, it means that the packet consists of multiple buffers and it continues with the next buffer in the following descriptor. When a descriptor with XDP_PKT_CONTD == 0 is received, it means that this is the last buffer of the packet. AF_XDP guarantees that only a complete packet (all frames in the packet) is sent to the application. If there is not enough space in the AF_XDP Rx ring, all frames of the packet will be dropped.

If application reads a batch of descriptors, using for example the libxdp interfaces, it is not guaranteed that the batch will end with a full packet. It might end in the middle of a packet and the rest of the buffers of that packet will arrive at the beginning of the next batch, since the libxdp interface does not read the whole ring (unless you have an enormous batch size or a very small ring size).

An example program each for Rx and Tx multi-buffer support can be found later in this document.

1.6.1 Usage

In order to use AF_XDP sockets two parts are needed. The user-space application and the XDP program. For a complete setup and usage example, please refer to the sample application. The user-space side is xdpsock_user.c and the XDP side is part of libbpf.

The XDP code sample included in tools/lib/bpf/xsk.c is the following:

```
SEC("xdp_sock") int xdp_sock_prog(struct xdp_md *ctx)
{
    int index = ctx->rx_queue_index;

    // A set entry here means that the corresponding queue_id
    // has an active AF_XDP socket bound to it.
    if (bpf_map_lookup_elem(&xsk_map, &index))
        return bpf_redirect_map(&xsk_map, index, 0);
```

```

    return XDP_PASS;
}

```

A simple but not so performance ring dequeue and enqueue could look like this:

```

// struct xdp_rxtx_ring {
//     __u32 *producer;
//     __u32 *consumer;
//     struct xdp_desc *desc;
// };

// struct xdp_umem_ring {
//     __u32 *producer;
//     __u32 *consumer;
//     __u64 *desc;
// };

// typedef struct xdp_rxtx_ring RING;
// typedef struct xdp_umem_ring RING;

// typedef struct xdp_desc RING_TYPE;
// typedef __u64 RING_TYPE;

int dequeue_one(RING *ring, RING_TYPE *item)
{
    __u32 entries = *ring->producer - *ring->consumer;

    if (entries == 0)
        return -1;

    // read-barrier!

    *item = ring->desc[*ring->consumer & (RING_SIZE - 1)];
    (*ring->consumer)++;
    return 0;
}

int enqueue_one(RING *ring, const RING_TYPE *item)
{
    u32 free_entries = RING_SIZE - (*ring->producer - *ring->consumer);

    if (free_entries == 0)
        return -1;

    ring->desc[*ring->producer & (RING_SIZE - 1)] = *item;

    // write-barrier!

    (*ring->producer)++;
    return 0;
}

```

```
}
```

But please use the libbpf functions as they are optimized and ready to use. Will make your life easier.

1.6.2 Usage Multi-Buffer Rx

Here is a simple Rx path pseudo-code example (using libxdp interfaces for simplicity). Error paths have been excluded to keep it short:

```
void rx_packets(struct xsk_socket_info *xsk)
{
    static bool new_packet = true;
    u32 idx_rx = 0, idx_fq = 0;
    static char *pkt;

    int rcvd = xsk_ring_cons__peek(&xsk->rx, opt_batch_size, &idx_rx);

    xsk_ring_prod__reserve(&xsk->umem->fq, rcvd, &idx_fq);

    for (int i = 0; i < rcvd; i++) {
        struct xdp_desc *desc = xsk_ring_cons__rx_desc(&xsk->rx, idx_rx++);
        char *frag = xsk_umem__get_data(xsk->umem->buffer, desc->addr);
        bool eop = !(desc->options & XDP_PKT_CONTD);

        if (new_packet)
            pkt = frag;
        else
            add_frag_to_pkt(pkt, frag);

        if (eop)
            process_pkt(pkt);

        new_packet = eop;

        *xsk_ring_prod__fill_addr(&xsk->umem->fq, idx_fq++) = desc->addr;
    }

    xsk_ring_prod__submit(&xsk->umem->fq, rcvd);
    xsk_ring_cons__release(&xsk->rx, rcvd);
}
```

1.6.3 Usage Multi-Buffer Tx

Here is an example Tx path pseudo-code (using libxdp interfaces for simplicity) ignoring that the umem is finite in size, and that we eventually will run out of packets to send. Also assumes pkts.addr points to a valid location in the umem.

```
void tx_packets(struct xsk_socket_info *xsk, struct pkt *pkts,
                 int batch_size)
{
    u32 idx, i, pkt_nb = 0;

    xsk_ring_prod_reserve(&xsk->tx, batch_size, &idx);

    for (i = 0; i < batch_size;) {
        u64 addr = pkts[pkt_nb].addr;
        u32 len = pkts[pkt_nb].size;

        do {
            struct xdp_desc *tx_desc;

            tx_desc = xsk_ring_prod_tx_desc(&xsk->tx, idx + i++);
            tx_desc->addr = addr;

            if (len > xsk_frame_size) {
                tx_desc->len = xsk_frame_size;
                tx_desc->options = XDP_PKT_CONTD;
            } else {
                tx_desc->len = len;
                tx_desc->options = 0;
                pkt_nb++;
            }
            len -= tx_desc->len;
            addr += xsk_frame_size;

            if (i == batch_size) {
                /* Remember len, addr, pkt_nb for next iteration.
                 * Skipped for simplicity.
                 */
                break;
            }
        } while (len);
    }

    xsk_ring_prod_submit(&xsk->tx, i);
}
```

1.6.4 Probing for Multi-Buffer Support

To discover if a driver supports multi-buffer AF_XDP in SKB or DRV mode, use the XDP_FEATURES feature of netlink in linux/netdev.h to query for NETDEV_XDP_ACT_RX_SG support. This is the same flag as for querying for XDP multi-buffer support. If XDP supports multi-buffer in a driver, then AF_XDP will also support that in SKB and DRV mode.

To discover if a driver supports multi-buffer AF_XDP in zero-copy mode, use XDP_FEATURES and first check the NETDEV_XDP_ACT_XSK_ZEROCOPY flag. If it is set, it means that at least zero-copy is supported and you should go and check the netlink attribute NETDEV_A_DEV_XDP_ZC_MAX_SEGS in linux/netdev.h. An unsigned integer value will be returned stating the max number of frags that are supported by this device in zero-copy mode. These are the possible return values:

1: Multi-buffer for zero-copy is not supported by this device, as max one fragment supported means that multi-buffer is not possible.

>=2: Multi-buffer is supported in zero-copy mode for this device. The returned number signifies the max number of frags supported.

For an example on how these are used through libbpf, please take a look at tools/testing/selftests/bpf/xskxceiver.c.

1.6.5 Multi-Buffer Support for Zero-Copy Drivers

Zero-copy drivers usually use the batched APIs for Rx and Tx processing. Note that the Tx batch API guarantees that it will provide a batch of Tx descriptors that ends with full packet at the end. This to facilitate extending a zero-copy driver with multi-buffer support.

1.7 Sample application

There is a xdpsock benchmarking/test application included that demonstrates how to use AF_XDP sockets with private UMEMs. Say that you would like your UDP traffic from port 4242 to end up in queue 16, that we will enable AF_XDP on. Here, we use ethtool for this:

```
ethtool -N p3p2 rx-flow-hash udp4 fn
ethtool -N p3p2 flow-type udp4 src-port 4242 dst-port 4242 \
    action 16
```

Running the rxdrop benchmark in XDP_DRV mode can then be done using:

```
samples/bpf/xdpsock -i p3p2 -q 16 -r -N
```

For XDP_SKB mode, use the switch "-S" instead of "-N" and all options can be displayed with "-h", as usual.

This sample application uses libbpf to make the setup and usage of AF_XDP simpler. If you want to know how the raw uapi of AF_XDP is really used to make something more advanced, take a look at the libbpf code in tools/lib/bpf/xsk.[ch].

1.8 FAQ

Q: I am not seeing any traffic on the socket. What am I doing wrong?

A: When a netdev of a physical NIC is initialized, Linux usually

allocates one RX and TX queue pair per core. So on a 8 core system, queue ids 0 to 7 will be allocated, one per core. In the AF_XDP bind call or the xsk_socket_create libbpf function call, you specify a specific queue id to bind to and it is only the traffic towards that queue you are going to get on your socket. So in the example above, if you bind to queue 0, you are NOT going to get any traffic that is distributed to queues 1 through 7. If you are lucky, you will see the traffic, but usually it will end up on one of the queues you have not bound to.

There are a number of ways to solve the problem of getting the traffic you want to the queue id you bound to. If you want to see all the traffic, you can force the netdev to only have 1 queue, queue id 0, and then bind to queue 0. You can use ethtool to do this:

```
sudo ethtool -L <interface> combined 1
```

If you want to only see part of the traffic, you can program the NIC through ethtool to filter out your traffic to a single queue id that you can bind your XDP socket to. Here is one example in which UDP traffic to and from port 4242 are sent to queue 2:

```
sudo ethtool -N <interface> rx-flow-hash udp4 fn
sudo ethtool -N <interface> flow-type udp4 src-port 4242 dst-port \
4242 action 2
```

A number of other ways are possible all up to the capabilities of the NIC you have.

**Q: Can I use the XSKMAP to implement a switch between different umems
in copy mode?**

A: The short answer is no, that is not supported at the moment. The

XSKMAP can only be used to switch traffic coming in on queue id X to sockets bound to the same queue id X. The XSKMAP can contain sockets bound to different queue ids, for example X and Y, but only traffic coming in from queue id Y can be directed to sockets bound to the same queue id Y. In zero-copy mode, you should use the switch, or other distribution mechanism, in your NIC to direct traffic to the correct queue id and socket.

Q: My packets are sometimes corrupted. What is wrong?

A: Care has to be taken not to feed the same buffer in the UMEM into

more than one ring at the same time. If you for example feed the same buffer into the FILL ring and the TX ring at the same time, the NIC might receive data into the buffer at the same time it is sending it. This will cause some packets to become corrupted. Same thing goes for feeding the same buffer into the FILL rings belonging to different queue ids or netdevs bound with the XDP_SHARED_UMEM flag.

1.9 Credits

- Björn Töpel (AF_XDP core)
- Magnus Karlsson (AF_XDP core)
- Alexander Duyck
- Alexei Starovoitov
- Daniel Borkmann
- Jesper Dangaard Brouer
- John Fastabend
- Jonathan Corbet (LWN coverage)
- Michael S. Tsirkin
- Qi Z Zhang
- Willem de Bruijn

BARE UDP TUNNELLING MODULE DOCUMENTATION

There are various L3 encapsulation standards using UDP being discussed to leverage the UDP based load balancing capability of different networks. MPLSoUDP (<https://tools.ietf.org/html/rfc7510>) is one among them.

The Bareudp tunnel module provides a generic L3 encapsulation support for tunnelling different L3 protocols like MPLS, IP, NSH etc. inside a UDP tunnel.

2.1 Special Handling

The bareudp device supports special handling for MPLS & IP as they can have multiple ethertypes. MPLS protocol can have ethertypes ETH_P_MPLS_UC (unicast) & ETH_P_MPLS_MC (multicast). IP protocol can have ethertypes ETH_P_IP (v4) & ETH_P_IPV6 (v6). This special handling can be enabled only for ethertypes ETH_P_IP & ETH_P_MPLS_UC with a flag called multiproto mode.

2.2 Usage

1) Device creation & deletion

- `ip link add dev bareudp0 type bareudp dstport 6635 ethertype mpls_uc`

This creates a bareudp tunnel device which tunnels L3 traffic with ethertype 0x8847 (MPLS traffic). The destination port of the UDP header will be set to 6635. The device will listen on UDP port 6635 to receive traffic.

- `ip link delete bareudp0`

2) Device creation with multiproto mode enabled

The multiproto mode allows bareudp tunnels to handle several protocols of the same family. It is currently only available for IP and MPLS. This mode has to be enabled explicitly with the "multiproto" flag.

- `ip link add dev bareudp0 type bareudp dstport 6635 ethertype ipv4 multiproto`

For an IPv4 tunnel the multiproto mode allows the tunnel to also handle IPv6.

- `ip link add dev bareudp0 type bareudp dstport 6635 ethertype mpls_uc multiproto`

For MPLS, the multiproto mode allows the tunnel to handle both unicast and multicast MPLS packets.

3) Device Usage

The bareudp device could be used along with OVS or flower filter in TC. The OVS or TC flower layer must set the tunnel information in SKB dst field before sending packet buffer to the bareudp device for transmission. On reception the bareudp device extracts and stores the tunnel information in SKB dst field before passing the packet buffer to the network stack.

BATMAN-ADV

Batman advanced is a new approach to wireless networking which does no longer operate on the IP basis. Unlike the batman daemon, which exchanges information using UDP packets and sets routing tables, batman-advanced operates on ISO/OSI Layer 2 only and uses and routes (or better: bridges) Ethernet Frames. It emulates a virtual network switch of all nodes participating. Therefore all nodes appear to be link local, thus all higher operating protocols won't be affected by any changes within the network. You can run almost any protocol above batman advanced, prominent examples are: IPv4, IPv6, DHCP, IPX.

Batman advanced was implemented as a Linux kernel driver to reduce the overhead to a minimum. It does not depend on any (other) network driver, and can be used on wifi as well as ethernet lan, vpn, etc ... (anything with ethernet-style layer 2).

3.1 Configuration

Load the batman-adv module into your kernel:

```
$ insmod batman-adv.ko
```

The module is now waiting for activation. You must add some interfaces on which batman-adv can operate. The batman-adv soft-interface can be created using the iproute2 tool ip:

```
$ ip link add name bat0 type batadv
```

To activate a given interface simply attach it to the **bat0** interface:

```
$ ip link set dev eth0 master bat0
```

Repeat this step for all interfaces you wish to add. Now batman-adv starts using/broadcasting on this/these interface(s).

To deactivate an interface you have to detach it from the "bat0" interface:

```
$ ip link set dev eth0 nomaster
```

The same can also be done using the batctl interface subcommand:

```
batctl -m bat0 interface create  
batctl -m bat0 interface add -M eth0
```

To detach eth0 and destroy bat0:

```
batctl -m bat0 interface del -M eth0  
batctl -m bat0 interface destroy
```

There are additional settings for each batadv mesh interface, vlan and hardif which can be modified using batctl. Detailed information about this can be found in its manual.

For instance, you can check the current originator interval (value in milliseconds which determines how often batman-adv sends its broadcast packets):

```
$ batctl -M bat0 orig_interval  
1000
```

and also change its value:

```
$ batctl -M bat0 orig_interval 3000
```

In very mobile scenarios, you might want to adjust the originator interval to a lower value. This will make the mesh more responsive to topology changes, but will also increase the overhead.

Information about the current state can be accessed via the batadv generic netlink family. batctl provides a human readable version via its debug tables subcommands.

3.2 Usage

To make use of your newly created mesh, batman advanced provides a new interface "bat0" which you should use from this point on. All interfaces added to batman advanced are not relevant any longer because batman handles them for you. Basically, one "hands over" the data by using the batman interface and batman will make sure it reaches its destination.

The "bat0" interface can be used like any other regular interface. It needs an IP address which can be either statically configured or dynamically (by using DHCP or similar services):

```
NodeA: ip link set up dev bat0  
NodeA: ip addr add 192.168.0.1/24 dev bat0  
  
NodeB: ip link set up dev bat0  
NodeB: ip addr add 192.168.0.2/24 dev bat0  
NodeB: ping 192.168.0.1
```

Note: In order to avoid problems remove all IP addresses previously assigned to interfaces now used by batman advanced, e.g.:

```
$ ip addr flush dev eth0
```

3.3 Logging/Debugging

All error messages, warnings and information messages are sent to the kernel log. Depending on your operating system distribution this can be read in one of a number of ways. Try using the commands: `dmesg`, `logread`, or looking in the files `/var/log/kern.log` or `/var/log/syslog`. All batman-adv messages are prefixed with "batman-adv:". So to see just these messages try:

```
$ dmesg | grep batman-adv
```

When investigating problems with your mesh network, it is sometimes necessary to see more detailed debug messages. This must be enabled when compiling the batman-adv module. When building batman-adv as part of the kernel, use "make menuconfig" and enable the option B.A.T.M.A.N. debugging (`CONFIG_BATMAN_ADV_DEBUG=y`).

Those additional debug messages can be accessed using the perf infrastructure:

```
$ trace-cmd stream -e batadv:batadv_dbg
```

The additional debug output is by default disabled. It can be enabled during run time:

```
$ batctl -m bat0 loglevel routes tt
```

will enable debug messages for when routes and translation table entries change.

Counters for different types of packets entering and leaving the batman-adv module are available through ethtool:

```
$ ethtool --statistics bat0
```

3.4 batctl

As batman advanced operates on layer 2, all hosts participating in the virtual switch are completely transparent for all protocols above layer 2. Therefore the common diagnosis tools do not work as expected. To overcome these problems, batctl was created. At the moment the batctl contains ping, traceroute, tcpdump and interfaces to the kernel module settings.

For more information, please see the manpage (`man batctl`).

batctl is available on <https://www.open-mesh.org/>

3.5 Contact

Please send us comments, experiences, questions, anything :)

IRC:

#batadv on ircs://irc.hackint.org/

Mailing-list:

`b.a.t.m.a.n@lists.open-mesh.org` (optional subscription at <https://lists.open-mesh.org/mailman3/postorius/lists/b.a.t.m.a.n.lists.open-mesh.org/>)

You can also contact the Authors:

- Marek Lindner <mareklindner@neomailbox.ch>
- Simon Wunderlich <sw@simonwunderlich.de>

SOCKETCAN - CONTROLLER AREA NETWORK

4.1 Overview / What is SocketCAN

The socketcan package is an implementation of CAN protocols (Controller Area Network) for Linux. CAN is a networking technology which has widespread use in automation, embedded devices, and automotive fields. While there have been other CAN implementations for Linux based on character devices, SocketCAN uses the Berkeley socket API, the Linux network stack and implements the CAN device drivers as network interfaces. The CAN socket API has been designed as similar as possible to the TCP/IP protocols to allow programmers, familiar with network programming, to easily learn how to use CAN sockets.

4.2 Motivation / Why Using the Socket API

There have been CAN implementations for Linux before SocketCAN so the question arises, why we have started another project. Most existing implementations come as a device driver for some CAN hardware, they are based on character devices and provide comparatively little functionality. Usually, there is only a hardware-specific device driver which provides a character device interface to send and receive raw CAN frames, directly to/from the controller hardware. Queueing of frames and higher-level transport protocols like ISO-TP have to be implemented in user space applications. Also, most character-device implementations support only one single process to open the device at a time, similar to a serial interface. Exchanging the CAN controller requires employment of another device driver and often the need for adaption of large parts of the application to the new driver's API.

SocketCAN was designed to overcome all of these limitations. A new protocol family has been implemented which provides a socket interface to user space applications and which builds upon the Linux network layer, enabling use all of the provided queueing functionality. A device driver for CAN controller hardware registers itself with the Linux network layer as a network device, so that CAN frames from the controller can be passed up to the network layer and on to the CAN protocol family module and also vice-versa. Also, the protocol family module provides an API for transport protocol modules to register, so that any number of transport protocols can be loaded or unloaded dynamically. In fact, the can core module alone does not provide any protocol and cannot be used without loading at least one additional protocol module. Multiple sockets can be opened at the same time, on different or the same protocol module and they can listen/send frames on different or the same CAN IDs. Several sockets listening on the same interface for frames with the same CAN ID are all passed the same received matching CAN frames. An application wishing to communicate using a specific transport protocol, e.g. ISO-TP, just selects that protocol when opening the socket, and then can read and write application data byte streams, without having to deal with CAN-IDs, frames, etc.

Similar functionality visible from user-space could be provided by a character device, too, but this would lead to a technically inelegant solution for a couple of reasons:

- **Intricate usage:** Instead of passing a protocol argument to socket(2) and using bind(2) to select a CAN interface and CAN ID, an application would have to do all these operations using ioctl(2)s.
- **Code duplication:** A character device cannot make use of the Linux network queueing code, so all that code would have to be duplicated for CAN networking.
- **Abstraction:** In most existing character-device implementations, the hardware-specific device driver for a CAN controller directly provides the character device for the application to work with. This is at least very unusual in Unix systems for both, char and block devices. For example you don't have a character device for a certain UART of a serial interface, a certain sound chip in your computer, a SCSI or IDE controller providing access to your hard disk or tape streamer device. Instead, you have abstraction layers which provide a unified character or block device interface to the application on the one hand, and a interface for hardware-specific device drivers on the other hand. These abstractions are provided by subsystems like the tty layer, the audio subsystem or the SCSI and IDE subsystems for the devices mentioned above.

The easiest way to implement a CAN device driver is as a character device without such a (complete) abstraction layer, as is done by most existing drivers. The right way, however, would be to add such a layer with all the functionality like registering for certain CAN IDs, supporting several open file descriptors and (de)multiplexing CAN frames between them, (sophisticated) queueing of CAN frames, and providing an API for device drivers to register with. However, then it would be no more difficult, or may be even easier, to use the networking framework provided by the Linux kernel, and this is what SocketCAN does.

The use of the networking framework of the Linux kernel is just the natural and most appropriate way to implement CAN for Linux.

4.3 SocketCAN Concept

As described in [Motivation / Why Using the Socket API](#) the main goal of SocketCAN is to provide a socket interface to user space applications which builds upon the Linux network layer. In contrast to the commonly known TCP/IP and ethernet networking, the CAN bus is a broadcast-only(!) medium that has no MAC-layer addressing like ethernet. The CAN-identifier (can_id) is used for arbitration on the CAN-bus. Therefore the CAN-IDs have to be chosen uniquely on the bus. When designing a CAN-ECU network the CAN-IDs are mapped to be sent by a specific ECU. For this reason a CAN-ID can be treated best as a kind of source address.

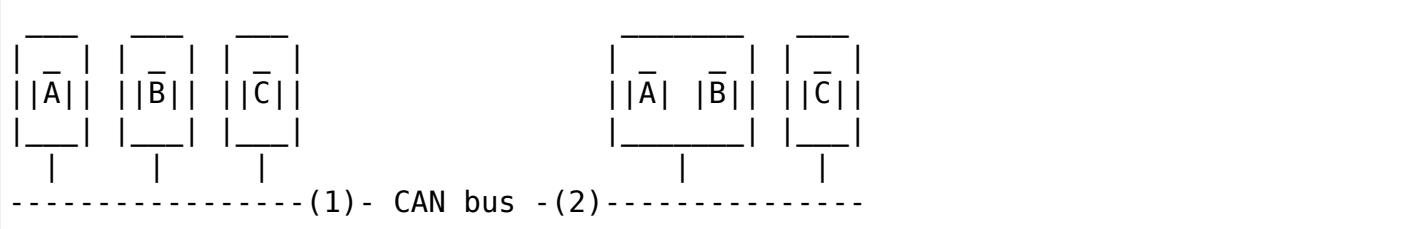
4.3.1 Receive Lists

The network transparent access of multiple applications leads to the problem that different applications may be interested in the same CAN-IDs from the same CAN network interface. The SocketCAN core module - which implements the protocol family CAN - provides several high efficient receive lists for this reason. If e.g. a user space application opens a CAN RAW socket, the raw protocol module itself requests the (range of) CAN-IDs from the SocketCAN core that are requested by the user. The subscription and unsubscription of CAN-IDs can be done for specific CAN interfaces or for all(!) known CAN interfaces with the can_rx_(un)register() functions provided to CAN protocol modules by the SocketCAN core (see [SocketCAN Core Module](#)).

To optimize the CPU usage at runtime the receive lists are split up into several specific lists per device that match the requested filter complexity for a given use-case.

4.3.2 Local Loopback of Sent Frames

As known from other networking concepts the data exchanging applications may run on the same or different nodes without any change (except for the according addressing information):



To ensure that application A receives the same information in the example (2) as it would receive in example (1) there is need for some kind of local loopback of the sent CAN frames on the appropriate node.

The Linux network devices (by default) just can handle the transmission and reception of media dependent frames. Due to the arbitration on the CAN bus the transmission of a low prio CAN-ID may be delayed by the reception of a high prio CAN frame. To reflect the correct¹ traffic on the node the loopback of the sent data has to be performed right after a successful transmission. If the CAN network interface is not capable of performing the loopback for some reason the SocketCAN core can do this task as a fallback solution. See [Local Loopback of Sent Frames](#) for details (recommended).

The loopback functionality is enabled by default to reflect standard networking behaviour for CAN applications. Due to some requests from the RT-SocketCAN group the loopback optionally may be disabled for each separate socket. See sockopts from the CAN RAW sockets in [RAW Protocol Sockets with can_filters \(SOCK_RAW\)](#).

4.3.3 Network Problem Notifications

The use of the CAN bus may lead to several problems on the physical and media access control layer. Detecting and logging of these lower layer problems is a vital requirement for CAN users to identify hardware issues on the physical transceiver layer as well as arbitration problems and error frames caused by the different ECUs. The occurrence of detected errors are important for diagnosis and have to be logged together with the exact timestamp. For this reason the CAN interface driver can generate so called Error Message Frames that can optionally be passed to the user application in the same way as other CAN frames. Whenever an error on the physical layer or the MAC layer is detected (e.g. by the CAN controller) the driver creates an appropriate error message frame. Error messages frames can be requested by the user application using the common CAN filter mechanisms. Inside this filter definition the (interested) type of errors may be selected. The reception of error messages is disabled by default. The format of the CAN error message frame is briefly described in the Linux header file "include/uapi/linux/can/error.h".

¹ you really like to have this when you're running analyser tools like 'candump' or 'cansniffer' on the (same) node.

4.4 How to use SocketCAN

Like TCP/IP, you first need to open a socket for communicating over a CAN network. Since SocketCAN implements a new protocol family, you need to pass PF_CAN as the first argument to the socket(2) system call. Currently, there are two CAN protocols to choose from, the raw socket protocol and the broadcast manager (BCM). So to open a socket, you would write:

```
s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
```

and:

```
s = socket(PF_CAN, SOCK_DGRAM, CAN_BCM);
```

respectively. After the successful creation of the socket, you would normally use the bind(2) system call to bind the socket to a CAN interface (which is different from TCP/IP due to different addressing - see [SocketCAN Concept](#)). After binding (CAN_RAW) or connecting (CAN_BCM) the socket, you can read(2) and write(2) from/to the socket or use send(2), sendto(2), sendmsg(2) and the recv* counterpart operations on the socket as usual. There are also CAN specific socket options described below.

The Classical CAN frame structure (aka CAN 2.0B), the CAN FD frame structure and the sock-addr structure are defined in include/linux/can.h:

```
struct can_frame {
    canid_t can_id; /* 32 bit CAN_ID + EFF/RTR/ERR flags */
    union {
        /* CAN frame payload length in byte (0 .. CAN_MAX_DLEN)
         * was previously named can_dlc so we need to carry that
         * name for legacy support
         */
        __u8 len;
        __u8 can_dlc; /* deprecated */
    };
    __u8 __pad; /* padding */
    __u8 __res0; /* reserved / padding */
    __u8 len8_dlc; /* optional DLC for 8 byte payload length (9 .. 15) */
    __u8 data[8] __attribute__((aligned(8)));
};
```

Remark: The len element contains the payload length in bytes and should be used instead of can_dlc. The deprecated can_dlc was misleadingly named as it always contained the plain payload length in bytes and not the so called 'data length code' (DLC).

To pass the raw DLC from/to a Classical CAN network device the len8_dlc element can contain values 9 .. 15 when the len element is 8 (the real payload length for all DLC values greater or equal to 8).

The alignment of the (linear) payload data[] to a 64bit boundary allows the user to define their own structs and unions to easily access the CAN payload. There is no given byteorder on the CAN bus by default. A read(2) system call on a CAN_RAW socket transfers a struct can_frame to the user space.

The sockaddr_can structure has an interface index like the PF_PACKET socket, that also binds to a specific interface:

```
struct sockaddr_can {
    sa_family_t can_family;
    int         can_ifindex;
    union {
        /* transport protocol class address info (e.g. ISOTP) */
        struct { canid_t rx_id, tx_id; } tp;

        /* J1939 address information */
        struct {
            /* 8 byte name when using dynamic addressing */
            __u64 name;

            /* pgn:
             * 8 bit: PS in PDU2 case, else 0
             * 8 bit: PF
             * 1 bit: DP
             * 1 bit: reserved
             */
            __u32 pgn;

            /* 1 byte address */
            __u8 addr;
        } j1939;

        /* reserved for future CAN protocols address information */
    } can_addr;
};
```

To determine the interface index an appropriate ioctl() has to be used (example for CAN_RAW sockets without error checking):

```
int s;
struct sockaddr_can addr;
struct ifreq ifr;

s = socket(PF_CAN, SOCK_RAW, CAN_RAW);

strcpy(ifr.ifr_name, "can0" );
ioctl(s, SIOCGIFINDEX, &ifr);

addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;

bind(s, (struct sockaddr *)&addr, sizeof(addr));

(..)
```

To bind a socket to all(!) CAN interfaces the interface index must be 0 (zero). In this case the socket receives CAN frames from every enabled CAN interface. To determine the originating

CAN interface the system call recvfrom(2) may be used instead of read(2). To send on a socket that is bound to 'any' interface sendto(2) is needed to specify the outgoing interface.

Reading CAN frames from a bound CAN_RAW socket (see above) consists of reading a struct can_frame:

```
struct can_frame frame;

nbytes = read(s, &frame, sizeof(struct can_frame));

if (nbytes < 0) {
    perror("can raw socket read");
    return 1;
}

/* paranoid check ... */
if (nbytes < sizeof(struct can_frame)) {
    fprintf(stderr, "read: incomplete CAN frame\n");
    return 1;
}

/* do something with the received CAN frame */
```

Writing CAN frames can be done similarly, with the write(2) system call:

```
nbytes = write(s, &frame, sizeof(struct can_frame));
```

When the CAN interface is bound to 'any' existing CAN interface (addr.can_ifindex = 0) it is recommended to use recvfrom(2) if the information about the originating CAN interface is needed:

```
struct sockaddr_can addr;
struct ifreq ifr;
socklen_t len = sizeof(addr);
struct can_frame frame;

nbytes = recvfrom(s, &frame, sizeof(struct can_frame),
                  0, (struct sockaddr*)&addr, &len);

/* get interface name of the received CAN frame */
ifr.ifr_ifindex = addr.can_ifindex;
ioctl(s, SIOCGIFNAME, &ifr);
printf("Received a CAN frame from interface %s", ifr.ifr_name);
```

To write CAN frames on sockets bound to 'any' CAN interface the outgoing interface has to be defined certainly:

```
strcpy(ifr.ifr_name, "can0");
ioctl(s, SIOCGIFINDEX, &ifr);
addr.can_ifindex = ifr.ifr_ifindex;
addr.can_family = AF_CAN;

nbytes = sendto(s, &frame, sizeof(struct can_frame),
                0, (struct sockaddr*)&addr, sizeof(addr));
```

An accurate timestamp can be obtained with an ioctl(2) call after reading a message from the socket:

```
struct timeval tv;
ioctl(s, SIOCGSTAMP, &tv);
```

The timestamp has a resolution of one microsecond and is set automatically at the reception of a CAN frame.

Remark about CAN FD (flexible data rate) support:

Generally the handling of CAN FD is very similar to the formerly described examples. The new CAN FD capable CAN controllers support two different bitrates for the arbitration phase and the payload phase of the CAN FD frame and up to 64 bytes of payload. This extended payload length breaks all the kernel interfaces (ABI) which heavily rely on the CAN frame with fixed eight bytes of payload (struct can_frame) like the CAN_RAW socket. Therefore e.g. the CAN_RAW socket supports a new socket option CAN_RAW_FD_FRAMES that switches the socket into a mode that allows the handling of CAN FD frames and Classical CAN frames simultaneously (see [RAW Socket Option CAN_RAW_FD_FRAMES](#)).

The struct canfd_frame is defined in include/linux/can.h:

```
struct canfd_frame {
    canid_t can_id; /* 32 bit CAN_ID + EFF/RTR/ERR flags */
    __u8 len; /* frame payload length in byte (0 .. 64) */
    __u8 flags; /* additional flags for CAN FD */
    __u8 __res0; /* reserved / padding */
    __u8 __res1; /* reserved / padding */
    __u8 data[64] __attribute__((aligned(8)));
};
```

The struct canfd_frame and the existing struct can_frame have the can_id, the payload length and the payload data at the same offset inside their structures. This allows to handle the different structures very similar. When the content of a struct can_frame is copied into a struct canfd_frame all structure elements can be used as-is - only the data[] becomes extended.

When introducing the struct canfd_frame it turned out that the data length code (DLC) of the struct can_frame was used as a length information as the length and the DLC has a 1:1 mapping in the range of 0 .. 8. To preserve the easy handling of the length information the canfd_frame.len element contains a plain length value from 0 .. 64. So both canfd_frame.len and can_frame.len are equal and contain a length information and no DLC. For details about the distinction of CAN and CAN FD capable devices and the mapping to the bus-relevant data length code (DLC), see [CAN FD \(Flexible Data Rate\) Driver Support](#).

The length of the two CAN(FD) frame structures define the maximum transfer unit (MTU) of the CAN(FD) network interface and skbuff data length. Two definitions are specified for CAN specific MTUs in include/linux/can.h:

```
#define CAN_MTU (sizeof(struct can_frame)) == 16 => Classical CAN frame
#define CANFD_MTU (sizeof(struct canfd_frame)) == 72 => CAN FD frame
```

4.4.1 RAW Protocol Sockets with can_filters (SOCK_RAW)

Using CAN_RAW sockets is extensively comparable to the commonly known access to CAN character devices. To meet the new possibilities provided by the multi user SocketCAN approach, some reasonable defaults are set at RAW socket binding time:

- The filters are set to exactly one filter receiving everything
- The socket only receives valid data frames (=> no error message frames)
- The loopback of sent CAN frames is enabled (see *Local Loopback of Sent Frames*)
- The socket does not receive its own sent frames (in loopback mode)

These default settings may be changed before or after binding the socket. To use the referenced definitions of the socket options for CAN_RAW sockets, include <linux/can/raw.h>.

RAW socket option CAN_RAW_FILTER

The reception of CAN frames using CAN_RAW sockets can be controlled by defining 0 .. n filters with the CAN_RAW_FILTER socket option.

The CAN filter structure is defined in include/linux/can.h:

```
struct can_filter {
    canid_t can_id;
    canid_t can_mask;
};
```

A filter matches, when:

```
<received_can_id> & mask == can_id & mask
```

which is analogous to known CAN controllers hardware filter semantics. The filter can be inverted in this semantic, when the CAN_INV_FILTER bit is set in can_id element of the can_filter structure. In contrast to CAN controller hardware filters the user may set 0 .. n receive filters for each open socket separately:

```
struct can_filter rfilter[2];

rfilter[0].can_id    = 0x123;
rfilter[0].can_mask = CAN_SFF_MASK;
rfilter[1].can_id    = 0x200;
rfilter[1].can_mask = 0x700;

setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &rfilter, sizeof(rfilter));
```

To disable the reception of CAN frames on the selected CAN_RAW socket:

```
setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, NULL, 0);
```

To set the filters to zero filters is quite obsolete as to not read data causes the raw socket to discard the received CAN frames. But having this 'send only' use-case we may remove the receive list in the Kernel to save a little (really a very little!) CPU usage.

CAN Filter Usage Optimisation

The CAN filters are processed in per-device filter lists at CAN frame reception time. To reduce the number of checks that need to be performed while walking through the filter lists the CAN core provides an optimized filter handling when the filter subscription focusses on a single CAN ID.

For the possible 2048 SFF CAN identifiers the identifier is used as an index to access the corresponding subscription list without any further checks. For the 2^{29} possible EFF CAN identifiers a 10 bit XOR folding is used as hash function to retrieve the EFF table index.

To benefit from the optimized filters for single CAN identifiers the CAN_SFF_MASK or CAN_EFF_MASK have to be set into can_filter.mask together with set CAN_EFF_FLAG and CAN_RTR_FLAG bits. A set CAN_EFF_FLAG bit in the can_filter.mask makes clear that it matters whether a SFF or EFF CAN ID is subscribed. E.g. in the example from above:

```
rfilter[0].can_id = 0x123;
rfilter[0].can_mask = CAN_SFF_MASK;
```

both SFF frames with CAN ID 0x123 and EFF frames with 0xXXXXX123 can pass.

To filter for only 0x123 (SFF) and 0x12345678 (EFF) CAN identifiers the filter has to be defined in this way to benefit from the optimized filters:

```
struct can_filter rfilter[2];

rfilter[0].can_id = 0x123;
rfilter[0].can_mask = (CAN_EFF_FLAG | CAN_RTR_FLAG | CAN_SFF_MASK);
rfilter[1].can_id = 0x12345678 | CAN_EFF_FLAG;
rfilter[1].can_mask = (CAN_EFF_FLAG | CAN_RTR_FLAG | CAN_EFF_MASK);

setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &rfilter, sizeof(rfilter));
```

RAW Socket Option CAN_RAW_ERR_FILTER

As described in [Network Problem Notifications](#) the CAN interface driver can generate so called Error Message Frames that can optionally be passed to the user application in the same way as other CAN frames. The possible errors are divided into different error classes that may be filtered using the appropriate error mask. To register for every possible error condition CAN_ERR_MASK can be used as value for the error mask. The values for the error mask are defined in linux/can/error.h:

```
can_err_mask_t err_mask = ( CAN_ERR_TX_TIMEOUT | CAN_ERR_BUSOFF );

setsockopt(s, SOL_CAN_RAW, CAN_RAW_ERR_FILTER,
           &err_mask, sizeof(err_mask));
```

RAW Socket Option CAN_RAW_LOOPBACK

To meet multi user needs the local loopback is enabled by default (see [Local Loopback of Sent Frames](#) for details). But in some embedded use-cases (e.g. when only one application uses the CAN bus) this loopback functionality can be disabled (separately for each socket):

```
int loopback = 0; /* 0 = disabled, 1 = enabled (default) */

setsockopt(s, SOL_CAN_RAW, CAN_RAW_LOOPBACK, &loopback, sizeof(loopback));
```

RAW socket option CAN_RAW_RECV_OWN_MSGS

When the local loopback is enabled, all the sent CAN frames are looped back to the open CAN sockets that registered for the CAN frames' CAN-ID on this given interface to meet the multi user needs. The reception of the CAN frames on the same socket that was sending the CAN frame is assumed to be unwanted and therefore disabled by default. This default behaviour may be changed on demand:

```
int recv_own_msgs = 1; /* 0 = disabled (default), 1 = enabled */

setsockopt(s, SOL_CAN_RAW, CAN_RAW_RECV_OWN_MSGS,
           &recv_own_msgs, sizeof(recv_own_msgs));
```

Note that reception of a socket's own CAN frames are subject to the same filtering as other CAN frames (see [RAW socket option CAN_RAW_FILTER](#)).

RAW Socket Option CAN_RAW_FD_FRAMES

CAN FD support in CAN_RAW sockets can be enabled with a new socket option CAN_RAW_FD_FRAMES which is off by default. When the new socket option is not supported by the CAN_RAW socket (e.g. on older kernels), switching the CAN_RAW_FD_FRAMES option returns the error -ENOPROTOOPT.

Once CAN_RAW_FD_FRAMES is enabled the application can send both CAN frames and CAN FD frames. OTOH the application has to handle CAN and CAN FD frames when reading from the socket:

CAN_RAW_FD_FRAMES enabled: CAN_MTU and CANFD_MTU are allowed
CAN_RAW_FD_FRAMES disabled: only CAN_MTU is allowed (default)

Example:

```
[ remember: CANFD_MTU == sizeof(struct canfd_frame) ]

struct canfd_frame cfd;

nbytes = read(s, &cfд, CANFD_MTU);

if (nbytes == CANFD_MTU) {
    printf("got CAN FD frame with length %d\n", cfd.len);
    /* cfd.flags contains valid data */
```

```

} else if (nbytes == CAN_MTU) {
    printf("got Classical CAN frame with length %d\n", cfd.len);
    /* cfd.flags is undefined */
} else {
    fprintf(stderr, "read: invalid CAN(FD) frame\n");
    return 1;
}

/* the content can be handled independently from the received MTU size */

printf("can_id: %X data length: %d data: ", cfd.can_id, cfd.len);
for (i = 0; i < cfd.len; i++)
    printf("%02X ", cfd.data[i]);

```

When reading with size CANFD_MTU only returns CAN_MTU bytes that have been received from the socket a Classical CAN frame has been read into the provided CAN FD structure. Note that the canfd_frame.flags data field is not specified in the struct can_frame and therefore it is only valid in CANFD_MTU sized CAN FD frames.

Implementation hint for new CAN applications:

To build a CAN FD aware application use struct canfd_frame as basic CAN data structure for CAN_RAW based applications. When the application is executed on an older Linux kernel and switching the CAN_RAW_FD_FRAMES socket option returns an error: No problem. You'll get Classical CAN frames or CAN FD frames and can process them the same way.

When sending to CAN devices make sure that the device is capable to handle CAN FD frames by checking if the device maximum transfer unit is CANFD_MTU. The CAN device MTU can be retrieved e.g. with a SIOCGIFMTU ioctl() syscall.

RAW socket option CAN_RAW_JOIN_FILTERS

The CAN_RAW socket can set multiple CAN identifier specific filters that lead to multiple filters in the af_can.c filter processing. These filters are independent from each other which leads to logical OR'ed filters when applied (see [RAW socket option CAN_RAW_FILTER](#)).

This socket option joins the given CAN filters in the way that only CAN frames are passed to user space that matched *all* given CAN filters. The semantic for the applied filters is therefore changed to a logical AND.

This is useful especially when the filterset is a combination of filters where the CAN_INV_FILTER flag is set in order to notch single CAN IDs or CAN ID ranges from the incoming traffic.

RAW Socket Returned Message Flags

When using recvmsg() call, the msg->msg_flags may contain following flags:

MSG_DONTROUTE:

set when the received frame was created on the local host.

MSG_CONFIRM:

set when the frame was sent via the socket it is received on. This flag can be interpreted as a 'transmission confirmation' when the CAN driver supports the echo of frames on driver level, see [Local Loopback of Sent Frames](#) and [Local Loopback of Sent Frames](#). In order to receive such messages, CAN_RAW_RECV_OWN_MSGS must be set.

4.4.2 Broadcast Manager Protocol Sockets (SOCK_DGRAM)

The Broadcast Manager protocol provides a command based configuration interface to filter and send (e.g. cyclic) CAN messages in kernel space.

Receive filters can be used to down sample frequent messages; detect events such as message contents changes, packet length changes, and do time-out monitoring of received messages.

Periodic transmission tasks of CAN frames or a sequence of CAN frames can be created and modified at runtime; both the message content and the two possible transmit intervals can be altered.

A BCM socket is not intended for sending individual CAN frames using the struct can_frame as known from the CAN_RAW socket. Instead a special BCM configuration message is defined. The basic BCM configuration message used to communicate with the broadcast manager and the available operations are defined in the linux/can/bcm.h include. The BCM message consists of a message header with a command ('opcode') followed by zero or more CAN frames. The broadcast manager sends responses to user space in the same form:

```
struct bcm_msg_head {
    __u32 opcode;                      /* command */
    __u32 flags;                       /* special flags */
    __u32 count;                        /* run 'count' times with ival1 */
    struct timeval ival1, ival2;        /* count and subsequent interval */
    canid_t can_id;                   /* unique can_id for task */
    __u32 nframes;                     /* number of can_frames following */
    struct can_frame frames[0];
};
```

The aligned payload 'frames' uses the same basic CAN frame structure defined at the beginning of [RAW Socket Option CAN_RAW_FD_FRAMES](#) and in the include/linux/can.h include. All messages to the broadcast manager from user space have this structure.

Note a CAN_BCM socket must be connected instead of bound after socket creation (example without error checking):

```
int s;
struct sockaddr_can addr;
struct ifreq ifr;

s = socket(PF_CAN, SOCK_DGRAM, CAN_BCM);
```

```

strcpy(ifr.ifr_name, "can0");
ioctl(s, SIOCGIFINDEX, &ifr);

addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;

connect(s, (struct sockaddr *)&addr, sizeof(addr));

(...)

```

The broadcast manager socket is able to handle any number of in flight transmissions or receive filters concurrently. The different RX/TX jobs are distinguished by the unique can_id in each BCM message. However additional CAN_BCM sockets are recommended to communicate on multiple CAN interfaces. When the broadcast manager socket is bound to 'any' CAN interface (=> the interface index is set to zero) the configured receive filters apply to any CAN interface unless the sendto() syscall is used to overrule the 'any' CAN interface index. When using recvfrom() instead of read() to retrieve BCM socket messages the originating CAN interface is provided in can_ifindex.

Broadcast Manager Operations

The opcode defines the operation for the broadcast manager to carry out, or details the broadcast managers response to several events, including user requests.

Transmit Operations (user space to broadcast manager):

TX_SETUP:

Create (cyclic) transmission task.

TX_DELETE:

Remove (cyclic) transmission task, requires only can_id.

TX_READ:

Read properties of (cyclic) transmission task for can_id.

TX_SEND:

Send one CAN frame.

Transmit Responses (broadcast manager to user space):

TX_STATUS:

Reply to TX_READ request (transmission task configuration).

TX_EXPIRED:

Notification when counter finishes sending at initial interval 'ival1'. Requires the TX_COUNTEVT flag to be set at TX_SETUP.

Receive Operations (user space to broadcast manager):

RX_SETUP:

Create RX content filter subscription.

RX_DELETE:

Remove RX content filter subscription, requires only can_id.

RX_READ:

Read properties of RX content filter subscription for can_id.

Receive Responses (broadcast manager to user space):

RX_STATUS:

Reply to RX_READ request (filter task configuration).

RX_TIMEOUT:

Cyclic message is detected to be absent (timer ival1 expired).

RX_CHANGED:

BCM message with updated CAN frame (detected content change). Sent on first message received or on receipt of revised CAN messages.

Broadcast Manager Message Flags

When sending a message to the broadcast manager the 'flags' element may contain the following flag definitions which influence the behaviour:

SETTIMER:

Set the values of ival1, ival2 and count

STARTTIMER:

Start the timer with the actual values of ival1, ival2 and count. Starting the timer leads simultaneously to emit a CAN frame.

TX_COUNTEVT:

Create the message TX_EXPIRED when count expires

TX_ANNOUNCE:

A change of data by the process is emitted immediately.

TX_CP_CAN_ID:

Copies the can_id from the message header to each subsequent frame in frames. This is intended as usage simplification. For TX tasks the unique can_id from the message header may differ from the can_id(s) stored for transmission in the subsequent struct can_frame(s).

RX_FILTER_ID:

Filter by can_id alone, no frames required (nframes=0).

RX_CHECK_DLC:

A change of the DLC leads to an RX_CHANGED.

RX_NO_AUTOTIMER:

Prevent automatically starting the timeout monitor.

RX_ANNOUNCE_RESUME:

If passed at RX_SETUP and a receive timeout occurred, a RX_CHANGED message will be generated when the (cyclic) receive restarts.

TX_RESET_MULTI_IDX:

Reset the index for the multiple frame transmission.

RX_RTR_FRAME:

Send reply for RTR-request (placed in op->frames[0]).

CAN_FD_FRAME:

The CAN frames following the bcm_msg_head are struct canfd_frame's

Broadcast Manager Transmission Timers

Periodic transmission configurations may use up to two interval timers. In this case the BCM sends a number of messages ('count') at an interval 'ival1', then continuing to send at another given interval 'ival2'. When only one timer is needed 'count' is set to zero and only 'ival2' is used. When SET_TIMER and START_TIMER flag were set the timers are activated. The timer values can be altered at runtime when only SET_TIMER is set.

Broadcast Manager message sequence transmission

Up to 256 CAN frames can be transmitted in a sequence in the case of a cyclic TX task configuration. The number of CAN frames is provided in the 'nframes' element of the BCM message head. The defined number of CAN frames are added as array to the TX_SETUP BCM configuration message:

```
/* create a struct to set up a sequence of four CAN frames */
struct {
    struct bcm_msg_head msg_head;
    struct can_frame frame[4];
} mytxmsg;

(..)
mytxmsg.msg_head.nframes = 4;
(..)

write(s, &mytxmsg, sizeof(mytxmsg));
```

With every transmission the index in the array of CAN frames is increased and set to zero at index overflow.

Broadcast Manager Receive Filter Timers

The timer values ival1 or ival2 may be set to non-zero values at RX_SETUP. When the SET_TIMER flag is set the timers are enabled:

ival1:

Send RX_TIMEOUT when a received message is not received again within the given time. When START_TIMER is set at RX_SETUP the timeout detection is activated directly - even without a former CAN frame reception.

ival2:

Throttle the received message rate down to the value of ival2. This is useful to reduce messages for the application when the signal inside the CAN frame is stateless as state changes within the ival2 period may get lost.

Broadcast Manager Multiplex Message Receive Filter

To filter for content changes in multiplex message sequences an array of more than one CAN frames can be passed in a RX_SETUP configuration message. The data bytes of the first CAN frame contain the mask of relevant bits that have to match in the subsequent CAN frames with the received CAN frame. If one of the subsequent CAN frames is matching the bits in that frame data mark the relevant content to be compared with the previous received content. Up to 257 CAN frames (multiplex filter bit mask CAN frame plus 256 CAN filters) can be added as array to the TX_SETUP BCM configuration message:

```
/* usually used to clear CAN frame data[] - beware of endian problems! */
#define U64_DATA(p) (*(unsigned long long*)(p)->data)

struct {
    struct bcm_msg_head msg_head;
    struct can_frame frame[5];
} msg;

msg.msg_head.opcode = RX_SETUP;
msg.msg_head.can_id = 0x42;
msg.msg_head.flags = 0;
msg.msg_head.nframes = 5;
U64_DATA(&msg.frame[0]) = 0xFF0000000000000ULL; /* MUX mask */
U64_DATA(&msg.frame[1]) = 0x0100000000000FFULL; /* data mask (MUX 0x01) */
U64_DATA(&msg.frame[2]) = 0x0200FFFF000000FFULL; /* data mask (MUX 0x02) */
U64_DATA(&msg.frame[3]) = 0x330000FFFFFF0003ULL; /* data mask (MUX 0x33) */
U64_DATA(&msg.frame[4]) = 0x4F07FC0FF0000000ULL; /* data mask (MUX 0x4F) */

write(s, &msg, sizeof(msg));
```

Broadcast Manager CAN FD Support

The programming API of the CAN_BCM depends on struct can_frame which is given as array directly behind the bcm_msg_head structure. To follow this schema for the CAN FD frames a new flag 'CAN_FD_FRAME' in the bcm_msg_head flags indicates that the concatenated CAN frame structures behind the bcm_msg_head are defined as struct canfd_frame:

```
struct {
    struct bcm_msg_head msg_head;
    struct canfd_frame frame[5];
} msg;

msg.msg_head.opcode = RX_SETUP;
msg.msg_head.can_id = 0x42;
msg.msg_head.flags = CAN_FD_FRAME;
msg.msg_head.nframes = 5;
(...)
```

When using CAN FD frames for multiplex filtering the MUX mask is still expected in the first 64 bit of the struct canfd_frame data section.

4.4.3 Connected Transport Protocols (SOCK_SEQPACKET)

(to be written)

4.4.4 Unconnected Transport Protocols (SOCK_DGRAM)

(to be written)

4.5 SocketCAN Core Module

The SocketCAN core module implements the protocol family PF_CAN. CAN protocol modules are loaded by the core module at runtime. The core module provides an interface for CAN protocol modules to subscribe needed CAN IDs (see [Receive Lists](#)).

4.5.1 can.ko Module Params

- **stats_timer**: To calculate the SocketCAN core statistics (e.g. current/maximum frames per second) this 1 second timer is invoked at can.ko module start time by default. This timer can be disabled by using stattimer=0 on the module commandline.
- **debug**: (removed since SocketCAN SVN r546)

4.5.2 procfs content

As described in [Receive Lists](#) the SocketCAN core uses several filter lists to deliver received CAN frames to CAN protocol modules. These receive lists, their filters and the count of filter matches can be checked in the appropriate receive list. All entries contain the device and a protocol module identifier:

```
foo@bar:~$ cat /proc/net/can/recvlist_all

receive list 'rx_all':
(vcan3: no entry)
(vcan2: no entry)
(vcan1: no entry)
device    can_id      can_mask   function  userdata   matches   ident
  vcan0      000      00000000   f88e6370   f6c6f400       0   raw
(any: no entry)
```

In this example an application requests any CAN traffic from vcan0:

```
recvlist_all - list for unfiltered entries (no filter operations)
recvlist_eff - list for single extended frame (EFF) entries
recvlist_err - list for error message frames masks
recvlist_fil - list for mask/value filters
recvlist_inv - list for mask/value filters (inverse semantic)
recvlist_sff - list for single standard frame (SFF) entries
```

Additional procfs files in /proc/net/can:

stats	- SocketCAN core statistics (rx/tx frames, match ratios, ...)
reset_stats	- manual statistic reset
version	- prints SocketCAN core and ABI version (removed in Linux 5.10)

4.5.3 Writing Own CAN Protocol Modules

To implement a new protocol in the protocol family PF_CAN a new protocol has to be defined in include/linux/can.h . The prototypes and definitions to use the SocketCAN core can be accessed by including include/linux/can/core.h . In addition to functions that register the CAN protocol and the CAN device notifier chain there are functions to subscribe CAN frames received by CAN interfaces and to send CAN frames:

can_rx_register	- subscribe CAN frames from a specific interface
can_rx_unregister	- unsubscribe CAN frames from a specific interface
can_send	- transmit a CAN frame (optional with local loopback)

For details see the kerneldoc documentation in net/can/af_can.c or the source code of net/can/raw.c or net/can/bcm.c .

4.6 CAN Network Drivers

Writing a CAN network device driver is much easier than writing a CAN character device driver. Similar to other known network device drivers you mainly have to deal with:

- TX: Put the CAN frame from the socket buffer to the CAN controller.
- RX: Put the CAN frame from the CAN controller to the socket buffer.

See e.g. at *Network Devices, the Kernel, and You!* . The differences for writing CAN network device driver are described below:

4.6.1 General Settings

```
dev->type = ARPHRD_CAN; /* the netdevice hardware type */
dev->flags = IFF_NOARP; /* CAN has no arp */

dev->mtu = CAN_MTU; /* sizeof(struct can_frame) -> Classical CAN interface */

or alternative, when the controller supports CAN with flexible data rate:
dev->mtu = CANFD_MTU; /* sizeof(struct canfd_frame) -> CAN FD interface */
```

The struct can_frame or struct canfd_frame is the payload of each socket buffer (skbuff) in the protocol family PF_CAN.

4.6.2 Local Loopback of Sent Frames

As described in [Local Loopback of Sent Frames](#) the CAN network device driver should support a local loopback functionality similar to the local echo e.g. of tty devices. In this case the driver flag IFF_ECHO has to be set to prevent the PF_CAN core from locally echoing sent frames (aka loopback) as fallback solution:

```
dev->flags = (IFF_NOARP | IFF_ECHO);
```

4.6.3 CAN Controller Hardware Filters

To reduce the interrupt load on deep embedded systems some CAN controllers support the filtering of CAN IDs or ranges of CAN IDs. These hardware filter capabilities vary from controller to controller and have to be identified as not feasible in a multi-user networking approach. The use of the very controller specific hardware filters could make sense in a very dedicated use-case, as a filter on driver level would affect all users in the multi-user system. The high efficient filter sets inside the PF_CAN core allow to set different multiple filters for each socket separately. Therefore the use of hardware filters goes to the category 'handmade tuning on deep embedded systems'. The author is running a MPC603e @133MHz with four SJA1000 CAN controllers from 2002 under heavy bus load without any problems ...

4.6.4 Switchable Termination Resistors

CAN bus requires a specific impedance across the differential pair, typically provided by two 120Ohm resistors on the farthest nodes of the bus. Some CAN controllers support activating / deactivating a termination resistor(s) to provide the correct impedance.

Query the available resistances:

```
$ ip -details link show can0
...
termination 120 [ 0, 120 ]
```

Activate the terminating resistor:

```
$ ip link set dev can0 type can termination 120
```

Deactivate the terminating resistor:

```
$ ip link set dev can0 type can termination 0
```

To enable termination resistor support to a can-controller, either implement in the controller's struct can-priv:

```
termination_const
termination_const_cnt
do_set_termination
```

or add gpio control with the device tree entries from Documentation/devicetree/bindings/net/can/can-controller.yaml

4.6.5 The Virtual CAN Driver (vcan)

Similar to the network loopback devices, vcan offers a virtual local CAN interface. A full qualified address on CAN consists of

- a unique CAN Identifier (CAN ID)
- the CAN bus this CAN ID is transmitted on (e.g. can0)

so in common use cases more than one virtual CAN interface is needed.

The virtual CAN interfaces allow the transmission and reception of CAN frames without real CAN controller hardware. Virtual CAN network devices are usually named 'vcanX', like vcan0 vcan1 vcan2 ... When compiled as a module the virtual CAN driver module is called vcan.ko

Since Linux Kernel version 2.6.24 the vcan driver supports the Kernel netlink interface to create vcan network devices. The creation and removal of vcan network devices can be managed with the ip(8) tool:

- Create a virtual CAN network interface:
\$ ip link add type vcan
- Create a virtual CAN network interface with a specific name 'vcan42':
\$ ip link add dev vcan42 type vcan
- Remove a (virtual CAN) network interface 'vcan42':
\$ ip link del vcan42

4.6.6 The CAN Network Device Driver Interface

The CAN network device driver interface provides a generic interface to setup, configure and monitor CAN network devices. The user can then configure the CAN device, like setting the bit-timing parameters, via the netlink interface using the program "ip" from the "IPROUTE2" utility suite. The following chapter describes briefly how to use it. Furthermore, the interface uses a common data structure and exports a set of common functions, which all real CAN network device drivers should use. Please have a look to the SJA1000 or MSCAN driver to understand how to use them. The name of the module is can-dev.ko.

Netlink interface to set/get devices properties

The CAN device must be configured via netlink interface. The supported netlink message types are defined and briefly described in "include/linux/can/netlink.h". CAN link support for the program "ip" of the IPROUTE2 utility suite is available and it can be used as shown below:

Setting CAN device properties:

```
$ ip link set can0 type can help
Usage: ip link set DEVICE type can
      [ bitrate BITRATE [ sample-point SAMPLE-POINT] ] |
      [ tq TQ prop-seg PROP_SEG phase-seg1 PHASE-SEG1
        phase-seg2 PHASE-SEG2 [ sjw SJW ] ]
      [ dbitrate BITRATE [ dsample-point SAMPLE-POINT] ] |
```

```
[ dtq TQ dprop-seg PROP-SEG dphase-seg1 PHASE-SEG1
  dphase-seg2 PHASE-SEG2 [ dsjw SJW ] ]

[ loopback { on | off } ]
[ listen-only { on | off } ]
[ triple-sampling { on | off } ]
[ one-shot { on | off } ]
[ berr-reporting { on | off } ]
[ fd { on | off } ]
[ fd-non-iso { on | off } ]
[ presume-ack { on | off } ]
[ cc-len8-dlc { on | off } ]

[ restart-ms TIME-MS ]
[ restart ]

Where: BITRATE      := { 1..1000000 }
        SAMPLE-POINT := { 0.000..0.999 }
        TQ            := { NUMBER }
        PROP-SEG      := { 1..8 }
        PHASE-SEG1    := { 1..8 }
        PHASE-SEG2    := { 1..8 }
        SJW           := { 1..4 }
        RESTART-MS   := { 0 | NUMBER }
```

Display CAN device details and statistics:

```
$ ip -details -statistics link show can0
2: can0: <NOARP,UP,LOWER_UP,ECHO> mtu 16 qdisc pfifo_fast state UP qlen 10
link/can
can <TRIPLE-SAMPLING> state ERROR-ACTIVE restart-ms 100
bitrate 125000 sample_point 0.875
tq 125 prop-seg 6 phase-seg1 7 phase-seg2 2 sjw 1
sja1000: tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1
clock 8000000
re-started bus-errors arbit-lost error-warn error-pass bus-off
41       17457     0       41       42       41
RX: bytes packets errors dropped overrun mcast
140859   17608   17457   0       0       0
TX: bytes packets errors dropped carrier collsns
861      112      0       41       0       0
```

More info to the above output:

"<TRIPLE-SAMPLING>"

Shows the list of selected CAN controller modes: LOOPBACK, LISTEN-ONLY, or TRIPLE-SAMPLING.

"state ERROR-ACTIVE"

The current state of the CAN controller: "ERROR-ACTIVE", "ERROR-WARNING", "ERROR-PASSIVE", "BUS-OFF" or "STOPPED"

"restart-ms 100"

Automatic restart delay time. If set to a non-zero value, a restart of the CAN controller will be triggered automatically in case of a bus-off condition after the specified delay time in milliseconds. By default it's off.

"bitrate 125000 sample-point 0.875"

Shows the real bit-rate in bits/sec and the sample-point in the range 0.000..0.999. If the calculation of bit-timing parameters is enabled in the kernel (CONFIG_CAN_CALC_BITTIMING=y), the bit-timing can be defined by setting the "bitrate" argument. Optionally the "sample-point" can be specified. By default it's 0.000 assuming CIA-recommended sample-points.

"tq 125 prop-seg 6 phase-seg1 7 phase-seg2 2 sjw 1"

Shows the time quanta in ns, propagation segment, phase buffer segment 1 and 2 and the synchronisation jump width in units of tq. They allow to define the CAN bit-timing in a hardware independent format as proposed by the Bosch CAN 2.0 spec (see chapter 8 of <http://www.semiconductors.bosch.de/pdf/can2spec.pdf>).

"sja1000: tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1 clock 8000000"

Shows the bit-timing constants of the CAN controller, here the "sja1000". The minimum and maximum values of the time segment 1 and 2, the synchronisation jump width in units of tq, the bitrate pre-scaler and the CAN system clock frequency in Hz. These constants could be used for user-defined (non-standard) bit-timing calculation algorithms in user-space.

"re-started bus-errors arbit-lost error-warn error-pass bus-off"

Shows the number of restarts, bus and arbitration lost errors, and the state changes to the error-warning, error-passive and bus-off state. RX overrun errors are listed in the "overrun" field of the standard network statistics.

Setting the CAN Bit-Timing

The CAN bit-timing parameters can always be defined in a hardware independent format as proposed in the Bosch CAN 2.0 specification specifying the arguments "tq", "prop_seg", "phase_seg1", "phase_seg2" and "sjw":

```
$ ip link set canX type can tq 125 prop-seg 6 \
                           phase-seg1 7 phase-seg2 2 sjw 1
```

If the kernel option CONFIG_CAN_CALC_BITTIMING is enabled, CIA recommended CAN bit-timing parameters will be calculated if the bit-rate is specified with the argument "bitrate":

```
$ ip link set canX type can bitrate 125000
```

Note that this works fine for the most common CAN controllers with standard bit-rates but may *fail* for exotic bit-rates or CAN system clock frequencies. Disabling CONFIG_CAN_CALC_BITTIMING saves some space and allows user-space tools to solely determine and set the bit-timing parameters. The CAN controller specific bit-timing constants can be used for that purpose. They are listed by the following command:

```
$ ip -details link show can0
...
sja1000: clock 8000000 tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1
```

Starting and Stopping the CAN Network Device

A CAN network device is started or stopped as usual with the command "ifconfig canX up/down" or "ip link set canX up/down". Be aware that you *must* define proper bit-timing parameters for real CAN devices before you can start it to avoid error-prone default settings:

```
$ ip link set canX up type can bitrate 125000
```

A device may enter the "bus-off" state if too many errors occurred on the CAN bus. Then no more messages are received or sent. An automatic bus-off recovery can be enabled by setting the "restart-ms" to a non-zero value, e.g.:

```
$ ip link set canX type can restart-ms 100
```

Alternatively, the application may realize the "bus-off" condition by monitoring CAN error message frames and do a restart when appropriate with the command:

```
$ ip link set canX type can restart
```

Note that a restart will also create a CAN error message frame (see also *Network Problem Notifications*).

4.6.7 CAN FD (Flexible Data Rate) Driver Support

CAN FD capable CAN controllers support two different bitrates for the arbitration phase and the payload phase of the CAN FD frame. Therefore a second bit timing has to be specified in order to enable the CAN FD bitrate.

Additionally CAN FD capable CAN controllers support up to 64 bytes of payload. The representation of this length in can_frame.len and canfd_frame.len for userspace applications and inside the Linux network layer is a plain value from 0 .. 64 instead of the CAN 'data length code'. The data length code was a 1:1 mapping to the payload length in the Classical CAN frames anyway. The payload length to the bus-relevant DLC mapping is only performed inside the CAN drivers, preferably with the helper functions can_fd_dlc2len() and can_fd_len2dlc().

The CAN netdevice driver capabilities can be distinguished by the network devices maximum transfer unit (MTU):

```
MTU = 16 (CAN_MTU) => sizeof(struct can_frame) => Classical CAN device
MTU = 72 (CANFD_MTU) => sizeof(struct canfd_frame) => CAN FD capable device
```

The CAN device MTU can be retrieved e.g. with a SIOCGIFMTU ioctl() syscall. N.B. CAN FD capable devices can also handle and send Classical CAN frames.

When configuring CAN FD capable CAN controllers an additional 'data' bitrate has to be set. This bitrate for the data phase of the CAN FD frame has to be at least the bitrate which was configured for the arbitration phase. This second bitrate is specified analogue to the first bitrate but the bitrate setting keywords for the 'data' bitrate start with 'd' e.g. dbitrate, dsample-point, dsjw or dtq and similar settings. When a data bitrate is set within the configuration process the controller option "fd on" can be specified to enable the CAN FD mode in the CAN controller. This controller option also switches the device MTU to 72 (CANFD_MTU).

The first CAN FD specification presented as whitepaper at the International CAN Conference 2012 needed to be improved for data integrity reasons. Therefore two CAN FD implementations have to be distinguished today:

- ISO compliant: The ISO 11898-1:2015 CAN FD implementation (default)
- non-ISO compliant: The CAN FD implementation following the 2012 whitepaper

Finally there are three types of CAN FD controllers:

1. ISO compliant (fixed)
2. non-ISO compliant (fixed, like the M_CAN IP core v3.0.1 in m_can.c)
3. ISO/non-ISO CAN FD controllers (switchable, like the PEAK PCAN-USB FD)

The current ISO/non-ISO mode is announced by the CAN controller driver via netlink and displayed by the 'ip' tool (controller option FD-NON-ISO). The ISO/non-ISO-mode can be altered by setting 'fd-non-iso {on|off}' for switchable CAN FD controllers only.

Example configuring 500 kbit/s arbitration bitrate and 4 Mbit/s data bitrate:

```
$ ip link set can0 up type can bitrate 500000 sample-point 0.75 \
                           dbitrate 4000000 dsample-point 0.8 fd on
$ ip -details link show can0
5: can0: <NOARP,UP,LOWER_UP,ECHO> mtu 72 qdisc pfifo_fast state UNKNOWN \
      mode DEFAULT group default qlen 10
link/can  promiscuity 0
can <FD> state ERROR-ACTIVE (berr-counter tx 0 rx 0) restart-ms 0
      bitrate 500000 sample-point 0.750
      tq 50 prop-seg 14 phase-seg1 15 phase-seg2 10 sjw 1
      pcan_usb_pro_fd: tseg1 1..64 tseg2 1..16 sjw 1..16 brp 1..1024 \
      brp-inc 1
      dbitrate 4000000 dsample-point 0.800
      dtq 12 dprop-seg 7 dphase-seg1 8 dphase-seg2 4 dsjw 1
      pcan_usb_pro_fd: dtseg1 1..16 dtseg2 1..8 dsjw 1..4 dbrp 1..1024 \
      dbrp-inc 1
      clock 80000000
```

Example when 'fd-non-iso on' is added on this switchable CAN FD adapter:

```
can <FD,FD-NON-ISO> state ERROR-ACTIVE (berr-counter tx 0 rx 0) restart-ms 0
```

4.6.8 Supported CAN Hardware

Please check the "Kconfig" file in "drivers/net/can" to get an actual list of the support CAN hardware. On the SocketCAN project website (see [SocketCAN Resources](#)) there might be further drivers available, also for older kernel versions.

4.7 SocketCAN Resources

The Linux CAN / SocketCAN project resources (project site / mailing list) are referenced in the MAINTAINERS file in the Linux source tree. Search for CAN NETWORK [LAYERS|DRIVERS].

4.8 Credits

- Oliver Hartkopp (PF_CAN core, filters, drivers, bcm, SJA1000 driver)
- Urs Thuermann (PF_CAN core, kernel integration, socket interfaces, raw, vcan)
- Jan Kizka (RT-SocketCAN core, Socket-API reconciliation)
- Wolfgang Grandegger (RT-SocketCAN core & drivers, Raw Socket-API reviews, CAN device driver interface, MSCAN driver)
- Robert Schwebel (design reviews, PTXdist integration)
- Marc Kleine-Budde (design reviews, Kernel 2.6 cleanups, drivers)
- Benedikt Spranger (reviews)
- Thomas Gleixner (LKML reviews, coding style, posting hints)
- Andrey Volkov (kernel subtree structure, ioctls, MSCAN driver)
- Matthias Brukner (first SJA1000 CAN netdevice implementation Q2/2003)
- Klaus Hitschler (PEAK driver integration)
- Uwe Koppe (CAN netdevices with PF_PACKET approach)
- Michael Schulze (driver layer loopback requirement, RT CAN drivers review)
- Pavel Pisa (Bit-timing calculation)
- Sascha Hauer (SJA1000 platform driver)
- Sebastian Haas (SJA1000 EMS PCI driver)
- Markus Plessing (SJA1000 EMS PCI driver)
- Per Dalen (SJA1000 Kvaser PCI driver)
- Sam Ravnborg (reviews, coding style, kbuild help)

THE UCAN PROTOCOL

UCAN is the protocol used by the microcontroller-based USB-CAN adapter that is integrated on System-on-Modules from Theobroma Systems and that is also available as a standalone USB stick.

The UCAN protocol has been designed to be hardware-independent. It is modeled closely after how Linux represents CAN devices internally. All multi-byte integers are encoded as Little Endian.

All structures mentioned in this document are defined in `drivers/net/can/usb/ucan.c`.

5.1 USB Endpoints

UCAN devices use three USB endpoints:

CONTROL endpoint

The driver sends device management commands on this endpoint

IN endpoint

The device sends CAN data frames and CAN error frames

OUT endpoint

The driver sends CAN data frames on the out endpoint

5.2 CONTROL Messages

UCAN devices are configured using vendor requests on the control pipe.

To support multiple CAN interfaces in a single USB device all configuration commands target the corresponding interface in the USB descriptor.

The driver uses `ucan_ctrl_command_in/out` and `ucan_device_request_in` to deliver commands to the device.

5.2.1 Setup Packet

<code>bmRequestType</code>	Direction Vendor (Interface or Device)
<code>bRequest</code>	Command Number
<code>wValue</code>	Subcommand Number (16 Bit) or 0 if not used
<code>wIndex</code>	USB Interface Index (0 for device commands)
<code>wLength</code>	<ul style="list-style-type: none"> • Host to Device - Number of bytes to transmit • Device to Host - Maximum Number of bytes to receive. If the device send less. Common ZLP semantics are used.

5.2.2 Error Handling

The device indicates failed control commands by stalling the pipe.

5.2.3 Device Commands

UCAN_DEVICE_GET_FW_STRING

Dev2Host; optional

Request the device firmware string.

5.2.4 Interface Commands

UCAN_COMMAND_START

Host2Dev; mandatory

Bring the CAN interface up.

Payload Format

`ucan_ctl_payload_t.cmd_start`

<code>mode</code>	or mask of <code>UCAN_MODE_*</code>
-------------------	-------------------------------------

UCAN_COMMAND_STOP

Host2Dev; mandatory

Stop the CAN interface

Payload Format

empty

UCAN_COMMAND_RESET

Host2Dev; mandatory

Reset the CAN controller (including error counters)

Payload Format

empty

UCAN_COMMAND_GET

Host2Dev; mandatory

Get Information from the Device

Subcommands

UCAN_COMMAND_GET_INFO

Request the device information structure `ucan_ctl_payload_t.device_info`.

See the `device_info` field for details, and `uapi/linux/can/netlink.h` for an explanation of the `can_bittiming` fields.

Payload Format

`ucan_ctl_payload_t.device_info`

UCAN_COMMAND_GET_PROTOCOL_VERSION

Request the device protocol version `ucan_ctl_payload_t.protocol_version`. The current protocol version is 3.

Payload Format

`ucan_ctl_payload_t.protocol_version`

Note: Devices that do not implement this command use the old protocol version 1

UCAN_COMMAND_SET_BITTIMING

Host2Dev; mandatory

Setup bittiming by sending the structure `ucan_ctl_payload_t.cmd_set_bittiming` (see `struct bittiming` for details)

Payload Format

`ucan_ctl_payload_t.cmd_set_bittiming`.

UCAN_SLEEP/WAKE*Host2Dev; optional*

Configure sleep and wake modes. Not yet supported by the driver.

UCAN_FILTER*Host2Dev; optional*

Setup hardware CAN filters. Not yet supported by the driver.

5.2.5 Allowed interface commands

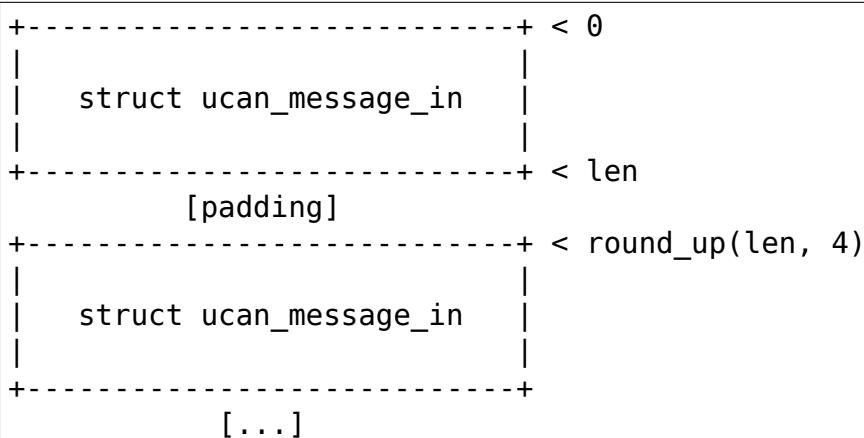
Legal Device State	Command	New Device State
stopped	SET_BITTIMING	stopped
stopped	START	started
started	STOP or RESET	stopped
stopped	STOP or RESET	stopped
started	RESTART	started
any	GET	<i>no change</i>

5.3 IN Message Format

A data packet on the USB IN endpoint contains one or more `ucan_message_in` values. If multiple messages are batched in a USB data packet, the `len` field can be used to jump to the next `ucan_message_in` value (take care to sanity-check the `len` value against the actual data size).

5.3.1 len field

Each `ucan_message_in` must be aligned to a 4-byte boundary (relative to the start of the start of the data buffer). That means that there may be padding bytes between multiple `ucan_message_in` values:



5.3.2 type field

The type field specifies the type of the message.

UCAN_IN_RX

subtype

zero

Data received from the CAN bus (ID + payload).

UCAN_IN_TX_COMPLETE

subtype

zero

The CAN device has sent a message to the CAN bus. It answers with a list of tuples <echo-ids, flags>.

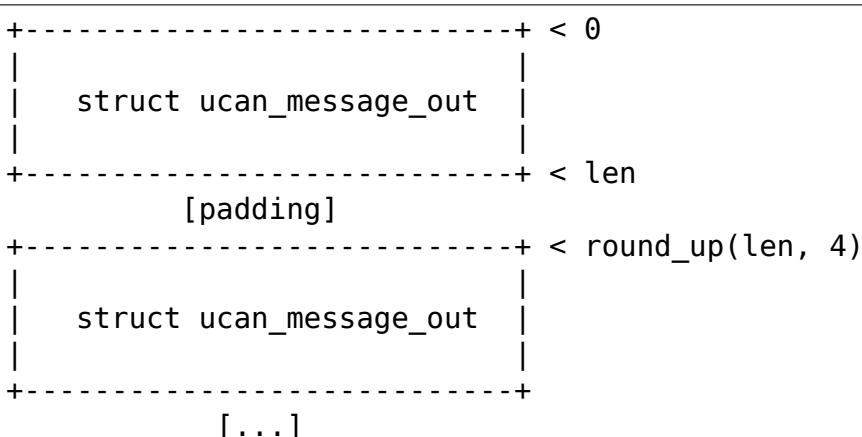
The echo-id identifies the frame from (echos the id from a previous UCAN_OUT_TX message). The flag indicates the result of the transmission. Whereas a set Bit 0 indicates success. All other bits are reserved and set to zero.

5.3.3 Flow Control

When receiving CAN messages there is no flow control on the USB buffer. The driver has to handle inbound message quickly enough to avoid drops. In case the device buffer overflow the condition is reported by sending corresponding error frames (see [CAN Error Handling](#))

5.4 OUT Message Format

A data packet on the USB OUT endpoint contains one or more `struct ucan_message_out` values. If multiple messages are batched into one data packet, the device uses the `len` field to jump to the next `ucan_message_out` value. Each `ucan_message_out` must be aligned to 4 bytes (relative to the start of the data buffer). The mechanism is same as described in [len field](#).



5.4.1 type field

In protocol version 3 only UCAN_OUT_TX is defined, others are used only by legacy devices (protocol version 1).

UCAN_OUT_TX

subtype

echo id to be replied within a CAN_IN_TX_COMPLETE message

Transmit a CAN frame. (parameters: id, data)

5.4.2 Flow Control

When the device outbound buffers are full it starts sending *NAKs* on the *OUT* pipe until more buffers are available. The driver stops the queue when a certain threshold of out packets are incomplete.

5.5 CAN Error Handling

If error reporting is turned on the device encodes errors into CAN error frames (see uapi/linux/can/error.h) and sends it using the IN endpoint. The driver updates its error statistics and forwards it.

Although UCAN devices can suppress error frames completely, in Linux the driver is always interested. Hence, the device is always started with the UCAN_MODE_BERR_REPORT set. Filtering those messages for the user space is done by the driver.

5.5.1 Bus OFF

- The device does not recover from bus off automatically.
- Bus OFF is indicated by an error frame (see uapi/linux/can/error.h)
- Bus OFF recovery is started by UCAN_COMMAND_RESTART
- Once Bus OFF recover is completed the device sends an error frame indicating that it is on ERROR-ACTIVE state.
- During Bus OFF no frames are sent by the device.
- During Bus OFF transmission requests from the host are completed immediately with the success bit left unset.

5.6 Example Conversation

- 1) Device is connected to USB
- 2) Host sends command UCAN_COMMAND_RESET, subcmd 0
- 3) Host sends command UCAN_COMMAND_GET, subcmd UCAN_COMMAND_GET_INFO
- 4) Device sends UCAN_IN_DEVICE_INFO
- 5) Host sends command UCAN_OUT_SET_BITTIMING
- 6) Host sends command UCAN_COMMAND_START, subcmd 0, mode UCAN_MODE_BERR_REPORT

HARDWARE DEVICE DRIVERS

Contents:

6.1 Asynchronous Transfer Mode (ATM) Device Drivers

Contents:

6.1.1 ATM cxacru device driver

Firmware is required for this device: <http://accessrunner.sourceforge.net/>

While it is capable of managing/maintaining the ADSL connection without the module loaded, the device will sometimes stop responding after unloading the driver and it is necessary to unplug/remove power to the device to fix this.

Note: support for cxacru-cf.bin has been removed. It was not loaded correctly so it had no effect on the device configuration. Fixing it could have stopped existing devices working when an invalid configuration is supplied.

There is a script cxacru-cf.py to convert an existing file to the sysfs form.

Detected devices will appear as ATM devices named "cxacru". In /sys/class/atm/ these are directories named cxacruN where N is the device number. A symlink named device points to the USB interface device's directory which contains several sysfs attribute files for retrieving device statistics:

- ads_l_controller_version
- ads_l_headend
- ads_l_headend_environment
 - Information about the remote headend.
- ads_l_config
 - Configuration writing interface.
 - Write parameters in hexadecimal format <index>=<value>, separated by whitespace, e.g.:
"1=0 a=5"
- Up to 7 parameters at a time will be sent and the modem will restart the ADSL connection when any value is set. These are logged for future reference.

- downstream_attenuation (dB)
- downstream_bits_per_frame
- downstream_rate (kbps)
- downstream_snr_margin (dB)
 - Downstream stats.
- upstream_attenuation (dB)
- upstream_bits_per_frame
- upstream_rate (kbps)
- upstream_snr_margin (dB)
- transmitter_power (dBm/Hz)
 - Upstream stats.
- downstream_crc_errors
- downstream_fec_errors
- downstream_hec_errors
- upstream_crc_errors
- upstream_fec_errors
- upstream_hec_errors
 - Error counts.
- line_startable
 - Indicates that ADSL support on the device is/can be enabled, see `adsl_start`.
- line_status
 - "initialising"
 - "down"
 - "attempting to activate"
 - "training"
 - "channel analysis"
 - "exchange"
 - "waiting"
 - "up"

Changes between "down" and "attempting to activate" if there is no signal.

- link_status
 - "not connected"
 - "connected"
 - "lost"

- mac_address
- modulation
 - "" (when not connected)
 - "ANSI T1.413"
 - "ITU-T G.992.1 (G.DMT)"
 - "ITU-T G.992.2 (G.LITE)"
- startup_attempts
 - Count of total attempts to initialise ADSL.

To enable/disable ADSL, the following can be written to the adsl_state file:

- "start"
- "stop"
- "restart" (stops, waits 1.5s, then starts)
- "poll" (used to resume status polling if it was disabled due to failure)

Changes in adsl/line state are reported via kernel log messages:

```
[4942145.150704] ATM dev 0: ADSL state: running
[4942243.663766] ATM dev 0: ADSL line: down
[4942249.665075] ATM dev 0: ADSL line: attempting to activate
[4942253.654954] ATM dev 0: ADSL line: training
[4942255.666387] ATM dev 0: ADSL line: channel analysis
[4942259.656262] ATM dev 0: ADSL line: exchange
[2635357.696901] ATM dev 0: ADSL line: up (8128 kb/s down | 832 kb/s up)
```

6.1.2 FORE Systems PCA-200E/SBA-200E ATM NIC driver

This driver adds support for the FORE Systems 200E-series ATM adapters to the Linux operating system. It is based on the earlier PCA-200E driver written by Uwe Dannowski.

The driver simultaneously supports PCA-200E and SBA-200E adapters on i386, alpha (untested), powerpc, sparc and sparc64 archs.

The intent is to enable the use of different models of FORE adapters at the same time, by hosts that have several bus interfaces (such as PCI+SBUS, or PCI+EISA).

Only PCI and SBUS devices are currently supported by the driver, but support for other bus interfaces such as EISA should not be too hard to add.

Firmware Copyright Notice

Please read the fore200e_firmware_copyright file present in the linux/drivers/atm directory for details and restrictions.

Firmware Updates

The FORE Systems 200E-series driver is shipped with firmware data being uploaded to the ATM adapters at system boot time or at module loading time. The supplied firmware images should work with all adapters.

However, if you encounter problems (the firmware doesn't start or the driver is unable to read the PROM data), you may consider trying another firmware version. Alternative binary firmware images can be found somewhere on the ForeThought CD-ROM supplied with your adapter by FORE Systems.

You can also get the latest firmware images from FORE Systems at https://en.wikipedia.org/wiki/FORE_Systems. Register TACTics Online and go to the 'software updates' pages. The firmware binaries are part of the various ForeThought software distributions.

Notice that different versions of the PCA-200E firmware exist, depending on the endianness of the host architecture. The driver is shipped with both little and big endian PCA firmware images.

Name and location of the new firmware images can be set at kernel configuration time:

1. Copy the new firmware binary files (with .bin, .bin1 or .bin2 suffix) to some directory, such as linux/drivers/atm.
2. Reconfigure your kernel to set the new firmware name and location. Expected pathnames are absolute or relative to the drivers/atm directory.
3. Rebuild and re-install your kernel or your module.

Feedback

Feedback is welcome. Please send success stories/bug reports/patches/improvement/comments/flames to <lizzi@cnam.fr>.

6.1.3 ATM (i)Chip IA Linux Driver Source

READ ME FIRST

Read This Before You Begin!

Description

This is the README file for the Interphase PCI ATM (i)Chip IA Linux driver source release.

The features and limitations of this driver are as follows:

- A single VPI (VPI value of 0) is supported.
- Supports 4K VCs for the server board (with 512K control memory) and 1K VCs for the client board (with 128K control memory).
- UBR, ABR and CBR service categories are supported.
- Only AAL5 is supported.
- Supports setting of PCR on the VCs.
- Multiple adapters in a system are supported.
- All variants of Interphase ATM PCI (i)Chip adapter cards are supported, including x575 (OC3, control memory 128K, 512K and packet memory 128K, 512K and 1M), x525 (UTP25) and x531 (DS3 and E3). See <http://www.ipphase.com/> for details.
- Only x86 platforms are supported.
- SMP is supported.

Before You Start

Installation

1. Installing the adapters in the system

To install the ATM adapters in the system, follow the steps below.

- Login as root.
- Shut down the system and power off the system.
- Install one or more ATM adapters in the system.
- Connect each adapter to a port on an ATM switch. The green 'Link' LED on the front panel of the adapter will be on if the adapter is connected to the switch properly when the system is powered up.
- Power on and boot the system.

2. [Removed]

3. Rebuild kernel with ABR support

[a. and b. removed]

- Reconfigure the kernel, choose the Interphase ia driver through "make menuconfig" or "make xconfig".
- Rebuild the kernel, loadable modules and the atm tools.
- Install the new built kernel and modules and reboot.

4. Load the adapter hardware driver (ia driver) if it is built as a module

- a. Login as root.
- b. Change directory to /lib/modules/<kernel-version>/atm.
- c. Run "insmod suni.o;insmod iphase.o" The yellow 'status' LED on the front panel of the adapter will blink while the driver is loaded in the system.
- d. To verify that the 'ia' driver is loaded successfully, run the following command:

```
cat /proc/atm/devices
```

If the driver is loaded successfully, the output of the command will be similar to the following lines:

Itf	Type	ESI/"MAC"	addr	AAL(TX,err,RX,err,drop) ...		
0	ia	xxxxxxxxxx	0 (0 0 0 0 0)	5 (0 0 0 0 0)		

You can also check the system log file /var/log/messages for messages related to the ATM driver.

5. Ia Driver Configuration

5.1 Configuration of adapter buffers

The (i)Chip boards have 3 different packet RAM size variants: 128K, 512K and 1M. The RAM size decides the number of buffers and buffer size. The default size and number of buffers are set as following:

Total	Rx RAM	Tx RAM	Rx Buf	Tx Buf	Rx buf	Tx buf
RAM size	size	size	size	size	cnt	cnt
128K	64K	64K	10K	10K	6	6
512K	256K	256K	10K	10K	25	25
1M	512K	512K	10K	10K	51	51

These setting should work well in most environments, but can be changed by typing the following command:

```
insmod <IA_DIR>/ia.o IA_RX_BUF=<RX_CNT> IA_RX_BUF_SZ=<RX_SIZE> \
IA_TX_BUF=<TX_CNT> IA_TX_BUF_SZ=<TX_SIZE>
```

Where:

- RX_CNT = number of receive buffers in the range (1-128)
 - RX_SIZE = size of receive buffers in the range (48-64K)
 - TX_CNT = number of transmit buffers in the range (1-128)
 - TX_SIZE = size of transmit buffers in the range (48-64K)
1. Transmit and receive buffer size must be a multiple of 4.
 2. Care should be taken so that the memory required for the transmit and receive buffers is less than or equal to the total adapter packet memory.

5.2 Turn on ia debug trace

When the ia driver is built with the CONFIG_ATM_IA_DEBUG flag, the driver can provide more debug trace if needed. There is a bit mask variable, IADebugFlag,

which controls the output of the traces. You can find the bit map of the IA Debug Flag in iphase.h. The debug trace can be turned on through the insmod command line option, for example, "insmod iphase.o IA Debug Flag=0xffffffff" can turn on all the debug traces together with loading the driver.

6. Ia Driver Test Using ttcp_atm and PVC

For the PVC setup, the test machines can either be connected back-to-back or through a switch. If connected through the switch, the switch must be configured for the PVC(s).

a. For UBR test:

At the test machine intended to receive data, type:

```
ttcp_atm -r -a -s 0.100
```

At the other test machine, type:

```
ttcp_atm -t -a -s 0.100 -n 10000
```

Run "ttcp_atm -h" to display more options of the ttcp_atm tool.

b. For ABR test:

It is the same as the UBR testing, but with an extra command option:

```
-Pabr:max_pcr=<xxx>
```

where:

xxx = the maximum peak cell rate, from 170 - 353207.

This option must be set on both the machines.

c. For CBR test:

It is the same as the UBR testing, but with an extra command option:

```
-Pcbr:max_pcr=<xxx>
```

where:

xxx = the maximum peak cell rate, from 170 - 353207.

This option may only be set on the transmit machine.

Outstanding Issues

Contact Information

Customer Support:

United States:	Telephone:	(214) 654-5555
	Fax:	(214) 654-5500
	E-Mail:	intouch@iphase.com
Europe:	Telephone:	33 (0)1 41 15 44 00
	Fax:	33 (0)1 41 15 12 13

World Wide Web:	http://www.ipphase.com
Anonymous FTP:	ftp.ipphase.com

6.2 Cable Modem Device Drivers

Contents:

6.2.1 SB100 device driver

sb1000 is a module network device driver for the General Instrument (also known as NextLevel) SURFboard1000 internal cable modem board. This is an ISA card which is used by a number of cable TV companies to provide cable modem access. It's a one-way downstream-only cable modem, meaning that your upstream net link is provided by your regular phone modem.

This driver was written by Franco Venturi <fventuri@mediaone.net>. He deserves a great deal of thanks for this wonderful piece of code!

Needed tools

Support for this device is now a part of the standard Linux kernel. The driver source code file is drivers/net/sb1000.c. In addition to this you will need:

1. The "cmconfig" program. This is a utility which supplements "ifconfig" to configure the cable modem and network interface (usually called "cm0");
2. Several PPP scripts which live in /etc/ppp to make connecting via your cable modem easy.

These utilities can be obtained from:

<http://www.jacksonville.net/~fventuri/>

in Franco's original source code distribution .tar.gz file. Support for the sb1000 driver can be found at:

- <http://web.archive.org/web/%2E/http://home.adelphia.net/~siglercm/sb1000.html>
- <http://web.archive.org/web/%2E/http://linuxpower.cx/~cable/>

along with these utilities.

3. The standard isapnp tools. These are necessary to configure your SB1000 card at boot time (or afterwards by hand) since it's a PnP card.

If you don't have these installed as a standard part of your Linux distribution, you can find them at:

<http://www.roestock.demon.co.uk/isapnptools/>

or check your Linux distribution binary CD or their web site. For help with isapnp, pndump, or /etc/isapnp.conf, go to:

<http://www.roestock.demon.co.uk/isapnptools/isapnpfaq.html>

Using the driver

To make the SB1000 card work, follow these steps:

1. Run `make config`, or `make menuconfig`, or `make xconfig`, whichever you prefer, in the top kernel tree directory to set up your kernel configuration. Make sure to say "Y" to "Prompt for development drivers" and to say "M" to the sb1000 driver. Also say "Y" or "M" to all the standard networking questions to get TCP/IP and PPP networking support.
2. **BEFORE** you build the kernel, edit `drivers/net/sb1000.c`. Make sure to redefine the value of `READ_DATA_PORT` to match the I/O address used by `isapnp` to access your PnP cards. This is the value of `READPORT` in `/etc/isapnp.conf` or given by the output of `pnpdump`.
3. Build and install the kernel and modules as usual.
4. Boot your new kernel following the usual procedures.
5. Set up to configure the new SB1000 PnP card by capturing the output of "pnpdump" to a file and editing this file to set the correct I/O ports, IRQ, and DMA settings for all your PnP cards. Make sure none of the settings conflict with one another. Then test this configuration by running the "isapnp" command with your new config file as the input. Check for errors and fix as necessary. (As an aside, I use I/O ports 0x110 and 0x310 and IRQ 11 for my SB1000 card and these work well for me. YMMV.) Then save the finished config file as `/etc/isapnp.conf` for proper configuration on subsequent reboots.
6. Download the original file `sb1000-1.1.2.tar.gz` from Franco's site or one of the others referenced above. As root, unpack it into a temporary directory and do a `make cmconfig` and then `install -c cmconfig /usr/local/sbin`. Don't do `make install` because it expects to find all the utilities built and ready for installation, not just `cmconfig`.
7. As root, copy all the files under the `ppp/` subdirectory in Franco's tar file into `/etc/ppp`, being careful not to overwrite any files that are already in there. Then modify `ppp@gi-on` to set the correct login name, phone number, and frequency for the cable modem. Also edit `pap-secrets` to specify your login name and password and any site-specific information you need.
8. Be sure to modify `/etc/ppp/firewall` to use `ipchains` instead of the older `ipfwadm` commands from the 2.0.x kernels. There's a neat utility to convert `ipfwadm` commands to `ipchains` commands:

<http://users.dhp.com/~whisper/ipfwadm2ipchains/>

You may also wish to modify the `firewall` script to implement a different firewalling scheme.

9. Start the PPP connection via the script `/etc/ppp/ppp@gi-on`. You must be root to do this. It's better to use a utility like `sudo` to execute frequently used commands like this with root permissions if possible. If you connect successfully the cable modem interface will come up and you'll see a driver message like this at the console:

```
cm0: sb1000 at (0x110,0x310), csn 1, S/N 0x2a0d16d8, IRQ 11.
sb1000.c:v1.1.2 6/01/98 (fventuri@mediaone.net)
```

The "ifconfig" command should show two new interfaces, `ppp0` and `cm0`.

The command "`cmconfig cm0`" will give you information about the cable modem interface.

10. Try pinging a site via `ping -c 5 www.yahoo.com`, for example. You should see packets received.

11. If you can't get site names (like www.yahoo.com) to resolve into IP addresses (like 204.71.200.67), be sure your /etc/resolv.conf file has no syntax errors and has the right nameserver IP addresses in it. If this doesn't help, try something like ping -c 5 204.71.200.67 to see if the networking is running but the DNS resolution is where the problem lies.
12. If you still have problems, go to the support web sites mentioned above and read the information and documentation there.

Common problems

1. Packets go out on the ppp0 interface but don't come back on the cm0 interface. It looks like I'm connected but I can't even ping any numerical IP addresses. (This happens predominantly on Debian systems due to a default boot-time configuration script.)

Solution

As root echo 0 > /proc/sys/net/ipv4/conf/cm0/rp_filter so it can share the same IP address as the ppp0 interface. Note that this command should probably be added to the /etc/ppp/cablemodem script *right*between* the "/sbin/ifconfig" and "/sbin/cmconfig" commands. You may need to do this to /proc/sys/net/ipv4/conf/ppp0/rp_filter as well. If you do this to /proc/sys/net/ipv4/conf/default/rp_filter on each reboot (in rc.local or some such) then any interfaces can share the same IP addresses.

2. I get "unresolved symbol" error messages on executing insmod sb1000.o.

Solution

You probably have a non-matching kernel source tree and /usr/include/linux and /usr/include/asm header files. Make sure you install the correct versions of the header files in these two directories. Then rebuild and reinstall the kernel.

3. When isapnp runs it reports an error, and my SB1000 card isn't working.

Solution

There's a problem with later versions of isapnp using the "(CHECK)" option in the lines that allocate the two I/O addresses for the SB1000 card. This first popped up on RH 6.0. Delete "(CHECK)" for the SB1000 I/O addresses. Make sure they don't conflict with any other pieces of hardware first! Then rerun isapnp and go from there.

4. I can't execute the /etc/ppp/ppp@gi-on file.

Solution

As root do chmod ug+x /etc/ppp/ppp@gi-on.

5. The firewall script isn't working (with 2.2.x and higher kernels).

Solution

Use the ipfwadm2ipchains script referenced above to convert the /etc/ppp/firewall script from the deprecated ipfwadm commands to ipchains.

6. I'm getting tons of firewall deny messages in the /var/kern.log, /var/messages, and/or /var/syslog files, and they're filling up my /var partition!!!

Solution

First, tell your ISP that you're receiving DoS (Denial of Service) and/or portscanning (UDP connection attempts) attacks! Look over the deny messages to figure out what the attack is and where it's coming from. Next, edit /etc/ppp/cablemodem and make sure the ",no-broadcast" option is turned on to the "cmconfig" command (uncomment that line). If you're

not receiving these denied packets on your broadcast interface (IP address xxx.yyy.zzz.255 typically), then someone is attacking your machine in particular. Be careful out there....

7. Everything seems to work fine but my computer locks up after a while (and typically during a lengthy download through the cable modem)!

Solution

You may need to add a short delay in the driver to 'slow down' the SURFboard because your PC might not be able to keep up with the transfer rate of the SB1000. To do this, it's probably best to download Franco's sb1000-1.1.2.tar.gz archive and build and install sb1000.o manually. You'll want to edit the 'Makefile' and look for the 'SB1000_DELAY' define. Uncomment those 'CFLAGS' lines (and comment out the default ones) and try setting the delay to something like 60 microseconds with: '-DSB1000_DELAY=60'. Then do `make` and as root `make install` and try it out. If it still doesn't work or you like playing with the driver, you may try other numbers. Remember though that the higher the delay, the slower the driver (which slows down the rest of the PC too when it is actively used). Thanks to Ed Daiga for this tip!

Credits

This README came from Franco Venturi's original README file which is still supplied with his driver .tar.gz archive. I and all other sb1000 users owe Franco a tremendous "Thank you!" Additional thanks goes to Carl Patten and Ralph Bonnell who are now managing the Linux SB1000 web site, and to the SB1000 users who reported and helped debug the common problems listed above.

Clemmitt Sigler csigler@vt.edu

6.3 Controller Area Network (CAN) Device Drivers

Device drivers for CAN devices.

Contents:

6.3.1 can327: ELM327 driver for Linux SocketCAN

Authors

Max Staudt <max@enpas.org>

Motivation

This driver aims to lower the initial cost for hackers interested in working with CAN buses. CAN adapters are expensive, few, and far between. ELM327 interfaces are cheap and plentiful. Let's use ELM327s as CAN adapters.

Introduction

This driver is an effort to turn abundant ELM327 based OBD interfaces into full fledged (as far as possible) CAN interfaces.

Since the ELM327 was never meant to be a stand alone CAN controller, the driver has to switch between its modes as quickly as possible in order to fake full-duplex operation.

As such, can327 is a best effort driver. However, this is more than enough to implement simple request-response protocols (such as OBD II), and to monitor broadcast messages on a bus (such as in a vehicle).

Most ELM327s come as nondescript serial devices, attached via USB or Bluetooth. The driver cannot recognize them by itself, and as such it is up to the user to attach it in form of a TTY line discipline (similar to PPP, SLIP, slcan, ...).

This driver is meant for ELM327 versions 1.4b and up, see below for known limitations in older controllers and clones.

Data sheet

The official data sheets can be found at ELM electronics' home page:

<https://www.elmelectronics.com/>

How to attach the line discipline

Every ELM327 chip is factory programmed to operate at a serial setting of 38400 baud/s, 8 data bits, no parity, 1 stopbit.

If you have kept this default configuration, the line discipline can be attached on a command prompt as follows:

```
sudo ldattach \
    --debug \
    --speed 38400 \
    --eightbits \
    --noparity \
    --onestopbit \
    --iflag -ICRNL,INLCR,-IXOFF \
    30 \
    /dev/ttyUSB0
```

To change the ELM327's serial settings, please refer to its data sheet. This needs to be done before attaching the line discipline.

Once the ldisc is attached, the CAN interface starts out unconfigured. Set the speed before starting it:

```
# The interface needs to be down to change parameters
sudo ip link set can0 down
sudo ip link set can0 type can bitrate 500000
sudo ip link set can0 up
```

500000 bit/s is a common rate for OBD-II diagnostics. If you're connecting straight to a car's OBD port, this is the speed that most cars (but not all!) expect.

After this, you can set out as usual with candump, cansniffer, etc.

How to check the controller version

Use a terminal program to attach to the controller.

After issuing the "AT WS" command, the controller will respond with its version:

```
>AT WS
```

```
ELM327 v1.4b
```

```
>
```

Note that clones may claim to be any version they like. It is not indicative of their actual feature set.

Communication example

This is a short and incomplete introduction on how to talk to an ELM327. It is here to guide understanding of the controller's and the driver's limitation (listed below) as well as manual testing.

The ELM327 has two modes:

- Command mode
- Reception mode

In command mode, it expects one command per line, terminated by CR. By default, the prompt is a ">", after which a command can be entered:

```
>ATE1
```

```
OK
```

```
>
```

The init script in the driver switches off several configuration options that are only meaningful in the original OBD scenario the chip is meant for, and are actually a hindrance for can327.

When a command is not recognized, such as by an older version of the ELM327, a question mark is printed as a response instead of OK:

```
>ATUNKNOWN
```

```
?
```

```
>
```

At present, can327 does not evaluate this response. See the section below on known limitations for details.

When a CAN frame is to be sent, the target address is configured, after which the frame is sent as a command that consists of the data's hex dump:

```
>ATSH123
OK
>DEADBEEF12345678
OK
>
```

The above interaction sends the SFF frame "DE AD BE EF 12 34 56 78" with (11 bit) CAN ID 0x123. For this to function, the controller must be configured for SFF sending mode (using "AT PB", see code or datasheet).

Once a frame has been sent and wait-for-reply mode is on (ATR1, configured on `listen-only=off`), or when the reply timeout expires and the driver sets the controller into monitoring mode (ATMA), the ELM327 will send one line for each received CAN frame, consisting of CAN ID, DLC, and data:

```
123 8 DEADBEEF12345678
```

For EFF (29 bit) CAN frames, the address format is slightly different, which can327 uses to tell the two apart:

```
12 34 56 78 8 DEADBEEF12345678
```

The ELM327 will receive both SFF and EFF frames - the current CAN config (ATPB) does not matter.

If the ELM327's internal UART sending buffer runs full, it will abort the monitoring mode, print "BUFFER FULL" and drop back into command mode. Note that in this case, unlike with other error messages, the error message may appear on the same line as the last (usually incomplete) data frame:

```
12 34 56 78 8 DEADBEEF123 BUFFER FULL
```

Known limitations of the controller

- Clone devices ("v1.5" and others)

Sending RTR frames is not supported and will be dropped silently.

Receiving RTR with DLC 8 will appear to be a regular frame with the last received frame's DLC and payload.

"AT CSM" (CAN Silent Monitoring, i.e. don't send CAN ACKs) is not supported, and is hard coded to ON. Thus, frames are not ACKed while listening: "AT MA" (Monitor All) will always be "silent". However, immediately after sending a frame, the ELM327 will be in "receive reply" mode, in which it *does* ACK any received frames. Once the bus goes silent, or an error occurs (such as BUFFER FULL), or the receive reply timeout runs out, the ELM327 will end reply reception mode on its own and can327 will fall back to "AT MA" in order to keep monitoring the bus.

Other limitations may apply, depending on the clone and the quality of its firmware.

- All versions

No full duplex operation is supported. The driver will switch between input/output mode as quickly as possible.

The length of outgoing RTR frames cannot be set. In fact, some clones (tested with one identifying as "v1.5") are unable to send RTR frames at all.

We don't have a way to get real-time notifications on CAN errors. While there is a command (AT CS) to retrieve some basic stats, we don't poll it as it would force us to interrupt reception mode.

- Versions prior to 1.4b

These versions do not send CAN ACKs when in monitoring mode (AT MA). However, they do send ACKs while waiting for a reply immediately after sending a frame. The driver maximizes this time to make the controller as useful as possible.

Starting with version 1.4b, the ELM327 supports the "AT CSM" command, and the "listen-only" CAN option will take effect.

- Versions prior to 1.4

These chips do not support the "AT PB" command, and thus cannot change bitrate or SFF/EFF mode on-the-fly. This will have to be programmed by the user before attaching the line discipline. See the data sheet for details.

- Versions prior to 1.3

These chips cannot be used at all with can327. They do not support the "AT D1" command, which is necessary to avoid parsing conflicts on incoming data, as well as distinction of RTR frame lengths.

Specifically, this allows for easy distinction of SFF and EFF frames, and to check whether frames are complete. While it is possible to deduce the type and length from the length of the line the ELM327 sends us, this method fails when the ELM327's UART output buffer overruns. It may abort sending in the middle of the line, which will then be mistaken for something else.

Known limitations of the driver

- No 8/7 timing.

ELM327 can only set CAN bitrates that are of the form 500000/n, where n is an integer divisor. However there is an exception: With a separate flag, it may set the speed to be 8/7 of the speed indicated by the divisor. This mode is not currently implemented.

- No evaluation of command responses.

The ELM327 will reply with OK when a command is understood, and with ? when it is not. The driver does not currently check this, and simply assumes that the chip understands every command. The driver is built such that functionality degrades gracefully nevertheless. See the section on known limitations of the controller.

- No use of hardware CAN ID filtering

An ELM327's UART sending buffer will easily overflow on heavy CAN bus load, resulting in the "BUFFER FULL" message. Using the hardware filters available through "AT CF xxx" and "AT CM xxx" would be helpful here, however SocketCAN does not currently provide a facility to make use of such hardware features.

Rationale behind the chosen configuration

AT E1

Echo on

We need this to be able to get a prompt reliably.

AT S1

Spaces on

We need this to distinguish 11/29 bit CAN addresses received.

Note: We can usually do this using the line length (odd/even), but this fails if the line is not transmitted fully to the host (BUFFER FULL).

AT D1

DLC on

We need this to tell the "length" of RTR frames.

A note on CAN bus termination

Your adapter may have resistors soldered in which are meant to terminate the bus. This is correct when it is plugged into a OBD-II socket, but not helpful when trying to tap into the middle of an existing CAN bus.

If communications don't work with the adapter connected, check for the termination resistors on its PCB and try removing them.

6.3.2 CTU CAN FD Driver

Author: Martin Jerabek <martin.jerabek01@gmail.com>

About CTU CAN FD IP Core

CTU CAN FD is an open source soft core written in VHDL. It originated in 2015 as Ondrej Ille's project at the Department of Measurement of FEE at CTU.

The SocketCAN driver for Xilinx Zynq SoC based MicroZed board [Vivado integration](#) and Intel Cyclone V 5CSEMA4U23C6 based DE0-Nano-SoC Terasic board [QSys integration](#) has been developed as well as support for [PCIe integration](#) of the core.

In the case of Zynq, the core is connected via the APB system bus, which does not have enumeration support, and the device must be specified in Device Tree. This kind of devices is called platform device in the kernel and is handled by a platform device driver.

The basic functional model of the CTU CAN FD peripheral has been accepted into QEMU mainline. See QEMU [CAN emulation support](#) for CAN FD buses, host connection and CTU CAN FD core emulation. The development version of emulation support can be cloned from [ctu-canfd](#) branch of QEMU local development [repository](#).

About SocketCAN

SocketCAN is a standard common interface for CAN devices in the Linux kernel. As the name suggests, the bus is accessed via sockets, similarly to common network devices. The reasoning behind this is in depth described in [Linux SocketCAN](#). In short, it offers a natural way to implement and work with higher layer protocols over CAN, in the same way as, e.g., UDP/IP over Ethernet.

Device probe

Before going into detail about the structure of a CAN bus device driver, let's reiterate how the kernel gets to know about the device at all. Some buses, like PCI or PCIe, support device enumeration. That is, when the system boots, it discovers all the devices on the bus and reads their configuration. The kernel identifies the device via its vendor ID and device ID, and if there is a driver registered for this identifier combination, its probe method is invoked to populate the driver's instance for the given hardware. A similar situation goes with USB, only it allows for device hot-plug.

The situation is different for peripherals which are directly embedded in the SoC and connected to an internal system bus (AXI, APB, Avalon, and others). These buses do not support enumeration, and thus the kernel has to learn about the devices from elsewhere. This is exactly what the Device Tree was made for.

Device tree

An entry in device tree states that a device exists in the system, how it is reachable (on which bus it resides) and its configuration – registers address, interrupts and so on. An example of such a device tree is given in .

```
/ {
    /* ... */
    amba: amba {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "simple-bus";

        CTU_CAN_FD_0: CTU_CAN_FD@43c30000 {
            compatible = "ctu,ctucanfd";
            interrupt-parent = <&intc>;
            interrupts = <0 30 4>;
            clocks = <&clkc 15>;
            reg = <0x43c30000 0x10000>;
        };
    };
};
```

Driver structure

The driver can be divided into two parts - platform-dependent device discovery and set up, and platform-independent CAN network device implementation.

Platform device driver

In the case of Zynq, the core is connected via the AXI system bus, which does not have enumeration support, and the device must be specified in Device Tree. This kind of devices is called *platform device* in the kernel and is handled by a *platform device driver*¹.

A platform device driver provides the following things:

- A *probe* function
- A *remove* function
- A table of *compatible* devices that the driver can handle

The *probe* function is called exactly once when the device appears (or the driver is loaded, whichever happens later). If there are more devices handled by the same driver, the *probe* function is called for each one of them. Its role is to allocate and initialize resources required for handling the device, as well as set up low-level functions for the platform-independent layer, e.g., *read_reg* and *write_reg*. After that, the driver registers the device to a higher layer, in our case as a *network device*.

The *remove* function is called when the device disappears, or the driver is about to be unloaded. It serves to free the resources allocated in *probe* and to unregister the device from higher layers.

Finally, the table of *compatible* devices states which devices the driver can handle. The Device Tree entry *compatible* is matched against the tables of all *platform drivers*.

```
/* Match table for OF platform binding */
static const struct of_device_id ctucan_of_match[] = {
    { .compatible = "ctu,canfd-2", },
    { .compatible = "ctu,ctucanfd", },
    { /* end of list */ },
};

MODULE_DEVICE_TABLE(of, ctucan_of_match);

static int ctucan_probe(struct platform_device *pdev);
static int ctucan_remove(struct platform_device *pdev);

static struct platform_driver ctucanfd_driver = {
    .probe  = ctucan_probe,
    .remove = ctucan_remove,
    .driver = {
        .name = DRIVER_NAME,
        .of_match_table = ctucan_of_match,
    },
};
module_platform_driver(ctucanfd_driver);
```

¹ Other buses have their own specific driver interface to set up the device.

Network device driver

Each network device must support at least these operations:

- Bring the device up: `ndo_open`
- Bring the device down: `ndo_close`
- Submit TX frames to the device: `ndo_start_xmit`
- Signal TX completion and errors to the network subsystem: ISR
- Submit RX frames to the network subsystem: ISR and NAPI

There are two possible event sources: the device and the network subsystem. Device events are usually signaled via an interrupt, handled in an Interrupt Service Routine (ISR). Handlers for the events originating in the network subsystem are then specified in `struct net_device_ops`.

When the device is brought up, e.g., by calling `ip link set can0 up`, the driver's function `ndo_open` is called. It should validate the interface configuration and configure and enable the device. The analogous opposite is `ndo_close`, called when the device is being brought down, be it explicitly or implicitly.

When the system should transmit a frame, it does so by calling `ndo_start_xmit`, which enqueues the frame into the device. If the device HW queue (FIFO, mailboxes or whatever the implementation is) becomes full, the `ndo_start_xmit` implementation informs the network subsystem that it should stop the TX queue (via `netif_stop_queue`). It is then re-enabled later in ISR when the device has some space available again and is able to enqueue another frame.

All the device events are handled in ISR, namely:

1. **TX completion.** When the device successfully finishes transmitting a frame, the frame is echoed locally. On error, an informative error frame² is sent to the network subsystem instead. In both cases, the software TX queue is resumed so that more frames may be sent.
2. **Error condition.** If something goes wrong (e.g., the device goes bus-off or RX overrun happens), error counters are updated, and informative error frames are enqueued to SW RX queue.
3. **RX buffer not empty.** In this case, read the RX frames and enqueue them to SW RX queue. Usually NAPI is used as a middle layer (see).

NAPI

The frequency of incoming frames can be high and the overhead to invoke the interrupt service routine for each frame can cause significant system load. There are multiple mechanisms in the Linux kernel to deal with this situation. They evolved over the years of Linux kernel development and enhancements. For network devices, the current standard is NAPI – *the New API*. It is similar to classical top-half/bottom-half interrupt handling in that it only acknowledges the interrupt in the ISR and signals that the rest of the processing should be done in softirq context. On top of that, it offers the possibility to *poll* for new frames for a while. This has a potential to avoid the costly round of enabling interrupts, handling an incoming IRQ in ISR, re-enabling the softirq and switching context back to softirq.

² Not to be mistaken with CAN Error Frame. This is a `can_frame` with `CAN_ERR_FLAG` set and some error info in its `data` field.

See [Documentation/networking/napi.rst](#) for more information.

Integrating the core to Xilinx Zynq

The core interfaces a simple subset of the Avalon (search for Intel **Avalon Interface Specifications**) bus as it was originally used on Alterra FPGA chips, yet Xilinx natively interfaces with AXI (search for ARM **AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite, ACE and ACE-Lite**). The most obvious solution would be to use an Avalon/AXI bridge or implement some simple conversion entity. However, the core's interface is half-duplex with no handshake signaling, whereas AXI is full duplex with two-way signaling. Moreover, even AXI-Lite slave interface is quite resource-intensive, and the flexibility and speed of AXI are not required for a CAN core.

Thus a much simpler bus was chosen – APB (Advanced Peripheral Bus) (search for ARM **AMBA APB Protocol Specification**). APB-AXI bridge is directly available in Xilinx Vivado, and the interface adaptor entity is just a few simple combinatorial assignments.

Finally, to be able to include the core in a block diagram as a custom IP, the core, together with the APB interface, has been packaged as a Vivado component.

CTU CAN FD Driver design

The general structure of a CAN device driver has already been examined in . The next paragraphs provide a more detailed description of the CTU CAN FD core driver in particular.

Low-level driver

The core is not intended to be used solely with SocketCAN, and thus it is desirable to have an OS-independent low-level driver. This low-level driver can then be used in implementations of OS driver or directly either on bare metal or in a user-space application. Another advantage is that if the hardware slightly changes, only the low-level driver needs to be modified.

The code³ is in part automatically generated and in part written manually by the core author, with contributions of the thesis' author. The low-level driver supports operations such as: set bit timing, set controller mode, enable/disable, read RX frame, write TX frame, and so on.

Configuring bit timing

On CAN, each bit is divided into four segments: SYNC, PROP, PHASE1, and PHASE2. Their duration is expressed in multiples of a Time Quantum (details in [CAN Specification, Version 2.0](#), chapter 8). When configuring bitrate, the durations of all the segments (and time quantum) must be computed from the bitrate and Sample Point. This is performed independently for both the Nominal bitrate and Data bitrate for CAN FD.

SocketCAN is fairly flexible and offers either highly customized configuration by setting all the segment durations manually, or a convenient configuration by setting just the bitrate and sample point (and even that is chosen automatically per Bosch recommendation if not specified). However, each CAN controller may have different base clock frequency and different width of segment duration registers. The algorithm thus needs the minimum and maximum values for

³ Available in CTU CAN FD repository https://gitlab.fel.cvut.cz/canbus/ctucanfd_ip_core

the durations (and clock prescaler) and tries to optimize the numbers to fit both the constraints and the requested parameters.

```
struct can_bittiming_const {
    char name[16];          /* Name of the CAN controller hardware */
    __u32 tseg1_min;       /* Time segment 1 = prop_seg + phase_segl */
    __u32 tseg1_max;
    __u32 tseg2_min;       /* Time segment 2 = phase_seg2 */
    __u32 tseg2_max;
    __u32 sjw_max;         /* Synchronisation jump width */
    __u32 brp_min;         /* Bit-rate prescaler */
    __u32 brp_max;
    __u32 brp_inc;
};
```

[lst:can_bittiming_const]

A curious reader will notice that the durations of the segments PROP_SEG and PHASE_SEG1 are not determined separately but rather combined and then, by default, the resulting TSEG1 is evenly divided between PROP_SEG and PHASE_SEG1. In practice, this has virtually no consequences as the sample point is between PHASE_SEG1 and PHASE_SEG2. In CTU CAN FD, however, the duration registers PROP and PH1 have different widths (6 and 7 bits, respectively), so the auto-computed values might overflow the shorter register and must thus be redistributed among the two⁴.

Handling RX

Frame reception is handled in NAPI queue, which is enabled from ISR when the RXNE (RX FIFO Not Empty) bit is set. Frames are read one by one until either no frame is left in the RX FIFO or the maximum work quota has been reached for the NAPI poll run (see). Each frame is then passed to the network interface RX queue.

An incoming frame may be either a CAN 2.0 frame or a CAN FD frame. The way to distinguish between these two in the kernel is to allocate either `struct can_frame` or `struct canfd_frame`, the two having different sizes. In the controller, the information about the frame type is stored in the first word of RX FIFO.

This brings us a chicken-egg problem: we want to allocate the `skb` for the frame, and only if it succeeds, fetch the frame from FIFO; otherwise keep it there for later. But to be able to allocate the correct `skb`, we have to fetch the first word of FIFO. There are several possible solutions:

1. Read the word, then allocate. If it fails, discard the rest of the frame. When the system is low on memory, the situation is bad anyway.
2. Always allocate `skb` big enough for an FD frame beforehand. Then tweak the `skb` internals to look like it has been allocated for the smaller CAN 2.0 frame.
3. Add option to peek into the FIFO instead of consuming the word.
4. If the allocation fails, store the read word into driver's data. On the next try, use the stored word instead of reading it again.

⁴ As is done in the low-level driver functions `ctucan_hw_set_nom_bittiming` and `ctucan_hw_set_data_bittiming`.

Option 1 is simple enough, but not very satisfying if we could do better. Option 2 is not acceptable, as it would require modifying the private state of an integral kernel structure. The slightly higher memory consumption is just a virtual cherry on top of the “cake”. Option 3 requires non-trivial HW changes and is not ideal from the HW point of view.

Option 4 seems like a good compromise, with its disadvantage being that a partial frame may stay in the FIFO for a prolonged time. Nonetheless, there may be just one owner of the RX FIFO, and thus no one else should see the partial frame (disregarding some exotic debugging scenarios). Besides, the driver resets the core on its initialization, so the partial frame cannot be “adopted” either. In the end, option 4 was selected⁵.

Timestamping RX frames

The CTU CAN FD core reports the exact timestamp when the frame has been received. The timestamp is by default captured at the sample point of the last bit of EOF but is configurable to be captured at the SOF bit. The timestamp source is external to the core and may be up to 64 bits wide. At the time of writing, passing the timestamp from kernel to userspace is not yet implemented, but is planned in the future.

Handling TX

The CTU CAN FD core has 4 independent TX buffers, each with its own state and priority. When the core wants to transmit, a TX buffer in Ready state with the highest priority is selected.

The priorities are 3bit numbers in register TX_PRIORITY (nibble-aligned). This should be flexible enough for most use cases. SocketCAN, however, supports only one FIFO queue for outgoing frames⁶. The buffer priorities may be used to simulate the FIFO behavior by assigning each buffer a distinct priority and *rotating* the priorities after a frame transmission is completed.

In addition to priority rotation, the SW must maintain head and tail pointers into the FIFO formed by the TX buffers to be able to determine which buffer should be used for next frame (`txb_head`) and which should be the first completed one (`txb_tail`). The actual buffer indices are (obviously) modulo 4 (number of TX buffers), but the pointers must be at least one bit wider to be able to distinguish between FIFO full and FIFO empty – in this situation, $txb_head \equiv txb_tail \pmod{4}$. An example of how the FIFO is maintained, together with priority rotation, is depicted in

TXB#	0	1	2	3
Seq	A	B	C	
Prio	7	6	5	4
		T		H

⁵ At the time of writing this thesis, option 1 is still being used and the modification is queued in gitlab issue #222

⁶ Strictly speaking, multiple CAN TX queues are supported since v4.19 can: enable multi-queue for SocketCAN devices but no mainline driver is using them yet.

TXB#	0	1	2	3
Seq		B	C	
Prio	4	7	6	5
		T		H

TXB#	0	1	2	3	0'
Seq	E	B	C	D	
Prio	4	7	6	5	
		T			H

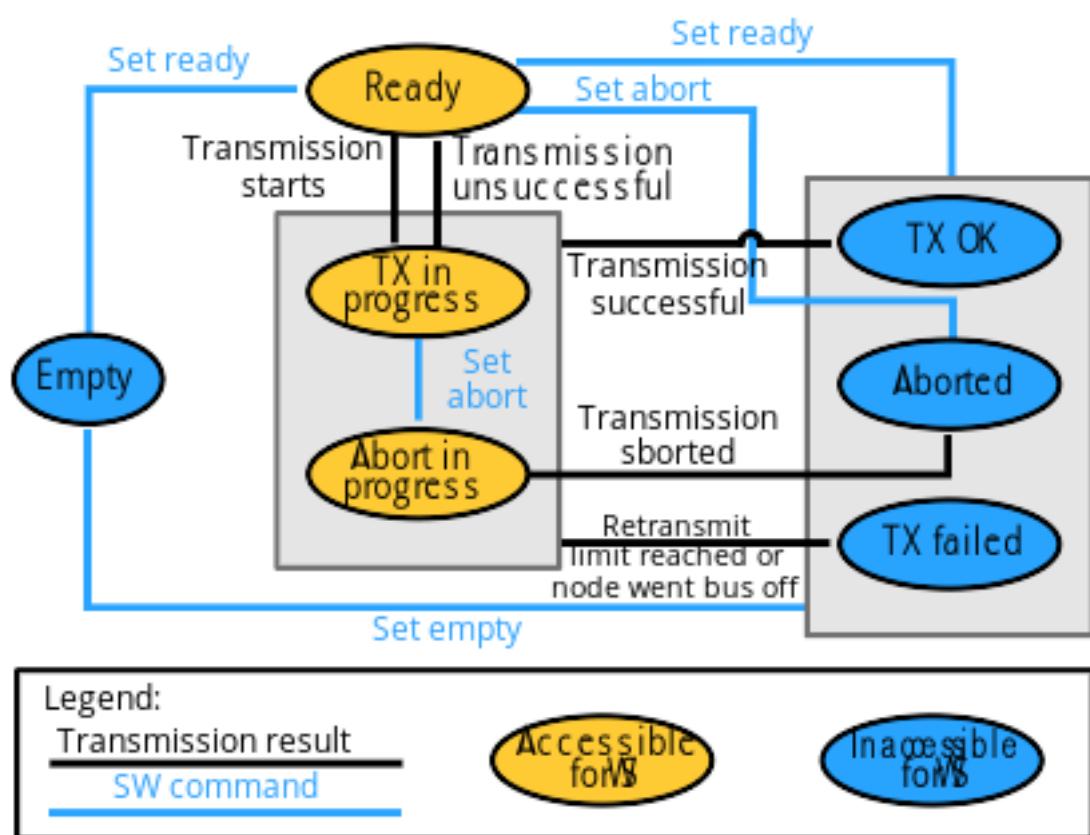


Fig. 1: TX Buffer states with possible transitions

Timestamping TX frames

When submitting a frame to a TX buffer, one may specify the timestamp at which the frame should be transmitted. The frame transmission may start later, but not sooner. Note that the timestamp does not participate in buffer prioritization – that is decided solely by the mechanism described above.

Support for time-based packet transmission was recently merged to Linux v4.19 [Time-based packet transmission](#), but it remains yet to be researched whether this functionality will be practical for CAN.

Also similarly to retrieving the timestamp of RX frames, the core supports retrieving the timestamp of TX frames – that is the time when the frame was successfully delivered. The particulars are very similar to timestamping RX frames and are described in .

Handling RX buffer overrun

When a received frame does no more fit into the hardware RX FIFO in its entirety, RX FIFO overrun flag (STATUS[DOR]) is set and Data Overrun Interrupt (DOI) is triggered. When servicing the interrupt, care must be taken first to clear the DOR flag (via COMMAND[CDO]) and after that clear the DOI interrupt flag. Otherwise, the interrupt would be immediately⁷ rearmed.

Note: During development, it was discussed whether the internal HW pipelining cannot disrupt this clear sequence and whether an additional dummy cycle is necessary between clearing the flag and the interrupt. On the Avalon interface, it indeed proved to be the case, but APB being safe because it uses 2-cycle transactions. Essentially, the DOR flag would be cleared, but DOI register's Preset input would still be high the cycle when the DOI clear request would also be applied (by setting the register's Reset input high). As Set had higher priority than Reset, the DOI flag would not be reset. This has been already fixed by swapping the Set/Reset priority (see issue #187).

Reporting Error Passive and Bus Off conditions

It may be desirable to report when the node reaches *Error Passive*, *Error Warning*, and *Bus Off* conditions. The driver is notified about error state change by an interrupt (EPI, EWLI), and then proceeds to determine the core's error state by reading its error counters.

There is, however, a slight race condition here – there is a delay between the time when the state transition occurs (and the interrupt is triggered) and when the error counters are read. When EPI is received, the node may be either *Error Passive* or *Bus Off*. If the node goes *Bus Off*, it obviously remains in the state until it is reset. Otherwise, the node is *or was Error Passive*. However, it may happen that the read state is *Error Warning* or even *Error Active*. It may be unclear whether and what exactly to report in that case, but I personally entertain the idea that the past error condition should still be reported. Similarly, when EWLI is received but the state is later detected to be *Error Passive*, *Error Passive* should be reported.

⁷ Or rather in the next clock cycle

CTU CAN FD Driver Sources Reference

```
int ctucan_probe_common(struct device *dev, void __iomem *addr, int irq, unsigned int
                        ntxbufs, unsigned long can_clk_rate, int pm_enable_call, void
                        (*set_drvdata_fnc)(struct device *dev, struct net_device *ndev))
```

Device type independent registration call

Parameters

struct device *dev

Handle to the generic device structure

void __iomem *addr

Base address of CTU CAN FD core address

int irq

Interrupt number

unsigned int ntxbufs

Number of implemented Tx buffers

unsigned long can_clk_rate

Clock rate, if 0 then clock are taken from device node

int pm_enable_call

Whether pm_runtime_enable should be called

void (*set_drvdata_fnc)(struct device *dev, struct net_device *ndev)

Function to set network driver data for physical device

Description

This function does all the memory allocation and registration for the CAN device.

Return

0 on success and failure value on error

const char *ctucan_state_to_str(enum can_state state)

Converts CAN controller state code to corresponding text

Parameters

enum can_state state

CAN controller state code

Return

Pointer to string representation of the error state

int ctucan_reset(struct net_device *ndev)

Issues software reset request to CTU CAN FD

Parameters

struct net_device *ndev

Pointer to net_device structure

Return

0 for success, -ETIMEDOUT if CAN controller does not leave reset

`int ctucan_set_btr(struct net_device *ndev, struct can_bittiming *bt, bool nominal)`

Sets CAN bus bit timing in CTU CAN FD

Parameters

`struct net_device *ndev`

Pointer to net_device structure

`struct can_bittiming *bt`

Pointer to Bit timing structure

`bool nominal`

True - Nominal bit timing, False - Data bit timing

Return

0 - OK, -EPERM if controller is enabled

`int ctucan_set_bittiming(struct net_device *ndev)`

CAN set nominal bit timing routine

Parameters

`struct net_device *ndev`

Pointer to net_device structure

Return

0 on success, -EPERM on error

`int ctucan_set_data_bittiming(struct net_device *ndev)`

CAN set data bit timing routine

Parameters

`struct net_device *ndev`

Pointer to net_device structure

Return

0 on success, -EPERM on error

`int ctucan_set_secondary_sample_point(struct net_device *ndev)`

Sets secondary sample point in CTU CAN FD

Parameters

`struct net_device *ndev`

Pointer to net_device structure

Return

0 on success, -EPERM if controller is enabled

`void ctucan_set_mode(struct ctucan_priv *priv, const struct can_ctrlmode *mode)`

Sets CTU CAN FDs mode

Parameters

`struct ctucan_priv *priv`

Pointer to private data

```
const struct can_ctrlmode *mode
    Pointer to controller modes to be set

int ctucan_chip_start(struct net_device *ndev)
    This routine starts the driver
```

Parameters

```
struct net_device *ndev
    Pointer to net_device structure
```

Description

Routine expects that chip is in reset state. It setups initial Tx buffers for FIFO priorities, sets bittiming, enables interrupts, switches core to operational mode and changes controller state to CAN_STATE_STOPPED.

Return

0 on success and failure value on error

```
int ctucan_do_set_mode(struct net_device *ndev, enum can_mode mode)
    Sets mode of the driver
```

Parameters

```
struct net_device *ndev
    Pointer to net_device structure
```

```
enum can_mode mode
    Tells the mode of the driver
```

Description

This check the drivers state and calls the corresponding modes to set.

Return

0 on success and failure value on error

```
enum ctucan_txtb_status ctucan_get_tx_status(struct ctucan_priv *priv, u8 buf)
    Gets status of TXT buffer
```

Parameters

```
struct ctucan_priv *priv
    Pointer to private data
```

```
u8 buf
    Buffer index (0-based)
```

Return

Status of TXT buffer

```
bool ctucan_is_txt_buf_writable(struct ctucan_priv *priv, u8 buf)
    Checks if frame can be inserted to TXT Buffer
```

Parameters

```
struct ctucan_priv *priv
    Pointer to private data
```

u8 buf

Buffer index (0-based)

Return

True - Frame can be inserted to TXT Buffer, False - If attempted, frame will not be inserted to TXT Buffer

bool ctucan_insert_frame(struct ctucan_priv *priv, const struct canfd_frame *cf, u8 buf, bool isfdf)

Inserts frame to TXT buffer

Parameters

struct ctucan_priv *priv

Pointer to private data

const struct canfd_frame *cf

Pointer to CAN frame to be inserted

u8 buf

TXT Buffer index to which frame is inserted (0-based)

bool isfdf

True - CAN FD Frame, False - CAN 2.0 Frame

Return

True - Frame inserted successfully

False - Frame was not inserted due to one of:

1. TXT Buffer is not writable (it is in wrong state)
2. Invalid TXT buffer index
3. Invalid frame length

void ctucan_give_txb_cmd(struct ctucan_priv *priv, enum ctucan_txb_command cmd, u8 buf)

Applies command on TXT buffer

Parameters

struct ctucan_priv *priv

Pointer to private data

enum ctucan_txb_command cmd

Command to give

u8 buf

Buffer index (0-based)

netdev_tx_t ctucan_start_xmit(struct sk_buff *skb, struct net_device *ndev)

Starts the transmission

Parameters

struct sk_buff *skb

sk_buff pointer that contains data to be Txed

struct net_device *ndev
 Pointer to net_device structure

Description

Invoked from upper layers to initiate transmission. Uses the next available free TXT Buffer and populates its fields to start the transmission.

Return

NETDEV_TX_OK on success, NETDEV_TX_BUSY when no free TXT buffer is available,
 negative return values reserved for error cases

void **ctucan_read_rx_frame**(struct ctucan_priv *priv, struct canfd_frame *cf, u32 ffw)
 Reads frame from RX FIFO

Parameters

struct ctucan_priv *priv
 Pointer to CTU CAN FD's private data

struct canfd_frame *cf
 Pointer to CAN frame struct

u32 ffw
 Previously read frame format word

Note

Frame format word must be read separately and provided in 'ffw'.

int **ctucan_rx**(struct *net_device* *ndev)
 Called from CAN ISR to complete the received frame processing

Parameters

struct net_device *ndev
 Pointer to net_device structure

Description

This function is invoked from the CAN isr(poll) to process the Rx frames. It does minimal processing and invokes "netif_receive_skb" to complete further processing.

Return

1 when frame is passed to the network layer, 0 when the first frame word is read but
 system is out of free SKBs temporally and left code to resolve skb allocation later, -EAGAIN
 in a case of empty Rx FIFO.

enum can_state **ctucan_read_fault_state**(struct ctucan_priv *priv)
 Reads CTU CAN FDs fault confinement state.

Parameters

struct ctucan_priv *priv
 Pointer to private data

Return

Fault confinement state of controller

```
void ctucan_get_rec_tec(struct ctucan_priv *priv, struct can_berr_counter *bec)
    Reads REC/TEC counter values from controller
```

Parameters

```
struct ctucan_priv *priv
```

Pointer to private data

```
struct can_berr_counter *bec
```

Pointer to Error counter structure

```
void ctucan_err_interrupt(struct net_device *ndev, u32 isr)
```

Error frame ISR

Parameters

```
struct net_device *ndev
```

net_device pointer

```
u32 isr
```

interrupt status register value

Description

This is the CAN error interrupt and it will check the type of error and forward the error frame to upper layers.

```
int ctucan_rx_poll(struct napi_struct *napi, int quota)
```

Poll routine for rx packets (NAPI)

Parameters

```
struct napi_struct *napi
```

NAPI structure pointer

```
int quota
```

Max number of rx packets to be processed.

Description

This is the poll routine for rx part. It will process the packets maximux quota value.

Return

Number of packets received

```
void ctucan_rotate_txb_prio(struct net_device *ndev)
```

Rotates priorities of TXT Buffers

Parameters

```
struct net_device *ndev
```

net_device pointer

```
void ctucan_tx_interrupt(struct net_device *ndev)
```

Tx done Isr

Parameters

```
struct net_device *ndev
```

net_device pointer

```
irqreturn_t ctucan_interrupt(int irq, void *dev_id)
```

CAN Isr

Parameters

int irq

irq number

void *dev_id

device id pointer

Description

This is the CTU CAN FD ISR. It checks for the type of interrupt and invokes the corresponding ISR.

Return

IRQ_NONE - If CAN device is in sleep mode, IRQ_HANDLED otherwise

```
void ctucan_chip_stop(struct net_device *ndev)
```

Driver stop routine

Parameters

struct net_device *ndev

Pointer to net_device structure

Description

This is the drivers stop routine. It will disable the interrupts and disable the controller.

```
int ctucan_open(struct net_device *ndev)
```

Driver open routine

Parameters

struct net_device *ndev

Pointer to net_device structure

Description

This is the driver open routine.

Return

0 on success and failure value on error

```
int ctucan_close(struct net_device *ndev)
```

Driver close routine

Parameters

struct net_device *ndev

Pointer to net_device structure

Return

0 always

```
int ctucan_get_berr_counter(const struct net_device *ndev, struct can_berr_counter *bec)
    error counter routine
```

Parameters

const struct net_device *ndev
Pointer to net_device structure

struct can_berr_counter *bec
Pointer to can_berr_counter structure

Description

This is the driver error counter routine.

Return

0 on success and failure value on error

```
int ctucan_pci_probe(struct pci_dev *pdev, const struct pci_device_id *ent)
    PCI registration call
```

Parameters

struct pci_dev *pdev
Handle to the pci device structure

const struct pci_device_id *ent
Pointer to the entry from ctucan_pci_tbl

Description

This function does all the memory allocation and registration for the CAN device.

Return

0 on success and failure value on error

```
void ctucan_pci_remove(struct pci_dev *pdev)
    Unregister the device after releasing the resources
```

Parameters

struct pci_dev *pdev
Handle to the pci device structure

Description

This function frees all the resources allocated to the device.

Return

0 always

```
int ctucan_platform_probe(struct platform_device *pdev)
    Platform registration call
```

Parameters

struct platform_device *pdev
Handle to the platform device structure

Description

This function does all the memory allocation and registration for the CAN device.

Return

0 on success and failure value on error

`void ctucan_platform_remove(struct platform_device *pdev)`

Unregister the device after releasing the resources

Parameters

`struct platform_device *pdev`

Handle to the platform device structure

Description

This function frees all the resources allocated to the device.

Return

0 always

CTU CAN FD IP Core and Driver Development Acknowledgment

- Odrej Ille <ondrej ille@gmail.com>
 - started the project as student at Department of Measurement, FEE, CTU
 - invested great amount of personal time and enthusiasm to the project over years
 - worked on more funded tasks
- Department of Measurement, Faculty of Electrical Engineering, Czech Technical University
 - is the main investor into the project over many years
 - uses project in their CAN/CAN FD diagnostics framework for Skoda Auto
- Digiteq Automotive
 - funding of the project CAN FD Open Cores Support Linux Kernel Based Systems
 - negotiated and paid CTU to allow public access to the project
 - provided additional funding of the work
- Department of Control Engineering, Faculty of Electrical Engineering, Czech Technical University
 - solving the project CAN FD Open Cores Support Linux Kernel Based Systems
 - providing GitLab management
 - virtual servers and computational power for continuous integration
 - providing hardware for HIL continuous integration tests
- PiKRON Ltd.
 - minor funding to initiate preparation of the project open-sourcing

- Petr Porazil <porazil@pikron.com>
 - design of PCIe transceiver addon board and assembly of boards
 - design and assembly of MZ_APO baseboard for MicroZed/Zynq based system
- Martin Jerabek <martin.jerabek01@gmail.com>
 - Linux driver development
 - continuous integration platform architect and GHDL updates
 - thesis [Open-source and Open-hardware CAN FD Protocol Support](#)
- Jiri Novak <jnovak@fel.cvut.cz>
 - project initiation, management and use at Department of Measurement, FEE, CTU
- Pavel Pisa <pisa@cmp.felk.cvut.cz>
 - initiate open-sourcing, project coordination, management at Department of Control Engineering, FEE, CTU
- Jaroslav Beran <jara.beran@gmail.com>
- system integration for Intel SoC, core and driver testing and updates
- Carsten Emde ([OSADL](#))
- provided OSADL expertise to discuss IP core licensing
- pointed to possible deadlock for LGPL and CAN bus possible patent case which lead to relicense IP core design to BSD like license
- Reiner Zitzmann and Holger Zeltwanger ([CAN in Automation](#))
- provided suggestions and help to inform community about the project and invited us to events focused on CAN bus future development directions
- Jan Charvat
- implemented CTU CAN FD functional model for QEMU which has been integrated into QEMU mainline ([docs/system/devices/can.rst](#))
- Bachelor thesis Model of CAN FD Communication Controller for QEMU Emulator

Notes

6.3.3 Flexcan CAN Controller driver

Authors: Marc Kleine-Budde <mkl@pengutronix.de>, Dario Binacchi <dario.binacchi@amarulasolutions.com>

On/off RTR frames reception

For most flexcan IP cores the driver supports 2 RX modes:

- FIFO
- mailbox

The older flexcan cores (integrated into the i.MX25, i.MX28, i.MX35 and i.MX53 SOCs) only receive RTR frames if the controller is configured for RX-FIFO mode.

The RX FIFO mode uses a hardware FIFO with a depth of 6 CAN frames, while the mailbox mode uses a software FIFO with a depth of up to 62 CAN frames. With the help of the bigger buffer, the mailbox mode performs better under high system load situations.

As reception of RTR frames is part of the CAN standard, all flexcan cores come up in a mode where RTR reception is possible.

With the "rx-rtr" private flag the ability to receive RTR frames can be waived at the expense of losing the ability to receive RTR messages. This trade off is beneficial in certain use cases.

"rx-rtr" on

Receive RTR frames. (default)

The CAN controller can and will receive RTR frames.

On some IP cores the controller cannot receive RTR frames in the more performant "RX mailbox" mode and will use "RX FIFO" mode instead.

"rx-rtr" off

Waive ability to receive RTR frames. (not supported on all IP cores)

This mode activates the "RX mailbox mode" for better performance, on some IP cores RTR frames cannot be received anymore.

The setting can only be changed if the interface is down:

```
ip link set dev can0 down
ethtool --set-priv-flags can0 rx-rtr {off|on}
ip link set dev can0 up
```

6.4 Cellular Modem Device Drivers

Contents:

6.4.1 Rmnet Driver

1. Introduction

rmnet driver is used for supporting the Multiplexing and aggregation Protocol (MAP). This protocol is used by all recent chipsets using Qualcomm Technologies, Inc. modems.

This driver can be used to register onto any physical network device in IP mode. Physical transports include USB, HSIC, PCIe and IP accelerator.

Multiplexing allows for creation of logical netdevices (rmnet devices) to handle multiple private data networks (PDN) like a default internet, tethering, multimedia messaging service (MMS) or IP media subsystem (IMS). Hardware sends packets with MAP headers to rmnet. Based on the multiplexer id, rmnet routes to the appropriate PDN after removing the MAP header.

Aggregation is required to achieve high data rates. This involves hardware sending aggregated bunch of MAP frames. rmnet driver will de-aggregate these MAP frames and send them to appropriate PDN's.

2. Packet format

- a. MAP packet v1 (data / control)

MAP header fields are in big endian format.

Packet format:

Bit	0	1	2-7	8-15	16-31
Function	Command / Data	Reserved	Pad	Multiplexer ID	Payload length
Bit	32-x				
Function	Raw bytes				

Command (1)/ Data (0) bit value is to indicate if the packet is a MAP command or data packet. Command packet is used for transport level flow control. Data packets are standard IP packets.

Reserved bits must be zero when sent and ignored when received.

Padding is the number of bytes to be appended to the payload to ensure 4 byte alignment.

Multiplexer ID is to indicate the PDN on which data has to be sent.

Payload length includes the padding length but does not include MAP header length.

- b. Map packet v4 (data / control)

MAP header fields are in big endian format.

Packet format:

Bit	0	1	2-7	8-15	16-31
Function	Command / Data	Reserved	Pad	Multiplexer ID	Payload length
Bit	32-(x-33)	(x-32)-x			
Function	Raw bytes	Checksum offload header			

Command (1)/ Data (0) bit value is to indicate if the packet is a MAP command or data packet. Command packet is used for transport level flow control. Data packets are standard IP packets.

Reserved bits must be zero when sent and ignored when received.

Padding is the number of bytes to be appended to the payload to ensure 4 byte alignment.

Multiplexer ID is to indicate the PDN on which data has to be sent.

Payload length includes the padding length but does not include MAP header length.

Checksum offload header, has the information about the checksum processing done by the hardware. Checksum offload header fields are in big endian format.

Packet format:

Bit Function	0-14 Reserved	15 Valid	16-31 Checksum start offset
Bit Function	31-47 Checksum length	48-64	Checksum value

Reserved bits must be zero when sent and ignored when received.

Valid bit indicates whether the partial checksum is calculated and is valid. Set to 1, if its is valid. Set to 0 otherwise.

Padding is the number of bytes to be appended to the payload to ensure 4 byte alignment.

Checksum start offset, Indicates the offset in bytes from the beginning of the IP header, from which modem computed checksum.

Checksum length is the Length in bytes starting from CKSUM_START_OFFSET, over which checksum is computed.

Checksum value, indicates the checksum computed.

c. MAP packet v5 (data / control)

MAP header fields are in big endian format.

Packet format:

Bit Function	0 Command / Data	1 Next header	2-7 Pad	8-15 Multiplexer ID	16-31 Payload length
Bit Function	32-x Raw bytes				

Command (1)/ Data (0) bit value is to indicate if the packet is a MAP command or data packet. Command packet is used for transport level flow control. Data packets are standard IP packets.

Next header is used to indicate the presence of another header, currently is limited to checksum header.

Padding is the number of bytes to be appended to the payload to ensure 4 byte alignment.

Multiplexer ID is to indicate the PDN on which data has to be sent.

Payload length includes the padding length but does not include MAP header length.

d. Checksum offload header v5

Checksum offload header fields are in big endian format.

Bit 0 - 6 7 8-15 16-31 Function Header Type Next Header Checksum Valid Reserved

Header Type is to indicate the type of header, this usually is set to CHECKSUM

Header types = ===== 0 Reserved 1 Reserved 2 checksum header

Checksum Valid is to indicate whether the header checksum is valid. Value of 1 implies that checksum is calculated on this packet and is valid, value of 0 indicates that the calculated packet checksum is invalid.

Reserved bits must be zero when sent and ignored when received.

e. MAP packet v1/v5 (command specific):

Bit	0	1	2 - 7	8 - 15	16 - 31
Function length	Command	Reserved	Pad	Multiplexer ID	Payload
Bit	32 - 39	40 - 45	46 - 47	48 - 63	
Function	Command name	Reserved	Command Type	Reserved	
Bit	64 - 95				
Function	Transaction ID				
Bit	96 - 127				
Function	Command data				

Command 1 indicates disabling flow while 2 is enabling flow

Command types

0	for MAP command request
1	is to acknowledge the receipt of a command
2	is for unsupported commands
3	is for error during processing of commands

f. Aggregation

Aggregation is multiple MAP packets (can be data or command) delivered to rmnet in a single linear skb. rmnet will process the individual packets and either ACK the MAP command or deliver the IP packet to the network stack as needed

MAP header|IP Packet|Optional padding|MAP header|IP Packet|Optional padding....

MAP header|IP Packet|Optional padding|MAP header|Command Packet|Optional pad...

3. Userspace configuration

rmnet userspace configuration is done through netlink using iproute2 <https://git.kernel.org/pub/scm/network/iproute2/iproute2.git/>

The driver uses rtnl_link_ops for communication.

6.5 Ethernet Device Drivers

Device drivers for Ethernet and Ethernet-based virtual function devices.

Contents:

6.5.1 Linux and the 3Com EtherLink III Series Ethercards (driver v1.18c and higher)

This file contains the instructions and caveats for v1.18c and higher versions of the 3c509 driver. You should not use the driver without reading this file.

release 1.0

28 February 2002

Current maintainer (corrections to):

David Ruggiero <jdr@farfalle.com>

Introduction

The following are notes and information on using the 3Com EtherLink III series ethercards in Linux. These cards are commonly known by the most widely-used card's 3Com model number, 3c509. They are all 10mb/s ISA-bus cards and shouldn't be (but sometimes are) confused with the similarly-numbered PCI-bus "3c905" (aka "Vortex" or "Boomerang") series. Kernel support for the 3c509 family is provided by the module 3c509.c, which has code to support all of the following models:

- 3c509 (original ISA card)
- 3c509B (later revision of the ISA card; supports full-duplex)
- 3c589 (PCMCIA)
- 3c589B (later revision of the 3c589; supports full-duplex)
- 3c579 (EISA)

Large portions of this documentation were heavily borrowed from the guide written the original author of the 3c509 driver, Donald Becker. The master copy of that document, which contains notes on older versions of the driver, currently resides on Scyld web server: <http://www.scyld.com/>.

Special Driver Features

Overriding card settings

The driver allows boot- or load-time overriding of the card's detected IOADDR, IRQ, and transceiver settings, although this capability shouldn't generally be needed except to enable full-duplex mode (see below). An example of the syntax for LILO parameters for doing this:

```
ether=10,0x310,3,0x3c509,eth0
```

This configures the first found 3c509 card for IRQ 10, base I/O 0x310, and transceiver type 3 (10base2). The flag "0x3c509" must be set to avoid conflicts with other card types when overriding the I/O address. When the driver is loaded as a module, only the IRQ may be overridden. For example, setting two cards to IRQ10 and IRQ11 is done by using the irq module option:

```
options 3c509 irq=10,11
```

Full-duplex mode

The v1.18c driver added support for the 3c509B's full-duplex capabilities. In order to enable and successfully use full-duplex mode, three conditions must be met:

- (a) You must have a Etherlink III card model whose hardware supports full-duplex operations. Currently, the only members of the 3c509 family that are positively known to support full-duplex are the 3c509B (ISA bus) and 3c589B (PCMCIA) cards. Cards without the "B" model designation do *not* support full-duplex mode; these include the original 3c509 (no "B"), the original 3c589, the 3c529 (MCA bus), and the 3c579 (EISA bus).
- (b) You must be using your card's 10baseT transceiver (i.e., the RJ-45 connector), not its AUI (thick-net) or 10base2 (thin-net/coax) interfaces. AUI and 10base2 network cabling is physically incapable of full-duplex operation.
- (c) Most importantly, your 3c509B must be connected to a link partner that is itself full-duplex capable. This is almost certainly one of two things: a full-duplex-capable Ethernet switch (*not* a hub), or a full-duplex-capable NIC on another system that's connected directly to the 3c509B via a crossover cable.

Full-duplex mode can be enabled using 'ethtool'.

Warning: Extremely important caution concerning full-duplex mode

Understand that the 3c509B's hardware's full-duplex support is much more limited than that provided by more modern network interface cards. Although at the physical layer of the network it fully supports full-duplex operation, the card was designed before the current Ethernet auto-negotiation (N-way) spec was written. This means that the 3c509B family ***cannot and will not auto-negotiate a full-duplex connection with its link partner under any circumstances, no matter how it is initialized***. If the full-duplex mode of the 3c509B is enabled, its link partner will very likely need to be independently forced into full-duplex mode as well; otherwise various nasty failures will occur - at the very least, you'll see massive numbers of packet collisions. This is one of very rare circumstances where disabling auto-negotiation and forcing the duplex mode of a network interface card or switch would ever be necessary or desirable.

Available Transceiver Types

For versions of the driver v1.18c and above, the available transceiver types are:

0	transceiver type from EEPROM config (normally 10baseT); force half-duplex
1	AUI (thick-net / DB15 connector)
2	(undefined)
3	10base2 (thin-net == coax / BNC connector)
4	10baseT (RJ-45 connector); force half-duplex mode
8	transceiver type and duplex mode taken from card's EEPROM config settings
12	10baseT (RJ-45 connector); force full-duplex mode

Prior to driver version 1.18c, only transceiver codes 0-4 were supported. Note that the new transceiver codes 8 and 12 are the *only* ones that will enable full-duplex mode, no matter what the card's detected EEPROM settings might be. This insured that merely upgrading the driver

from an earlier version would never automatically enable full-duplex mode in an existing installation; it must always be explicitly enabled via one of these code in order to be activated.

The transceiver type can be changed using 'ethtool'.

Interpretation of error messages and common problems

Error Messages

eth0: Infinite loop in interrupt, status 2011. These are "mostly harmless" message indicating that the driver had too much work during that interrupt cycle. With a status of 0x2011 you are receiving packets faster than they can be removed from the card. This should be rare or impossible in normal operation. Possible causes of this error report are:

- a "green" mode enabled that slows the processor down when there is no keyboard activity.
- some other device or device driver hogging the bus or disabling interrupts. Check /proc/interrupts for excessive interrupt counts. The timer tick interrupt should always be incrementing faster than the others.

No received packets

If a 3c509, 3c562 or 3c589 can successfully transmit packets, but never receives packets (as reported by /proc/net/dev or 'ifconfig') you likely have an interrupt line problem. Check /proc/interrupts to verify that the card is actually generating interrupts. If the interrupt count is not increasing you likely have a physical conflict with two devices trying to use the same ISA IRQ line. The common conflict is with a sound card on IRQ10 or IRQ5, and the easiest solution is to move the 3c509 to a different interrupt line. If the device is receiving packets but 'ping' doesn't work, you have a routing problem.

Tx Carrier Errors Reported in /proc/net/dev

If an EtherLink III appears to transmit packets, but the "Tx carrier errors" field in /proc/net/dev increments as quickly as the Tx packet count, you likely have an unterminated network or the incorrect media transceiver selected.

3c509B card is not detected on machines with an ISA PnP BIOS.

While the updated driver works with most PnP BIOS programs, it does not work with all. This can be fixed by disabling PnP support using the 3Com-supplied setup program.

3c509 card is not detected on overclocked machines

Increase the delay time in `id_read_eeprom()` from the current value, 500, to an absurdly high value, such as 5000.

Decoding Status and Error Messages

The bits in the main status register are:

value	description
0x01	Interrupt latch
0x02	Tx overrun, or Rx underrun
0x04	Tx complete
0x08	Tx FIFO room available
0x10	A complete Rx packet has arrived
0x20	A Rx packet has started to arrive
0x40	The driver has requested an interrupt
0x80	Statistics counter nearly full

The bits in the transmit (Tx) status word are:

value	description
0x02	Out-of-window collision.
0x04	Status stack overflow (normally impossible).
0x08	16 collisions.
0x10	Tx underrun (not enough PCI bus bandwidth).
0x20	Tx jabber.
0x40	Tx interrupt requested.
0x80	Status is valid (this should always be set).

When a transmit error occurs the driver produces a status message such as:

```
eth0: Transmit error, Tx status register 82
```

The two values typically seen here are:

0x82

Out of window collision. This typically occurs when some other Ethernet host is incorrectly set to full duplex on a half duplex network.

0x88

16 collisions. This typically occurs when the network is exceptionally busy or when another host doesn't correctly back off after a collision. If this error is mixed with 0x82 errors it is the result of a host incorrectly set to full duplex (see above).

Both of these errors are the result of network problems that should be corrected. They do not represent driver malfunction.

Revision history (this file)

28Feb02 v1.0 DR New; major portions based on Becker original 3c509 docs

6.5.2 3Com Vortex device driver

Andrew Morton

30 April 2000

This document describes the usage and errata of the 3Com "Vortex" device driver for Linux, 3c59x.c.

The driver was written by Donald Becker <becker@scyld.com>

Don is no longer the prime maintainer of this version of the driver. Please report problems to one or more of:

- Andrew Morton
- Netdev mailing list <netdev@vger.kernel.org>
- Linux kernel mailing list <linux-kernel@vger.kernel.org>

Please note the 'Reporting and Diagnosing Problems' section at the end of this file.

Since kernel 2.3.99-pre6, this driver incorporates the support for the 3c575-series Cardbus cards which used to be handled by 3c575_cb.c.

This driver supports the following hardware:

- 3c590 Vortex 10Mbps
- 3c592 EISA 10Mbps Demon/Vortex
- 3c597 EISA Fast Demon/Vortex
- 3c595 Vortex 100baseTx
- 3c595 Vortex 100baseT4
- 3c595 Vortex 100base-MII
- 3c900 Boomerang 10baseT
- 3c900 Boomerang 10Mbps Combo
- 3c900 Cyclone 10Mbps TPO
- 3c900 Cyclone 10Mbps Combo
- 3c900 Cyclone 10Mbps TPC

- 3c900B-FL Cyclone 10base-FL
- 3c905 Boomerang 100baseTx
- 3c905 Boomerang 100baseT4
- 3c905B Cyclone 100baseTx
- 3c905B Cyclone 10/100/BNC
- 3c905B-FX Cyclone 100baseFx
- 3c905C Tornado
- 3c920B-EMB-WNM (ATI Radeon 9100 IGP)
- 3c980 Cyclone
- 3c980C Python-T
- 3cSOHO100-TX Hurricane
- 3c555 Laptop Hurricane
- 3c556 Laptop Tornado
- 3c556B Laptop Hurricane
- 3c575 [Megahertz] 10/100 LAN CardBus
- 3c575 Boomerang CardBus
- 3CCFE575BT Cyclone CardBus
- 3CCFE575CT Tornado CardBus
- 3CCFE656 Cyclone CardBus
- 3CCFEM656B Cyclone+Winmodem CardBus
- 3CXFEM656C Tornado+Winmodem CardBus
- 3c450 HomePNA Tornado
- 3c920 Tornado
- 3c982 Hydra Dual Port A
- 3c982 Hydra Dual Port B
- 3c905B-T4
- 3c920B-EMB-WNM Tornado

Module parameters

There are several parameters which may be provided to the driver when its module is loaded. These are usually placed in `/etc/modprobe.d/*.conf` configuration files. Example:

```
options 3c59x debug=3 rx_copybreak=300
```

If you are using the PCMCIA tools (`cardmgr`) then the options may be placed in `/etc/pcmcia/config.opts`:

```
module "3c59x" opts "debug=3 rx_copybreak=300"
```

The supported parameters are:

debug=N

Where N is a number from 0 to 7. Anything above 3 produces a lot of output in your system logs. debug=1 is default.

options=N1,N2,N3,...

Each number in the list provides an option to the corresponding network card. So if you have two 3c905's and you wish to provide them with option 0x204 you would use:

```
options=0x204,0x204
```

The individual options are composed of a number of bitfields which have the following meanings:

Possible media type settings

0	10baseT
1	10Mbs AUI
2	undefined
3	10base2 (BNC)
4	100base-TX
5	100base-FX
6	MII (Media Independent Interface)
7	Use default setting from EEPROM
8	Autonegotiate
9	External MII
10	Use default setting from EEPROM

When generating a value for the 'options' setting, the above media selection values may be OR'ed (or added to) the following:

0x8000	Set driver debugging level to 7
0x4000	Set driver debugging level to 2
0x0400	Enable Wake-on-LAN
0x0200	Force full duplex mode.
0x0010	Bus-master enable bit (Old Vortex cards only)

For example:

```
insmod 3c59x options=0x204
```

will force full-duplex 100base-TX, rather than allowing the usual autonegotiation.

global_options=N

Sets the **options** parameter for all 3c59x NICs in the machine. Entries in the **options** array above will override any setting of this.

full_duplex=N1,N2,N3...

Similar to bit 9 of 'options'. Forces the corresponding card into full-duplex mode. Please use this in preference to the `options` parameter.

In fact, please don't use this at all! You're better off getting autonegotiation working properly.

`global_full_duplex=N1`

Sets full duplex mode for all 3c59x NICs in the machine. Entries in the `full_duplex` array above will override any setting of this.

`flow_ctrl=N1,N2,N3...`

Use 802.3x MAC-layer flow control. The 3com cards only support the PAUSE command, which means that they will stop sending packets for a short period if they receive a PAUSE frame from the link partner.

The driver only allows flow control on a link which is operating in full duplex mode.

This feature does not appear to work on the 3c905 - only 3c905B and 3c905C have been tested.

The 3com cards appear to only respond to PAUSE frames which are sent to the reserved destination address of 01:80:c2:00:00:01. They do not honour PAUSE frames which are sent to the station MAC address.

`rx_copybreak=M`

The driver preallocates 32 full-sized (1536 byte) network buffers for receiving. When a packet arrives, the driver has to decide whether to leave the packet in its full-sized buffer, or to allocate a smaller buffer and copy the packet across into it.

This is a speed/space tradeoff.

The value of `rx_copybreak` is used to decide when to make the copy. If the packet size is less than `rx_copybreak`, the packet is copied. The default value for `rx_copybreak` is 200 bytes.

`max_interrupt_work=N`

The driver's interrupt service routine can handle many receive and transmit packets in a single invocation. It does this in a loop. The value of `max_interrupt_work` governs how many times the interrupt service routine will loop. The default value is 32 loops. If this is exceeded the interrupt service routine gives up and generates a warning message "eth0: Too much work in interrupt".

`hw_checksums=N1,N2,N3,...`

Recent 3com NICs are able to generate IPv4, TCP and UDP checksums in hardware. Linux has used the Rx checksumming for a long time. The "zero copy" patch which is planned for the 2.4 kernel series allows you to make use of the NIC's DMA scatter/gather and transmit checksumming as well.

The driver is set up so that, when the zero-copy patch is applied, all Tornado and Cyclone devices will use S/G and Tx checksums.

This module parameter has been provided so you can override this decision. If you think that Tx checksums are causing a problem, you may disable the feature with `hw_checksums=0`.

If you think your NIC should be performing Tx checksumming and the driver isn't enabling it, you can force the use of hardware Tx checksumming with `hw_checksums=1`.

The driver drops a message in the logfiles to indicate whether or not it is using hardware scatter/gather and hardware Tx checksums.

Scatter/gather and hardware checksums provide considerable performance improvement for the `sendfile()` system call, but a small decrease in throughput for `send()`. There is no effect upon receive efficiency.

`compaq_ioaddr=N, compaq_irq=N, compaq_device_id=N`

"Variables to work-around the Compaq PCI BIOS32 problem"....

`watchdog=N`

Sets the time duration (in milliseconds) after which the kernel decides that the transmitter has become stuck and needs to be reset. This is mainly for debugging purposes, although it may be advantageous to increase this value on LANs which have very high collision rates. The default value is 5000 (5.0 seconds).

`enable_wol=N1,N2,N3,...`

Enable Wake-on-LAN support for the relevant interface. Donald Becker's `ether-wake` application may be used to wake suspended machines.

Also enables the NIC's power management support.

`global_enable_wol=N`

Sets `enable_wol` mode for all 3c59x NICs in the machine. Entries in the `enable_wol` array above will override any setting of this.

Media selection

A number of the older NICs such as the 3c590 and 3c900 series have 10base2 and AUI interfaces.

Prior to January, 2001 this driver would autoselect the 10base2 or AUI port if it didn't detect activity on the 10baseT port. It would then get stuck on the 10base2 port and a driver reload was necessary to switch back to 10baseT. This behaviour could not be prevented with a module option override.

Later (current) versions of the driver _do_ support locking of the media type. So if you load the driver module with

`modprobe 3c59x options=0`

it will permanently select the 10baseT port. Automatic selection of other media types does not occur.

Transmit error, Tx status register 82

This is a common error which is almost always caused by another host on the same network being in full-duplex mode, while this host is in half-duplex mode. You need to find that other host and make it run in half-duplex mode or fix this host to run in full-duplex mode.

As a last resort, you can force the 3c59x driver into full-duplex mode with

```
options 3c59x full_duplex=1
```

but this has to be viewed as a workaround for broken network gear and should only really be used for equipment which cannot autonegotiate.

Additional resources

Details of the device driver implementation are at the top of the source file.

Additional documentation is available at Don Becker's Linux Drivers site:

<http://www.scyld.com/vortex.html>

Donald Becker's driver development site:

<http://www.scyld.com/network.html>

Donald's vortex-diag program is useful for inspecting the NIC's state:

http://www.scyld.com/ethercard_diag.html

Donald's mii-diag program may be used for inspecting and manipulating the NIC's Media Independent Interface subsystem:

http://www.scyld.com/ethercard_diag.html#mii-diag

Donald's wake-on-LAN page:

<http://www.scyld.com/wakeonlan.html>

3Com's DOS-based application for setting up the NICs EEPROMs:

<ftp://ftp.3com.com/pub/nic/3c90x/3c90xx2.exe>

Autonegotiation notes

The driver uses a one-minute heartbeat for adapting to changes in the external LAN environment if link is up and 5 seconds if link is down. This means that when, for example, a machine is unplugged from a hubbed 10baseT LAN plugged into a switched 100baseT LAN, the throughput will be quite dreadful for up to sixty seconds. Be patient.

Cisco interoperability note from Walter Wong <wcw+@CMU.EDU>:

On a side note, adding HAS_NWAY seems to share a problem with the Cisco 6509 switch. Specifically, you need to change the spanning tree parameter for the port the machine is plugged into to 'portfast' mode. Otherwise, the negotiation fails. This has been an issue we've noticed for a while but haven't had the time to track down.

Cisco switches (Jeff Busch <jbusch@deja.com>)

My "standard config" for ports to which PC's/servers connect directly:

```
interface FastEthernet0/N
description machinename
load-interval 30
spanning-tree portfast
```

If autonegotiation is a problem, you may need to specify "speed 100" and "duplex full" as well (or "speed 10" and "duplex half").

WARNING: DO NOT hook up hubs/switches/bridges to these specially-configured ports! The switch will become very confused.

Reporting and diagnosing problems

Maintainers find that accurate and complete problem reports are invaluable in resolving driver problems. We are frequently not able to reproduce problems and must rely on your patience and efforts to get to the bottom of the problem.

If you believe you have a driver problem here are some of the steps you should take:

- Is it really a driver problem?

Eliminate some variables: try different cards, different computers, different cables, different ports on the switch/hub, different versions of the kernel or of the driver, etc.

- OK, it's a driver problem.

You need to generate a report. Typically this is an email to the maintainer and/or netdev@vger.kernel.org. The maintainer's email address will be in the driver source or in the MAINTAINERS file.

- The contents of your report will vary a lot depending upon the problem. If it's a kernel crash then you should refer to 'Documentation/admin-guide/reporting-issues.rst'.

But for most problems it is useful to provide the following:

- Kernel version, driver version
- A copy of the banner message which the driver generates when it is initialised.
For example:

```
eth0: 3Com PCI 3c905C Tornado at 0xa400, 00:50:da:6a:88:f0, IRQ 19
8K byte-wide RAM 5:3 Rx:Tx split, autoselect/Autonegotiate interface. MII
transceiver found at address 24, status 782d. Enabling bus-master transmits
and whole-frame receives.
```

NOTE: You must provide the `debug=2` modprobe option to generate a full detection message. Please do this:

```
modprobe 3c59x debug=2
```

- If it is a PCI device, the relevant output from '`lspci -vx`', eg:

```
00:09.0 Ethernet controller: 3Com Corporation 3c905C-TX [FastEtherlink] (rev 74)
    Subsystem: 3Com Corporation: Unknown device 9200
    Flags: bus master, medium devsel, latency 32, IRQ 19
    I/O ports at a400 [size=128]
    Memory at db000000 (32-bit, non-prefetchable) [size=128]
    Expansion ROM at <unassigned> [disabled] [size=128K]
    Capabilities: [dc] Power Management version 2
00: b7 10 00 92 07 00 10 02 74 00 00 02 08 20 00 00
10: 01 a4 00 00 00 00 00 db 00 00 00 00 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 00 b7 10 00 10
30: 00 00 00 00 dc 00 00 00 00 00 00 00 00 05 01 0a 0a
```

- A description of the environment: 10baseT? 100baseT? full/half duplex? switched or hubbed?
- Any additional module parameters which you may be providing to the driver.
- Any kernel logs which are produced. The more the merrier. If this is a large file and you are sending your report to a mailing list, mention that you have the logfile, but don't send it. If you're reporting direct to the maintainer then just send it.

To ensure that all kernel logs are available, add the following line to /etc/syslog.conf:

```
kern.* /var/log/messages
```

Then restart syslogd with:

```
/etc/rc.d/init.d/syslog restart
```

(The above may vary, depending upon which Linux distribution you use).

- If your problem is reproducible then that's great. Try the following:
 - 1) Increase the debug level. Usually this is done via:
 - a) modprobe driver debug=7
 - b) In /etc/modprobe.d/driver.conf: options driver debug=7
 - 2) Recreate the problem with the higher debug level, send all logs to the maintainer.
 - 3) Download your card's diagnostic tool from Donald Becker's website <http://www.scyld.com/ethercard_diag.html>. Download mii-diag.c as well. Build these.
 - a) Run 'vortex-diag -aaee' and 'mii-diag -v' when the card is working correctly. Save the output.
 - b) Run the above commands when the card is malfunctioning. Send both sets of output.

Finally, please be patient and be prepared to do some work. You may end up working on this problem for a week or more as the maintainer asks more questions, asks for more tests, asks

for patches to be applied, etc. At the end of it all, the problem may even remain unresolved.

6.5.3 Linux kernel driver for Elastic Network Adapter (ENA) family

Overview

ENA is a networking interface designed to make good use of modern CPU features and system architectures.

The ENA device exposes a lightweight management interface with a minimal set of memory mapped registers and extendible command set through an Admin Queue.

The driver supports a range of ENA devices, is link-speed independent (i.e., the same driver is used for 10GbE, 25GbE, 40GbE, etc), and has a negotiated and extendible feature set.

Some ENA devices support SR-IOV. This driver is used for both the SR-IOV Physical Function (PF) and Virtual Function (VF) devices.

ENA devices enable high speed and low overhead network traffic processing by providing multiple Tx/Rx queue pairs (the maximum number is advertised by the device via the Admin Queue), a dedicated MSI-X interrupt vector per Tx/Rx queue pair, adaptive interrupt moderation, and CPU cacheline optimized data placement.

The ENA driver supports industry standard TCP/IP offload features such as checksum offload. Receive-side scaling (RSS) is supported for multi-core scaling.

The ENA driver and its corresponding devices implement health monitoring mechanisms such as watchdog, enabling the device and driver to recover in a manner transparent to the application, as well as debug logs.

Some of the ENA devices support a working mode called Low-latency Queue (LLQ), which saves several more microseconds.

ENA Source Code Directory Structure

ena_com.[ch]	Management communication layer. This layer is responsible for the handling all the management (admin) communication between the device and the driver.
ena_eth_com.[ch]	Tx/Rx data path.
ena_admin_defs.h	Definition of ENA management interface.
ena_eth_io_defs.h	Definition of ENA data path interface.
ena_common_defs.h	Common definitions for ena_com layer.
ena_regs_defs.h	Definition of ENA PCI memory-mapped (MMIO) registers.
ena_netdev.[ch]	Main Linux kernel driver.
ena_ethtool.c	ethtool callbacks.
ena_xdp.[ch]	XDP files
ena_pci_id_tbl.h	Supported device IDs.

Management Interface:

ENA management interface is exposed by means of:

- PCIe Configuration Space
- Device Registers
- Admin Queue (AQ) and Admin Completion Queue (ACQ)
- Asynchronous Event Notification Queue (AENQ)

ENA device MMIO Registers are accessed only during driver initialization and are not used during further normal device operation.

AQ is used for submitting management commands, and the results/responses are reported asynchronously through ACQ.

ENA introduces a small set of management commands with room for vendor-specific extensions. Most of the management operations are framed in a generic Get/Set feature command.

The following admin queue commands are supported:

- Create I/O submission queue
- Create I/O completion queue
- Destroy I/O submission queue
- Destroy I/O completion queue
- Get feature
- Set feature
- Configure AENQ
- Get statistics

Refer to `ena_admin_defs.h` for the list of supported Get/Set Feature properties.

The Asynchronous Event Notification Queue (AENQ) is a uni-directional queue used by the ENA device to send to the driver events that cannot be reported using ACQ. AENQ events are subdivided into groups. Each group may have multiple syndromes, as shown below

The events are:

Group	Syndrome
Link state change	X
Fatal error	X
Notification	Suspend traffic
Notification	Resume traffic
Keep-Alive	X

ACQ and AENQ share the same MSI-X vector.

Keep-Alive is a special mechanism that allows monitoring the device's health. A Keep-Alive event is delivered by the device every second. The driver maintains a watchdog (WD) handler which logs the current state and statistics. If the keep-alive events aren't delivered as expected the WD resets the device and the driver.

Data Path Interface

I/O operations are based on Tx and Rx Submission Queues (Tx SQ and Rx SQ correspondingly). Each SQ has a completion queue (CQ) associated with it.

The SQs and CQs are implemented as descriptor rings in contiguous physical memory.

The ENA driver supports two Queue Operation modes for Tx SQs:

- **Regular mode:** In this mode the Tx SQs reside in the host's memory. The ENA device fetches the ENA Tx descriptors and packet data from host memory.
- **Low Latency Queue (LLQ) mode or "push-mode":** In this mode the driver pushes the transmit descriptors and the first 96 bytes of the packet directly to the ENA device memory space. The rest of the packet payload is fetched by the device. For this operation mode, the driver uses a dedicated PCI device memory BAR, which is mapped with write-combine capability.

Note that not all ENA devices support LLQ, and this feature is negotiated with the device upon initialization. If the ENA device does not support LLQ mode, the driver falls back to the regular mode.

The Rx SQs support only the regular mode.

The driver supports multi-queue for both Tx and Rx. This has various benefits:

- Reduced CPU/thread/process contention on a given Ethernet interface.
- Cache miss rate on completion is reduced, particularly for data cache lines that hold the `sk_buff` structures.
- Increased process-level parallelism when handling received packets.
- Increased data cache hit rate, by steering kernel processing of packets to the CPU, where the application thread consuming the packet is running.
- In hardware interrupt re-direction.

Interrupt Modes

The driver assigns a single MSI-X vector per queue pair (for both Tx and Rx directions). The driver assigns an additional dedicated MSI-X vector for management (for ACQ and AENQ).

Management interrupt registration is performed when the Linux kernel probes the adapter, and it is de-registered when the adapter is removed. I/O queue interrupt registration is performed when the Linux interface of the adapter is opened, and it is de-registered when the interface is closed.

The management interrupt is named:

```
ena-mgmt@pci:<PCI domain:bus:slot.function>
```

and for each queue pair, an interrupt is named:

```
<interface name>-Tx-Rx-<queue index>
```

The ENA device operates in auto-mask and auto-clear interrupt modes. That is, once MSI-X is delivered to the host, its Cause bit is automatically cleared and the interrupt is masked. The interrupt is unmasked by the driver after NAPI processing is complete.

Interrupt Moderation

ENA driver and device can operate in conventional or adaptive interrupt moderation mode.

In conventional mode the driver instructs device to postpone interrupt posting according to static interrupt delay value. The interrupt delay value can be configured through *ethtool(8)*. The following *ethtool* parameters are supported by the driver: tx-usecs, rx-usecs

In adaptive interrupt moderation mode the interrupt delay value is updated by the driver dynamically and adjusted every NAPI cycle according to the traffic nature.

Adaptive coalescing can be switched on/off through *ethtool(8)*'s adaptive_rx on|off parameter.

More information about Adaptive Interrupt Moderation (DIM) can be found in *Net DIM - Generic Network Dynamic Interrupt Moderation*

RX copybreak

The rx_copybreak is initialized by default to ENA_DEFAULT_RX_COPYBREAK and can be configured by the ETHTOOL_STUNABLE command of the SIOCETHTOOL ioctl.

Statistics

The user can obtain ENA device and driver statistics using *ethtool*. The driver can collect regular or extended statistics (including per-queue stats) from the device.

In addition the driver logs the stats to syslog upon device reset.

MTU

The driver supports an arbitrarily large MTU with a maximum that is negotiated with the device. The driver configures MTU using the SetFeature command (ENA_ADMIN_MTU property). The user can change MTU via *ip(8)* and similar legacy tools.

Stateless Offloads

The ENA driver supports:

- IPv4 header checksum offload
- TCP/UDP over IPv4/IPv6 checksum offloads

RSS

- The ENA device supports RSS that allows flexible Rx traffic steering.
- Toeplitz and CRC32 hash functions are supported.
- Different combinations of L2/L3/L4 fields can be configured as inputs for hash functions.
- The driver configures RSS settings using the AQ SetFeature command (ENA_ADMIN_RSS_HASH_FUNCTION, ENA_ADMIN_RSS_HASH_INPUT and ENA_ADMIN_RSS_INDIRECTION_TABLE_CONFIG properties).
- If the NETIF_F_RXHASH flag is set, the 32-bit result of the hash function delivered in the Rx CQ descriptor is set in the received SKB.
- The user can provide a hash key, hash function, and configure the indirection table through *ethtool(8)*.

DATA PATH

Tx

`ena_start_xmit()` is called by the stack. This function does the following:

- Maps data buffers (`skb->data` and `frags`).
- Populates `ena_buf` for the push buffer (if the driver and device are in push mode).
- Prepares ENA bufs for the remaining frags.
- Allocates a new request ID from the empty `req_id` ring. The request ID is the index of the packet in the Tx info. This is used for out-of-order Tx completions.
- Adds the packet to the proper place in the Tx ring.
- Calls `ena_com_prepare_tx()`, an ENA communication layer that converts the `ena_bufs` to ENA descriptors (and adds meta ENA descriptors as needed).
 - This function also copies the ENA descriptors and the push buffer to the Device memory space (if in push mode).
- Writes a doorbell to the ENA device.
- When the ENA device finishes sending the packet, a completion interrupt is raised.
- The interrupt handler schedules NAPI.
- The `ena_clean_tx_irq()` function is called. This function handles the completion descriptors generated by the ENA, with a single completion descriptor per completed packet.
 - `req_id` is retrieved from the completion descriptor. The `tx_info` of the packet is retrieved via the `req_id`. The data buffers are unmapped and `req_id` is returned to the empty `req_id` ring.
 - The function stops when the completion descriptors are completed or the budget is reached.

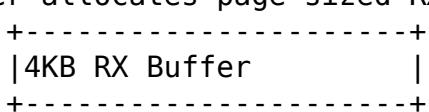
Rx

- When a packet is received from the ENA device.
- The interrupt handler schedules NAPI.
- The `ena_clean_rx_irq()` function is called. This function calls `ena_com_rx_pkt()`, an ENA communication layer function, which returns the number of descriptors used for a new packet, and zero if no new packet is found.
- `ena_rx_skb()` checks packet length:
 - If the packet is small (`len < rx_copybreak`), the driver allocates a SKB for the new packet, and copies the packet payload into the SKB data buffer.
 - * In this way the original data buffer is not passed to the stack and is reused for future Rx packets.
 - Otherwise the function unmaps the Rx buffer, sets the first descriptor as `skb`'s linear part and the other descriptors as the `skb`'s frags.
- The new SKB is updated with the necessary information (protocol, checksum hw verify result, etc), and then passed to the network stack, using the NAPI interface function `napi_gro_receive()`.

Dynamic RX Buffers (DRB)

Each RX descriptor in the RX ring is a single memory page (which is either 4KB or 16KB long depending on system's configurations). To reduce the memory allocations required when dealing with a high rate of small packets, the driver tries to reuse the remaining RX descriptor's space if more than 2KB of this page remain unused.

A simple example of this mechanism is the following sequence of events:

1. Driver allocates page-sized RX buffer and passes it to hardware

2. A 300Bytes packet is received on this buffer
3. The driver increases the ref count on this page and returns it back to HW as an RX buffer of size 4KB - 300Bytes = 3796 Bytes


This mechanism isn't used when an XDP program is loaded, or when the RX packet is less than `rx_copybreak` bytes (in which case the packet is copied out of the RX buffer into the linear part of a new skb allocated for it and the RX buffer remains the same size, see [RX copybreak](#)).

6.5.4 Altera Triple-Speed Ethernet MAC driver

Copyright © 2008-2014 Altera Corporation

This is the driver for the Altera Triple-Speed Ethernet (TSE) controllers using the SGDMA and MSGDMA soft DMA IP components. The driver uses the platform bus to obtain component resources. The designs used to test this driver were built for a Cyclone(R) V SOC FPGA board, a Cyclone(R) V FPGA board, and tested with ARM and NIOS processor hosts separately. The anticipated use cases are simple communications between an embedded system and an external peer for status and simple configuration of the embedded system.

For more information visit www.altera.com and www.rocketboards.org. Support forums for the driver may be found on www.rocketboards.org, and a design used to test this driver may be found there as well. Support is also available from the maintainer of this driver, found in MAINTAINERS.

The Triple-Speed Ethernet, SGDMA, and MSGDMA components are all soft IP components that can be assembled and built into an FPGA using the Altera Quartus toolchain. Quartus 13.1 and 14.0 were used to build the design that this driver was tested against. The sopc2dts tool is used to create the device tree for the driver, and may be found at rocketboards.org.

The driver probe function examines the device tree and determines if the Triple-Speed Ethernet instance is using an SGDMA or MSGDMA component. The probe function then installs the appropriate set of DMA routines to initialize, setup transmits, receives, and interrupt handling primitives for the respective configurations.

The SGDMA component is to be deprecated in the near future (over the next 1-2 years as of this writing in early 2014) in favor of the MSGDMA component. SGDMA support is included for existing designs and reference in case a developer wishes to support their own soft DMA logic and driver support. Any new designs should not use the SGDMA.

The SGDMA supports only a single transmit or receive operation at a time, and therefore will not perform as well compared to the MSGDMA soft IP. Please visit www.altera.com for known, documented SGDMA errata.

Scatter-gather DMA is not supported by the SGDMA or MSGDMA at this time. Scatter-gather DMA will be added to a future maintenance update to this driver.

Jumbo frames are not supported at this time.

The driver limits PHY operations to 10/100Mbps, and has not yet been fully tested for 1Gbps. This support will be added in a future maintenance update.

1. Kernel Configuration

The kernel configuration option is ALTERA_TSE:

Device Drivers ---> Network device support ---> Ethernet driver support ---> Altera Triple-Speed Ethernet MAC support (ALTERA_TSE)

2. Driver parameters list

- debug: message level (0: no output, 16: all);
- dma_rx_num: Number of descriptors in the RX list (default is 64);
- dma_tx_num: Number of descriptors in the TX list (default is 64).

3. Command line options

Driver parameters can be also passed in command line by using:

```
altera_tse=dma_rx_num:128,dma_tx_num:512
```

4. Driver information and notes

4.1. Transmit process

When the driver's transmit routine is called by the kernel, it sets up a transmit descriptor by calling the underlying DMA transmit routine (SGDMA or MSGDMA), and initiates a transmit operation. Once the transmit is complete, an interrupt is driven by the transmit DMA logic. The driver handles the transmit completion in the context of the interrupt handling chain by recycling resource required to send and track the requested transmit operation.

4.2. Receive process

The driver will post receive buffers to the receive DMA logic during driver initialization. Receive buffers may or may not be queued depending upon the underlying DMA logic (MSGDMA is able queue receive buffers, SGDMA is not able to queue receive buffers to the SGDMA receive logic). When a packet is received, the DMA logic generates an interrupt. The driver handles a receive interrupt by obtaining the DMA receive logic status, reaping receive completions until no more receive completions are available.

4.3. Interrupt Mitigation

The driver is able to mitigate the number of its DMA interrupts using NAPI for receive operations. Interrupt mitigation is not yet supported for transmit operations, but will be added in a future maintenance release.

4.4) Ethtool support

Ethtool is supported. Driver statistics and internal errors can be taken using: ethtool -S ethX command. It is possible to dump registers etc.

4.5) PHY Support

The driver is compatible with PAL to work with PHY and GPHY devices.

4.7) List of source files:

- Kconfig
- Makefile
- altera_tse_main.c: main network device driver
- altera_tse_ethtool.c: ethtool support
- altera_tse.h: private driver structure and common definitions
- altera_msgdma.h: MSGDMA implementation function definitions
- altera_sgdma.h: SGDMA implementation function definitions
- altera_msgdma.c: MSGDMA implementation
- altera_sgdma.c: SGDMA implementation
- altera_sgdmahw.h: SGDMA register and descriptor definitions
- altera_msgdmahw.h: MSGDMA register and descriptor definitions
- altera_utils.c: Driver utility functions
- altera_utils.h: Driver utility function definitions

5. Debug Information

The driver exports debug information such as internal statistics, debug information, MAC and DMA registers etc.

A user may use the ethtool support to get statistics: e.g. using: ethtool -S ethX (that shows the statistics counters) or sees the MAC registers: e.g. using: ethtool -d ethX

The developer can also use the "debug" module parameter to get further debug information.

6. Statistics Support

The controller and driver support a mix of IEEE standard defined statistics, RFC defined statistics, and driver or Altera defined statistics. The four specifications containing the standard definitions for these statistics are as follows:

- IEEE 802.3-2012 - IEEE Standard for Ethernet.
- RFC 2863 found at <http://www.rfc-editor.org/rfc/rfc2863.txt>.
- RFC 2819 found at <http://www.rfc-editor.org/rfc/rfc2819.txt>.
- Altera Triple Speed Ethernet User Guide, found at <http://www.altera.com>

The statistics supported by the TSE and the device driver are as follows:

"tx_packets" is equivalent to aFramesTransmittedOK defined in IEEE 802.3-2012, Section 5.2.2.1.2. This statistic is the count of frames that are successfully transmitted.

"rx_packets" is equivalent to aFramesReceivedOK defined in IEEE 802.3-2012, Section 5.2.2.1.5. This statistic is the count of frames that are successfully received. This count does not include any error packets such as CRC errors, length errors, or alignment errors.

"rx_crc_errors" is equivalent to aFrameCheckSequenceErrors defined in IEEE 802.3-2012, Section 5.2.2.1.6. This statistic is the count of frames that are an integral number of bytes in length and do not pass the CRC test as the frame is received.

"rx_align_errors" is equivalent to aAlignmentErrors defined in IEEE 802.3-2012, Section 5.2.2.1.7. This statistic is the count of frames that are not an integral number of bytes in length and do not pass the CRC test as the frame is received.

"tx_bytes" is equivalent to aOctetsTransmittedOK defined in IEEE 802.3-2012, Section 5.2.2.1.8. This statistic is the count of data and pad bytes successfully transmitted from the interface.

"rx_bytes" is equivalent to aOctetsReceivedOK defined in IEEE 802.3-2012, Section 5.2.2.1.14. This statistic is the count of data and pad bytes successfully received by the controller.

"tx_pause" is equivalent to aPAUSEMACCtrlFramesTransmitted defined in IEEE 802.3-2012, Section 30.3.4.2. This statistic is a count of PAUSE frames transmitted from the network controller.

"rx_pause" is equivalent to aPAUSEMACCtrlFramesReceived defined in IEEE 802.3-2012, Section 30.3.4.3. This statistic is a count of PAUSE frames received by the network controller.

"rx_errors" is equivalent to ifInErrors defined in RFC 2863. This statistic is a count of the number of packets received containing errors that prevented the packet from being delivered to a higher level protocol.

"tx_errors" is equivalent to ifOutErrors defined in RFC 2863. This statistic is a count of the number of packets that could not be transmitted due to errors.

"rx_unicast" is equivalent to ifInUcastPkts defined in RFC 2863. This statistic is a count of the number of packets received that were not addressed to the broadcast address or a multicast group.

"rx_multicast" is equivalent to ifInMulticastPkts defined in RFC 2863. This statistic is a count of the number of packets received that were addressed to a multicast address group.

"rx_broadcast" is equivalent to ifInBroadcastPkts defined in RFC 2863. This statistic is a count of the number of packets received that were addressed to the broadcast address.

"tx_discards" is equivalent to ifOutDiscards defined in RFC 2863. This statistic is the number of outbound packets not transmitted even though an error was not detected. An example of a reason this might occur is to free up internal buffer space.

"tx_unicast" is equivalent to ifOutUcastPkts defined in RFC 2863. This statistic counts the number of packets transmitted that were not addressed to a multicast group or broadcast address.

"tx_multicast" is equivalent to ifOutMulticastPkts defined in RFC 2863. This statistic counts the number of packets transmitted that were addressed to a multicast group.

"tx_broadcast" is equivalent to ifOutBroadcastPkts defined in RFC 2863. This statistic counts the number of packets transmitted that were addressed to a broadcast address.

"ether_drops" is equivalent to etherStatsDropEvents defined in RFC 2819. This statistic counts the number of packets dropped due to lack of internal controller resources.

"rx_total_bytes" is equivalent to etherStatsOctets defined in RFC 2819. This statistic counts the total number of bytes received by the controller, including error and discarded packets.

"rx_total_packets" is equivalent to etherStatsPkts defined in RFC 2819. This statistic counts the total number of packets received by the controller, including error, discarded, unicast, multicast, and broadcast packets.

"rx_undersize" is equivalent to etherStatsUndersizePkts defined in RFC 2819. This statistic counts the number of correctly formed packets received less than 64 bytes long.

"rx_oversize" is equivalent to etherStatsOversizePkts defined in RFC 2819. This statistic counts the number of correctly formed packets greater than 1518 bytes long.

"rx_64_bytes" is equivalent to etherStatsPkts64Octets defined in RFC 2819. This statistic counts the total number of packets received that were 64 octets in length.

"rx_65_127_bytes" is equivalent to etherStatsPkts65to127Octets defined in RFC 2819. This statistic counts the total number of packets received that were between 65 and 127 octets in length inclusive.

"rx_128_255_bytes" is equivalent to etherStatsPkts128to255Octets defined in RFC 2819. This statistic is the total number of packets received that were between 128 and 255 octets in length inclusive.

"rx_256_511_bytes" is equivalent to etherStatsPkts256to511Octets defined in RFC 2819. This statistic is the total number of packets received that were between 256 and 511 octets in length inclusive.

"rx_512_1023_bytes" is equivalent to etherStatsPkts512to1023Octets defined in RFC 2819. This statistic is the total number of packets received that were between 512 and 1023 octets in length inclusive.

"rx_1024_1518_bytes" is equivalent to etherStatsPkts1024to1518Octets defined in RFC 2819. This statistic is the total number of packets received that were between 1024 and 1518 octets in length inclusive.

"rx_gte_1519_bytes" is a statistic defined specific to the behavior of the Altera TSE. This statistic counts the number of received good and errored frames between the length of 1519 and the maximum frame length configured in the frm_length register. See the Altera TSE User Guide for More details.

“rx_jabbers” is equivalent to etherStatsJabbers defined in RFC 2819. This statistic is the total number of packets received that were longer than 1518 octets, and had either a bad CRC with an integral number of octets (CRC Error) or a bad CRC with a non-integral number of octets (Alignment Error).

“rx_runt” is equivalent to etherStatsFragments defined in RFC 2819. This statistic is the total number of packets received that were less than 64 octets in length and had either a bad CRC with an integral number of octets (CRC error) or a bad CRC with a non-integral number of octets (Alignment Error).

6.5.5 Linux Driver for the AMD/Pensando(R) DSC adapter family

Copyright(c) 2023 Advanced Micro Devices, Inc

Identifying the Adapter

To find if one or more AMD/Pensando PCI Core devices are installed on the host, check for the PCI devices:

```
# lspci -d 1dd8:100c
b5:00.0 Processing accelerators: Pensando Systems Device 100c
b6:00.0 Processing accelerators: Pensando Systems Device 100c
```

If such devices are listed as above, then the pds_core.ko driver should find and configure them for use. There should be log entries in the kernel messages such as these:

```
$ dmesg | grep pds_core
pds_core 0000:b5:00.0: 252.048 Gb/s available PCIe bandwidth (16.0 GT/s PCIe
    ↳ x16 link)
pds_core 0000:b5:00.0: FW: 1.60.0-73
pds_core 0000:b6:00.0: 252.048 Gb/s available PCIe bandwidth (16.0 GT/s PCIe
    ↳ x16 link)
pds_core 0000:b6:00.0: FW: 1.60.0-73
```

Driver and firmware version information can be gathered with devlink:

```
$ devlink dev info pci/0000:b5:00.0
pci/0000:b5:00.0:
  driver pds_core
  serial_number FLM18420073
  versions:
    fixed:
      asic.id 0x0
      asic.rev 0x0
    running:
      fw 1.51.0-73
    stored:
      fw.goldfw 1.15.9-C-22
      fw.mainfwa 1.60.0-73
      fw.mainfbw 1.60.0-57
```

Info versions

The pds_core driver reports the following versions

Table 1: devlink info versions implemented

Name	Type	Description
fw_run	Version	Version of firmware running on the device
fw_stored_goldfw	Version	Version of firmware stored in the goldfw slot
fw_stored_mainfwa	Version	Version of firmware stored in the mainfwa slot
fw_stored_mainfb	Version	Version of firmware stored in the mainfb slot
asic_id	fixed	The ASIC type for this device
asic_rev	fixed	The revision of the ASIC for this device

Parameters

The pds_core driver implements the following generic parameters for controlling the functionality to be made available as auxiliary_bus devices.

Table 2: Generic parameters implemented

Name	Type	Description
enable_vdpftime	boolean	Enables vDPA functionality through an auxiliary_bus device

Firmware Management

The flash command can update the DSC firmware. The downloaded firmware will be saved into either of firmware bank 1 or bank 2, whichever is not currently in use, and that bank will be used for the next boot:

```
# devlink dev flash pci/0000:b5:00.0 \
    file pensando/dsc_fw_1.63.0-22.tar
```

Health Reporters

The driver supports a devlink health reporter for FW status:

```
# devlink health show pci/0000:2b:00.0 reporter fw
pci/0000:2b:00.0:
    reporter fw
        state healthy error 0 recover 0
# devlink health diagnose pci/0000:2b:00.0 reporter fw
Status: healthy State: 1 Generation: 0 Recoveries: 0
```

Enabling the driver

The driver is enabled via the standard kernel configuration system, using the make command:

```
make oldconfig/menuconfig/etc.
```

The driver is located in the menu structure at:

```
-> Device Drivers
    -> Network device support (NETDEVICES [=y])
        -> Ethernet driver support
            -> AMD devices
                -> AMD/Pensando Ethernet PDS_CORE Support
```

Support

For general Linux networking support, please use the netdev mailing list, which is monitored by AMD/Pensando personnel:

```
netdev@vger.kernel.org
```

6.5.6 PCI vDPA driver for the AMD/Pensando(R) DSC adapter family

AMD/Pensando vDPA VF Device Driver

Copyright(c) 2023 Advanced Micro Devices, Inc

Overview

The pds_vdpa driver is an auxiliary bus driver that supplies a vDPA device for use by the virtio network stack. It is used with the Pensando Virtual Function devices that offer vDPA and virtio queue services. It depends on the pds_core driver and hardware for the PF and VF PCI handling as well as for device configuration services.

Using the device

The pds_vdpa device is enabled via multiple configuration steps and depends on the pds_core driver to create and enable SR-IOV Virtual Function devices. After the VFs are enabled, we enable the vDPA service in the pds_core device to create the auxiliary devices used by pds_vdpa.

Example steps:

```
#!/bin/bash

modprobe pds_core
modprobe vdpa
```

```

modprobe pds_vdpa

PF_BDF=`ls /sys/module/pds_core/drivers/pci\:pds_core/*/sriov_numvfs | awk -F /
→ '{print $7}'` 

# Enable vDPA VF auxiliary device(s) in the PF
devlink dev param set pci/$PF_BDF name enable_vnet cmode runtime value true

# Create a VF for vDPA use
echo 1 > /sys/bus/pci/drivers/pds_core/$PF_BDF/sriov_numvfs

# Find the vDPA services/devices available
PDS_VDPA_MGMT=`vdpa mgmtdev show | grep vDPA | head -1 | cut -d: -f1` 

# Create a vDPA device for use in virtio network configurations
vdpa dev add name vdpa1 mgmtdev $PDS_VDPA_MGMT mac 00:11:22:33:44:55

# Set up an ethernet interface on the vdpa device
modprobe virtio_vdpa

```

Enabling the driver

The driver is enabled via the standard kernel configuration system, using the make command:

```
make oldconfig/menuconfig/etc.
```

The driver is located in the menu structure at:

- > **Device Drivers**
 - > **Network device support (NETDEVICES [=y])**
 - > **Ethernet driver support**
 - > **Pensando devices**
 - > Pensando Ethernet PDS_VDPA Support

Support

For general Linux networking support, please use the netdev mailing list, which is monitored by Pensando personnel:

```
netdev@vger.kernel.org
```

For more specific support needs, please use the Pensando driver support email:

```
drivers@pensando.io
```

6.5.7 PCI VFIO driver for the AMD/Pensando(R) DSC adapter family

AMD/Pensando Linux VFIO PCI Device Driver Copyright(c) 2023 Advanced Micro Devices, Inc.

Overview

The pds-vfio-pci module is a PCI driver that supports Live Migration capable Virtual Function (VF) devices in the DSC hardware.

Using the device

The pds-vfio-pci device is enabled via multiple configuration steps and depends on the pds_core driver to create and enable SR-IOV Virtual Function devices.

Shown below are the steps to bind the driver to a VF and also to the associated auxiliary device created by the pds_core driver. This example assumes the pds_core and pds-vfio-pci modules are already loaded.

```
#!/bin/bash

PF_BUS="0000:60"
PF_BDF="0000:60:00.0"
VF_BDF="0000:60:00.1"

# Prevent non-vfio VF driver from probing the VF device
echo 0 > /sys/class/pci_bus/$PF_BUS/device/$PF_BDF/sriov_drivers_autoprobe

# Create single VF for Live Migration via pds_core
echo 1 > /sys/bus/pci/drivers/pds_core/$PF_BDF/sriov_numvfs

# Allow the VF to be bound to the pds-vfio-pci driver
echo "pds-vfio-pci" > /sys/class/pci_bus/$PF_BUS/device/$VF_BDF/driver_override

# Bind the VF to the pds-vfio-pci driver
echo "$VF_BDF" > /sys/bus/pci/drivers/pds-vfio-pci/bind
```

After performing the steps above, a file in /dev/vfio/<iommu_group> should have been created.

Enabling the driver

The driver is enabled via the standard kernel configuration system, using the make command:

```
make oldconfig/menuconfig/etc.
```

The driver is located in the menu structure at:

- > **Device Drivers**
 - > **VFIO Non-Privileged userspace driver framework**
 - > VFIO support for PDS PCI devices

Support

For general Linux networking support, please use the netdev mailing list, which is monitored by Pensando personnel:

```
netdev@vger.kernel.org
```

For more specific support needs, please use the Pensando driver support email:

```
drivers@pensando.io
```

6.5.8 Marvell(Aquantia) AQton Driver

For the aQuantia Multi-Gigabit PCI Express Family of Ethernet Adapters

Identifying Your Adapter

The driver in this release is compatible with AQC-100, AQC-107, AQC-108 based ethernet adapters.

SFP+ Devices (for AQC-100 based adapters)

This release tested with passive Direct Attach Cables (DAC) and SFP+/LC Optical Transceiver.

Configuration

Viewing Link Messages

Link messages will not be displayed to the console if the distribution is restricting system messages. In order to see network driver link messages on your console, set dmesg to eight by entering the following:

```
dmesg -n 8
```

Note: This setting is not saved across reboots.

Jumbo Frames

The driver supports Jumbo Frames for all adapters. Jumbo Frames support is enabled by changing the MTU to a value larger than the default of 1500. The maximum value for the MTU is 16000. Use the *ip* command to increase the MTU size. For example:

```
ip link set mtu 16000 dev enp1s0
```

ethtool

The driver utilizes the ethtool interface for driver configuration and diagnostics, as well as displaying statistical information. The latest ethtool version is required for this functionality.

NAPI

NAPI (Rx polling mode) is supported in the atlantic driver.

Supported ethtool options

Viewing adapter settings

```
ethtool <ethX>
```

Output example:

```
Settings for enpl1s0:  
Supported ports: [ TP ]  
Supported link modes: 100baseT/Full  
                                1000baseT/Full  
                                10000baseT/Full  
                                2500baseT/Full  
                                5000baseT/Full  
Supported pause frame use: Symmetric  
Supports auto-negotiation: Yes  
Supported FEC modes: Not reported  
Advertised link modes: 100baseT/Full  
                                1000baseT/Full  
                                10000baseT/Full  
                                2500baseT/Full  
                                5000baseT/Full  
Advertised pause frame use: Symmetric  
Advertised auto-negotiation: Yes  
Advertised FEC modes: Not reported  
Speed: 10000Mb/s  
Duplex: Full  
Port: Twisted Pair  
PHYAD: 0  
Transceiver: internal  
Auto-negotiation: on  
MDI-X: Unknown  
Supports Wake-on: g  
Wake-on: d  
Link detected: yes
```

Note: AQrate speeds (2.5/5 Gb/s) will be displayed only with linux kernels > 4.10.

But you can still use these speeds:

```
ethtool -s eth0 autoneg off speed 2500
```

Viewing adapter information

```
ethtool -i <ethX>
```

Output example:

```
driver: atlantic
version: 5.2.0-050200rc5-generic-kern
firmware-version: 3.1.78
expansion-rom-version:
bus-info: 0000:01:00.0
supports-statistics: yes
supports-test: no
supports-eeprom-access: no
supports-register-dump: yes
supports-priv-flags: no
```

Viewing Ethernet adapter statistics

```
ethtool -S <ethX>
```

Output example:

```
NIC statistics:
InPackets: 13238607
InUCast: 13293852
InMCast: 52
InBCast: 3
InErrors: 0
OutPackets: 23703019
OutUCast: 23704941
OutMCast: 67
OutBCast: 11
InUCastOctects: 213182760
OutUCastOctects: 22698443
InMCastOctects: 6600
OutMCastOctects: 8776
InBCastOctects: 192
OutBCastOctects: 704
InOctects: 2131839552
OutOctects: 226938073
InPacketsDma: 95532300
OutPacketsDma: 59503397
```

```
InOctetsDma: 1137102462
OutOctetsDma: 2394339518
InDroppedDma: 0
Queue[0] InPackets: 23567131
Queue[0] OutPackets: 20070028
Queue[0] InJumboPackets: 0
Queue[0] InLroPackets: 0
Queue[0] InErrors: 0
Queue[1] InPackets: 45428967
Queue[1] OutPackets: 11306178
Queue[1] InJumboPackets: 0
Queue[1] InLroPackets: 0
Queue[1] InErrors: 0
Queue[2] InPackets: 3187011
Queue[2] OutPackets: 13080381
Queue[2] InJumboPackets: 0
Queue[2] InLroPackets: 0
Queue[2] InErrors: 0
Queue[3] InPackets: 23349136
Queue[3] OutPackets: 15046810
Queue[3] InJumboPackets: 0
Queue[3] InLroPackets: 0
Queue[3] InErrors: 0
```

Interrupt coalescing support

ITR mode, TX/RX coalescing timings could be viewed with:

```
ethtool -c <ethX>
```

and changed with:

```
ethtool -C <ethX> tx-usecs <usecs> rx-usecs <usecs>
```

To disable coalescing:

```
ethtool -C <ethX> tx-usecs 0 rx-usecs 0 tx-max-frames 1 tx-max-frames 1
```

Wake on LAN support

WOL support by magic packet:

```
ethtool -s <ethX> wol g
```

To disable WOL:

```
ethtool -s <ethX> wol d
```

Set and check the driver message level

Set message level

```
ethtool -s <ethX> mslvl <level>
```

Level values:

0x0001	general driver status.
0x0002	hardware probing.
0x0004	link state.
0x0008	periodic status check.
0x0010	interface being brought down.
0x0020	interface being brought up.
0x0040	receive error.
0x0080	transmit error.
0x0200	interrupt handling.
0x0400	transmit completion.
0x0800	receive completion.
0x1000	packet contents.
0x2000	hardware status.
0x4000	Wake-on-LAN status.

By default, the level of debugging messages is set 0x0001(general driver status).

Check message level

```
ethtool <ethX> | grep "Current message level"
```

If you want to disable the output of messages:

```
ethtool -s <ethX> mslvl 0
```

RX flow rules (ntuple filters)

There are separate rules supported, that applies in that order:

1. 16 VLAN ID rules
2. 16 L2 EtherType rules
3. 8 L3/L4 5-Tuple rules

The driver utilizes the ethtool interface for configuring ntuple filters, via `ethtool -N <device> <filter>`.

To enable or disable the RX flow rules:

```
ethtool -K ethX ntuple <on|off>
```

When disabling ntuple filters, all the user programmed filters are flushed from the driver cache and hardware. All needed filters must be re-added when ntuple is re-enabled.

Because of the fixed order of the rules, the location of filters is also fixed:

- Locations 0 - 15 for VLAN ID filters
- Locations 16 - 31 for L2 EtherType filters
- Locations 32 - 39 for L3/L4 5-tuple filters (locations 32, 36 for IPv6)

The L3/L4 5-tuple (protocol, source and destination IP address, source and destination TCP/UDP/SCTP port) is compared against 8 filters. For IPv4, up to 8 source and destination addresses can be matched. For IPv6, up to 2 pairs of addresses can be supported. Source and destination ports are only compared for TCP/UDP/SCTP packets.

To add a filter that directs packet to queue 5, use <-N|-U|--config-nfc|--config-ntuple> switch:

```
ethtool -N <ethX> flow-type udp4 src-ip 10.0.0.1 dst-ip 10.0.0.2 src-
↪port 2000 dst-port 2001 action 5 <loc 32>
```

- action is the queue number.
- loc is the rule number.

For flow-type ip4|udp4|tcp4|sctp4|ip6|udp6|tcp6|sctp6 you must set the loc number within 32 - 39. For flow-type ip4|udp4|tcp4|sctp4|ip6|udp6|tcp6|sctp6 you can set 8 rules for traffic IPv4 or you can set 2 rules for traffic IPv6. Loc number traffic IPv6 is 32 and 36. At the moment you can not use IPv4 and IPv6 filters at the same time.

Example filter for IPv6 filter traffic:

```
sudo ethtool -N <ethX> flow-type tcp6 src-ip 2001:db8:0:f101::1 dst-ip ↪
↪2001:db8:0:f101::2 action 1 loc 32
sudo ethtool -N <ethX> flow-type ip6 src-ip 2001:db8:0:f101::2 dst-ip ↪
↪2001:db8:0:f101::5 action -1 loc 36
```

Example filter for IPv4 filter traffic:

```
sudo ethtool -N <ethX> flow-type udp4 src-ip 10.0.0.4 dst-ip 10.0.0.7 ↪
↪src-port 2000 dst-port 2001 loc 32
sudo ethtool -N <ethX> flow-type tcp4 src-ip 10.0.0.3 dst-ip 10.0.0.9 ↪
↪src-port 2000 dst-port 2001 loc 33
sudo ethtool -N <ethX> flow-type ip4 src-ip 10.0.0.6 dst-ip 10.0.0.4 ↪
↪loc 34
```

If you set action -1, then all traffic corresponding to the filter will be discarded.

The maximum value action is 31.

The VLAN filter (VLAN id) is compared against 16 filters. VLAN id must be accompanied by mask 0xF000. That is to distinguish VLAN filter from L2 Ethertype filter with UserPriority since both User Priority and VLAN ID are passed in the same 'vlan' parameter.

To add a filter that directs packets from VLAN 2001 to queue 5:

```
ethtool -N <ethX> flow-type ip4 vlan 2001 m 0xF000 action 1 loc 0
```

L2 EtherType filters allows filter packet by EtherType field or both EtherType and User Priority (PCP) field of 802.1Q. UserPriority (vlan) parameter must be accompanied by mask 0x1FFF. That is to distinguish VLAN filter from L2 Ethertype filter with UserPriority since both User Priority and VLAN ID are passed in the same 'vlan' parameter.

To add a filter that directs IP4 packess of priority 3 to queue 3:

```
ethtool -N <ethX> flow-type ether proto 0x800 vlan 0x600 m 0x1FFF ↵
    ↵action 3 loc 16
```

To see the list of filters currently present:

```
ethtool <-u|-n|--show-nfc|--show-ntuple> <ethX>
```

Rules may be deleted from the table itself. This is done using:

```
sudo ethtool <-N|-U|--config-nfc|--config-ntuple> <ethX> delete <loc>
```

- loc is the rule number to be deleted.

Rx filters is an interface to load the filter table that funnels all flow into queue 0 unless an alternative queue is specified using "action". In that case, any flow that matches the filter criteria will be directed to the appropriate queue. RX filters is supported on all kernels 2.6.30 and later.

RSS for UDP

Currently, NIC does not support RSS for fragmented IP packets, which leads to incorrect working of RSS for fragmented UDP traffic. To disable RSS for UDP the RX Flow L3/L4 rule may be used.

Example:

```
ethtool -N eth0 flow-type udp4 action 0 loc 32
```

UDP GSO hardware offload

UDP GSO allows to boost UDP tx rates by offloading UDP headers allocation into hardware. A special userspace socket option is required for this, could be validated with /kernel/tools/testing/selftests/net/:

```
udpgso_bench_tx -u -4 -D 10.0.1.1 -s 6300 -S 100
```

Will cause sending out of 100 byte sized UDP packets formed from single 6300 bytes user buffer.

UDP GSO is configured by:

```
ethtool -K eth0 tx-udp-segmentation on
```

Private flags (testing)

Atlantic driver supports private flags for hardware custom features:

```
$ ethtool --show-priv-flags ethX

Private flags for ethX:
DMASystemLoopback : off
PKTSystemLoopback : off
DMANetworkLoopback : off
PHYInternalLoopback: off
PHYExternalLoopback: off
```

Example:

```
$ ethtool --set-priv-flags ethX DMASystemLoopback on
```

DMASystemLoopback: DMA Host loopback. PKTSystemLoopback: Packet buffer host loopback. DMANetworkLoopback: Network side loopback on DMA block. PHYInternalLoopback: Internal loopback on Phy. PHYExternalLoopback: External loopback on Phy (with loopback ethernet cable).

Command Line Parameters

The following command line parameters are available on atlantic driver:

aq_itr -Interrupt throttling mode

Accepted values: 0, 1, 0xFFFF

Default value: 0xFFFF

0	Disable interrupt throttling.
1	Enable interrupt throttling and use specified tx and rx rates.
0xFFFF	Auto throttling mode. Driver will choose the best RX and TX interrupt throttling settings based on link speed.

aq_itr_tx - TX interrupt throttle rate

Accepted values: 0 - 0x1FF

Default value: 0

TX side throttling in microseconds. Adapter will setup maximum interrupt delay to this value. Minimum interrupt delay will be a half of this value

aq_itr_rx - RX interrupt throttle rate

Accepted values: 0 - 0x1FF

Default value: 0

RX side throttling in microseconds. Adapter will setup maximum interrupt delay to this value. Minimum interrupt delay will be a half of this value

Note: ITR settings could be changed in runtime by ethtool -c means (see below)

Config file parameters

For some fine tuning and performance optimizations, some parameters can be changed in the {source_dir}/aq_cfg.h file.

AQ_CFG_RX_PAGEORDER

Default value: 0

RX page order override. That's a power of 2 number of RX pages allocated for each descriptor. Received descriptor size is still limited by AQ_CFG_RX_FRAME_MAX.

Increasing pageorder makes page reuse better (actual on iommu enabled systems).

AQ_CFG_RX_REFILL_THRES

Default value: 32

RX refill threshold. RX path will not refill freed descriptors until the specified number of free descriptors is observed. Larger values may help better page reuse but may lead to packet drops as well.

AQ_CFG_VECS_DEF

Number of queues

Valid Range: 0 - 8 (up to AQ_CFG_VECS_MAX)

Default value: 8

Notice this value will be capped by the number of cores available on the system.

AQ_CFG_IS_RSS_DEF

Enable/disable Receive Side Scaling

This feature allows the adapter to distribute receive processing across multiple CPU-cores and to prevent from overloading a single CPU core.

Valid values

0	disabled
1	enabled

Default value: 1

AQ_CFG_NUM_RSS_QUEUES_DEF

Number of queues for Receive Side Scaling

Valid Range: 0 - 8 (up to AQ_CFG_VECS_DEF)

Default value: AQ_CFG_VECS_DEF

AQ_CFG_IS_LRO_DEF

Enable/disable Large Receive Offload

This offload enables the adapter to coalesce multiple TCP segments and indicate them as a single coalesced unit to the OS networking subsystem.

The system consumes less energy but it also introduces more latency in packets processing.

Valid values

0	disabled
1	enabled

Default value: 1

AQ_CFG_TX_CLEAN_BUDGET

Maximum descriptors to cleanup on TX at once.

Default value: 256

After the aq_cfg.h file changed the driver must be rebuilt to take effect.

Support

If an issue is identified with the released source code on the supported kernel with a supported adapter, email the specific information related to the issue to aqn_support@marvell.com

License

aQuantia Corporation Network Driver

Copyright © 2014 - 2019 aQuantia Corporation.

This program is free software; you can redistribute it and/or modify it under the terms and conditions of the GNU General Public License, version 2, as published by the Free Software Foundation.

6.5.9 Chelsio N210 10Gb Ethernet Network Controller

Driver Release Notes for Linux

Version 2.1.1

June 20, 2005

Introduction

This document describes the Linux driver for Chelsio 10Gb Ethernet Network Controller. This driver supports the Chelsio N210 NIC and is backward compatible with the Chelsio N110 model 10Gb NICs.

Features

Adaptive Interrupts (adaptive-rx)

This feature provides an adaptive algorithm that adjusts the interrupt coalescing parameters, allowing the driver to dynamically adapt the latency settings to achieve the highest performance during various types of network load.

The interface used to control this feature is ethtool. Please see the ethtool manpage for additional usage information.

By default, adaptive-rx is disabled. To enable adaptive-rx:

```
ethtool -C <interface> adaptive-rx on
```

To disable adaptive-rx, use ethtool:

```
ethtool -C <interface> adaptive-rx off
```

After disabling adaptive-rx, the timer latency value will be set to 50us. You may set the timer latency after disabling adaptive-rx:

```
ethtool -C <interface> rx-usecs <microseconds>
```

An example to set the timer latency value to 100us on eth0:

```
ethtool -C eth0 rx-usecs 100
```

You may also provide a timer latency value while disabling adaptive-rx:

```
ethtool -C <interface> adaptive-rx off rx-usecs <microseconds>
```

If adaptive-rx is disabled and a timer latency value is specified, the timer will be set to the specified value until changed by the user or until adaptive-rx is enabled.

To view the status of the adaptive-rx and timer latency values:

```
ethtool -c <interface>
```

TCP Segmentation Offloading (TSO) Support

This feature, also known as "large send", enables a system's protocol stack to offload portions of outbound TCP processing to a network interface card thereby reducing system CPU utilization and enhancing performance.

The interface used to control this feature is ethtool version 1.8 or higher. Please see the ethtool manpage for additional usage information.

By default, TSO is enabled. To disable TSO:

```
ethtool -K <interface> tso off
```

To enable TSO:

```
ethtool -K <interface> tso on
```

To view the status of TSO:

```
ethtool -k <interface>
```

Performance

The following information is provided as an example of how to change system parameters for "performance tuning" and what value to use. You may or may not want to change these system parameters, depending on your server/workstation application. Doing so is not warranted in any way by Chelsio Communications, and is done at "YOUR OWN RISK". Chelsio will not be held responsible for loss of data or damage to equipment.

Your distribution may have a different way of doing things, or you may prefer a different method. These commands are shown only to provide an example of what to do and are by no means definitive.

Making any of the following system changes will only last until you reboot your system. You may want to write a script that runs at boot-up which includes the optimal settings for your system.

Setting PCI Latency Timer:

```
setpci -d 1425::
```

- 0x0c.l=0x0000F800

Disabling TCP timestamp:

```
sysctl -w net.ipv4.tcp_timestamps=0
```

Disabling SACK:

```
sysctl -w net.ipv4.tcp_sack=0
```

Setting large number of incoming connection requests:

```
sysctl -w net.ipv4.tcp_max_syn_backlog=3000
```

Setting maximum receive socket buffer size:

```
sysctl -w net.core.rmem_max=1024000
```

Setting maximum send socket buffer size:

```
sysctl -w net.core.wmem_max=1024000
```

Set smp_affinity (on a multiprocessor system) to a single CPU:

```
echo 1 > /proc/irq/<interrupt_number>/smp_affinity
```

Setting default receive socket buffer size:

```
sysctl -w net.core.rmem_default=524287
```

Setting default send socket buffer size:

```
sysctl -w net.core.wmem_default=524287
```

Setting maximum option memory buffers:

```
sysctl -w net.core.optmem_max=524287
```

Setting maximum backlog (# of unprocessed packets before kernel drops):

```
sysctl -w net.core.netdev_max_backlog=300000
```

Setting TCP read buffers (min/default/max):

```
sysctl -w net.ipv4.tcp_rmem="10000000 10000000 10000000"
```

Setting TCP write buffers (min/pressure/max):

```
sysctl -w net.ipv4.tcp_wmem="10000000 10000000 10000000"
```

Setting TCP buffer space (min/pressure/max):

```
sysctl -w net.ipv4.tcp_mem="10000000 10000000 10000000"
```

TCP window size for single connections:

The receive buffer (RX_WINDOW) size must be at least as large as the Bandwidth-Delay Product of the communication link between the sender and receiver. Due to the variations of RTT, you may want to increase the buffer size up to 2 times the Bandwidth-Delay Product. Reference page 289 of "TCP/IP Illustrated, Volume 1, The Protocols" by W. Richard Stevens.

At 10Gb speeds, use the following formula:

```
RX_WINDOW >= 1.25MBytes * RTT(in milliseconds)  
Example for RTT with 100us: RX_WINDOW = (1,250,000 * 0.1) = 125,000
```

RX_WINDOW sizes of 256KB - 512KB should be sufficient.

Setting the min, max, and default receive buffer (RX_WINDOW) size:

```
sysctl -w net.ipv4.tcp_rmem="<min> <default> <max>"
```

TCP window size for multiple connections:

The receive buffer (RX_WINDOW) size may be calculated the same as single connections, but should be divided by the number of connections. The smaller window prevents congestion and facilitates better pacing, especially if/when MAC level flow control does not work well or when it is not supported on the machine. Experimentation may be necessary to attain the correct value. This method is provided as a starting point for the correct receive buffer size.

Setting the min, max, and default receive buffer (RX_WINDOW) size is performed in the same manner as single connection.

Driver Messages

The following messages are the most common messages logged by syslog. These may be found in /var/log/messages.

Driver up:

```
Chelsio Network Driver - version 2.1.1
```

NIC detected:

```
eth#: Chelsio N210 1x10GBaseX NIC (rev #), PCIX 133MHz/64-bit
```

Link up:

```
eth#: link is up at 10 Gbps, full duplex
```

Link down:

```
eth#: link is down
```

Known Issues

These issues have been identified during testing. The following information is provided as a workaround to the problem. In some cases, this problem is inherent to Linux or to a particular Linux Distribution and/or hardware platform.

1. Large number of TCP retransmits on a multiprocessor (SMP) system.

On a system with multiple CPUs, the interrupt (IRQ) for the network controller may be bound to more than one CPU. This will cause TCP retransmits if the packet data were to be split across different CPUs and re-assembled in a different order than expected.

To eliminate the TCP retransmits, set smp_affinity on the particular interrupt to a single CPU. You can locate the interrupt (IRQ) used on the N110/N210 by using ifconfig:

```
ifconfig <dev_name> | grep Interrupt
```

Set the smp_affinity to a single CPU:

```
echo 1 > /proc/irq/<interrupt_number>/smp_affinity
```

It is highly suggested that you do not run the irqbalance daemon on your system, as this will change any smp_affinity setting you have applied. The irqbalance daemon runs on a 10 second interval and binds interrupts to the least loaded CPU determined by the daemon. To disable this daemon:

```
chkconfig --level 2345 irqbalance off
```

By default, some Linux distributions enable the kernel feature, irqbalance, which performs the same function as the daemon. To disable this feature, add the following line to your bootloader:

```
noirqbalance
```

Example using the Grub bootloader::

```
title Red Hat Enterprise Linux AS (2.4.21-27.ELsmp)
root (hd0,0)
kernel /vmlinuz-2.4.21-27.ELsmp ro root=/dev/hda3
→noirqbalance
initrd /initrd-2.4.21-27.ELsmp.img
```

- After running insmod, the driver is loaded and the incorrect network interface is brought up without running ifup.

When using 2.4.x kernels, including RHEL kernels, the Linux kernel invokes a script named "hotplug". This script is primarily used to automatically bring up USB devices when they are plugged in, however, the script also attempts to automatically bring up a network interface after loading the kernel module. The hotplug script does this by scanning the ifcfg-eth# config files in /etc/sysconfig/network-scripts, looking for HWADDR=<mac_address>.

If the hotplug script does not find the HWADDR within any of the ifcfg-eth# files, it will bring up the device with the next available interface name. If this interface is already configured for a different network card, your new interface will have incorrect IP address and network settings.

To solve this issue, you can add the HWADDR=<mac_address> key to the interface config file of your network controller.

To disable this "hotplug" feature, you may add the driver (module name) to the "blacklist" file located in /etc/hotplug. It has been noted that this does not work for network devices because the net.agent script does not use the blacklist file. Simply remove, or rename, the net.agent script located in /etc/hotplug to disable this feature.

- Transport Protocol (TP) hangs when running heavy multi-connection traffic on an AMD Opteron system with HyperTransport PCI-X Tunnel chipset.

If your AMD Opteron system uses the AMD-8131 HyperTransport PCI-X Tunnel chipset, you may experience the "133-Mhz Mode Split Completion Data Corruption" bug identified by AMD while using a 133Mhz PCI-X card on the bus PCI-X bus.

AMD states, "Under highly specific conditions, the AMD-8131 PCI-X Tunnel can provide stale data via split completion cycles to a PCI-X card that is operating at 133 Mhz", causing data corruption.

AMD's provides three workarounds for this problem, however, Chelsio recommends the first option for best performance with this bug:

For 133Mhz secondary bus operation, limit the transaction length and the number of outstanding transactions, via BIOS configuration programming of the PCI-X card, to the following:

Data Length (bytes): 1k

Total allowed outstanding transactions: 2

Please refer to AMD 8131-HT/PCI-X Errata 26310 Rev 3.08 August 2004, section 56, "133-MHz Mode Split Completion Data Corruption" for more details with this bug and workarounds suggested by AMD.

It may be possible to work outside AMD's recommended PCI-X settings, try increasing the Data Length to 2k bytes for increased performance. If you have issues with these settings, please revert to the "safe" settings and duplicate the problem before submitting a bug or asking for support.

Note: The default setting on most systems is 8 outstanding transactions and 2k bytes data length.

4. On multiprocessor systems, it has been noted that an application which is handling 10Gb networking can switch between CPUs causing degraded and/or unstable performance.

If running on an SMP system and taking performance measurements, it is suggested you either run the latest netperf-2.4.0+ or use a binding tool such as Tim Hockin's procstate utilities (runon) <<http://www.hockin.org/~thockin/procstate/>>.

Binding netserver and netperf (or other applications) to particular CPUs will have a significant difference in performance measurements. You may need to experiment which CPU to bind the application to in order to achieve the best performance for your system.

If you are developing an application designed for 10Gb networking, please keep in mind you may want to look at kernel functions `sched_setaffinity` & `sched_getaffinity` to bind your application.

If you are just running user-space applications such as ftp, telnet, etc., you may want to try the runon tool provided by Tim Hockin's procstate utility. You could also try binding the interface to a particular CPU: runon 0 ifup eth0

Support

If you have problems with the software or hardware, please contact our customer support team via email at support@chelsio.com or check our website at <http://www.chelsio.com>

Chelsio Communications
370 San Aleso Ave.
Suite 100
Sunnyvale, CA 94085
<http://www.chelsio.com>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License, version 2, as published by the Free Software Foundation.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

THIS SOFTWARE IS PROVIDED AS IS AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Copyright © 2003-2005 Chelsio Communications. All rights reserved.

6.5.10 Cirrus Logic LAN CS8900/CS8920 Ethernet Adapters

Note: This document was contributed by Cirrus Logic for kernel 2.2.5. This version has been updated for 2.3.48 by Andrew Morton.

Still, this is too outdated! A major cleanup is needed here.

Cirrus make a copy of this driver available at their website, as described below. In general, you should use the driver version which comes with your Linux distribution.

Linux Network Interface Driver ver. 2.00 <kernel 2.3.48>

1. Cirrus Logic LAN CS8900/CS8920 Ethernet Adapters

1.1. Product Overview

The CS8900-based ISA Ethernet Adapters from Cirrus Logic follow IEEE 802.3 standards and support half or full-duplex operation in ISA bus computers on 10 Mbps Ethernet networks. The adapters are designed for operation in 16-bit ISA or EISA bus expansion slots and are available in 10BaseT-only or 3-media configurations (10BaseT, 10Base2, and AUI for 10Base-5 or fiber networks).

CS8920-based adapters are similar to the CS8900-based adapter with additional features for Plug and Play (PnP) support and Wakeup Frame recognition. As such, the configuration procedures differ somewhat between the two types of adapters. Refer to the "Adapter Configuration" section for details on configuring both types of adapters.

1.2. Driver Description

The CS8900/CS8920 Ethernet Adapter driver for Linux supports the Linux v2.3.48 or greater kernel. It can be compiled directly into the kernel or loaded at run-time as a device driver module.

1.2.1 Driver Name: cs89x0

1.2.2 Files in the Driver Archive:

The files in the driver at Cirrus' website include:

readme.txt	this file
build	batch file to compile cs89x0.c.
cs89x0.c	driver C code
cs89x0.h	driver header file
cs89x0.o	pre-compiled module (for v2.2.5 kernel)
config/Config.in	sample file to include cs89x0 driver in the kernel.
config/Makefile	sample file to include cs89x0 driver in the kernel.
config/Space.c	sample file to include cs89x0 driver in the kernel.

1.3. System Requirements

The following hardware is required:

- Cirrus Logic LAN (CS8900/20-based) Ethernet ISA Adapter
- IBM or IBM-compatible PC with:
 - * An 80386 or higher processor
 - * 16 bytes of contiguous IO space available between 210h - 370h
 - * One available IRQ (5,10,11,or 12 for the CS8900, 3-7,9-15 for CS8920).
- Appropriate cable (and connector for AUI, 10BASE-2) for your network topology.

The following software is required:

- LINUX kernel version 2.3.48 or higher
 - CS8900/20 Setup Utility (DOS-based)
 - LINUX kernel sources for your kernel (if compiling into kernel)
 - GNU Toolkit (gcc and make) v2.6 or above (if compiling into kernel or a module)

1.4. Licensing Information

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, version 1.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

For a full copy of the GNU General Public License, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

2. Adapter Installation and Configuration

Both the CS8900 and CS8920-based adapters can be configured using parameters stored in an on-board EEPROM. You must use the DOS-based CS8900/20 Setup Utility if you want to change the adapter's configuration in EEPROM.

When loading the driver as a module, you can specify many of the adapter's configuration parameters on the command-line to override the EEPROM's settings or for interface configuration when an EEPROM is not used. (CS8920-based adapters must use an EEPROM.) See Section 3.0 LOADING THE DRIVER AS A MODULE.

Since the CS8900/20 Setup Utility is a DOS-based application, you must install and configure the adapter in a DOS-based system using the CS8900/20 Setup Utility before installation in the target LINUX system. (Not required if installing a CS8900-based adapter and the default configuration is acceptable.)

2.1. CS8900-based Adapter Configuration

CS8900-based adapters shipped from Cirrus Logic have been configured with the following "default" settings:

Operation Mode:	Memory Mode
IRQ:	10
Base I/O Address:	300
Memory Base Address:	D0000
Optimization:	DOS Client
Transmission Mode:	Half-duplex
BootProm:	None
Media Type:	Autodetect (3-media cards) or 10BASE-T (10BASE-T only adapter)

You should only change the default configuration settings if conflicts with another adapter exists. To change the adapter's configuration, run the CS8900/20 Setup Utility.

2.2. CS8920-based Adapter Configuration

CS8920-based adapters are shipped from Cirrus Logic configured as Plug and Play (PnP) enabled. However, since the cs89x0 driver does NOT support PnP, you must install the CS8920 adapter in a DOS-based PC and run the CS8900/20 Setup Utility to disable PnP and configure the adapter before installation in the target Linux system. Failure to do this will leave the adapter inactive and the driver will be unable to communicate with the adapter.

```
*****
*          CS8920 -BASED ADAPTERS:          *
*          *****                                *
*  CS8920 -BASED ADAPTERS ARE PLUG and PLAY ENABLED BY DEFAULT.  *
*  THE CS89X0 DRIVER DOES NOT SUPPORT PnP. THEREFORE, YOU MUST   *
*  RUN THE CS8900/20 SETUP UTILITY TO DISABLE PnP SUPPORT AND     *
*  TO ACTIVATE THE ADAPTER.                                         *
*****
```

3. Loading the Driver as a Module

If the driver is compiled as a loadable module, you can load the driver module with the 'modprobe' command. Many of the adapter's configuration parameters can be specified as command-line arguments to the load command. This facility provides a means to override the EEPROM's settings or for interface configuration when an EEPROM is not used.

Example:

```
insmod cs89x0.o io=0x200 irq=0xA media=aui
```

This example loads the module and configures the adapter to use an IO port base address of 200h, interrupt 10, and use the AUI media connection. The following configuration options are available on the command line:

io=###	- specify IO address (200h-360h)
irq=##	- specify interrupt level
use_dma=1	- Enable DMA
dma=#	- specify dma channel (Driver is compiled to support Rx DMA only)
dmasize#= (16 or 64)	- DMA size 16K or 64K. Default value is set to 16.
media=rj45	- specify media type
or media=bnc	
or media=aui	
or media=auto	
duplex=full	- specify forced half/full/autonegotiate duplex
or duplex=half	
or duplex=auto	
debug=#	- debug level (only available if the driver was compiled for debugging)

Notes:

- a) If an EEPROM is present, any specified command-line parameter will override the corresponding configuration value stored in EEPROM.
- b) The "io" parameter must be specified on the command-line.
- c) The driver's hardware probe routine is designed to avoid writing to I/O space until it knows that there is a cs89x0 card at the written addresses. This could cause problems with device probing. To avoid this behaviour, add one to the io= module parameter. This doesn't actually change the I/O address, but it is a flag to tell the driver to partially initialise the hardware before trying to identify the card. This could be dangerous if you are not sure that there is a cs89x0 card at the provided address.

For example, to scan for an adapter located at IO base 0x300, specify an IO address of 0x301.

- d) The "duplex=auto" parameter is only supported for the CS8920.
- e) The minimum command-line configuration required if an EEPROM is not present is:
io irq media type (no autodetect)
- f) The following additional parameters are CS89XX defaults (values used with no EEPROM or command-line argument).

- DMA Burst = enabled
- IOCHRDY Enabled = enabled
- UseSA = enabled
- CS8900 defaults to half-duplex if not specified on command-line
- CS8920 defaults to autoneg if not specified on command-line
- Use reset defaults for other config parameters
- dma_mode = 0

- g) You can use ifconfig to set the adapter's Ethernet address.
- h) Many Linux distributions use the 'modprobe' command to load modules. This program uses the '/etc/conf.modules' file to determine configuration information which is passed to a driver module when it is loaded. All the configuration options which are described above may be placed within /etc/conf.modules.

For example:

```
> cat /etc/conf.modules
...
alias eth0 cs89x0
options cs89x0 io=0x0200 dma=5 use_dma=1
...
```

In this example we are telling the module system that the ethernet driver for this machine should use the cs89x0 driver. We are asking 'modprobe' to pass the 'io', 'dma' and 'use_dma' arguments to the driver when it is loaded.

- i) Cirrus recommend that the cs89x0 use the ISA DMA channels 5, 6 or 7. You will probably find that other DMA channels will not work.
- j) The cs89x0 supports DMA for receiving only. DMA mode is significantly more efficient. Flooding a 400 MHz Celeron machine with large ping packets consumes 82% of its CPU capacity in non-DMA mode. With DMA this is reduced to 45%.
- k) If your Linux kernel was compiled with inbuilt plug-and-play support you will be able to find information about the cs89x0 card with the command:

```
cat /proc/isapnp
```

- l) If during DMA operation you find erratic behavior or network data corruption you should use your PC's BIOS to slow the EISA bus clock.
- m) If the cs89x0 driver is compiled directly into the kernel (non-modular) then its I/O address is automatically determined by ISA bus probing. The IRQ number, media options, etc are determined from the card's EEPROM.
- n) If the cs89x0 driver is compiled directly into the kernel, DMA mode may be selected by providing the kernel with a boot option 'cs89x0_dma=N' where 'N' is the desired DMA channel number (5, 6 or 7).

Kernel boot options may be provided on the LILO command line:

```
LIL0 boot: linux cs89x0_dma=5
```

or they may be placed in /etc/lilo.conf:

```
image=/boot/bzImage-2.3.48
append="cs89x0_dma=5"
label=linux
root=/dev/hda5
read-only
```

The DMA Rx buffer size is hardwired to 16 kbytes in this mode. (64k mode is not available).

4. Compiling the Driver

The cs89x0 driver can be compiled directly into the kernel or compiled into a loadable device driver module.

Just use the standard way to configure the driver and compile the Kernel.

4.1. Compiling the Driver to Support Rx DMA

The compile-time optionality for DMA was removed in the 2.3 kernel series. DMA support is now unconditionally part of the driver. It is enabled by the 'use_dma=1' module option.

5. Testing and Troubleshooting

5.1. Known Defects and Limitations

Refer to the RELEASE.TXT file distributed as part of this archive for a list of known defects, driver limitations, and work arounds.

5.2. Testing the Adapter

Once the adapter has been installed and configured, the diagnostic option of the CS8900/20 Setup Utility can be used to test the functionality of the adapter and its network connection. Use the diagnostics 'Self Test' option to test the functionality of the adapter with the hardware configuration you have assigned. You can use the diagnostics 'Network Test' to test the ability of the adapter to communicate across the Ethernet with another PC equipped with a CS8900/20-based adapter card (it must also be running the CS8900/20 Setup Utility).

Note: The Setup Utility's diagnostics are designed to run in a DOS-only operating system environment. DO NOT run the diagnostics from a DOS or command prompt session under Windows 95, Windows NT, OS/2, or other operating system.

To run the diagnostics tests on the CS8900/20 adapter:

1. Boot DOS on the PC and start the CS8900/20 Setup Utility.

2. The adapter's current configuration is displayed. Hit the ENTER key to get to the main menu.
4. Select 'Diagnostics' (ALT-G) from the main menu. * Select 'Self-Test' to test the adapter's basic functionality. * Select 'Network Test' to test the network connection and cabling.

5.2.1. Diagnostic Self-test

The diagnostic self-test checks the adapter's basic functionality as well as its ability to communicate across the ISA bus based on the system resources assigned during hardware configuration. The following tests are performed:

- IO Register Read/Write Test

The IO Register Read/Write test insures that the CS8900/20 can be accessed in IO mode, and that the IO base address is correct.

- Shared Memory Test

The Shared Memory test insures the CS8900/20 can be accessed in memory mode and that the range of memory addresses assigned does not conflict with other devices in the system.

- Interrupt Test

The Interrupt test insures there are no conflicts with the assigned IRQ signal.

- EEPROM Test

The EEPROM test insures the EEPROM can be read.

- Chip RAM Test

The Chip RAM test insures the 4K of memory internal to the CS8900/20 is working properly.

- Internal Loop-back Test

The Internal Loop Back test insures the adapter's transmitter and receiver are operating properly. If this test fails, make sure the adapter's cable is connected to the network (check for LED activity for example).

- Boot PROM Test

The Boot PROM test insures the Boot PROM is present, and can be read. Failure indicates the Boot PROM was not successfully read due to a hardware problem or due to a conflicts on the Boot PROM address assignment. (Test only applies if the adapter is configured to use the Boot PROM option.)

Failure of a test item indicates a possible system resource conflict with another device on the ISA bus. In this case, you should use the Manual Setup option to reconfigure the adapter by selecting a different value for the system resource that failed.

5.2.2. Diagnostic Network Test

The Diagnostic Network Test verifies a working network connection by transferring data between two CS8900/20 adapters installed in different PCs on the same network. (Note: the diagnostic network test should not be run between two nodes across a router.)

This test requires that each of the two PCs have a CS8900/20-based adapter installed and have the CS8900/20 Setup Utility running. The first PC is configured as a Responder and the other PC is configured as an Initiator. Once the Initiator is started, it sends data frames to the Responder which returns the frames to the Initiator.

The total number of frames received and transmitted are displayed on the Initiator's display, along with a count of the number of frames received and transmitted OK or in error. The test can be terminated anytime by the user at either PC.

To setup the Diagnostic Network Test:

1. Select a PC with a CS8900/20-based adapter and a known working network connection to act as the Responder. Run the CS8900/20 Setup Utility and select 'Diagnostics -> Network Test -> Responder' from the main menu. Hit ENTER to start the Responder.
2. Return to the PC with the CS8900/20-based adapter you want to test and start the CS8900/20 Setup Utility.
3. From the main menu, Select 'Diagnostic -> Network Test -> Initiator'. Hit ENTER to start the test.

You may stop the test on the Initiator at any time while allowing the Responder to continue running. In this manner, you can move to additional PCs and test them by starting the Initiator on another PC without having to stop/start the Responder.

5.3. Using the Adapter's LEDs

The 2 and 3-media adapters have two LEDs visible on the back end of the board located near the 10Base-T connector.

Link Integrity LED: A "steady" ON of the green LED indicates a valid 10Base-T connection. (Only applies to 10Base-T. The green LED has no significance for a 10Base-2 or AUI connection.)

TX/RX LED: The yellow LED lights briefly each time the adapter transmits or receives data. (The yellow LED will appear to "flicker" on a typical network.)

5.4. Resolving I/O Conflicts

An IO conflict occurs when two or more adapter use the same ISA resource (IO address, memory address or IRQ). You can usually detect an IO conflict in one of four ways after installing and or configuring the CS8900/20-based adapter:

1. The system does not boot properly (or at all).
2. The driver cannot communicate with the adapter, reporting an "Adapter not found" error message.
3. You cannot connect to the network or the driver will not load.

4. If you have configured the adapter to run in memory mode but the driver reports it is using IO mode when loading, this is an indication of a memory address conflict.

If an IO conflict occurs, run the CS8900/20 Setup Utility and perform a diagnostic self-test. Normally, the ISA resource in conflict will fail the self-test. If so, reconfigure the adapter selecting another choice for the resource in conflict. Run the diagnostics again to check for further IO conflicts.

In some cases, such as when the PC will not boot, it may be necessary to remove the adapter and reconfigure it by installing it in another PC to run the CS8900/20 Setup Utility. Once reinstalled in the target system, run the diagnostics self-test to ensure the new configuration is free of conflicts before loading the driver again.

When manually configuring the adapter, keep in mind the typical ISA system resource usage as indicated in the tables below.

I/O Address	Device	IRQ	Device
200-20F	Game I/O adapter	3	COM2, Bus Mouse
230-23F	Bus Mouse	4	COM1
270-27F	LPT3: third parallel port	5	LPT2
2F0-2FF	COM2: second serial port	6	Floppy Disk
controller			
320-32F	Fixed disk controller	7	LPT1
		8	Real-time Clock
		9	EGA/VGA display
adapter			
		12	Mouse (PS/2)
Memory Address	Device	13	Math Coprocessor
		14	Hard Disk controller
A000-BFFF	EGA Graphics Adapter		
A000-C7FF	VGA Graphics Adapter		
B000-BFFF	Mono Graphics Adapter		
B800-BFFF	Color Graphics Adapter		
E000-FFFF	AT BIOS		

6. Technical Support

6.1. Contacting Cirrus Logic's Technical Support

Cirrus Logic's CS89XX Technical Application Support can be reached at:

Telephone	: (800) 888-5016 (from inside U.S. and Canada) :(512) 442-7555 (from outside the U.S. and Canada)
Fax	: (512) 912-3871
Email	: ethernet@crystal.cirrus.com
WWW	: http://www.cirrus.com

6.2. Information Required before Contacting Technical Support

Before contacting Cirrus Logic for technical support, be prepared to provide as much of the following information as possible.

- 1.) Adapter type (CRD8900, CDB8900, CDB8920, etc.)
- 2.) Adapter configuration
 - IO Base, Memory Base, IO or memory mode enabled, IRQ, DMA channel
 - Plug and Play enabled/disabled (CS8920-based adapters only)
 - Configured for media auto-detect or specific media type (which type).
- 3.) PC System's Configuration
 - Plug and Play system (yes/no)
 - BIOS (make and version)
 - System make and model
 - CPU (type and speed)
 - System RAM
 - SCSI Adapter
- 4.) Software
 - CS89XX driver and version
 - Your network operating system and version
 - Your system's OS version
 - Version of all protocol support files
- 5.) Any Error Message displayed.

6.3 Obtaining the Latest Driver Version

You can obtain the latest CS89XX drivers and support software from Cirrus Logic's Web site. You can also contact Cirrus Logic's Technical Support (email: ethernet@crystal.cirrus.com) and request that you be registered for automatic software-update notification.

Cirrus Logic maintains a web page at <http://www.cirrus.com> with the latest drivers and technical publications.

6.4. Current maintainer

In February 2000 the maintenance of this driver was assumed by Andrew Morton.

6.5 Kernel module parameters

For use in embedded environments with no cs89x0 EEPROM, the kernel boot parameter `cs89x0_media=` has been implemented. Usage is:

```
cs89x0_media=rj45      or  
cs89x0_media=aui       or  
cs89x0_media=bnc
```

6.5.11 D-Link DL2000-based Gigabit Ethernet Adapter Installation

May 23, 2002

Compatibility List

Adapter Support:

- D-Link DGE-550T Gigabit Ethernet Adapter.
- D-Link DGE-550SX Gigabit Ethernet Adapter.
- D-Link DL2000-based Gigabit Ethernet Adapter.

The driver support Linux kernel 2.4.7 later. We had tested it on the environments below.

- . Red Hat v6.2 (update kernel to 2.4.7) . Red Hat v7.0 (update kernel to 2.4.7) . Red Hat v7.1 (kernel 2.4.7) . Red Hat v7.2 (kernel 2.4.7-10)

Quick Install

Install linux driver as following command:

```
1. make all  
2. insmod dl2k.ko  
3. ifconfig eth0 up 10.xxx.xxx.xxx netmask 255.0.0.0  
                                ^^^^^^\\          ^^^^^^\\  
                                IP           NETMASK
```

Now eth0 should active, you can test it by "ping" or get more information by "ifconfig". If tested ok, continue the next step.

4. cp dl2k.ko /lib/modules/`uname -r`/kernel/drivers/net
5. Add the following line to /etc/modprobe.d/dl2k.conf:

```
alias eth0 dl2k
```

6. Run depmod to updated module indexes.

7. Run `netconfig` or `netconf` to create configuration script `ifcfg-eth0` located at `/etc/sysconfig/network-scripts` or create it manually.
[see - Configuration Script Sample]
8. Driver will automatically load and configure at next boot time.

Compiling the Driver

In Linux, NIC drivers are most commonly configured as loadable modules. The approach of building a monolithic kernel has become obsolete. The driver can be compiled as part of a monolithic kernel, but is strongly discouraged. The remainder of this section assumes the driver is built as a loadable module. In the Linux environment, it is a good idea to rebuild the driver from the source instead of relying on a precompiled version. This approach provides better reliability since a precompiled driver might depend on libraries or kernel features that are not present in a given Linux installation.

The 3 files necessary to build Linux device driver are `dl2k.c`, `dl2k.h` and `Makefile`. To compile, the Linux installation must include the `gcc` compiler, the kernel source, and the kernel headers. The Linux driver supports Linux Kernels 2.4.7. Copy the files to a directory and enter the following command to compile and link the driver:

CD-ROM drive

```
[root@XXX /] mkdir cdrom
[root@XXX /] mount -r -t iso9660 -o conv=auto /dev/cdrom /cdrom
[root@XXX /] cd root
[root@XXX /root] mkdir dl2k
[root@XXX /root] cd dl2k
[root@XXX dl2k] cp /cdrom/linux/dl2k.tgz /root/dl2k
[root@XXX dl2k] tar xfvz dl2k.tgz
[root@XXX dl2k] make all
```

Floppy disc drive

```
[root@XXX /] cd root
[root@XXX /root] mkdir dl2k
[root@XXX /root] cd dl2k
[root@XXX dl2k] mcopy a:/linux/dl2k.tgz /root/dl2k
[root@XXX dl2k] tar xfvz dl2k.tgz
[root@XXX dl2k] make all
```

Installing the Driver

Manual Installation

Once the driver has been compiled, it must be loaded, enabled, and bound to a protocol stack in order to establish network connectivity. To load a module enter the command:

```
insmod dl2k.o
```

or:

```
insmod dl2k.o <optional parameter> ; add parameter
```

example:

```
insmod dl2k.o media=100mbps_hd
```

or::

```
insmod dl2k.o media=3
```

or::

```
insmod dl2k.o media=3,2      ; for 2 cards
```

Please reference the list of the command line parameters supported by the Linux device driver below.

The insmod command only loads the driver and gives it a name of the form eth0, eth1, etc. To bring the NIC into an operational state, it is necessary to issue the following command:

```
ifconfig eth0 up
```

Finally, to bind the driver to the active protocol (e.g., TCP/IP with Linux), enter the following command:

```
ifup eth0
```

Note that this is meaningful only if the system can find a configuration script that contains the necessary network information. A sample will be given in the next paragraph.

The commands to unload a driver are as follows:

```
ifdown eth0  
ifconfig eth0 down  
rmmod dl2k.o
```

The following are the commands to list the currently loaded modules and to see the current network configuration:

```
lsmod  
ifconfig
```

Automated Installation

This section describes how to install the driver such that it is automatically loaded and configured at boot time. The following description is based on a Red Hat 6.0/7.0 distribution, but it can easily be ported to other distributions as well.

Red Hat v6.x/v7.x

1. Copy dl2k.o to the network modules directory, typically /lib/modules/2.x.x-xx/net or /lib/modules/2.x.x/kernel/drivers/net.
2. Locate the boot module configuration file, most commonly in the /etc/modprobe.d/ directory. Add the following lines:

```
alias ethx dl2k  
options dl2k <optional parameters>
```

where ethx will be eth0 if the NIC is the only ethernet adapter, eth1 if one other ethernet adapter is installed, etc. Refer to the table in the previous section for the list of optional parameters.

3. Locate the network configuration scripts, normally the /etc/sysconfig/network-scripts directory, and create a configuration script named ifcfg-ethx that contains network information.
4. Note that for most Linux distributions, Red Hat included, a configuration utility with a graphical user interface is provided to perform steps 2 and 3 above.

Parameter Description

You can install this driver without any additional parameter. However, if you are going to have extensive functions then it is necessary to set extra parameter. Below is a list of the command line parameters supported by the Linux device driver.

mtu=packet_size	Specifies the maximum packet size. default is 1500.																										
media=media_type	Specifies the media type the NIC operates at. autosense Autosensing active media.																										
	<table border="1"> <tr><td>10mbps_hd</td><td>10Mbps half duplex.</td></tr> <tr><td>10mbps_fd</td><td>10Mbps full duplex.</td></tr> <tr><td>100mbps_hd</td><td>100Mbps half duplex.</td></tr> <tr><td>100mbps_fd</td><td>100Mbps full duplex.</td></tr> <tr><td>1000mbps_fd</td><td>1000Mbps full duplex.</td></tr> <tr><td>1000mbps_hd</td><td>1000Mbps half duplex.</td></tr> <tr><td>0</td><td>Autosensing active media.</td></tr> <tr><td>1</td><td>10Mbps half duplex.</td></tr> <tr><td>2</td><td>10Mbps full duplex.</td></tr> <tr><td>3</td><td>100Mbps half duplex.</td></tr> <tr><td>4</td><td>100Mbps full duplex.</td></tr> <tr><td>5</td><td>1000Mbps half duplex.</td></tr> <tr><td>6</td><td>1000Mbps full duplex.</td></tr> </table>	10mbps_hd	10Mbps half duplex.	10mbps_fd	10Mbps full duplex.	100mbps_hd	100Mbps half duplex.	100mbps_fd	100Mbps full duplex.	1000mbps_fd	1000Mbps full duplex.	1000mbps_hd	1000Mbps half duplex.	0	Autosensing active media.	1	10Mbps half duplex.	2	10Mbps full duplex.	3	100Mbps half duplex.	4	100Mbps full duplex.	5	1000Mbps half duplex.	6	1000Mbps full duplex.
10mbps_hd	10Mbps half duplex.																										
10mbps_fd	10Mbps full duplex.																										
100mbps_hd	100Mbps half duplex.																										
100mbps_fd	100Mbps full duplex.																										
1000mbps_fd	1000Mbps full duplex.																										
1000mbps_hd	1000Mbps half duplex.																										
0	Autosensing active media.																										
1	10Mbps half duplex.																										
2	10Mbps full duplex.																										
3	100Mbps half duplex.																										
4	100Mbps full duplex.																										
5	1000Mbps half duplex.																										
6	1000Mbps full duplex.																										
	By default, the NIC operates at autosense. 1000mbps_fd and 1000mbps_hd types are only available for fiber adapter.																										
vlan=n	Specifies the VLAN ID. If vlan=0, the Virtual Local Area Network (VLAN) function is disable.																										
jumbo=[0 1]	Specifies the jumbo frame support. If jumbo=1, the NIC accept jumbo frames. By default, this function is disabled. Jumbo frame usually improve the performance int gigabit. This feature need jumbo frame compatible remote.																										
rx_coalesce=m	Number of rx frame handled each interrupt.																										
rx_timeout=n	Rx DMA wait time for an interrupt. If set rx_coalesce > 0, hardware only assert an interrupt for m frames. Hardware won't assert rx interrupt until m frames received or reach timeout of n * 640 nano seconds. Set proper rx_coalesce and rx_timeout can reduce congestion collapse and overload which has been a bottleneck for high speed network. For example, rx_coalesce=10 rx_timeout=800. that is, hardware assert only 1 interrupt for 10 frames received or timeout of 512 us.																										
tx_coalesce=n	Number of tx frame handled each interrupt. Set n > 1 can reduce the interrupts conges- tion usually lower performance of high speed network card. Default is 16.																										
tx_flow=[1 0]	Specifies the Tx flow control. If tx_flow=0, the Tx flow control disable else driver autode- tect.																										
rx_flow=[1 0]	Specifies the Rx flow control. If rx_flow=0, the Rx flow control disable else driver auto- detect.																										

Configuration Script Sample

Here is a sample of a simple configuration script:

```
DEVICE=eth0
USERCTL=no
ONBOOT=yes
P0OTPR0T0=none
BROADCAST=207.200.5.255
NETWORK=207.200.5.0
NETMASK=255.255.255.0
IPADDR=207.200.5.2
```

Troubleshooting

Q1. Source files contain ^ M behind every line.

Make sure all files are Unix file format (no LF). Try the following shell command to convert files:

```
cat dl2k.c | col -b > dl2k.tmp
mv dl2k.tmp dl2k.c
```

OR:

```
cat dl2k.c | tr -d "\r" > dl2k.tmp
mv dl2k.tmp dl2k.c
```

Q2: Could not find header files (*.h)?

To compile the driver, you need kernel header files. After installing the kernel source, the header files are usually located in /usr/src/linux/include, which is the default include directory configured in Makefile. For some distributions, there is a copy of header files in /usr/src/include/linux and /usr/src/include/asm, that you can change the INCLUDEDIR in Makefile to /usr/include without installing kernel source.

Note that RH 7.0 didn't provide correct header files in /usr/include, including those files will make a wrong version driver.

6.5.12 DM9000 Network driver

Copyright 2008 Simtec Electronics,

Ben Dooks <ben@simtec.co.uk> <ben-linux@fluff.org>

Introduction

This file describes how to use the DM9000 platform-device based network driver that is contained in the files drivers/net/dm9000.c and drivers/net/dm9000.h.

The driver supports three DM9000 variants, the DM9000E which is the first chip supported as well as the newer DM9000A and DM9000B devices. It is currently maintained and tested by Ben Dooks, who should be CC: to any patches for this driver.

Defining the platform device

The minimum set of resources attached to the platform device are as follows:

- 1) The physical address of the address register
- 2) The physical address of the data register
- 3) The IRQ line the device's interrupt pin is connected to.

These resources should be specified in that order, as the ordering of the two address regions is important (the driver expects these to be address and then data).

An example from arch/arm/mach-s3c/mach-bast.c is:

```
static struct resource bast_dm9k_resource[] = {
    [0] = {
        .start = S3C2410_CS5 + BAST_PA_DM9000,
        .end   = S3C2410_CS5 + BAST_PA_DM9000 + 3,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = S3C2410_CS5 + BAST_PA_DM9000 + 0x40,
        .end   = S3C2410_CS5 + BAST_PA_DM9000 + 0x40 + 0x3f,
        .flags = IORESOURCE_MEM,
    },
    [2] = {
        .start = IRQ_DM9000,
        .end   = IRQ_DM9000,
        .flags = IORESOURCE_IRQ | IORESOURCE_IRQ_HIGHLEVEL,
    }
};

static struct platform_device bast_device_dm9k = {
    .name          = "dm9000",
    .id            = 0,
    .num_resources = ARRAY_SIZE(bast_dm9k_resource),
    .resource      = bast_dm9k_resource,
};
```

Note the setting of the IRQ trigger flag in bast_dm9k_resource[2].flags, as this will generate a warning if it is not present. The trigger from the flags field will be passed to request_irq() when registering the IRQ handler to ensure that the IRQ is setup correctly.

This shows a typical platform device, without the optional configuration platform data supplied. The next example uses the same resources, but adds the optional platform data to pass extra configuration data:

```
static struct dm9000_plat_data bast_dm9k_platdata = {
    .flags          = DM9000_PLATF_16BITONLY,
};

static struct platform_device bast_device_dm9k = {
    .name           = "dm9000",
    .id             = 0,
    .num_resources  = ARRAY_SIZE(bast_dm9k_resource),
    .resource       = bast_dm9k_resource,
    .dev            = {
        .platform_data = &bast_dm9k_platdata,
    }
};
```

The platform data is defined in include/linux/dm9000.h and described below.

Platform data

Extra platform data for the DM9000 can describe the IO bus width to the device, whether or not an external PHY is attached to the device and the availability of an external configuration EEPROM.

The flags for the platform data .flags field are as follows:

DM9000_PLATF_8BITONLY

The IO should be done with 8bit operations.

DM9000_PLATF_16BITONLY

The IO should be done with 16bit operations.

DM9000_PLATF_32BITONLY

The IO should be done with 32bit operations.

DM9000_PLATF_EXT_PHY

The chip is connected to an external PHY.

DM9000_PLATF_NO_EEPROM

This can be used to signify that the board does not have an EEPROM, or that the EEPROM should be hidden from the user.

DM9000_PLATF_SIMPLE_PHY

Switch to using the simpler PHY polling method which does not try and read the MII PHY state regularly. This is only available when using the internal PHY. See the section on link state polling for more information.

The config symbol **DM9000_FORCE_SIMPLE_PHY_POLL**, Kconfig entry "Force simple NSR based PHY polling" allows this flag to be forced on at build time.

PHY Link state polling

The driver keeps track of the link state and informs the network core about link (carrier) availability. This is managed by several methods depending on the version of the chip and on which PHY is being used.

For the internal PHY, the original (and currently default) method is to read the MII state, either when the status changes if we have the necessary interrupt support in the chip or every two seconds via a periodic timer.

To reduce the overhead for the internal PHY, there is now the option of using the DM9000_FORCE_SIMPLE_PHY_POLL config, or DM9000_PLATF_SIMPLE_PHY platform data option to read the summary information without the expensive MII accesses. This method is faster, but does not print as much information.

When using an external PHY, the driver currently has to poll the MII link status as there is no method for getting an interrupt on link change.

DM9000A / DM9000B

These chips are functionally similar to the DM9000E and are supported easily by the same driver. The features are:

- 1) Interrupt on internal PHY state change. This means that the periodic polling of the PHY status may be disabled on these devices when using the internal PHY.
- 2) TCP/UDP checksum offloading, which the driver does not currently support.

ethtool

The driver supports the ethtool interface for access to the driver state information, the PHY state and the EEPROM.

6.5.13 Davicom DM9102(A)/DM9132/DM9801 fast ethernet driver for Linux

Note: This driver doesn't have a maintainer.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

This driver provides kernel support for Davicom DM9102(A)/DM9132/DM9801 ethernet cards (CNET 10/100 ethernet cards uses Davicom chipset too, so this driver supports CNET cards too).If you didn't compile this driver as a module, it will automatically load itself on boot and print a line similar to:

```
dmfe: Davicom DM9xxx net driver, version 1.36.4 (2002-01-17)
```

If you compiled this driver as a module, you have to load it on boot. You can load it with command:

```
insmod dmfe
```

This way it will autodetect the device mode. This is the suggested way to load the module. Or you can pass a mode= setting to module while loading, like:

```
insmod dmfe mode=0 # Force 10M Half Duplex
insmod dmfe mode=1 # Force 100M Half Duplex
insmod dmfe mode=4 # Force 10M Full Duplex
insmod dmfe mode=5 # Force 100M Full Duplex
```

Next you should configure your network interface with a command similar to:

```
ifconfig eth0 172.22.3.18
               ^^^^^^^^^^
               Your IP Address
```

Then you may have to modify the default routing table with command:

```
route add default eth0
```

Now your ethernet card should be up and running.

TODO:

- Implement pci_driver::suspend() and pci_driver::resume() power management methods.
- Check on 64 bit boxes.
- Check and fix on big endian boxes.
- Test and make sure PCI latency is now correct for all cases.

Authors:

Sten Wang <sten_wang@davicom.com.tw> : Original Author

Contributors:

- Marcelo Tosatti <marcelo@conectiva.com.br>
- Alan Cox <alan@lxorguk.ukuu.org.uk>
- Jeff Garzik <jgarzik@pobox.com>
- Vojtech Pavlik <vojtech@suse.cz>

6.5.14 The QorIQ DPAA Ethernet Driver

Authors: - Madalin Bucur <madalin.bucur@nxp.com> - Camelia Groza <camelia.groza@nxp.com>

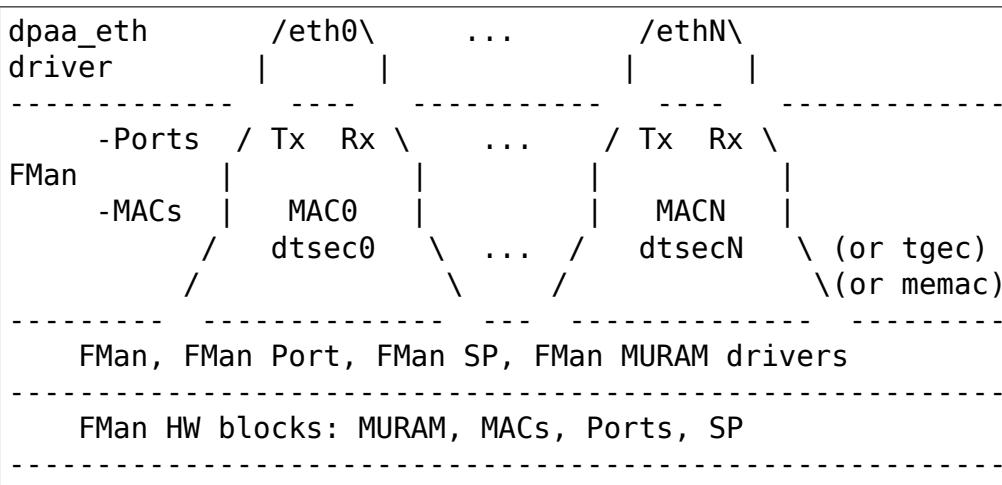
DPAA Ethernet Overview

DPAA stands for Data Path Acceleration Architecture and it is a set of networking acceleration IPs that are available on several generations of SoCs, both on PowerPC and ARM64.

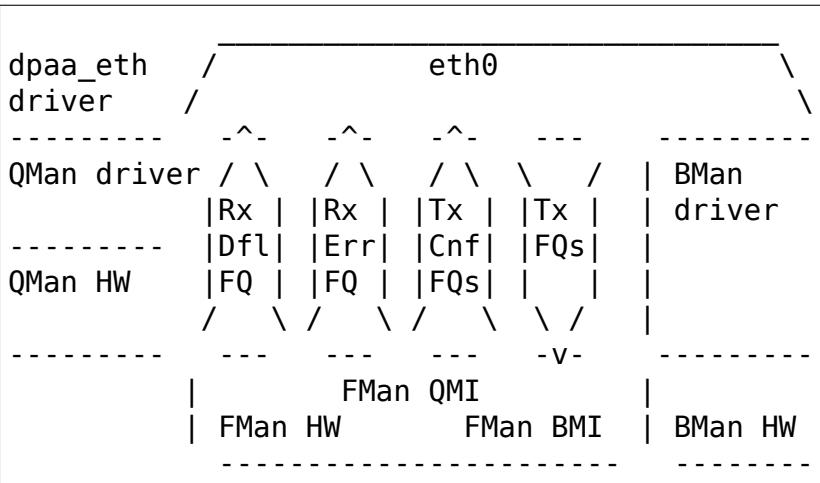
The Freescale DPAA architecture consists of a series of hardware blocks that support Ethernet connectivity. The Ethernet driver depends upon the following drivers in the Linux kernel:

- **Peripheral Access Memory Unit (PAMU) (* needed only for PPC platforms)**
drivers/iommu/fsl_*
 - **Frame Manager (FMan)**
drivers/net/ethernet/freescale/fman
 - **Queue Manager (QMan), Buffer Manager (BMan)**
drivers/soc/fsl/qbman

A simplified view of the dpaa_eth interfaces mapped to FMan MACs:



The dpaa eth relation to the QMan, BMan and FMan:



where the acronyms used above (and in the code) are:

DPAA	Data Path Acceleration Architecture
FMan	DPAA Frame Manager
QMan	DPAA Queue Manager
BMan	DPAA Buffers Manager
QMI	QMan interface in FMan
BMI	BMan interface in FMan
FMan SP	FMan Storage Profiles
MURAM	Multi-user RAM in FMan
FQ	QMan Frame Queue
Rx Dfl FQ	default reception FQ
Rx Err FQ	Rx error frames FQ
Tx Cnf FQ	Tx confirmation FQs
Tx FQs	transmission frame queues
dtsec	datapath three speed Ethernet controller (10/100/1000 Mbps)
tgec	ten gigabit Ethernet controller (10 Gbps)
memac	multirate Ethernet MAC (10/100/1000/10000)

DPAA Ethernet Supported SoCs

The DPAA drivers enable the Ethernet controllers present on the following SoCs:

PPC - P1023 - P2041 - P3041 - P4080 - P5020 - P5040 - T1023 - T1024 - T1040 - T1042 - T2080
- T4240 - B4860

ARM - LS1043A - LS1046A

Configuring DPAA Ethernet in your kernel

To enable the DPAA Ethernet driver, the following Kconfig options are required:

```
# common for arch/arm64 and arch/powerpc platforms
CONFIG_FSL_DPAA=y
CONFIG_FSL_FMAN=y
CONFIG_FSL_DPAA_ETH=y
CONFIG_FSL_XGMAC_MDIO=y

# for arch/powerpc only
CONFIG_FSL_PAMU=y

# common options needed for the PHYs used on the RDBs
CONFIG_VITESSE_PHY=y
CONFIG_REALTEK_PHY=y
CONFIG_AQUANTIA_PHY=y
```

DPAA Ethernet Frame Processing

On Rx, buffers for the incoming frames are retrieved from the buffers found in the dedicated interface buffer pool. The driver initializes and seeds these with one page buffers.

On Tx, all transmitted frames are returned to the driver through Tx confirmation frame queues. The driver is then responsible for freeing the buffers. In order to do this properly, a backpointer is added to the buffer before transmission that points to the skb. When the buffer returns to the driver on a confirmation FQ, the skb can be correctly consumed.

DPAA Ethernet Features

Currently the DPAA Ethernet driver enables the basic features required for a Linux Ethernet driver. The support for advanced features will be added gradually.

The driver has Rx and Tx checksum offloading for UDP and TCP. Currently the Rx checksum offload feature is enabled by default and cannot be controlled through ethtool. Also, rx-flow-hash and rx-hashing was added. The addition of RSS provides a big performance boost for the forwarding scenarios, allowing different traffic flows received by one interface to be processed by different CPUs in parallel.

The driver has support for multiple prioritized Tx traffic classes. Priorities range from 0 (lowest) to 3 (highest). These are mapped to HW workqueues with strict priority levels. Each traffic class contains NR_CPU TX queues. By default, only one traffic class is enabled and the lowest priority Tx queues are used. Higher priority traffic classes can be enabled with the mqpriority qdisc. For example, all four traffic classes are enabled on an interface with the following command. Furthermore, skb priority levels are mapped to traffic classes as follows:

- priorities 0 to 3 - traffic class 0 (low priority)
- priorities 4 to 7 - traffic class 1 (medium-low priority)
- priorities 8 to 11 - traffic class 2 (medium-high priority)
- priorities 12 to 15 - traffic class 3 (high priority)

```
tc qdisc add dev <int> root handle 1: \
    mqpriority num_tc 4 map 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3 hw 1
```

DPAA IRQ Affinity and Receive Side Scaling

Traffic coming on the DPAA Rx queues or on the DPAA Tx confirmation queues is seen by the CPU as ingress traffic on a certain portal. The DPAA QMan portal interrupts are affined each to a certain CPU. The same portal interrupt services all the QMan portal consumers.

By default the DPAA Ethernet driver enables RSS, making use of the DPAA FMan Parser and Keygen blocks to distribute traffic on 128 hardware frame queues using a hash on IP v4/v6 source and destination and L4 source and destination ports, in present in the received frame. When RSS is disabled, all traffic received by a certain interface is received on the default Rx frame queue. The default DPAA Rx frame queues are configured to put the received traffic into a pool channel that allows any available CPU portal to dequeue the ingress traffic. The default frame queues have the HOLDACTIVE option set, ensuring that traffic bursts from a certain queue are serviced by the same CPU. This ensures a very low rate of frame reordering.

A drawback of this is that only one CPU at a time can service the traffic received by a certain interface when RSS is not enabled.

To implement RSS, the DPAA Ethernet driver allocates an extra set of 128 Rx frame queues that are configured to dedicated channels, in a round-robin manner. The mapping of the frame queues to CPUs is now hardcoded, there is no indirection table to move traffic for a certain FQ (hash result) to another CPU. The ingress traffic arriving on one of these frame queues will arrive at the same portal and will always be processed by the same CPU. This ensures intra-flow order preservation and workload distribution for multiple traffic flows.

RSS can be turned off for a certain interface using ethtool, i.e.:

```
# ethtool -N fm1-mac9 rx-flow-hash tcp4 ""
```

To turn it back on, one needs to set rx-flow-hash for tcp4/6 or udp4/6:

```
# ethtool -N fm1-mac9 rx-flow-hash udp4 sfdn
```

There is no independent control for individual protocols, any command run for one of tcp4|udp4|ah4|esp4|sctp4|tcp6|udp6|ah6|esp6|sctp6 is going to control the rx-flow-hashing for all protocols on that interface.

Besides using the FMan Keygen computed hash for spreading traffic on the 128 Rx FQs, the DPAA Ethernet driver also sets the skb hash value when the NETIF_F_RXHASH feature is on (active by default). This can be turned on or off through ethtool, i.e.:

```
# ethtool -K fm1-mac9 rx-hashing off
# ethtool -k fm1-mac9 | grep hash
receive-hashing: off
# ethtool -K fm1-mac9 rx-hashing on
Actual changes:
receive-hashing: on
# ethtool -k fm1-mac9 | grep hash
receive-hashing: on
```

Please note that Rx hashing depends upon the rx-flow-hashing being on for that interface - turning off rx-flow-hashing will also disable the rx-hashing (without ethtool reporting it as off as that depends on the NETIF_F_RXHASH feature flag).

Debugging

The following statistics are exported for each interface through ethtool:

- interrupt count per CPU
- Rx packets count per CPU
- Tx packets count per CPU
- Tx confirmed packets count per CPU
- Tx S/G frames count per CPU
- Tx error count per CPU
- Rx error count per CPU

- Rx error count per type
- congestion related statistics:
 - congestion status
 - time spent in congestion
 - number of time the device entered congestion
 - dropped packets count per cause

The driver also exports the following information in sysfs:

- the FQ IDs for each FQ type /sys/devices/platform/soc/<addr>.fman/<addr>.ethernet/dpaa-ethernet.<id>/net/fm<nr>-mac<nr>/fqids
- the ID of the buffer pool in use /sys/devices/platform/soc/<addr>.fman/<addr>.ethernet/dpaa-ethernet.<id>/net/fm<nr>-mac<nr>/bpids

6.5.15 DPAA2 Documentation

DPAA2 (Data Path Acceleration Architecture Gen2) Overview

Copyright

© 2015 Freescale Semiconductor Inc.

Copyright

© 2018 NXP

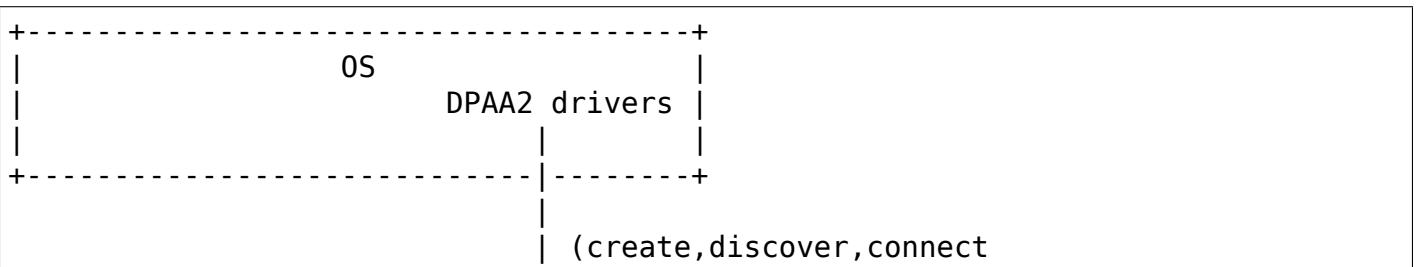
This document provides an overview of the Freescale DPAA2 architecture and how it is integrated into the Linux kernel.

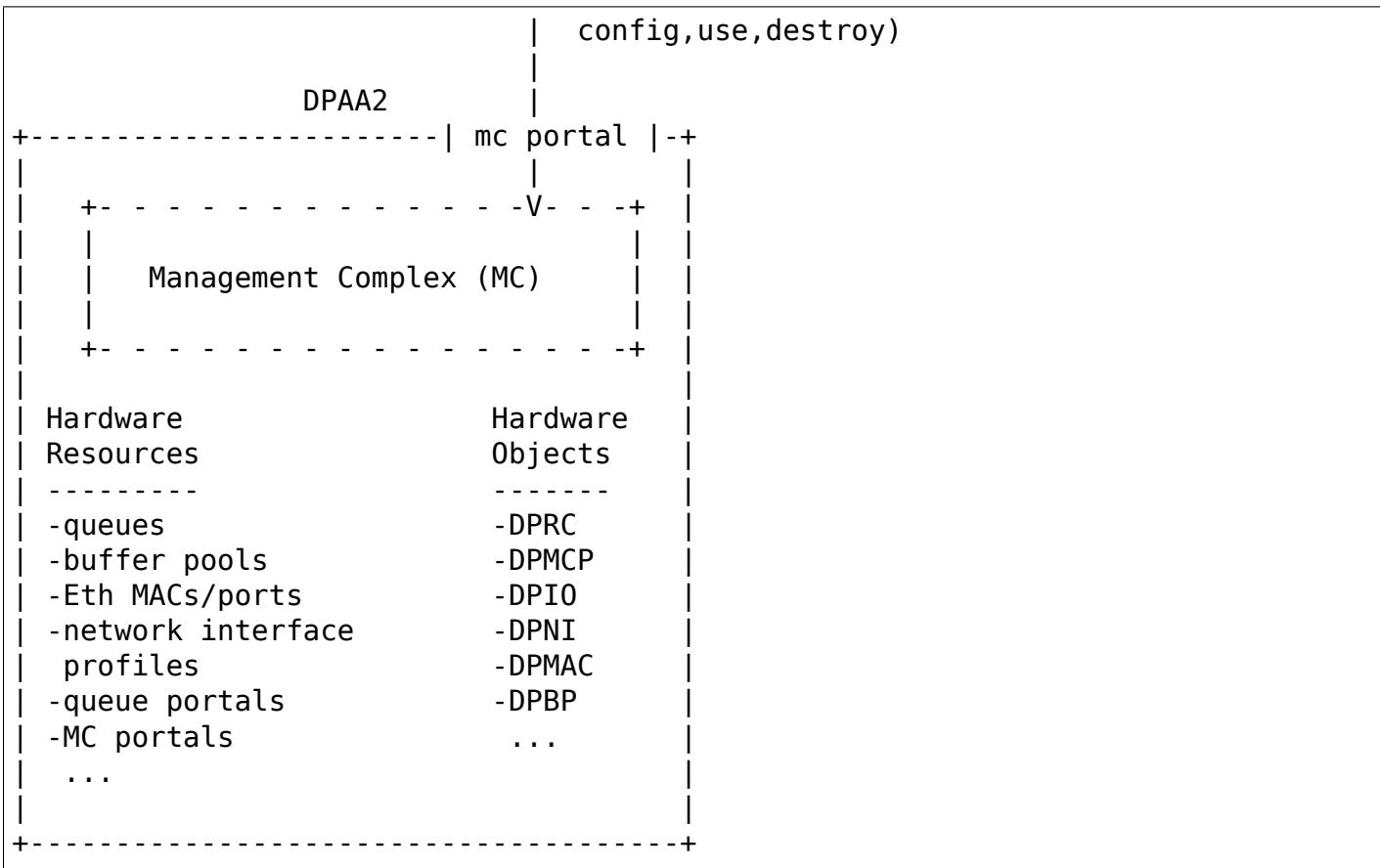
Introduction

DPAA2 is a hardware architecture designed for high-speed network packet processing. DPAA2 consists of sophisticated mechanisms for processing Ethernet packets, queue management, buffer management, autonomous L2 switching, virtual Ethernet bridging, and accelerator (e.g. crypto) sharing.

A DPAA2 hardware component called the Management Complex (or MC) manages the DPAA2 hardware resources. The MC provides an object-based abstraction for software drivers to use the DPAA2 hardware. The MC uses DPAA2 hardware resources such as queues, buffer pools, and network ports to create functional objects/devices such as network interfaces, an L2 switch, or accelerator instances. The MC provides memory-mapped I/O command interfaces (MC portals) which DPAA2 software drivers use to operate on DPAA2 objects.

The diagram below shows an overview of the DPAA2 resource management architecture:





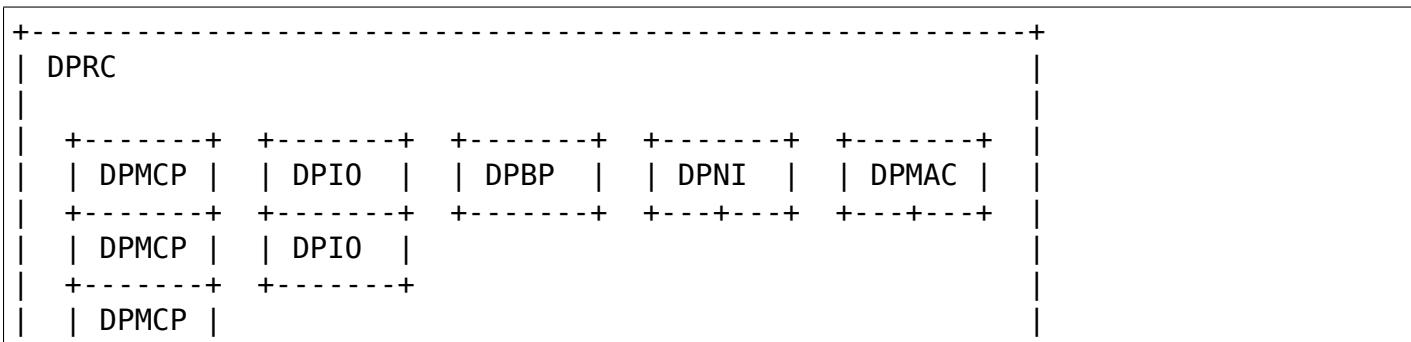
The MC mediates operations such as create, discover, connect, configuration, and destroy. Fast-path operations on data, such as packet transmit/receive, are not mediated by the MC and are done directly using memory mapped regions in DPIO objects.

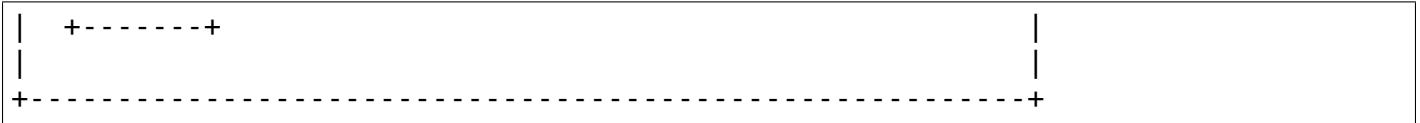
Overview of DPAA2 Objects

The section provides a brief overview of some key DPAA2 objects. A simple scenario is described illustrating the objects involved in creating a network interfaces.

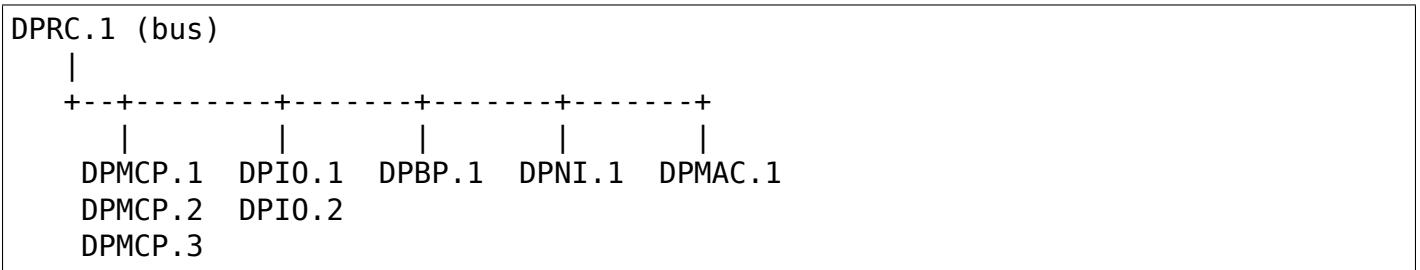
DPRC (Datapath Resource Container)

A DPRC is a container object that holds all the other types of DPAA2 objects. In the example diagram below there are 8 objects of 5 types (DPMCP, DPIO, DPBP, DPNI, and DPMAC) in the container.





From the point of view of an OS, a DPRC behaves similar to a plug and play bus, like PCI. DPRC commands can be used to enumerate the contents of the DPRC, discover the hardware objects present (including mappable regions and interrupts).



Hardware objects can be created and destroyed dynamically, providing the ability to hot plug/unplug objects in and out of the DPRC.

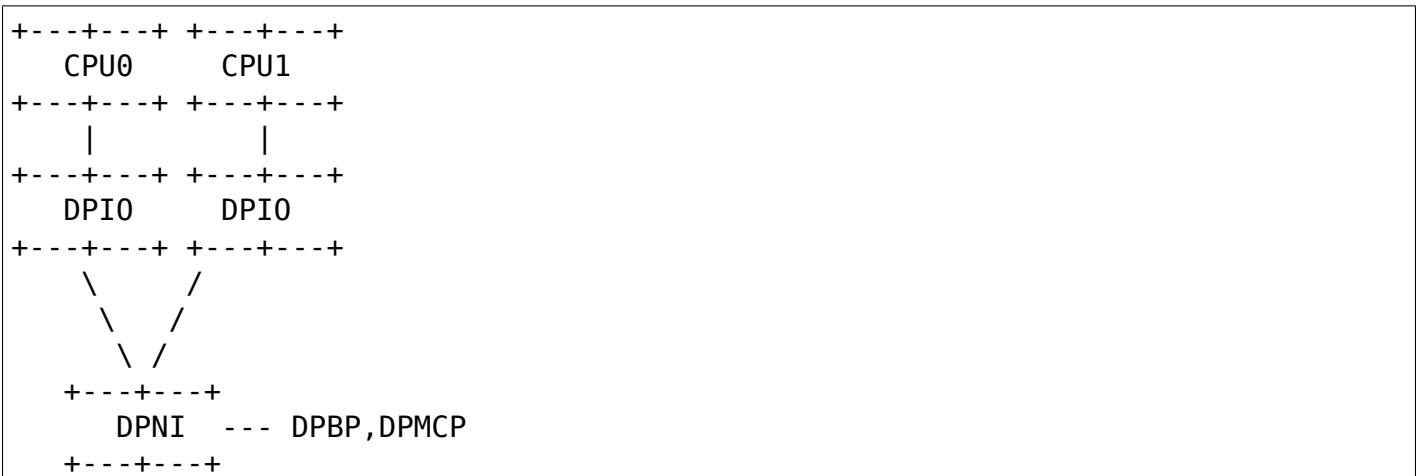
A DPRC has a mappable MMIO region (an MC portal) that can be used to send MC commands. It has an interrupt for status events (like hotplug). All objects in a container share the same hardware "isolation context". This means that with respect to an IOMMU the isolation granularity is at the DPRC (container) level, not at the individual object level.

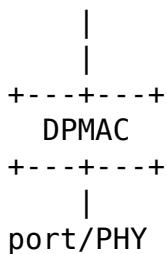
DPRCs can be defined statically and populated with objects via a config file passed to the MC when firmware starts it.

DPAA2 Objects for an Ethernet Network Interface

A typical Ethernet NIC is monolithic-- the NIC device contains TX/RX queuing mechanisms, configuration mechanisms, buffer management, physical ports, and interrupts. DPAA2 uses a more granular approach utilizing multiple hardware objects. Each object provides specialized functions. Groups of these objects are used by software to provide Ethernet network interface functionality. This approach provides efficient use of finite hardware resources, flexibility, and performance advantages.

The diagram below shows the objects needed for a simple network interface configuration on a system with 2 CPUs.





Below the objects are described. For each object a brief description is provided along with a summary of the kinds of operations the object supports and a summary of key resources of the object (MMIO regions and IRQs).

DPMAC (Datapath Ethernet MAC)

Represents an Ethernet MAC, a hardware device that connects to an Ethernet PHY and allows physical transmission and reception of Ethernet frames.

- MMIO regions: none
- IRQs: DPNI link change
- commands: set link up/down, link config, get stats, IRQ config, enable, reset

DPNI (Datapath Network Interface)

Contains TX/RX queues, network interface configuration, and RX buffer pool configuration mechanisms. The TX/RX queues are in memory and are identified by queue number.

- MMIO regions: none
- IRQs: link state
- commands: port config, offload config, queue config, parse/classify config, IRQ config, enable, reset

DPIO (Datapath I/O)

Provides interfaces to enqueue and dequeue packets and do hardware buffer pool management operations. The DPAA2 architecture separates the mechanism to access queues (the DPIO object) from the queues themselves. The DPIO provides an MMIO interface to enqueue/dequeue packets. To enqueue something a descriptor is written to the DPIO MMIO region, which includes the target queue number. There will typically be one DPIO assigned to each CPU. This allows all CPUs to simultaneously perform enqueue/dequeued operations. DPIOs are expected to be shared by different DPAA2 drivers.

- MMIO regions: queue operations, buffer management
- IRQs: data availability, congestion notification, buffer pool depletion
- commands: IRQ config, enable, reset

DPBP (Datapath Buffer Pool)

Represents a hardware buffer pool.

- MMIO regions: none
- IRQs: none
- commands: enable, reset

DPMCP (Datapath MC Portal)

Provides an MC command portal. Used by drivers to send commands to the MC to manage objects.

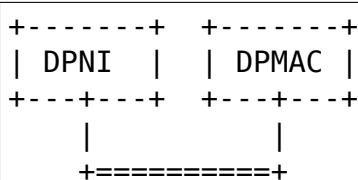
- MMIO regions: MC command portal
- IRQs: command completion
- commands: IRQ config, enable, reset

Object Connections

Some objects have explicit relationships that must be configured:

- DPNI <--> DPMAC
- DPNI <--> DPNI
- DPNI <--> L2-switch-port

A DPNI must be connected to something such as a DPMAC, another DPNI, or L2 switch port. The DPNI connection is made via a DPRC command.



- DPNI <--> DPBP

A network interface requires a 'buffer pool' (DPBP object) which provides a list of pointers to memory where received Ethernet data is to be copied. The Ethernet driver configures the DPBPs associated with the network interface.

Interrupts

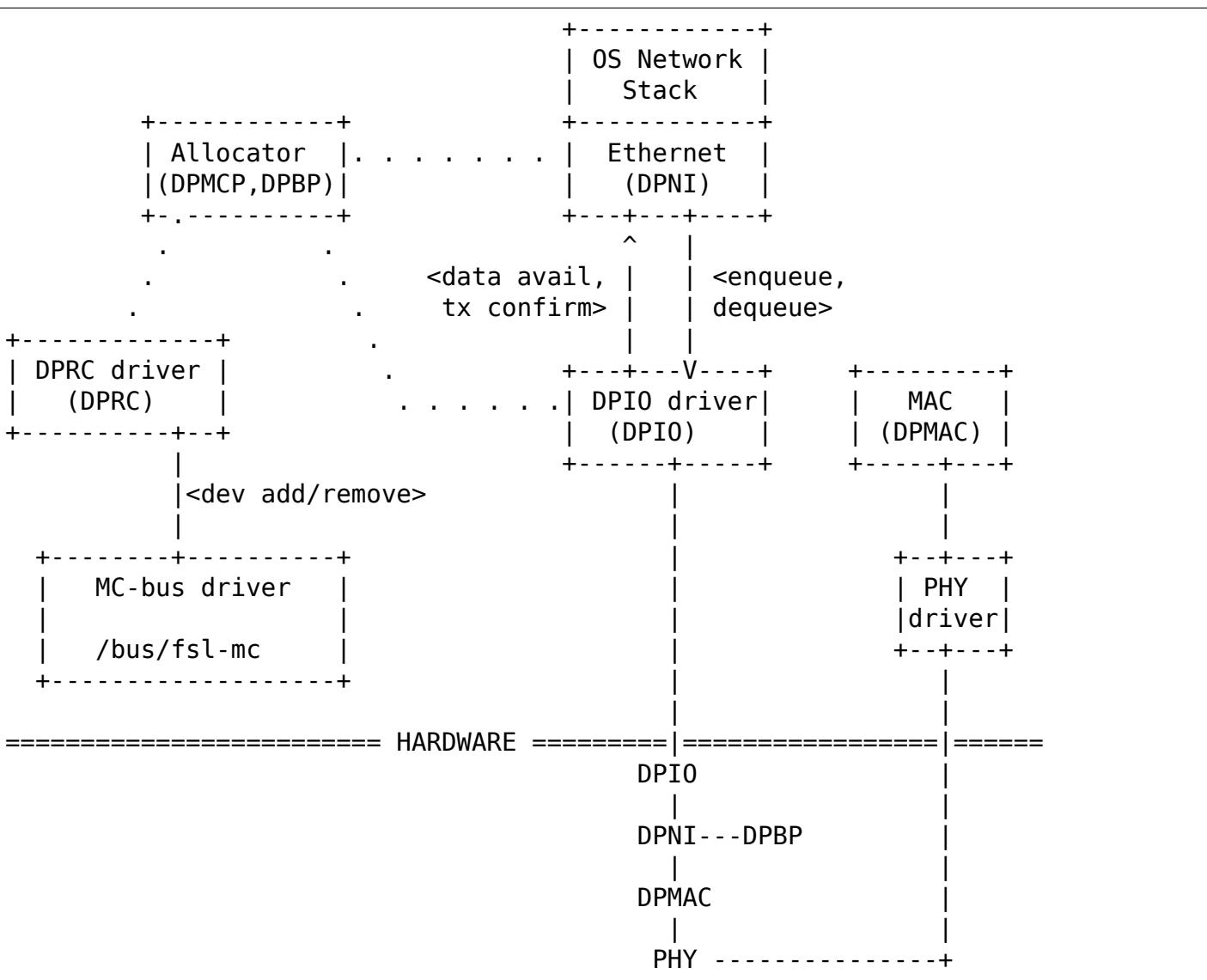
All interrupts generated by DPAA2 objects are message interrupts. At the hardware level message interrupts generated by devices will normally have 3 components-- 1) a non-spoofable 'device-id' expressed on the hardware bus, 2) an address, 3) a data value.

In the case of DPAA2 devices/objects, all objects in the same container/DPRC share the same 'device-id'. For ARM-based SoC this is the same as the stream ID.

DPAA2 Linux Drivers Overview

This section provides an overview of the Linux kernel drivers for DPAA2-- 1) the bus driver and associated "DPAA2 infrastructure" drivers and 2) functional object drivers (such as Ethernet).

As described previously, a DPRC is a container that holds the other types of DPAA2 objects. It is functionally similar to a plug-and-play bus controller. Each object in the DPRC is a Linux "device" and is bound to a driver. The diagram below shows the Linux drivers involved in a networking scenario and the objects bound to each driver. A brief description of each driver follows.



A brief description of each driver is provided below.

MC-bus driver

The MC-bus driver is a platform driver and is probed from a node in the device tree (compatible "fsl,qoriq-mc") passed in by boot firmware. It is responsible for bootstrapping the DPAA2 kernel infrastructure. Key functions include:

- registering a new bus type named "fsl-mc" with the kernel, and implementing bus callbacks (e.g. match/uevent/dev_groups)
- implementing APIs for DPAA2 driver registration and for device add/remove
- creates an MSI IRQ domain
- doing a 'device add' to expose the 'root' DPRC, in turn triggering a bind of the root DPRC to the DPRC driver

The binding for the MC-bus device-tree node can be consulted at *Documentation/devicetree/bindings/misc/fsl,qoriq-mc.txt*. The sysfs bind/unbind interfaces for the MC-bus can be consulted at *Documentation/ABI/testing/sysfs-bus-fsl-mc*.

DPRC driver

The DPRC driver is bound to DPRC objects and does runtime management of a bus instance. It performs the initial bus scan of the DPRC and handles interrupts for container events such as hot plug by re-scanning the DPRC.

Allocator

Certain objects such as DPMCP and DPBP are generic and fungible, and are intended to be used by other drivers. For example, the DPAA2 Ethernet driver needs:

- DPMCPs to send MC commands, to configure network interfaces
- DPBPs for network buffer pools

The allocator driver registers for these allocatable object types and those objects are bound to the allocator when the bus is probed. The allocator maintains a pool of objects that are available for allocation by other DPAA2 drivers.

DPIO driver

The DPIO driver is bound to DPIO objects and provides services that allow other drivers such as the Ethernet driver to enqueue and dequeue data for their respective objects. Key services include:

- data availability notifications
- hardware queuing operations (enqueue and dequeue of data)
- hardware buffer pool management

To transmit a packet the Ethernet driver puts data on a queue and invokes a DPIO API. For receive, the Ethernet driver registers a data availability notification callback. To dequeue a packet a DPIO API is used. There is typically one DPIO object per physical CPU for optimum performance, allowing different CPUs to simultaneously enqueue and dequeue data.

The DPIO driver operates on behalf of all DPAA2 drivers active in the kernel-- Ethernet, crypto, compression, etc.

Ethernet driver

The Ethernet driver is bound to a DPNI and implements the kernel interfaces needed to connect the DPAA2 network interface to the network stack. Each DPNI corresponds to a Linux network interface.

MAC driver

An Ethernet PHY is an off-chip, board specific component and is managed by the appropriate PHY driver via an mdio bus. The MAC driver plays a role of being a proxy between the PHY driver and the MC. It does this proxy via the MC commands to a DPMAC object. If the PHY driver signals a link change, the MAC driver notifies the MC via a DPMAC command. If a network interface is brought up or down, the MC notifies the DPMAC driver via an interrupt and the driver can take appropriate action.

DPAA2 DPIO (Data Path I/O) Overview

Copyright

© 2016-2018 NXP

This document provides an overview of the Freescale DPAA2 DPIO drivers

Introduction

A DPAA2 DPIO (Data Path I/O) is a hardware object that provides interfaces to enqueue and dequeue frames to/from network interfaces and other accelerators. A DPIO also provides hardware buffer pool management for network interfaces.

This document provides an overview the Linux DPIO driver, its subcomponents, and its APIs.

See [DPAA2 \(Data Path Acceleration Architecture Gen2\) Overview](#) for a general overview of DPAA2 and the general DPAA2 driver architecture in Linux.

Driver Overview

The DPIO driver is bound to DPIO objects discovered on the fsl-mc bus and provides services that:

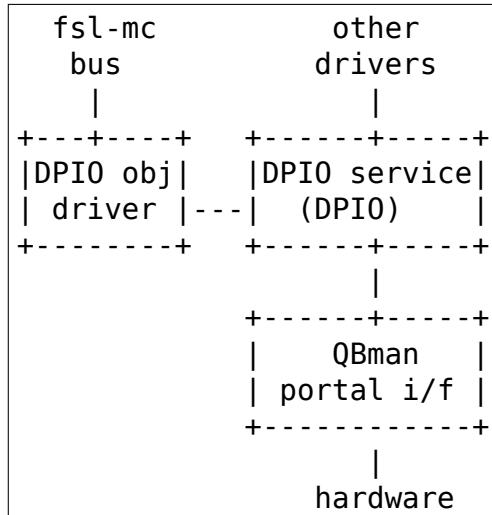
- A. allow other drivers, such as the Ethernet driver, to enqueue and dequeue frames for their respective objects
- B. allow drivers to register callbacks for data availability notifications when data becomes available on a queue or channel
- C. allow drivers to manage hardware buffer pools

The Linux DPIO driver consists of 3 primary components--

DPIO object driver-- fsl-mc driver that manages the DPIO object

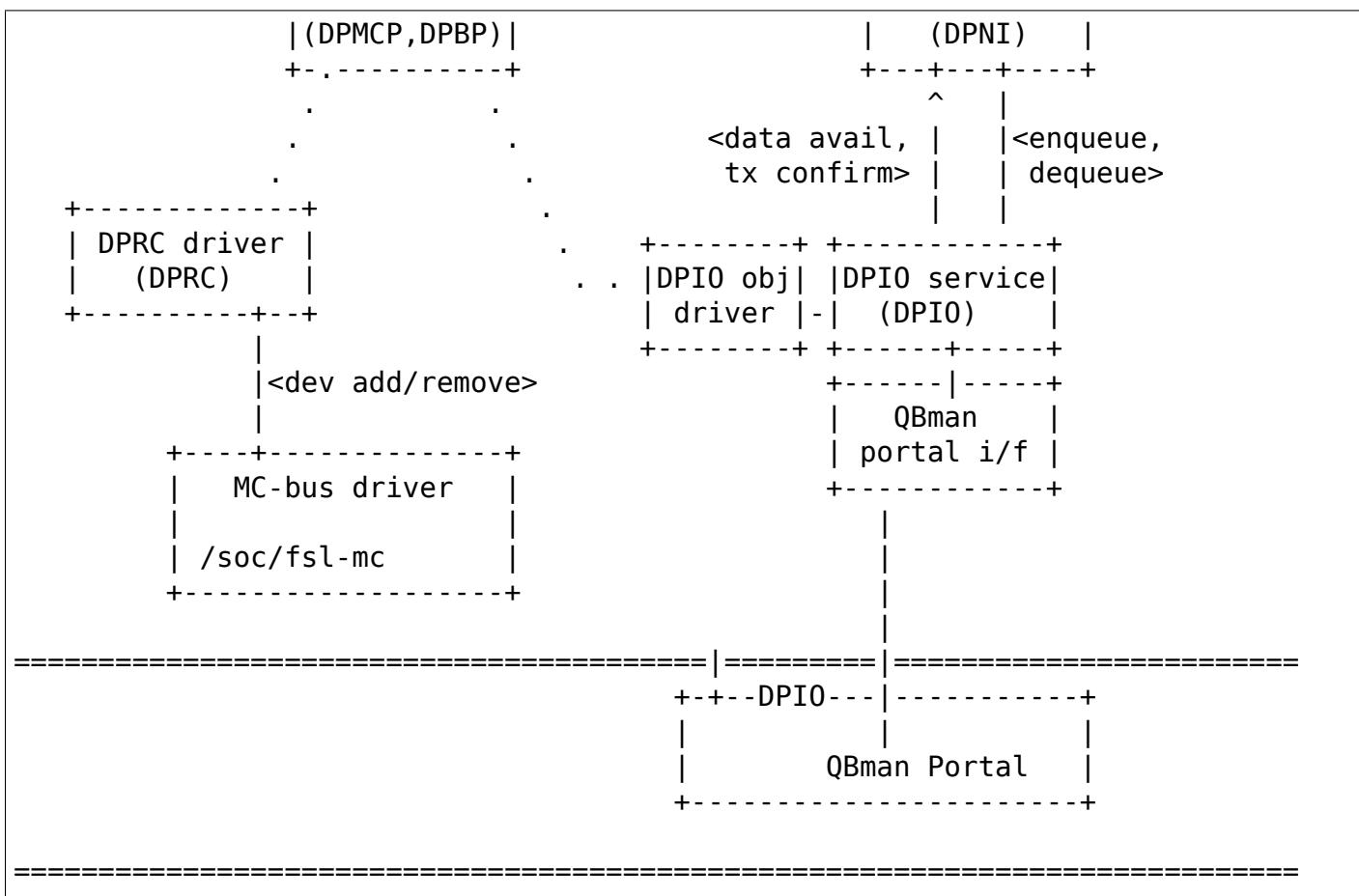
DPIO service-- provides APIs to other Linux drivers for services

QBman portal interface-- sends portal commands, gets responses:



The diagram below shows how the DPIO driver components fit with the other DPAA2 Linux driver components:





DPIO Object Driver (dpio-driver.c)

The dpio-driver component registers with the fsl-mc bus to handle objects of type "dpio". The implementation of probe() handles basic initialization of the DPIO including mapping of the DPIO regions (the QBman SW portal) and initializing interrupts and registering irq handlers. The dpio-driver registers the probed DPIO with dpio-service.

DPIO service (dpio-service.c, dpaa2-io.h)

The dpio service component provides queuing, notification, and buffers management services to DPAA2 drivers, such as the Ethernet driver. A system will typically allocate 1 DPIO object per CPU to allow queuing operations to happen simultaneously across all CPUs.

Notification handling

```

dpa2_io_service_register()
dpa2_io_service_deregister()
dpa2_io_service_rearm()

```

Queuing

```
dpa2_io_service_pull_fq()
```

```
dpaa2_io_service_pull_channel()
dpaa2_io_service_enqueue_fq()
dpaa2_io_service_enqueue_qd()
dpaa2_io_store_create()
dpaa2_io_store_destroy()
dpaa2_io_store_next()
```

Buffer pool management

```
dpaa2_io_service_release()
dpaa2_io_service_acquire()
```

QBman portal interface (qbman-portal.c)

The qbman-portal component provides APIs to do the low level hardware bit twiddling for operations such as:

- initializing Qman software portals
- building and sending portal commands
- portal interrupt configuration and processing

The qbman-portal APIs are not public to other drivers, and are only used by dpio-service.

Other (dpaa2-fd.h, dpaa2-global.h)

Frame descriptor and scatter-gather definitions and the APIs used to manipulate them are defined in dpaa2-fd.h.

Dequeue result struct and parsing APIs are defined in dpaa2-global.h.

DPAA2 Ethernet driver

Copyright

© 2017-2018 NXP

This file provides documentation for the Freescale DPAA2 Ethernet driver.

Supported Platforms

This driver provides networking support for Freescale DPAA2 SoCs, e.g. LS2080A, LS2088A, LS1088A.

Architecture Overview

Unlike regular NICs, in the DPAA2 architecture there is no single hardware block representing network interfaces; instead, several separate hardware resources concur to provide the networking functionality:

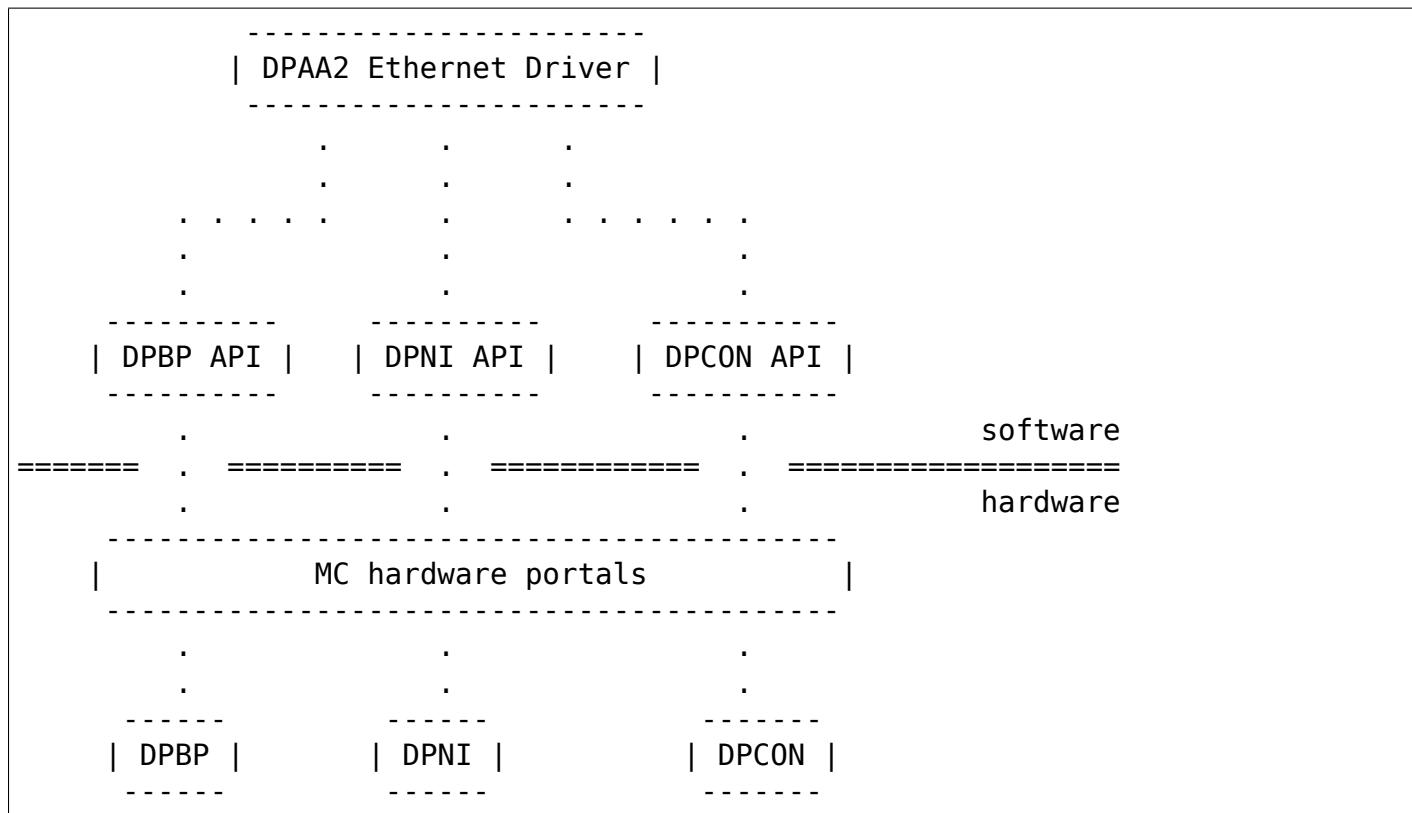
- network interfaces
- queues, channels
- buffer pools
- MAC/PHY

All hardware resources are allocated and configured through the Management Complex (MC) portals. MC abstracts most of these resources as DPAA2 objects and exposes ABIs through which they can be configured and controlled. A few hardware resources, like queues, do not have a corresponding MC object and are treated as internal resources of other objects.

For a more detailed description of the DPAA2 architecture and its object abstractions see [DPAA2 \(Data Path Acceleration Architecture Gen2\) Overview](#).

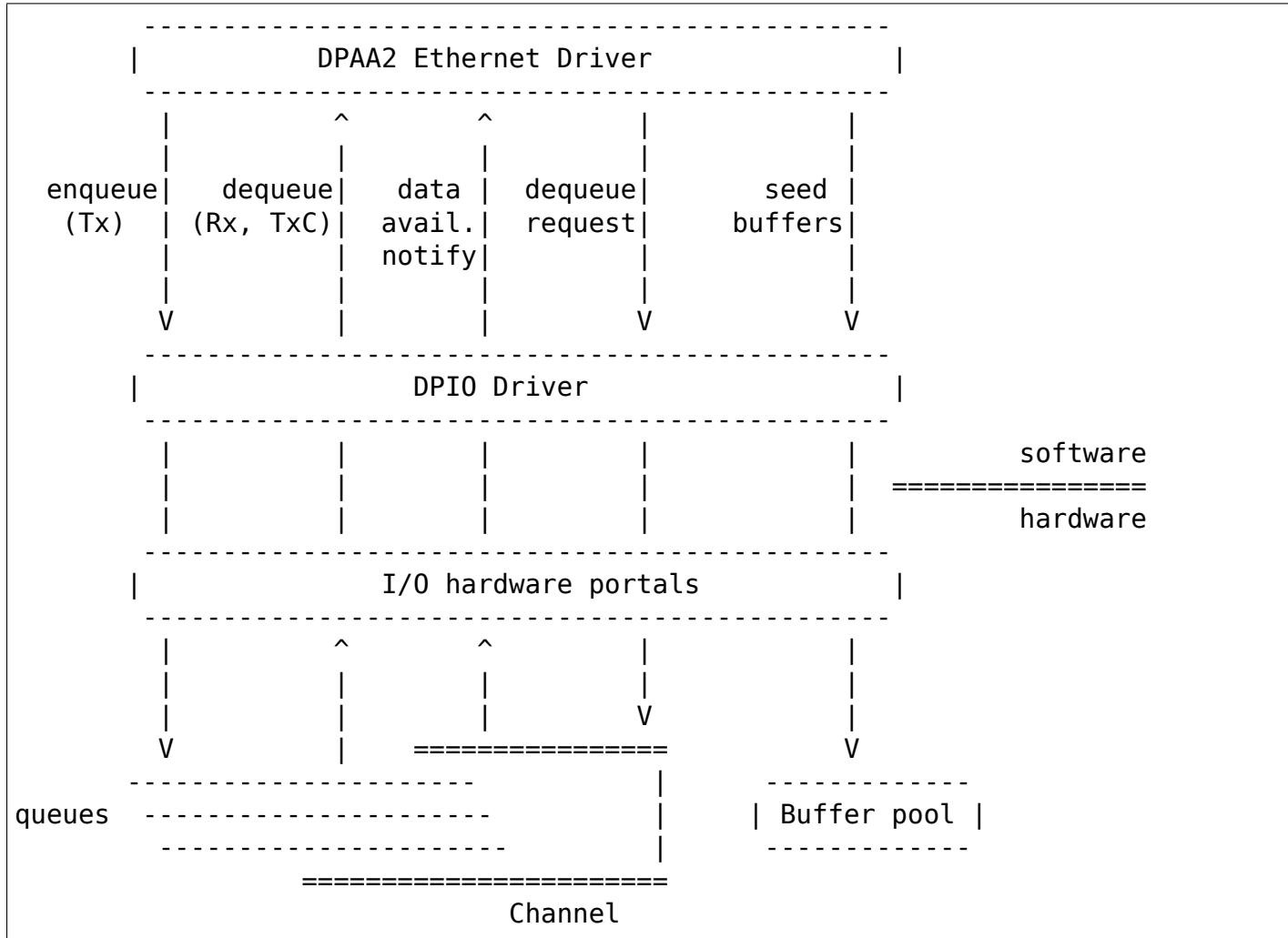
Each Linux net device is built on top of a Datapath Network Interface (DPNI) object and uses Buffer Pools (DPBPs), I/O Portals (DPIOs) and Concentrators (DPCONs).

Configuration interface:



The DPNIs are network interfaces without a direct one-on-one mapping to PHYs. DPBPs represent hardware buffer pools. Packet I/O is performed in the context of DPCON objects, using DPIO portals for managing and communicating with the hardware resources.

Datapath (I/O) interface:



Datapath I/O (DPIO) portals provide enqueue and dequeue services, data availability notifications and buffer pool management. DPIOs are shared between all DPAA2 objects (and implicitly all DPAA2 kernel drivers) that work with data frames, but must be affine to the CPUs for the purpose of traffic distribution.

Frames are transmitted and received through hardware frame queues, which can be grouped in channels for the purpose of hardware scheduling. The Ethernet driver enqueues TX frames on egress queues and after transmission is complete a TX confirmation frame is sent back to the CPU.

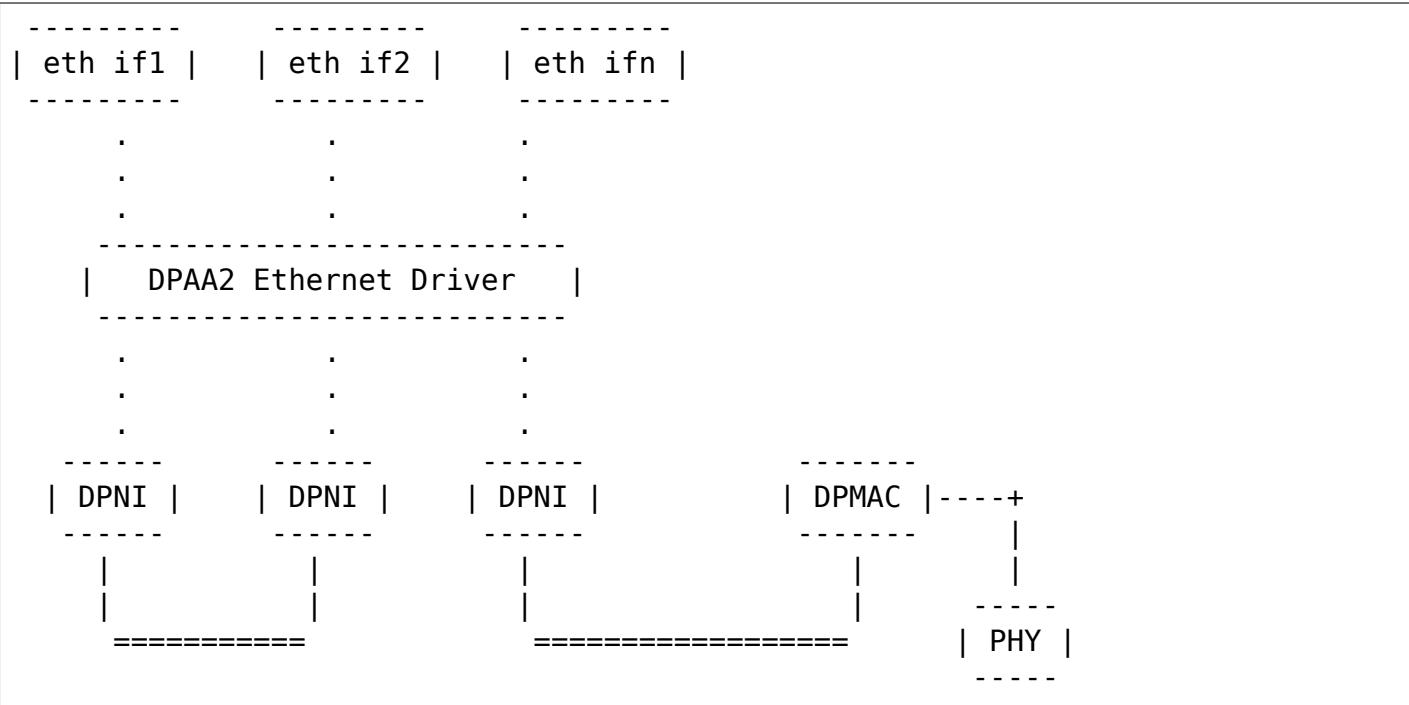
When frames are available on ingress queues, a data availability notification is sent to the CPU; notifications are raised per channel, so even if multiple queues in the same channel have available frames, only one notification is sent. After a channel fires a notification, it must be explicitly rearmed.

Each network interface can have multiple Rx, Tx and confirmation queues affined to CPUs, and one channel (DPCON) for each CPU that services at least one queue. DPCONs are used to distribute ingress traffic to different CPUs via the cores' affine DPIOs.

The role of hardware buffer pools is storage of ingress frame data. Each network interface has a privately owned buffer pool which it seeds with kernel allocated buffers.

DPNIs are decoupled from PHYs; a DPNI can be connected to a PHY through a DPMAC object or to another DPNI through an internal link, but the connection is managed by MC and completely

transparent to the Ethernet driver.



Creating a Network Interface

A net device is created for each DPNI object probed on the MC bus. Each DPNI has a number of properties which determine the network interface configuration options and associated hardware resources.

DPNI objects (and the other DPAA2 objects needed for a network interface) can be added to a container on the MC bus in one of two ways: statically, through a Datapath Layout Binary file (DPL) that is parsed by MC at boot time; or created dynamically at runtime, via the DPAA2 objects APIs.

Features & Offloads

Hardware checksum offloading is supported for TCP and UDP over IPv4/6 frames. The checksum offloads can be independently configured on RX and TX through ethtool.

Hardware offload of unicast and multicast MAC filtering is supported on the ingress path and permanently enabled.

Scatter-gather frames are supported on both RX and TX paths. On TX, SG support is configurable via ethtool; on RX it is always enabled.

The DPAA2 hardware can process jumbo Ethernet frames of up to 10K bytes.

The Ethernet driver defines a static flow hashing scheme that distributes traffic based on a 5-tuple key: src IP, dst IP, IP proto, L4 src port, L4 dst port. No user configuration is supported for now.

Hardware specific statistics for the network interface as well as some non-standard driver stats can be consulted through ethtool -S option.

DPAA2 MAC / PHY support

Copyright

© 2019 NXP

Overview

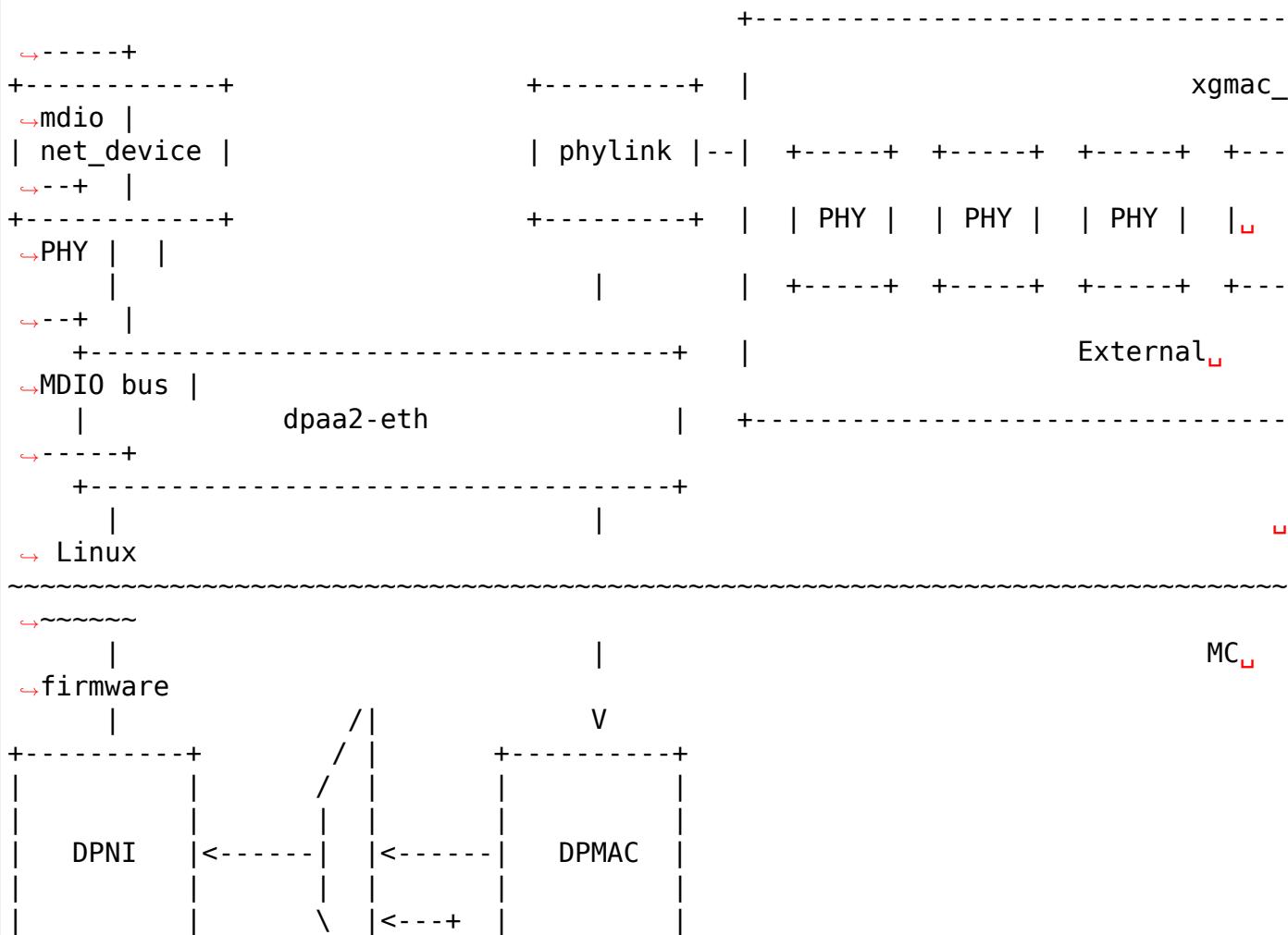
The DPAA2 MAC / PHY support consists of a set of APIs that help DPAA2 network drivers (dpaa2-eth, dpaa2-ethsw) interact with the PHY library.

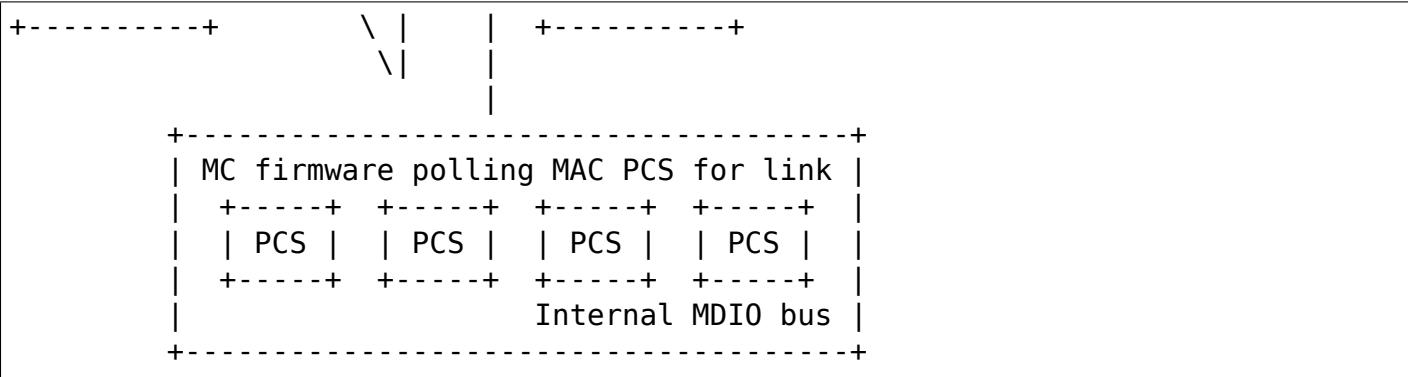
DPAA2 Software Architecture

Among other DPAA2 objects, the fsl-mc bus exports DPNI objects (abstracting a network interface) and DPMAC objects (abstracting a MAC). The dpaa2-eth driver probes on the DPNI object and connects to and configures a DPMAC object with the help of phylink.

Data connections may be established between a DPNI and a DPMAC, or between two DPNI objects. Depending on the connection type, the netif_carrier_[on/off] is handled directly by the dpaa2-eth driver or by phylink.

Sources of abstracted link state information presented by the MC firmware





Depending on an MC firmware configuration setting, each MAC may be in one of two modes:

- DPMAC_LINK_TYPE_FIXED: the link state management is handled exclusively by the MC firmware by polling the MAC PCS. Without the need to register a phylink instance, the dpaa2-eth driver will not bind to the connected dpmac object at all.
- DPMAC_LINK_TYPE_PHY: The MC firmware is left waiting for link state update events, but those are in fact passed strictly between the dpaa2-mac (based on phylink) and its attached net_device driver (dpaa2-eth, dpaa2-ethsw), effectively bypassing the firmware.

Implementation

At probe time or when a DPNI's endpoint is dynamically changed, the dpaa2-eth is responsible to find out if the peer object is a DPMAC and if this is the case, to integrate it with PHYLINK using the `dpaa2_mac_connect()` API, which will do the following:

- look up the device tree for PHYLINK-compatible of binding (phy-handle)
- will create a PHYLINK instance associated with the received net_device
- connect to the PHY using `phylink_of_phy_connect()`

The following phylink_mac_ops callback are implemented:

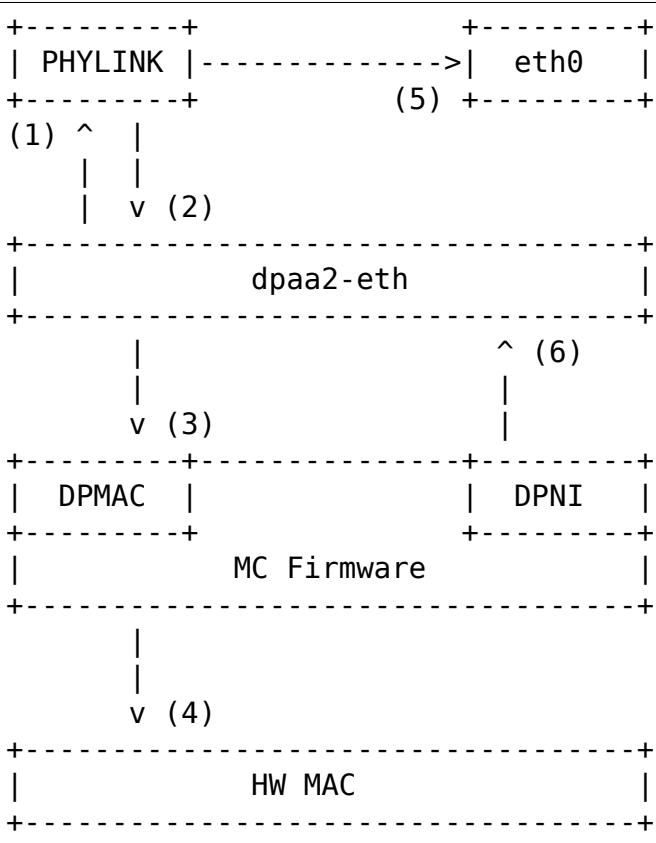
- `.validate()` will populate the supported linkmodes with the MAC capabilities only when the `phy_interface_t` is RGMII_*(at the moment, this is the only link type supported by the driver).
- `.mac_config()` will configure the MAC in the new configuration using the `dpmac_set_link_state()` MC firmware API.
- `.mac_link_up()` / `.mac_link_down()` will update the MAC link using the same API described above.

At driver unbind() or when the DPNI object is disconnected from the DPMAC, the dpaa2-eth driver calls `dpaa2_mac_disconnect()` which will, in turn, disconnect from the PHY and destroy the PHYLINK instance.

In case of a DPNI-DPMAC connection, an 'ip link set dev eth0 up' would start the following sequence of operations:

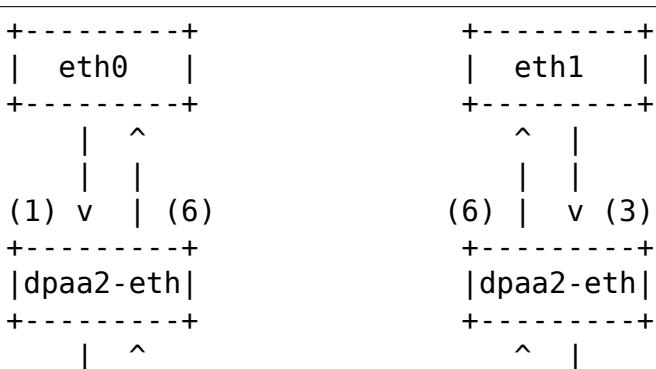
- (1) `phylink_start()` called from `.dev_open()`.
- (2) The `.mac_config()` and `.mac_link_up()` callbacks are called by PHYLINK.
- (3) In order to configure the HW MAC, the MC Firmware API `dpmac_set_link_state()` is called.

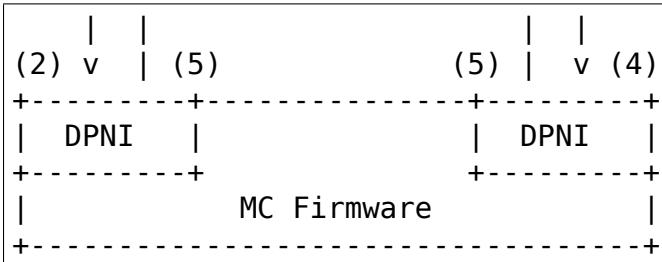
- (4) The firmware will eventually setup the HW MAC in the new configuration.
- (5) A `netif_carrier_on()` call is made directly from PHYLINK on the associated net_device.
- (6) The dpaa2-eth driver handles the LINK_STATE_CHANGE irq in order to enable/disable Rx taildrop based on the pause frame settings.



In case of a DPNI-DPNI connection, a usual sequence of operations looks like the following:

- (1) ip link set dev eth0 up
- (2) The dpni_enable() MC API called on the associated `fsl_mc_device`.
- (3) ip link set dev eth1 up
- (4) The dpni_enable() MC API called on the associated `fsl_mc_device`.
- (5) The LINK_STATE_CHANGED irq is received by both instances of the dpaa2-eth driver because now the operational link state is up.
- (6) The `netif_carrier_on()` is called on the exported net_device from `link_state_update()`.





Exported API

Any DPAA2 driver that drivers endpoints of DPMAC objects should service its `_EVENT_ENDPOINT_CHANGED` irq and connect/disconnect from the associated DPMAC when necessary using the below listed API:

```
- int dpaa2_mac_connect(struct dpaa2_mac *mac);
- void dpaa2_mac_disconnect(struct dpaa2_mac *mac);
```

A phylink integration is necessary only when the partner DPMAC is not of `TYPE_FIXED`. This means it is either of `TYPE_PHY`, or of `TYPE_BACKPLANE` (the difference being the two that in the `TYPE_BACKPLANE` mode, the MC firmware does not access the PCS registers). One can check for this condition using the following helper:

```
- static inline bool dpaa2_mac_is_type_phy(struct dpaa2_mac *mac);
```

Before connection to a MAC, the caller must allocate and populate the `dpaa2_mac` structure with the associated `net_device`, a pointer to the MC portal to be used and the actual `fsl_mc_device` structure of the DPMAC.

DPAA2 Switch driver

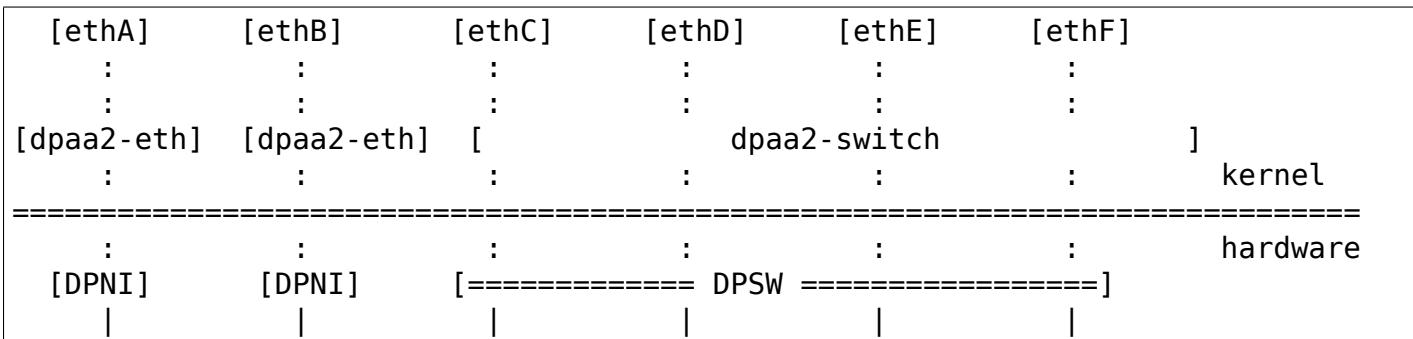
Copyright

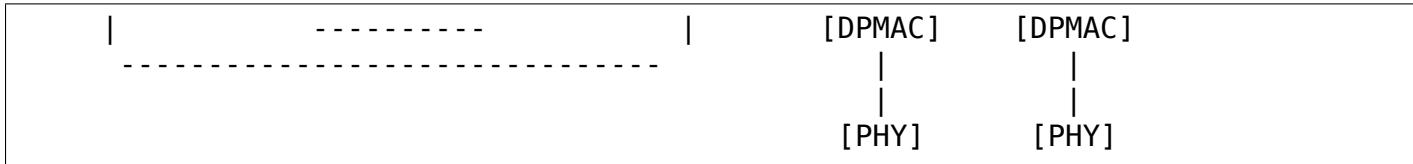
© 2021 NXP

The DPAA2 Switch driver probes on the Datapath Switch (DPSW) object which can be instantiated on the following DPAA2 SoCs and their variants: LS2088A and LX2160A.

The driver uses the switch device driver model and exposes each switch port as a network interface, which can be included in a bridge or used as a standalone interface. Traffic switched between ports is offloaded into the hardware.

The DPSW can have ports connected to DPNI or to DPMACs for external access.





Creating an Ethernet Switch

The dpaa2-switch driver probes on DPSW devices found on the fsl-mc bus. These devices can be either created statically through the boot time configuration file - DataPath Layout (DPL) - or at runtime using the DPAA2 object APIs (incorporated already into the restool userspace tool).

At the moment, the dpaa2-switch driver imposes the following restrictions on the DPSW object that it will probe:

- The minimum number of FDBs should be at least equal to the number of switch interfaces. This is necessary so that separation of switch ports can be done, ie when not under a bridge, each switch port will have its own FDB.

```
fsl_dpaa2_switch dpsw.0: The number of FDBs is lower than the number of ports, cannot probe
```

- Both the broadcast and flooding configuration should be per FDB. This enables the driver to restrict the broadcast and flooding domains of each FDB depending on the switch ports that are sharing it (aka are under the same bridge).

```
fsl_dpaa2_switch dpsw.0: Flooding domain is not per FDB, cannot probe
fsl_dpaa2_switch dpsw.0: Broadcast domain is not per FDB, cannot probe
```

- The control interface of the switch should not be disabled (DPSW_OPT_CTRL_IF_DIS not passed as a create time option). Without the control interface, the driver is not capable to provide proper Rx/Tx traffic support on the switch port netdevices.

```
fsl_dpaa2_switch dpsw.0: Control Interface is disabled, cannot probe
```

Besides the configuration of the actual DPSW object, the dpaa2-switch driver will need the following DPAA2 objects:

- 1 DPMCP - A Management Command Portal object is needed for any interaction with the MC firmware.
- 1 DPBP - A Buffer Pool is used for seeding buffers intended for the Rx path on the control interface.
- Access to at least one DPIO object (Software Portal) is needed for any enqueue/dequeue operation to be performed on the control interface queues. The DPIO object will be shared, no need for a private one.

Switching features

The driver supports the configuration of L2 forwarding rules in hardware for port bridging as well as standalone usage of the independent switch interfaces.

The hardware is not configurable with respect to VLAN awareness, thus any DPAA2 switch port should be used only in usecases with a VLAN aware bridge:

```
$ ip link add dev br0 type bridge vlan_filtering 1
$ ip link add dev br1 type bridge
$ ip link set dev ethX master br1
Error: fsl_dpaa2_switch: Cannot join a VLAN-unaware bridge
```

Topology and loop detection through STP is supported when `stp_state 1` is used at bridge create

```
$ ip link add dev br0 type bridge vlan_filtering 1 stp_state 1
```

L2 FDB manipulation (add/delete/dump) is supported.

HW FDB learning can be configured on each switch port independently through bridge commands. When the HW learning is disabled, a fast age procedure will be run and any previously learnt addresses will be removed.

```
$ bridge link set dev ethX learning off
$ bridge link set dev ethX learning on
```

Restricting the unknown unicast and multicast flooding domain is supported, but not independently of each other:

```
$ ip link set dev ethX type bridge_slave flood off mcast_flood off
$ ip link set dev ethX type bridge_slave flood off mcast_flood on
Error: fsl_dpaa2_switch: Cannot configure multicast flooding independently of unicast.
```

Broadcast flooding on a switch port can be disabled/enabled through the `brport` sysfs:

```
$ echo 0 > /sys/bus/fsl-mc/devices/dpsw.Y/net/ethX/brport/broadcast_flood
```

Offloads

Routing actions (redirect, trap, drop)

The DPAA2 switch is able to offload flow-based redirection of packets making use of ACL tables. Shared filter blocks are supported by sharing a single ACL table between multiple ports.

The following flow keys are supported:

- Ethernet: `dst_mac/src_mac`
- IPv4: `dst_ip/src_ip/ip_proto/tos`

- VLAN: vlan_id/vlan_prio/vlan_tpid/vlan_dei
- L4: dst_port/src_port

Also, the matchall filter can be used to redirect the entire traffic received on a port.

As per flow actions, the following are supported:

- drop
- mirrored egress redirect
- trap

Each ACL entry (filter) can be setup with only one of the listed actions.

Example 1: send frames received on eth4 with a SA of 00:01:02:03:04:05 to the CPU:

```
$ tc qdisc add dev eth4 clsact
$ tc filter add dev eth4 ingress flower src_mac 00:01:02:03:04:05 skip_sw \
  ↳action trap
```

Example 2: drop frames received on eth4 with VID 100 and PCP of 3:

```
$ tc filter add dev eth4 ingress protocol 802.1q flower skip_sw vlan_id 100 \
  ↳vlan_prio 3 action drop
```

Example 3: redirect all frames received on eth4 to eth1:

```
$ tc filter add dev eth4 ingress matchall action mirrored egress redirect dev \
  ↳eth1
```

Example 4: Use a single shared filter block on both eth5 and eth6:

```
$ tc qdisc add dev eth5 ingress_block 1 clsact
$ tc qdisc add dev eth6 ingress_block 1 clsact
$ tc filter add block 1 ingress flower dst_mac 00:01:02:03:04:04 skip_sw \
  ↳action trap
$ tc filter add block 1 ingress protocol ipv4 flower src_ip 192.168.1.1 skip_ \
  ↳sw \
  ↳action mirrored egress redirect dev eth3
```

Mirroring

The DPAA2 switch supports only per port mirroring and per VLAN mirroring. Adding mirroring filters in shared blocks is also supported.

When using the tc-flower classifier with the 802.1q protocol, only the "vlan_id" key will be accepted. Mirroring based on any other fields from the 802.1q protocol will be rejected:

```
$ tc qdisc add dev eth8 ingress_block 1 clsact
$ tc filter add block 1 ingress protocol 802.1q flower skip_sw vlan_prio 3 \
  ↳action mirrored egress mirror dev eth6
Error: fsl_dpaa2_switch: Only matching on VLAN ID supported.
We have an error talking to the kernel
```

If a mirroring VLAN filter is requested on a port, the VLAN must be installed on the switch port in question either using "bridge" or by creating a VLAN upper device if the switch port is used as a standalone interface:

```
$ tc qdisc add dev eth8 ingress_block 1 clsact
$ tc filter add block 1 ingress protocol 802.1q flower skip_sw vlan_id 200 ↴
  ↪action mirred egress mirror dev eth6
Error: VLAN must be installed on the switch port.
We have an error talking to the kernel

$ bridge vlan add vid 200 dev eth8
$ tc filter add block 1 ingress protocol 802.1q flower skip_sw vlan_id 200 ↴
  ↪action mirred egress mirror dev eth6

$ ip link add link eth8 name eth8.200 type vlan id 200
$ tc filter add block 1 ingress protocol 802.1q flower skip_sw vlan_id 200 ↴
  ↪action mirred egress mirror dev eth6
```

Also, it should be noted that the mirrored traffic will be subject to the same egress restrictions as any other traffic. This means that when a mirrored packet will reach the mirror port, if the VLAN found in the packet is not installed on the port it will get dropped.

The DPAA2 switch supports only a single mirroring destination, thus multiple mirror rules can be installed but their "to" port has to be the same:

```
$ tc filter add block 1 ingress protocol 802.1q flower skip_sw vlan_id 200 ↴
  ↪action mirred egress mirror dev eth6
$ tc filter add block 1 ingress protocol 802.1q flower skip_sw vlan_id 100 ↴
  ↪action mirred egress mirror dev eth7
Error: fsl_dpaa2_switch: Multiple mirror ports not supported.
We have an error talking to the kernel
```

6.5.16 The Gianfar Ethernet Driver

Author

Andy Fleming <afleming@freescale.com>

Updated

2005-07-28

Checksum Offloading

The eTSEC controller (first included in parts from late 2005 like the 8548) has the ability to perform TCP, UDP, and IP checksums in hardware. The Linux kernel only offloads the TCP and UDP checksums (and always performs the pseudo header checksums), so the driver only supports checksumming for TCP/IP and UDP/IP packets. Use ethtool to enable or disable this feature for RX and TX.

VLAN

In order to use VLAN, please consult Linux documentation on configuring VLANs. The gianfar driver supports hardware insertion and extraction of VLAN headers, but not filtering. Filtering will be done by the kernel.

Multicasting

The gianfar driver supports using the group hash table on the TSEC (and the extended hash table on the eTSEC) for multicast filtering. On the eTSEC, the exact-match MAC registers are used before the hash tables. See Linux documentation on how to join multicast groups.

Padding

The gianfar driver supports padding received frames with 2 bytes to align the IP header to a 16-byte boundary, when supported by hardware.

Ethtool

The gianfar driver supports the use of ethtool for many configuration options. You must run ethtool only on currently open interfaces. See ethtool documentation for details.

6.5.17 Linux kernel driver for Compute Engine Virtual Ethernet (gve):

Supported Hardware

The GVE driver binds to a single PCI device id used by the virtual Ethernet device found in some Compute Engine VMs.

Field	Value	Comments
Vendor ID	0x1AE0	Google
Device ID	0x0042	
Sub-vendor ID	0x1AE0	Google
Sub-device ID	0x0058	
Revision ID	0x0	
Device Class	0x200	Ethernet

PCI Bars

The gVNIC PCI device exposes three 32-bit memory BARS: - Bar0 - Device configuration and status registers. - Bar1 - MSI-X vector table - Bar2 - IRQ, RX and TX doorbells

Device Interactions

The driver interacts with the device in the following ways:

- **Registers**
 - A block of MMIO registers
 - See `gve_register.h` for more detail
- **Admin Queue**
 - See description below
- **Reset**
 - At any time the device can be reset
- **Interrupts**
 - See supported interrupts below
- **Transmit and Receive Queues**
 - See description below

Descriptor Formats

GVE supports two descriptor formats: GQI and DQO. These two formats have entirely different descriptors, which will be described below.

Addressing Mode

GVE supports two addressing modes: QPL and RDA. QPL ("queue-page-list") mode communicates data through a set of pre-registered pages.

For RDA ("raw DMA addressing") mode, the set of pages is dynamic. Therefore, the packet buffers can be anywhere in guest memory.

Registers

All registers are MMIO.

The registers are used for initializing and configuring the device as well as querying device status in response to management interrupts.

Endianness

- Admin Queue messages and registers are all Big Endian.
- GQI descriptors and datapath registers are Big Endian.
- DQO descriptors and datapath registers are Little Endian.

Admin Queue (AQ)

The Admin Queue is a PAGE_SIZE memory block, treated as an array of AQ commands, used by the driver to issue commands to the device and set up resources. The driver and the device maintain a count of how many commands have been submitted and executed. To issue AQ commands, the driver must do the following (with proper locking):

- 1) Copy new commands into next available slots in the AQ array
- 2) Increment its counter by the number of new commands
- 3) Write the counter into the GVE_ADMIN_QUEUE_DOORBELL register
- 4) Poll the ADMIN_QUEUE_EVENT_COUNTER register until it equals the value written to the doorbell, or until a timeout.

The device will update the status field in each AQ command reported as executed through the ADMIN_QUEUE_EVENT_COUNTER register.

Device Resets

A device reset is triggered by writing 0x0 to the AQ PFN register. This causes the device to release all resources allocated by the driver, including the AQ itself.

Interrupts

The following interrupts are supported by the driver:

Management Interrupt

The management interrupt is used by the device to tell the driver to look at the GVE_DEVICE_STATUS register.

The handler for the management irq simply queues the service task in the workqueue to check the register and acks the irq.

Notification Block Interrupts

The notification block interrupts are used to tell the driver to poll the queues associated with that interrupt.

The handler for these irqs schedule the napi for that block to run and poll the queues.

GQI Traffic Queues

GQI queues are composed of a descriptor ring and a buffer and are assigned to a notification block.

The descriptor rings are power-of-two-sized ring buffers consisting of fixed-size descriptors. They advance their head pointer using a `_be32` doorbell located in Bar2. The tail pointers are advanced by consuming descriptors in-order and updating a `_be32` counter. Both the doorbell and the counter overflow to zero.

Each queue's buffers must be registered in advance with the device as a queue page list, and packet data can only be put in those pages.

Transmit

gve maps the buffers for transmit rings into a FIFO and copies the packets into the FIFO before sending them to the NIC.

Receive

The buffers for receive rings are put into a data ring that is the same length as the descriptor ring and the head and tail pointers advance over the rings together.

DQO Traffic Queues

- Every TX and RX queue is assigned a notification block.
- TX and RX buffers queues, which send descriptors to the device, use MMIO doorbells to notify the device of new descriptors.
- RX and TX completion queues, which receive descriptors from the device, use a "generation bit" to know when a descriptor was populated by the device. The driver initializes all bits with the "current generation". The device will populate received descriptors with the "next generation" which is inverted from the current generation. When the ring wraps, the current/next generation are swapped.
- It's the driver's responsibility to ensure that the RX and TX completion queues are not overrun. This can be accomplished by limiting the number of descriptors posted to HW.
- TX packets have a 16 bit `completion_tag` and RX buffers have a 16 bit `buffer_id`. These will be returned on the TX completion and RX queues respectively to let the driver know which packet/buffer was completed.

Transmit

A packet's buffers are DMA mapped for the device to access before transmission. After the packet was successfully transmitted, the buffers are unmapped.

Receive

The driver posts fixed sized buffers to HW on the RX buffer queue. The packet received on the associated RX queue may span multiple descriptors.

6.5.18 Linux Kernel Driver for Huawei Intelligent NIC(HiNIC) family

Overview:

HiNIC is a network interface card for the Data Center Area.

The driver supports a range of link-speed devices (10GbE, 25GbE, 40GbE, etc.). The driver supports also a negotiated and extendable feature set.

Some HiNIC devices support SR-IOV. This driver is used for Physical Function (PF).

HiNIC devices support MSI-X interrupt vector for each Tx/Rx queue and adaptive interrupt moderation.

HiNIC devices support also various offload features such as checksum offload, TCP Transmit Segmentation Offload(TSO), Receive-Side Scaling(RSS) and LRO(Large Receive Offload).

Supported PCI vendor ID/device IDs:

19e5:1822 - HiNIC PF

Driver Architecture and Source Code:

`hinic_dev` - Implement a Logical Network device that is independent from specific HW details about HW data structure formats.

`hinic_hwdev` - Implement the HW details of the device and include the components for accessing the PCI NIC.

`hinic_hwdev` contains the following components:

HW Interface:

The interface for accessing the pci device (DMA memory and PCI BARs). (`hinic_hw_if.c`, `hinic_hw_if.h`)

Configuration Status Registers Area that describes the HW Registers on the configuration and status BAR0. (`hinic_hw_csr.h`)

MGMT components:

Asynchronous Event Queues(AEQs) - The event queues for receiving messages from the MGMT modules on the cards. (`hinic_hw_eqs.c`, `hinic_hw_eqs.h`)

Application Programmable Interface commands(API CMD) - Interface for sending MGMT commands to the card. (`hinic_hw_api_cmd.c`, `hinic_hw_api_cmd.h`)

Management (MGMT) - the PF to MGMT channel that uses API CMD for sending MGMT commands to the card and receives notifications from the MGMT modules on the card by AEQs. Also set the addresses of the IO CMDQs in HW. (`hinic_hw_mgmt.c`, `hinic_hw_mgmt.h`)

IO components:

Completion Event Queues(CEQs) - The completion Event Queues that describe IO tasks that are finished. (`hinic_hw_eqs.c`, `hinic_hw_eqs.h`)

Work Queues(WQ) - Contain the memory and operations for use by CMD queues and the Queue Pairs. The WQ is a Memory Block in a Page. The Block contains pointers to Memory Areas that are the Memory for the Work Queue Elements(WQEs). (`hinic_hw_wq.c`, `hinic_hw_wq.h`)

Command Queues(CMDQ) - The queues for sending commands for IO management and is used to set the QPs addresses in HW. The commands completion events are accumulated on the CEQ that is configured to receive the CMDQ completion events. (`hinic_hw_cmdq.c`, `hinic_hw_cmdq.h`)

Queue Pairs(QPs) - The HW Receive and Send queues for Receiving and Transmitting Data. (`hinic_hw_qp.c`, `hinic_hw_qp.h`, `hinic_hw_qp_ctxt.h`)

IO - de/constructs all the IO components. (`hinic_hw_io.c`, `hinic_hw_io.h`)

HW device:

HW device - de/constructs the HW Interface, the MGMT components on the initialization of the driver and the IO components on the case of Interface UP/DOWN Events. (`hinic_hw_dev.c`, `hinic_hw_dev.h`)

hinic_dev contains the following components:

PCI ID table - Contains the supported PCI Vendor/Device IDs. (`hinic_pci_tbl.h`)

Port Commands - Send commands to the HW device for port management (MAC, Vlan, MTU, ...). (`hinic_port.c`, `hinic_port.h`)

Tx Queues - Logical Tx Queues that use the HW Send Queues for transmit. The Logical Tx queue is not dependent on the format of the HW Send Queue. (`hinic_tx.c`, `hinic_tx.h`)

Rx Queues - Logical Rx Queues that use the HW Receive Queues for receive. The Logical Rx queue is not dependent on the format of the HW Receive Queue. (`hinic_rx.c`, `hinic_rx.h`)

hinic_dev - de/constructs the Logical Tx and Rx Queues. (`hinic_main.c`, `hinic_dev.h`)

Miscellaneous

Common functions that are used by HW and Logical Device. (`hinic_common.c`, `hinic_common.h`)

Support

If an issue is identified with the released source code on the supported kernel with a supported adapter, email the specific information related to the issue to aviad.krawczyk@huawei.com.

6.5.19 Linux Base Driver for the Intel(R) PRO/100 Family of Adapters

June 1, 2018

Contents

- In This Release
- Identifying Your Adapter
- Building and Installation
- Driver Configuration Parameters
- Additional Configurations
- Known Issues
- Support

In This Release

This file describes the Linux Base Driver for the Intel(R) PRO/100 Family of Adapters. This driver includes support for Itanium(R)2-based systems.

For questions related to hardware requirements, refer to the documentation supplied with your Intel PRO/100 adapter.

The following features are now available in supported kernels:

- Native VLANs
- Channel Bonding (teaming)
- SNMP

Channel Bonding documentation can be found in the Linux kernel source: /Documentation/networking/bonding.rst

Identifying Your Adapter

For information on how to identify your adapter, and for the latest Intel network drivers, refer to the Intel Support website: <https://www.intel.com/support>

Driver Configuration Parameters

The default value for each parameter is generally the recommended setting, unless otherwise noted.

Rx Descriptors:

Number of receive descriptors. A receive descriptor is a data structure that describes a receive buffer and its attributes to the network controller. The data in the descriptor is used by the controller to write data from the controller to host memory. In the 3.x.x driver the valid range for this parameter is 64-256. The default value is 256. This parameter can be changed using the command:

```
ethtool -G eth? rx n
```

Where n is the number of desired Rx descriptors.

Tx Descriptors:

Number of transmit descriptors. A transmit descriptor is a data structure that describes a transmit buffer and its attributes to the network controller. The data in the descriptor is used by the controller to read data from the host memory to the controller. In the 3.x.x driver the valid range for this parameter is 64-256. The default value is 128. This parameter can be changed using the command:

```
ethtool -G eth? tx n
```

Where n is the number of desired Tx descriptors.

Speed/Duplex:

The driver auto-negotiates the link speed and duplex settings by default. The ethtool utility can be used as follows to force speed/duplex.:

```
ethtool -s eth? autoneg off speed {10|100} duplex {full|half}
```

NOTE: setting the speed/duplex to incorrect values will cause the link to fail.

Event Log Message Level:

The driver uses the message level flag to log events to syslog. The message level can be set at driver load time. It can also be set using the command:

```
ethtool -s eth? msghlvl n
```

Additional Configurations

Configuring the Driver on Different Distributions

Configuring a network driver to load properly when the system is started is distribution dependent. Typically, the configuration process involves adding an alias line to */etc/modprobe.d/*.conf* as well as editing other system startup scripts and/or configuration files. Many popular Linux distributions ship with tools to make these changes for you. To learn the proper way to configure a network device for your system, refer to your distribution documentation. If during this process you are asked for the driver or module name, the name for the Linux Base Driver for the Intel PRO/100 Family of Adapters is e100.

As an example, if you install the e100 driver for two PRO/100 adapters (eth0 and eth1), add the following to a configuration file in */etc/modprobe.d/*:

```
alias eth0 e100
alias eth1 e100
```

Viewing Link Messages

In order to see link messages and other Intel driver information on your console, you must set the dmesg level up to six. This can be done by entering the following on the command line before loading the e100 driver:

```
dmesg -n 6
```

If you wish to see all messages issued by the driver, including debug messages, set the dmesg level to eight.

NOTE: This setting is not saved across reboots.

ethtool

The driver utilizes the ethtool interface for driver configuration and diagnostics, as well as displaying statistical information. The ethtool version 1.6 or later is required for this functionality.

The latest release of ethtool can be found from <https://www.kernel.org/pub/software/network/ethtool/>

Enabling Wake on LAN (WoL)

WoL is provided through the ethtool utility. For instructions on enabling WoL with ethtool, refer to the ethtool man page. WoL will be enabled on the system during the next shut down or reboot. For this driver version, in order to enable WoL, the e100 driver must be loaded when shutting down or rebooting the system.

NAPI

NAPI (Rx polling mode) is supported in the e100 driver.

See [Documentation/networking/napi.rst](#) for more information.

Multiple Interfaces on Same Ethernet Broadcast Network

Due to the default ARP behavior on Linux, it is not possible to have one system on two IP networks in the same Ethernet broadcast domain (non-partitioned switch) behave as expected. All Ethernet interfaces will respond to IP traffic for any IP address assigned to the system. This results in unbalanced receive traffic.

If you have multiple interfaces in a server, either turn on ARP filtering by

- (1) entering:

```
echo 1 > /proc/sys/net/ipv4/conf/all/arp_filter
```

(this only works if your kernel's version is higher than 2.4.5), or

- (2) installing the interfaces in separate broadcast domains (either in different switches or in a switch partitioned to VLANs).

Support

For general information, go to the Intel support website at: <https://www.intel.com/support/>

If an issue is identified with the released source code on a supported kernel with a supported adapter, email the specific information related to the issue to intel-wired-lan@lists.osuosl.org.

6.5.20 Linux Base Driver for Intel(R) Ethernet Network Connection

Intel Gigabit Linux driver. Copyright(c) 1999 - 2013 Intel Corporation.

Contents

- Identifying Your Adapter
- Command Line Parameters
- Speed and Duplex Configuration
- Additional Configurations
- Support

Identifying Your Adapter

For more information on how to identify your adapter, go to the Adapter & Driver ID Guide at:

<http://support.intel.com/support/go/network/adapter/idguide.htm>

For the latest Intel network drivers for Linux, refer to the following website. In the search field, enter your adapter name or type, or use the networking link on the left to search for your adapter:

<http://support.intel.com/support/go/network/adapter/home.htm>

Command Line Parameters

The default value for each parameter is generally the recommended setting, unless otherwise noted.

NOTES:

For more information about the AutoNeg, Duplex, and Speed parameters, see the "Speed and Duplex Configuration" section in this document.

For more information about the InterruptThrottleRate, RxIntDelay, TxIntDelay, RxAbsIntDelay, and TxAbsIntDelay parameters, see the application note at: <http://www.intel.com/design/network/applnotes/ap450.htm>

AutoNeg

(Supported only on adapters with copper connections)

Valid Range

0x01-0x0F, 0x20-0x2F

Default Value

0x2F

This parameter is a bit-mask that specifies the speed and duplex settings advertised by the adapter. When this parameter is used, the Speed and Duplex parameters must not be specified.

NOTE:

Refer to the Speed and Duplex section of this readme for more information on the AutoNeg parameter.

Duplex

(Supported only on adapters with copper connections)

Valid Range

0-2 (0=auto-negotiate, 1=half, 2=full)

Default Value

0

This defines the direction in which data is allowed to flow. Can be either one or two-directional. If both Duplex and the link partner are set to auto-negotiate, the board auto-detects the correct duplex. If the link partner is forced (either full or half), Duplex defaults to half-duplex.

FlowControl

Valid Range

0-3 (0=none, 1=Rx only, 2=Tx only, 3=Rx&Tx)

Default Value

Reads flow control settings from the EEPROM

This parameter controls the automatic generation(Tx) and response(Rx) to Ethernet PAUSE frames.

InterruptThrottleRate

(not supported on Intel(R) 82542, 82543 or 82544-based adapters)

Valid Range

0,1,3,4,100-100000 (0=off, 1=dynamic, 3=dynamic conservative, 4=simplified balancing)

Default Value

3

The driver can limit the amount of interrupts per second that the adapter will generate for incoming packets. It does this by writing a value to the adapter that is based on the maximum amount of interrupts that the adapter will generate per second.

Setting InterruptThrottleRate to a value greater or equal to 100 will program the adapter to send out a maximum of that many interrupts per second, even if more packets have come in. This reduces interrupt load on the system and can lower CPU utilization under heavy load, but will increase latency as packets are not processed as quickly.

The default behaviour of the driver previously assumed a static InterruptThrottleRate value of 8000, providing a good fallback value for all traffic types, but lacking in small packet performance and latency. The hardware can handle many more small packets per second however, and for this reason an adaptive interrupt moderation algorithm was implemented.

Since 7.3.x, the driver has two adaptive modes (setting 1 or 3) in which it dynamically adjusts the InterruptThrottleRate value based on the traffic that it receives. After determining the type of incoming traffic in the last timeframe, it will adjust the InterruptThrottleRate to an appropriate value for that traffic.

The algorithm classifies the incoming traffic every interval into classes. Once the class is determined, the InterruptThrottleRate value is adjusted to suit that traffic type the best. There are three classes defined: "Bulk traffic", for large amounts of packets of normal size; "Low latency", for small amounts of traffic and/or a significant percentage of small packets; and "Lowest latency", for almost completely small packets or minimal traffic.

In dynamic conservative mode, the InterruptThrottleRate value is set to 4000 for traffic that falls in class "Bulk traffic". If traffic falls in the "Low latency" or "Lowest latency" class, the InterruptThrottleRate is increased stepwise to 20000. This default mode is suitable for most applications.

For situations where low latency is vital such as cluster or grid computing, the algorithm can reduce latency even more when InterruptThrottleRate is set to mode 1. In this mode, which operates the same as mode 3, the InterruptThrottleRate will be increased stepwise to 70000 for traffic in class "Lowest latency".

In simplified mode the interrupt rate is based on the ratio of TX and RX traffic. If the bytes per second rate is approximately equal, the interrupt rate will drop as low as 2000 interrupts per second. If the traffic is mostly transmit or mostly receive, the interrupt rate could be as high as 8000.

Setting InterruptThrottleRate to 0 turns off any interrupt moderation and may improve small packet latency, but is generally not suitable for bulk throughput traffic.

NOTE:

InterruptThrottleRate takes precedence over the TxAbsIntDelay and RxAbsIntDelay parameters. In other words, minimizing the receive and/or transmit absolute delays does not force the controller to generate more interrupts than what the Interrupt Throttle Rate allows.

CAUTION:

If you are using the Intel(R) PRO/1000 CT Network Connection (controller 82547), setting InterruptThrottleRate to a value greater than 75,000, may hang (stop transmitting) adapters under certain network conditions. If this occurs a NETDEV WATCHDOG message is logged in the system event log. In addition, the controller is automatically reset, restoring the network connection. To eliminate the potential for the hang, ensure that InterruptThrottleRate is set no greater than 75,000 and is not set to 0.

NOTE:

When e1000 is loaded with default settings and multiple adapters are in use simultaneously, the CPU utilization may increase non-linearly. In order to limit the CPU utilization without impacting the overall throughput, we recommend that you load the driver as follows:

```
modprobe e1000 InterruptThrottleRate=3000,3000,3000
```

This sets the InterruptThrottleRate to 3000 interrupts/sec for the first, second, and third instances of the driver. The range of 2000 to 3000 interrupts per second works on a majority of systems and is a good starting point, but the optimal value will be platform-specific. If CPU utilization is not a concern, use RX_POLLING (NAPI) and default driver settings.

RxDescriptors

Valid Range

- 48-256 for 82542 and 82543-based adapters
- 48-4096 for all other supported adapters

Default Value

256

This value specifies the number of receive buffer descriptors allocated by the driver. Increasing this value allows the driver to buffer more incoming packets, at the expense of increased system memory utilization.

Each descriptor is 16 bytes. A receive buffer is also allocated for each descriptor and can be either 2048, 4096, 8192, or 16384 bytes, depending on the MTU setting. The maximum MTU size is 16110.

NOTE:

MTU designates the frame size. It only needs to be set for Jumbo Frames. Depending

on the available system resources, the request for a higher number of receive descriptors may be denied. In this case, use a lower number.

RxIntDelay

Valid Range

0-65535 (0=off)

Default Value

0

This value delays the generation of receive interrupts in units of 1.024 microseconds. Receive interrupt reduction can improve CPU efficiency if properly tuned for specific network traffic. Increasing this value adds extra latency to frame reception and can end up decreasing the throughput of TCP traffic. If the system is reporting dropped receives, this value may be set too high, causing the driver to run out of available receive descriptors.

CAUTION:

When setting RxIntDelay to a value other than 0, adapters may hang (stop transmitting) under certain network conditions. If this occurs a NETDEV WATCHDOG message is logged in the system event log. In addition, the controller is automatically reset, restoring the network connection. To eliminate the potential for the hang ensure that RxIntDelay is set to 0.

RxAbsIntDelay

(This parameter is supported only on 82540, 82545 and later adapters.)

Valid Range

0-65535 (0=off)

Default Value

128

This value, in units of 1.024 microseconds, limits the delay in which a receive interrupt is generated. Useful only if RxIntDelay is non-zero, this value ensures that an interrupt is generated after the initial packet is received within the set amount of time. Proper tuning, along with RxIntDelay, may improve traffic throughput in specific network conditions.

Speed

(This parameter is supported only on adapters with copper connections.)

Valid Settings

0, 10, 100, 1000

Default Value

0 (auto-negotiate at all supported speeds)

Speed forces the line speed to the specified value in megabits per second (Mbps). If this parameter is not specified or is set to 0 and the link partner is set to auto-negotiate, the board will auto-detect the correct speed. Duplex should also be set when Speed is set to either 10 or 100.

TxDescriptors

Valid Range

- 48-256 for 82542 and 82543-based adapters
- 48-4096 for all other supported adapters

Default Value

256

This value is the number of transmit descriptors allocated by the driver. Increasing this value allows the driver to queue more transmits. Each descriptor is 16 bytes.

NOTE:

Depending on the available system resources, the request for a higher number of transmit descriptors may be denied. In this case, use a lower number.

TxIntDelay

Valid Range

0-65535 (0=off)

Default Value

8

This value delays the generation of transmit interrupts in units of 1.024 microseconds. Transmit interrupt reduction can improve CPU efficiency if properly tuned for specific network traffic. If the system is reporting dropped transmits, this value may be set too high causing the driver to run out of available transmit descriptors.

TxAbsIntDelay

(This parameter is supported only on 82540, 82545 and later adapters.)

Valid Range

0-65535 (0=off)

Default Value

32

This value, in units of 1.024 microseconds, limits the delay in which a transmit interrupt is generated. Useful only if TxIntDelay is non-zero, this value ensures that an interrupt is generated after the initial packet is sent on the wire within the set amount of time. Proper tuning, along with TxIntDelay, may improve traffic throughput in specific network conditions.

XsumRX

(This parameter is NOT supported on the 82542-based adapter.)

Valid Range

0-1

Default Value

1

A value of '1' indicates that the driver should enable IP checksum offload for received packets (both UDP and TCP) to the adapter hardware.

Copybreak

Valid Range

0-xxxxxxx (0=off)

Default Value

256

Usage

```
modprobe e1000.ko copybreak=128
```

Driver copies all packets below or equaling this size to a fresh RX buffer before handing it up the stack.

This parameter is different than other parameters, in that it is a single (not 1,1,1 etc.) parameter applied to all driver instances and it is also available during runtime at /sys/module/e1000/parameters/copybreak

SmartPowerDownEnable

Valid Range

0-1

Default Value

0 (disabled)

Allows PHY to turn off in lower power states. The user can turn off this parameter in supported chipsets.

Speed and Duplex Configuration

Three keywords are used to control the speed and duplex configuration. These keywords are Speed, Duplex, and AutoNeg.

If the board uses a fiber interface, these keywords are ignored, and the fiber interface board only links at 1000 Mbps full-duplex.

For copper-based boards, the keywords interact as follows:

- The default operation is auto-negotiate. The board advertises all supported speed and duplex combinations, and it links at the highest common speed and duplex mode IF the link partner is set to auto-negotiate.

- If Speed = 1000, limited auto-negotiation is enabled and only 1000 Mbps is advertised (The 1000BaseT spec requires auto-negotiation.)
- If Speed = 10 or 100, then both Speed and Duplex should be set. Auto-negotiation is disabled, and the AutoNeg parameter is ignored. Partner SHOULD also be forced.

The AutoNeg parameter is used when more control is required over the auto-negotiation process. It should be used when you wish to control which speed and duplex combinations are advertised during the auto-negotiation process.

The parameter may be specified as either a decimal or hexadecimal value as determined by the bitmap below.

Bit position	7	6	5	4	3	2	1	0
Decimal Value	128	64	32	16	8	4	2	1
Hex value	80	40	20	10	8	4	2	1
Speed (Mbps)	N/A	N/A	1000	N/A	100	100	10	10
Duplex			Full		Full	Half	Full	Half

Some examples of using AutoNeg:

```
modprobe e1000 AutoNeg=0x01 (Restricts autonegotiation to 10 Half)
modprobe e1000 AutoNeg=1 (Same as above)
modprobe e1000 AutoNeg=0x02 (Restricts autonegotiation to 10 Full)
modprobe e1000 AutoNeg=0x03 (Restricts autonegotiation to 10 Half or 10 Full)
modprobe e1000 AutoNeg=0x04 (Restricts autonegotiation to 100 Half)
modprobe e1000 AutoNeg=0x05 (Restricts autonegotiation to 10 Half or 100
Half)
modprobe e1000 AutoNeg=0x020 (Restricts autonegotiation to 1000 Full)
modprobe e1000 AutoNeg=32 (Same as above)
```

Note that when this parameter is used, Speed and Duplex must not be specified.

If the link partner is forced to a specific speed and duplex, then this parameter should not be used. Instead, use the Speed and Duplex parameters previously mentioned to force the adapter to the same speed and duplex.

Additional Configurations

Jumbo Frames

Jumbo Frames support is enabled by changing the MTU to a value larger than the default of 1500. Use the ifconfig command to increase the MTU size. For example:

```
ifconfig eth<x> mtu 9000 up
```

This setting is not saved across reboots. It can be made permanent if you add:

```
MTU=9000
```

to the file /etc/sysconfig/network-scripts/ifcfg-eth<x>. This example applies to the Red Hat distributions; other distributions may store this setting in a different location.

Notes:

Degradation in throughput performance may be observed in some Jumbo frames environments. If this is observed, increasing the application's socket buffer size and/or increasing the /proc/sys/net/ipv4/tcp_*mem entry values may help. See the specific application manual and /usr/src/linux*/Documentation/networking/ip-sysctl.txt for more details.

- The maximum MTU setting for Jumbo Frames is 16110. This value coincides with the maximum Jumbo Frames size of 16128.
- Using Jumbo frames at 10 or 100 Mbps is not supported and may result in poor performance or loss of link.
- Adapters based on the Intel(R) 82542 and 82573V/E controller do not support Jumbo Frames. These correspond to the following product names:

Intel(R) PRO/1000 Gigabit Server Adapter
Intel(R) PRO/1000 PM Network Connection

ethtool

The driver utilizes the ethtool interface for driver configuration and diagnostics, as well as displaying statistical information. The ethtool version 1.6 or later is required for this functionality.

The latest release of ethtool can be found from <https://www.kernel.org/pub/software/network/ethtool/>

Enabling Wake on LAN (WoL)

WoL is configured through the ethtool utility.

WoL will be enabled on the system during the next shut down or reboot. For this driver version, in order to enable WoL, the e1000 driver must be loaded when shutting down or rebooting the system.

Support

For general information, go to the Intel support website at: <http://support.intel.com>

If an issue is identified with the released source code on the supported kernel with a supported adapter, email the specific information related to the issue to intel-wired-lan@lists.osuosl.org.

6.5.21 Linux Driver for Intel(R) Ethernet Network Connection

Intel Gigabit Linux driver. Copyright(c) 2008-2018 Intel Corporation.

Contents

- Identifying Your Adapter
- Command Line Parameters
- Additional Configurations
- Support

Identifying Your Adapter

For information on how to identify your adapter, and for the latest Intel network drivers, refer to the Intel Support website: <https://www.intel.com/support>

Command Line Parameters

If the driver is built as a module, the following optional parameters are used by entering them on the command line with the modprobe command using this syntax:

```
modprobe e1000e [<option>=<VAL1>,<VAL2>,...]
```

There needs to be a <VAL#> for each network port in the system supported by this driver. The values will be applied to each instance, in function order. For example:

```
modprobe e1000e InterruptThrottleRate=16000,16000
```

In this case, there are two network ports supported by e1000e in the system. The default value for each parameter is generally the recommended setting, unless otherwise noted.

NOTE: A descriptor describes a data buffer and attributes related to the data buffer. This information is accessed by the hardware.

InterruptThrottleRate

Valid Range

0,1,3,4,100-100000

Default Value

3

Interrupt Throttle Rate controls the number of interrupts each interrupt vector can generate per second. Increasing ITR lowers latency at the cost of increased CPU utilization, though it may help throughput in some circumstances.

Setting InterruptThrottleRate to a value greater or equal to 100 will program the adapter to send out a maximum of that many interrupts per second, even if more packets have come in.

This reduces interrupt load on the system and can lower CPU utilization under heavy load, but will increase latency as packets are not processed as quickly.

The default behaviour of the driver previously assumed a static InterruptThrottleRate value of 8000, providing a good fallback value for all traffic types, but lacking in small packet performance and latency. The hardware can handle many more small packets per second however, and for this reason an adaptive interrupt moderation algorithm was implemented.

The driver has two adaptive modes (setting 1 or 3) in which it dynamically adjusts the InterruptThrottleRate value based on the traffic that it receives. After determining the type of incoming traffic in the last timeframe, it will adjust the InterruptThrottleRate to an appropriate value for that traffic.

The algorithm classifies the incoming traffic every interval into classes. Once the class is determined, the InterruptThrottleRate value is adjusted to suit that traffic type the best. There are three classes defined: "Bulk traffic", for large amounts of packets of normal size; "Low latency", for small amounts of traffic and/or a significant percentage of small packets; and "Lowest latency", for almost completely small packets or minimal traffic.

- **0: Off**

Turns off any interrupt moderation and may improve small packet latency. However, this is generally not suitable for bulk throughput traffic due to the increased CPU utilization of the higher interrupt rate.

- **1: Dynamic mode**

This mode attempts to moderate interrupts per vector while maintaining very low latency. This can sometimes cause extra CPU utilization. If planning on deploying e1000e in a latency sensitive environment, this parameter should be considered.

- **3: Dynamic Conservative mode (default)**

In dynamic conservative mode, the InterruptThrottleRate value is set to 4000 for traffic that falls in class "Bulk traffic". If traffic falls in the "Low latency" or "Lowest latency" class, the InterruptThrottleRate is increased stepwise to 20000. This default mode is suitable for most applications.

- **4: Simplified Balancing mode**

In simplified mode the interrupt rate is based on the ratio of TX and RX traffic. If the bytes per second rate is approximately equal, the interrupt rate will drop as low as 2000 interrupts per second. If the traffic is mostly transmit or mostly receive, the interrupt rate could be as high as 8000.

- **100-100000:**

Setting InterruptThrottleRate to a value greater or equal to 100 will program the adapter to send at most that many interrupts per second, even if more packets have come in. This reduces interrupt load on the system and can lower CPU utilization under heavy load, but will increase latency as packets are not processed as quickly.

NOTE: InterruptThrottleRate takes precedence over the TxAbsIntDelay and RxAbsIntDelay parameters. In other words, minimizing the receive and/or transmit absolute delays does not force the controller to generate more interrupts than what the Interrupt Throttle Rate allows.

RxIntDelay

Valid Range

0-65535 (0=off)

Default Value

0

This value delays the generation of receive interrupts in units of 1.024 microseconds. Receive interrupt reduction can improve CPU efficiency if properly tuned for specific network traffic. Increasing this value adds extra latency to frame reception and can end up decreasing the throughput of TCP traffic. If the system is reporting dropped receives, this value may be set too high, causing the driver to run out of available receive descriptors.

CAUTION: When setting RxIntDelay to a value other than 0, adapters may hang (stop transmitting) under certain network conditions. If this occurs a NETDEV WATCHDOG message is logged in the system event log. In addition, the controller is automatically reset, restoring the network connection. To eliminate the potential for the hang ensure that RxIntDelay is set to 0.

RxAbsIntDelay

Valid Range

0-65535 (0=off)

Default Value

8

This value, in units of 1.024 microseconds, limits the delay in which a receive interrupt is generated. This value ensures that an interrupt is generated after the initial packet is received within the set amount of time, which is useful only if RxIntDelay is non-zero. Proper tuning, along with RxIntDelay, may improve traffic throughput in specific network conditions.

TxIntDelay

Valid Range

0-65535 (0=off)

Default Value

8

This value delays the generation of transmit interrupts in units of 1.024 microseconds. Transmit interrupt reduction can improve CPU efficiency if properly tuned for specific network traffic. If the system is reporting dropped transmits, this value may be set too high causing the driver to run out of available transmit descriptors.

TxAbsIntDelay

Valid Range

0-65535 (0=off)

Default Value

32

This value, in units of 1.024 microseconds, limits the delay in which a transmit interrupt is generated. It is useful only if TxIntDelay is non-zero. It ensures that an interrupt is generated after the initial Packet is sent on the wire within the set amount of time. Proper tuning, along with TxIntDelay, may improve traffic throughput in specific network conditions.

copybreak

Valid Range

0-xxxxxxxx (0=off)

Default Value

256

The driver copies all packets below or equaling this size to a fresh receive buffer before handing it up the stack. This parameter differs from other parameters because it is a single (not 1,1,1 etc.) parameter applied to all driver instances and it is also available during runtime at /sys/module/e1000e/parameters/copybreak.

To use copybreak, type:

```
modprobe e1000e.ko copybreak=128
```

SmartPowerDownEnable

Valid Range

0,1

Default Value

0 (disabled)

Allows the PHY to turn off in lower power states. The user can turn off this parameter in supported chipsets.

KumeranLockLoss

Valid Range

0,1

Default Value

1 (enabled)

This workaround skips resetting the PHY at shutdown for the initial silicon releases of ICH8 systems.

IntMode

Valid Range

0-2

Default Value

0

Value	Interrupt Mode
0	Legacy
1	MSI
2	MSI-X

IntMode allows load time control over the type of interrupt registered for by the driver. MSI-X is required for multiple queue support, and some kernels and combinations of kernel .config options will force a lower level of interrupt support.

This command will show different values for each type of interrupt:

```
cat /proc/interrupts
```

CrcStripping

Valid Range

0,1

Default Value

1 (enabled)

Strip the CRC from received packets before sending up the network stack. If you have a machine with a BMC enabled but cannot receive IPMI traffic after loading or enabling the driver, try disabling this feature.

WriteProtectNVM

Valid Range

0,1

Default Value

1 (enabled)

If set to 1, configure the hardware to ignore all write/erase cycles to the GbE region in the ICHx NVM (in order to prevent accidental corruption of the NVM). This feature can be disabled by setting the parameter to 0 during initial driver load.

NOTE: The machine must be power cycled (full off/on) when enabling NVM writes via setting the parameter to zero. Once the NVM has been locked (via the parameter at 1 when the driver loads) it cannot be unlocked except via power cycle.

Debug

Valid Range

0-16 (0=none,...,16=all)

Default Value

0

This parameter adjusts the level of debug messages displayed in the system logs.

Additional Features and Configurations

Jumbo Frames

Jumbo Frames support is enabled by changing the Maximum Transmission Unit (MTU) to a value larger than the default value of 1500.

Use the ifconfig command to increase the MTU size. For example, enter the following where <x> is the interface number:

```
ifconfig eth<x> mtu 9000 up
```

Alternatively, you can use the ip command as follows:

```
ip link set mtu 9000 dev eth<x>
ip link set up dev eth<x>
```

This setting is not saved across reboots. The setting change can be made permanent by adding 'MTU=9000' to the file:

- For RHEL: /etc/sysconfig/network-scripts/ifcfg-eth<x>
- For SLES: /etc/sysconfig/network/<config_file>

NOTE: The maximum MTU setting for Jumbo Frames is 8996. This value coincides with the maximum Jumbo Frames size of 9018 bytes.

NOTE: Using Jumbo frames at 10 or 100 Mbps is not supported and may result in poor performance or loss of link.

NOTE: The following adapters limit Jumbo Frames sized packets to a maximum of 4088 bytes:

- Intel(R) 82578DM Gigabit Network Connection
- Intel(R) 82577LM Gigabit Network Connection

The following adapters do not support Jumbo Frames:

- Intel(R) PRO/1000 Gigabit Server Adapter
- Intel(R) PRO/1000 PM Network Connection
- Intel(R) 82562G 10/100 Network Connection
- Intel(R) 82562G-2 10/100 Network Connection
- Intel(R) 82562GT 10/100 Network Connection
- Intel(R) 82562GT-2 10/100 Network Connection

- Intel(R) 82562V 10/100 Network Connection
- Intel(R) 82562V-2 10/100 Network Connection
- Intel(R) 82566DC Gigabit Network Connection
- Intel(R) 82566DC-2 Gigabit Network Connection
- Intel(R) 82566DM Gigabit Network Connection
- Intel(R) 82566MC Gigabit Network Connection
- Intel(R) 82566MM Gigabit Network Connection
- Intel(R) 82567V-3 Gigabit Network Connection
- Intel(R) 82577LC Gigabit Network Connection
- Intel(R) 82578DC Gigabit Network Connection

NOTE: Jumbo Frames cannot be configured on an 82579-based Network device if MACSec is enabled on the system.

ethtool

The driver utilizes the ethtool interface for driver configuration and diagnostics, as well as displaying statistical information. The latest ethtool version is required for this functionality. Download it at:

<https://www.kernel.org/pub/software/network/ethtool/>

NOTE: When validating enable/disable tests on some parts (for example, 82578), it is necessary to add a few seconds between tests when working with ethtool.

Speed and Duplex Configuration

In addressing speed and duplex configuration issues, you need to distinguish between copper-based adapters and fiber-based adapters.

In the default mode, an Intel(R) Ethernet Network Adapter using copper connections will attempt to auto-negotiate with its link partner to determine the best setting. If the adapter cannot establish link with the link partner using auto-negotiation, you may need to manually configure the adapter and link partner to identical settings to establish link and pass packets. This should only be needed when attempting to link with an older switch that does not support auto-negotiation or one that has been forced to a specific speed or duplex mode. Your link partner must match the setting you choose. 1 Gbps speeds and higher cannot be forced. Use the autonegotiation advertising setting to manually set devices for 1 Gbps and higher.

Speed, duplex, and autonegotiation advertising are configured through the ethtool utility.

Caution: Only experienced network administrators should force speed and duplex or change autonegotiation advertising manually. The settings at the switch must always match the adapter settings. Adapter performance may suffer or your adapter may not operate if you configure the adapter differently from your switch.

An Intel(R) Ethernet Network Adapter using fiber-based connections, however, will not attempt to auto-negotiate with its link partner since those adapters operate only in full duplex and only at their native speed.

Enabling Wake on LAN (WoL)

WoL is configured through the ethtool utility.

WoL will be enabled on the system during the next shut down or reboot. For this driver version, in order to enable WoL, the e1000e driver must be loaded prior to shutting down or suspending the system.

NOTE: Wake on LAN is only supported on port A for the following devices: - Intel(R) PRO/1000 PT Dual Port Network Connection - Intel(R) PRO/1000 PT Dual Port Server Connection - Intel(R) PRO/1000 PT Dual Port Server Adapter - Intel(R) PRO/1000 PF Dual Port Server Adapter - Intel(R) PRO/1000 PT Quad Port Server Adapter - Intel(R) Gigabit PT Quad Port Server ExpressModule

Support

For general information, go to the Intel support website at: <https://www.intel.com/support/>

If an issue is identified with the released source code on a supported kernel with a supported adapter, email the specific information related to the issue to intel-wired-lan@lists.osuosl.org.

6.5.22 Linux Base Driver for Intel(R) Ethernet Multi-host Controller

August 20, 2018 Copyright(c) 2015-2018 Intel Corporation.

Contents

- Identifying Your Adapter
- Additional Configurations
- Performance Tuning
- Known Issues
- Support

Identifying Your Adapter

The driver in this release is compatible with devices based on the Intel(R) Ethernet Multi-host Controller.

For information on how to identify your adapter, and for the latest Intel network drivers, refer to the Intel Support website: <https://www.intel.com/support/>

Flow Control

The Intel(R) Ethernet Switch Host Interface Driver does not support Flow Control. It will not send pause frames. This may result in dropped frames.

Virtual Functions (VFs)

Use sysfs to enable VFs. Valid Range: 0-64

For example:

```
echo $num_vf_enabled > /sys/class/net/$dev/device/sriov_numvfs //enable VFs  
echo 0 > /sys/class/net/$dev/device/sriov_numvfs //disable VFs
```

NOTE: Neither the device nor the driver control how VFs are mapped into config space. Bus layout will vary by operating system. On operating systems that support it, you can check sysfs to find the mapping.

NOTE: When SR-IOV mode is enabled, hardware VLAN filtering and VLAN tag stripping/insertion will remain enabled. Please remove the old VLAN filter before the new VLAN filter is added. For example:

```
ip link set eth0 vf 0 vlan 100      // set vlan 100 for VF 0  
ip link set eth0 vf 0 vlan 0      // Delete vlan 100  
ip link set eth0 vf 0 vlan 200      // set a new vlan 200 for VF 0
```

Additional Features and Configurations

Jumbo Frames

Jumbo Frames support is enabled by changing the Maximum Transmission Unit (MTU) to a value larger than the default value of 1500.

Use the ifconfig command to increase the MTU size. For example, enter the following where <x> is the interface number:

```
ifconfig eth<x> mtu 9000 up
```

Alternatively, you can use the ip command as follows:

```
ip link set mtu 9000 dev eth<x>  
ip link set up dev eth<x>
```

This setting is not saved across reboots. The setting change can be made permanent by adding 'MTU=9000' to the file:

- For RHEL: /etc/sysconfig/network-scripts/ifcfg-eth<x>
- For SLES: /etc/sysconfig/network/<config_file>

NOTE: The maximum MTU setting for Jumbo Frames is 15342. This value coincides with the maximum Jumbo Frames size of 15364 bytes.

NOTE: This driver will attempt to use multiple page sized buffers to receive each jumbo packet. This should help to avoid buffer starvation issues when allocating receive packets.

Generic Receive Offload, aka GRO

The driver supports the in-kernel software implementation of GRO. GRO has shown that by coalescing Rx traffic into larger chunks of data, CPU utilization can be significantly reduced when under large Rx load. GRO is an evolution of the previously-used LRO interface. GRO is able to coalesce other protocols besides TCP. It's also safe to use with configurations that are problematic for LRO, namely bridging and iSCSI.

Supported ethtool Commands and Options for Filtering

-n --show-nfc

Retrieves the receive network flow classification configurations.

rx-flow-hash tcp4|udp4|ah4|esp4|sctp4|tcp6|udp6|ah6|esp6|sctp6

Retrieves the hash options for the specified network traffic type.

-N --config-nfc

Configures the receive network flow classification.

rx-flow-hash tcp4|udp4|ah4|esp4|sctp4|tcp6|udp6|ah6|esp6|sctp6 m|v|t|s|d|f|n|r

Configures the hash options for the specified network traffic type.

- udp4: UDP over IPv4
- udp6: UDP over IPv6
- f Hash on bytes 0 and 1 of the Layer 4 header of the rx packet.
- n Hash on bytes 2 and 3 of the Layer 4 header of the rx packet.

Known Issues/Troubleshooting

Enabling SR-IOV in a 64-bit Microsoft Windows Server 2012/R2 guest OS under Linux KVM

KVM Hypervisor/VMM supports direct assignment of a PCIe device to a VM. This includes traditional PCIe devices, as well as SR-IOV-capable devices based on the Intel Ethernet Controller XL710.

Support

For general information, go to the Intel support website at: <https://www.intel.com/support/>

If an issue is identified with the released source code on a supported kernel with a supported adapter, email the specific information related to the issue to intel-wired-lan@lists.osuosl.org.

6.5.23 idpf Linux* Base Driver for the Intel(R) Infrastructure Data Path Function

Intel idpf Linux driver. Copyright(C) 2023 Intel Corporation.

Contents

- *idpf Linux* Base Driver for the Intel(R) Infrastructure Data Path Function*
 - *Identifying Your Adapter*
 - *Additional Features and Configurations*
 - * *ethtool*
 - * *Viewing Link Messages*
 - * *Jumbo Frames*
 - *Performance Optimization*
 - * *Interrupt Rate Limiting*
 - * *Virtualized Environments*
 - *Support*
 - *Trademarks*

The idpf driver serves as both the Physical Function (PF) and Virtual Function (VF) driver for the Intel(R) Infrastructure Data Path Function.

Driver information can be obtained using ethtool, lspci, and ip.

For questions related to hardware requirements, refer to the documentation supplied with your Intel adapter. All hardware requirements listed apply to use with Linux.

Identifying Your Adapter

For information on how to identify your adapter, and for the latest Intel network drivers, refer to the Intel Support website: <http://www.intel.com/support>

Additional Features and Configurations

ethtool

The driver utilizes the ethtool interface for driver configuration and diagnostics, as well as displaying statistical information. The latest ethtool version is required for this functionality. If you don't have one yet, you can obtain it at: <https://kernel.org/pub/software/network/ethtool/>

Viewing Link Messages

Link messages will not be displayed to the console if the distribution is restricting system messages. In order to see network driver link messages on your console, set dmesg to eight by entering the following:

```
# dmesg -n 8
```

Note: This setting is not saved across reboots.

Jumbo Frames

Jumbo Frames support is enabled by changing the Maximum Transmission Unit (MTU) to a value larger than the default value of 1500.

Use the ip command to increase the MTU size. For example, enter the following where <ethX> is the interface number:

```
# ip link set mtu 9000 dev <ethX>
# ip link set up dev <ethX>
```

Note: The maximum MTU setting for jumbo frames is 9706. This corresponds to the maximum jumbo frame size of 9728 bytes.

Note: This driver will attempt to use multiple page sized buffers to receive each jumbo packet. This should help to avoid buffer starvation issues when allocating receive packets.

Note: Packet loss may have a greater impact on throughput when you use jumbo frames. If you observe a drop in performance after enabling jumbo frames, enabling flow control may mitigate the issue.

Performance Optimization

Driver defaults are meant to fit a wide variety of workloads, but if further optimization is required, we recommend experimenting with the following settings.

Interrupt Rate Limiting

This driver supports an adaptive interrupt throttle rate (ITR) mechanism that is tuned for general workloads. The user can customize the interrupt rate control for specific workloads, via ethtool, adjusting the number of microseconds between interrupts.

To set the interrupt rate manually, you must disable adaptive mode:

```
# ethtool -C <ethX> adaptive-rx off adaptive-tx off
```

For lower CPU utilization:

- Disable adaptive ITR and lower Rx and Tx interrupts. The examples below affect every queue of the specified interface.
- Setting rx-usecs and tx-usecs to 80 will limit interrupts to about 12,500 interrupts per second per queue:

```
# ethtool -C <ethX> adaptive-rx off adaptive-tx off rx-usecs 80  
tx-usecs 80
```

For reduced latency:

- Disable adaptive ITR and ITR by setting rx-usecs and tx-usecs to 0 using ethtool:

```
# ethtool -C <ethX> adaptive-rx off adaptive-tx off rx-usecs 0  
tx-usecs 0
```

Per-queue interrupt rate settings:

- The following examples are for queues 1 and 3, but you can adjust other queues.
- To disable Rx adaptive ITR and set static Rx ITR to 10 microseconds or about 100,000 interrupts/second, for queues 1 and 3:

```
# ethtool --per-queue <ethX> queue_mask 0xa --coalesce adaptive-rx off  
rx-usecs 10
```

- To show the current coalesce settings for queues 1 and 3:

```
# ethtool --per-queue <ethX> queue_mask 0xa --show-coalesce
```

Virtualized Environments

In addition to the other suggestions in this section, the following may be helpful to optimize performance in VMs.

- Using the appropriate mechanism (vcpupin) in the VM, pin the CPUs to individual LCPUs, making sure to use a set of CPUs included in the device's local_cpulist: /sys/class/net/<ethX>/device/local_cpulist.
- Configure as many Rx/Tx queues in the VM as available. (See the idpf driver documentation for the number of queues supported.) For example:

```
# ethtool -L <virt_interface> rx <max> tx <max>
```

Support

For general information, go to the Intel support website at: <http://www.intel.com/support/>

If an issue is identified with the released source code on a supported kernel with a supported adapter, email the specific information related to the issue to intel-wired-lan@lists.osuosl.org.

Trademarks

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and/or other countries.

- Other names and brands may be claimed as the property of others.

6.5.24 Linux Base Driver for Intel(R) Ethernet Network Connection

Intel Gigabit Linux driver. Copyright(c) 1999-2018 Intel Corporation.

Contents

- Identifying Your Adapter
- Command Line Parameters
- Additional Configurations
- Support

Identifying Your Adapter

For information on how to identify your adapter, and for the latest Intel network drivers, refer to the Intel Support website: <https://www.intel.com/support>

Command Line Parameters

If the driver is built as a module, the following optional parameters are used by entering them on the command line with the modprobe command using this syntax:

```
modprobe igb [<option>=<VAL1>,<VAL2>,...]
```

There needs to be a <VAL#> for each network port in the system supported by this driver. The values will be applied to each instance, in function order. For example:

```
modprobe igb max_vfs=2,4
```

In this case, there are two network ports supported by igb in the system.

NOTE: A descriptor describes a data buffer and attributes related to the data buffer. This information is accessed by the hardware.

max_vfs

Valid Range

0-7

This parameter adds support for SR-IOV. It causes the driver to spawn up to max_vfs worth of virtual functions. If the value is greater than 0 it will also force the VMDq parameter to be 1 or more.

The parameters for the driver are referenced by position. Thus, if you have a dual port adapter, or more than one adapter in your system, and want N virtual functions per port, you must specify a number for each port with each parameter separated by a comma. For example:

```
modprobe igb max_vfs=4
```

This will spawn 4 VFs on the first port.

```
modprobe igb max_vfs=2,4
```

This will spawn 2 VFs on the first port and 4 VFs on the second port.

NOTE: Caution must be used in loading the driver with these parameters. Depending on your system configuration, number of slots, etc., it is impossible to predict in all cases where the positions would be on the command line.

NOTE: Neither the device nor the driver control how VFs are mapped into config space. Bus layout will vary by operating system. On operating systems that support it, you can check sysfs to find the mapping.

NOTE: When either SR-IOV mode or VMDq mode is enabled, hardware VLAN filtering and VLAN tag stripping/insertion will remain enabled. Please remove the old VLAN filter before the new VLAN filter is added. For example:

```
ip link set eth0 vf 0 vlan 100      // set vlan 100 for VF 0
ip link set eth0 vf 0 vlan 0       // Delete vlan 100
ip link set eth0 vf 0 vlan 200      // set a new vlan 200 for VF 0
```

Debug

Valid Range

0-16 (0=none,...,16=all)

Default Value

0

This parameter adjusts the level debug messages displayed in the system logs.

Additional Features and Configurations

Jumbo Frames

Jumbo Frames support is enabled by changing the Maximum Transmission Unit (MTU) to a value larger than the default value of 1500.

Use the ifconfig command to increase the MTU size. For example, enter the following where <x> is the interface number:

```
ifconfig eth<x> mtu 9000 up
```

Alternatively, you can use the ip command as follows:

```
ip link set mtu 9000 dev eth<x>
ip link set up dev eth<x>
```

This setting is not saved across reboots. The setting change can be made permanent by adding 'MTU=9000' to the file:

- For RHEL: /etc/sysconfig/network-scripts/ifcfg-eth<x>
- For SLES: /etc/sysconfig/network/<config_file>

NOTE: The maximum MTU setting for Jumbo Frames is 9216. This value coincides with the maximum Jumbo Frames size of 9234 bytes.

NOTE: Using Jumbo frames at 10 or 100 Mbps is not supported and may result in poor performance or loss of link.

ethtool

The driver utilizes the ethtool interface for driver configuration and diagnostics, as well as displaying statistical information. The latest ethtool version is required for this functionality. Download it at:

<https://www.kernel.org/pub/software/network/ethtool/>

Enabling Wake on LAN (WoL)

WoL is configured through the ethtool utility.

WoL will be enabled on the system during the next shut down or reboot. For this driver version, in order to enable WoL, the igb driver must be loaded prior to shutting down or suspending the system.

NOTE: Wake on LAN is only supported on port A of multi-port devices. Also Wake On LAN is not supported for the following device: - Intel(R) Gigabit VT Quad Port Server Adapter

Multiqueue

In this mode, a separate MSI-X vector is allocated for each queue and one for "other" interrupts such as link status change and errors. All interrupts are throttled via interrupt moderation. Interrupt moderation must be used to avoid interrupt storms while the driver is processing one interrupt. The moderation value should be at least as large as the expected time for the driver to process an interrupt. Multiqueue is off by default.

REQUIREMENTS: MSI-X support is required for Multiqueue. If MSI-X is not found, the system will fallback to MSI or to Legacy interrupts. This driver supports receive multiqueue on all kernels that support MSI-X.

NOTE: On some kernels a reboot is required to switch between single queue mode and multi-queue mode or vice-versa.

MAC and VLAN anti-spoofing feature

When a malicious driver attempts to send a spoofed packet, it is dropped by the hardware and not transmitted.

An interrupt is sent to the PF driver notifying it of the spoof attempt. When a spoofed packet is detected, the PF driver will send the following message to the system log (displayed by the "dmesg" command): Spoof event(s) detected on VF(n), where n = the VF that attempted to do the spoofing

Setting MAC Address, VLAN and Rate Limit Using IProute2 Tool

You can set a MAC address of a Virtual Function (VF), a default VLAN and the rate limit using the IProute2 tool. Download the latest version of the IProute2 tool from Sourceforge if your version does not have all the features you require.

Credit Based Shaper (Qav Mode)

When enabling the CBS qdisc in the hardware offload mode, traffic shaping using the CBS (described in the IEEE 802.1Q-2018 Section 8.6.8.2 and discussed in the Annex L) algorithm will run in the i210 controller, so it's more accurate and uses less CPU.

When using offloaded CBS, and the traffic rate obeys the configured rate (doesn't go above it), CBS should have little to no effect in the latency.

The offloaded version of the algorithm has some limits, caused by how the idle slope is expressed in the adapter's registers. It can only represent idle slopes in 16.38431 kbps units, which means that if a idle slope of 2576kbps is requested, the controller will be configured to use a idle slope of ~2589 kbps, because the driver rounds the value up. For more details, see the comments on `igb_config_tx_modes()`.

NOTE: This feature is exclusive to i210 models.

Support

For general information, go to the Intel support website at: <https://www.intel.com/support/>

If an issue is identified with the released source code on a supported kernel with a supported adapter, email the specific information related to the issue to intel-wired-lan@lists.osuosl.org.

6.5.25 Linux Base Virtual Function Driver for Intel(R) 1G Ethernet

Intel Gigabit Virtual Function Linux driver. Copyright(c) 1999-2018 Intel Corporation.

Contents

- Identifying Your Adapter
- Additional Configurations
- Support

This driver supports Intel 82576-based virtual function devices-based virtual function devices that can only be activated on kernels that support SR-IOV.

SR-IOV requires the correct platform and OS support.

The guest OS loading this driver must support MSI-X interrupts.

For questions related to hardware requirements, refer to the documentation supplied with your Intel adapter. All hardware requirements listed apply to use with Linux.

Driver information can be obtained using ethtool, lspci, and ifconfig. Instructions on updating ethtool can be found in the section Additional Configurations later in this document.

NOTE: There is a limit of a total of 32 shared VLANs to 1 or more VFs.

Identifying Your Adapter

For information on how to identify your adapter, and for the latest Intel network drivers, refer to the Intel Support website: <https://www.intel.com/support/>

Additional Features and Configurations

ethtool

The driver utilizes the ethtool interface for driver configuration and diagnostics, as well as displaying statistical information. The latest ethtool version is required for this functionality. Download it at:

<https://www.kernel.org/pub/software/network/ethtool/>

Support

For general information, go to the Intel support website at: <https://www.intel.com/support/>

If an issue is identified with the released source code on a supported kernel with a supported adapter, email the specific information related to the issue to intel-wired-lan@lists.osuosl.org.

6.5.26 Linux Base Driver for the Intel(R) Ethernet 10 Gigabit PCI Express Adapters

Intel 10 Gigabit Linux driver. Copyright(c) 1999-2018 Intel Corporation.

Contents

- Identifying Your Adapter
- Command Line Parameters
- Additional Configurations
- Known Issues
- Support

Identifying Your Adapter

The driver is compatible with devices based on the following:

- Intel(R) Ethernet Controller 82598
- Intel(R) Ethernet Controller 82599
- Intel(R) Ethernet Controller X520
- Intel(R) Ethernet Controller X540
- Intel(R) Ethernet Controller x550
- Intel(R) Ethernet Controller X552
- Intel(R) Ethernet Controller X553

For information on how to identify your adapter, and for the latest Intel network drivers, refer to the Intel Support website: <https://www.intel.com/support/>

SFP+ Devices with Pluggable Optics

82599-BASED ADAPTERS

NOTES: - If your 82599-based Intel(R) Network Adapter came with Intel optics or is an Intel(R) Ethernet Server Adapter X520-2, then it only supports Intel optics and/or the direct attach cables listed below. - When 82599-based SFP+ devices are connected back to back, they should be set to the same Speed setting via ethtool. Results may vary if you mix speed settings.

Supplier	Type	Part Numbers
SR Modules		
Intel	DUAL RATE 1G/10G SFP+ SR (bailed)	FTLX8571D3BCV-IT
Intel	DUAL RATE 1G/10G SFP+ SR (bailed)	AFBR-703SDZ-IN2
Intel	DUAL RATE 1G/10G SFP+ SR (bailed)	AFBR-703SDDZ-IN1
LR Modules		
Intel	DUAL RATE 1G/10G SFP+ LR (bailed)	FTLX1471D3BCV-IT
Intel	DUAL RATE 1G/10G SFP+ LR (bailed)	AFCT-701SDZ-IN2
Intel	DUAL RATE 1G/10G SFP+ LR (bailed)	AFCT-701SDDZ-IN1

The following is a list of 3rd party SFP+ modules that have received some testing. Not all modules are applicable to all devices.

Supplier	Type	Part Numbers
Finisar	SFP+ SR bailed, 10g single rate	FTLX8571D3BCL
Avago	SFP+ SR bailed, 10g single rate	AFBR-700SDZ
Finisar	SFP+ LR bailed, 10g single rate	FTLX1471D3BCL
Finisar	DUAL RATE 1G/10G SFP+ SR (No Bail)	FTLX8571D3QCV-IT
Avago	DUAL RATE 1G/10G SFP+ SR (No Bail)	AFBR-703SDZ-IN1
Finisar	DUAL RATE 1G/10G SFP+ LR (No Bail)	FTLX1471D3QCV-IT
Avago	DUAL RATE 1G/10G SFP+ LR (No Bail)	AFCT-701SDZ-IN1
Finisar	1000BASE-T SFP	FCLF8522P2BTL
Avago	1000BASE-T	ABCU-5710RZ
HP	1000BASE-SX SFP	453153-001

82599-based adapters support all passive and active limiting direct attach cables that comply with SFF-8431 v4.1 and SFF-8472 v10.4 specifications.

Laser turns off for SFP+ when ifconfig ethX down

"ifconfig ethX down" turns off the laser for 82599-based SFP+ fiber adapters. "ifconfig ethX up" turns on the laser. Alternatively, you can use "ip link set [down/up] dev ethX" to turn the laser off and on.

82599-based QSFP+ Adapters

NOTES: - If your 82599-based Intel(R) Network Adapter came with Intel optics, it only supports Intel optics. - 82599-based QSFP+ adapters only support 4x10 Gbps connections. 1x40 Gbps connections are not supported. QSFP+ link partners must be configured for 4x10 Gbps. - 82599-based QSFP+ adapters do not support automatic link speed detection. The link speed must be configured to either 10 Gbps or 1 Gbps to match the link partners speed capabilities. Incorrect speed configurations will result in failure to link. - Intel(R) Ethernet Converged Network Adapter X520-Q1 only supports the optics and direct attach cables listed below.

Supplier	Type	Part Numbers
Intel	DUAL RATE 1G/10G QSFP+ SRL (bailed)	E10GQSFPSR

82599-based QSFP+ adapters support all passive and active limiting QSFP+ direct attach cables that comply with SFF-8436 v4.1 specifications.

82598-BASED ADAPTERS

NOTES: - Intel(r) Ethernet Network Adapters that support removable optical modules only support their original module type (for example, the Intel(R) 10 Gigabit SR Dual Port Express Module only supports SR optical modules). If you plug in a different type of module, the driver will not load. - Hot Swapping/hot plugging optical modules is not supported. - Only single speed, 10 gigabit modules are supported. - LAN on Motherboard (LOMs) may support DA, SR, or LR modules. Other module types are not supported. Please see your system documentation for details.

The following is a list of SFP+ modules and direct attach cables that have received some testing. Not all modules are applicable to all devices.

Supplier	Type	Part Numbers
Finisar	SFP+ SR bailed, 10g single rate	FTLX8571D3BCL
Avago	SFP+ SR bailed, 10g single rate	AFBR-700SDZ
Finisar	SFP+ LR bailed, 10g single rate	FTLX1471D3BCL

82598-based adapters support all passive direct attach cables that comply with SFF-8431 v4.1 and SFF-8472 v10.4 specifications. Active direct attach cables are not supported.

Third party optic modules and cables referred to above are listed only for the purpose of highlighting third party specifications and potential compatibility, and are not recommendations or endorsements or sponsorship of any third party's product by Intel. Intel is not endorsing or promoting products made by any third party and the third party reference is provided only to share information regarding certain optic modules and cables with the above specifications. There may be other manufacturers or suppliers, producing or supplying optic modules and cables with similar or matching descriptions. Customers must use their own discretion and diligence to purchase optic modules and cables from any third party of their choice. Customers are solely responsible for assessing the suitability of the product and/or devices and for the selection of the vendor for purchasing any product. THE OPTIC MODULES AND CABLES REFERRED TO ABOVE ARE NOT WARRANTED OR SUPPORTED BY INTEL. INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF SUCH THIRD PARTY PRODUCTS OR SELECTION OF VENDOR BY CUSTOMERS.

Command Line Parameters

max_vfs

Valid Range

1-63

This parameter adds support for SR-IOV. It causes the driver to spawn up to max_vfs worth of virtual functions. If the value is greater than 0 it will also force the VMDq parameter to be 1 or more.

NOTE: This parameter is only used on kernel 3.7.x and below. On kernel 3.8.x and above, use sysfs to enable VFs. Also, for Red Hat distributions, this parameter is only used on version 6.6 and older. For version 6.7 and newer, use sysfs. For example:

```
#echo $num_vf_enabled > /sys/class/net/$dev/device/sriov_numvfs // enable VFs
#echo 0 > /sys/class/net/$dev/device/sriov_numvfs // disable VFs
```

The parameters for the driver are referenced by position. Thus, if you have a dual port adapter, or more than one adapter in your system, and want N virtual functions per port, you must specify a number for each port with each parameter separated by a comma. For example:

```
modprobe ixgbe max_vfs=4
```

This will spawn 4 VFs on the first port.

```
modprobe ixgbe max_vfs=2,4
```

This will spawn 2 VFs on the first port and 4 VFs on the second port.

NOTE: Caution must be used in loading the driver with these parameters. Depending on your system configuration, number of slots, etc., it is impossible to predict in all cases where the positions would be on the command line.

NOTE: Neither the device nor the driver control how VFs are mapped into config space. Bus layout will vary by operating system. On operating systems that support it, you can check sysfs to find the mapping.

NOTE: When either SR-IOV mode or VMDq mode is enabled, hardware VLAN filtering and VLAN tag stripping/insertion will remain enabled. Please remove the old VLAN filter before the new VLAN filter is added. For example,

```
ip link set eth0 vf 0 vlan 100 // set VLAN 100 for VF 0
ip link set eth0 vf 0 vlan 0 // Delete VLAN 100
ip link set eth0 vf 0 vlan 200 // set a new VLAN 200 for VF 0
```

With kernel 3.6, the driver supports the simultaneous usage of max_vfs and DCB features, subject to the constraints described below. Prior to kernel 3.6, the driver did not support the simultaneous operation of max_vfs greater than 0 and the DCB features (multiple traffic classes utilizing Priority Flow Control and Extended Transmission Selection).

When DCB is enabled, network traffic is transmitted and received through multiple traffic classes (packet buffers in the NIC). The traffic is associated with a specific class based on priority, which has a value of 0 through 7 used in the VLAN tag. When SR-IOV is not enabled, each traffic class is associated with a set of receive/transmit descriptor queue pairs. The number of queue pairs for a given traffic class depends on the hardware configuration. When SR-IOV is enabled, the descriptor queue pairs are grouped into pools. The Physical Function (PF) and each Virtual Function (VF) is allocated a pool of receive/transmit descriptor queue pairs. When multiple traffic classes are configured (for example, DCB is enabled), each pool contains a queue pair from each traffic class. When a single traffic class is configured in the hardware, the pools contain multiple queue pairs from the single traffic class.

The number of VFs that can be allocated depends on the number of traffic classes that can be enabled. The configurable number of traffic classes for each enabled VF is as follows: 0 - 15 VFs = Up to 8 traffic classes, depending on device support 16 - 31 VFs = Up to 4 traffic classes 32 - 63 VFs = 1 traffic class

When VFs are configured, the PF is allocated one pool as well. The PF supports the DCB features with the constraint that each traffic class will only use a single queue pair. When zero VFs are configured, the PF can support multiple queue pairs per traffic class.

allow_unsupported_sfp

Valid Range

0,1

Default Value

0 (disabled)

This parameter allows unsupported and untested SFP+ modules on 82599-based adapters, as long as the type of module is known to the driver.

debug

Valid Range

0-16 (0=none,...,16=all)

Default Value

0

This parameter adjusts the level of debug messages displayed in the system logs.

Additional Features and Configurations

Flow Control

Ethernet Flow Control (IEEE 802.3x) can be configured with ethtool to enable receiving and transmitting pause frames for ixgbe. When transmit is enabled, pause frames are generated when the receive packet buffer crosses a predefined threshold. When receive is enabled, the transmit unit will halt for the time delay specified when a pause frame is received.

NOTE: You must have a flow control capable link partner.

Flow Control is enabled by default.

Use ethtool to change the flow control settings. To enable or disable Rx or Tx Flow Control:

```
ethtool -A eth? rx <on|off> tx <on|off>
```

Note: This command only enables or disables Flow Control if auto-negotiation is disabled. If auto-negotiation is enabled, this command changes the parameters used for auto-negotiation with the link partner.

To enable or disable auto-negotiation:

```
ethtool -s eth? autoneg <on|off>
```

Note: Flow Control auto-negotiation is part of link auto-negotiation. Depending on your device, you may not be able to change the auto-negotiation setting.

NOTE: For 82598 backplane cards entering 1 gigabit mode, flow control default behavior is changed to off. Flow control in 1 gigabit mode on these devices can lead to transmit hangs.

Intel(R) Ethernet Flow Director

The Intel Ethernet Flow Director performs the following tasks:

- Directs receive packets according to their flows to different queues.
- Enables tight control on routing a flow in the platform.
- Matches flows and CPU cores for flow affinity.
- Supports multiple parameters for flexible flow classification and load balancing (in SFP mode only).

NOTE: Intel Ethernet Flow Director masking works in the opposite manner from subnet masking. In the following command:

```
#ethtool -N eth11 flow-type ip4 src-ip 172.4.1.2 m 255.0.0.0 dst-ip \
172.21.1.1 m 255.128.0.0 action 31
```

The src-ip value that is written to the filter will be 0.4.1.2, not 172.0.0.0 as might be expected. Similarly, the dst-ip value written to the filter will be 0.21.1.1, not 172.0.0.0.

To enable or disable the Intel Ethernet Flow Director:

```
# ethtool -K ethX ntuple <on|off>
```

When disabling ntuple filters, all the user programmed filters are flushed from the driver cache and hardware. All needed filters must be re-added when ntuple is re-enabled.

To add a filter that directs packet to queue 2, use -U or -N switch:

```
# ethtool -N ethX flow-type tcp4 src-ip 192.168.10.1 dst-ip \
192.168.10.2 src-port 2000 dst-port 2001 action 2 [loc 1]
```

To see the list of filters currently present:

```
# ethtool <-u|-n> ethX
```

Sideband Perfect Filters

Sideband Perfect Filters are used to direct traffic that matches specified characteristics. They are enabled through ethtool's ntuple interface. To add a new filter use the following command:

```
ethtool -U <device> flow-type <type> src-ip <ip> dst-ip <ip> src-port <port> \
dst-port <port> action <queue>
```

Where:

- <device> - the ethernet device to program <type> - can be ip4, tcp4, udp4, or sctp4
- <ip> - the IP address to match on
- <port> - the port number to match on
- <queue> - the queue to direct traffic towards (-1 discards the matched traffic)

Use the following command to delete a filter:

```
ethtool -U <device> delete <N>
```

Where <N> is the filter id displayed when printing all the active filters, and may also have been specified using "loc <N>" when adding the filter.

The following example matches TCP traffic sent from 192.168.0.1, port 5300, directed to 192.168.0.5, port 80, and sends it to queue 7:

```
ethtool -U enp130s0 flow-type tcp4 src-ip 192.168.0.1 dst-ip 192.168.0.5 \
src-port 5300 dst-port 80 action 7
```

For each flow-type, the programmed filters must all have the same matching input set. For example, issuing the following two commands is acceptable:

```
ethtool -U enp130s0 flow-type ip4 src-ip 192.168.0.1 src-port 5300 action 7
ethtool -U enp130s0 flow-type ip4 src-ip 192.168.0.5 src-port 55 action 10
```

Issuing the next two commands, however, is not acceptable, since the first specifies src-ip and the second specifies dst-ip:

```
ethtool -U enp130s0 flow-type ip4 src-ip 192.168.0.1 src-port 5300 action 7
ethtool -U enp130s0 flow-type ip4 dst-ip 192.168.0.5 src-port 55 action 10
```

The second command will fail with an error. You may program multiple filters with the same fields, using different values, but, on one device, you may not program two TCP4 filters with different matching fields.

Matching on a sub-portion of a field is not supported by the ixgbe driver, thus partial mask fields are not supported.

To create filters that direct traffic to a specific Virtual Function, use the "user-def" parameter. Specify the user-def as a 64 bit value, where the lower 32 bits represents the queue number, while the next 8 bits represent which VF. Note that 0 is the PF, so the VF identifier is offset by 1. For example:

```
... user-def 0x8000000002 ...
```

specifies to direct traffic to Virtual Function 7 (8 minus 1) into queue 2 of that VF.

Note that these filters will not break internal routing rules, and will not route traffic that otherwise would not have been sent to the specified Virtual Function.

Jumbo Frames

Jumbo Frames support is enabled by changing the Maximum Transmission Unit (MTU) to a value larger than the default value of 1500.

Use the ifconfig command to increase the MTU size. For example, enter the following where <x> is the interface number:

```
ifconfig eth<x> mtu 9000 up
```

Alternatively, you can use the ip command as follows:

```
ip link set mtu 9000 dev eth<x>
ip link set up dev eth<x>
```

This setting is not saved across reboots. The setting change can be made permanent by adding 'MTU=9000' to the file:

```
/etc/sysconfig/network-scripts/ifcfg-eth<x> // for RHEL
/etc/sysconfig/network/<config_file> // for SLES
```

NOTE: The maximum MTU setting for Jumbo Frames is 9710. This value coincides with the maximum Jumbo Frames size of 9728 bytes.

NOTE: This driver will attempt to use multiple page sized buffers to receive each jumbo packet. This should help to avoid buffer starvation issues when allocating receive packets.

NOTE: For 82599-based network connections, if you are enabling jumbo frames in a virtual function (VF), jumbo frames must first be enabled in the physical function (PF). The VF MTU setting cannot be larger than the PF MTU.

NBASE-T Support

The ixgbe driver supports NBASE-T on some devices. However, the advertisement of NBASE-T speeds is suppressed by default, to accommodate broken network switches which cannot cope with advertised NBASE-T speeds. Use the ethtool command to enable advertising NBASE-T speeds on devices which support it:

```
ethtool -s eth? advertise 0x1800000001028
```

On Linux systems with INTERFACES(5), this can be specified as a pre-up command in /etc/network/interfaces so that the interface is always brought up with NBASE-T support, e.g.:

```
iface eth? inet dhcp
    pre-up ethtool -s eth? advertise 0x1800000001028 || true
```

Generic Receive Offload, aka GRO

The driver supports the in-kernel software implementation of GRO. GRO has shown that by coalescing Rx traffic into larger chunks of data, CPU utilization can be significantly reduced when under large Rx load. GRO is an evolution of the previously-used LRO interface. GRO is able to coalesce other protocols besides TCP. It's also safe to use with configurations that are problematic for LRO, namely bridging and iSCSI.

Data Center Bridging (DCB)

NOTE: The kernel assumes that TC0 is available, and will disable Priority Flow Control (PFC) on the device if TC0 is not available. To fix this, ensure TC0 is enabled when setting up DCB on your switch.

DCB is a configuration Quality of Service implementation in hardware. It uses the VLAN priority tag (802.1p) to filter traffic. That means that there are 8 different priorities that traffic can be filtered into. It also enables priority flow control (802.1Qbb) which can limit or eliminate the number of dropped packets during network stress. Bandwidth can be allocated to each of these priorities, which is enforced at the hardware level (802.1Qaz).

Adapter firmware implements LLDP and DCBX protocol agents as per 802.1AB and 802.1Qaz respectively. The firmware based DCBX agent runs in willing mode only and can accept settings from a DCBX capable peer. Software configuration of DCBX parameters via dcbtool/lldptool are not supported.

The ixgbe driver implements the DCB netlink interface layer to allow user-space to communicate with the driver and query DCB configuration for the port.

ethtool

The driver utilizes the ethtool interface for driver configuration and diagnostics, as well as displaying statistical information. The latest ethtool version is required for this functionality. Download it at: <https://www.kernel.org/pub/software/network/ethtool/>

FCoE

The ixgbe driver supports Fiber Channel over Ethernet (FCoE) and Data Center Bridging (DCB). This code has no default effect on the regular driver operation. Configuring DCB and FCoE is outside the scope of this README. Refer to <http://www.open-fcoe.org/> for FCoE project information and contact ixgbe-eedc@lists.sourceforge.net for DCB information.

MAC and VLAN anti-spoofing feature

When a malicious driver attempts to send a spoofed packet, it is dropped by the hardware and not transmitted.

An interrupt is sent to the PF driver notifying it of the spoof attempt. When a spoofed packet is detected, the PF driver will send the following message to the system log (displayed by the "dmesg" command):

```
ixgbe ethX: ixgbe_spoof_check: n spoofed packets detected
```

where "x" is the PF interface number; and "n" is number of spoofed packets. NOTE: This feature can be disabled for a specific Virtual Function (VF):

```
ip link set <pf dev> vf <vf id> spoofchk {off|on}
```

IPsec Offload

The ixgbe driver supports IPsec Hardware Offload. When creating Security Associations with "ip xfrm ..." the 'offload' tag option can be used to register the IPsec SA with the driver in order to get higher throughput in the secure communications.

The offload is also supported for ixgbe's VFs, but the VF must be set as 'trusted' and the support must be enabled with:

```
ethtool --set-priv-flags eth<x> vf-ipsec on  
ip link set eth<x> vf <y> trust on
```

Known Issues/Troubleshooting

Enabling SR-IOV in a 64-bit Microsoft Windows Server 2012/R2 guest OS

Linux KVM Hypervisor/VMM supports direct assignment of a PCIe device to a VM. This includes traditional PCIe devices, as well as SR-IOV-capable devices based on the Intel Ethernet Controller XL710.

Support

For general information, go to the Intel support website at: <https://www.intel.com/support/>

If an issue is identified with the released source code on a supported kernel with a supported adapter, email the specific information related to the issue to intel-wired-lan@lists.osuosl.org.

6.5.27 Linux Base Virtual Function Driver for Intel(R) 10G Ethernet

Intel 10 Gigabit Virtual Function Linux driver. Copyright(c) 1999-2018 Intel Corporation.

Contents

- Identifying Your Adapter
- Known Issues
- Support

This driver supports 82599, X540, X550, and X552-based virtual function devices that can only be activated on kernels that support SR-IOV.

For questions related to hardware requirements, refer to the documentation supplied with your Intel adapter. All hardware requirements listed apply to use with Linux.

Identifying Your Adapter

The driver is compatible with devices based on the following:

- Intel(R) Ethernet Controller 82598
- Intel(R) Ethernet Controller 82599
- Intel(R) Ethernet Controller X520
- Intel(R) Ethernet Controller X540
- Intel(R) Ethernet Controller x550
- Intel(R) Ethernet Controller X552
- Intel(R) Ethernet Controller X553

For information on how to identify your adapter, and for the latest Intel network drivers, refer to the Intel Support website: <https://www.intel.com/support>

Known Issues/Troubleshooting

SR-IOV requires the correct platform and OS support.

The guest OS loading this driver must support MSI-X interrupts.

This driver is only supported as a loadable module at this time. Intel is not supplying patches against the kernel source to allow for static linking of the drivers.

VLANs: There is a limit of a total of 64 shared VLANs to 1 or more VFs.

Support

For general information, go to the Intel support website at: <https://www.intel.com/support>

If an issue is identified with the released source code on a supported kernel with a supported adapter, email the specific information related to the issue to intel-wired-lan@lists.osuosl.org.

6.5.28 Linux Base Driver for the Intel(R) Ethernet Controller 700 Series

Intel 40 Gigabit Linux driver. Copyright(c) 1999-2018 Intel Corporation.

Contents

- Overview
- Identifying Your Adapter
- Intel(R) Ethernet Flow Director
- Additional Configurations
- Known Issues
- Support

Driver information can be obtained using ethtool, lspci, and ifconfig. Instructions on updating ethtool can be found in the section Additional Configurations later in this document.

For questions related to hardware requirements, refer to the documentation supplied with your Intel adapter. All hardware requirements listed apply to use with Linux.

Identifying Your Adapter

The driver is compatible with devices based on the following:

- Intel(R) Ethernet Controller X710
- Intel(R) Ethernet Controller XL710
- Intel(R) Ethernet Network Connection X722
- Intel(R) Ethernet Controller XXV710

For the best performance, make sure the latest NVM/FW is installed on your device.

For information on how to identify your adapter, and for the latest NVM/FW images and Intel network drivers, refer to the Intel Support website: <https://www.intel.com/support>

SFP+ and QSFP+ Devices

For information about supported media, refer to this document: <https://www.intel.com/content/dam/www/public/us/en/documents/release-notes/xl710-ethernet-controller-feature-matrix.pdf>

NOTE: Some adapters based on the Intel(R) Ethernet Controller 700 Series only support Intel Ethernet Optics modules. On these adapters, other modules are not supported and will not function. In all cases Intel recommends using Intel Ethernet Optics; other modules may function but are not validated by Intel. Contact Intel for supported media types.

NOTE: For connections based on Intel(R) Ethernet Controller 700 Series, support is dependent on your system board. Please see your vendor for details.

NOTE: In systems that do not have adequate airflow to cool the adapter and optical modules, you must use high temperature optical modules.

Virtual Functions (VFs)

Use sysfs to enable VFs. For example:

```
#echo $num_vf_enabled > /sys/class/net/$dev/device/sriov_numvfs #enable VFs  
#echo 0 > /sys/class/net/$dev/device/sriov_numvfs #disable VFs
```

For example, the following instructions will configure PF eth0 and the first VF on VLAN 10:

```
$ ip link set dev eth0 vf 0 vlan 10
```

VLAN Tag Packet Steering

Allows you to send all packets with a specific VLAN tag to a particular SR-IOV virtual function (VF). Further, this feature allows you to designate a particular VF as trusted, and allows that trusted VF to request selective promiscuous mode on the Physical Function (PF).

To set a VF as trusted or untrusted, enter the following command in the Hypervisor:

```
# ip link set dev eth0 vf 1 trust [on|off]
```

Once the VF is designated as trusted, use the following commands in the VM to set the VF to promiscuous mode.

For promiscuous all:

```
#ip link set eth2 promisc on
```

Where eth2 is a VF interface in the VM

For promiscuous Multicast:

```
#ip link set eth2 allmulticast on
```

Where eth2 is a VF interface in the VM

NOTE: By default, the ethtool priv-flag vf-true-promisc-support is set to "off", meaning that promiscuous mode for the VF will be limited. To set the promiscuous mode for the VF to true promiscuous and allow the VF to see all ingress traffic, use the following command:

```
#ethtool -set-priv-flags p261p1 vf-true-promisc-support on
```

The vf-true-promisc-support priv-flag does not enable promiscuous mode; rather, it designates which type of promiscuous mode (limited or true) you will get when you enable promiscuous mode using the ip link commands above. Note that this is a global setting that affects the entire device. However, the vf-true-promisc-support priv-flag is only exposed to the first PF of the device. The PF remains in limited promiscuous mode (unless it is in MFP mode) regardless of the vf-true-promisc-support setting.

Now add a VLAN interface on the VF interface:

```
#ip link add link eth2 name eth2.100 type vlan id 100
```

Note that the order in which you set the VF to promiscuous mode and add the VLAN interface does not matter (you can do either first). The end result in this example is that the VF will get all traffic that is tagged with VLAN 100.

Intel(R) Ethernet Flow Director

The Intel Ethernet Flow Director performs the following tasks:

- Directs receive packets according to their flows to different queues.
- Enables tight control on routing a flow in the platform.
- Matches flows and CPU cores for flow affinity.
- Supports multiple parameters for flexible flow classification and load balancing (in SFP mode only).

NOTE: The Linux i40e driver supports the following flow types: IPv4, TCPv4, and UDPv4. For a given flow type, it supports valid combinations of IP addresses (source or destination) and UDP/TCP ports (source and destination). For example, you can supply only a source IP address, a source IP address and a destination port, or any combination of one or more of these four parameters.

NOTE: The Linux i40e driver allows you to filter traffic based on a user-defined flexible two-byte pattern and offset by using the ethtool user-def and mask fields. Only L3 and L4 flow types are supported for user-defined flexible filters. For a given flow type, you must clear all Intel Ethernet Flow Director filters before changing the input set (for that flow type).

To enable or disable the Intel Ethernet Flow Director:

```
# ethtool -K ethX ntuple <on|off>
```

When disabling ntuple filters, all the user programmed filters are flushed from the driver cache and hardware. All needed filters must be re-added when ntuple is re-enabled.

To add a filter that directs packet to queue 2, use -U or -N switch:

```
# ethtool -N ethX flow-type tcp4 src-ip 192.168.10.1 dst-ip \
192.168.10.2 src-port 2000 dst-port 2001 action 2 [loc 1]
```

To set a filter using only the source and destination IP address:

```
# ethtool -N ethX flow-type tcp4 src-ip 192.168.10.1 dst-ip \
192.168.10.2 action 2 [loc 1]
```

To see the list of filters currently present:

```
# ethtool <-u|-n> ethX
```

Application Targeted Routing (ATR) Perfect Filters

ATR is enabled by default when the kernel is in multiple transmit queue mode. An ATR Intel Ethernet Flow Director filter rule is added when a TCP-IP flow starts and is deleted when the flow ends. When a TCP-IP Intel Ethernet Flow Director rule is added from ethtool (Sideband filter), ATR is turned off by the driver. To re-enable ATR, the sideband can be disabled with the ethtool -K option. For example:

```
ethtool -K [adapter] ntuple [off|on]
```

If sideband is re-enabled after ATR is re-enabled, ATR remains enabled until a TCP-IP flow is added. When all TCP-IP sideband rules are deleted, ATR is automatically re-enabled.

Packets that match the ATR rules are counted in fdir_atr_match stats in ethtool, which also can be used to verify whether ATR rules still exist.

Sideband Perfect Filters

Sideband Perfect Filters are used to direct traffic that matches specified characteristics. They are enabled through ethtool's ntuple interface. To add a new filter use the following command:

```
ethtool -U <device> flow-type <type> src-ip <ip> dst-ip <ip> src-port <port> \
dst-port <port> action <queue>
```

Where:

- <device> - the ethernet device to program
- <type> - can be ip4, tcp4, udp4, or sctp4
- <ip> - the ip address to match on
- <port> - the port number to match on
- <queue> - the queue to direct traffic towards (-1 discards matching traffic)

Use the following command to display all of the active filters:

```
ethtool -u <device>
```

Use the following command to delete a filter:

```
ethtool -U <device> delete <N>
```

Where <N> is the filter id displayed when printing all the active filters, and may also have been specified using "loc <N>" when adding the filter.

The following example matches TCP traffic sent from 192.168.0.1, port 5300, directed to 192.168.0.5, port 80, and sends it to queue 7:

```
ethtool -U enp130s0 flow-type tcp4 src-ip 192.168.0.1 dst-ip 192.168.0.5 \
src-port 5300 dst-port 80 action 7
```

For each flow-type, the programmed filters must all have the same matching input set. For example, issuing the following two commands is acceptable:

```
ethtool -U enp130s0 flow-type ip4 src-ip 192.168.0.1 src-port 5300 action 7
ethtool -U enp130s0 flow-type ip4 src-ip 192.168.0.5 src-port 55 action 10
```

Issuing the next two commands, however, is not acceptable, since the first specifies src-ip and the second specifies dst-ip:

```
ethtool -U enp130s0 flow-type ip4 src-ip 192.168.0.1 src-port 5300 action 7
ethtool -U enp130s0 flow-type ip4 dst-ip 192.168.0.5 src-port 55 action 10
```

The second command will fail with an error. You may program multiple filters with the same fields, using different values, but, on one device, you may not program two tcp4 filters with different matching fields.

Matching on a sub-portion of a field is not supported by the i40e driver, thus partial mask fields are not supported.

The driver also supports matching user-defined data within the packet payload. This flexible data is specified using the "user-def" field of the ethtool command in the following way:

31 28 24 20 16	15 12 8 4 0
offset into packet payload	2 bytes of flexible data

For example,

```
... user-def 0xFFFF ...
```

tells the filter to look 4 bytes into the payload and match that value against 0xFFFF. The offset is based on the beginning of the payload, and not the beginning of the packet. Thus

```
flow-type tcp4 ... user-def 0x8BEAF ...
```

would match TCP/IPv4 packets which have the value 0xBEAF 8 bytes into the TCP/IPv4 payload.

Note that ICMP headers are parsed as 4 bytes of header and 4 bytes of payload. Thus to match the first byte of the payload, you must actually add 4 bytes to the offset. Also note that ip4 filters match both ICMP frames as well as raw (unknown) ip4 frames, where the payload will be the L3 payload of the IP4 frame.

The maximum offset is 64. The hardware will only read up to 64 bytes of data from the payload. The offset must be even because the flexible data is 2 bytes long and must be aligned to byte 0 of the packet payload.

The user-defined flexible offset is also considered part of the input set and cannot be programmed separately for multiple filters of the same type. However, the flexible data is not part of the input set and multiple filters may use the same offset but match against different data.

To create filters that direct traffic to a specific Virtual Function, use the "action" parameter. Specify the action as a 64 bit value, where the lower 32 bits represents the queue number, while the next 8 bits represent which VF. Note that 0 is the PF, so the VF identifier is offset by 1. For example:

```
... action 0x800000002 ...
```

specifies to direct traffic to Virtual Function 7 (8 minus 1) into queue 2 of that VF.

Note that these filters will not break internal routing rules, and will not route traffic that otherwise would not have been sent to the specified Virtual Function.

Setting the link-down-on-close Private Flag

When the link-down-on-close private flag is set to "on", the port's link will go down when the interface is brought down using the ifconfig ethX down command.

Use ethtool to view and set link-down-on-close, as follows:

```
ethtool --show-priv-flags ethX
ethtool --set-priv-flags ethX link-down-on-close [on|off]
```

Viewing Link Messages

Link messages will not be displayed to the console if the distribution is restricting system messages. In order to see network driver link messages on your console, set dmesg to eight by entering the following:

```
dmesg -n 8
```

NOTE: This setting is not saved across reboots.

Jumbo Frames

Jumbo Frames support is enabled by changing the Maximum Transmission Unit (MTU) to a value larger than the default value of 1500.

Use the ifconfig command to increase the MTU size. For example, enter the following where <x> is the interface number:

```
ifconfig eth<x> mtu 9000 up
```

Alternatively, you can use the ip command as follows:

```
ip link set mtu 9000 dev eth<x>
ip link set up dev eth<x>
```

This setting is not saved across reboots. The setting change can be made permanent by adding 'MTU=9000' to the file:

```
/etc/sysconfig/network-scripts/ifcfg-eth<x> // for RHEL
/etc/sysconfig/network/<config_file> // for SLES
```

NOTE: The maximum MTU setting for Jumbo Frames is 9702. This value coincides with the maximum Jumbo Frames size of 9728 bytes.

NOTE: This driver will attempt to use multiple page sized buffers to receive each jumbo packet. This should help to avoid buffer starvation issues when allocating receive packets.

ethtool

The driver utilizes the ethtool interface for driver configuration and diagnostics, as well as displaying statistical information. The latest ethtool version is required for this functionality. Download it at: <https://www.kernel.org/pub/software/network/ethtool/>

Supported ethtool Commands and Options for Filtering

-n --show-nfc

Retrieves the receive network flow classification configurations.

rx-flow-hash tcp4|udp4|ah4|esp4|sctp4|tcp6|udp6|ah6|esp6|sctp6

Retrieves the hash options for the specified network traffic type.

-N --config-nfc

Configures the receive network flow classification.

rx-flow-hash tcp4|udp4|ah4|esp4|sctp4|tcp6|udp6|ah6|esp6|sctp6 m|v|t|s|d|f|n|r...

Configures the hash options for the specified network traffic type.

udp4 UDP over IPv4 udp6 UDP over IPv6

f Hash on bytes 0 and 1 of the Layer 4 header of the Rx packet. n Hash on bytes 2 and 3 of the Layer 4 header of the Rx packet.

Speed and Duplex Configuration

In addressing speed and duplex configuration issues, you need to distinguish between copper-based adapters and fiber-based adapters.

In the default mode, an Intel(R) Ethernet Network Adapter using copper connections will attempt to auto-negotiate with its link partner to determine the best setting. If the adapter cannot establish link with the link partner using auto-negotiation, you may need to manually configure the adapter and link partner to identical settings to establish link and pass packets. This should only be needed when attempting to link with an older switch that does not support auto-negotiation or one that has been forced to a specific speed or duplex mode. Your link partner must match the setting you choose. 1 Gbps speeds and higher cannot be forced. Use the autonegotiation advertising setting to manually set devices for 1 Gbps and higher.

NOTE: You cannot set the speed for devices based on the Intel(R) Ethernet Network Adapter XXV710 based devices.

Speed, duplex, and autonegotiation advertising are configured through the ethtool utility.

Caution: Only experienced network administrators should force speed and duplex or change autonegotiation advertising manually. The settings at the switch must always match the adapter settings. Adapter performance may suffer or your adapter may not operate if you configure the adapter differently from your switch.

An Intel(R) Ethernet Network Adapter using fiber-based connections, however, will not attempt to auto-negotiate with its link partner since those adapters operate only in full duplex and only at their native speed.

NAPI

NAPI (Rx polling mode) is supported in the i40e driver.

See [Documentation/networking/napi.rst](#) for more information.

Flow Control

Ethernet Flow Control (IEEE 802.3x) can be configured with ethtool to enable receiving and transmitting pause frames for i40e. When transmit is enabled, pause frames are generated when the receive packet buffer crosses a predefined threshold. When receive is enabled, the transmit unit will halt for the time delay specified when a pause frame is received.

NOTE: You must have a flow control capable link partner.

Flow Control is on by default.

Use ethtool to change the flow control settings.

To enable or disable Rx or Tx Flow Control:

```
ethtool -A eth? rx <on|off> tx <on|off>
```

Note: This command only enables or disables Flow Control if auto-negotiation is disabled. If auto-negotiation is enabled, this command changes the parameters used for auto-negotiation with the link partner.

To enable or disable auto-negotiation:

```
ethtool -s eth? autoneg <on|off>
```

Note: Flow Control auto-negotiation is part of link auto-negotiation. Depending on your device, you may not be able to change the auto-negotiation setting.

RSS Hash Flow

Allows you to set the hash bytes per flow type and any combination of one or more options for Receive Side Scaling (RSS) hash byte configuration.

```
# ethtool -N <dev> rx-flow-hash <type> <option>
```

Where <type> is:

tcp4 signifying TCP over IPv4
udp4 signifying UDP over IPv4
tcp6 signifying TCP over IPv6
udp6 signifying UDP over IPv6

And <option> is one or more of:

s Hash on the IP source address of the Rx packet.
d Hash on the IP destination address of the Rx packet.
f Hash on bytes 0 and 1 of the Layer 4 header of the Rx packet.
n Hash on bytes 2 and 3 of the Layer 4 header of the Rx packet.

MAC and VLAN anti-spoofing feature

When a malicious driver attempts to send a spoofed packet, it is dropped by the hardware and not transmitted. NOTE: This feature can be disabled for a specific Virtual Function (VF):

```
ip link set <pf dev> vf <vf id> spoofchk {off|on}
```

IEEE 1588 Precision Time Protocol (PTP) Hardware Clock (PHC)

Precision Time Protocol (PTP) is used to synchronize clocks in a computer network. PTP support varies among Intel devices that support this driver. Use "ethtool -T <netdev name>" to get a definitive list of PTP capabilities supported by the device.

IEEE 802.1ad (QinQ) Support

The IEEE 802.1ad standard, informally known as QinQ, allows for multiple VLAN IDs within a single Ethernet frame. VLAN IDs are sometimes referred to as "tags," and multiple VLAN IDs are thus referred to as a "tag stack." Tag stacks allow L2 tunneling and the ability to segregate traffic within a particular VLAN ID, among other uses.

The following are examples of how to configure 802.1ad (QinQ):

```
ip link add link eth0 eth0.24 type vlan proto 802.1ad id 24
ip link add link eth0.24 eth0.24.371 type vlan proto 802.1q id 371
```

Where "24" and "371" are example VLAN IDs.

NOTES:

Receive checksum offloads, cloud filters, and VLAN acceleration are not supported for 802.1ad (QinQ) packets.

VXLAN and GENEVE Overlay HW Offloading

Virtual Extensible LAN (VXLAN) allows you to extend an L2 network over an L3 network, which may be useful in a virtualized or cloud environment. Some Intel(R) Ethernet Network devices perform VXLAN processing, offloading it from the operating system. This reduces CPU utilization.

VXLAN offloading is controlled by the Tx and Rx checksum offload options provided by ethtool. That is, if Tx checksum offload is enabled, and the adapter has the capability, VXLAN offloading is also enabled.

Support for VXLAN and GENEVE HW offloading is dependent on kernel support of the HW offloading features.

Multiple Functions per Port

Some adapters based on the Intel Ethernet Controller X710/XL710 support multiple functions on a single physical port. Configure these functions through the System Setup/BIOS.

Minimum TX Bandwidth is the guaranteed minimum data transmission bandwidth, as a percentage of the full physical port link speed, that the partition will receive. The bandwidth the partition is awarded will never fall below the level you specify.

The range for the minimum bandwidth values is: 1 to ((100 minus # of partitions on the physical port) plus 1) For example, if a physical port has 4 partitions, the range would be: 1 to ((100 - 4) + 1 = 97)

The Maximum Bandwidth percentage represents the maximum transmit bandwidth allocated to the partition as a percentage of the full physical port link speed. The accepted range of values is 1-100. The value is used as a limiter, should you chose that any one particular function not be able to consume 100% of a port's bandwidth (should it be available). The sum of all the values for Maximum Bandwidth is not restricted, because no more than 100% of a port's bandwidth can ever be used.

NOTE: X710/XXV710 devices fail to enable Max VFs (64) when Multiple Functions per Port (MFP) and SR-IOV are enabled. An error from i40e is logged that says "add vsi failed for VF N, aq_err 16". To workaround the issue, enable less than 64 virtual functions (VFs).

Data Center Bridging (DCB)

DCB is a configuration Quality of Service implementation in hardware. It uses the VLAN priority tag (802.1p) to filter traffic. That means that there are 8 different priorities that traffic can be filtered into. It also enables priority flow control (802.1Qbb) which can limit or eliminate the number of dropped packets during network stress. Bandwidth can be allocated to each of these priorities, which is enforced at the hardware level (802.1Qaz).

Adapter firmware implements LLDP and DCBX protocol agents as per 802.1AB and 802.1Qaz respectively. The firmware based DCBX agent runs in willing mode only and can accept settings from a DCBX capable peer. Software configuration of DCBX parameters via dcbtool/lldptool are not supported.

NOTE: Firmware LLDP can be disabled by setting the private flag disable-fw-lldp.

The i40e driver implements the DCB netlink interface layer to allow user-space to communicate with the driver and query DCB configuration for the port.

NOTE: The kernel assumes that TC0 is available, and will disable Priority Flow Control (PFC) on the device if TC0 is not available. To fix this, ensure TC0 is enabled when setting up DCB on your switch.

Interrupt Rate Limiting

Valid Range

0-235 (0=no limit)

The Intel(R) Ethernet Controller XL710 family supports an interrupt rate limiting mechanism. The user can control, via ethtool, the number of microseconds between interrupts.

Syntax:

```
# ethtool -C ethX rx-usecs-high N
```

The range of 0-235 microseconds provides an effective range of 4,310 to 250,000 interrupts per second. The value of rx-usecs-high can be set independently of rx-usecs and tx-usecs in the same ethtool command, and is also independent of the adaptive interrupt moderation algorithm. The underlying hardware supports granularity in 4-microsecond intervals, so adjacent values may result in the same interrupt rate.

One possible use case is the following:

```
# ethtool -C ethX adaptive-rx off adaptive-tx off rx-usecs-high 20 rx-usecs \
5 tx-usecs 5
```

The above command would disable adaptive interrupt moderation, and allow a maximum of 5 microseconds before indicating a receive or transmit was complete. However, instead of resulting in as many as 200,000 interrupts per second, it limits total interrupts per second to 50,000 via the rx-usecs-high parameter.

Performance Optimization

Driver defaults are meant to fit a wide variety of workloads, but if further optimization is required we recommend experimenting with the following settings.

NOTE: For better performance when processing small (64B) frame sizes, try enabling Hyper threading in the BIOS in order to increase the number of logical cores in the system and subsequently increase the number of queues available to the adapter.

Virtualized Environments

1. Disable XPS on both ends by using the included `virt_perf_default` script or by running the following command as root:

```
for file in `ls /sys/class/net/<ethX>/queues/tx-*`/xps_cpus`;
do echo 0 > $file; done
```

2. Using the appropriate mechanism (`vcpupin`) in the vm, pin the cpu's to individual lcpu's, making sure to use a set of cpu's included in the device's `local_cpulist`: `/sys/class/net/<ethX>/device/local_cpulist`.

3. Configure as many Rx/Tx queues in the VM as available. Do not rely on the default setting of 1.

Non-virtualized Environments

Pin the adapter's IRQs to specific cores by disabling the irqbalance service and using the included set_irq_affinity script. Please see the script's help text for further options.

- The following settings will distribute the IRQs across all the cores evenly:

```
# scripts/set_irq_affinity -x all <interface1> , [ <interface2>, ... ]
```

- The following settings will distribute the IRQs across all the cores that are local to the adapter (same NUMA node):

```
# scripts/set_irq_affinity -x local <interface1> ,[ <interface2>, ... ]
```

For very CPU intensive workloads, we recommend pinning the IRQs to all cores.

For IP Forwarding: Disable Adaptive ITR and lower Rx and Tx interrupts per queue using ethtool.

- Setting rx-usecs and tx-usecs to 125 will limit interrupts to about 8000 interrupts per second per queue.

```
# ethtool -C <interface> adaptive-rx off adaptive-tx off rx-usecs 125 \
tx-usecs 125
```

For lower CPU utilization: Disable Adaptive ITR and lower Rx and Tx interrupts per queue using ethtool.

- Setting rx-usecs and tx-usecs to 250 will limit interrupts to about 4000 interrupts per second per queue.

```
# ethtool -C <interface> adaptive-rx off adaptive-tx off rx-usecs 250 \
tx-usecs 250
```

For lower latency: Disable Adaptive ITR and ITR by setting Rx and Tx to 0 using ethtool.

```
# ethtool -C <interface> adaptive-rx off adaptive-tx off rx-usecs 0 \
tx-usecs 0
```

Application Device Queues (ADq)

Application Device Queues (ADq) allows you to dedicate one or more queues to a specific application. This can reduce latency for the specified application, and allow Tx traffic to be rate limited per application. Follow the steps below to set ADq.

1. Create traffic classes (TCs). Maximum of 8 TCs can be created per interface. The shaper bw_rlimit parameter is optional.

Example: Sets up two tcs, tc0 and tc1, with 16 queues each and max tx rate set to 1Gbit for tc0 and 3Gbit for tc1.

```
# tc qdisc add dev <interface> root mqprio num_tc 2 map 0 0 0 0 1 1 1 1
queues 16@0 16@16 hw 1 mode channel shaper bw_rlimit min_rate 1Gbit 2Gbit
max_rate 1Gbit 3Gbit
```

map: priority mapping for up to 16 priorities to tcs (e.g. map 0 0 0 0 1 1 1 1 sets priorities 0-3 to use tc0 and 4-7 to use tc1)

queues: for each tc, <num queues>@<offset> (e.g. queues 16@0 16@16 assigns 16 queues to tc0 at offset 0 and 16 queues to tc1 at offset 16. Max total number of queues for all tcs is 64 or number of cores, whichever is lower.)

hw 1 mode channel: ‘channel’ with ‘hw’ set to 1 is a new hardware offload mode in mqprior that makes full use of the mqprior options, the TCs, the queue configurations, and the QoS parameters.

shaper bw_rlimit: for each tc, sets minimum and maximum bandwidth rates. Totals must be equal or less than port speed.

For example: min_rate 1Gbit 3Gbit: Verify bandwidth limit using network monitoring tools such as *ifstat* or *sar -n DEV [interval] [number of samples]*

2. Enable HW TC offload on interface:

```
# ethtool -K <interface> hw-tc-offload on
```

3. Apply TCs to ingress (RX) flow of interface:

```
# tc qdisc add dev <interface> ingress
```

NOTES:

- Run all tc commands from the iproute2 <path to iproute2>/tc/ directory.
- ADq is not compatible with cloud filters.
- Setting up channels via ethtool (ethtool -L) is not supported when the TCs are configured using mqprior.
- You must have iproute2 latest version
- NVM version 6.01 or later is required.
- ADq cannot be enabled when any the following features are enabled: Data Center Bridging (DCB), Multiple Functions per Port (MFP), or Sideband Filters.
- If another driver (for example, DPDK) has set cloud filters, you cannot enable ADq.
- Tunnel filters are not supported in ADq. If encapsulated packets do arrive in non-tunnel mode, filtering will be done on the inner headers. For example, for VXLAN traffic in non-tunnel mode, PCTYPE is identified as a VXLAN encapsulated packet, outer headers are ignored. Therefore, inner headers are matched.
- If a TC filter on a PF matches traffic over a VF (on the PF), that traffic will be routed to the appropriate queue of the PF, and will not be passed on the VF. Such traffic will end up getting dropped higher up in the TCP/IP stack as it does not match PF address data.
- If traffic matches multiple TC filters that point to different TCs, that traffic will be duplicated and sent to all matching TC queues. The hardware switch mirrors the packet to a VSI list when multiple filters are matched.

Known Issues/Troubleshooting

NOTE: 1 Gb devices based on the Intel(R) Ethernet Network Connection X722 do not support the following features:

- Data Center Bridging (DCB)
- QOS
- VMQ
- SR-IOV
- Task Encapsulation offload (VXLAN, NVGRE)
- Energy Efficient Ethernet (EEE)
- Auto-media detect

Unexpected Issues when the device driver and DPDK share a device

Unexpected issues may result when an i40e device is in multi driver mode and the kernel driver and DPDK driver are sharing the device. This is because access to the global NIC resources is not synchronized between multiple drivers. Any change to the global NIC configuration (writing to a global register, setting global configuration by AQ, or changing switch modes) will affect all ports and drivers on the device. Loading DPDK with the "multi-driver" module parameter may mitigate some of the issues.

TC0 must be enabled when setting up DCB on a switch

The kernel assumes that TC0 is available, and will disable Priority Flow Control (PFC) on the device if TC0 is not available. To fix this, ensure TC0 is enabled when setting up DCB on your switch.

Support

For general information, go to the Intel support website at: <https://www.intel.com/support/>

If an issue is identified with the released source code on a supported kernel with a supported adapter, email the specific information related to the issue to intel-wired-lan@lists.osuosl.org.

6.5.29 Linux Base Driver for Intel(R) Ethernet Adaptive Virtual Function

Intel Ethernet Adaptive Virtual Function Linux driver. Copyright(c) 2013-2018 Intel Corporation.

Contents

- Overview
- Identifying Your Adapter
- Additional Configurations
- Known Issues/Troubleshooting
- Support

Overview

This file describes the iavf Linux Base Driver. This driver was formerly called i40evf.

The iavf driver supports the below mentioned virtual function devices and can only be activated on kernels running the i40e or newer Physical Function (PF) driver compiled with CONFIG_PCI_IOV. The iavf driver requires CONFIG_PCI_MSI to be enabled.

The guest OS loading the iavf driver must support MSI-X interrupts.

Identifying Your Adapter

The driver in this kernel is compatible with devices based on the following:

- Intel(R) XL710 X710 Virtual Function
- Intel(R) X722 Virtual Function
- Intel(R) XXV710 Virtual Function
- Intel(R) Ethernet Adaptive Virtual Function

For the best performance, make sure the latest NVM/FW is installed on your device.

For information on how to identify your adapter, and for the latest NVM/FW images and Intel network drivers, refer to the Intel Support website: <https://www.intel.com/support>

Additional Features and Configurations

Viewing Link Messages

Link messages will not be displayed to the console if the distribution is restricting system messages. In order to see network driver link messages on your console, set dmesg to eight by entering the following:

```
# dmesg -n 8
```

NOTE:

This setting is not saved across reboots.

ethtool

The driver utilizes the ethtool interface for driver configuration and diagnostics, as well as displaying statistical information. The latest ethtool version is required for this functionality. Download it at: <https://www.kernel.org/pub/software/network/ethtool/>

Setting VLAN Tag Stripping

If you have applications that require Virtual Functions (VFs) to receive packets with VLAN tags, you can disable VLAN tag stripping for the VF. The Physical Function (PF) processes requests issued from the VF to enable or disable VLAN tag stripping. Note that if the PF has assigned a VLAN to a VF, then requests from that VF to set VLAN tag stripping will be ignored.

To enable/disable VLAN tag stripping for a VF, issue the following command from inside the VM in which you are running the VF:

```
# ethtool -K <if_name> rxvlan on/off
```

or alternatively:

```
# ethtool --offload <if_name> rxvlan on/off
```

Adaptive Virtual Function

Adaptive Virtual Function (AVF) allows the virtual function driver, or VF, to adapt to changing feature sets of the physical function driver (PF) with which it is associated. This allows system administrators to update a PF without having to update all the VFs associated with it. All AVFs have a single common device ID and branding string.

AVFs have a minimum set of features known as "base mode," but may provide additional features depending on what features are available in the PF with which the AVF is associated. The following are base mode features:

- 4 Queue Pairs (QP) and associated Configuration Status Registers (CSRs) for Tx/Rx
- i40e descriptors and ring format
- Descriptor write-back completion
- 1 control queue, with i40e descriptors, CSRs and ring format
- 5 MSI-X interrupt vectors and corresponding i40e CSRs
- 1 Interrupt Throttle Rate (ITR) index
- 1 Virtual Station Interface (VSI) per VF
- 1 Traffic Class (TC), TC0
- Receive Side Scaling (RSS) with 64 entry indirection table and key, configured through the PF
- 1 unicast MAC address reserved per VF
- 16 MAC address filters for each VF

- Stateless offloads - non-tunneled checksums
- AVF device ID
- HW mailbox is used for VF to PF communications (including on Windows)

IEEE 802.1ad (QinQ) Support

The IEEE 802.1ad standard, informally known as QinQ, allows for multiple VLAN IDs within a single Ethernet frame. VLAN IDs are sometimes referred to as "tags," and multiple VLAN IDs are thus referred to as a "tag stack." Tag stacks allow L2 tunneling and the ability to segregate traffic within a particular VLAN ID, among other uses.

The following are examples of how to configure 802.1ad (QinQ):

```
# ip link add link eth0 eth0.24 type vlan proto 802.1ad id 24
# ip link add link eth0.24 eth0.24.371 type vlan proto 802.1Q id 371
```

Where "24" and "371" are example VLAN IDs.

NOTES:

Receive checksum offloads, cloud filters, and VLAN acceleration are not supported for 802.1ad (QinQ) packets.

Application Device Queues (ADq)

Application Device Queues (ADq) allows you to dedicate one or more queues to a specific application. This can reduce latency for the specified application, and allow Tx traffic to be rate limited per application. Follow the steps below to set ADq.

Requirements:

- The sch_mqpriv, act_mirred and cls_flower modules must be loaded
- The latest version of iproute2
- If another driver (for example, DPDK) has set cloud filters, you cannot enable ADQ
- Depending on the underlying PF device, ADQ cannot be enabled when the following features are enabled:
 - Data Center Bridging (DCB)
 - Multiple Functions per Port (MFP)
 - Sideband Filters

1. Create traffic classes (TCs). Maximum of 8 TCs can be created per interface. The shaper bw_rlimit parameter is optional.

Example: Sets up two tcs, tc0 and tc1, with 16 queues each and max tx rate set to 1Gbit for tc0 and 3Gbit for tc1.

```
tc qdisc add dev <interface> root mqpriv num_tc 2 map 0 0 0 0 1 1 1 1
queues 16@0 16@16 hw 1 mode channel shaper bw_rlimit min_rate 1Gbit 2Gbit
max_rate 1Gbit 3Gbit
```

map: priority mapping for up to 16 priorities to tc0s (e.g. map 0 0 0 0 1 1 1 1 sets priorities 0-3 to use tc0 and 4-7 to use tc1)

queues: for each tc, <num queues>@<offset> (e.g. queues 16@0 16@16 assigns 16 queues to tc0 at offset 0 and 16 queues to tc1 at offset 16. Max total number of queues for all tc0s is 64 or number of cores, whichever is lower.)

hw 1 mode channel: ‘channel’ with ‘hw’ set to 1 is a new hardware offload mode in mqprior that makes full use of the mqprior options, the TCs, the queue configurations, and the QoS parameters.

shaper bw_rlimit: for each tc, sets minimum and maximum bandwidth rates. Totals must be equal or less than port speed.

For example: min_rate 1Gbit 3Gbit: Verify bandwidth limit using network monitoring tools such as ifstat or sar -n DEV [interval] [number of samples]

NOTE:

Setting up channels via ethtool (ethtool -L) is not supported when the TCs are configured using mqprior.

2. Enable HW TC offload on interface:

```
# ethtool -K <interface> hw-tc-offload on
```

3. Apply TCs to ingress (RX) flow of interface:

```
# tc qdisc add dev <interface> ingress
```

NOTES:

- Run all tc commands from the iproute2 <path to iproute2>/tc/ directory
- ADq is not compatible with cloud filters
- Setting up channels via ethtool (ethtool -L) is not supported when the TCs are configured using mqprior
- You must have iproute2 latest version
- NVM version 6.01 or later is required
- ADq cannot be enabled when any the following features are enabled: Data Center Bridging (DCB), Multiple Functions per Port (MFP), or Sideband Filters
- If another driver (for example, DPDK) has set cloud filters, you cannot enable ADq
- Tunnel filters are not supported in ADq. If encapsulated packets do arrive in non-tunnel mode, filtering will be done on the inner headers. For example, for VXLAN traffic in non-tunnel mode, PCTYPE is identified as a VXLAN encapsulated packet, outer headers are ignored. Therefore, inner headers are matched.
- If a TC filter on a PF matches traffic over a VF (on the PF), that traffic will be routed to the appropriate queue of the PF, and will not be passed on the VF. Such traffic will end up getting dropped higher up in the TCP/IP stack as it does not match PF address data.
- If traffic matches multiple TC filters that point to different TCs, that traffic will be duplicated and sent to all matching TC queues. The hardware switch mirrors the packet to a VSI list when multiple filters are matched.

Known Issues/Troubleshooting

Bonding fails with VFs bound to an Intel(R) Ethernet Controller 700 series device

If you bind Virtual Functions (VFs) to an Intel(R) Ethernet Controller 700 series based device, the VF slaves may fail when they become the active slave. If the MAC address of the VF is set by the PF (Physical Function) of the device, when you add a slave, or change the active-backup slave, Linux bonding tries to sync the backup slave's MAC address to the same MAC address as the active slave. Linux bonding will fail at this point. This issue will not occur if the VF's MAC address is not set by the PF.

Traffic Is Not Being Passed Between VM and Client

You may not be able to pass traffic between a client system and a Virtual Machine (VM) running on a separate host if the Virtual Function (VF, or Virtual NIC) is not in trusted mode and spoof checking is enabled on the VF. Note that this situation can occur in any combination of client, host, and guest operating system. For information on how to set the VF to trusted mode, refer to the section "VLAN Tag Packet Steering" in this readme document. For information on setting spoof checking, refer to the section "MAC and VLAN anti-spoofing feature" in this readme document.

Do not unload port driver if VF with active VM is bound to it

Do not unload a port's driver if a Virtual Function (VF) with an active Virtual Machine (VM) is bound to it. Doing so will cause the port to appear to hang. Once the VM shuts down, or otherwise releases the VF, the command will complete.

Using four traffic classes fails

Do not try to reserve more than three traffic classes in the iavf driver. Doing so will fail to set any traffic classes and will cause the driver to write errors to stdout. Use a maximum of three queues to avoid this issue.

Multiple log error messages on iavf driver removal

If you have several VFs and you remove the iavf driver, several instances of the following log errors are written to the log:

```
Unable to send opcode 2 to PF, err I40E_ERR_QUEUE_EMPTY, aq_err ok
Unable to send the message to VF 2 aq_err 12
ARQ Overflow Error detected
```

Virtual machine does not get link

If the virtual machine has more than one virtual port assigned to it, and those virtual ports are bound to different physical ports, you may not get link on all of the virtual ports. The following command may work around the issue:

```
# ethtool -r <PF>
```

Where <PF> is the PF interface in the host, for example: p5p1. You may need to run the command more than once to get link on all virtual ports.

MAC address of Virtual Function changes unexpectedly

If a Virtual Function's MAC address is not assigned in the host, then the VF (virtual function) driver will use a random MAC address. This random MAC address may change each time the VF driver is reloaded. You can assign a static MAC address in the host machine. This static MAC address will survive a VF driver reload.

Driver Buffer Overflow Fix

The fix to resolve CVE-2016-8105, referenced in Intel SA-00069 <https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00069.html> is included in this and future versions of the driver.

Multiple Interfaces on Same Ethernet Broadcast Network

Due to the default ARP behavior on Linux, it is not possible to have one system on two IP networks in the same Ethernet broadcast domain (non-partitioned switch) behave as expected. All Ethernet interfaces will respond to IP traffic for any IP address assigned to the system. This results in unbalanced receive traffic.

If you have multiple interfaces in a server, either turn on ARP filtering by entering:

```
# echo 1 > /proc/sys/net/ipv4/conf/all/arp_filter
```

NOTE:

This setting is not saved across reboots. The configuration change can be made permanent by adding the following line to the file /etc/sysctl.conf:

```
net.ipv4.conf.all.arp_filter = 1
```

Another alternative is to install the interfaces in separate broadcast domains (either in different switches or in a switch partitioned to VLANs).

Rx Page Allocation Errors

'Page allocation failure. order:0' errors may occur under stress. This is caused by the way the Linux kernel reports this stressed condition.

Support

For general information, go to the Intel support website at: <https://support.intel.com>

If an issue is identified with the released source code on the supported kernel with a supported adapter, email the specific information related to the issue to intel-wired-lan@lists.osuosl.org.

6.5.30 Linux Base Driver for the Intel(R) Ethernet Controller 800 Series

Intel ice Linux driver. Copyright(c) 2018-2021 Intel Corporation.

Contents

- Overview
- Identifying Your Adapter
- Important Notes
- Additional Features & Configurations
- Performance Optimization

The associated Virtual Function (VF) driver for this driver is iavf.

Driver information can be obtained using ethtool and lspci.

For questions related to hardware requirements, refer to the documentation supplied with your Intel adapter. All hardware requirements listed apply to use with Linux.

This driver supports XDP (Express Data Path) and AF_XDP zero-copy. Note that XDP is blocked for frame sizes larger than 3KB.

Identifying Your Adapter

For information on how to identify your adapter, and for the latest Intel network drivers, refer to the Intel Support website: <https://www.intel.com/support>

Important Notes

Packet drops may occur under receive stress

Devices based on the Intel(R) Ethernet Controller 800 Series are designed to tolerate a limited amount of system latency during PCIe and DMA transactions. If these transactions take longer than the tolerated latency, it can impact the length of time the packets are buffered in the device and associated memory, which may result in dropped packets. These packet drops typically do not have a noticeable impact on throughput and performance under standard workloads.

If these packet drops appear to affect your workload, the following may improve the situation:

- 1) Make sure that your system's physical memory is in a high-performance configuration, as recommended by the platform vendor. A common recommendation is for all channels to be populated with a single DIMM module.
- 2) In your system's BIOS/UEFI settings, select the "Performance" profile.
- 3) Your distribution may provide tools like "tuned," which can help tweak kernel settings to achieve better standard settings for different workloads.

Configuring SR-IOV for improved network security

In a virtualized environment, on Intel(R) Ethernet Network Adapters that support SR-IOV, the virtual function (VF) may be subject to malicious behavior. Software-generated layer two frames, like IEEE 802.3x (link flow control), IEEE 802.1Qbb (priority based flow-control), and others of this type, are not expected and can throttle traffic between the host and the virtual switch, reducing performance. To resolve this issue, and to ensure isolation from unintended traffic streams, configure all SR-IOV enabled ports for VLAN tagging from the administrative interface on the PF. This configuration allows unexpected, and potentially malicious, frames to be dropped.

See "Configuring VLAN Tagging on SR-IOV Enabled Adapter Ports" later in this README for configuration instructions.

Do not unload port driver if VF with active VM is bound to it

Do not unload a port's driver if a Virtual Function (VF) with an active Virtual Machine (VM) is bound to it. Doing so will cause the port to appear to hang. Once the VM shuts down, or otherwise releases the VF, the command will complete.

Additional Features and Configurations

ethtool

The driver utilizes the ethtool interface for driver configuration and diagnostics, as well as displaying statistical information. The latest ethtool version is required for this functionality. Download it at: <https://kernel.org/pub/software/network/ethtool/>

NOTE: The rx_bytes value of ethtool does not match the rx_bytes value of Netdev, due to the 4-byte CRC being stripped by the device. The difference between the two rx_bytes values will be 4 x the number of Rx packets. For example, if Rx packets are 10 and Netdev (software statistics) displays rx_bytes as "X", then ethtool (hardware statistics) will display rx_bytes as "X+40" (4 bytes CRC x 10 packets).

Viewing Link Messages

Link messages will not be displayed to the console if the distribution is restricting system messages. In order to see network driver link messages on your console, set dmesg to eight by entering the following:

```
# dmesg -n 8
```

NOTE: This setting is not saved across reboots.

Dynamic Device Personalization

Dynamic Device Personalization (DDP) allows you to change the packet processing pipeline of a device by applying a profile package to the device at runtime. Profiles can be used to, for example, add support for new protocols, change existing protocols, or change default settings. DDP profiles can also be rolled back without rebooting the system.

The DDP package loads during device initialization. The driver looks for `intel/ice/ddp/ice.pkg` in your firmware root (typically `/lib/firmware/` or `/lib/firmware/updates/`) and checks that it contains a valid DDP package file.

NOTE: Your distribution should likely have provided the latest DDP file, but if `ice.pkg` is missing, you can find it in the `linux-firmware` repository or from intel.com.

If the driver is unable to load the DDP package, the device will enter Safe Mode. Safe Mode disables advanced and performance features and supports only basic traffic and minimal functionality, such as updating the NVM or downloading a new driver or DDP package. Safe Mode only applies to the affected physical function and does not impact any other PFs. See the "Intel(R) Ethernet Adapters and Devices User Guide" for more details on DDP and Safe Mode.

NOTES:

- If you encounter issues with the DDP package file, you may need to download an updated driver or DDP package file. See the log messages for more information.
- The `ice.pkg` file is a symbolic link to the default DDP package file.
- You cannot update the DDP package if any PF drivers are already loaded. To overwrite a package, unload all PFs and then reload the driver with the new package.
- Only the first loaded PF per device can download a package for that device.

You can install specific DDP package files for different physical devices in the same system. To install a specific DDP package file:

1. Download the DDP package file you want for your device.
2. Rename the file `ice-xxxxxxxxxxxxxx.pkg`, where '`xxxxxxxxxxxxxx`' is the unique 64-bit PCI Express device serial number (in hex) of the device you want the package downloaded on. The filename must include the complete serial number (including leading zeros) and be all lowercase. For example, if the 64-bit serial number is `b887a3ffffca0568`, then the file name would be `ice-b887a3ffffca0568.pkg`.

To find the serial number from the PCI bus address, you can use the following command:

```
# lspci -vv -s af:00.0 | grep -i Serial  
Capabilities: [150 v1] Device Serial Number b8-87-a3-ff-ff-ca-05-68
```

You can use the following command to format the serial number without the dashes:

```
# lspci -vv -s af:00.0 | grep -i Serial | awk '{print $7}' | sed s/-//g  
b887a3ffffca0568
```

3. Copy the renamed DDP package file to `/lib/firmware/updates/intel/ice/ddp/`. If the directory does not yet exist, create it before copying the file.
4. Unload all of the PFs on the device.
5. Reload the driver with the new package.

NOTE: The presence of a device-specific DDP package file overrides the loading of the default DDP package file (`ice.pkg`).

Intel(R) Ethernet Flow Director

The Intel Ethernet Flow Director performs the following tasks:

- Directs receive packets according to their flows to different queues
- Enables tight control on routing a flow in the platform
- Matches flows and CPU cores for flow affinity

NOTE: This driver supports the following flow types:

- IPv4
- TCPv4
- UDPv4
- SCTPv4
- IPv6
- TCPv6
- UDPv6
- SCTPv6

Each flow type supports valid combinations of IP addresses (source or destination) and UDP/TCP/SCTP ports (source and destination). You can supply only a source IP address, a source IP address and a destination port, or any combination of one or more of these four parameters.

NOTE: This driver allows you to filter traffic based on a user-defined flexible two-byte pattern and offset by using the `ethtool user-def` and `mask` fields. Only L3 and L4 flow types are supported for user-defined flexible filters. For a given flow type, you must clear all Intel Ethernet Flow Director filters before changing the input set (for that flow type).

Flow Director Filters

Flow Director filters are used to direct traffic that matches specified characteristics. They are enabled through ethtool's ntuple interface. To enable or disable the Intel Ethernet Flow Director and these filters:

```
# ethtool -K <ethX> ntuple <off|on>
```

NOTE: When you disable ntuple filters, all the user programmed filters are flushed from the driver cache and hardware. All needed filters must be re-added when ntuple is re-enabled.

To display all of the active filters:

```
# ethtool -u <ethX>
```

To add a new filter:

```
# ethtool -U <ethX> flow-type <type> src-ip <ip> [<ip_mask>] dst-ip <ip> [<ip_mask>] src-port <port> [<port_mask>] dst-port <port> [<port_mask>] action <queue>
```

Where:

- <ethX> - the Ethernet device to program
- <type> - can be ip4, tcp4, udp4, sctp4, ip6, tcp6, udp6, sctp6
- <ip> - the IP address to match on
- <ip_mask> - the IPv4 address to mask on
 - NOTE: These filters use inverted masks.
- <port> - the port number to match on
- <port_mask> - the 16-bit integer for masking
 - NOTE: These filters use inverted masks.
- <queue> - the queue to direct traffic toward (-1 discards the matched traffic)

To delete a filter:

```
# ethtool -U <ethX> delete <N>
```

Where <N> is the filter ID displayed when printing all the active filters, and may also have been specified using "loc <N>" when adding the filter.

EXAMPLES:

To add a filter that directs packet to queue 2:

```
# ethtool -U <ethX> flow-type tcp4 src-ip 192.168.10.1 dst-ip \
192.168.10.2 src-port 2000 dst-port 2001 action 2 [loc 1]
```

To set a filter using only the source and destination IP address:

```
# ethtool -U <ethX> flow-type tcp4 src-ip 192.168.10.1 dst-ip \
192.168.10.2 action 2 [loc 1]
```

To set a filter based on a user-defined pattern and offset:

```
# ethtool -U <ethX> flow-type tcp4 src-ip 192.168.10.1 dst-ip \
192.168.10.2 user-def 0xFFFF action 2 [loc 1]
```

where the value of the user-def field contains the offset (4 bytes) and the pattern (0xffff).

To match TCP traffic sent from 192.168.0.1, port 5300, directed to 192.168.0.5, port 80, and then send it to queue 7:

```
# ethtool -U enp130s0 flow-type tcp4 src-ip 192.168.0.1 dst-ip 192.168.0.5
src-port 5300 dst-port 80 action 7
```

To add a TCPv4 filter with a partial mask for a source IP subnet:

```
# ethtool -U <ethX> flow-type tcp4 src-ip 192.168.0.0 m 0.255.255.255 dst-ip
192.168.5.12 src-port 12600 dst-port 31 action 12
```

NOTES:

For each flow-type, the programmed filters must all have the same matching input set. For example, issuing the following two commands is acceptable:

```
# ethtool -U enp130s0 flow-type ip4 src-ip 192.168.0.1 src-port 5300 action 7
# ethtool -U enp130s0 flow-type ip4 src-ip 192.168.0.5 src-port 55 action 10
```

Issuing the next two commands, however, is not acceptable, since the first specifies src-ip and the second specifies dst-ip:

```
# ethtool -U enp130s0 flow-type ip4 src-ip 192.168.0.1 src-port 5300 action 7
# ethtool -U enp130s0 flow-type ip4 dst-ip 192.168.0.5 src-port 55 action 10
```

The second command will fail with an error. You may program multiple filters with the same fields, using different values, but, on one device, you may not program two tcp4 filters with different matching fields.

The ice driver does not support matching on a subportion of a field, thus partial mask fields are not supported.

Flex Byte Flow Director Filters

The driver also supports matching user-defined data within the packet payload. This flexible data is specified using the "user-def" field of the ethtool command in the following way:

31 28 24 20 16	15 12 8 4 0
offset into packet payload	2 bytes of flexible data

For example,

```
... user-def 0xFFFF ...
```

tells the filter to look 4 bytes into the payload and match that value against 0xFFFF. The offset is based on the beginning of the payload, and not the beginning of the packet. Thus

```
flow-type tcp4 ... user-def 0x8BEAF ...
```

would match TCP/IPv4 packets which have the value 0xBEAF 8 bytes into the TCP/IPv4 payload.

Note that ICMP headers are parsed as 4 bytes of header and 4 bytes of payload. Thus to match the first byte of the payload, you must actually add 4 bytes to the offset. Also note that ip4 filters match both ICMP frames as well as raw (unknown) ip4 frames, where the payload will be the L3 payload of the IP4 frame.

The maximum offset is 64. The hardware will only read up to 64 bytes of data from the payload. The offset must be even because the flexible data is 2 bytes long and must be aligned to byte 0 of the packet payload.

The user-defined flexible offset is also considered part of the input set and cannot be programmed separately for multiple filters of the same type. However, the flexible data is not part of the input set and multiple filters may use the same offset but match against different data.

RSS Hash Flow

Allows you to set the hash bytes per flow type and any combination of one or more options for Receive Side Scaling (RSS) hash byte configuration.

```
# ethtool -N <ethX> rx-flow-hash <type> <option>
```

Where <type> is:

- tcp4 signifying TCP over IPv4
- udp4 signifying UDP over IPv4
- tcp6 signifying TCP over IPv6
- udp6 signifying UDP over IPv6

And <option> is one or more of:

- s Hash on the IP source address of the Rx packet.
- d Hash on the IP destination address of the Rx packet.
- f Hash on bytes 0 and 1 of the Layer 4 header of the Rx packet.
- n Hash on bytes 2 and 3 of the Layer 4 header of the Rx packet.

Accelerated Receive Flow Steering (aRFS)

Devices based on the Intel(R) Ethernet Controller 800 Series support Accelerated Receive Flow Steering (aRFS) on the PF. aRFS is a load-balancing mechanism that allows you to direct packets to the same CPU where an application is running or consuming the packets in that flow.

NOTES:

- aRFS requires that ntuple filtering is enabled via ethtool.
- aRFS support is limited to the following packet types:
 - TCP over IPv4 and IPv6
 - UDP over IPv4 and IPv6
 - Nonfragmented packets

- aRFS only supports Flow Director filters, which consist of the source/destination IP addresses and source/destination ports.
- aRFS and ethtool's ntuple interface both use the device's Flow Director. aRFS and ntuple features can coexist, but you may encounter unexpected results if there's a conflict between aRFS and ntuple requests. See "Intel(R) Ethernet Flow Director" for additional information.

To set up aRFS:

1. Enable the Intel Ethernet Flow Director and ntuple filters using ethtool.

```
# ethtool -K <ethX> ntuple on
```

2. Set up the number of entries in the global flow table. For example:

```
# NUM_RPS_ENTRIES=16384
# echo $NUM_RPS_ENTRIES > /proc/sys/net/core/rps_sock_flow_entries
```

3. Set up the number of entries in the per-queue flow table. For example:

```
# NUM_RX_QUEUES=64
# for file in /sys/class/net/$IFACE/queues/rx-*/*; do
#   echo $((NUM_RPS_ENTRIES/NUM_RX_QUEUES)) > $file;
# done
```

4. Disable the IRQ balance daemon (this is only a temporary stop of the service until the next reboot).

```
# systemctl stop irqbalance
```

5. Configure the interrupt affinity.

See [/Documentation/core-api/irq/irq-affinity.rst](#)

To disable aRFS using ethtool:

```
# ethtool -K <ethX> ntuple off
```

NOTE: This command will disable ntuple filters and clear any aRFS filters in software and hardware.

Example Use Case:

1. Set the server application on the desired CPU (e.g., CPU 4).

```
# taskset -c 4 netserver
```

2. Use netperf to route traffic from the client to CPU 4 on the server with aRFS configured. This example uses TCP over IPv4.

```
# netperf -H <Host IPv4 Address> -t TCP_STREAM
```

Enabling Virtual Functions (VFs)

Use sysfs to enable virtual functions (VF).

For example, you can create 4 VFs as follows:

```
# echo 4 > /sys/class/net/<ethX>/device/sriov_numvfs
```

To disable VFs, write 0 to the same file:

```
# echo 0 > /sys/class/net/<ethX>/device/sriov_numvfs
```

The maximum number of VFs for the ice driver is 256 total (all ports). To check how many VFs each PF supports, use the following command:

```
# cat /sys/class/net/<ethX>/device/sriov_totalvfs
```

Note: You cannot use SR-IOV when link aggregation (LAG)/bonding is active, and vice versa. To enforce this, the driver checks for this mutual exclusion.

Displaying VF Statistics on the PF

Use the following command to display the statistics for the PF and its VFs:

```
# ip -s link show dev <ethX>
```

NOTE: The output of this command can be very large due to the maximum number of possible VFs.

The PF driver will display a subset of the statistics for the PF and for all VFs that are configured. The PF will always print a statistics block for each of the possible VFs, and it will show zero for all unconfigured VFs.

Configuring VLAN Tagging on SR-IOV Enabled Adapter Ports

To configure VLAN tagging for the ports on an SR-IOV enabled adapter, use the following command. The VLAN configuration should be done before the VF driver is loaded or the VM is booted. The VF is not aware of the VLAN tag being inserted on transmit and removed on received frames (sometimes called "port VLAN" mode).

```
# ip link set dev <ethX> vf <id> vlan <vlan id>
```

For example, the following will configure PF eth0 and the first VF on VLAN 10:

```
# ip link set dev eth0 vf 0 vlan 10
```

Enabling a VF link if the port is disconnected

If the physical function (PF) link is down, you can force link up (from the host PF) on any virtual functions (VF) bound to the PF.

For example, to force link up on VF 0 bound to PF eth0:

```
# ip link set eth0 vf 0 state enable
```

Note: If the command does not work, it may not be supported by your system.

Setting the MAC Address for a VF

To change the MAC address for the specified VF:

```
# ip link set <ethX> vf 0 mac <address>
```

For example:

```
# ip link set <ethX> vf 0 mac 00:01:02:03:04:05
```

This setting lasts until the PF is reloaded.

NOTE: Assigning a MAC address for a VF from the host will disable any subsequent requests to change the MAC address from within the VM. This is a security feature. The VM is not aware of this restriction, so if this is attempted in the VM, it will trigger MDD events.

Trusted VFs and VF Promiscuous Mode

This feature allows you to designate a particular VF as trusted and allows that trusted VF to request selective promiscuous mode on the Physical Function (PF).

To set a VF as trusted or untrusted, enter the following command in the Hypervisor:

```
# ip link set dev <ethX> vf 1 trust [on|off]
```

NOTE: It's important to set the VF to trusted before setting promiscuous mode. If the VM is not trusted, the PF will ignore promiscuous mode requests from the VF. If the VM becomes trusted after the VF driver is loaded, you must make a new request to set the VF to promiscuous.

Once the VF is designated as trusted, use the following commands in the VM to set the VF to promiscuous mode.

For promiscuous all:

```
# ip link set <ethX> promisc on  
Where <ethX> is a VF interface in the VM
```

For promiscuous Multicast:

```
# ip link set <ethX> allmulticast on  
Where <ethX> is a VF interface in the VM
```

NOTE: By default, the ethtool private flag vf-true-promisc-support is set to "off," meaning that promiscuous mode for the VF will be limited. To set the promiscuous mode for the VF to true promiscuous and allow the VF to see all ingress traffic, use the following command:

```
# ethtool --set-priv-flags <ethX> vf-true-promisc-support on
```

The vf-true-promisc-support private flag does not enable promiscuous mode; rather, it designates which type of promiscuous mode (limited or true) you will get when you enable promiscuous mode using the ip link commands above. Note that this is a global setting that affects the entire device. However, the vf-true-promisc-support private flag is only exposed to the first PF of the device. The PF remains in limited promiscuous mode regardless of the vf-true-promisc-support setting.

Next, add a VLAN interface on the VF interface. For example:

```
# ip link add link eth2 name eth2.100 type vlan id 100
```

Note that the order in which you set the VF to promiscuous mode and add the VLAN interface does not matter (you can do either first). The result in this example is that the VF will get all traffic that is tagged with VLAN 100.

Malicious Driver Detection (MDD) for VFs

Some Intel Ethernet devices use Malicious Driver Detection (MDD) to detect malicious traffic from the VF and disable Tx/Rx queues or drop the offending packet until a VF driver reset occurs. You can view MDD messages in the PF's system log using the dmesg command.

- If the PF driver logs MDD events from the VF, confirm that the correct VF driver is installed.
- To restore functionality, you can manually reload the VF or VM or enable automatic VF resets.
- When automatic VF resets are enabled, the PF driver will immediately reset the VF and reenable queues when it detects MDD events on the receive path.
- If automatic VF resets are disabled, the PF will not automatically reset the VF when it detects MDD events.

To enable or disable automatic VF resets, use the following command:

```
# ethtool --set-priv-flags <ethX> mdd-auto-reset-vf on|off
```

MAC and VLAN Anti-Spoofing Feature for VFs

When a malicious driver on a Virtual Function (VF) interface attempts to send a spoofed packet, it is dropped by the hardware and not transmitted.

NOTE: This feature can be disabled for a specific VF:

```
# ip link set <ethX> vf <vf id> spoofchk {off|on}
```

Jumbo Frames

Jumbo Frames support is enabled by changing the Maximum Transmission Unit (MTU) to a value larger than the default value of 1500.

Use the ifconfig command to increase the MTU size. For example, enter the following where <ethX> is the interface number:

```
# ifconfig <ethX> mtu 9000 up
```

Alternatively, you can use the ip command as follows:

```
# ip link set mtu 9000 dev <ethX>
# ip link set up dev <ethX>
```

This setting is not saved across reboots.

NOTE: The maximum MTU setting for jumbo frames is 9702. This corresponds to the maximum jumbo frame size of 9728 bytes.

NOTE: This driver will attempt to use multiple page sized buffers to receive each jumbo packet. This should help to avoid buffer starvation issues when allocating receive packets.

NOTE: Packet loss may have a greater impact on throughput when you use jumbo frames. If you observe a drop in performance after enabling jumbo frames, enabling flow control may mitigate the issue.

Speed and Duplex Configuration

In addressing speed and duplex configuration issues, you need to distinguish between copper-based adapters and fiber-based adapters.

In the default mode, an Intel(R) Ethernet Network Adapter using copper connections will attempt to auto-negotiate with its link partner to determine the best setting. If the adapter cannot establish link with the link partner using auto-negotiation, you may need to manually configure the adapter and link partner to identical settings to establish link and pass packets. This should only be needed when attempting to link with an older switch that does not support auto-negotiation or one that has been forced to a specific speed or duplex mode. Your link partner must match the setting you choose. 1 Gbps speeds and higher cannot be forced. Use the autonegotiation advertising setting to manually set devices for 1 Gbps and higher.

Speed, duplex, and autonegotiation advertising are configured through the ethtool utility. For the latest version, download and install ethtool from the following website:

<https://kernel.org/pub/software/network/ethtool/>

To see the speed configurations your device supports, run the following:

```
# ethtool <ethX>
```

Caution: Only experienced network administrators should force speed and duplex or change autonegotiation advertising manually. The settings at the switch must always match the adapter settings. Adapter performance may suffer or your adapter may not operate if you configure the adapter differently from your switch.

Data Center Bridging (DCB)

NOTE: The kernel assumes that TC0 is available, and will disable Priority Flow Control (PFC) on the device if TC0 is not available. To fix this, ensure TC0 is enabled when setting up DCB on your switch.

DCB is a configuration Quality of Service implementation in hardware. It uses the VLAN priority tag (802.1p) to filter traffic. That means that there are 8 different priorities that traffic can be filtered into. It also enables priority flow control (802.1Qbb) which can limit or eliminate the number of dropped packets during network stress. Bandwidth can be allocated to each of these priorities, which is enforced at the hardware level (802.1Qaz).

DCB is normally configured on the network using the DCBX protocol (802.1Qaz), a specialization of LLDP (802.1AB). The ice driver supports the following mutually exclusive variants of DCBX support:

- 1) Firmware-based LLDP Agent
- 2) Software-based LLDP Agent

In firmware-based mode, firmware intercepts all LLDP traffic and handles DCBX negotiation transparently for the user. In this mode, the adapter operates in "willing" DCBX mode, receiving DCB settings from the link partner (typically a switch). The local user can only query the negotiated DCB configuration. For information on configuring DCBX parameters on a switch, please consult the switch manufacturer's documentation.

In software-based mode, LLDP traffic is forwarded to the network stack and user space, where a software agent can handle it. In this mode, the adapter can operate in either "willing" or "nonwilling" DCBX mode and DCB configuration can be both queried and set locally. This mode requires the FW-based LLDP Agent to be disabled.

NOTE:

- You can enable and disable the firmware-based LLDP Agent using an ethtool private flag. Refer to the "FW-LLDP (Firmware Link Layer Discovery Protocol)" section in this README for more information.
- In software-based DCBX mode, you can configure DCB parameters using software LLDP/DCBX agents that interface with the Linux kernel's DCB Netlink API. We recommend using OpenLLDP as the DCBX agent when running in software mode. For more information, see the OpenLLDP man pages and <https://github.com/intel/openlldp>.
- The driver implements the DCB netlink interface layer to allow the user space to communicate with the driver and query DCB configuration for the port.
- iSCSI with DCB is not supported.

FW-LLDP (Firmware Link Layer Discovery Protocol)

Use ethtool to change FW-LLDP settings. The FW-LLDP setting is per port and persists across boots.

To enable LLDP:

```
# ethtool --set-priv-flags <ethX> fw-lldp-agent on
```

To disable LLDP:

```
# ethtool --set-priv-flags <ethX> fw-lldp-agent off
```

To check the current LLDP setting:

```
# ethtool --show-priv-flags <ethX>
```

NOTE: You must enable the UEFI HII "LLDP Agent" attribute for this setting to take effect. If "LLDP AGENT" is set to disabled, you cannot enable it from the OS.

Flow Control

Ethernet Flow Control (IEEE 802.3x) can be configured with ethtool to enable receiving and transmitting pause frames for ice. When transmit is enabled, pause frames are generated when the receive packet buffer crosses a predefined threshold. When receive is enabled, the transmit unit will halt for the time delay specified when a pause frame is received.

NOTE: You must have a flow control capable link partner.

Flow Control is disabled by default.

Use ethtool to change the flow control settings.

To enable or disable Rx or Tx Flow Control:

```
# ethtool -A <ethX> rx <on|off> tx <on|off>
```

Note: This command only enables or disables Flow Control if auto-negotiation is disabled. If auto-negotiation is enabled, this command changes the parameters used for auto-negotiation with the link partner.

Note: Flow Control auto-negotiation is part of link auto-negotiation. Depending on your device, you may not be able to change the auto-negotiation setting.

NOTE:

- The ice driver requires flow control on both the port and link partner. If flow control is disabled on one of the sides, the port may appear to hang on heavy traffic.
- You may encounter issues with link-level flow control (LFC) after disabling DCB. The LFC status may show as enabled but traffic is not paused. To resolve this issue, disable and reenable LFC using ethtool:

```
# ethtool -A <ethX> rx off tx off
# ethtool -A <ethX> rx on tx on
```

NAPI

This driver supports NAPI (Rx polling mode).

See [Documentation/networking/napi.rst](#) for more information.

MACVLAN

This driver supports MACVLAN. Kernel support for MACVLAN can be tested by checking if the MACVLAN driver is loaded. You can run 'lsmod | grep macvlan' to see if the MACVLAN driver is loaded or run 'modprobe macvlan' to try to load the MACVLAN driver.

NOTE:

- In passthru mode, you can only set up one MACVLAN device. It will inherit the MAC address of the underlying PF (Physical Function) device.

IEEE 802.1ad (QinQ) Support

The IEEE 802.1ad standard, informally known as QinQ, allows for multiple VLAN IDs within a single Ethernet frame. VLAN IDs are sometimes referred to as "tags," and multiple VLAN IDs are thus referred to as a "tag stack." Tag stacks allow L2 tunneling and the ability to segregate traffic within a particular VLAN ID, among other uses.

NOTES:

- Receive checksum offloads and VLAN acceleration are not supported for 802.1ad (QinQ) packets.
- 0x88A8 traffic will not be received unless VLAN stripping is disabled with the following command:

```
# ethtool -K <ethX> rxvlan off
```

- 0x88A8/0x8100 double VLANs cannot be used with 0x8100 or 0x8100/0x8100 VLANs configured on the same port. 0x88A8/0x8100 traffic will not be received if 0x8100 VLANs are configured.
- The VF can only transmit 0x88A8/0x8100 (i.e., 802.1ad/802.1Q) traffic if:
 - 1) The VF is not assigned a port VLAN.
 - 2) spoofchk is disabled from the PF. If you enable spoofchk, the VF will not transmit 0x88A8/0x8100 traffic.
- The VF may not receive all network traffic based on the Inner VLAN header when VF true promiscuous mode (vf-true-promisc-support) and double VLANs are enabled in SR-IOV mode.

The following are examples of how to configure 802.1ad (QinQ):

```
# ip link add link eth0 eth0.24 type vlan proto 802.1ad id 24
# ip link add link eth0.24 eth0.24.371 type vlan proto 802.1Q id 371
```

Where "24" and "371" are example VLAN IDs.

Tunnel/Overlay Stateless Offloads

Supported tunnels and overlays include VXLAN, GENEVE, and others depending on hardware and software configuration. Stateless offloads are enabled by default.

To view the current state of all offloads:

```
# ethtool -k <ethX>
```

UDP Segmentation Offload

Allows the adapter to offload transmit segmentation of UDP packets with payloads up to 64K into valid Ethernet frames. Because the adapter hardware is able to complete data segmentation much faster than operating system software, this feature may improve transmission performance. In addition, the adapter may use fewer CPU resources.

NOTE:

- The application sending UDP packets must support UDP segmentation offload.

To enable/disable UDP Segmentation Offload, issue the following command:

```
# ethtool -K <ethX> tx-udp-segmentation [off|on]
```

GNSS module

Requires kernel compiled with CONFIG_GNSS=y or CONFIG_GNSS=m. Allows user to read messages from the GNSS hardware module and write supported commands. If the module is physically present, a GNSS device is spawned: /dev/gnss<id>. The protocol of write command is dependent on the GNSS hardware module as the driver writes raw bytes by the GNSS object to the receiver through i2c. Please refer to the hardware GNSS module documentation for configuration details.

Firmware (FW) logging

The driver supports FW logging via the debugfs interface on PF 0 only. The FW running on the NIC must support FW logging; if the FW doesn't support FW logging the 'fwlog' file will not get created in the ice debugfs directory.

Module configuration

Firmware logging is configured on a per module basis. Each module can be set to a value independent of the other modules (unless the module 'all' is specified). The modules will be instantiated under the 'fwlog/modules' directory.

The user can set the log level for a module by writing to the module file like this:

```
# echo <log_level> > /sys/kernel/debug/ice/0000\:18\:00.0/fwlog/modules/
↳<module>
```

where

- `log_level` is a name as described below. Each level includes the messages from the previous/lower level
 - `none`
 - `error`
 - `warning`
 - `normal`
 - `verbose`
- `module` is a name that represents the module to receive events for. The module names are
 - `general`
 - `ctrl`
 - `link`
 - `link_topo`
 - `dnl`
 - `i2c`
 - `sdp`
 - `mdio`
 - `adminq`
 - `hdma`
 - `lldp`
 - `dcbx`
 - `dcb`
 - `xlr`
 - `nvm`
 - `auth`
 - `vpd`
 - `iosf`
 - `parser`
 - `sw`
 - `scheduler`
 - `txq`
 - `rsvd`
 - `post`
 - `watchdog`
 - `task_dispatch`

- mng
- sync
- health
- tsdrv
- pfreg
- mdlver
- all

The name 'all' is special and allows the user to set all of the modules to the specified log_level or to read the log_level of all of the modules.

Example usage to configure the modules

To set a single module to 'verbose':

```
# echo verbose > /sys/kernel/debug/ice/0000\:18\:00.0/fwlog/modules/link
```

To set multiple modules then issue the command multiple times:

```
# echo verbose > /sys/kernel/debug/ice/0000\:18\:00.0/fwlog/modules/link
# echo warning > /sys/kernel/debug/ice/0000\:18\:00.0/fwlog/modules/ctrl
# echo none > /sys/kernel/debug/ice/0000\:18\:00.0/fwlog/modules/dcb
```

To set all the modules to the same value:

```
# echo normal > /sys/kernel/debug/ice/0000\:18\:00.0/fwlog/modules/all
```

To read the log_level of a specific module (e.g. module 'general'):

```
# cat /sys/kernel/debug/ice/0000\:18\:00.0/fwlog/modules/general
```

To read the log_level of all the modules:

```
# cat /sys/kernel/debug/ice/0000\:18\:00.0/fwlog/modules/all
```

Enabling FW log

Configuring the modules indicates to the FW that the configured modules should generate events that the driver is interested in, but it **does not** send the events to the driver until the enable message is sent to the FW. To do this the user can write a 1 (enable) or 0 (disable) to 'fwlog/enable'. An example is:

```
# echo 1 > /sys/kernel/debug/ice/0000\:18\:00.0/fwlog/enable
```

Retrieving FW log data

The FW log data can be retrieved by reading from 'fwlog/data'. The user can write any value to 'fwlog/data' to clear the data. The data can only be cleared when FW logging is disabled. The FW log data is a binary file that is sent to Intel and used to help debug user issues.

An example to read the data is:

```
# cat /sys/kernel/debug/ice/0000\:18\:00.0/fwlog/data > fwlog.bin
```

An example to clear the data is:

```
# echo 0 > /sys/kernel/debug/ice/0000\:18\:00.0/fwlog/data
```

Changing how often the log events are sent to the driver

The driver receives FW log data from the Admin Receive Queue (ARQ). The frequency that the FW sends the ARQ events can be configured by writing to 'fwlog/nr_messages'. The range is 1-128 (1 means push every log message, 128 means push only when the max AQ command buffer is full). The suggested value is 10. The user can see what the value is configured to by reading 'fwlog/nr_messages'. An example to set the value is:

```
# echo 50 > /sys/kernel/debug/ice/0000\:18\:00.0/fwlog/nr_messages
```

Configuring the amount of memory used to store FW log data

The driver stores FW log data within the driver. The default size of the memory used to store the data is 1MB. Some use cases may require more or less data so the user can change the amount of memory that is allocated for FW log data. To change the amount of memory then write to 'fwlog/log_size'. The value must be one of: 128K, 256K, 512K, 1M, or 2M. FW logging must be disabled to change the value. An example of changing the value is:

```
# echo 128K > /sys/kernel/debug/ice/0000\:18\:00.0/fwlog/log_size
```

Performance Optimization

Driver defaults are meant to fit a wide variety of workloads, but if further optimization is required, we recommend experimenting with the following settings.

Rx Descriptor Ring Size

To reduce the number of Rx packet discards, increase the number of Rx descriptors for each Rx ring using ethtool.

Check if the interface is dropping Rx packets due to buffers being full (rx_dropped.nic can mean that there is no PCIe bandwidth):

```
# ethtool -S <ethX> | grep "rx_dropped"
```

If the previous command shows drops on queues, it may help to increase the number of descriptors using 'ethtool -G':

```
# ethtool -G <ethX> rx <N>
Where <N> is the desired number of ring entries/descriptors
```

This can provide temporary buffering for issues that create latency while the CPUs process descriptors.

Interrupt Rate Limiting

This driver supports an adaptive interrupt throttle rate (ITR) mechanism that is tuned for general workloads. The user can customize the interrupt rate control for specific workloads, via ethtool, adjusting the number of microseconds between interrupts.

To set the interrupt rate manually, you must disable adaptive mode:

```
# ethtool -C <ethX> adaptive-rx off adaptive-tx off
```

For lower CPU utilization:

Disable adaptive ITR and lower Rx and Tx interrupts. The examples below affect every queue of the specified interface.

Setting rx-usecs and tx-usecs to 80 will limit interrupts to about 12,500 interrupts per second per queue:

```
# ethtool -C <ethX> adaptive-rx off adaptive-tx off rx-usecs 80 tx-
↪usecs 80
```

For reduced latency:

Disable adaptive ITR and ITR by setting rx-usecs and tx-usecs to 0 using ethtool:

```
# ethtool -C <ethX> adaptive-rx off adaptive-tx off rx-usecs 0 tx-
↪usecs 0
```

Per-queue interrupt rate settings:

The following examples are for queues 1 and 3, but you can adjust other queues.

To disable Rx adaptive ITR and set static Rx ITR to 10 microseconds or about 100,000 interrupts/second, for queues 1 and 3:

```
# ethtool --per-queue <ethX> queue_mask 0xa --coalesce adaptive-rx off
rx-usecs 10
```

To show the current coalesce settings for queues 1 and 3:

```
# ethtool --per-queue <ethX> queue_mask 0xa --show-coalesce
```

Bounding interrupt rates using rx-usecs-high:

Valid Range

0-236 (0=no limit)

The range of 0-236 microseconds provides an effective range of 4,237 to 250,000 interrupts per second. The value of rx-usecs-high can be set independently of rx-usecs and tx-usecs in the same ethtool command, and is also independent of the adaptive interrupt moderation algorithm. The underlying hardware supports granularity in 4-microsecond intervals, so adjacent values may result in the same interrupt rate.

The following command would disable adaptive interrupt moderation, and allow a maximum of 5 microseconds before indicating a receive or transmit was complete. However, instead of resulting in as many as 200,000 interrupts per second, it limits total interrupts per second to 50,000 via the rx-usecs-high parameter.

```
# ethtool -C <ethX> adaptive-rx off adaptive-tx off rx-usecs-high 20
rx-usecs 5 tx-usecs 5
```

Virtualized Environments

In addition to the other suggestions in this section, the following may be helpful to optimize performance in VMs.

Using the appropriate mechanism (vcpupin) in the VM, pin the CPUs to individual LCPUs, making sure to use a set of CPUs included in the device's local_cpulist: /sys/class/net/<ethX>/device/local_cpulist.

Configure as many Rx/Tx queues in the VM as available. (See the iavf driver documentation for the number of queues supported.) For example:

```
# ethtool -L <virt_interface> rx <max> tx <max>
```

Support

For general information, go to the Intel support website at: <https://www.intel.com/support/>

If an issue is identified with the released source code on a supported kernel with a supported adapter, email the specific information related to the issue to intel-wired-lan@lists.osuosl.org.

Trademarks

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and/or other countries.

- Other names and brands may be claimed as the property of others.

6.5.31 Marvell OcteonTx2 RVU Kernel Drivers

Copyright (c) 2020 Marvell International Ltd.

Contents

- *Overview*
- *Drivers*
- *Basic packet flow*
- *Devlink health reporters*
- *Quality of service*

Overview

Resource virtualization unit (RVU) on Marvell's OcteonTX2 SOC maps HW resources from the network, crypto and other functional blocks into PCI-compatible physical and virtual functions. Each functional block again has multiple local functions (LFs) for provisioning to PCI devices. RVU supports multiple PCIe SRIOV physical functions (PFs) and virtual functions (VFs). PF0 is called the administrative / admin function (AF) and has privileges to provision RVU functional block's LFs to each of the PF/VF.

RVU managed networking functional blocks

- Network pool or buffer allocator (NPA)
- Network interface controller (NIX)
- Network parser CAM (NPC)
- Schedule/Synchronize/Order unit (SSO)
- Loopback interface (LBK)

RVU managed non-networking functional blocks

- Crypto accelerator (CPT)
- Scheduled timers unit (TIM)
- Schedule/Synchronize/Order unit (SSO) Used for both networking and non networking usecases

Resource provisioning examples

- A PF/VF with NIX-LF & NPA-LF resources works as a pure network device
- A PF/VF with CPT-LF resource works as a pure crypto offload device.

RVU functional blocks are highly configurable as per software requirements.

Firmware setups following stuff before kernel boots

- Enables required number of RVU PFs based on number of physical links.
- Number of VFs per PF are either static or configurable at compile time. Based on config, firmware assigns VFs to each of the PFs.
- Also assigns MSIX vectors to each of PF and VFs.
- These are not changed after kernel boot.

Drivers

Linux kernel will have multiple drivers registering to different PF and VFs of RVU. Wrt networking there will be 3 flavours of drivers.

Admin Function driver

As mentioned above RVU PF0 is called the admin function (AF), this driver supports resource provisioning and configuration of functional blocks. Doesn't handle any I/O. It sets up few basic stuff but most of the functionality is achieved via configuration requests from PFs and VFs.

PF/VFs communicates with AF via a shared memory region (mailbox). Upon receiving requests AF does resource provisioning and other HW configuration. AF is always attached to host kernel, but PFs and their VFs may be used by host kernel itself, or attached to VMs or to userspace applications like DPDK etc. So AF has to handle provisioning/configuration requests sent by any device from any domain.

AF driver also interacts with underlying firmware to

- Manage physical ethernet links ie CGX LMACs.
- Retrieve information like speed, duplex, autoneg etc
- Retrieve PHY EEPROM and stats.
- Configure FEC, PAM modes
- etc

From pure networking side AF driver supports following functionality.

- Map a physical link to a RVU PF to which a netdev is registered.
- Attach NIX and NPA block LFs to RVU PF/VF which provide buffer pools, RQs, SQs for regular networking functionality.
- Flow control (pause frames) enable/disable/config.
- HW PTP timestamping related config.
- NPC parser profile config, basically how to parse pkt and what info to extract.
- NPC extract profile config, what to extract from the pkt to match data in MCAM entries.
- Manage NPC MCAM entries, upon request can frame and install requested packet forwarding rules.

- Defines receive side scaling (RSS) algorithms.
- Defines segmentation offload algorithms (eg TSO)
- VLAN stripping, capture and insertion config.
- SSO and TIM blocks config which provide packet scheduling support.
- Debugfs support, to check current resource provisioning, current status of NPA pools, NIX RQ, SQ and CQs, various stats etc which helps in debugging issues.
- And many more.

Physical Function driver

This RVU PF handles IO, is mapped to a physical ethernet link and this driver registers a net-dev. This supports SR-IOV. As said above this driver communicates with AF with a mailbox. To retrieve information from physical links this driver talks to AF and AF gets that info from firmware and responds back ie cannot talk to firmware directly.

Supports ethtool for configuring links, RSS, queue count, queue size, flow control, ntuple filters, dump PHY EEPROM, config FEC etc.

Virtual Function driver

There are two types VFs, VFs that share the physical link with their parent SR-IOV PF and the VFs which work in pairs using internal HW loopback channels (LBK).

Type1:

- These VFs and their parent PF share a physical link and used for outside communication.
- VFs cannot communicate with AF directly, they send mbox message to PF and PF forwards that to AF. AF after processing, responds back to PF and PF forwards the reply to VF.
- From functionality point of view there is no difference between PF and VF as same type HW resources are attached to both. But user would be able to configure few stuff only from PF as PF is treated as owner/admin of the link.

Type2:

- RVU PF0 ie admin function creates these VFs and maps them to loopback block's channels.
- A set of two VFs (VF0 & VF1, VF2 & VF3 .. so on) works as a pair ie pkts sent out of VF0 will be received by VF1 and vice versa.
- These VFs can be used by applications or virtual machines to communicate between them without sending traffic outside. There is no switch present in HW, hence the support for loopback VFs.
- These communicate directly with AF (PF0) via mbox.

Except for the IO channels or links used for packet reception and transmission there is no other difference between these VF types. AF driver takes care of IO channel mapping, hence same VF driver works for both types of devices.

Basic packet flow

Ingress

1. CGX LMAC receives packet.
2. Forwards the packet to the NIX block.
3. Then submitted to NPC block for parsing and then MCAM lookup to get the destination RVU device.
4. NIX LF attached to the destination RVU device allocates a buffer from RQ mapped buffer pool of NPA block LF.
5. RQ may be selected by RSS or by configuring MCAM rule with a RQ number.
6. Packet is DMA'ed and driver is notified.

Egress

1. Driver prepares a send descriptor and submits to SQ for transmission.
2. The SQ is already configured (by AF) to transmit on a specific link/channel.
3. The SQ descriptor ring is maintained in buffers allocated from SQ mapped pool of NPA block LF.
4. NIX block transmits the pkt on the designated channel.
5. NPC MCAM entries can be installed to divert pkt onto a different channel.

Devlink health reporters

NPA Reporters

The NPA reporters are responsible for reporting and recovering the following group of errors:

1. GENERAL events
 - Error due to operation of unmapped PF.
 - Error due to disabled alloc/free for other HW blocks (NIX, SSO, TIM, DPI and AURA).
2. ERROR events
 - Fault due to NPA_AQ_INST_S read or NPA_AQ_RES_S write.
 - AQ Doorbell Error.
3. RAS events
 - RAS Error Reporting for NPA_AQ_INST_S/NPA_AQ_RES_S.

4. RVU events

- Error due to unmapped slot.

Sample Output:

```
~# devlink health
pci/0002:01:00.0:
  reporter hw_npa_intr
    state healthy error 2872 recover 2872 last_dump_date 2020-12-10 last_
    ↵dump_time 09:39:09 grace_period 0 auto_recover true auto_dump true
  reporter hw_npa_gen
    state healthy error 2872 recover 2872 last_dump_date 2020-12-11 last_
    ↵dump_time 04:43:04 grace_period 0 auto_recover true auto_dump true
  reporter hw_npa_err
    state healthy error 2871 recover 2871 last_dump_date 2020-12-10 last_
    ↵dump_time 09:39:17 grace_period 0 auto_recover true auto_dump true
  reporter hw_npa_ras
    state healthy error 0 recover 0 last_dump_date 2020-12-10 last_dump_time_
    ↵09:32:40 grace_period 0 auto_recover true auto_dump true
```

Each reporter dumps the

- Error Type
- Error Register value
- Reason in words

For example:

```
~# devlink health dump show pci/0002:01:00.0 reporter hw_npa_gen
NPA_AF_GENERAL:
  NPA General Interrupt Reg : 1
  NIX0: free disabled RX
~# devlink health dump show pci/0002:01:00.0 reporter hw_npa_intr
NPA_AF_RVU:
  NPA RVU Interrupt Reg : 1
  Unmap Slot Error
~# devlink health dump show pci/0002:01:00.0 reporter hw_npa_err
NPA_AF_ERR:
  NPA Error Interrupt Reg : 4096
  AQ Doorbell Error
```

NIX Reporters

The NIX reporters are responsible for reporting and recovering the following group of errors:

1. GENERAL events

- Receive mirror/multicast packet drop due to insufficient buffer.
- SMQ Flush operation.

2. ERROR events

- Memory Fault due to WQE read/write from multicast/mirror buffer.
- Receive multicast/mirror replication list error.
- Receive packet on an unmapped PF.
- Fault due to NIX_AQ_INST_S read or NIX_AQ_RES_S write.
- AQ Doorbell Error.

3. RAS events

- RAS Error Reporting for NIX Receive Multicast/Mirror Entry Structure.
- RAS Error Reporting for WQE/Packet Data read from Multicast/Mirror Buffer..
- RAS Error Reporting for NIX_AQ_INST_S/NIX_AQ_RES_S.

4. RVU events

- Error due to unmapped slot.

Sample Output:

```
~# ./devlink health
pci/0002:01:00.0:
reporter hw_npa_intr
  state healthy error 0 recover 0 grace_period 0 auto_recover true auto_dumpu
✓true
reporter hw_npa_gen
  state healthy error 0 recover 0 grace_period 0 auto_recover true auto_dumpu
✓true
reporter hw_npa_err
  state healthy error 0 recover 0 grace_period 0 auto_recover true auto_dumpu
✓true
reporter hw_npa_ras
  state healthy error 0 recover 0 grace_period 0 auto_recover true auto_dumpu
✓true
reporter hw_nix_intr
  state healthy error 1121 recover 1121 last_dump_date 2021-01-19 last_dump_
✓time 05:42:26 grace_period 0 auto_recover true auto_dump true
reporter hw_nix_gen
  state healthy error 949 recover 949 last_dump_date 2021-01-19 last_dump_
✓time 05:42:43 grace_period 0 auto_recover true auto_dump true
reporter hw_nix_err
  state healthy error 1147 recover 1147 last_dump_date 2021-01-19 last_dump_
✓time 05:42:59 grace_period 0 auto_recover true auto_dump true
reporter hw_nix_ras
  state healthy error 409 recover 409 last_dump_date 2021-01-19 last_dump_
✓time 05:43:16 grace_period 0 auto_recover true auto_dump true
```

Each reporter dumps the

- Error Type
- Error Register value
- Reason in words

For example:

```
~# devlink health dump show pci/0002:01:00.0 reporter hw_nix_intr
NIX_AF_RVU:
    NIX RVU Interrupt Reg : 1
    Unmap Slot Error
~# devlink health dump show pci/0002:01:00.0 reporter hw_nix_gen
NIX_AF_GENERAL:
    NIX General Interrupt Reg : 1
    Rx multicast pkt drop
~# devlink health dump show pci/0002:01:00.0 reporter hw_nix_err
NIX_AF_ERR:
    NIX Error Interrupt Reg : 64
    Rx on unmapped PF_FUNC
```

Quality of service

Hardware algorithms used in scheduling

octeontx2 silicon and CN10K transmit interface consists of five transmit levels starting from SMQ/MDQ, TL4 to TL1. Each packet will traverse MDQ, TL4 to TL1 levels. Each level contains an array of queues to support scheduling and shaping. The hardware uses the below algorithms depending on the priority of scheduler queues. once the user creates tc classes with different priorities, the driver configures schedulers allocated to the class with specified priority along with rate-limiting configuration.

1. Strict Priority

- Once packets are submitted to MDQ, hardware picks all active MDQs having different priority using strict priority.

2. Round Robin

- Active MDQs having the same priority level are chosen using round robin.

Setup HTB offload

1. Enable HW TC offload on the interface:

```
# ethtool -K <interface> hw-tc-offload on
```

2. Create htb root:

```
# tc qdisc add dev <interface> clsact
# tc qdisc replace dev <interface> root handle 1: htb offload
```

3. Create tc classes with different priorities:

```
# tc class add dev <interface> parent 1: classid 1:1 htb rate 10Gbit prio 1
# tc class add dev <interface> parent 1: classid 1:2 htb rate 10Gbit prio 7
```

4. Create tc classes with same priorities and different quantum:

```
# tc class add dev <interface> parent 1: classid 1:1 htb rate 10Gbit prio
↪2 quantum 409600

# tc class add dev <interface> parent 1: classid 1:2 htb rate 10Gbit prio
↪2 quantum 188416

# tc class add dev <interface> parent 1: classid 1:3 htb rate 10Gbit prio
↪2 quantum 32768
```

6.5.32 Linux kernel networking driver for Marvell's Octeon PCI Endpoint NIC

Network driver for Marvell's Octeon PCI EndPoint NIC. Copyright (c) 2020 Marvell International Ltd.

Contents

- *Overview*
- *Supported Devices*
- *Interface Control*

Overview

This driver implements networking functionality of Marvell's Octeon PCI EndPoint NIC.

Supported Devices

Currently, this driver support following devices:

- Network controller: Cavium, Inc. Device b100
- Network controller: Cavium, Inc. Device b200
- Network controller: Cavium, Inc. Device b400
- Network controller: Cavium, Inc. Device b900
- Network controller: Cavium, Inc. Device ba00
- Network controller: Cavium, Inc. Device bc00
- Network controller: Cavium, Inc. Device bd00

Interface Control

Network Interface control like changing mtu, link speed, link down/up are done by writing command to mailbox command queue, a mailbox interface implemented through a reserved region in BAR4. This driver writes the commands into the mailbox and the firmware on the Octeon device processes them. The firmware also sends unsolicited notifications to driver for events suchs as link change, through notification queue implemented as part of mailbox interface.

6.5.33 Mellanox ConnectX(R) mlx5 core VPI Network Driver

Copyright

© 2019, Mellanox Technologies LTD.

Copyright

© 2020-2023, NVIDIA CORPORATION & AFFILIATES. All rights reserved.

Contents:

Enabling the driver and kconfig options

Copyright

© 2023, NVIDIA CORPORATION & AFFILIATES. All rights reserved.

mlx5 core is modular and most of the major mlx5 core driver features can be selected (compiled in/out)

at build time via kernel Kconfig flags.

Basic features, ethernet net device rx/tx offloads and XDP, are available with the most basic flags

CONFIG_MLX5_CORE=y/m and CONFIG_MLX5_CORE_EN=y.

For the list of advanced features, please see below.

CONFIG_MLX5_BRIDGE=(y/n)

Enable *Ethernet Bridging (BRIDGE) offloading support*.

This will provide the ability to add representors of mlx5 uplink and VF ports to Bridge and offloading rules for traffic between such ports.

Supports VLANs (trunk and access modes).

CONFIG_MLX5_CORE=(y/m/n) (module mlx5_core.ko)

The driver can be enabled by choosing CONFIG_MLX5_CORE=y/m in kernel config.

This will provide mlx5 core driver for mlx5 ulps to interface with (mlx5e, mlx5_ib).

CONFIG_MLX5_CORE_EN=(y/n)

Choosing this option will allow basic ethernet netdevice support with all of the standard rx/tx offloads.

mlx5e is the mlx5 ulp driver which provides netdevice kernel interface, when chosen, mlx5e will be built-in into mlx5_core.ko.

CONFIG_MLX5_CORE_EN_DCB=(y/n):

Enables Data Center Bridging (DCB) Support.

CONFIG_MLX5_CORE_IPOIB=(y/n)

IPoIB offloads & acceleration support.

Requires CONFIG_MLX5_CORE_EN to provide an accelerated interface for the rdma IPoIB ulp netdevice.

CONFIG_MLX5_CLS_ACT=(y/n)

Enables offload support for TC classifier action (NET_CLS_ACT).

Works in both native NIC mode and Switchdev SRIOV mode.

Flow-based classifiers, such as those registered through *tc-flower(8)*, are processed by the device, rather than the host. Actions that would then overwrite matching classification results would then be instant due to the offload.

CONFIG_MLX5_EN_ARFS=(y/n)

Enables Hardware-accelerated receive flow steering (arfs) support, and ntuple filtering.

<https://enterprise-support.nvidia.com/s/article/howto-configure-arfs-on-connectx-4>

CONFIG_MLX5_EN_IPSEC=(y/n)

Enables *IPSec XFRM cryptography-offload acceleration*.

CONFIG_MLX5_MACSEC=(y/n)

Build support for MACsec cryptography-offload acceleration in the NIC.

CONFIG_MLX5_EN_RXNFC=(y/n)

Enables ethtool receive network flow classification, which allows user defined flow rules to direct traffic into arbitrary rx queue via ethtool set/get_rxnfc API.

CONFIG_MLX5_EN_TLS=(y/n)

TLS cryptography-offload acceleration.

CONFIG_MLX5_ESWITCH=(y/n)

Ethernet SRIOV E-Switch support in ConnectX NIC. E-Switch provides internal SRIOV packet steering

and switching for the enabled VFs and PF in two available modes:

- 1) Legacy SRIOV mode (L2 mac vlan steering based).
- 2) *Switchdev mode (eswitch offloads)*.

CONFIG_MLX5_FPGA=(y/n)

Build support for the Innova family of network cards by Mellanox Technologies.

Innova network cards are comprised of a ConnectX chip and an FPGA chip on one board.

If you select this option, the mlx5_core driver will include the Innova FPGA core and allow building sandbox-specific client drivers.

CONFIG_MLX5_INFINIBAND=(y/n/m) (module mlx5_ib.ko)

Provides low-level InfiniBand/RDMA and RoCE support.

CONFIG_MLX5_MPFS=(y/n)

Ethernet Multi-Physical Function Switch (MPFS) support in ConnectX NIC.

MPFs is required for when Multi-Host configuration is enabled to allow passing user configured unicast MAC addresses to the requesting PF.

CONFIG_MLX5_SF=(y/n)

Build support for subfunction.

Subfunctions are more light weight than PCI SRIOV VFs. Choosing this option will enable support for creating subfunction devices.

CONFIG_MLX5_SF_MANAGER=(y/n)

Build support for subfunction port in the NIC. A Mellanox subfunction port is managed through devlink. A subfunction supports RDMA, netdevice and vdpa device. It is similar to a SRIOV VF but it doesn't require SRIOV support.

CONFIG_MLX5_SW_STEERING=(y/n)

Build support for software-managed steering in the NIC.

CONFIG_MLX5_TC_CT=(y/n)

Support offloading connection tracking rules via tc ct action.

CONFIG_MLX5_TC_SAMPLE=(y/n)

Support offloading sample rules via tc sample action.

CONFIG_MLX5_VDPA=(y/n)

Support library for Mellanox VDPA drivers. Provides code that is common for all types of VDPA drivers. The following drivers are planned: net, block.

CONFIG_MLX5_VDPA_NET=(y/n)

VDPA network driver for ConnectX6 and newer. Provides offloading of virtio net datapath such that descriptors put on the ring will be executed by the hardware. It also supports a variety of stateless offloads depending on the actual device used and firmware version.

CONFIG_MLX5_VFIO_PCI=(y/n)

This provides migration support for MLX5 devices using the VFIO framework.

External options (Choose if the corresponding mlx5 feature is required)

- CONFIG_MLXFW: When chosen, mlx5 firmware flashing support will be enabled (via devlink and ethtool).
- CONFIG_PTP_1588_CLOCK: When chosen, mlx5 ptp support will be enabled
- CONFIG_VXLAN: When chosen, mlx5 vxlan support will be enabled.

Switchdev

Copyright

© 2023, NVIDIA CORPORATION & AFFILIATES. All rights reserved.

Bridge offload

The mlx5 driver implements support for offloading bridge rules when in switchdev mode. Linux bridge FDBs are automatically offloaded when mlx5 switchdev representor is attached to bridge.

- Change device to switchdev mode:

```
$ devlink dev eswitch set pci/0000:06:00.0 mode switchdev
```

- Attach mlx5 switchdev representor 'enp8s0f0' to bridge netdev 'bridge1':

```
$ ip link set enp8s0f0 master bridge1
```

VLANs

Following bridge VLAN functions are supported by mlx5:

- VLAN filtering (including multiple VLANs per port):

```
$ ip link set bridge1 type bridge vlan_filtering 1
$ bridge vlan add dev enp8s0f0 vid 2-3
```

- VLAN push on bridge ingress:

```
$ bridge vlan add dev enp8s0f0 vid 3 pvid
```

- VLAN pop on bridge egress:

```
$ bridge vlan add dev enp8s0f0 vid 3 untagged
```

Subfunction

Subfunction which are spawned over the E-switch are created only with devlink device, and by default all the SF auxiliary devices are disabled. This will allow user to configure the SF before the SF have been fully probed, which will save time.

Usage example:

- Create SF:

```
$ devlink port add pci/0000:08:00.0 flavour pcisf pfnum 0 sfnum 11
$ devlink port function set pci/0000:08:00.0/32768 hw_addr ↴
  ↪00:00:00:00:00:11 state active
```

- Enable ETH auxiliary device:

```
$ devlink dev param set auxiliary/mlx5_core.sf.1 name enable_eth value ↴
  ↪true cmode driverinit
```

- Now, in order to fully probe the SF, use devlink reload:

```
$ devlink dev reload auxiliary/mlx5_core.sf.1
```

mlx5 supports ETH,rdma and vdpa (vnet) auxiliary devices devlink params (see [Documentation/networking/devlink/devlink-params.rst](#)).

mlx5 supports subfunction management using devlink port (see [Documentation/networking/devlink/devlink-port.rst](#)) interface.

A subfunction has its own function capabilities and its own resources. This means a subfunction has its own dedicated queues (txq, rxq, cq, eq). These queues are neither shared nor stolen from the parent PCI function.

When a subfunction is RDMA capable, it has its own QP1, GID table, and RDMA resources neither shared nor stolen from the parent PCI function.

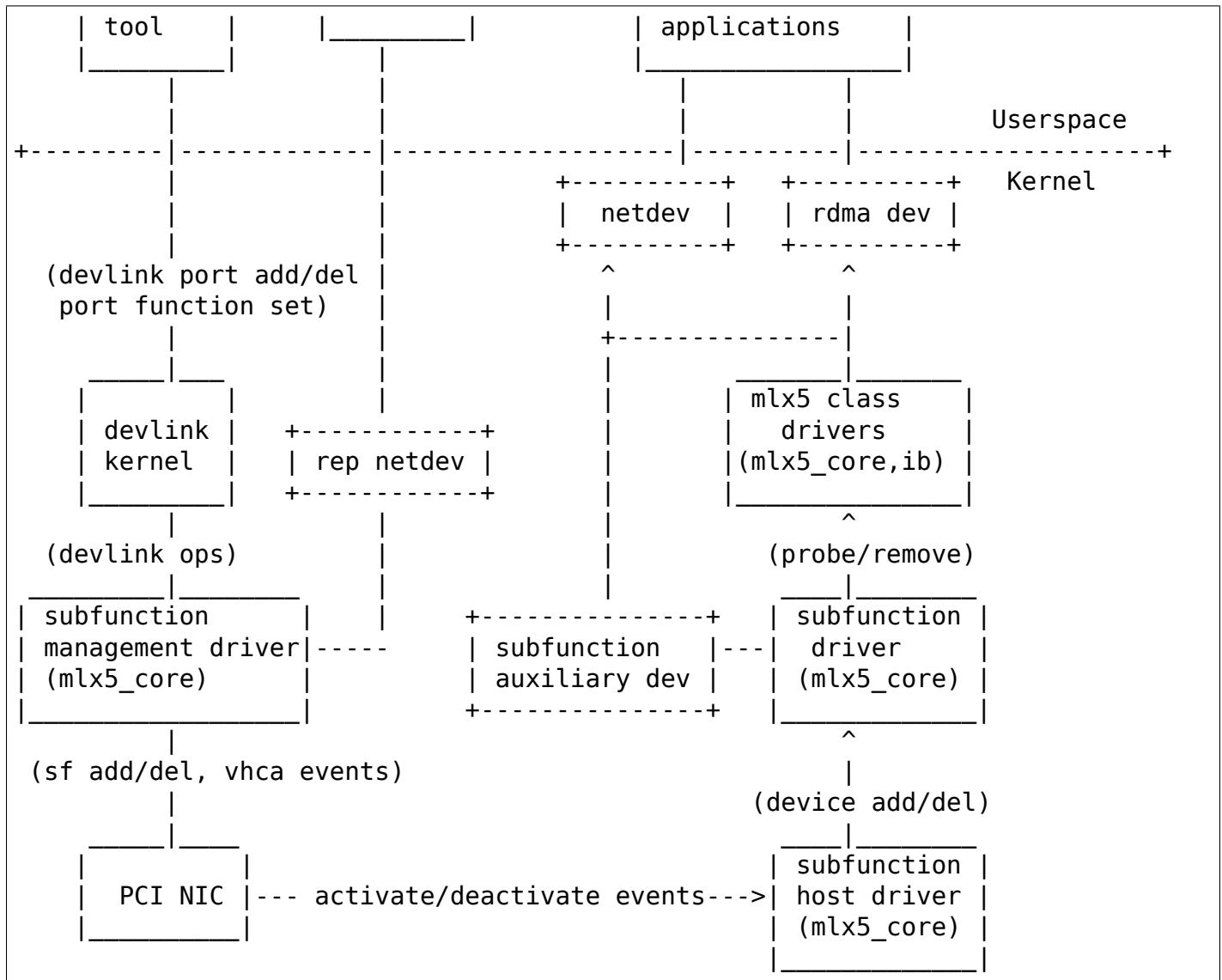
A subfunction has a dedicated window in PCI BAR space that is not shared with the other subfunctions or the parent PCI function. This ensures that all devices (netdev, rdma, vdpa, etc.) of the subfunction accesses only assigned PCI BAR space.

A subfunction supports eswitch representation through which it supports tc offloads. The user configures eswitch to send/receive packets from/to the subfunction port.

Subfunctions share PCI level resources such as PCI MSI-X IRQs with other subfunctions and/or with its parent PCI function.

Example mlx5 software, system, and device view:





Subfunction is created using devlink port interface.

- Change device to switchdev mode:

```
$ devlink dev eswitch set pci/0000:06:00.0 mode switchdev
```

- Add a devlink port of subfunction flavour:

```
$ devlink port add pci/0000:06:00.0 flavour pcisf pfnum 0 sfnum 88
pci/0000:06:00.0/32768: type eth netdev eth6 flavour pcisf controller 0
  ↳ pfnum 0 sfnum 88 external false splittable false
    function:
      hw_addr 00:00:00:00:00:00 state inactive opstate detached
```

- Show a devlink port of the subfunction:

```
$ devlink port show pci/0000:06:00.0/32768
pci/0000:06:00.0/32768: type eth netdev enp6s0pf0sf88 flavour pcisf pfnum
 ↗0 sfnum 88
    function:
```

```
hw_addr 00:00:00:00:00:00 state inactive opstate detached
```

- Delete a devlink port of subfunction after use:

```
$ devlink port del pci/0000:06:00.0/32768
```

Function attributes

The mlx5 driver provides a mechanism to setup PCI VF/SF function attributes in a unified way for SmartNIC and non-SmartNIC.

This is supported only when the eswitch mode is set to switchdev. Port function configuration of the PCI VF/SF is supported through devlink eswitch port.

Port function attributes should be set before PCI VF/SF is enumerated by the driver.

MAC address setup

mlx5 driver support devlink port function attr mechanism to setup MAC address. (refer to [Devlink Port](#))

RoCE capability setup

Not all mlx5 PCI devices/SFs require RoCE capability.

When RoCE capability is disabled, it saves 1 Mbytes worth of system memory per PCI devices/SF.

mlx5 driver support devlink port function attr mechanism to setup RoCE capability. (refer to [Devlink Port](#))

migratable capability setup

User who wants mlx5 PCI VFs to be able to perform live migration need to explicitly enable the VF migratable capability.

mlx5 driver support devlink port function attr mechanism to setup migratable capability. (refer to [Devlink Port](#))

IPsec crypto capability setup

User who wants mlx5 PCI VFs to be able to perform IPsec crypto offloading need to explicitly enable the VF ipsec_crypto capability. Enabling IPsec capability for VFs is supported starting with ConnectX6dx devices and above. When a VF has IPsec capability enabled, any IPsec offloading is blocked on the PF.

mlx5 driver support devlink port function attr mechanism to setup ipsec_crypto capability. (refer to [Devlink Port](#))

IPsec packet capability setup

User who wants mlx5 PCI VFs to be able to perform IPsec packet offloading need to explicitly enable the VF ipsec_packet capability. Enabling IPsec capability for VFs is supported starting with ConnectX6dx devices and above. When a VF has IPsec capability enabled, any IPsec offloading is blocked on the PF.

mlx5 driver support devlink port function attr mechanism to setup ipsec_packet capability. (refer to [Devlink Port](#))

SF state setup

To use the SF, the user must activate the SF using the SF function state attribute.

- Get the state of the SF identified by its unique devlink port index:

```
$ devlink port show ens2f0npf0sf88
pci/0000:06:00.0/32768: type eth netdev ens2f0npf0sf88 flavour pcisf
  ↪controller 0 pfnum 0 sfnum 88 external false splittable false
    function:
      hw_addr 00:00:00:00:88:88 state inactive opstate detached
```

- Activate the function and verify its state is active:

```
$ devlink port function set ens2f0npf0sf88 state active

$ devlink port show ens2f0npf0sf88
pci/0000:06:00.0/32768: type eth netdev ens2f0npf0sf88 flavour pcisf
  ↪controller 0 pfnum 0 sfnum 88 external false splittable false
    function:
      hw_addr 00:00:00:00:88:88 state active opstate detached
```

Upon function activation, the PF driver instance gets the event from the device that a particular SF was activated. It's the cue to put the device on bus, probe it and instantiate the devlink instance and class specific auxiliary devices for it.

- Show the auxiliary device and port of the subfunction:

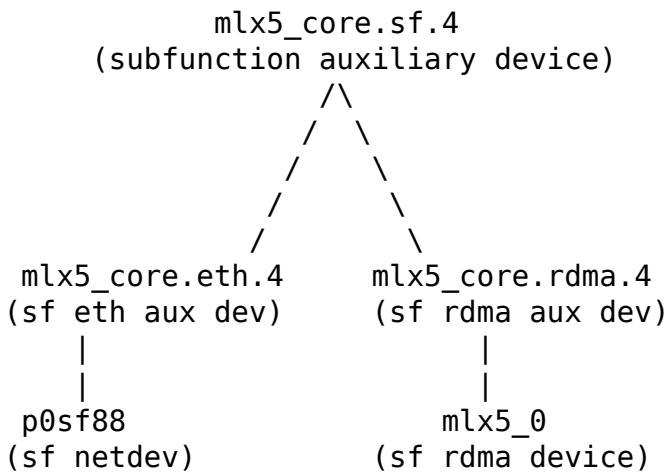
```
$ devlink dev show
devlink dev show auxiliary/mlx5_core.sf.4

$ devlink port show auxiliary/mlx5_core.sf.4/1
auxiliary/mlx5_core.sf.4/1: type eth netdev p0sf88 flavour virtual port 0
  ↪splittable false

$ rdma link show mlx5_0/1
link mlx5_0/1 state ACTIVE physical_state LINK_UP netdev p0sf88

$ rdma dev show
8: rocep6s0f1: node_type ca fw 16.29.0550 node_guid 248a:0703:00b3:d113
  ↪sys_image_guid 248a:0703:00b3:d112
13: mlx5_0: node_type ca fw 16.29.0550 node_guid 0000:00ff:fe00:8888 sys_
  ↪image_guid 248a:0703:00b3:d112
```

- Subfunction auxiliary device and class device hierarchy:



Additionally, the SF port also gets the event when the driver attaches to the auxiliary device of the subfunction. This results in changing the operational state of the function. This provides visibility to the user to decide when is it safe to delete the SF port for graceful termination of the subfunction.

- Show the SF port operational state:

```
$ devlink port show ens2f0npf0sf88
pci/0000:06:00.0/32768: type eth netdev ens2f0npf0sf88 flavour pcisf
  ↳ controller 0 pfnum 0 sfnum 88 external false splittable false
    function:
      hw_addr 00:00:00:00:88:88 state active opstate attached
```

Tracepoints

Copyright

© 2023, NVIDIA CORPORATION & AFFILIATES. All rights reserved.

mlx5 driver provides internal tracepoints for tracking and debugging using kernel tracepoints interfaces (refer to Documentation/trace/ftrace.rst).

For the list of support mlx5 events, check /sys/kernel/tracing/events/mlx5/.

tc and eswitch offloads tracepoints:

- mlx5e_configure_flower: trace flower filter actions and cookies offloaded to mlx5:

```
$ echo mlx5:mlx5e_configure_flower >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
tc-6535 [019] ...1 2672.404466: mlx5e_configure_flower: ↳
  ↳ cookie=0000000067874a55 actions= REDIRECT
```

- mlx5e_delete_flower: trace flower filter actions and cookies deleted from mlx5:

```
$ echo mlx5:mlx5e_delete_flower >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
tc-6569 [010] .N.1 2686.379075: mlx5e_delete_flower:_
↪cookie=0000000067874a55 actions= NULL
```

- mlx5e_stats_flower: trace flower stats request:

```
$ echo mlx5:mlx5e_stats_flower >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
tc-6546 [010] ...1 2679.704889: mlx5e_stats_flower:_
↪cookie=0000000060eb3d6a bytes=0 packets=0 lastused=4295560217
```

- mlx5e_tc_update_neigh_used_value: trace tunnel rule neigh update value offloaded to mlx5:

```
$ echo mlx5:mlx5e_tc_update_neigh_used_value >> /sys/kernel/tracing/set_
↪event
$ cat /sys/kernel/tracing/trace
...
kworker/u48:4-8806 [009] ...1 55117.882428: mlx5e_tc_update_neigh_used_
↪value: netdev: ens1f0 IPv4: 1.1.1.10 IPv6: ::ffff:1.1.1.10 neigh_used=1
```

- mlx5e_rep_neigh_update: trace neigh update tasks scheduled due to neigh state change events:

```
$ echo mlx5:mlx5e_rep_neigh_update >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
kworker/u48:7-2221 [009] ...1 1475.387435: mlx5e_rep_neigh_update:_
↪netdev: ens1f0 MAC: 24:8a:07:9a:17:9a IPv4: 1.1.1.10 IPv6: ::ffff:1.1.1.
↪10 neigh_connected=1
```

Bridge offloads tracepoints:

- mlx5_esw_bridge_fdb_entry_init: trace bridge FDB entry offloaded to mlx5:

```
$ echo mlx5:mlx5_esw_bridge_fdb_entry_init >> set_event
$ cat /sys/kernel/tracing/trace
...
kworker/u20:9-2217 [003] ...1 318.582243: mlx5_esw_bridge_fdb_entry_
↪init: net_device=enp8s0f0_0 addr=e4:fd:05:08:00:02 vid=0 flags=0 used=0
```

- mlx5_esw_bridge_fdb_entry_cleanup: trace bridge FDB entry deleted from mlx5:

```
$ echo mlx5:mlx5_esw_bridge_fdb_entry_cleanup >> set_event
$ cat /sys/kernel/tracing/trace
...
ip-2581 [005] ...1 318.629871: mlx5_esw_bridge_fdb_entry_cleanup: net_
↪device=enp8s0f0_1 addr=e4:fd:05:08:00:03 vid=0 flags=0 used=16
```

- mlx5_esw_bridge_fdb_entry_refresh: trace bridge FDB entry offload refreshed in mlx5:

```
$ echo mlx5:mlx5_esw_bridge_fdb_entry_refresh >> set_event
$ cat /sys/kernel/tracing/trace
...
kworker/u20:8-3849 [003] ...1 466716: mlx5_esw_bridge_fdb_entry_
refresh: net_device=enp8s0f0_0 addr=e4:fd:05:08:00:02 vid=3 flags=0
used=0
```

- mlx5_esw_bridge_vlan_create: trace bridge VLAN object add on mlx5 representor:

```
$ echo mlx5:mlx5_esw_bridge_vlan_create >> set_event
$ cat /sys/kernel/tracing/trace
...
ip-2560 [007] ...1 318.460258: mlx5_esw_bridge_vlan_create: vid=1
flags=6
```

- mlx5_esw_bridge_vlan_cleanup: trace bridge VLAN object delete from mlx5 representor:

```
$ echo mlx5:mlx5_esw_bridge_vlan_cleanup >> set_event
$ cat /sys/kernel/tracing/trace
...
bridge-2582 [007] ...1 318.653496: mlx5_esw_bridge_vlan_cleanup:
vid=2 flags=8
```

- mlx5_esw_bridge_vport_init: trace mlx5 vport assigned with bridge upper device:

```
$ echo mlx5:mlx5_esw_bridge_vport_init >> set_event
$ cat /sys/kernel/tracing/trace
...
ip-2560 [007] ...1 318.458915: mlx5_esw_bridge_vport_init: vport_num=1
```

- mlx5_esw_bridge_vport_cleanup: trace mlx5 vport removed from bridge upper device:

```
$ echo mlx5:mlx5_esw_bridge_vport_cleanup >> set_event
$ cat /sys/kernel/tracing/trace
...
ip-5387 [000] ...1 573713: mlx5_esw_bridge_vport_cleanup: vport_
num=1
```

Eswitch QoS tracepoints:

- mlx5_esw_vport_qos_create: trace creation of transmit scheduler arbiter for vport:

```
$ echo mlx5:mlx5_esw_vport_qos_create >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
<...>-23496 [018] .... 73136.838831: mlx5_esw_vport_qos_create:
(0000:82:00.0) vport=2 tsar_ix=4 bw_share=0, max_rate=0
group=00000007b576bb3
```

- mlx5_esw_vport_qos_config: trace configuration of transmit scheduler arbiter for vport:

```
$ echo mlx5:mlx5_esw_vport_qos_config >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
<...>-26548 [023] .... 75754.223823: mlx5_esw_vport_qos_config: u
  ↳(0000:82:00.0) vport=1 tsar_ix=3 bw_share=34, max_rate=10000 u
  ↳group=000000007b576bb3
```

- mlx5_esw_vport_qos_destroy: trace deletion of transmit scheduler arbiter for vport:

```
$ echo mlx5:mlx5_esw_vport_qos_destroy >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
<...>-27418 [004] .... 76546.680901: mlx5_esw_vport_qos_destroy: u
  ↳(0000:82:00.0) vport=1 tsar_ix=3
```

- mlx5_esw_group_qos_create: trace creation of transmit scheduler arbiter for rate group:

```
$ echo mlx5:mlx5_esw_group_qos_create >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
<...>-26578 [008] .... 75776.022112: mlx5_esw_group_qos_create: u
  ↳(0000:82:00.0) group=000000008dac63ea tsar_ix=5
```

- mlx5_esw_group_qos_config: trace configuration of transmit scheduler arbiter for rate group:

```
$ echo mlx5:mlx5_esw_group_qos_config >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
<...>-27303 [020] .... 76461.455356: mlx5_esw_group_qos_config: u
  ↳(0000:82:00.0) group=000000008dac63ea tsar_ix=5 bw_share=100 max_
  ↳rate=20000
```

- mlx5_esw_group_qos_destroy: trace deletion of transmit scheduler arbiter for group:

```
$ echo mlx5:mlx5_esw_group_qos_destroy >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
<...>-27418 [006] .... 76547.187258: mlx5_esw_group_qos_destroy: u
  ↳(0000:82:00.0) group=000000007b576bb3 tsar_ix=1
```

SF tracepoints:

- mlx5_sf_add: trace addition of the SF port:

```
$ echo mlx5:mlx5_sf_add >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
devlink-9363 [031] ..... 24610.188722: mlx5_sf_add: (0000:06:00.0) port_
  ↳index=32768 controller=0 hw_id=0x8000 sfnum=88
```

- mlx5_sf_free: trace freeing of the SF port:

```
$ echo mlx5:mlx5_sf_free >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
devlink-9830      [038] ..... 26300.404749: mlx5_sf_free: (0000:06:00.0) ↳
    ↪port_index=32768 controller=0 hw_id=0x8000
```

- mlx5_sf_activate: trace activation of the SF port:

```
$ echo mlx5:mlx5_sf_activate >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
devlink-29841      [008] ..... 3669.635095: mlx5_sf_activate: (0000:08:00.0) ↳
    ↪port_index=32768 controller=0 hw_id=0x8000
```

- mlx5_sf_deactivate: trace deactivation of the SF port:

```
$ echo mlx5:mlx5_sf_deactivate >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
devlink-29994      [008] ..... 4015.969467: mlx5_sf_deactivate: (0000:08:00.0) ↳
    ↪port_index=32768 controller=0 hw_id=0x8000
```

- mlx5_sf_hwc_alloc: trace allocating of the hardware SF context:

```
$ echo mlx5:mlx5_sf_hwc_alloc >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
devlink-9775      [031] ..... 26296.385259: mlx5_sf_hwc_alloc: (0000:06:00.0) ↳
    ↪controller=0 hw_id=0x8000 sfnum=88
```

- mlx5_sf_hwc_free: trace freeing of the hardware SF context:

```
$ echo mlx5:mlx5_sf_hwc_free >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
kworker/u128:3-9093      [046] ..... 24625.365771: mlx5_sf_hwc_free: (0000:06:00.0) ↳
    ↪hw_id=0x8000
```

- mlx5_sf_hwc_deferred_free: trace deferred freeing of the hardware SF context:

```
$ echo mlx5:mlx5_sf_hwc_deferred_free >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
devlink-9519      [046] ..... 24624.400271: mlx5_sf_hwc_deferred_free: (0000:06:00.0) ↳
    ↪hw_id=0x8000
```

- mlx5_sf_update_state: trace state updates for SF contexts:

```
$ echo mlx5:mlx5_sf_update_state >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
kworker/u20:3-29490      [009] ..... 4141.453530: mlx5_sf_update_state: (0000:08:00.0) ↳
    ↪port_index=32768 controller=0 hw_id=0x8000 state=2
```

- mlx5_sf_vhca_event: trace SF vhca event and state:

```
$ echo mlx5:mlx5_sf_vhca_event >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
kworker/u128:3-9093      [046] ..... 24625.365525: mlx5_sf_vhca_event:_
↪(0000:06:00.0) hw_id=0x8000 sfnum=88 vhca_state=1
```

- mlx5_sf_dev_add: trace SF device add event:

```
$ echo mlx5:mlx5_sf_dev_add>> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
kworker/u128:3-9093      [000] ..... 24616.524495: mlx5_sf_dev_add:_
↪(0000:06:00.0) sfdev=00000000fc5d96fd aux_id=4 hw_id=0x8000 sfnum=88
```

- mlx5_sf_dev_del: trace SF device delete event:

```
$ echo mlx5:mlx5_sf_dev_del >> /sys/kernel/tracing/set_event
$ cat /sys/kernel/tracing/trace
...
kworker/u128:3-9093      [044] ..... 24624.400749: mlx5_sf_dev_del:_
↪(0000:06:00.0) sfdev=00000000fc5d96fd aux_id=4 hw_id=0x8000 sfnum=88
```

Ethtool counters

Copyright

© 2023, NVIDIA CORPORATION & AFFILIATES. All rights reserved.

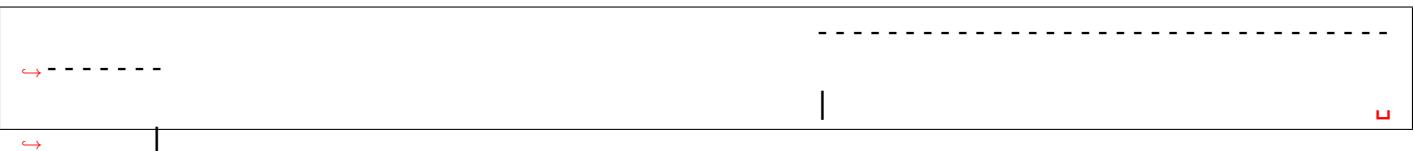
Contents

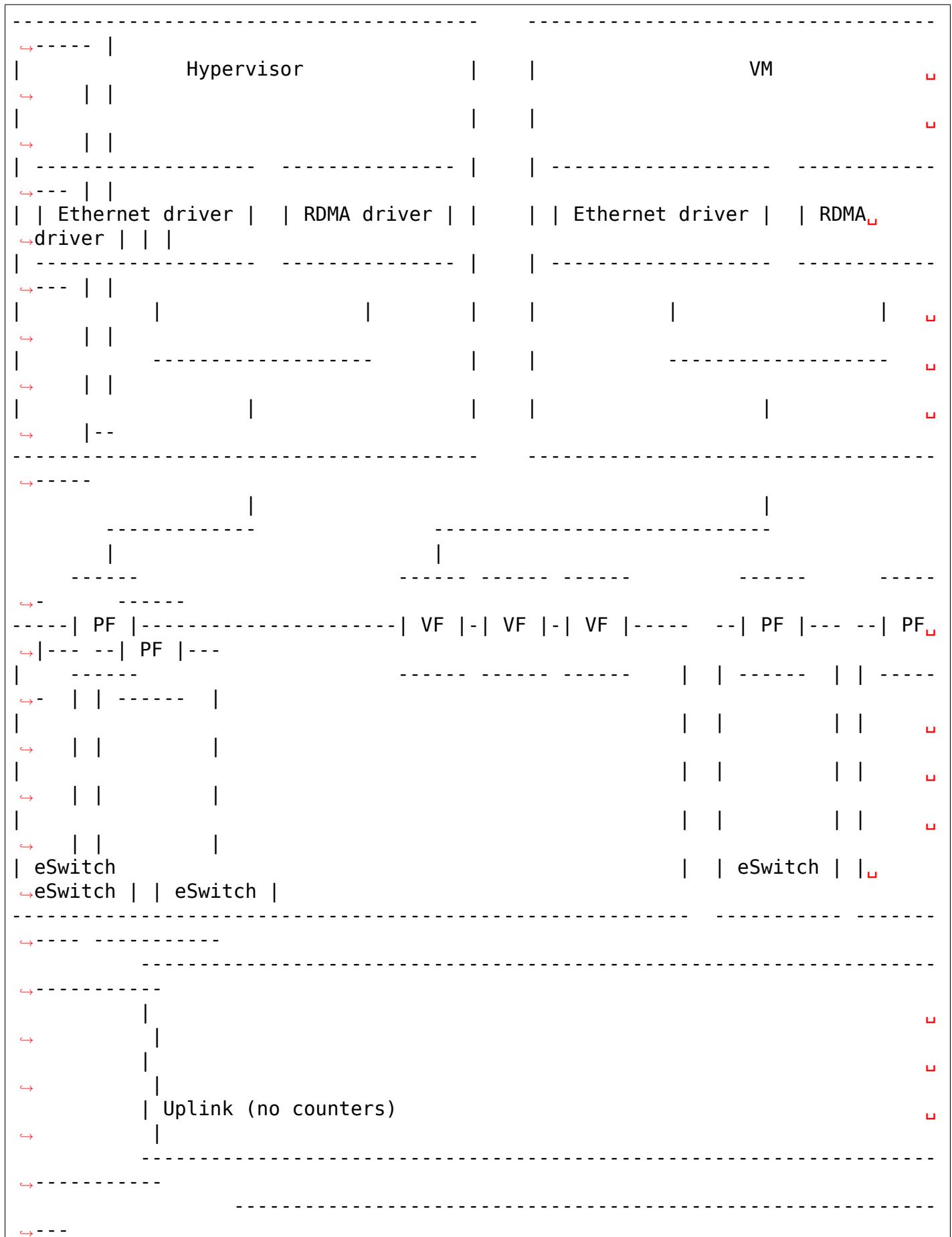
- *Overview*
- *Groups*
- *Types*
- *Descriptions*

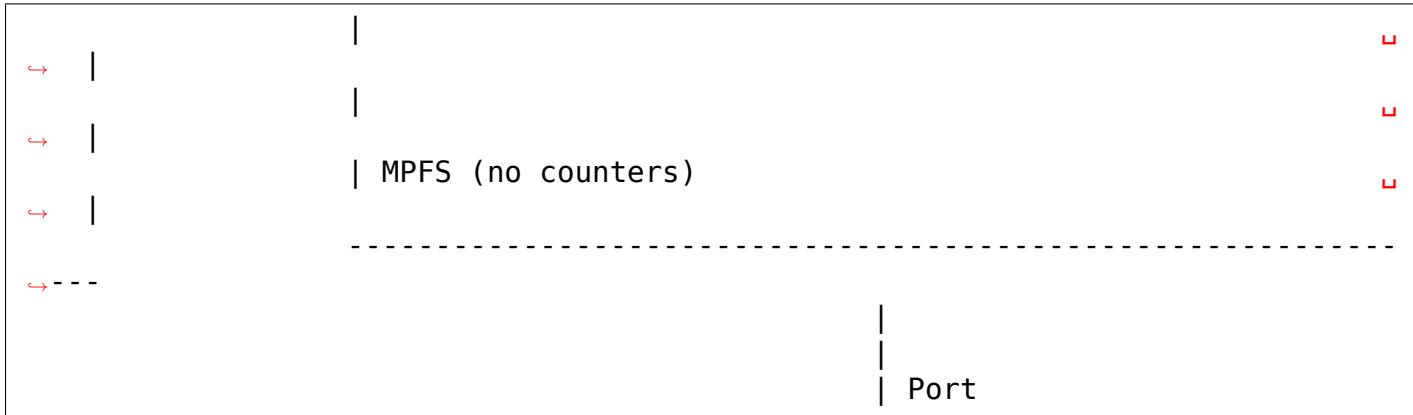
Overview

There are several counter groups based on where the counter is being counted. In addition, each group of counters may have different counter types.

These counter groups are based on which component in a networking setup, illustrated below, that they describe:







Groups

Ring

Software counters populated by the driver stack.

Netdev

An aggregation of software ring counters.

vPort counters

Traffic counters and drops due to steering or no buffers. May indicate issues with NIC. These counters include Ethernet traffic counters (including Raw Ethernet) and RDMA/RoCE traffic counters.

Physical port counters

Counters that collect statistics about the PFs and VFs. May indicate issues with NIC, link, or network. This measuring point holds information on standardized counters like IEEE 802.3, RFC2863, RFC 2819, RFC 3635 and additional counters like flow control, FEC and more. Physical port counters are not exposed to virtual machines.

Priority Port Counters

A set of the physical port counters, per priority per port.

Types

Counters are divided into three types.

Traffic Informative Counters

Counters which count traffic. These counters can be used for load estimation or for general debug.

Traffic Acceleration Counters

Counters which count traffic that was accelerated by Mellanox driver or by hardware. The counters are an additional layer to the informative counter set, and the same traffic is counted in both informative and acceleration counters.

Error Counters

Increment of these counters might indicate a problem. Each of these counters has an explanation and correction action.

Statistic can be fetched via the *ip link* or *ethtool* commands. *ethtool* provides more detailed information.:

```
ip -s link show <if-name>
ethtool -S <if-name>
```

Descriptions

XSK, PTP, and QoS counters that are similar to counters defined previously will not be separately listed. For example, *ptp_tx[i].packets* will not be explicitly documented since *tx[i].packets* describes the behavior of both counters, except *ptp_tx[i].packets* is only counted when precision time protocol is used.

Ring / Netdev Counter

The following counters are available per ring or software port.

These counters provide information on the amount of traffic that was accelerated by the NIC. The counters are counting the accelerated traffic in addition to the standard counters which counts it (i.e. accelerated traffic is counted twice).

The counter names in the table below refers to both ring and port counters. The notation for ring counters includes the [i] index without the braces. The notation for port counters doesn't include the [i]. A counter name *rx[i].packets* will be printed as *rx0.packets* for ring 0 and *rx_packets* for the software port.

Table 3: Ring / Software Port Counter Table

Counter	Description	Type
<i>rx[i].packets</i>	The number of packets received on ring i.	Informative
<i>rx[i].bytes</i>	The number of bytes received on ring i.	Informative
<i>tx[i].packets</i>	The number of packets transmitted on ring i.	Informative
<i>tx[i].bytes</i>	The number of bytes transmitted on ring i.	Informative
<i>tx[i].recover</i>	The number of times the SQ was recovered.	Error
<i>tx[i].cqes</i>	Number of CQEs events on SQ issued on ring i.	Informative
<i>tx[i].cqe_err</i>	The number of error CQEs encountered on the SQ for ring i.	Error
<i>tx[i].tso_packets</i>	The number of TSO packets transmitted on ring i ¹ .	Acceleration
<i>tx[i].tso_bytes</i>	The number of TSO bytes transmitted on ring i ^{Page 311, 1} .	Acceleration

continues on next page

Table 3 – continued from previous page

<i>tx[i].tso_inner_packets</i>	The number of TSO packets which are indicated to be carry internal encapsulation transmitted on ring i Page 311, 1 .	Acceleration
<i>tx[i].tso_inner_bytes</i>	The number of TSO bytes which are indicated to be carry internal encapsulation transmitted on ring i Page 311, 1 .	Acceleration
<i>rx[i].gro_packets</i>	Number of received packets processed using hardware-accelerated GRO. The number of hardware GRO offloaded packets received on ring i.	Acceleration
<i>rx[i].gro_bytes</i>	Number of received bytes processed using hardware-accelerated GRO. The number of hardware GRO offloaded bytes received on ring i.	Acceleration
<i>rx[i].gro_skbs</i>	The number of receive SKBs constructed while performing hardware-accelerated GRO.	Informative
<i>rx[i].gro_match_packets</i>	Number of received packets processed using hardware-accelerated GRO that met the flow table match criteria.	Informative
<i>rx[i].gro_large_hds</i>	Number of receive packets using hardware-accelerated GRO that have large headers that require additional memory to be allocated.	Informative
<i>rx[i].lro_packets</i>	The number of LRO packets received on ring i Page 311, 1 .	Acceleration
<i>rx[i].lro_bytes</i>	The number of LRO bytes received on ring i Page 311, 1 .	Acceleration
<i>rx[i].ecn_mark</i>	The number of received packets where the ECN mark was turned on.	Informative

continues on next page

Table 3 – continued from previous page

<code>rx_oversize_pkts_buffer</code>	The number of dropped received packets due to length which arrived to RQ and exceed software buffer size allocated by the device for incoming traffic. It might imply that the device MTU is larger than the software buffers size.	Error
<code>rx_oversize_pkts_sw_drop</code>	Number of received packets dropped in software because the CQE data is larger than the MTU size.	Error
<code>rx[i].csum_unnecessary</code>	Packets received with a <i>CHECKSUM_UNNECESSARY</i> on ring i ^{Page 311, 1} .	Acceleration
<code>rx[i].csum_unnecessary_inner</code>	Packets received with inner encapsulation with a <i>CHECKSUM_UNNECESSARY</i> on ring i ^{Page 311, 1} .	Acceleration
<code>rx[i].csum_none</code>	Packets received with a <i>CHECKSUM_NONE</i> on ring i ^{Page 311, 1} .	Acceleration
<code>rx[i].csum_complete</code>	Packets received with a <i>CHECKSUM_COMPLETE</i> on ring i ^{Page 311, 1} .	Acceleration
<code>rx[i].csum_complete_tail</code>	Number of received packets that had checksum calculation computed, potentially needed padding, and were able to do so with <i>CHECKSUM_PARTIAL</i> .	Informative
<code>rx[i].csum_complete_tail_slow</code>	Number of received packets that need padding larger than eight bytes for the checksum.	Informative
<code>tx[i].csum_partial</code>	Packets transmitted with a <i>CHECKSUM_PARTIAL</i> on ring i ^{Page 311, 1} .	Acceleration
<code>tx[i].csum_partial_inner</code>	Packets transmitted with inner encapsulation with a <i>CHECKSUM_PARTIAL</i> on ring i ^{Page 311, 1} .	Acceleration
<code>tx[i].csum_none</code>	Packets transmitted with no hardware checksum acceleration on ring i .	Informative

continues on next page

Table 3 – continued from previous page

<i>tx[i].stopped</i>	/	Events where SQ was full on ring i. If this counter is increased, check the amount of buffers allocated for transmission.	Informative
<i>tx[i].wake</i>	/	Events where SQ was full and has become not full on ring i.	Informative
<i>tx[i].dropped</i>	/	Packets transmitted that were dropped due to DMA mapping failure on ring i. If this counter is increased, check the amount of buffers allocated for transmission.	Error
<i>tx[i].nop</i>		The number of nop WQEs (empty WQEs) inserted to the SQ (related to ring i) due to the reach of the end of the cyclic buffer. When reaching near to the end of cyclic buffer the driver may add those empty WQEs to avoid handling a state where a WQE starts in the end of the queue and ends in the beginning of the queue. This is a normal condition.	Informative
<i>tx[i].added_vlan_packets</i>		The number of packets sent where vlan tag insertion was offloaded to the hardware.	Acceleration
<i>rx[i].removed_vlan_packets</i>		The number of packets received where vlan tag stripping was offloaded to the hardware.	Acceleration
<i>rx[i].wqe_err</i>		The number of wrong op-codes received on ring i.	Error
<i>rx[i].mpwqe_frag</i>		The number of WQEs that failed to allocate compound page and hence fragmented MPWQE's (Multi Packet WQEs) were used on ring i. If this counter raise, it may suggest that there is no enough memory for large pages, the driver allocated fragmented pages. This is not abnormal condition.	Informative
<i>rx[i].mpwqe_filler_cqes</i>		The number of filler CQEs events that were issued on ring i.	Informative

continues on next page

Table 3 – continued from previous page

<code>rx[i].mpwqe.filler.strides</code>	The number of strides consumed by filler CQEs on ring i.	Informative
<code>tx[i].mpwqe.blks</code>	The number of send blocks processed from Multi-Packet WQEs (mpwqe).	Informative
<code>tx[i].mpwqe.pkts</code>	The number of send packets processed from Multi-Packet WQEs (mpwqe).	Informative
<code>rx[i].cqe.compress.blks</code>	The number of receive blocks with CQE compression on ring i ^{Page 311, 1} .	Acceleration
<code>rx[i].cqe.compress.pkts</code>	The number of receive packets with CQE compression on ring i ^{Page 311, 1} .	Acceleration
<code>rx[i].arfs.add</code>	The number of aRFS flow rules added to the device for direct RQ steering on ring i ^{Page 311, 1} .	Acceleration
<code>rx[i].arfs.request.in</code>	Number of flow rules that have been requested to move into ring i for direct RQ steering ^{Page 311, 1} .	Acceleration
<code>rx[i].arfs.request.out</code>	Number of flow rules that have been requested to move out of ring i ^{Page 311, 1} .	Acceleration
<code>rx[i].arfs.expired</code>	Number of flow rules that have been expired and removed ^{Page 311, 1} .	Acceleration
<code>rx[i].arfs.err</code>	Number of flow rules that failed to be added to the flow table.	Error
<code>rx[i].recover</code>	The number of times the RQ was recovered.	Error
<code>tx[i].xmit.more</code>	The number of packets sent with <code>xmit.more</code> indication set on the skbuff (no doorbell).	Acceleration
<code>ch[i].poll</code>	The number of invocations of NAPI poll of channel i.	Informative
<code>ch[i].arm</code>	The number of times the NAPI poll function completed and armed the completion queues on channel i.	Informative
<code>ch[i].aff.change</code>	The number of times the NAPI poll function explicitly stopped execution on a CPU due to a change in affinity, on channel i.	Informative

continues on next page

Table 3 – continued from previous page

<i>ch[i].events</i>	The number of hard interrupt events on the completion queues of channel i.	Informative
<i>ch[i].eq_rearm</i>	The number of times the EQ was recovered.	Error
<i>ch[i].force_irq</i>	Number of times NAPI is triggered by XSK wakeups by posting a NOP to ICOSQ.	Acceleration
<i>rx[i].congest_umr</i>	The number of times an outstanding UMR request is delayed due to congestion, on ring i.	Informative
<i>rx_pp_alloc_fast</i>	Number of successful fast path allocations.	Informative
<i>rx_pp_alloc_slow</i>	Number of slow path order-0 allocations.	Informative
<i>rx_pp_alloc_slow_high_order</i>	Number of slow path high order allocations.	Informative
<i>rx_pp_alloc_empty</i>	Counter is incremented when ptr ring is empty, so a slow path allocation was forced.	Informative
<i>rx_pp_alloc_refill</i>	Counter is incremented when an allocation which triggered a refill of the cache.	Informative
<i>rx_pp_alloc_waive</i>	Counter is incremented when pages obtained from the ptr ring that cannot be added to the cache due to a NUMA mismatch.	Informative
<i>rx_pp_recycle_cached</i>	Counter is incremented when recycling placed page in the page pool cache.	Informative
<i>rx_pp_recycle_cache_full</i>	Counter is incremented when page pool cache was full.	Informative
<i>rx_pp_recycle_ring</i>	Counter is incremented when page placed into the ptr ring.	Informative
<i>rx_pp_recycle_ring_full</i>	Counter is incremented when page released from page pool because the ptr ring was full.	Informative
<i>rx_pp_recycle_released_ref</i>	Counter is incremented when page released (and not recycled) because refcnt > 1.	Informative
<i>rx[i].xsk_buff_alloc_err</i>	The number of times allocating an skb or XSK buffer failed in the XSK RQ context.	Error

continues on next page

Table 3 – continued from previous page

<code>rx[i].xdp_tx_xmit</code>	The number of packets forwarded back to the port due to XDP program <i>XDP_TX</i> action (bouncing). These packets are not counted by other software counters. These packets are counted by physical port and vPort counters.	Informative
<code>rx[i].xdp_tx_mpwqe</code>	Number of multi-packet WQEs transmitted by the netdev and <i>XDP_TX</i> -ed by the netdev during the RQ context.	Acceleration
<code>rx[i].xdp_tx_inlnw</code>	Number of WQE data segments transmitted where the data could be inlined in the WQE and then <i>XDP_TX</i> -ed during the RQ context.	Acceleration
<code>rx[i].xdp_tx_nops</code>	Number of NOP WQEBS (WQE building blocks) received posted to the XDP SQ.	Acceleration
<code>rx[i].xdp_tx_full</code>	The number of packets that should have been forwarded back to the port due to <i>XDP_TX</i> action but were dropped due to full tx queue. These packets are not counted by other software counters. These packets are counted by physical port and vPort counters. You may open more rx queues and spread traffic rx over all queues and/or increase rx ring size.	Error
<code>rx[i].xdp_tx_err</code>	The number of times an <i>XDP_TX</i> error such as frame too long and frame too short occurred on <i>XDP_TX</i> ring of RX ring.	Error
<code>rx[i].xdp_tx_cqes</code> <small>/</small> <code>rx_xdp_tx_cqe</code> ^{Page 311, 2}	The number of completions received on the CQ of the <i>XDP_TX</i> ring.	Informative

continues on next page

Table 3 – continued from previous page

<code>rx[i].xdp_drop</code>	The number of packets dropped due to XDP program <code>XDP_DROP</code> action. These packets are not counted by other software counters. These packets are counted by physical port and vPort counters.	Informative
<code>rx[i].xdp_redirect</code>	The number of times an XDP redirect action was triggered on ring i.	Acceleration
<code>tx[i].xdp_xmit</code>	The number of packets redirected to the interface(due to XDP redirect). These packets are not counted by other software counters. These packets are counted by physical port and vPort counters.	Informative
<code>tx[i].xdp_full</code>	The number of packets redirected to the interface(due to XDP redirect), but were dropped due to full tx queue. These packets are not counted by other software counters. You may enlarge tx queues.	Informative
<code>tx[i].xdp_mpwqe</code>	Number of multi-packet WQEs offloaded onto the NIC that were <code>XDP_REDIRECT</code> -ed from other netdevs.	Acceleration
<code>tx[i].xdp_inlnw</code>	Number of WQE data segments where the data could be inlined in the WQE where the data segments were <code>XDP_REDIRECT</code> -ed from other netdevs.	Acceleration
<code>tx[i].xdp_nops</code>	Number of NOP WQE-BBs (WQE building blocks) posted to the SQ that were <code>XDP_REDIRECT</code> -ed from other netdevs.	Acceleration
<code>tx[i].xdp_err</code>	The number of packets redirected to the interface(due to XDP redirect) but were dropped due to error such as frame too long and frame too short.	Error

continues on next page

Table 3 – continued from previous page

<code>tx[i].xdp_cqes</code>	The number of completions received for packets redirected to the interface(due to XDP redirect) on the CQ.	Informative
<code>tx[i].xsk_xmit</code>	The number of packets transmitted using XSK zero-copy functionality.	Acceleration
<code>tx[i].xsk_mpwqe</code>	Number of multi-packet WQEs offloaded onto the NIC that were <i>XDP_REDIRECT</i> -ed from other netdevs.	Acceleration
<code>tx[i].xsk_inlnw</code>	Number of WQE data segments where the data could be inlined in the WQE that are transmitted using XSK zero-copy.	Acceleration
<code>tx[i].xsk_full</code>	Number of times doorbell is rung in XSK zero-copy mode when SQ is full.	Error
<code>tx[i].xsk_err</code>	Number of errors that occurred in XSK zero-copy mode such as if the data size is larger than the MTU size.	Error
<code>tx[i].xsk_cqes</code>	Number of CQEs processed in XSK zero-copy mode.	Acceleration
<code>tx_tls_ctx</code>	Number of TLS TX HW offload contexts added to device for encryption.	Acceleration
<code>tx_tls_del</code>	Number of TLS TX HW offload contexts removed from device (connection closed).	Acceleration
<code>tx_tls_pool_alloc</code>	Number of times a unit of work is successfully allocated in the TLS HW offload pool.	Acceleration
<code>tx_tls_pool_free</code>	Number of times a unit of work is freed in the TLS HW offload pool.	Acceleration
<code>rx_tls_ctx</code>	Number of TLS RX HW offload contexts added to device for decryption.	Acceleration
<code>rx_tls_del</code>	Number of TLS RX HW offload contexts deleted from device (connection has finished).	Acceleration
<code>rx[i].tls_decrypted_packets</code>	Number of successfully decrypted RX packets which were part of a TLS stream.	Acceleration

continues on next page

Table 3 – continued from previous page

<i>rx[i].tls_decrypted_bytes</i>	Number of TLS payload bytes in RX packets which were successfully decrypted.	Acceleration
<i>rx[i].tls_resync_req_pkt</i>	Number of received TLS packets with a resync request.	Acceleration
<i>rx[i].tls_resync_req_start</i>	Number of times the TLS async resync request was started.	Acceleration
<i>rx[i].tls_resync_req_end</i>	Number of times the TLS async resync request properly ended with providing the HW tracked tcp-seq.	Acceleration
<i>rx[i].tls_resync_req_skip</i>	Number of times the TLS async resync request procedure was started but not properly ended.	Error
<i>rx[i].tls_resync_res_ok</i>	Number of times the TLS resync response call to the driver was successfully handled.	Acceleration
<i>rx[i].tls_resync_res_retry</i>	Number of times the TLS resync response call to the driver was reattempted when ICOSQ is full.	Error
<i>rx[i].tls_resync_res_skip</i>	Number of times the TLS resync response call to the driver was terminated unsuccessfully.	Error
<i>rx[i].tls_err</i>	Number of times when CQE TLS offload was problematic.	Error
<i>tx[i].tls_encrypted_packets</i>	The number of send packets that are TLS encrypted by the kernel.	Acceleration
<i>tx[i].tls_encrypted_bytes</i>	The number of send bytes that are TLS encrypted by the kernel.	Acceleration
<i>tx[i].tls_ooo</i>	Number of times out of order TLS SQE fragments were handled on ring i.	Acceleration
<i>tx[i].tls_dump_packets</i>	Number of TLS decrypted packets copied over from NIC over DMA.	Acceleration
<i>tx[i].tls_dump_bytes</i>	Number of TLS decrypted bytes copied over from NIC over DMA.	Acceleration
<i>tx[i].tls_resync_bytes</i>	Number of TLS bytes requested to be resynchronized in order to be decrypted.	Acceleration

continues on next page

Table 3 – continued from previous page

<code>tx[i].tls_skip_no_sync_data</code>	Number of TLS send data that can safely be skipped / do not need to be decrypted.	Acceleration
<code>tx[i].tls_drop_no_sync_data</code>	Number of TLS send data that were dropped due to re-transmission of TLS data.	Acceleration
<code>ptp_cq[i].abort</code>	Number of times a CQE has to be skipped in precision time protocol due to a skew between the port timestamp and CQE timestamp being greater than 128 seconds.	Error
<code>ptp_cq[i].abort_abs_diff_ns</code>	Accumulation of time differences between the port timestamp and CQE timestamp when the difference is greater than 128 seconds in precision time protocol.	Error
<code>ptp_cq[i].late_cqe</code>	Number of times a CQE has been delivered on the PTP timestamping CQ when the CQE was not expected since a certain amount of time had elapsed where the device typically ensures not posting the CQE.	Error

vPort Counters

Counters on the NIC port that is connected to a eSwitch.

¹ Traffic acceleration counter.

² The corresponding ring and global counters do not share the same name (i.e. do not follow the common naming scheme).

Table 4: vPort Counter Table

Counter	Description	Type
<code>rx_vport_unicast_packets</code>	Unicast packets received, steered to a port including Raw Ethernet QP/DPDK traffic, excluding RDMA traffic.	Informative
<code>rx_vport_unicast_bytes</code>	Unicast bytes received, steered to a port including Raw Ethernet QP/DPDK traffic, excluding RDMA traffic.	Informative
<code>tx_vport_unicast_packets</code>	Unicast packets transmitted, steered from a port including Raw Ethernet QP/DPDK traffic, excluding RDMA traffic.	Informative
<code>tx_vport_unicast_bytes</code>	Unicast bytes transmitted, steered from a port including Raw Ethernet QP/DPDK traffic, excluding RDMA traffic.	Informative
<code>rx_vport_multicast_packets</code>	Multicast packets received, steered to a port including Raw Ethernet QP/DPDK traffic, excluding RDMA traffic.	Informative
<code>rx_vport_multicast_bytes</code>	Multicast bytes received, steered to a port including Raw Ethernet QP/DPDK traffic, excluding RDMA traffic.	Informative
<code>tx_vport_multicast_packets</code>	Multicast packets transmitted, steered from a port including Raw Ethernet QP/DPDK traffic, excluding RDMA traffic.	Informative
<code>tx_vport_multicast_bytes</code>	Multicast bytes transmitted, steered from a port including Raw Ethernet QP/DPDK traffic, excluding RDMA traffic.	Informative
<code>rx_vport_broadcast_packets</code>	Broadcast packets received, steered to a port including Raw Ethernet QP/DPDK traffic, excluding RDMA traffic.	Informative
<code>rx_vport_broadcast_bytes</code>	Broadcast bytes received, steered to a port including Raw Ethernet QP/DPDK traffic, excluding RDMA traffic.	Informative
<code>tx_vport_broadcast_packets</code>	Broadcast packets transmitted, steered from a port including Raw Ethernet QP/DPDK traffic, excluding RDMA traffic.	Informative
<code>tx_vport_broadcast_bytes</code>	Broadcast bytes transmitted, steered from a port including Raw Ethernet QP/DPDK traffic, excluding RDMA traffic.	Informative
<code>rx_vport_rdma_unicast_packets</code>	RDMA unicast packets received, steered to a port (counters counts RoCE/UD/RC traffic) ¹ .	Acceleration
<code>rx_vport_rdma_unicast_bytes</code>	RDMA unicast bytes received, steered to a port (counters counts RoCE/UD/RC traffic) ^{Page 311, 1} .	Acceleration
<code>tx_vport_rdma_unicast_packets</code>	RDMA unicast packets transmitted, steered from a port (counters counts RoCE/UD/RC traffic) ^{Page 311, 1} .	Acceleration
<code>tx_vport_rdma_unicast_bytes</code>	RDMA unicast bytes transmitted, steered from a port (counters counts RoCE/UD/RC traffic) ^{Page 311, 1} .	Acceleration
<code>rx_vport_rdma_multicast_packets</code>	RDMA multicast packets received, steered to a port (counters counts RoCE/UD/RC traffic) ^{Page 311, 1} .	Acceleration
<code>rx_vport_rdma_multicast_bytes</code>	RDMA multicast bytes received, steered to a port (counters counts RoCE/UD/RC traffic) ^{Page 311, 1} .	Acceleration

Physical Port Counters

The physical port counters are the counters on the external port connecting the adapter to the network. This measuring point holds information on standardized counters like IEEE 802.3, RFC2863, RFC 2819, RFC 3635 and additional counters like flow control, FEC and more.

Counter	Description
<code>rx_packets_phy</code>	The number of packets received on the physical port. This counter
<code>tx_packets_phy</code>	The number of packets transmitted on the physical port.
<code>rx_bytes_phy</code>	The number of bytes received on the physical port, including Eth
<code>tx_bytes_phy</code>	The number of bytes transmitted on the physical port.
<code>rx_multicast_phy</code>	The number of multicast packets received on the physical port.
<code>tx_multicast_phy</code>	The number of multicast packets transmitted on the physical port
<code>rx_broadcast_phy</code>	The number of broadcast packets received on the physical port.
<code>tx_broadcast_phy</code>	The number of broadcast packets transmitted on the physical port
<code>rx_crc_errors_phy</code>	The number of dropped received packets due to FCS (Frame Che
<code>rx_in_range_len_errors_phy</code>	The number of received packets dropped due to length/type error
<code>rx_out_of_range_len_phy</code>	The number of received packets dropped due to length greater th
<code>rx_oversize_pkts_phy</code>	The number of dropped received packets due to length which exce
<code>rx_symbol_err_phy</code>	The number of received packets dropped due to physical coding e
<code>rx_mac_control_phy</code>	The number of MAC control packets received on the physical por
<code>tx_mac_control_phy</code>	The number of MAC control packets transmitted on the physical p
<code>rx_pause_ctrl_phy</code>	The number of link layer pause packets received on a physical po
<code>tx_pause_ctrl_phy</code>	The number of link layer pause packets transmitted on a physical
<code>rx_unsupported_op_phy</code>	The number of MAC control packets received with unsupported o
<code>rx_discards_phy</code>	The number of received packets dropped due to lack of buffers or
<code>tx_discards_phy</code>	The number of packets which were discarded on transmission, ev
<code>tx_errors_phy</code>	The number of transmitted packets dropped due to a length whic
<code>rx_undersize_pkts_phy</code>	The number of received packets dropped due to length which is s
<code>rx_fragments_phy</code>	The number of received packets dropped due to a length which is
<code>rx_jabbers_phy</code>	The number of received packets d due to a length which is longer
<code>rx_64_bytes_phy</code>	The number of packets received on the physical port with size of
<code>rx_65_to_127_bytes_phy</code>	The number of packets received on the physical port with size of
<code>rx_128_to_255_bytes_phy</code>	The number of packets received on the physical port with size of
<code>rx_256_to_511_bytes_phy</code>	The number of packets received on the physical port with size of
<code>rx_512_to_1023_bytes_phy</code>	The number of packets received on the physical port with size of
<code>rx_1024_to_1518_bytes_phy</code>	The number of packets received on the physical port with size of
<code>rx_1519_to_2047_bytes_phy</code>	The number of packets received on the physical port with size of
<code>rx_2048_to_4095_bytes_phy</code>	The number of packets received on the physical port with size of
<code>rx_4096_to_8191_bytes_phy</code>	The number of packets received on the physical port with size of
<code>rx_8192_to_10239_bytes_phy</code>	The number of packets received on the physical port with size of
<code>link_down_events_phy</code>	The number of times where the link operative state changed to d
<code>rx_out_of_buffer</code>	Number of times receive queue had no software buffers allocated
<code>module_bus_stuck</code>	The number of times that module's I ² C bus (data or clock) short-w
<code>module_high_temp</code>	The number of times that the module temperature was too high.
<code>module_bad_shorted</code>	The number of times that the module cables were shorted. You m
<code>module_unplug</code>	The number of times that module was ejected.

<code>rx_buffer_passed_thres_phy</code>	The number of events where the port receive buffer was over 85%
<code>tx_pause_storm_warning_events</code>	The number of times the device was sending pauses for a long period
<code>tx_pause_storm_error_events</code>	The number of times the device was sending pauses for a long period
<code>rx[i].buff_alloc_err</code>	Failed to allocate a buffer to received packet (or SKB) on ring i.
<code>rx_bits_phy</code>	This counter provides information on the total amount of traffic transferred
<code>rx_pcs_symbol_err_phy</code>	This counter counts the number of symbol errors that wasn't corrected by FEC
<code>rx_corrected_bits_phy</code>	The number of corrected bits on this port according to active FEC
<code>rx_err_lane[l].phy</code>	This counter counts the number of physical raw errors per lane l
<code>rx_global_pause</code>	The number of pause packets received on the physical port. If this counter reaches the limit, the port will be disabled
<code>rx_global_pause_duration</code>	The duration of pause received (in microSec) on the physical port
<code>tx_global_pause</code>	The number of pause packets transmitted on a physical port. If this counter reaches the limit, the port will be disabled
<code>tx_global_pause_duration</code>	The duration of pause transmitter (in microSec) on the physical port
<code>rx_global_pause_transition</code>	The number of times a transition from Xoff to Xon on the physical port
<code>rx_if_down_packets</code>	The number of received packets that were dropped due to interface down

Priority Port Counters

The following counters are physical port counters that are counted per L2 priority (0-7).

Note: *p* in the counter name represents the priority.

Table 6: Priority Port Counter Table

Counter	Description	Type
<i>rx_prio[p]_bytes</i>	The number of bytes received with priority p on the physical port.	Informative
<i>rx_prio[p]_packets</i>	The number of packets received with priority p on the physical port.	Informative
<i>tx_prio[p]_bytes</i>	The number of bytes transmitted on priority p on the physical port.	Informative
<i>tx_prio[p]_packets</i>	The number of packets transmitted on priority p on the physical port.	Informative
<i>rx_prio[p]_pause</i>	The number of pause packets received with priority p on a physical port. If this counter is increasing, it implies that the network is congested and cannot absorb the traffic coming from the adapter. Note: This counter is available only if PFC was enabled on priority p.	Informative
<i>rx_prio[p]_pause_duration</i>	The duration of pause received (in microSec) on priority p on the physical port. The counter represents the time the port did not send any traffic on this priority. If this counter is increasing, it implies that the network is congested and cannot absorb the traffic coming from the adapter. Note: This counter is available only if PFC was enabled on priority p.	Informative
<i>rx_prio[p]_pause_transition</i>	The number of times a transition from Xoff to Xon on priority p on the physical port has occurred. Note: This counter is available only if PFC was enabled on priority p.	Informative
<i>tx_prio[p]_pause</i>	The number of pause packets transmitted on priority p on a physical port. If this counter is increasing, it implies that the adapter is congested and cannot absorb the traffic coming from the network. Note: This counter is available only if PFC was enabled on priority p.	Informative
<i>tx_prio[p]_pause_duration</i>	The duration of pause transmitter (in microSec) on priority p on the physical port. Note: This counter is available only if PFC was enabled on priority p.	Informative
<i>rx_prio[p]_buf_discard</i>	The number of packets discarded by device due to lack of per host receive buffers.	Informative
<i>rx_prio[p]_cong_discard</i>	The number of packets discarded by device due to per host congestion.	Informative
<i>rx_prio[p]_marked</i>	The number of packets ecn marked by device due to per host congestion.	Informative
<i>rx_prio[p]_discards</i>	The number of packets discarded by device due to lack of receive buffers.	Informative

Device Counters

Table 7: Device Counter Table

Counter	Description	Type
<i>rx_pci_signal_integrity</i>	Counts physical layer PCIe signal integrity errors, the number of transitions to recovery due to Framing errors and CRC (dlp and tlp). If this counter is raising, try moving the adapter card to a different slot to rule out a bad PCI slot. Validate that you are running with the latest firmware available and latest server BIOS version.	Error
<i>tx_pci_signal_integrity</i>	Counts physical layer PCIe signal integrity errors, the number of transition to recovery initiated by the other side (moving to recovery due to getting TS/EIEOS). If this counter is raising, try moving the adapter card to a different slot to rule out a bad PCI slot. Validate that you are running with the latest firmware available and latest server BIOS version.	Error
<i>out-bound_pci_buffer_overflow</i>	The number of packets dropped due to pci buffer overflow. If this counter is raising in high rate, it might indicate that the receive traffic rate for a host is larger than the PCIe bus and therefore a congestion occurs.	Informative
<i>outbound_pci_stalled_rd</i>	The percentage (in the range 0...100) of time within the last second that the NIC had out-bound non-posted reads requests but could not perform the operation due to insufficient posted credits.	Informative
<i>outbound_pci_stalled_wr</i>	The percentage (in the range 0...100) of time within the last second that the NIC had out-bound posted writes requests but could not perform the operation due to insufficient posted credits.	Informative
<i>out-bound_pci_stalled_rd_events</i>	The number of seconds where <i>out-bound_pci_stalled_rd</i> was above 30%.	Informative
<i>out-bound_pci_stalled_wr_events</i>	The number of seconds where <i>out-bound_pci_stalled_wr</i> was above 30%.	Informative
<i>dev_out_of_buffer</i>	The number of times the device owned queue had not enough buffers allocated.	Error

6.5.34 Hyper-V network driver

Compatibility

This driver is compatible with Windows Server 2012 R2, 2016 and Windows 10.

Features

Checksum offload

The netvsc driver supports checksum offload as long as the Hyper-V host version does. Windows Server 2016 and Azure support checksum offload for TCP and UDP for both IPv4 and IPv6. Windows Server 2012 only supports checksum offload for TCP.

Receive Side Scaling

Hyper-V supports receive side scaling. For TCP & UDP, packets can be distributed among available queues based on IP address and port number.

For TCP & UDP, we can switch hash level between L3 and L4 by ethtool command. TCP/UDP over IPv4 and v6 can be set differently. The default hash level is L4. We currently only allow switching TX hash level from within the guests.

On Azure, fragmented UDP packets have high loss rate with L4 hashing. Using L3 hashing is recommended in this case.

For example, for UDP over IPv4 on eth0:

To include UDP port numbers in hashing:

```
ethtool -N eth0 rx-flow-hash udp4 sdfn
```

To exclude UDP port numbers in hashing:

```
ethtool -N eth0 rx-flow-hash udp4 sd
```

To show UDP hash level:

```
ethtool -n eth0 rx-flow-hash udp4
```

Generic Receive Offload, aka GRO

The driver supports GRO and it is enabled by default. GRO coalesces like packets and significantly reduces CPU usage under heavy Rx load.

Large Receive Offload (LRO), or Receive Side Coalescing (RSC)

The driver supports LRO/RSC in the vSwitch feature. It reduces the per packet processing overhead by coalescing multiple TCP segments when possible. The feature is enabled by default on VMs running on Windows Server 2019 and later. It may be changed by ethtool command:

```
ethtool -K eth0 lro on  
ethtool -K eth0 lro off
```

SR-IOV support

Hyper-V supports SR-IOV as a hardware acceleration option. If SR-IOV is enabled in both the vSwitch and the guest configuration, then the Virtual Function (VF) device is passed to the guest as a PCI device. In this case, both a synthetic (netvsc) and VF device are visible in the guest OS and both NIC's have the same MAC address.

The VF is enslaved by netvsc device. The netvsc driver will transparently switch the data path to the VF when it is available and up. Network state (addresses, firewall, etc) should be applied only to the netvsc device; the slave device should not be accessed directly in most cases. The exceptions are if some special queue discipline or flow direction is desired, these should be applied directly to the VF slave device.

Receive Buffer

Packets are received into a receive area which is created when device is probed. The receive area is broken into MTU sized chunks and each may contain one or more packets. The number of receive sections may be changed via ethtool Rx ring parameters.

There is a similar send buffer which is used to aggregate packets for sending. The send area is broken into chunks, typically of 6144 bytes, each of section may contain one or more packets. Small packets are usually transmitted via copy to the send buffer. However, if the buffer is temporarily exhausted, or the packet to be transmitted is an LSO packet, the driver will provide the host with pointers to the data from the SKB. This attempts to achieve a balance between the overhead of data copy and the impact of remapping VM memory to be accessible by the host.

XDP support

XDP (eXpress Data Path) is a feature that runs eBPF bytecode at the early stage when packets arrive at a NIC card. The goal is to increase performance for packet processing, reducing the overhead of SKB allocation and other upper network layers.

hv_netvsc supports XDP in native mode, and transparently sets the XDP program on the associated VF NIC as well.

Setting / unsetting XDP program on synthetic NIC (netvsc) propagates to VF NIC automatically. Setting / unsetting XDP program on VF NIC directly is not recommended,

also not propagated to synthetic NIC, and may be overwritten by setting of synthetic NIC.

XDP program cannot run with LRO (RSC) enabled, so you need to disable LRO before running XDP:

```
ethtool -K eth0 lro off
```

XDP_REDIRECT action is not yet supported.

6.5.35 Neterion's (Formerly S2io) Xframe I/II PCI-X 10GbE driver

Release notes for Neterion's (Formerly S2io) Xframe I/II PCI-X 10GbE driver.

1. Introduction

This Linux driver supports Neterion's Xframe I PCI-X 1.0 and Xframe II PCI-X 2.0 adapters. It supports several features such as jumbo frames, MSI/MSI-X, checksum offloads, TSO, UFO and so on. See below for complete list of features.

All features are supported for both IPv4 and IPv6.

2. Identifying the adapter/interface

- Insert the adapter(s) in your system.
- Build and load driver:

```
# insmod s2io.ko
```

- View log messages:

```
# dmesg | tail -40
```

You will see messages similar to:

```
eth3: Neterion Xframe I 10GbE adapter (rev 3), Version 2.0.9.1, Intr type INTA
eth4: Neterion Xframe II 10GbE adapter (rev 2), Version 2.0.9.1, Intr type INTA
eth4: Device is on 64 bit 133MHz PCIX(M1) bus
```

The above messages identify the adapter type(Xframe I/II), adapter revision, driver version, interface name(eth3, eth4), Interrupt type(INTA, MSI, MSI-X). In case of Xframe II, the PCI/PCI-X bus width and frequency are displayed as well.

To associate an interface with a physical adapter use "ethtool -p <ethX>". The corresponding adapter's LED will blink multiple times.

3. Features supported

- a. Jumbo frames. Xframe I/II supports MTU up to 9600 bytes, modifiable using ip command.
- b. Offloads. Supports checksum offload(TCP/UDP/IP) on transmit and receive, TSO.
- c. Multi-buffer receive mode. Scattering of packet across multiple buffers. Currently driver supports 2-buffer mode which yields significant performance improvement on certain platforms(SGI Altix, IBM xSeries).
- d. MSI/MSI-X. Can be enabled on platforms which support this feature resulting in noticeable performance improvement (up to 7% on certain platforms).
- e. Statistics. Comprehensive MAC-level and software statistics displayed using "ethtool -S" option.
- f. Multi-FIFO/Ring. Supports up to 8 transmit queues and receive rings, with multiple steering options.

4. Command line parameters

a. **tx_fifo_num**

Number of transmit queues

Valid range: 1-8

Default: 1

b. **rx_ring_num**

Number of receive rings

Valid range: 1-8

Default: 1

c. **tx_fifo_len**

Size of each transmit queue

Valid range: Total length of all queues should not exceed 8192

Default: 4096

d. **rx_ring_sz**

Size of each receive ring(in 4K blocks)

Valid range: Limited by memory on system

Default: 30

e. **intr_type**

Specifies interrupt type. Possible values 0(INTA), 2(MSI-X)

Valid values: 0, 2

Default: 2

5. Performance suggestions

General:

- a. Set MTU to maximum(9000 for switch setup, 9600 in back-to-back configuration)
- b. Set TCP windows size to optimal value.

For instance, for MTU=1500 a value of 210K has been observed to result in good performance:

```
# sysctl -w net.ipv4.tcp_rmem="210000 210000 210000"
# sysctl -w net.ipv4.tcp_wmem="210000 210000 210000"
```

For MTU=9000, TCP window size of 10 MB is recommended:

```
# sysctl -w net.ipv4.tcp_rmem="10000000 10000000 10000000"
# sysctl -w net.ipv4.tcp_wmem="10000000 10000000 10000000"
```

Transmit performance:

- a. By default, the driver respects BIOS settings for PCI bus parameters. However, you may want to experiment with PCI bus parameters max-split-transactions(MOST) and MMRBC (use setpci command).

A MOST value of 2 has been found optimal for Opterons and 3 for Itanium.

It could be different for your hardware.

Set MMRBC to 4K**.

For example you can set

For opteron:

```
#setpci -d 17d5:* 62=1d
```

For Itanium:

```
#setpci -d 17d5:* 62=3d
```

For detailed description of the PCI registers, please see Xframe User Guide.

- b. Ensure Transmit Checksum offload is enabled. Use ethtool to set/verify this parameter.
- c. Turn on TSO(using "ethtool -K"):

```
# ethtool -K <ethX> tso on
```

Receive performance:

- a. By default, the driver respects BIOS settings for PCI bus parameters. However, you may want to set PCI latency timer to 248:

```
#setpci -d 17d5:* LATENCY_TIMER=f8
```

For detailed description of the PCI registers, please see Xframe User Guide.

- b. Use 2-buffer mode. This results in large performance boost on certain platforms(eg. SGI Altix, IBM xSeries).

- c. Ensure Receive Checksum offload is enabled. Use "ethtool -K ethX" command to set/verify this option.
- d. Enable NAPI feature(in kernel configuration Device Drivers ---> Network device support ---> Ethernet (10000 Mbit) ---> S2IO 10GbE Xframe NIC) to bring down CPU utilization.

Note: For AMD opteron platforms with 8131 chipset, MMRBC=1 and MOST=1 are recommended as safe parameters.

For more information, please review the AMD8131 errata at http://vip.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/26310_AMD-8131_HyperTransport_PCI-X_Tunnel_Revision_Guide_rev_3_18.pdf

6. Support

For further support please contact either your 10GbE Xframe NIC vendor (IBM, HP, SGI etc.)

6.5.36 Network Flow Processor (NFP) Kernel Drivers

Copyright

© 2019, Netronome Systems, Inc.

Copyright

© 2022, Coragine, Inc.

Contents

- *Overview*
- *Acquiring Firmware*
- *Devlink Info*
- *Configure Device*
- *Statistics*

Overview

This driver supports Netronome and Coragine's line of Network Flow Processor devices, including the NFP3800, NFP4000, NFP5000, and NFP6000 models, which are also incorporated in the companies' family of Agilio SmartNICs. The SR-IOV physical and virtual functions for these devices are supported by the driver.

Acquiring Firmware

The NFP3800, NFP4000 and NFP6000 devices require application specific firmware to function. Application firmware can be located either on the host file system or in the device flash (if supported by management firmware).

Firmware files on the host filesystem contain card type (*AMDA-** string), media config etc. They should be placed in */lib/firmware/netronome* directory to load firmware from the host file system.

Firmware for basic NIC operation is available in the upstream *linux-firmware.git* repository.

A more comprehensive list of firmware can be downloaded from the [Corigine Support site](#).

Firmware in NVRAM

Recent versions of management firmware supports loading application firmware from flash when the host driver gets probed. The firmware loading policy configuration may be used to configure this feature appropriately.

Devlink or ethtool can be used to update the application firmware on the device flash by providing the appropriate *nic_AMDA*.nffw* file to the respective command. Users need to take care to write the correct firmware image for the card and media configuration to flash.

Available storage space in flash depends on the card being used.

Dealing with multiple projects

NFP hardware is fully programmable therefore there can be different firmware images targeting different applications.

When using application firmware from host, we recommend placing actual firmware files in application-named subdirectories in */lib/firmware/netronome* and linking the desired files, e.g.:

```
$ tree /lib/firmware/netronome/
/lib/firmware/netronome/
└── bpf
    ├── nic_AMDA0081-0001_1x40.nffw
    └── nic_AMDA0081-0001_4x10.nffw
└── flower
    ├── nic_AMDA0081-0001_1x40.nffw
    └── nic_AMDA0081-0001_4x10.nffw
└── nic
    ├── nic_AMDA0081-0001_1x40.nffw
    └── nic_AMDA0081-0001_4x10.nffw
    └── nic_AMDA0081-0001_1x40.nffw -> bpf/nic_AMDA0081-0001_1x40.nffw
    └── nic_AMDA0081-0001_4x10.nffw -> bpf/nic_AMDA0081-0001_4x10.nffw
3 directories, 8 files
```

You may need to use *hard* instead of *symbolic links* on distributions which use old *mkinitrd* command instead of *dracut* (e.g. Ubuntu).

After changing firmware files you may need to regenerate the initramfs image. Initramfs contains drivers and firmware files your system may need to boot. Refer to the documentation of your distribution to find out how to update initramfs. Good indication of stale initramfs is system loading wrong driver or firmware on boot, but when driver is later reloaded manually everything works correctly.

Selecting firmware per device

Most commonly all cards on the system use the same type of firmware. If you want to load a specific firmware image for a specific card, you can use either the PCI bus address or serial number. The driver will print which files it's looking for when it recognizes a NFP device:

```
nfp: Looking for firmware file in order of priority:  
nfp: netronome/serial-00-12-34-aa-bb-cc-10-ff.nffw: not found  
nfp: netronome/pci-0000:02:00.0.nffw: not found  
nfp: netronome/nic_AMDA0081-0001_1x40.nffw: found, loading...
```

In this case if file (or link) called *serial-00-12-34-aa-bb-5d-10-ff.nffw* or *pci-0000:02:00.0.nffw* is present in */lib/firmware/netronome* this firmware file will take precedence over *nic_AMDA** files.

Note that *serial-** and *pci-** files are **not** automatically included in initramfs, you will have to refer to documentation of appropriate tools to find out how to include them.

Running firmware version

The version of the loaded firmware for a particular <netdev> interface, (e.g. *enp4s0*), or an interface's port <netdev port> (e.g. *enp4s0np0*) can be displayed with the ethtool command:

```
$ ethtool -i <netdev>
```

Firmware loading policy

Firmware loading policy is controlled via three HWinfo parameters stored as key value pairs in the device flash:

app_fw_from_flash

Defines which firmware should take precedence, 'Disk' (0), 'Flash' (1) or the 'Preferred' (2) firmware. When 'Preferred' is selected, the management firmware makes the decision over which firmware will be loaded by comparing versions of the flash firmware and the host supplied firmware. This variable is configurable using the 'fw_load_policy' devlink parameter.

abi_drv_reset

Defines if the driver should reset the firmware when the driver is probed, either 'Disk' (0) if firmware was found on disk, 'Always' (1) reset or 'Never' (2) reset. Note that the device is always reset on driver unload if firmware was loaded when the driver was probed. This variable is configurable using the 'reset_dev_on_drv_probe' devlink parameter.

abi_drv_load_ifc

Defines a list of PF devices allowed to load FW on the device. This variable is not currently user configurable.

Devlink Info

The devlink info command displays the running and stored firmware versions on the device, serial number and board information.

Devlink info command example (replace PCI address):

```
$ devlink dev info pci/0000:03:00.0
pci/0000:03:00.0:
  driver nfp
  serial_number CSAAMDA2001-1003000111
  versions:
    fixed:
      board.id AMDA2001-1003
      board.rev 01
      board.manufacture CSA
      board.model mozart
    running:
      fw.mgmt 22.10.0-rc3
      fw.cpld 0x1000003
      fw.app nic-22.09.0
      chip.init AMDA-2001-1003 1003000111
    stored:
      fw.bundle_id bspbundle_1003000111
      fw.mgmt 22.10.0-rc3
      fw.cpld 0x0
      chip.init AMDA-2001-1003 1003000111
```

Configure Device

This section explains how to use Agilio SmartNICs running basic NIC firmware.

Configure interface link-speed

The following steps explains how to change between 10G mode and 25G mode on Agilio CX 2x25GbE cards. The changing of port speed must be done in order, port 0 (p0) must be set to 10G before port 1 (p1) may be set to 10G.

Down the respective interface(s):

```
$ ip link set dev <netdev port 0> down
$ ip link set dev <netdev port 1> down
```

Set interface link-speed to 10G:

```
$ ethtool -s <netdev port 0> speed 10000
$ ethtool -s <netdev port 1> speed 10000
```

Set interface link-speed to 25G:

```
$ ethtool -s <netdev port 0> speed 25000
$ ethtool -s <netdev port 1> speed 25000
```

Reload driver for changes to take effect:

```
$ rmmod nfp; modprobe nfp
```

Configure interface Maximum Transmission Unit (MTU)

The MTU of interfaces can temporarily be set using the iproute2, ip link or ifconfig tools. Note that this change will not persist. Setting this via Network Manager, or another appropriate OS configuration tool, is recommended as changes to the MTU using Network Manager can be made to persist.

Set interface MTU to 9000 bytes:

```
$ ip link set dev <netdev port> mtu 9000
```

It is the responsibility of the user or the orchestration layer to set appropriate MTU values when handling jumbo frames or utilizing tunnels. For example, if packets sent from a VM are to be encapsulated on the card and egress a physical port, then the MTU of the VF should be set to lower than that of the physical port to account for the extra bytes added by the additional header. If a setup is expected to see fallback traffic between the SmartNIC and the kernel then the user should also ensure that the PF MTU is appropriately set to avoid unexpected drops on this path.

Configure Forward Error Correction (FEC) modes

Agilio SmartNICs support FEC mode configuration, e.g. Auto, Firecode Base-R, ReedSolomon and Off modes. Each physical port's FEC mode can be set independently using ethtool. The supported FEC modes for an interface can be viewed using:

```
$ ethtool <netdev>
```

The currently configured FEC mode can be viewed using:

```
$ ethtool --show-fec <netdev>
```

To force the FEC mode for a particular port, auto-negotiation must be disabled (see the [Auto-negotiation](#) section). An example of how to set the FEC mode to Reed-Solomon is:

```
$ ethtool --set-fec <netdev> encoding rs
```

Auto-negotiation

To change auto-negotiation settings, the link must first be put down. After the link is down, auto-negotiation can be enabled or disabled using:

```
ethtool -s <netdev> autoneg <on|off>
```

Statistics

Following device statistics are available through the `ethtool -S` interface:

Table 8: NFP device statistics

Name	ID	Meaning
dev_rx_discards	1	<p>Packet can be discarded on the RX path for one of the following reasons:</p> <ul style="list-style-type: none"> The NIC is not in promisc mode, and the destination MAC address doesn't match the interfaces' MAC address. The received packet is larger than the max buffer size on the host. I.e. it exceeds the Layer 3 MRU. There is no freelist descriptor available on the host for the packet. It is likely that the NIC couldn't cache one in time. A BPF program discarded the packet. The datapath drop action was executed. The MAC discarded the packet due to lack of ingress buffer space on the NIC.
dev_rx_errors	2	<p>A packet can be counted (and dropped) as RX error for the following reasons:</p> <ul style="list-style-type: none"> A problem with the VEB lookup (only when SR-IOV is used). A physical layer problem that causes Ethernet errors, like FCS or alignment errors. The cause is usually faulty cables or SFPs.
dev_rx_bytes	3	Total number of bytes received.
dev_rx_uc_bytes	4	Unicast bytes received.
dev_rx_mc_bytes	5	Multicast bytes received.
dev_rx_bc_bytes	6	Broadcast bytes received.
dev_rx_pkts	7	Total number of packets received.
dev_rx_mc_pkts	8	Multicast packets received.
328 dev_rx_bc_pkts	9	Chapter 6. Hardware Device Drivers Broadcast packets received.
dev_tx_discards	10	A packet can be discarded in the TX direction if the MAC is being flow controlled and

Note that statistics unknown to the driver will be displayed as `dev_unknown_stat$ID`, where \$ID refers to the second column above.

6.5.37 Linux Driver for the Pensando(R) Ethernet adapter family

Pensando Linux Ethernet driver. Copyright(c) 2019 Pensando Systems, Inc

Contents

- Identifying the Adapter
- Enabling the driver
- Configuring the driver
- Statistics
- Support

Identifying the Adapter

To find if one or more Pensando PCI Ethernet devices are installed on the host, check for the PCI devices:

```
$ lspci -d 1dd8:  
b5:00.0 Ethernet controller: Device 1dd8:1002  
b6:00.0 Ethernet controller: Device 1dd8:1002
```

If such devices are listed as above, then the ionic.ko driver should find and configure them for use. There should be log entries in the kernel messages such as these:

```
$ dmesg | grep ionic  
ionic 0000:b5:00.0: 126.016 Gb/s available PCIe bandwidth (8.0 GT/s PCIe x16  
↳ link)  
ionic 0000:b5:00.0 enp181s0: renamed from eth0  
ionic 0000:b5:00.0 enp181s0: Link up - 100 Gbps  
ionic 0000:b6:00.0: 126.016 Gb/s available PCIe bandwidth (8.0 GT/s PCIe x16  
↳ link)  
ionic 0000:b6:00.0 enp182s0: renamed from eth0  
ionic 0000:b6:00.0 enp182s0: Link up - 100 Gbps
```

Driver and firmware version information can be gathered with either of ethtool or devlink tools:

```
$ ethtool -i enp181s0  
driver: ionic  
version: 5.7.0  
firmware-version: 1.8.0-28  
...  
  
$ devlink dev info pci/0000:b5:00.0  
pci/0000:b5:00.0:  
    driver ionic
```

```
serial_number FLM18420073
versions:
    fixed:
        asic.id 0x0
        asic.rev 0x0
    running:
        fw 1.8.0-28
```

See [ionic devlink support](#) for more information on the devlink dev info data.

Enabling the driver

The driver is enabled via the standard kernel configuration system, using the make command:

```
make oldconfig/menuconfig/etc.
```

The driver is located in the menu structure at:

- > **Device Drivers**
 - > **Network device support (NETDEVICES [=y])**
 - > **Ethernet driver support**
 - > **Pensando devices**
 - > Pensando Ethernet IONIC Support

Configuring the Driver

MTU

Jumbo frame support is available with a maximum size of 9194 bytes.

Interrupt coalescing

Interrupt coalescing can be configured by changing the rx-usecs value with the "ethtool -C" command. The rx-usecs range is 0-190. The tx-usecs value reflects the rx-usecs value as they are tied together on the same interrupt.

SR-IOV

Minimal SR-IOV support is currently offered and can be enabled by setting the sysfs 'sriov_numvfs' value, if supported by your particular firmware configuration.

Statistics

Basic hardware stats

The commands `netstat -i`, `ip -s link show`, and `ifconfig` show a limited set of statistics taken directly from firmware. For example:

```
$ ip -s link show enp181s0
7: enp181s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP modeu
  ↳DEFAULT group default qlen 1000
    link/ether 00:ae:cd:00:07:68 brd ff:ff:ff:ff:ff:ff
    RX: bytes  packets errors dropped overrun mcast
      414        5        0        0        0
    TX: bytes  packets errors dropped carrier collsns
      1384       18        0        0        0
```

`ethtool -S`

The statistics shown from the `ethtool -S` command includes a combination of driver counters and firmware counters, including port and queue specific values. The driver values are counters computed by the driver, and the firmware values are gathered by the firmware from the port hardware and passed through the driver with no further interpretation.

Driver port specific:

```
tx_packets: 12
tx_bytes: 964
rx_packets: 5
rx_bytes: 414
tx_tso: 0
tx_tso_bytes: 0
tx_csum_none: 12
tx_csum: 0
rx_csum_none: 0
rx_csum_complete: 3
rx_csum_error: 0
```

Driver queue specific:

```
tx_0_pkts: 3
tx_0_bytes: 294
tx_0_clean: 3
tx_0_dma_map_err: 0
tx_0_linearize: 0
tx_0_frags: 0
tx_0_tso: 0
tx_0_tso_bytes: 0
tx_0_csum_none: 3
tx_0_csum: 0
tx_0_vlan_inserted: 0
rx_0_pkts: 2
```

```
rx_0_bytes: 120
rx_0_dma_map_err: 0
rx_0_alloc_err: 0
rx_0_csum_none: 0
rx_0_csum_complete: 0
rx_0_csum_error: 0
rx_0_dropped: 0
rx_0_vlan_stripped: 0
```

Firmware port specific:

```
hw_tx_dropped: 0
hw_rx_dropped: 0
hw_rx_over_errors: 0
hw_rx_missed_errors: 0
hw_tx_aborted_errors: 0
frames_rx_ok: 15
frames_rx_all: 15
frames_rx_bad_fcs: 0
frames_rx_bad_all: 0
octets_rx_ok: 1290
octets_rx_all: 1290
frames_rx_unicast: 10
frames_rx_multicast: 5
frames_rx_broadcast: 0
frames_rx_pause: 0
frames_rx_bad_length: 0
frames_rx_undersized: 0
frames_rx_oversized: 0
frames_rx_fragments: 0
frames_rx_jabber: 0
frames_rx_pripause: 0
frames_rx_stomped_crc: 0
frames_rx_too_long: 0
frames_rx_vlan_good: 3
frames_rx_dropped: 0
frames_rx_less_than_64b: 0
frames_rx_64b: 4
frames_rx_65b_127b: 11
frames_rx_128b_255b: 0
frames_rx_256b_511b: 0
frames_rx_512b_1023b: 0
frames_rx_1024b_1518b: 0
frames_rx_1519b_2047b: 0
frames_rx_2048b_4095b: 0
frames_rx_4096b_8191b: 0
frames_rx_8192b_9215b: 0
frames_rx_other: 0
frames_tx_ok: 31
frames_tx_all: 31
frames_tx_bad: 0
```

```
octets_tx_ok: 2614
octets_tx_total: 2614
frames_tx_unicast: 8
frames_tx_multicast: 21
frames_tx_broadcast: 2
frames_tx_pause: 0
frames_tx_pripause: 0
frames_tx_vlan: 0
frames_tx_less_than_64b: 0
frames_tx_64b: 4
frames_tx_65b_127b: 27
frames_tx_128b_255b: 0
frames_tx_256b_511b: 0
frames_tx_512b_1023b: 0
frames_tx_1024b_1518b: 0
frames_tx_1519b_2047b: 0
frames_tx_2048b_4095b: 0
frames_tx_4096b_8191b: 0
frames_tx_8192b_9215b: 0
frames_tx_other: 0
frames_tx_pri_0: 0
frames_tx_pri_1: 0
frames_tx_pri_2: 0
frames_tx_pri_3: 0
frames_tx_pri_4: 0
frames_tx_pri_5: 0
frames_tx_pri_6: 0
frames_tx_pri_7: 0
frames_rx_pri_0: 0
frames_rx_pri_1: 0
frames_rx_pri_2: 0
frames_rx_pri_3: 0
frames_rx_pri_4: 0
frames_rx_pri_5: 0
frames_rx_pri_6: 0
frames_rx_pri_7: 0
tx_pripause_0_lus_count: 0
tx_pripause_1_lus_count: 0
tx_pripause_2_lus_count: 0
tx_pripause_3_lus_count: 0
tx_pripause_4_lus_count: 0
tx_pripause_5_lus_count: 0
tx_pripause_6_lus_count: 0
tx_pripause_7_lus_count: 0
rx_pripause_0_lus_count: 0
rx_pripause_1_lus_count: 0
rx_pripause_2_lus_count: 0
rx_pripause_3_lus_count: 0
rx_pripause_4_lus_count: 0
rx_pripause_5_lus_count: 0
```

```
rx_pripause_6_lus_count: 0
rx_pripause_7_lus_count: 0
rx_pause_lus_count: 0
frames_tx_truncated: 0
```

Support

For general Linux networking support, please use the netdev mailing list, which is monitored by Pensando personnel:

```
netdev@vger.kernel.org
```

For more specific support needs, please use the Pensando driver support email:

```
drivers@pensando.io
```

6.5.38 SMC 9xxxx Driver

Revision 0.12

3/5/96

Copyright 1996 Erik Stahlman

Released under terms of the GNU General Public License.

This file contains the instructions and caveats for my SMC9xxx driver. You should not be using the driver without reading this file.

Things to note about installation:

1. The driver should work on all kernels from 1.2.13 until 1.3.71. (A kernel patch is supplied for 1.3.71)
2. If you include this into the kernel, you might need to change some options, such as for forcing IRQ.
3. To compile as a module, run 'make'. Make will give you the appropriate options for various kernel support.
4. Loading the driver as a module:

```
use: insmod smc9194.o
optional parameters:
    io=xxxx   : your base address
    irq=xx    : your irq
    ifport=x  :    0 for whatever is default
                  1 for twisted pair
                  2 for AUI ( or BNC on some cards )
```

How to obtain the latest version?

FTP:

<ftp://fenris.campus.vt.edu/sm9/sm9-12.tar.gz> <ftp://sfbox.vt.edu/filebox/F/fenris/sm9/sm9-12.tar.gz>

Contacting me:

erik@mail.vt.edu

6.5.39 Linux Driver for the Synopsys(R) Ethernet Controllers "stmmac"

Authors: Giuseppe Cavallaro <peppe.cavallaro@st.com>, Alexandre Torgue <alexandre.torgue@st.com>, Jose Abreu <joabreu@synopsys.com>

Contents

- In This Release
- Feature List
- Kernel Configuration
- Command Line Parameters
- Driver Information and Notes
- Debug Information
- Support

In This Release

This file describes the stmmac Linux Driver for all the Synopsys(R) Ethernet Controllers.

Currently, this network device driver is for all STi embedded MAC/GMAC (i.e. 7xxx/5xxx SoCs), SPEAr (arm), Loongson1B (mips) and XILINX XC2V3000 FF1152AMT0221 D1215994A VIRTEX FPGA board. The Synopsys Ethernet QoS 5.0 IPK is also supported.

DesignWare(R) Cores Ethernet MAC 10/100/1000 Universal version 3.70a (and older) and DesignWare(R) Cores Ethernet Quality-of-Service version 4.0 (and upper) have been used for developing this driver as well as DesignWare(R) Cores XGMAC - 10G Ethernet MAC and DesignWare(R) Cores Enterprise MAC - 100G Ethernet MAC.

This driver supports both the platform bus and PCI.

This driver includes support for the following Synopsys(R) DesignWare(R) Cores Ethernet Controllers and corresponding minimum and maximum versions:

Controller Name	Min. Version	Max. Version	Abbrev. Name
Ethernet MAC Universal	N/A	3.73a	GMAC
Ethernet Quality-of-Service	4.00a	N/A	GMAC4+
XGMAC - 10G Ethernet MAC	2.10a	N/A	XGMAC2+
XLGMAC - 100G Ethernet MAC	2.00a	N/A	XLGMAC2+

For questions related to hardware requirements, refer to the documentation supplied with your Ethernet adapter. All hardware requirements listed apply to use with Linux.

Feature List

The following features are available in this driver:

- GMII/MII/RGMII/SGMII/RMII/XGMII/XLGMII Interface
- Half-Duplex / Full-Duplex Operation
- Energy Efficient Ethernet (EEE)
- IEEE 802.3x PAUSE Packets (Flow Control)
- RMON/MIB Counters
- IEEE 1588 Timestamping (PTP)
- Pulse-Per-Second Output (PPS)
- MDIO Clause 22 / Clause 45 Interface
- MAC Loopback
- ARP Offloading
- Automatic CRC / PAD Insertion and Checking
- Checksum Offload for Received and Transmitted Packets
- Standard or Jumbo Ethernet Packets
- Source Address Insertion / Replacement
- VLAN TAG Insertion / Replacement / Deletion / Filtering (HASH and PERFECT)
- Programmable TX and RX Watchdog and Coalesce Settings
- Destination Address Filtering (PERFECT)
- HASH Filtering (Multicast)
- Layer 3 / Layer 4 Filtering
- Remote Wake-Up Detection
- Receive Side Scaling (RSS)
- Frame Preemption for TX and RX
- Programmable Burst Length, Threshold, Queue Size
- Multiple Queues (up to 8)
- Multiple Scheduling Algorithms (TX: WRR, DWRR, WFQ, SP, CBS, EST, TBS; RX: WRR, SP)
- Flexible RX Parser
- TCP / UDP Segmentation Offload (TSO, USO)
- Split Header (SPH)
- Safety Features (ECC Protection, Data Parity Protection)
- Selftests using Ethtool

Kernel Configuration

The kernel configuration option is **CONFIG_STMMAC_ETH**:

- **CONFIG_STMMAC_PLATFORM**: is to enable the platform driver.
- **CONFIG_STMMAC_PCI**: is to enable the pci driver.

Command Line Parameters

If the driver is built as a module the following optional parameters are used by entering them on the command line with the modprobe command using this syntax (e.g. for PCI module):

```
modprobe stmmac_pci [<option>=<VAL1>,<VAL2>,...]
```

Driver parameters can be also passed in command line by using:

```
stmmaceth=watchdog:100,chain_mode=1
```

The default value for each parameter is generally the recommended setting, unless otherwise noted.

watchdog

Valid Range

5000-None

Default Value

5000

This parameter overrides the transmit timeout in milliseconds.

debug

Valid Range

0-16 (0=none,...,16=all)

Default Value

0

This parameter adjusts the level of debug messages displayed in the system logs.

phyaddr

Valid Range

0-31

Default Value

-1

This parameter overrides the physical address of the PHY device.

flow_ctrl

Valid Range

0-3 (0=off,1=rx,2=tx,3=rx/tx)

Default Value

3

This parameter changes the default Flow Control ability.

pause

Valid Range

0-65535

Default Value

65535

This parameter changes the default Flow Control Pause time.

tc

Valid Range

64-256

Default Value

64

This parameter changes the default HW FIFO Threshold control value.

buf_sz

Valid Range

1536-16384

Default Value

1536

This parameter changes the default RX DMA packet buffer size.

eee_timer

Valid Range

0-None

Default Value

1000

This parameter changes the default LPI TX Expiration time in milliseconds.

chain_mode

Valid Range

0-1 (0=off,1=on)

Default Value

0

This parameter changes the default mode of operation from Ring Mode to Chain Mode.

Driver Information and Notes

Transmit Process

The xmit method is invoked when the kernel needs to transmit a packet; it sets the descriptors in the ring and informs the DMA engine that there is a packet ready to be transmitted.

By default, the driver sets the NETIF_F_SG bit in the features field of the net_device structure, enabling the scatter-gather feature. This is true on chips and configurations where the checksum can be done in hardware.

Once the controller has finished transmitting the packet, timer will be scheduled to release the transmit resources.

Receive Process

When one or more packets are received, an interrupt happens. The interrupts are not queued, so the driver has to scan all the descriptors in the ring during the receive process.

This is based on NAPI, so the interrupt handler signals only if there is work to be done, and it exits. Then the poll method will be scheduled at some future point.

The incoming packets are stored, by the DMA, in a list of pre-allocated socket buffers in order to avoid the memcpy (zero-copy).

Interrupt Mitigation

The driver is able to mitigate the number of its DMA interrupts using NAPI for the reception on chips older than the 3.50. New chips have an HW RX Watchdog used for this mitigation.

Mitigation parameters can be tuned by ethtool.

WoL

Wake up on Lan feature through Magic and Unicast frames are supported for the GMAC, GMAC4/5 and XGMAC core.

DMA Descriptors

Driver handles both normal and alternate descriptors. The latter has been only tested on DesignWare(R) Cores Ethernet MAC Universal version 3.41a and later.

stmmac supports DMA descriptor to operate both in dual buffer (RING) and linked-list(CHAINED) mode. In RING each descriptor points to two data buffer pointers whereas in CHAINED mode they point to only one data buffer pointer. RING mode is the default.

In CHAINED mode each descriptor will have pointer to next descriptor in the list, hence creating the explicit chaining in the descriptor itself, whereas such explicit chaining is not possible in RING mode.

Extended Descriptors

The extended descriptors give us information about the Ethernet payload when it is carrying PTP packets or TCP/UDP/ICMP over IP. These are not available on GMAC Synopsys(R) chips older than the 3.50. At probe time the driver will decide if these can be actually used. This support also is mandatory for PTPv2 because the extra descriptors are used for saving the hardware timestamps and Extended Status.

Ethtool Support

Ethtool is supported. For example, driver statistics (including RMON), internal errors can be taken using:

```
ethtool -S ethX
```

Ethtool selftests are also supported. This allows to do some early sanity checks to the HW using MAC and PHY loopback mechanisms:

```
ethtool -t ethX
```

Jumbo and Segmentation Offloading

Jumbo frames are supported and tested for the GMAC. The GSO has been also added but it's performed in software. LRO is not supported.

TSO Support

TSO (TCP Segmentation Offload) feature is supported by GMAC > 4.x and XGMAC chip family. When a packet is sent through TCP protocol, the TCP stack ensures that the SKB provided to the low level driver (stmmac in our case) matches with the maximum frame len (IP header + TCP header + payload \leq 1500 bytes (for MTU set to 1500)). It means that if an application using TCP want to send a packet which will have a length (after adding headers) $>$ 1514 the packet will be split in several TCP packets: The data payload is split and headers (TCP/IP ..) are added. It is done by software.

When TSO is enabled, the TCP stack doesn't care about the maximum frame length and provide SKB packet to stmmac as it is. The GMAC IP will have to perform the segmentation by it self to match with maximum frame length.

This feature can be enabled in device tree through `snp, tso` entry.

Energy Efficient Ethernet

Energy Efficient Ethernet (EEE) enables IEEE 802.3 MAC sublayer along with a family of Physical layer to operate in the Low Power Idle (LPI) mode. The EEE mode supports the IEEE 802.3 MAC operation at 100Mbps, 1000Mbps and 1Gbps.

The LPI mode allows power saving by switching off parts of the communication device functionality when there is no data to be transmitted & received. The system on both the side of the link can disable some functionalities and save power during the period of low-link utilization. The MAC controls whether the system should enter or exit the LPI mode and communicate this to PHY.

As soon as the interface is opened, the driver verifies if the EEE can be supported. This is done by looking at both the DMA HW capability register and the PHY devices MCD registers.

To enter in TX LPI mode the driver needs to have a software timer that enable and disable the LPI mode when there is nothing to be transmitted.

Precision Time Protocol (PTP)

The driver supports the IEEE 1588-2002, Precision Time Protocol (PTP), which enables precise synchronization of clocks in measurement and control systems implemented with technologies such as network communication.

In addition to the basic timestamp features mentioned in IEEE 1588-2002 Timestamps, new GMAC cores support the advanced timestamp features. IEEE 1588-2008 can be enabled when configuring the Kernel.

SGMII/RGMII Support

New GMAC devices provide own way to manage RGMII/SGMII. This information is available at run-time by looking at the HW capability register. This means that the stmmac can manage auto-negotiation and link status w/o using the PHYLIB stuff. In fact, the HW provides a subset of extended registers to restart the ANE, verify Full/Half duplex mode and Speed. Thanks to these registers, it is possible to look at the Auto-negotiated Link Parter Ability.

Physical

The driver is compatible with Physical Abstraction Layer to be connected with PHY and GPHY devices.

Platform Information

Several information can be passed through the platform and device-tree.

```
struct plat_stmmacenet_data {
```

1) Bus identifier:

```
    int bus_id;
```

2) PHY Physical Address. If set to -1 the driver will pick the first PHY it finds:

```
    int phy_addr;
```

3) PHY Device Interface:

```
    int interface;
```

4) Specific platform fields for the MDIO bus:

```
    struct stmmac_mdio_bus_data *mdio_bus_data;
```

5) Internal DMA parameters:

```
    struct stmmac_dma_cfg *dma_cfg;
```

6) Fixed CSR Clock Range selection:

```
    int clk_csr;
```

7) HW uses the GMAC core:

```
    int has_gmac;
```

8) If set the MAC will use Enhanced Descriptors:

```
    int enh_desc;
```

9) Core is able to perform TX Checksum and/or RX Checksum in HW:

```
int tx_coe;
int rx_coe;
```

11) Some HWs are not able to perform the csum in HW for over-sized frames due to limited buffer sizes. Setting this flag the csum will be done in SW on JUMBO frames:

```
int bugged_jumbo;
```

12) Core has the embedded power module:

```
int pmt;
```

13) Force DMA to use the Store and Forward mode or Threshold mode:

```
int force_sf_dma_mode;
int force_thresh_dma_mode;
```

15) Force to disable the RX Watchdog feature and switch to NAPI mode:

```
int riwt_off;
```

16) Limit the maximum operating speed and MTU:

```
int max_speed;
int maxmtu;
```

18) Number of Multicast/Unicast filters:

```
int multicast_filter_bins;
int unicast_filter_entries;
```

20) Limit the maximum TX and RX FIFO size:

```
int tx_fifo_size;
int rx_fifo_size;
```

21) Use the specified number of TX and RX Queues:

```
u32 rx_queues_to_use;
u32 tx_queues_to_use;
```

22) Use the specified TX and RX scheduling algorithm:

```
u8 rx_sched_algorithm;
u8 tx_sched_algorithm;
```

23) Internal TX and RX Queue parameters:

```
struct stmmac_rxq_cfg rx_queues_cfg[MTL_MAX_RX_QUEUES];
struct stmmac_txq_cfg tx_queues_cfg[MTL_MAX_TX_QUEUES];
```

24) This callback is used for modifying some syscfg registers (on ST SoCs) according to the link speed negotiated by the physical layer:

```
void (*fix_mac_speed)(void *priv, unsigned int speed);
```

25) Callbacks used for calling a custom initialization; This is sometimes necessary on some platforms (e.g. ST boxes) where the HW needs to have set some PIO lines or system cfg registers. init/exit callbacks should not use or modify platform data:

```
int (*init)(struct platform_device *pdev, void *priv);
void (*exit)(struct platform_device *pdev, void *priv);
```

26) Perform HW setup of the bus. For example, on some ST platforms this field is used to configure the AMBA bridge to generate more efficient STBus traffic:

```
struct mac_device_info *(*setup)(void *priv);
void *bsp_priv;
```

27) Internal clocks and rates:

```
struct clk *stmmac_clk;
struct clk *pclk;
struct clk *clk_ptp_ref;
unsigned int clk_ptp_rate;
unsigned int clk_ref_rate;
s32 ptp_max_adj;
```

28) Main reset:

```
struct reset_control *stmmac_rst;
```

29) AXI Internal Parameters:

```
struct stmmac_axi *axi;
```

30) HW uses GMAC>4 cores:

```
int has_gmac4;
```

31) HW is sun8i based:

```
bool has_sun8i;
```

32) Enables TSO feature:

```
bool tso_en;
```

33) Enables Receive Side Scaling (RSS) feature:

```
int rss_en;
```

34) MAC Port selection:

```
int mac_port_sel_speed;
```

35) Enables TX LPI Clock Gating:

```
bool en_tx_lpi_clockgating;
```

36) HW uses XGMAC>2.10 cores:

```
int has_xgmac;
```

```
}
```

For MDIO bus data, we have:

```
struct stmmac_mdio_bus_data {
```

1) PHY mask passed when MDIO bus is registered:

```
unsigned int phy_mask;
```

2) List of IRQs, one per PHY:

```
int *irqs;
```

3) If IRQs is NULL, use this for probed PHY:

```
int probed_phy_irq;
```

4) Set to true if PHY needs reset:

```
bool needs_reset;
```

```
}
```

For DMA engine configuration, we have:

```
struct stmmac_dma_cfg {
```

1) Programmable Burst Length (TX and RX):

```
int pbl;
```

2) If set, DMA TX / RX will use this value rather than pbl:

```
int txpbl;
int rxpbl;
```

3) Enable 8xPBL:

```
bool pblx8;
```

4) Enable Fixed or Mixed burst:

```
int fixed_burst;
int mixed_burst;
```

- 5) Enable Address Aligned Beats:

```
bool aal;
```

- 6) Enable Enhanced Addressing (> 32 bits):

```
bool eame;
```

```
}
```

For DMA AXI parameters, we have:

```
struct stmmac_axi {
```

- 1) Enable AXI LPI:

```
bool axi_lpi_en;
bool axi_xit_frm;
```

- 2) Set AXI Write / Read maximum outstanding requests:

```
u32 axi_wr_osr_lmt;
u32 axi_rd_osr_lmt;
```

- 3) Set AXI 4KB bursts:

```
bool axi_kbbe;
```

- 4) Set AXI maximum burst length map:

```
u32 axi_blen[AXI_BLEN];
```

- 5) Set AXI Fixed burst / mixed burst:

```
bool axi_fb;
bool axi_mb;
```

- 6) Set AXI rebuild incr mode:

```
bool axi_rb;
```

```
}
```

For the RX Queues configuration, we have:

```
struct stmmac_rxq_cfg {
```

- 1) Mode to use (DCB or AVB):

```
u8 mode_to_use;
```

2) DMA channel to use:

```
u32 chan;
```

3) Packet routing, if applicable:

```
u8 pkt_route;
```

4) Use priority routing, and priority to route:

```
bool use_prio;  
u32 prio;
```

```
}
```

For the TX Queues configuration, we have:

```
struct stmmac_txq_cfg {
```

1) Queue weight in scheduler:

```
u32 weight;
```

2) Mode to use (DCB or AVB):

```
u8 mode_to_use;
```

3) Credit Base Shaper Parameters:

```
u32 send_slope;  
u32 idle_slope;  
u32 high_credit;  
u32 low_credit;
```

4) Use priority scheduling, and priority:

```
bool use_prio;  
u32 prio;
```

```
}
```

Device Tree Information

Please refer to the following document: Documentation/devicetree/bindings/net/snps,dwmac.yaml

HW Capabilities

Note that, starting from new chips, where it is available the HW capability register, many configurations are discovered at run-time for example to understand if EEE, HW csum, PTP, enhanced descriptor etc are actually available. As strategy adopted in this driver, the information from the HW capability register can replace what has been passed from the platform.

Debug Information

The driver exports many information i.e. internal statistics, debug information, MAC and DMA registers etc.

These can be read in several ways depending on the type of the information actually needed.

For example a user can use the ethtool support to get statistics: e.g. using: `ethtool -S ethX` (that shows the Management counters (MMC) if supported) or sees the MAC/DMA registers: e.g. using: `ethtool -d ethX`

Compiling the Kernel with `CONFIG_DEBUG_FS` the driver will export the following debugfs entries:

- `descriptors_status`: To show the DMA TX/RX descriptor rings
- `dma_cap`: To show the HW Capabilities

Developer can also use the `debug` module parameter to get further debug information (please see: NETIF Msg Level).

Support

If an issue is identified with the released source code on a supported kernel with a supported adapter, email the specific information related to the issue to netdev@vger.kernel.org

6.5.40 Texas Instruments CPSW ethernet driver

Multiqueue & CBS & MQPRIO

The cpsw has 3 CBS shapers for each external ports. This document describes MQPRIO and CBS Qdisc offload configuration for cpsw driver based on examples. It potentially can be used in audio video bridging (AVB) and time sensitive networking (TSN).

The following examples were tested on AM572x EVM and BBB boards.

Test setup

Under consideration two examples with AM572x EVM running cpsw driver in dual_emac mode.
Several prerequisites:

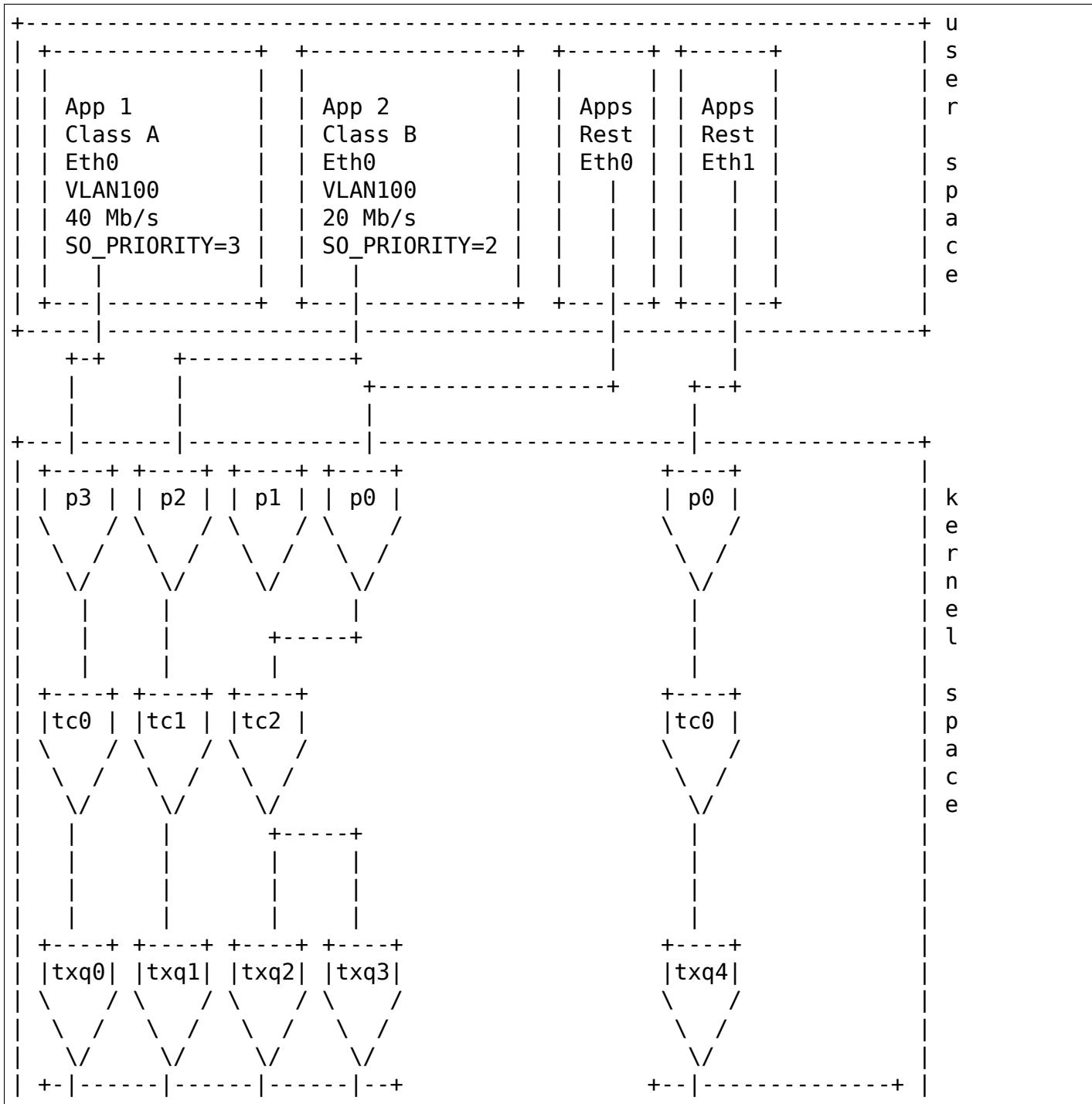
- TX queues must be rated starting from txq0 that has highest priority
 - Traffic classes are used starting from 0, that has highest priority
 - CBS shapers should be used with rated queues
 - The bandwidth for CBS shapers has to be set a little bit more than potential incoming rate, thus, rate of all incoming tx queues has to be a little less
 - Real rates can differ, due to discreetness
 - Map skb-priority to txq is not enough, also skb-priority to l2 prio map has to be created with ip or vconfig tool
 - Any l2/socket prio (0 - 7) for classes can be used, but for simplicity default values are used: 3 and 2
 - only 2 classes tested: A and B, but checked and can work with more, maximum allowed 4, but only for 3 rate can be set.

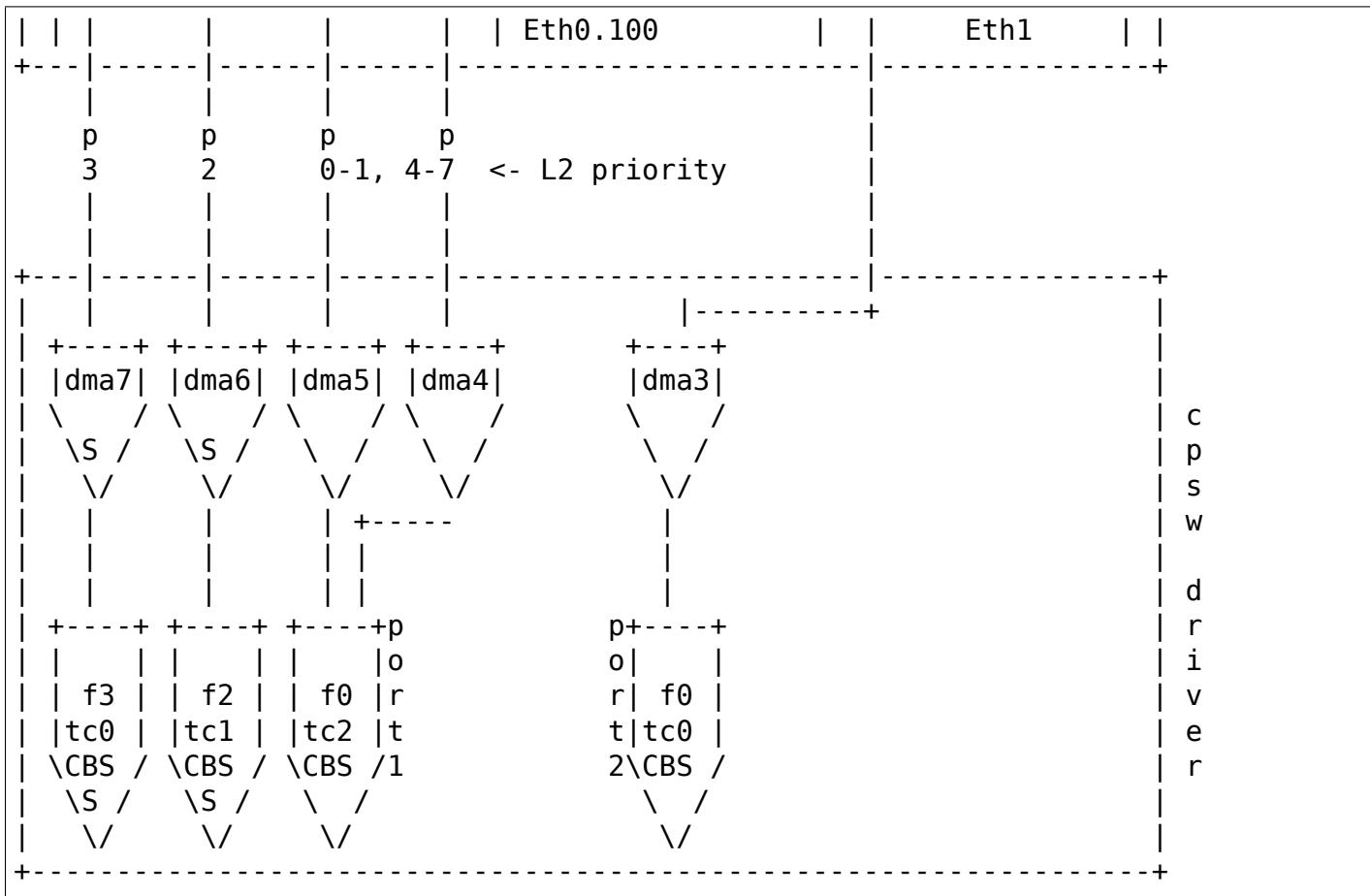
Test setup for examples

Example 1: One port tx AVB configuration scheme for target board

(prints and scheme for AM572x evm, applicable for single port boards)

- tc - traffic class
- txq - transmit queue
- p - priority
- f - fifo (cpsw fifo)
- S - shaper configured





```
1) // Add 4 tx queues, for interface Eth0, and 1 tx queue for Eth1  
$ ethtool -L eth0 rx 1 tx 5  
rx unmodified, ignoring
```

```
2) // Check if num of queues is set correctly:  
$ ethtool -l eth0  
Channel parameters for eth0:  
Pre-set maximums:  
RX:          8  
TX:          8  
Other:        0  
Combined:     0  
Current hardware settings:  
RX:          1  
TX:          5  
Other:        0  
Combined:     0
```

```
3) // TX queues must be rated starting from 0, so set bws for tx0 and tx1  
// Set rates 40 and 20 Mb/s appropriately.  
// Pay attention, real speed can differ a bit due to discreteness.  
// Leave last 2 tx queues not rated.  
$ echo 40 > /sys/class/net/eth0/queues/tx-0/tx_maxrate  
$ echo 20 > /sys/class/net/eth0/queues/tx-1/tx_maxrate
```

```
4) // Check maximum rate of tx (cpdma) queues:  
$ cat /sys/class/net/eth0/queues/tx-*/tx_maxrate  
40  
20  
0  
0  
0
```

```
5) // Map skb->priority to traffic class:  
// 3pri -> tc0, 2pri -> tc1, (0,1,4-7)pri -> tc2  
// Map traffic class to transmit queue:  
// tc0 -> txq0, tc1 -> txq1, tc2 -> (txq2, txq3)  
$ tc qdisc replace dev eth0 handle 100: parent root mqprio num_tc 3 \  
map 2 2 1 0 2 2 2 2 2 2 2 2 2 2 2 2 queues 1@0 1@1 2@2 hw 1
```

5a)

```
// As two interface sharing same set of tx queues, assign all traffic
// coming to interface Eth1 to separate queue in order to not mix it
// with traffic from interface Eth0, so use separate txq to send
// packets to Eth1, so all prio -> tc0 and tc0 -> txq4
// Here hw 0, so here still default configuration for eth1 in hw
$ tc qdisc replace dev eth1 handle 100: parent root mqprio num_tc 1 \
map 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 queues 1@4 hw 0
```

```
6) // Check classes settings
$ tc -g class show dev eth0
+--(100:ffe2) mqprio
|    +--(100:3) mqprio
|    +--(100:4) mqprio
|
+--(100:ffe1) mqprio
|    +--(100:2) mqprio
|
+--(100:ffe0) mqprio
|    +--(100:1) mqprio

$ tc -g class show dev eth1
+--(100:ffe0) mqprio
|    +--(100:5) mqprio
```

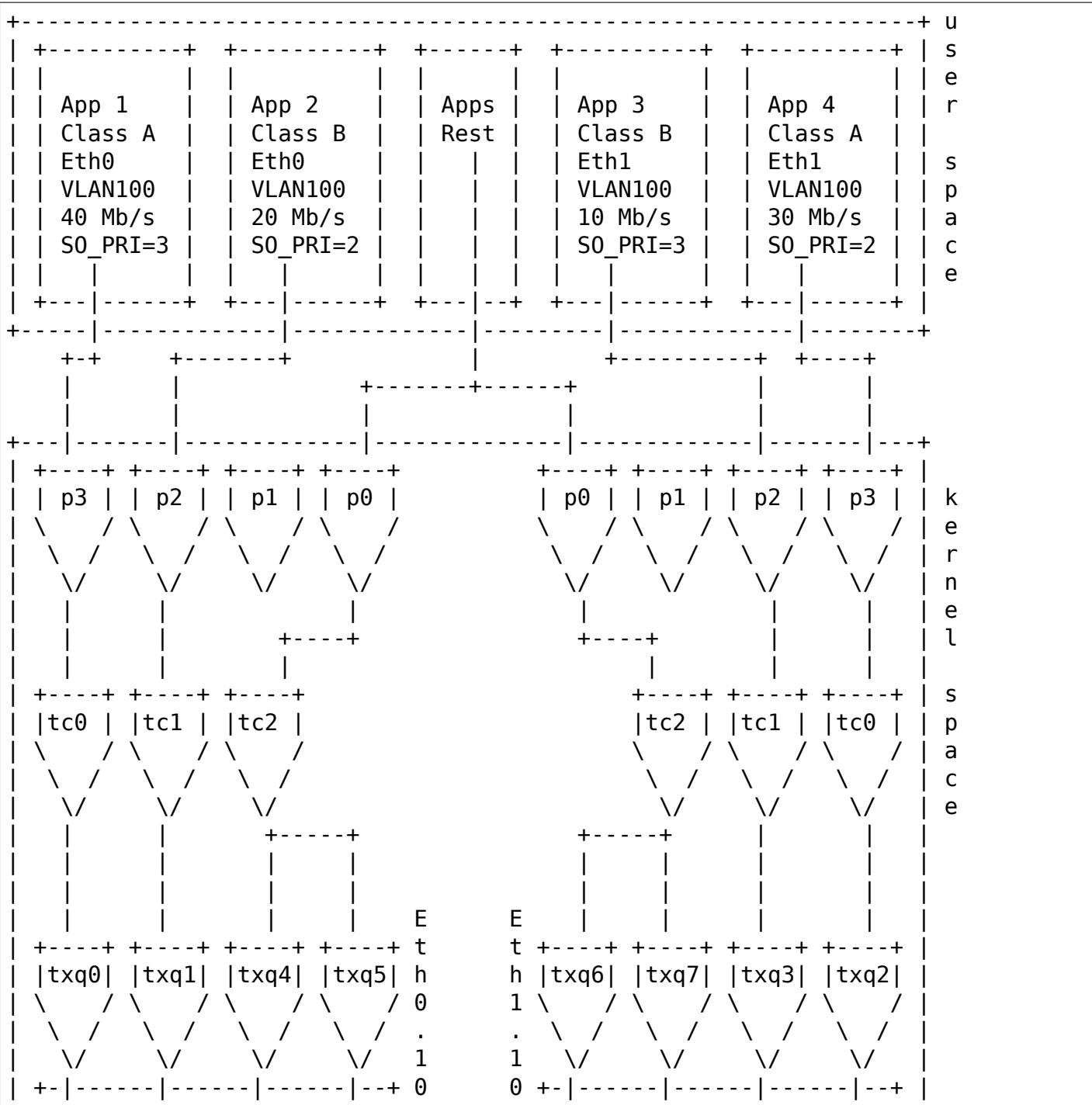
```
7) // Set rate for class A - 41 Mbit (tc0, txq0) using CBS Qdisc
    // Set it +1 Mb for reserve (important!)
    // here only idle slope is important, others arg are ignored
    // Pay attention, real speed can differ a bit due to discreteness
    $ tc qdisc add dev eth0 parent 100:1 cbs locredit -1438 \
        hicredit 62 sendslope -959000 idleslope 41000 offload 1
    net eth0: set FIF03 bw = 50
```

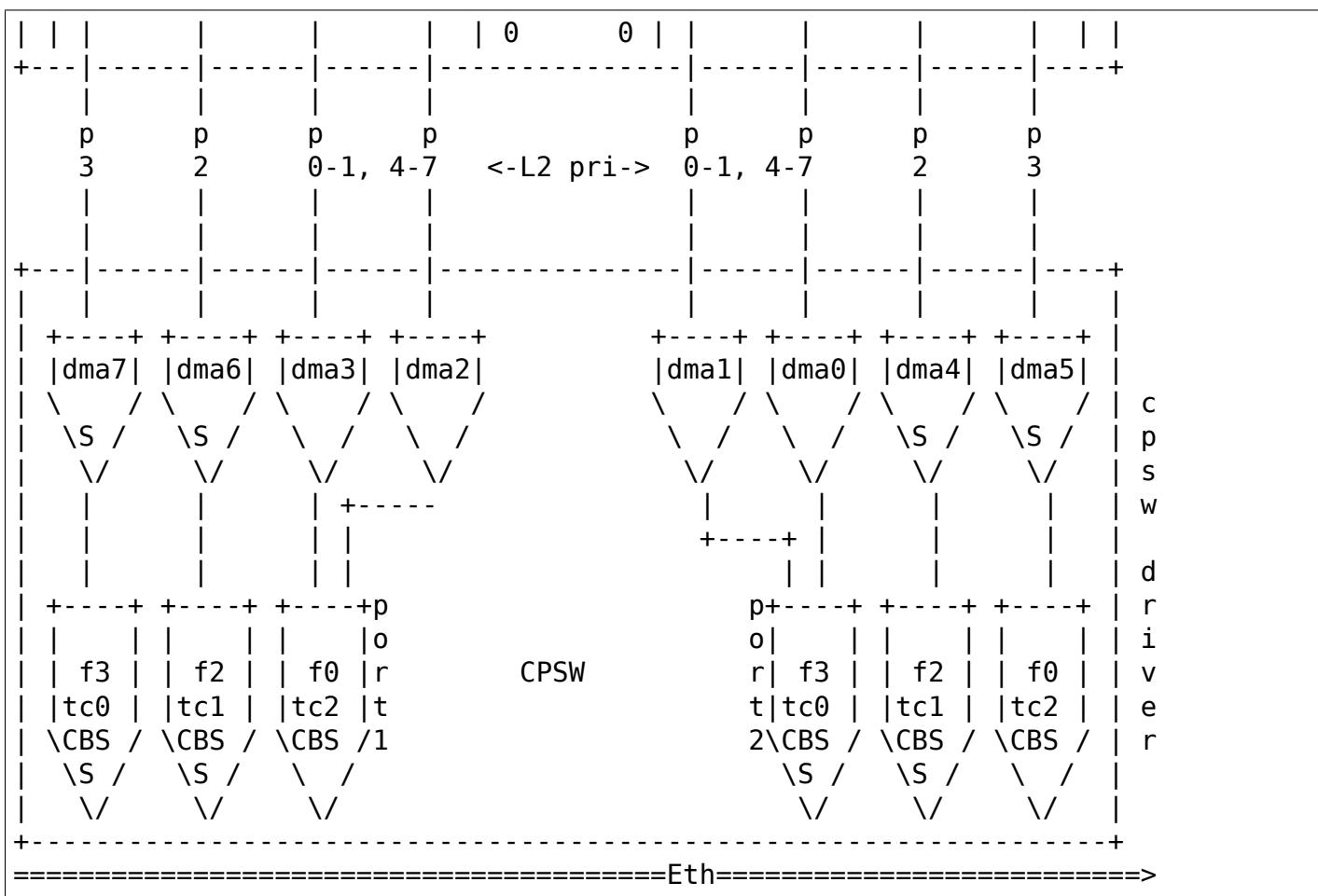
- 8) // Set rate for class B - 21 Mbit (tcl, txql) using CBS Qdisc:
 // Set it +1 Mb for reserve (important!)
 \$ tc qdisc add dev eth0 parent 100:2 cbs locredit -1468 \
 hicredit 65 sendslope -979000 idleslope 21000 offload 1
 net eth0: set FIFO2 bw = 30
- 9) // Create vlan 100 to map sk->priority to vlan qos
 \$ ip link add link eth0 name eth0.100 type vlan id 100
 8021q: 802.1Q VLAN Support v1.8
 8021q: adding VLAN 0 to HW filter on device eth0
 8021q: adding VLAN 0 to HW filter on device eth1
 net eth0: Adding vlanid 100 to vlan filter
- 10) // Map skb->priority to L2 prio, 1 to 1
 \$ ip link set eth0.100 type vlan \
 egress 0:0 1:1 2:2 3:3 4:4 5:5 6:6 7:7
- 11) // Check egress map for vlan 100
 \$ cat /proc/net/vlan/eth0.100
 [...]
 INGRESS priority mappings: 0:0 1:0 2:0 3:0 4:0 5:0 6:0 7:0
 EGRESS priority mappings: 0:0 1:1 2:2 3:3 4:4 5:5 6:6 7:7
- 12) // Run your appropriate tools with socket option "SO_PRIORITY"
 // to 3 for class A and/or to 2 for class B
 // (I took at <https://www.spinics.net/lists/netdev/msg460869.html>)
 ./tsn_talker -d 18:03:73:66:87:42 -i eth0.100 -p3 -s 1500&
 ./tsn_talker -d 18:03:73:66:87:42 -i eth0.100 -p2 -s 1500&
- 13) // run your listener on workstation (should be in same vlan)
 // (I took at <https://www.spinics.net/lists/netdev/msg460869.html>)
 ./tsn_listener -d 18:03:73:66:87:42 -i enp5s0 -s 1500
 Receiving data rate: 39012 kbps
 Receiving data rate: 39000 kbps
- 14) // Restore default configuration if needed
 \$ ip link del eth0.100
 \$ tc qdisc del dev eth1 root

```
$ tc qdisc del dev eth0 root
net eth0: Prev FIF02 is shaped
net eth0: set FIF03 bw = 0
net eth0: set FIF02 bw = 0
$ ethtool -L eth0 rx 1 tx 1
```

Example 2: Two port tx AVB configuration scheme for target board

(prints and scheme for AM572x evm, for dual emac boards only)





- 1) // Add 8 tx queues, for interface Eth0, but they are common, so are accessed
 // by two interfaces Eth0 and Eth1.

```
$ ethtool -L eth1 rx 1 tx 8
rx unmodified, ignoring
```
- 2) // Check if num of queues is set correctly:

```
$ ethtool -l eth0
Channel parameters for eth0:
Pre-set maximums:
RX:          8
TX:          8
Other:        0
Combined:     0
Current hardware settings:
RX:          1
TX:          8
Other:        0
Combined:     0
```
- 3) // TX queues must be rated starting from 0, so set bws for tx0 and tx1 for Eth0
 // and for tx2 and tx3 for Eth1. That is, rates 40 and 20 Mb/s appropriately

```
// TX queues must be rated starting from 0, so set bws for tx0 and tx1 for Eth0
// and for tx2 and tx3 for Eth1. That is, rates 40 and 20 Mb/s appropriately
```

```
// for Eth0 and 30 and 10 Mb/s for Eth1.  
// Real speed can differ a bit due to discreetness  
// Leave last 4 tx queues as not rated  
$ echo 40 > /sys/class/net/eth0/queues/tx-0/tx_maxrate  
$ echo 20 > /sys/class/net/eth0/queues/tx-1/tx_maxrate  
$ echo 30 > /sys/class/net/eth1/queues/tx-2/tx_maxrate  
$ echo 10 > /sys/class/net/eth1/queues/tx-3/tx_maxrate
```

- ```
4) // Check maximum rate of tx (cpdma) queues:
$ cat /sys/class/net/eth0/queues/tx-*/tx_maxrate
40
20
30
10
0
0
0
0
```

- ```
5) // Map skb->priority to traffic class for Eth0:  
// 3pri -> tc0, 2pri -> tc1, (0,1,4-7)pri -> tc2  
// Map traffic class to transmit queue:  
// tc0 -> txq0, tc1 -> txq1, tc2 -> (txq4, txq5)  
$ tc qdisc replace dev eth0 handle 100: parent root mqprio num_tc 3 \  
map 2 2 1 0 2 2 2 2 2 2 2 2 2 2 2 2 queues 1@0 1@1 2@4 hw 1
```

- ```
6) // Check classes settings
$ tc -g class show dev eth0
+--(100:ffe2) mqprio
| +--(100:5) mqprio
| +--(100:6) mqprio
|
+--(100:ffe1) mqprio
| +--(100:2) mqprio
|
+--(100:ffe0) mqprio
 +--(100:1) mqprio
```

- ```
7) // Set rate for class A - 41 Mbit (tc0, txq0) using CBS Qdisc for Eth0  
// here only idle slope is important, others ignored  
// Real speed can differ a bit due to discreetness  
$ tc qdisc add dev eth0 parent 100:1 cbs lopcredit -1470 \  
hicredit 62 sendslope -959000 idleslope 41000 offload 1  
net eth0: set FIFO3 bw = 50
```

- ```
8) // Set rate for class B - 21 Mbit (tc1, txq1) using CBS Qdisc for Eth0
$ tc qdisc add dev eth0 parent 100:2 cbs lcredit -1470 \
hicredit 65 sendslope -979000 idleslope 21000 offload 1
net eth0: set FIF02 bw = 30
```

- 9) // Create vlan 100 to map sk->priority to vlan qos for Eth0  
\$ ip link add link eth0 name eth0.100 type vlan id 100  
net eth0: Adding vlanid 100 to vlan filter
- 10) // Map skb->priority to L2 prio for Eth0.100, one to one  
\$ ip link set eth0.100 type vlan \  
egress 0:0 1:1 2:2 3:3 4:4 5:5 6:6 7:7
- 11) // Check egress map for vlan 100  
\$ cat /proc/net/vlan/eth0.100  
[...]  
INGRESS priority mappings: 0:0 1:0 2:0 3:0 4:0 5:0 6:0 7:0  
EGRESS priority mappings: 0:0 1:1 2:2 3:3 4:4 5:5 6:6 7:7
- 12) // Map skb->priority to traffic class for Eth1:  
// 3pri -> tc0, 2pri -> tc1, (0,1,4-7)pri -> tc2  
// Map traffic class to transmit queue:  
// tc0 -> txq2, tc1 -> txq3, tc2 -> (txq6, txq7)  
\$ tc qdisc replace dev eth1 handle 100: parent root mqpprio num\_tc 3 \  
map 2 2 1 0 2 2 2 2 2 2 2 2 2 2 2 2 queues 1@2 1@3 2@6 hw 1
- 13) // Check classes settings  
\$ tc -g class show dev eth1  
+---(100:ffe2) mqpprio  
| +---(100:7) mqpprio  
| +---(100:8) mqpprio  
|  
+---(100:ffe1) mqpprio  
| +---(100:4) mqpprio  
|  
+---(100:ffe0) mqpprio  
+---(100:3) mqpprio
- 14) // Set rate for class A - 31 Mbit (tc0, txq2) using CBS Qdisc for Eth1  
// here only idle slope is important, others ignored, but calculated  
// for interface speed - 100Mb for eth1 port.  
// Set it +1 Mb for reserve (important!)  
\$ tc qdisc add dev eth1 parent 100:3 cbs locredit -1035 \  
hicredit 465 sendslope -69000 idleslope 31000 offload 1  
net eth1: set FIFO3 bw = 31
- 15) // Set rate for class B - 11 Mbit (tc1, txq3) using CBS Qdisc for Eth1  
// Set it +1 Mb for reserve (important!)  
\$ tc qdisc add dev eth1 parent 100:4 cbs locredit -1335 \  
hicredit 405 sendslope -89000 idleslope 11000 offload 1  
net eth1: set FIFO2 bw = 11
- 16) // Create vlan 100 to map sk->priority to vlan qos for Eth1  
\$ ip link add link eth1 name eth1.100 type vlan id 100

- ```
net eth1: Adding vlanid 100 to vlan filter
```
- 17) // Map skb->priority to L2 prio for Eth1.100, one to one
\$ ip link set eth1.100 type vlan \
egress 0:0 1:1 2:2 3:3 4:4 5:5 6:6 7:7
- 18) // Check egress map for vlan 100
\$ cat /proc/net/vlan/eth1.100
[...]
INGRESS priority mappings: 0:0 1:0 2:0 3:0 4:0 5:0 6:0 7:0
EGRESS priority mappings: 0:0 1:1 2:2 3:3 4:4 5:5 6:6 7:7
- 19) // Run appropriate tools with socket option "SO_PRIORITY" to 3
// for class A and to 2 for class B. For both interfaces
./tsn_talker -d 18:03:73:66:87:42 -i eth0.100 -p2 -s 1500&
./tsn_talker -d 18:03:73:66:87:42 -i eth0.100 -p3 -s 1500&
./tsn_talker -d 20:cf:30:85:7d:fd -i eth1.100 -p2 -s 1500&
./tsn_talker -d 20:cf:30:85:7d:fd -i eth1.100 -p3 -s 1500&
- 20) // run your listener on workstation (should be in same vlan)
// (I took at <https://www.spinics.net/lists/netdev/msg460869.html>)
./tsn_listener -d 18:03:73:66:87:42 -i enp5s0 -s 1500
Receiving data rate: 39012 kbps
Receiving data rate: 39000 kbps
- 21) // Restore default configuration if needed
\$ ip link del eth1.100
\$ ip link del eth0.100
\$ tc qdisc del dev eth1 root
net eth1: Prev FIF02 is shaped
net eth1: set FIF03 bw = 0
net eth1: set FIF02 bw = 0
\$ tc qdisc del dev eth0 root
net eth0: Prev FIF02 is shaped
net eth0: set FIF03 bw = 0
net eth0: set FIF02 bw = 0
\$ ethtool -L eth0 rx 1 tx 1

6.5.41 Texas Instruments CPSW switchdev based ethernet driver

Version

2.0

Port renaming

On older udev versions renaming of ethX to swXpY will not be automatically supported

In order to rename via udev:

```
ip -d link show dev sw0p1 | grep switchid
SUBSYSTEM=="net", ACTION=="add", ATTR{phys_switch_id}==<switchid>, \
ATTR{phys_port_name}!="", NAME="sw0$attr{phys_port_name}"
```

Dual mac mode

- The new (cpsw_new.c) driver is operating in dual-emac mode by default, thus working as 2 individual network interfaces. Main differences from legacy CPSW driver are:
- optimized promiscuous mode: The P0_UNI_FLOOD (both ports) is enabled in addition to ALLMULTI (current port) instead of ALE_BYPASS. So, Ports in promiscuous mode will keep possibility of mcast and vlan filtering, which provides significant benefits when ports are joined to the same bridge, but without enabling "switch" mode, or to different bridges.
- learning disabled on ports as it makes not too much sense for segregated ports - no forwarding in HW.
- enabled basic support for devlink.

```
devlink dev show
platform/48484000.switch

devlink dev param show
platform/48484000.switch:
name switch_mode type driver-specific
values:
    cmode runtime value false
name ale_bypass type driver-specific
values:
    cmode runtime value false
```

Devlink configuration parameters

See [ti-cpsw-switch devlink support](#)

Bridging in dual mac mode

The dual_mac mode requires two vids to be reserved for internal purposes, which, by default, equal CPSW Port numbers. As result, bridge has to be configured in vlan unaware mode or default_pvid has to be adjusted:

```
ip link add name br0 type bridge
ip link set dev br0 type bridge vlan_filtering 0
echo 0 > /sys/class/net/br0/bridge/default_pvid
ip link set dev sw0p1 master br0
ip link set dev sw0p2 master br0
```

or:

```
ip link add name br0 type bridge
ip link set dev br0 type bridge vlan_filtering 0
echo 100 > /sys/class/net/br0/bridge/default_pvid
ip link set dev br0 type bridge vlan_filtering 1
ip link set dev sw0p1 master br0
ip link set dev sw0p2 master br0
```

Enabling "switch"

The Switch mode can be enabled by configuring devlink driver parameter "switch_mode" to 1/true:

```
devlink dev param set platform/48484000.switch \
name switch_mode value 1 cmode runtime
```

This can be done regardless of the state of Port's netdev devices - UP/DOWN, but Port's netdev devices have to be in UP before joining to the bridge to avoid overwriting of bridge configuration as CPSW switch driver copletly reloads its configuration when first Port changes its state to UP.

When the both interfaces joined the bridge - CPSW switch driver will enable marking packets with offload_fwd_mark flag unless "ale_bypass=0"

All configuration is implemented via switchdev API.

Bridge setup

```
devlink dev param set platform/48484000.switch \
name switch_mode value 1 cmode runtime

ip link add name br0 type bridge
ip link set dev br0 type bridge ageing_time 1000
ip link set dev sw0p1 up
ip link set dev sw0p2 up
ip link set dev sw0p1 master br0
ip link set dev sw0p2 master br0

[*] bridge vlan add dev br0 vid 1 pvid untagged self
[*] if vlan_filtering=1. where default_pvid=1
```

Note. Steps [*] are mandatory.

On/off STP

```
ip link set dev BRDEV type bridge stp_state 1/0
```

VLAN configuration

```
bridge vlan add dev br0 vid 1 pvid untagged self <---- add cpu port to VLAN 1
```

Note. This step is mandatory for bridge/default_pvid.

Add extra VLANs

1. untagged:

```
bridge vlan add dev sw0p1 vid 100 pvid untagged master
bridge vlan add dev sw0p2 vid 100 pvid untagged master
bridge vlan add dev br0 vid 100 pvid untagged self <---- Add cpu port to
  ↘ VLAN100
```

2. tagged:

```
bridge vlan add dev sw0p1 vid 100 master
bridge vlan add dev sw0p2 vid 100 master
bridge vlan add dev br0 vid 100 pvid tagged self <---- Add cpu port to
  ↘ VLAN100
```

FDBs

FDBs are automatically added on the appropriate switch port upon detection

Manually adding FDBs:

```
bridge fdb add aa:bb:cc:dd:ee:ff dev sw0p1 master vlan 100
bridge fdb add aa:bb:cc:dd:ee:fe dev sw0p2 master <---- Add on all VLANs
```

MDBs

MDBs are automatically added on the appropriate switch port upon detection

Manually adding MDBs:

```
bridge mdb add dev br0 port sw0p1 grp 239.1.1.1 permanent vid 100
bridge mdb add dev br0 port sw0p1 grp 239.1.1.1 permanent <---- Add on all
  ↳ VLANs
```

Multicast flooding

CPU port mcast_flooding is always on

Turning flooding on/off on switch ports: bridge link set dev sw0p1 mcast_flood on/off

Access and Trunk port

```
bridge vlan add dev sw0p1 vid 100 pvid untagged master
bridge vlan add dev sw0p2 vid 100 master
```

```
bridge vlan add dev br0 vid 100 self
ip link add link br0 name br0.100 type vlan id 100
```

Note. Setting PVID on Bridge device itself working only for default VLAN (default_pvid).

NFS

The only way for NFS to work is by chrooting to a minimal environment when switch configuration that will affect connectivity is needed. Assuming you are booting NFS with eth1 interface(the script is hacky and it's just there to prove NFS is doable).

setup.sh:

```
#!/bin/sh
mkdir proc
mount -t proc none /proc
ifconfig br0 > /dev/null
if [ $? -ne 0 ]; then
```

```

echo "Setting up bridge"
ip link add name br0 type bridge
ip link set dev br0 type bridge ageing_time 1000
ip link set dev br0 type bridge vlan_filtering 1

ip link set eth1 down
ip link set eth1 name sw0p1
ip link set dev sw0p1 up
ip link set dev sw0p2 up
ip link set dev sw0p2 master br0
ip link set dev sw0p1 master br0
bridge vlan add dev br0 vid 1 pvid untagged self
ifconfig sw0p1 0.0.0.0
udhcpc -i br0
fi
umount /proc

```

run_nfs.sh::

```

#!/bin/sh
mkdir /tmp/root/bin -p
mkdir /tmp/root/lib -p

cp -r /lib/ /tmp/root/
cp -r /bin/ /tmp/root/
cp /sbin/ip /tmp/root/bin
cp /sbin/bridge /tmp/root/bin
cp /sbin/ifconfig /tmp/root/bin
cp /sbin/udhcpc /tmp/root/bin
cp /path/to/setup.sh /tmp/root/bin
chroot /tmp/root/ busybox sh /bin/setup.sh

run ./run_nfs.sh

```

6.5.42 Texas Instruments K3 AM65 CPSW NUSS switchdev based ethernet driver

Version
1.0

Port renaming

In order to rename via udev:

```
ip -d link show dev sw0p1 | grep switchid  
  
SUBSYSTEM=="net", ACTION=="add", ATTR{phys_switch_id}==<switchid>, \  
ATTR{phys_port_name}!="", NAME="sw0$attr{phys_port_name}"
```

Multi mac mode

- The driver is operating in multi-mac mode by default, thus working as N individual network interfaces.

Devlink configuration parameters

See [am65-cpsw-nuss devlink support](#)

Enabling "switch"

The Switch mode can be enabled by configuring devlink driver parameter "switch_mode" to 1/true:

```
devlink dev param set platform/c000000.ethernet \  
name switch_mode value true cmode runtime
```

This can be done regardless of the state of Port's netdev devices - UP/DOWN, but Port's netdev devices have to be in UP before joining to the bridge to avoid overwriting of bridge configuration as CPSW switch driver completely reloads its configuration when first port changes its state to UP.

When the both interfaces joined the bridge - CPSW switch driver will enable marking packets with offload_fwd_mark flag.

All configuration is implemented via switchdev API.

Bridge setup

```
devlink dev param set platform/c000000.ethernet \  
name switch_mode value true cmode runtime  
  
ip link add name br0 type bridge  
ip link set dev br0 type bridge ageing_time 1000  
ip link set dev sw0p1 up  
ip link set dev sw0p2 up  
ip link set dev sw0p1 master br0  
ip link set dev sw0p2 master br0  
  
[*] bridge vlan add dev br0 vid 1 pvid untagged self
```

```
[*] if vlan_filtering=1. where default_pvid=1
```

Note. Steps [*] are mandatory.

On/off STP

```
ip link set dev BRDEV type bridge stp_state 1/0
```

VLAN configuration

```
bridge vlan add dev br0 vid 1 pvid untagged self <---- add cpu port to VLAN 1
```

Note. This step is mandatory for bridge/default_pvid.

Add extra VLANs

1. untagged:

```
bridge vlan add dev sw0p1 vid 100 pvid untagged master
bridge vlan add dev sw0p2 vid 100 pvid untagged master
bridge vlan add dev br0 vid 100 pvid untagged self <---- Add cpu port to VLAN100
```

2. tagged:

```
bridge vlan add dev sw0p1 vid 100 master
bridge vlan add dev sw0p2 vid 100 master
bridge vlan add dev br0 vid 100 pvid tagged self <---- Add cpu port to VLAN100
```

FDBs

FDBs are automatically added on the appropriate switch port upon detection

Manually adding FDBs:

```
bridge fdb add aa:bb:cc:dd:ee:ff dev sw0p1 master vlan 100
bridge fdb add aa:bb:cc:dd:ee:fe dev sw0p2 master <---- Add on all VLANs
```

MDBs

MDBs are automatically added on the appropriate switch port upon detection

Manually adding MDBs:

```
bridge mdb add dev br0 port sw0p1 grp 239.1.1.1 permanent vid 100
bridge mdb add dev br0 port sw0p1 grp 239.1.1.1 permanent <---- Add on all
    ↳ VLANs
```

Multicast flooding

CPU port mcast_flooding is always on

Turning flooding on/off on switch ports: bridge link set dev sw0p1 mcast_flood on/off

Access and Trunk port

```
bridge vlan add dev sw0p1 vid 100 pvid untagged master
bridge vlan add dev sw0p2 vid 100 master
```

```
bridge vlan add dev br0 vid 100 self
ip link add link br0 name br0.100 type vlan id 100
```

Note. Setting PVID on Bridge device itself works only for default VLAN (default_pvid).

6.5.43 TLAN driver for Linux

Version

1.14a

(C) 1997-1998 Caldera, Inc.

(C) 1998 James Banks

(C) 1999-2001 Torben Mathiasen <tmm@image.dk, torben.mathiasen@compaq.com>

For driver information/updates visit <http://www.compaq.com>

I. Supported Devices

Only PCI devices will work with this driver.

Supported:

Vendor ID	Device ID	Name
0e11	ae32	Compaq Netelligent 10/100 TX PCI UTP
0e11	ae34	Compaq Netelligent 10 T PCI UTP
0e11	ae35	Compaq Integrated NetFlex 3/P
0e11	ae40	Compaq Netelligent Dual 10/100 TX PCI UTP
0e11	ae43	Compaq Netelligent Integrated 10/100 TX UTP
0e11	b011	Compaq Netelligent 10/100 TX Embedded UTP
0e11	b012	Compaq Netelligent 10 T/2 PCI UTP/Coax
0e11	b030	Compaq Netelligent 10/100 TX UTP
0e11	f130	Compaq NetFlex 3/P
0e11	f150	Compaq NetFlex 3/P
108d	0012	Olicom OC-2325
108d	0013	Olicom OC-2183
108d	0014	Olicom OC-2326

Caveats:

I am not sure if 100BaseTX daughterboards (for those cards which support such things) will work. I haven't had any solid evidence either way.

However, if a card supports 100BaseTx without requiring an add on daughterboard, it should work with 100BaseTx.

The "Netelligent 10 T/2 PCI UTP/Coax" (b012) device is untested, but I do not expect any problems.

II. Driver Options

1. You can append debug=x to the end of the insmod line to get debug messages, where x is a bit field where the bits mean the following:

0x01	Turn on general debugging messages.
0x02	Turn on receive debugging messages.
0x04	Turn on transmit debugging messages.
0x08	Turn on list debugging messages.

2. You can append aui=1 to the end of the insmod line to cause the adapter to use the AUI interface instead of the 10 Base T interface. This is also what to do if you want to use the BNC connector on a TLAN based device. (Setting this option on a device that does not have an AUI/BNC connector will probably cause it to not function correctly.)
3. You can set duplex=1 to force half duplex, and duplex=2 to force full duplex.
4. You can set speed=10 to force 10Mbs operation, and speed=100 to force 100Mbs operation. (I'm not sure what will happen if a card which only supports 10Mbs is forced into 100Mbs mode.)
5. You have to use speed=X duplex=Y together now. If you just do "insmod tlan.o speed=100" the driver will do Auto-Neg. To force a 10Mbps Half-Duplex link do "insmod tlan.o speed=10 duplex=1".

6. If the driver is built into the kernel, you can use the 3rd and 4th parameters to set aui and debug respectively. For example:

```
ether=0,0,0x1,0x7,eth0
```

This sets aui to 0x1 and debug to 0x7, assuming eth0 is a supported TLAN device.

The bits in the third byte are assigned as follows:

0x01	aui
0x02	use half duplex
0x04	use full duplex
0x08	use 10BaseT
0x10	use 100BaseTx

You also need to set both speed and duplex settings when forcing speeds with kernel-parameters. ether=0,0,0x12,0,eth0 will force link to 100Mbps Half-Duplex.

7. If you have more than one tlan adapter in your system, you can use the above options on a per adapter basis. To force a 100Mbit/HD link with your eth1 adapter use:

```
insmod tlan speed=0,100 duplex=0,1
```

Now eth0 will use auto-neg and eth1 will be forced to 100Mbit/HD. Note that the tlan driver supports a maximum of 8 adapters.

III. Things to try if you have problems

1. Make sure your card's PCI id is among those listed in section I, above.
2. Make sure routing is correct.
3. Try forcing different speed/duplex settings

There is also a tlan mailing list which you can join by sending "subscribe tlan" in the body of an email to majordomo@vuser.vu.union.edu.

There is also a tlan website at <http://www.compaq.com>

6.5.44 The Spidernet Device Driver

Written by Linas Vepstas <linas@austin.ibm.com>

Version of 7 June 2007

Abstract

This document sketches the structure of portions of the spidernet device driver in the Linux kernel tree. The spidernet is a gigabit ethernet device built into the Toshiba southbridge commonly used in the SONY Playstation 3 and the IBM QS20 Cell blade.

The Structure of the RX Ring.

The receive (RX) ring is a circular linked list of RX descriptors, together with three pointers into the ring that are used to manage its contents.

The elements of the ring are called "descriptors" or "descrs"; they describe the received data. This includes a pointer to a buffer containing the received data, the buffer size, and various status bits.

There are three primary states that a descriptor can be in: "empty", "full" and "not-in-use". An "empty" or "ready" descriptor is ready to receive data from the hardware. A "full" descriptor has data in it, and is waiting to be emptied and processed by the OS. A "not-in-use" descriptor is neither empty or full; it is simply not ready. It may not even have a data buffer in it, or is otherwise unusable.

During normal operation, on device startup, the OS (specifically, the spidernet device driver) allocates a set of RX descriptors and RX buffers. These are all marked "empty", ready to receive data. This ring is handed off to the hardware, which sequentially fills in the buffers, and marks them "full". The OS follows up, taking the full buffers, processing them, and re-marking them empty.

This filling and emptying is managed by three pointers, the "head" and "tail" pointers, managed by the OS, and a hardware current descriptor pointer (GDACTDPA). The GDACTDPA points at the descr currently being filled. When this descr is filled, the hardware marks it full, and advances the GDACTDPA by one. Thus, when there is flowing RX traffic, every descr behind it should be marked "full", and everything in front of it should be "empty". If the hardware discovers that the current descr is not empty, it will signal an interrupt, and halt processing.

The tail pointer tails or trails the hardware pointer. When the hardware is ahead, the tail pointer will be pointing at a "full" descr. The OS will process this descr, and then mark it "not-in-use", and advance the tail pointer. Thus, when there is flowing RX traffic, all of the descrs in front of the tail pointer should be "full", and all of those behind it should be "not-in-use". When RX traffic is not flowing, then the tail pointer can catch up to the hardware pointer. The OS will then note that the current tail is "empty", and halt processing.

The head pointer (somewhat mis-named) follows after the tail pointer. When traffic is flowing, then the head pointer will be pointing at a "not-in-use" descr. The OS will perform various housekeeping duties on this descr. This includes allocating a new data buffer and dma-mapping it so as to make it visible to the hardware. The OS will then mark the descr as "empty", ready to receive data. Thus, when there is flowing RX traffic, everything in front of the head pointer should be "not-in-use", and everything behind it should be "empty". If no RX traffic is flowing, then the head pointer can catch up to the tail pointer, at which point the OS will notice that the head descr is "empty", and it will halt processing.

Thus, in an idle system, the GDACTDPA, tail and head pointers will all be pointing at the same descr, which should be "empty". All of the other descrs in the ring should be "empty" as well.

The `show_rx_chain()` routine will print out the locations of the GDACTDPA, tail and head pointers. It will also summarize the contents of the ring, starting at the tail pointer, and listing the

status of the descrs that follow.

A typical example of the output, for a nearly idle system, might be:

```
net eth1: Total number of descrs=256
net eth1: Chain tail located at descr=20
net eth1: Chain head is at 20
net eth1: HW curr desc (GDACTDPA) is at 21
net eth1: Have 1 descrs with stat=x40800101
net eth1: HW next desc (GDACNEXTDA) is at 22
net eth1: Last 255 descrs with stat=xa0800000
```

In the above, the hardware has filled in one descr, number 20. Both head and tail are pointing at 20, because it has not yet been emptied. Meanwhile, hw is pointing at 21, which is free.

The "Have nnn decrs" refers to the descr starting at the tail: in this case, nnn=1 descr, starting at descr 20. The "Last nnn descrs" refers to all of the rest of the descrs, from the last status change. The "nnn" is a count of how many descrs have exactly the same status.

The status x4... corresponds to "full" and status xa... corresponds to "empty". The actual value printed is RXCOMST_A.

In the device driver source code, a different set of names are used for these same concepts, so that:

```
"empty" == SPIDER_NET_DESCR_CARDOWNED == 0xa
"full"  == SPIDER_NET_DESCR_FRAME_END == 0x4
"not in use" == SPIDER_NET_DESCR_NOT_IN_USE == 0xf
```

The RX RAM full bug/feature

As long as the OS can empty out the RX buffers at a rate faster than the hardware can fill them, there is no problem. If, for some reason, the OS fails to empty the RX ring fast enough, the hardware GDACTDPA pointer will catch up to the head, notice the not-empty condition, and stop. However, RX packets may still continue arriving on the wire. The spidernet chip can save some limited number of these in local RAM. When this local ram fills up, the spider chip will issue an interrupt indicating this (GHIINT0STS will show ERRINT, and the GRMFLLINT bit will be set in GHIINT1STS). When the RX ram full condition occurs, a certain bug/feature is triggered that has to be specially handled. This section describes the special handling for this condition.

When the OS finally has a chance to run, it will empty out the RX ring. In particular, it will clear the descriptor on which the hardware had stopped. However, once the hardware has decided that a certain descriptor is invalid, it will not restart at that descriptor; instead it will restart at the next descr. This potentially will lead to a deadlock condition, as the tail pointer will be pointing at this descr, which, from the OS point of view, is empty; the OS will be waiting for this descr to be filled. However, the hardware has skipped this descr, and is filling the next descrs. Since the OS doesn't see this, there is a potential deadlock, with the OS waiting for one descr to fill, while the hardware is waiting for a different set of descrs to become empty.

A call to show_rx_chain() at this point indicates the nature of the problem. A typical print when the network is hung shows the following:

```
net eth1: Spider RX RAM full, incoming packets might be discarded!
net eth1: Total number of descrs=256
net eth1: Chain tail located at descr=255
net eth1: Chain head is at 255
net eth1: HW curr desc (GDACTDPA) is at 0
net eth1: Have 1 descrs with stat=xa0800000
net eth1: HW next desc (GDACNEXTDA) is at 1
net eth1: Have 127 descrs with stat=x40800101
net eth1: Have 1 descrs with stat=x40800001
net eth1: Have 126 descrs with stat=x40800101
net eth1: Last 1 descrs with stat=xa0800000
```

Both the tail and head pointers are pointing at descr 255, which is marked xa... which is "empty". Thus, from the OS point of view, there is nothing to be done. In particular, there is the implicit assumption that everything in front of the "empty" descr must surely also be empty, as explained in the last section. The OS is waiting for descr 255 to become non-empty, which, in this case, will never happen.

The HW pointer is at descr 0. This descr is marked 0x4.. or "full". Since its already full, the hardware can do nothing more, and thus has halted processing. Notice that descrs 0 through 254 are all marked "full", while descr 254 and 255 are empty. (The "Last 1 descrs" is descr 254, since tail was at 255.) Thus, the system is deadlocked, and there can be no forward progress; the OS thinks there's nothing to do, and the hardware has nowhere to put incoming data.

This bug/feature is worked around with the `spider_net_resync_head_ptr()` routine. When the driver receives RX interrupts, but an examination of the RX chain seems to show it is empty, then it is probable that the hardware has skipped a descr or two (sometimes dozens under heavy network conditions). The `spider_net_resync_head_ptr()` subroutine will search the ring for the next full descr, and the driver will resume operations there. Since this will leave "holes" in the ring, there is also a `spider_net_resync_tail_ptr()` that will skip over such holes.

As of this writing, the `spider_net_resync()` strategy seems to work very well, even under heavy network loads.

The TX ring

The TX ring uses a low-watermark interrupt scheme to make sure that the TX queue is appropriately serviced for large packet sizes.

For packet sizes greater than about 1KBytes, the kernel can fill the TX ring quicker than the device can drain it. Once the ring is full, the netdev is stopped. When there is room in the ring, the netdev needs to be reawakened, so that more TX packets are placed in the ring. The hardware can empty the ring about four times per jiffy, so its not appropriate to wait for the poll routine to refill, since the poll routine runs only once per jiffy. The low-watermark mechanism marks a descr about 1/4th of the way from the bottom of the queue, so that an interrupt is generated when the descr is processed. This interrupt wakes up the netdev, which can then refill the queue. For large packets, this mechanism generates a relatively small number of interrupts, about 1K/sec. For smaller packets, this will drop to zero interrupts, as the hardware can empty the queue faster than the kernel can fill it.

6.5.45 Linux Base Driver for WangXun(R) 10 Gigabit PCI Express Adapters

WangXun 10 Gigabit Linux driver. Copyright (c) 2015 - 2022 Beijing WangXun Technology Co., Ltd.

Contents

- Support

Support

If you got any problem, contact Wangxun support team via nic-support@net-swift.com and Cc: netdev.

6.5.46 Linux Base Driver for WangXun(R) Gigabit PCI Express Adapters

WangXun Gigabit Linux driver. Copyright (c) 2019 - 2022 Beijing WangXun Technology Co., Ltd.

Support

If you have problems with the software or hardware, please contact our customer support team via email at nic-support@net-swift.com or check our website at <https://www.net-swift.com>

6.6 Fiber Distributed Data Interface (FDDI) Device Drivers

Contents:

6.6.1 Notes on the DEC FDDIcontroller 700 (DEFZA-xx) driver

Version

v.1.1.4

DEC FDDIcontroller 700 is DEC's first-generation TURBOchannel FDDI network card, designed in 1990 specifically for the DECstation 5000 model 200 workstation. The board is a single attachment station and it was manufactured in two variations, both of which are supported.

First is the SAS MMF DEFZA-AA option, the original design implementing the standard MMF-PMD, however with a pair of ST connectors rather than the usual MIC connector. The other one is the SAS ThinWire/STP DEFZA-CA option, denoted 700-C, with the network medium selectable by a switch between the DEC proprietary ThinWire-PMD using a BNC connector and the standard STP-PMD using a DE-9F connector. This option can interface to a DECconcentrator 500 device and, in the case of the STP-PMD, also other FDDI equipment and was designed to make it easier to transition from existing IEEE 802.3 10BASE2 Ethernet and IEEE 802.5 Token Ring networks by providing means to reuse existing cabling.

This driver handles any number of cards installed in a single system. They get fddi0, fddi1, etc. interface names assigned in the order of increasing TURBOchannel slot numbers.

The board only supports DMA on the receive side. Transmission involves the use of PIO. As a result under a heavy transmission load there will be a significant impact on system performance.

The board supports a 64-entry CAM for matching destination addresses. Two entries are pre-occupied by the Directed Beacon and Ring Purger multicast addresses and the rest is used as a multicast filter. An all-multi mode is also supported for LLC frames and it is used if requested explicitly or if the CAM overflows. The promiscuous mode supports separate enables for LLC and SMT frames, but this driver doesn't support changing them individually.

Known problems:

None.

To do:

5. MAC address change. The card does not support changing the Media Access Controller's address registers but a similar effect can be achieved by adding an alias to the CAM. There is no way to disable matching against the original address though.
7. Queueing incoming/outgoing SMT frames in the driver if the SMT receive/RMC transmit ring is full. (?)
8. Retrieving/reporting FDDI/SNMP stats.

Both success and failure reports are welcome.

Maciej W. Rozycki <macro@orcam.me.uk>

6.6.2 SysKonnect driver - SKFP

© Copyright 1998-2000 SysKonnect,

skfp.txt created 11-May-2000

Readme File for skfp.o v2.06

1. Overview

This README explains how to use the driver 'skfp' for Linux with your network adapter.

Chapter 2: Contains a list of all network adapters that are supported by this driver.

Chapter 3:

Gives some general information.

Chapter 4: Describes common problems and solutions.

Chapter 5: Shows the changed functionality of the adapter LEDs.

Chapter 6: History of development.

2. Supported adapters

The network driver 'skfp' supports the following network adapters: SysKonnect adapters:

- SK-5521 (SK-NET FDDI-UP)
- SK-5522 (SK-NET FDDI-UP DAS)
- SK-5541 (SK-NET FDDI-FP)
- SK-5543 (SK-NET FDDI-LP)
- SK-5544 (SK-NET FDDI-LP DAS)
- SK-5821 (SK-NET FDDI-UP64)
- SK-5822 (SK-NET FDDI-UP64 DAS)
- SK-5841 (SK-NET FDDI-FP64)
- SK-5843 (SK-NET FDDI-LP64)
- SK-5844 (SK-NET FDDI-LP64 DAS)

Compaq adapters (not tested):

- Netelligent 100 FDDI DAS Fibre SC
- Netelligent 100 FDDI SAS Fibre SC
- Netelligent 100 FDDI DAS UTP
- Netelligent 100 FDDI SAS UTP
- Netelligent 100 FDDI SAS Fibre MIC

3. General Information

From v2.01 on, the driver is integrated in the linux kernel sources. Therefore, the installation is the same as for any other adapter supported by the kernel.

Refer to the manual of your distribution about the installation of network adapters.

Makes my life much easier :-)

4. Troubleshooting

If you run into problems during installation, check those items:

Problem:

The FDDI adapter cannot be found by the driver.

Reason:

Look in /proc/pci for the following entry:

'FDDI network controller: SysKonnect SK-FDDI-PCI ...'

If this entry exists, then the FDDI adapter has been found by the system and should be able to be used.

If this entry does not exist or if the file '/proc/pci' is not there, then you may have a hardware problem or PCI support may not be enabled in your kernel.

The adapter can be checked using the diagnostic program which is available from the SysKonnect web site:

www.syskonnect.de

Some COMPAQ machines have a problem with PCI under Linux. This is described in the 'PCI howto' document (included in some distributions or available from the www, e.g. at 'www.linux.org') and no workaround is available.

Problem:

You want to use your computer as a router between multiple IP subnetworks (using multiple adapters), but you cannot reach computers in other subnetworks.

Reason:

Either the router's kernel is not configured for IP forwarding or there is a problem with the routing table and gateway configuration in at least one of the computers.

If your problem is not listed here, please contact our technical support for help.

You can send email to: linux@syskonnect.de

When contacting our technical support, please ensure that the following information is available:

- System Manufacturer and Model
- Boards in your system
- Distribution
- Kernel version

5. Function of the Adapter LEDs

The functionality of the LED's on the FDDI network adapters was changed in SMT version v2.82. With this new SMT version, the yellow LED works as a ring operational indicator. An active yellow LED indicates that the ring is down. The green LED on the adapter now works as a link indicator where an active GREEN LED indicates that the respective port has a physical connection.

With versions of SMT prior to v2.82 a ring up was indicated if the yellow LED was off while the green LED(s) showed the connection status of the adapter. During a ring down the green LED was off and the yellow LED was on.

All implementations indicate that a driver is not loaded if all LEDs are off.

6. History

v2.06 (20000511) (In-Kernel version)

New features:

- 64 bit support
- new pci dma interface
- in kernel 2.3.99

v2.05 (20000217) (In-Kernel version)

New features:

- Changes for 2.3.45 kernel

v2.04 (20000207) (Standalone version)

New features:

- Added rx/tx byte counter

v2.03 (20000111) (Standalone version)

Problems fixed:

- Fixed printk statements from v2.02

v2.02 (991215) (Standalone version)

Problems fixed:

- Removed unnecessary output
- Fixed path for "printver.sh" in makefile

v2.01 (991122) (In-Kernel version)

New features:

- Integration in Linux kernel sources
- Support for memory mapped I/O.

v2.00 (991112)

New features:

- Full source released under GPL

v1.05 (991023)

Problems fixed:

- Compilation with kernel version 2.2.13 failed

v1.04 (990427)

Changes:

- New SMT module included, changing LED functionality

Problems fixed:

- Synchronization on SMP machines was buggy

v1.03 (990325)

Problems fixed:

- Interrupt routing on SMP machines could be incorrect

v1.02 (990310)

New features:

- Support for kernel versions 2.2.x added
- Kernel patch instead of private duplicate of kernel functions

v1.01 (980812)

Problems fixed:

Connection hangup with telnet Slow telnet connection

v1.00 beta 01 (980507)

New features:

None.

Problems fixed:

None.

Known limitations:

- tar archive instead of standard package format (rpm).
- FDDI statistic is empty.
- not tested with 2.1.xx kernels
- integration in kernel not tested
- not tested simultaneously with FDDI adapters from other vendors.
- only X86 processors supported.
- SBA (Synchronous Bandwidth Allocator) parameters can not be configured.
- does not work on some COMPAQ machines. See the PCI howto document for details about this problem.
- data corruption with kernel versions below 2.0.33.

6.7 Amateur Radio Device Drivers

Contents:

6.7.1 Linux Drivers for Baycom Modems

Thomas M. Sailer, HB9JNX/AE4WA, <sailer@ife.ee.ethz.ch>

The drivers for the baycom modems have been split into separate drivers as they did not share any code, and the driver and device names have changed.

This document describes the Linux Kernel Drivers for simple Baycom style amateur radio modems.

The following drivers are available:**baycom_ser_fdx:**

This driver supports the SER12 modems either full or half duplex. Its baud rate may be changed via the baud module parameter, therefore it supports just about every bit bang modem on a serial port. Its devices are called bcsf0 through bcsf3. This is the recommended driver for SER12 type modems, however if you have a broken UART clone that does not have working delta status bits, you may try baycom_ser_hdx.

baycom_ser_hdx:

This is an alternative driver for SER12 type modems. It only supports half duplex, and only 1200 baud. Its devices are called bcsh0 through bcsh3. Use this driver only if baycom_ser_fdx does not work with your UART.

baycom_par:

This driver supports the par96 and picpar modems. Its devices are called bcp0 through bcp3.

baycom_epp:

This driver supports the EPP modem. Its devices are called bce0 through bce3. This driver is work-in-progress.

The following modems are supported:

ser12	This is a very simple 1200 baud AFSK modem. The modem consists only of a modulator/demodulator chip, usually a TI TCM3105. The computer is responsible for regenerating the receiver bit clock, as well as for handling the HDLC protocol. The modem connects to a serial port, hence the name. Since the serial port is not used as an async serial port, the kernel driver for serial ports cannot be used, and this driver only supports standard serial hardware (8250, 16450, 16550)
par96	This is a modem for 9600 baud FSK compatible to the G3RUH standard. The modem does all the filtering and regenerates the receiver clock. Data is transferred from and to the PC via a shift register. The shift register is filled with 16 bits and an interrupt is signalled. The PC then empties the shift register in a burst. This modem connects to the parallel port, hence the name. The modem leaves the implementation of the HDLC protocol and the scrambler polynomial to the PC.
picpar	This is a redesign of the par96 modem by Henning Rech, DF9IC. The modem is protocol compatible to par96, but uses only three low power ICs and can therefore be fed from the parallel port and does not require an additional power supply. Furthermore, it incorporates a carrier detect circuitry.
EPP	This is a high-speed modem adaptor that connects to an enhanced parallel port. Its target audience is users working over a high speed hub (76.8kbit/s).
eppfpga	This is a redesign of the EPP adaptor.

All of the above modems only support half duplex communications. However, the driver supports the KISS (see below) fullduplex command. It then simply starts to send as soon as there's a packet to transmit and does not care about DCD, i.e. it starts to send even if there's someone else on the channel. This command is required by some implementations of the DAMA channel access protocol.

The Interface of the drivers

Unlike previous drivers, these drivers are no longer character devices, but they are now true kernel network interfaces. Installation is therefore simple. Once installed, four interfaces named bc{sf,sh,p,e}[0-3] are available. sethdlc from the ax25 utilities may be used to set driver states etc. Users of userland AX.25 stacks may use the net2kiss utility (also available in the ax25 utilities package) to convert packets of a network interface to a KISS stream on a pseudo tty. There's also a patch available from me for WAMPES which allows attaching a kernel network interface directly.

Configuring the driver

Every time a driver is inserted into the kernel, it has to know which modems it should access at which ports. This can be done with the setbaycom utility. If you are only using one modem, you can also configure the driver from the insmod command line (or by means of an option line in /etc/modprobe.d/*.conf).

Examples:

```
modprobe baycom_ser_fdx mode="ser12*" iobase=0x3f8 irq=4
sethdlc -i bcsf0 -p mode "ser12*" io 0x3f8 irq 4
```

Both lines configure the first port to drive a ser12 modem at the first serial port (COM1 under DOS). The * in the mode parameter instructs the driver to use the software DCD algorithm (see below):

```
insmod baycom_par mode="picpar" iobase=0x378
sethdlc -i bcp0 -p mode "picpar" io 0x378
```

Both lines configure the first port to drive a picpar modem at the first parallel port (LPT1 under DOS). (Note: picpar implies hardware DCD, par96 implies software DCD).

The channel access parameters can be set with sethdlc -a or kissparms. Note that both utilities interpret the values slightly differently.

Hardware DCD versus Software DCD

To avoid collisions on the air, the driver must know when the channel is busy. This is the task of the DCD circuitry/software. The driver may either utilise a software DCD algorithm (options=1) or use a DCD signal from the hardware (options=0).

ser12	if software DCD is utilised, the radio's squelch should always be open. It is highly recommended to use the software DCD algorithm, as it is much faster than most hardware squelch circuitry. The disadvantage is a slightly higher load on the system.
par96	the software DCD algorithm for this type of modem is rather poor. The modem simply does not provide enough information to implement a reasonable DCD algorithm in software. Therefore, if your radio feeds the DCD input of the PAR96 modem, the use of the hardware DCD circuitry is recommended.
picpar	the picpar modem features a builtin DCD hardware, which is highly recommended.

Compatibility with the rest of the Linux kernel

The serial driver and the baycom serial drivers compete for the same hardware resources. Of course only one driver can access a given interface at a time. The serial driver grabs all interfaces it can find at startup time. Therefore the baycom drivers subsequently won't be able to access a serial port. You might therefore find it necessary to release a port owned by the serial driver with 'setserial /dev/ttyS# uart none', where # is the number of the interface. The baycom drivers do not reserve any ports at startup, unless one is specified on the 'insmod' command line. Another method to solve the problem is to compile all drivers as modules and leave it to kmod to load the correct driver depending on the application.

The parallel port drivers (baycom_par, baycom_epp) now use the parport subsystem to arbitrate the ports between different client drivers.

vy 73s de

Tom Sailer, sailer@ife.ee.ethz.ch

hb9jnx @ hb9w.ampr.org

6.7.2 SCC.C - Linux driver for Z8530 based HDLC cards for AX.25

This is a subset of the documentation. To use this driver you MUST have the full package from: Internet:

1. ftp://ftp.ccac.rwth-aachen.de/pub/jr/z8530drv-utils_3.0-3.tar.gz
2. ftp://ftp.pspt.fi/pub/ham/linux/ax25/z8530drv-utils_3.0-3.tar.gz

Please note that the information in this document may be hopelessly outdated. A new version of the documentation, along with links to other important Linux Kernel AX.25 documentation and programs, is available on <http://yaina.de/jreuter>

Copyright © 1993,2000 by Joerg Reuter DL1BKE <jreuter@yaina.de>

portions Copyright © 1993 Guido ten Dolle PE1NNZ

for the complete copyright notice see >> Copying.Z8530DRV <<

1. Initialization of the driver

To use the driver, 3 steps must be performed:

1. if compiled as module: loading the module
2. Setup of hardware, MODEM and KISS parameters with sccinit
3. Attach each channel to the Linux kernel AX.25 with "ifconfig"

Unlike the versions below 2.4 this driver is a real network device driver. If you want to run xNOS instead of our fine kernel AX.25 use a 2.x version (available from above sites) or read the AX.25-HOWTO on how to emulate a KISS TNC on network device drivers.

1.1 Loading the module

**(If you're going to compile the driver as a part of the kernel image,
skip this chapter and continue with 1.2)**

Before you can use a module, you'll have to load it with:

```
insmod scc.o
```

please read 'man insmod' that comes with module-init-tools.

You should include the insmod in one of the /etc/rc.d/rc.* files, and don't forget to insert a call of sccinit after that. It will read your /etc/z8530drv.conf.

1.2. /etc/z8530drv.conf

To setup all parameters you must run /sbin/sccinit from one of your rc.*-files. This has to be done BEFORE you can "ifconfig" an interface. Sccinit reads the file /etc/z8530drv.conf and sets the hardware, MODEM and KISS parameters. A sample file is delivered with this package. Change it to your needs.

The file itself consists of two main sections.

1.2.1 configuration of hardware parameters

The hardware setup section defines the following parameters for each Z8530:

```
chip      1
data_a   0x300          # data port A
ctrl_a   0x304          # control port A
data_b   0x301          # data port B
ctrl_b   0x305          # control port B
irq      5              # IRQ No. 5
pclock   4915200        # clock
board    BAYCOM         # hardware type
escc     no             # enhanced SCC chip? (8580/85180/85280)
vector   0              # latch for interrupt vector
special  no             # address of special function register
option   0              # option to set via sfr
```

chip

- this is just a delimiter to make sccinit a bit simpler to program. A parameter has no effect.

data_a

- the address of the data port A of this Z8530 (needed)

ctrl_a

- the address of the control port A (needed)

data_b

- the address of the data port B (needed)

ctrl_b

- the address of the control port B (needed)

irq

- the used IRQ for this chip. Different chips can use different IRQs or the same. If they share an interrupt, it needs to be specified within one chip-definition only.

pclock - the clock at the PCLK pin of the Z8530 (option, 4915200 is default), measured in Hertz

board

- the "type" of the board:

SCC type	value
PA0HZP SCC card	PA0HZP
EAGLE card	EAGLE
PC100 card	PC100
PRIMUS-PC (DG9BL) card	PRIMUS
BayCom (U)SCC card	BAYCOM

escc

- if you want support for ESCC chips (8580, 85180, 85280), set this to "yes" (option, defaults to "no")

vector

- address of the vector latch (aka "intack port") for PA0HZP cards. There can be only one vector latch for all chips! (option, defaults to 0)

special

- address of the special function register on several cards. (option, defaults to 0)

option - The value you write into that register (option, default is 0)

You can specify up to four chips (8 channels). If this is not enough, just change:

```
#define MAXSCC 4
```

to a higher value.

Example for the BAYCOM USCC:

```
chip      1
data_a   0x300          # data port A
ctrl_a   0x304          # control port A
data_b   0x301          # data port B
ctrl_b   0x305          # control port B
irq      5              # IRQ No. 5 (#)
board    BAYCOM          # hardware type (*)
```

```
#  
# SCC chip 2  
#  
chip 2  
data_a 0x302  
ctrl_a 0x306  
data_b 0x303  
ctrl_b 0x307  
board BAYCOM
```

An example for a PA0HZP card:

```
chip 1  
data_a 0x153  
data_b 0x151  
ctrl_a 0x152  
ctrl_b 0x150  
irq 9  
pclock 4915200  
board PA0HZP  
vector 0x168  
escc no  
#  
#  
#  
chip 2  
data_a 0x157  
data_b 0x155  
ctrl_a 0x156  
ctrl_b 0x154  
irq 9  
pclock 4915200  
board PA0HZP  
vector 0x168  
escc no
```

A DRSI would should probably work with this:

(actually: two DRSI cards...)

```
chip 1  
data_a 0x303  
data_b 0x301  
ctrl_a 0x302  
ctrl_b 0x300  
irq 7  
pclock 4915200  
board DRSI
```

```
escc no
#
#
#
chip 2
data_a 0x313
data_b 0x311
ctrl_a 0x312
ctrl_b 0x310
irq 7
pclock 4915200
board DRSI
escc no
```

Note that you cannot use the on-board baudrate generator off DRSI cards. Use "mode dpll" for clock source (see below).

This is based on information provided by Mike Bilow (and verified by Paul Helay)

The utility "gencfg"

If you only know the parameters for the PE1CHL driver for DOS, run gencfg. It will generate the correct port addresses (I hope). Its parameters are exactly the same as the ones you use with the "attach scc" command in net, except that the string "init" must not appear. Example:

```
gencfg 2 0x150 4 2 0 1 0x168 9 4915200
```

will print a skeleton z8530drv.conf for the OptoSCC to stdout.

```
gencfg 2 0x300 2 4 5 -4 0 7 4915200 0x10
```

does the same for the BAYCOM USCC card. In my opinion it is much easier to edit scc_config.h...

1.2.2 channel configuration

The channel definition is divided into three sub sections for each channel:

An example for scc0:

```
# DEVICE

device scc0      # the device for the following params

# MODEM / BUFFERS

speed 1200          # the default baudrate
clock dpll          # clock source:
                     #       dpll      = normal half duplex operation
                     #       external = MODEM provides own Rx/Tx clock
                     #       divider  = use full duplex divider if
                     #                  installed (1)
```

```

mode nrzi          # HDLC encoding mode
#           nrzi = 1k2 MODEM, G3RUH 9k6 MODEM
#           nrz  = DF9IC 9k6 MODEM
#
bufsize 384       # size of buffers. Note that this must include
# the AX.25 header, not only the data field!
# (optional, defaults to 384)

# KISS (Layer 1)

txdelay 36         # (see chapter 1.4)
persist 64
slot 8
tail 8
fulldup 0
wait 12
min 3
maxkey 7
idle 3
maxdef 120
group 0
txoff off
softdcd on
slip off

```

The order WITHIN these sections is unimportant. The order OF these sections IS important. The MODEM parameters are set with the first recognized KISS parameter...

Please note that you can initialize the board only once after boot (or insmod). You can change all parameters but "mode" and "clock" later with the Sccparam program or through KISS. Just to avoid security holes...

- (1) this divider is usually mounted on the SCC-PBC (PA0HZP) or not present at all (BayCom). It feeds back the output of the DPLL (digital pll) as transmit clock. Using this mode without a divider installed will normally result in keying the transceiver until maxkey expires --- of course without sending anything (useful).

2. Attachment of a channel by your AX.25 software

2.1 Kernel AX.25

To set up an AX.25 device you can simply type:

```
ifconfig scc0 44.128.1.1 hw ax25 dl0tha-7
```

This will create a network interface with the IP number 44.128.20.107 and the callsign "dl0tha". If you do not have any IP number (yet) you can use any of the 44.128.0.0 network. Note that you do not need axattach. The purpose of axattach (like slattach) is to create a KISS network device linked to a TTY. Please read the documentation of the ax25-utils and the AX.25-HOWTO to learn how to set the parameters of the kernel AX.25.

2.2 NOS, NET and TFKISS

Since the TTY driver (aka KISS TNC emulation) is gone you need to emulate the old behaviour. The cost of using these programs is that you probably need to compile the kernel AX.25, regardless of whether you actually use it or not. First setup your /etc/ax25/axports, for example:

```
9k6      dl0tha-9  9600  255 4 9600 baud port (scc3)
axlink   dl0tha-15 38400 255 4 Link to NOS
```

Now "ifconfig" the scc device:

```
ifconfig scc3 44.128.1.1 hw ax25 dl0tha-9
```

You can now axattach a pseudo-TTY:

```
axattach /dev/ptys0 axlink
```

and start your NOS and attach /dev/ptys0 there. The problem is that NOS is reachable only via digipeating through the kernel AX.25 (disastrous on a DAMA controlled channel). To solve this problem, configure "rxecho" to echo the incoming frames from "9k6" to "axlink" and outgoing frames from "axlink" to "9k6" and start:

```
rxecho
```

Or simply use "kissbridge" coming with z8530drv-utils:

```
ifconfig scc3 hw ax25 dl0tha-9
kissbridge scc3 /dev/ptys0
```

3. Adjustment and Display of parameters

3.1 Displaying SCC Parameters:

Once a SCC channel has been attached, the parameter settings and some statistic information can be shown using the param program:

```
dl1bke-u:~$ sccstat scc0
```

Parameters:

```
speed      : 1200 baud
txdelay    : 36
persist    : 255
slottime   : 0
txtail     : 8
fulldup   : 1
waittime  : 12
mintime   : 3 sec
maxkeyup  : 7 sec
idletime   : 3 sec
maxdefer  : 120 sec
```

```
group      : 0x00
txoff     : off
softdcd   : on
SLIP       : off
```

Status:

HDLC	Z8530	Interrupts	Buffers
<hr/>			
Sent :	273	RxOver :	0 RxInts : 125074 Size : 384
Received :	1095	TxUnder :	0 TxInts : 4684 NoSpace : 0
RxErrors :	1591		ExInts : 11776
TxErrors :	0		SpInts : 1503
Tx State :	idle		

The status info shown is:

Sent	number of frames transmitted
Received	number of frames received
RxErrors	number of receive errors (CRC, ABORT)
TxErrors	number of discarded Tx frames (due to various reasons)
Tx State	status of the Tx interrupt handler: idle/busy/active/tail (2)
RxOver	number of receiver overruns
TxUnder	number of transmitter underruns
RxInts	number of receiver interrupts
TxInts	number of transmitter interrupts
EpInts	number of receiver special condition interrupts
SpInts	number of external/status interrupts
Size	maximum size of an AX.25 frame (<i>with</i> AX.25 headers!)
NoSpace	number of times a buffer could not get allocated

An overrun is abnormal. If lots of these occur, the product of baudrate and number of interfaces is too high for the processing power of your computer. NoSpace errors are unlikely to be caused by the driver or the kernel AX.25.

3.2 Setting Parameters

The setting of parameters of the emulated KISS TNC is done in the same way in the SCC driver. You can change parameters by using the kissparms program from the ax25-utils package or use the program "scparam":

```
scparam <device> <paramname> <decimal-|hexadecimal value>
```

You can change the following parameters:

param	value
speed	1200
txdelay	36
persist	255
slottime	0
txtail	8
fulldup	1
waittime	12
mintime	3
maxkeyup	7
idletime	3
maxdefer	120
group	0x00
txoff	off
softdcd	on
SLIP	off

The parameters have the following meaning:

speed:

The baudrate on this channel in bits/sec

Example: `sccparam /dev/scc3 speed 9600`

txdelay:

The delay (in units of 10 ms) after keying of the transmitter, until the first byte is sent. This is usually called "TXDELAY" in a TNC. When 0 is specified, the driver will just wait until the CTS signal is asserted. This assumes the presence of a timer or other circuitry in the MODEM and/or transmitter, that asserts CTS when the transmitter is ready for data. A normal value of this parameter is 30-36.

Example: `sccparam /dev/scc0 txd 20`

persist:

This is the probability that the transmitter will be keyed when the channel is found to be free. It is a value from 0 to 255, and the probability is $(\text{value}+1)/256$. The value should be somewhere near 50-60, and should be lowered when the channel is used more heavily.

Example: `sccparam /dev/scc2 persist 20`

slottime:

This is the time between samples of the channel. It is expressed in units of 10 ms. About 200-300 ms (value 20-30) seems to be a good value.

Example: `sccparam /dev/scc0 slot 20`

tail:

The time the transmitter will remain keyed after the last byte of a packet has been transferred to the SCC. This is necessary because the CRC and a flag still have to leave the SCC before the transmitter is keyed down. The value depends on the baudrate selected. A few character times should be sufficient, e.g. 40ms at 1200 baud. (value 4) The value of this parameter is in 10 ms units.

Example: `sccparam /dev/scc2 4`

full:

The full-duplex mode switch. This can be one of the following values:

0: The interface will operate in CSMA mode (the normal half-duplex packet radio operation)

1: Fullduplex mode, i.e. the transmitter will be keyed at

any time, without checking the received carrier. It will be unkeyed when there are no packets to be sent.

2: Like 1, but the transmitter will remain keyed, also

when there are no packets to be sent. Flags will be sent in that case, until a timeout (parameter 10) occurs.

Example: `sccparam /dev/scc0 fulldup off`

wait:

The initial waittime before any transmit attempt, after the frame has been queue for transmit. This is the length of the first slot in CSMA mode. In full duplex modes it is set to 0 for maximum performance. The value of this parameter is in 10 ms units.

Example: `sccparam /dev/scc1 wait 4`

maxkey:

The maximal time the transmitter will be keyed to send packets, in seconds. This can be useful on busy CSMA channels, to avoid "getting a bad reputation" when you are generating a lot of traffic. After the specified time has elapsed, no new frame will be started. Instead, the transmitter will be switched off for a specified time (parameter min), and then the selected algorithm for keyup will be started again. The value 0 as well as "off" will disable this feature, and allow infinite transmission time.

Example: `sccparam /dev/scc0 maxk 20`

min:

This is the time the transmitter will be switched off when the maximum transmission time is exceeded.

Example: `sccparam /dev/scc3 min 10`

idle:

This parameter specifies the maximum idle time in full duplex 2 mode, in seconds. When no frames have been sent for this time, the transmitter will be keyed down. A value of 0 is has same result as the fullduplex mode 1. This parameter can be disabled.

Example: `sccparam /dev/scc2 idle off # transmit forever`

maxdefer

This is the maximum time (in seconds) to wait for a free channel to send. When this timer expires the transmitter will be keyed IMMEDIATELY. If you love to get trouble with other users you should set this to a very low value ;-)

Example: `sccparam /dev/scc0 maxdefer 240 # 2 minutes`

txoff:

When this parameter has the value 0, the transmission of packets is enable. Otherwise it is disabled.

Example: `sccparam /dev/scc2 txoff on`

group:

It is possible to build special radio equipment to use more than one frequency on the same band, e.g. using several receivers and only one transmitter that can be switched between frequencies. Also, you can connect several radios that are active on the same band. In these cases, it is not possible, or not a good idea, to transmit on more than one frequency. The SCC driver provides a method to lock transmitters on different interfaces, using the "param <interface> group <x>" command. This will only work when you are using CSMA mode (parameter full = 0).

The number <x> must be 0 if you want no group restrictions, and can be computed as follows to create restricted groups: <x> is the sum of some OCTAL numbers:

200	This transmitter will only be keyed when all other transmitters in the group are off.
100	This transmitter will only be keyed when the carrier detect of all other interfaces in the group is off.
0xx	A byte that can be used to define different groups. Interfaces are in the same group, when the logical AND between their xx values is nonzero.

Examples:

When 2 interfaces use group 201, their transmitters will never be keyed at the same time.

When 2 interfaces use group 101, the transmitters will only key when both channels are clear at the same time. When group 301, the transmitters will not be keyed at the same time.

Don't forget to convert the octal numbers into decimal before you set the parameter.

Example: (to be written)

softdcd:

use a software dcd instead of the real one... Useful for a very slow squelch.

Example: sccparam /dev/scc0 soft on

4. Problems

If you have tx-problems with your BayCom USCC card please check the manufacturer of the 8530. SGS chips have a slightly different timing. Try Zilog... A solution is to write to register 8 instead to the data port, but this won't work with the ESCC chips. *SIGH!*

A very common problem is that the PTT locks until the maxkeyup timer expires, although interrupts and clock source are correct. In most cases compiling the driver with CONFIG_SCC_DELAY (set with make config) solves the problems. For more hints read the (pseudo) FAQ and the documentation coming with z8530drv-utils.

I got reports that the driver has problems on some 386-based systems. (i.e. Amstrad) Those systems have a bogus AT bus timing which will lead to delayed answers on interrupts. You can recognize these problems by looking at the output of Sccstat for the suspected port. If it shows under- and overruns you own such a system.

Delayed processing of received data: This depends on

- the kernel version

- kernel profiling compiled or not
- a high interrupt load
- a high load of the machine --- running X, Xmorph, XV and Povray, while compiling the kernel... hmm ... even with 32 MB RAM ... ;-) Or running a named for the whole .ampr.org domain on an 8 MB box...
- using information from rxecho or kissbridge.

Kernel panics: please read /linux/README and find out if it really occurred within the scc driver.

If you cannot solve a problem, send me

- a description of the problem,
- information on your hardware (computer system, scc board, modem)
- your kernel version
- the output of cat /proc/net/z8530

4. Thor RLC100

Mysteriously this board seems not to work with the driver. Anyone got it up-and-running?

Many thanks to Linus Torvalds and Alan Cox for including the driver in the Linux standard distribution and their support.

Joerg Reuter ampr-net: dl1bke@db0pra.ampr.org
 AX-25 : DL1BKE @ DB0ABH.#BAY.DEU.EU
 Internet: j.reuter@yaina.de
 WWW : <http://yaina.de/jreuter>

6.8 Wi-Fi Device Drivers

Contents:

6.8.1 Intel(R) PRO/Wireless 2100 Driver for Linux

Support for:

- Intel(R) PRO/Wireless 2100 Network Connection

Copyright © 2003-2006, Intel Corporation

README.ipw2100

Version

git-1.1.5

Date

January 25, 2006

0. IMPORTANT INFORMATION BEFORE USING THIS DRIVER

Important Notice FOR ALL USERS OR DISTRIBUTORS!!!!

Intel wireless LAN adapters are engineered, manufactured, tested, and quality checked to ensure that they meet all necessary local and governmental regulatory agency requirements for the regions that they are designated and/or marked to ship into. Since wireless LANs are generally unlicensed devices that share spectrum with radars, satellites, and other licensed and unlicensed devices, it is sometimes necessary to dynamically detect, avoid, and limit usage to avoid interference with these devices. In many instances Intel is required to provide test data to prove regional and local compliance to regional and governmental regulations before certification or approval to use the product is granted. Intel's wireless LAN's EEPROM, firmware, and software driver are designed to carefully control parameters that affect radio operation and to ensure electromagnetic compliance (EMC). These parameters include, without limitation, RF power, spectrum usage, channel scanning, and human exposure.

For these reasons Intel cannot permit any manipulation by third parties of the software provided in binary format with the wireless WLAN adapters (e.g., the EEPROM and firmware). Furthermore, if you use any patches, utilities, or code with the Intel wireless LAN adapters that have been manipulated by an unauthorized party (i.e., patches, utilities, or code (including open source code modifications) which have not been validated by Intel), (i) you will be solely responsible for ensuring the regulatory compliance of the products, (ii) Intel will bear no liability, under any theory of liability for any issues associated with the modified products, including without limitation, claims under the warranty and/or issues arising from regulatory non-compliance, and (iii) Intel will not provide or be required to assist in providing support to any third parties for such modified products.

Note: Many regulatory agencies consider Wireless LAN adapters to be modules, and accordingly, condition system-level regulatory approval upon receipt and review of test data documenting that the antennas and system configuration do not cause the EMC and radio operation to be non-compliant.

The drivers available for download from SourceForge are provided as a part of a development project. Conformance to local regulatory requirements is the responsibility of the individual developer. As such, if you are interested in deploying or shipping a driver as part of solution intended to be used for purposes other than development, please obtain a tested driver from Intel Customer Support at:

<https://www.intel.com/support/wireless/sb/CS-006408.htm>

1. Introduction

This document provides a brief overview of the features supported by the IPW2100 driver project. The main project website, where the latest development version of the driver can be found, is:

<http://ipw2100.sourceforge.net>

There you can find the not only the latest releases, but also information about potential fixes and patches, as well as links to the development mailing list for the driver project.

2. Release git-1.1.5 Current Supported Features

- Managed (BSS) and Ad-Hoc (IBSS)
- WEP (shared key and open)
- Wireless Tools support
- 802.1x (tested with XSupplicant 1.0.1)

Enabled (but not supported) features: - Monitor/RFMon mode - WPA/WPA2

The distinction between officially supported and enabled is a reflection on the amount of validation and interoperability testing that has been performed on a given feature.

3. Command Line Parameters

If the driver is built as a module, the following optional parameters are used by entering them on the command line with the modprobe command using this syntax:

```
modprobe ipw2100 [<option>=<VAL1><,VAL2>...]
```

For example, to disable the radio on driver loading, enter:

```
modprobe ipw2100 disable=1
```

The ipw2100 driver supports the following module parameters:

Name	Value	Example	Meaning
debug	0x0-0xffffffff	debug=1024	Debug level set to 1024
mode	0,1,2	mode=1	AdHoc
channel	int	channel=3	Only valid in AdHoc or Monitor
associate	boolean	associate=0	Do NOT auto associate
disable	boolean	disable=1	Do not power the HW

4. Sysfs Helper Files

There are several ways to control the behavior of the driver. Many of the general capabilities are exposed through the Wireless Tools (iwconfig). There are a few capabilities that are exposed through entries in the Linux Sysfs.

Driver Level

For the driver level files, look in /sys/bus/pci/drivers/ipw2100/

debug_level

This controls the same global as the 'debug' module parameter. For information on the various debugging levels available, run the 'dvals' script found in the driver source directory.

Note: 'debug_level' is only enabled if CONFIG_IPW2100_DEBUG is turn on.

Device Level

For the device level files look in:

```
/sys/bus/pci/drivers/ipw2100/{PCI-ID}/
```

For example:

```
/sys/bus/pci/drivers/ipw2100/0000:02:01.0
```

For the device level files, see /sys/bus/pci/drivers/ipw2100:

rf_kill

read

0	RF kill not enabled (radio on)
1	SW based RF kill active (radio off)
2	HW based RF kill active (radio off)
3	Both HW and SW RF kill active (radio off)

write

0	If SW based RF kill active, turn the radio back on
1	If radio is on, activate SW based RF kill

Note: If you enable the SW based RF kill and then toggle the HW based RF kill from ON -> OFF -> ON, the radio will NOT come back on

5. Radio Kill Switch

Most laptops provide the ability for the user to physically disable the radio. Some vendors have implemented this as a physical switch that requires no software to turn the radio off and on. On other laptops, however, the switch is controlled through a button being pressed and a software driver then making calls to turn the radio off and on. This is referred to as a "software based RF kill switch"

See the Sysfs helper file 'rf_kill' for determining the state of the RF switch on your system.

6. Dynamic Firmware

As the firmware is licensed under a restricted use license, it can not be included within the kernel sources. To enable the IPW2100 you will need a firmware image to load into the wireless NIC's processors.

You can obtain these images from <<http://ipw2100.sf.net/firmware.php>>.

See INSTALL for instructions on installing the firmware.

7. Power Management

The IPW2100 supports the configuration of the Power Save Protocol through a private wireless extension interface. The IPW2100 supports the following different modes:

off	No power management. Radio is always on.
on	Automatic power management
1-5	Different levels of power management. The higher the number the greater the power savings, but with an impact to packet latencies.

Power management works by powering down the radio after a certain interval of time has passed where no packets are passed through the radio. Once powered down, the radio remains in that state for a given period of time. For higher power savings, the interval between last packet processed to sleep is shorter and the sleep period is longer.

When the radio is asleep, the access point sending data to the station must buffer packets at the AP until the station wakes up and requests any buffered packets. If you have an AP that does not correctly support the PSP protocol you may experience packet loss or very poor performance while power management is enabled. If this is the case, you will need to try and find a firmware update for your AP, or disable power management (via `iwconfig eth1 power off`)

To configure the power level on the IPW2100 you use a combination of `iwconfig` and `iwpriv`. `iwconfig` is used to turn power management on, off, and set it to auto.

<code>iwconfig eth1 power off</code>	Disables radio power down
<code>iwconfig eth1 power on</code>	Enables radio power management to last set level (defaults to AUTO)
<code>iwpriv eth1 set_power 0</code>	Sets power level to AUTO and enables power management if not previously enabled.
<code>iwpriv eth1 set_power 1-5</code>	Set the power level as specified, enabling power management if not previously enabled.

You can view the current power level setting via:

```
iwpriv eth1 get_power
```

It will return the current period or timeout that is configured as a string in the form of xxxx/yyyy(z) where xxxx is the timeout interval (amount of time after packet processing), yyyy is the period to sleep (amount of time to wait before powering the radio and querying the access point for buffered packets), and z is the 'power level'. If power management is turned off the xxxx/yyyy will be replaced with 'off' -- the level reported will be the active level if `iwconfig eth1 power on` is invoked.

8. Support

For general development information and support, go to:

<http://ipw2100.sf.net/>

The ipw2100 1.1.0 driver and firmware can be downloaded from:

<http://support.intel.com>

For installation support on the ipw2100 1.1.0 driver on Linux kernels 2.6.8 or greater, email support is available from:

<http://supportmail.intel.com>

9. License

Copyright © 2003 - 2006 Intel Corporation. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License (version 2) as published by the Free Software Foundation.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

The full GNU General Public License is included in this distribution in the file called LICENSE.

License Contact Information:

James P. Ketrenos <ipw2100-admin@linux.intel.com>

Intel Corporation, 5200 N.E. Elam Young Parkway, Hillsboro, OR 97124-6497

6.8.2 Intel(R) PRO/Wireless 2915ABG Driver for Linux

Support for:

- Intel(R) PRO/Wireless 2200BG Network Connection
- Intel(R) PRO/Wireless 2915ABG Network Connection

Note: The Intel(R) PRO/Wireless 2915ABG Driver for Linux and Intel(R) PRO/Wireless 2200BG Driver for Linux is a unified driver that works on both hardware adapters listed above. In this document the Intel(R) PRO/Wireless 2915ABG Driver for Linux will be used to reference the unified driver.

Copyright © 2004-2006, Intel Corporation

README.ipw2200

Version

1.1.2

Date

March 30, 2006

0. IMPORTANT INFORMATION BEFORE USING THIS DRIVER

Important Notice FOR ALL USERS OR DISTRIBUTORS!!!!

Intel wireless LAN adapters are engineered, manufactured, tested, and quality checked to ensure that they meet all necessary local and governmental regulatory agency requirements for the regions that they are designated and/or marked to ship into. Since wireless LANs are generally unlicensed devices that share spectrum with radars, satellites, and other licensed and unlicensed devices, it is sometimes necessary to dynamically detect, avoid, and limit usage to avoid interference with these devices. In many instances Intel is required to provide test data to prove regional and local compliance to regional and governmental regulations before certification or approval to use the product is granted. Intel's wireless LAN's EEPROM, firmware, and software driver are designed to carefully control parameters that affect radio operation and to ensure electromagnetic compliance (EMC). These parameters include, without limitation, RF power, spectrum usage, channel scanning, and human exposure.

For these reasons Intel cannot permit any manipulation by third parties of the software provided in binary format with the wireless WLAN adapters (e.g., the EEPROM and firmware). Furthermore, if you use any patches, utilities, or code with the Intel wireless LAN adapters that have been manipulated by an unauthorized party (i.e., patches, utilities, or code (including open source code modifications) which have not been validated by Intel), (i) you will be solely responsible for ensuring the regulatory compliance of the products, (ii) Intel will bear no liability, under any theory of liability for any issues associated with the modified products, including without limitation, claims under the warranty and/or issues arising from regulatory non-compliance, and (iii) Intel will not provide or be required to assist in providing support to any third parties for such modified products.

Note: Many regulatory agencies consider Wireless LAN adapters to be modules, and accordingly, condition system-level regulatory approval upon receipt and review of test data documenting that the antennas and system configuration do not cause the EMC and radio operation to be non-compliant.

The drivers available for download from SourceForge are provided as a part of a development project. Conformance to local regulatory requirements is the responsibility of the individual developer. As such, if you are interested in deploying or shipping a driver as part of solution intended to be used for purposes other than development, please obtain a tested driver from Intel Customer Support at:

<http://support.intel.com>

1. Introduction

The following sections attempt to provide a brief introduction to using the Intel(R) PRO/Wireless 2915ABG Driver for Linux.

This document is not meant to be a comprehensive manual on understanding or using wireless technologies, but should be sufficient to get you moving without wires on Linux.

For information on building and installing the driver, see the INSTALL file.

1.1. Overview of Features

The current release (1.1.2) supports the following features:

- BSS mode (Infrastructure, Managed)
- IBSS mode (Ad-Hoc)
- WEP (OPEN and SHARED KEY mode)
- 802.1x EAP via wpa_supplicant and xsupplicant
- Wireless Extension support
- Full B and G rate support (2200 and 2915)
- Full A rate support (2915 only)
- Transmit power control
- S state support (ACPI suspend/resume)

The following features are currently enabled, but not officially supported:

- WPA
- long/short preamble support
- Monitor mode (aka RFMon)

The distinction between officially supported and enabled is a reflection on the amount of validation and interoperability testing that has been performed on a given feature.

1.2. Command Line Parameters

Like many modules used in the Linux kernel, the Intel(R) PRO/Wireless 2915ABG Driver for Linux allows configuration options to be provided as module parameters. The most common way to specify a module parameter is via the command line.

The general form is:

```
% modprobe ipw2200 parameter=value
```

Where the supported parameter are:

associate

Set to 0 to disable the auto scan-and-associate functionality of the driver. If disabled, the driver will not attempt to scan for and associate to a network until it has been configured with one or more properties for the target network, for example configuring the network SSID. Default is 0 (do not auto-associate)

Example: % modprobe ipw2200 associate=0

auto_create

Set to 0 to disable the auto creation of an Ad-Hoc network matching the channel and network name parameters provided. Default is 1.

channel

channel number for association. The normal method for setting the channel

would be to use the standard wireless tools (i.e. *iwconfig eth1 channel 10*), but it is useful sometimes to set this while debugging. Channel 0 means 'ANY'

debug

If using a debug build, this is used to control the amount of debug info is logged. See the 'dvals' and 'load' script for more info on how to use this (the dvals and load scripts are provided as part of the ipw2200 development snapshot releases available from the SourceForge project at <http://ipw2200.sf.net>)

led

Can be used to turn on experimental LED code. 0 = Off, 1 = On. Default is 1.

mode

Can be used to set the default mode of the adapter. 0 = Managed, 1 = Ad-Hoc, 2 = Monitor

1.3. Wireless Extension Private Methods

As an interface designed to handle generic hardware, there are certain capabilities not exposed through the normal Wireless Tool interface. As such, a provision is provided for a driver to declare custom, or private, methods. The Intel(R) PRO/Wireless 2915ABG Driver for Linux defines several of these to configure various settings.

The general form of using the private wireless methods is:

```
% iwpriv $IFNAME method parameters
```

Where \$IFNAME is the interface name the device is registered with (typically eth1, customized via one of the various network interface name managers, such as ifrename)

The supported private methods are:

get_mode

Can be used to report out which IEEE mode the driver is configured to support.
Example:

```
% iwpriv eth1 get_mode eth1 get_mode:802.11bg (6)
```

set_mode

Can be used to configure which IEEE mode the driver will support.

Usage:

```
% iwpriv eth1 set_mode {mode}
```

Where {mode} is a number in the range 1-7:

1	802.11a (2915 only)
2	802.11b
3	802.11ab (2915 only)
4	802.11g
5	802.11ag (2915 only)
6	802.11bg
7	802.11abg (2915 only)

get_preamble

Can be used to report configuration of preamble length.

set_preamble

Can be used to set the configuration of preamble length:

Usage:

```
% iwpriv eth1 set_preamble {mode}
```

Where {mode} is one of:

1	Long preamble only
0	Auto (long or short based on connection)

1.4. Sysfs Helper Files

The Linux kernel provides a pseudo file system that can be used to access various components of the operating system. The Intel(R) PRO/Wireless 2915ABG Driver for Linux exposes several configuration parameters through this mechanism.

An entry in the sysfs can support reading and/or writing. You can typically query the contents of a sysfs entry through the use of cat, and can set the contents via echo. For example:

```
% cat /sys/bus/pci/drivers/ipw2200/debug_level
```

Will report the current debug level of the driver's logging subsystem (only available if CONFIG_IPW2200_DEBUG was configured when the driver was built).

You can set the debug level via:

```
% echo $VALUE > /sys/bus/pci/drivers/ipw2200/debug_level
```

Where \$VALUE would be a number in the case of this sysfs entry. The input to sysfs files does not have to be a number. For example, the firmware loader used by hotplug utilizes sysfs entries for transferring the firmware image from user space into the driver.

The Intel(R) PRO/Wireless 2915ABG Driver for Linux exposes sysfs entries at two levels -- driver level, which apply to all instances of the driver (in the event that there are more than one device installed) and device level, which applies only to the single specific instance.

1.4.1 Driver Level Sysfs Helper Files

For the driver level files, look in /sys/bus/pci/drivers/ipw2200/

debug_level

This controls the same global as the 'debug' module parameter

1.4.2 Device Level Sysfs Helper Files

For the device level files, look in:

```
/sys/bus/pci/drivers/ipw2200/{PCI-ID}/
```

For example::

```
/sys/bus/pci/drivers/ipw2200/0000:02:01.0
```

For the device level files, see /sys/bus/pci/drivers/ipw2200:

rf_kill

read -

0	RF kill not enabled (radio on)
1	SW based RF kill active (radio off)
2	HW based RF kill active (radio off)
3	Both HW and SW RF kill active (radio off)

write -

0	If SW based RF kill active, turn the radio back on
1	If radio is on, activate SW based RF kill

Note: If you enable the SW based RF kill and then toggle the HW based RF kill from ON -> OFF -> ON, the radio will NOT come back on

ucode

read-only access to the ucode version number

led

read -

0	LED code disabled
1	LED code enabled

write -

0	Disable LED code
1	Enable LED code

Note: The LED code has been reported to hang some systems when running ifconfig and is therefore disabled by default.

1.5. Supported channels

Upon loading the Intel(R) PRO/Wireless 2915ABG Driver for Linux, a message stating the detected geography code and the number of 802.11 channels supported by the card will be displayed in the log.

The geography code corresponds to a regulatory domain as shown in the table below.

Code	Geography	Supported channels	
		802.11bg	802.11a
---	Restricted	11	0
ZZF	Custom US/Canada	11	8
ZZD	Rest of World	13	0
ZZA	Custom USA & Europe & High	11	13
ZZB	Custom NA & Europe	11	13
ZZC	Custom Japan	11	4
ZZM	Custom	11	0
ZZE	Europe	13	19
ZZJ	Custom Japan	14	4
ZZR	Rest of World	14	0
ZZH	High Band	13	4
ZZG	Custom Europe	13	4
ZZK	Europe	13	24
ZZL	Europe	11	13

2. Ad-Hoc Networking

When using a device in an Ad-Hoc network, it is useful to understand the sequence and requirements for the driver to be able to create, join, or merge networks.

The following attempts to provide enough information so that you can have a consistent experience while using the driver as a member of an Ad-Hoc network.

2.1. Joining an Ad-Hoc Network

The easiest way to get onto an Ad-Hoc network is to join one that already exists.

2.2. Creating an Ad-Hoc Network

An Ad-Hoc networks is created using the syntax of the Wireless tool.

For Example: iwconfig eth1 mode ad-hoc essid testing channel 2

2.3. Merging Ad-Hoc Networks

3. Interaction with Wireless Tools

3.1 iwconfig mode

When configuring the mode of the adapter, all run-time configured parameters are reset to the value used when the module was loaded. This includes channels, rates, ESSID, etc.

3.2 iwconfig sens

The 'iwconfig ethX sens XX' command will not set the signal sensitivity threshold, as described in iwconfig documentation, but rather the number of consecutive missed beacons that will trigger handover, i.e. roaming to another access point. At the same time, it will set the disassociation threshold to 3 times the given value.

4. About the Version Numbers

Due to the nature of open source development projects, there are frequently changes being incorporated that have not gone through a complete validation process. These changes are incorporated into development snapshot releases.

Releases are numbered with a three level scheme:

major.minor.development

Any version where the 'development' portion is 0 (for example 1.0.0, 1.1.0, etc.) indicates a stable version that will be made available for kernel inclusion.

Any version where the 'development' portion is not a 0 (for example 1.0.1, 1.1.5, etc.) indicates a development version that is being made available for testing and cutting edge users. The stability and functionality of the development releases are not known. We make efforts to try and keep all snapshots reasonably stable, but due to the frequency of their release, and the desire to get those releases available as quickly as possible, unknown anomalies should be expected.

The major version number will be incremented when significant changes are made to the driver. Currently, there are no major changes planned.

5. Firmware installation

The driver requires a firmware image, download it and extract the files under /lib/firmware (or wherever your hotplug's firmware.agent will look for firmware files)

The firmware can be downloaded from the following URL:

<http://ipw2200.sf.net/>

6. Support

For direct support of the 1.0.0 version, you can contact <http://supportmail.intel.com>, or you can use the open source project support.

For general information and support, go to:

<http://ipw2200.sf.net/>

7. License

Copyright © 2003 - 2006 Intel Corporation. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License version 2 as published by the Free Software Foundation.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

The full GNU General Public License is included in this distribution in the file called LICENSE.

Contact Information:

James P. Ketrenos <ipw2100-admin@linux.intel.com>

Intel Corporation, 5200 N.E. Elam Young Parkway, Hillsboro, OR 97124-6497

6.9 WWAN Device Drivers

Contents:

6.9.1 IOSM Driver for Intel M.2 PCIe based Modems

The IOSM (IPC over Shared Memory) driver is a WWAN PCIe host driver developed for linux or chrome platform for data exchange over PCIe interface between Host platform & Intel M.2 Modem. The driver exposes interface conforming to the MBIM protocol [1]. Any front end application (eg: Modem Manager) could easily manage the MBIM interface to enable data communication towards WWAN.

Basic usage

MBIM functions are inactive when unmanaged. The IOSM driver only provides a userspace interface MBIM "WWAN PORT" representing MBIM control channel and does not play any role in managing the functionality. It is the job of a userspace application to detect port enumeration and enable MBIM functionality.

Examples of few such userspace application are: - mbimcli (included with the libmbim [2] library), and - Modem Manager [3]

Management Applications to carry out below required actions for establishing MBIM IP session:

- open the MBIM control channel
- configure network connection settings
- connect to network
- configure IP network interface

Management application development

The driver and userspace interfaces are described below. The MBIM protocol is described in [1] Mobile Broadband Interface Model v1.0 Errata-1.

MBIM control channel userspace ABI

/dev/wwan0mbim0 character device

The driver exposes an MBIM interface to the MBIM function by implementing MBIM WWAN Port. The userspace end of the control channel pipe is a /dev/wwan0mbim0 character device. Application shall use this interface for MBIM protocol communication.

Fragmentation

The userspace application is responsible for all control message fragmentation and defragmentation as per MBIM specification.

/dev/wwan0mbim0 write()

The MBIM control messages from the management application must not exceed the negotiated control message size.

/dev/wwan0mbim0 read()

The management application must accept control messages of up the negotiated control message size.

MBIM data channel userspace ABI

wwan0-X network device

The IOSM driver exposes IP link interface "wwan0-X" of type "wwan" for IP traffic. Iproute network utility is used for creating "wwan0-X" network interface and for associating it with MBIM IP session. The Driver supports up to 8 IP sessions for simultaneous IP communication.

The userspace management application is responsible for creating new IP link prior to establishing MBIM IP session where the SessionId is greater than 0.

For example, creating new IP link for a MBIM IP session with SessionId 1:

```
ip link add dev wwan0-1 parentdev-name wwan0 type wwan linkid 1
```

The driver will automatically map the "wwan0-1" network device to MBIM IP session 1.

References

[1] "MBIM (Mobile Broadband Interface Model) Errata-1"

- <https://www.usb.org/document-library/>

[2] libmbim - "a glib-based library for talking to WWAN modems and

devices which speak the Mobile Interface Broadband Model (MBIM) protocol" - <http://www.freedesktop.org/wiki/Software/libmbim/>

[3] Modem Manager - "a DBus-activated daemon which controls mobile

broadband (2G/3G/4G) devices and connections" - <http://www.freedesktop.org/wiki/Software/ModemManager/>

6.9.2 t7xx driver for MTK PCIe based T700 5G modem

The t7xx driver is a WWAN PCIe host driver developed for linux or Chrome OS platforms for data exchange over PCIe interface between Host platform & MediaTek's T700 5G modem. The driver exposes an interface conforming to the MBIM protocol [1]. Any front end application (e.g. Modem Manager) could easily manage the MBIM interface to enable data communication towards WWAN. The driver also provides an interface to interact with the MediaTek's modem via AT commands.

Basic usage

MBIM & AT functions are inactive when unmanaged. The t7xx driver provides WWAN port userspace interfaces representing MBIM & AT control channels and does not play any role in managing their functionality. It is the job of a userspace application to detect port enumeration and enable MBIM & AT functionalities.

Examples of few such userspace applications are:

- mbimcli (included with the libmbim [2] library), and
- Modem Manager [3]

Management Applications to carry out below required actions for establishing MBIM IP session:

- open the MBIM control channel
- configure network connection settings
- connect to network
- configure IP network interface

Management Applications to carry out below required actions for send an AT command and receive response:

- open the AT control channel using a UART tool or a special user tool

Management application development

The driver and userspace interfaces are described below. The MBIM protocol is described in [1] Mobile Broadband Interface Model v1.0 Errata-1.

MBIM control channel userspace ABI

/dev/wwan0mbim0 character device

The driver exposes an MBIM interface to the MBIM function by implementing MBIM WWAN Port. The userspace end of the control channel pipe is a /dev/wwan0mbim0 character device. Application shall use this interface for MBIM protocol communication.

Fragmentation

The userspace application is responsible for all control message fragmentation and defragmentation as per MBIM specification.

/dev/wwan0mbim0 write()

The MBIM control messages from the management application must not exceed the negotiated control message size.

/dev/wwan0mbim0 read()

The management application must accept control messages of up the negotiated control message size.

MBIM data channel userspace ABI

wwan0-X network device

The t7xx driver exposes IP link interface "wwan0-X" of type "wwan" for IP traffic. Iproute network utility is used for creating "wwan0-X" network interface and for associating it with MBIM IP session.

The userspace management application is responsible for creating new IP link prior to establishing MBIM IP session where the SessionId is greater than 0.

For example, creating new IP link for a MBIM IP session with SessionId 1:

```
ip link add dev wwan0-1 parentdev wwan0 type wwan linkid 1
```

The driver will automatically map the "wwan0-1" network device to MBIM IP session 1.

AT port userspace ABI

/dev/wwan0at0 character device

The driver exposes an AT port by implementing AT WWAN Port. The userspace end of the control port is a /dev/wwan0at0 character device. Application shall use this interface to issue AT commands.

The MediaTek's T700 modem supports the 3GPP TS 27.007 [4] specification.

References

[1] *MBIM (Mobile Broadband Interface Model) Errata-1*

- <https://www.usb.org/document-library/>

[2] *libmbim "a glib-based library for talking to WWAN modems and devices which speak the Mobile Interface Broadband Model (MBIM) protocol"*

- <http://www.freedesktop.org/wiki/Software/libmbim/>

[3] *Modem Manager "a DBus-activated daemon which controls mobile broadband (2G/3G/4G/5G) devices and connections"*

- <http://www.freedesktop.org/wiki/Software/ModemManager/>

[4] *Specification # 27.007 - 3GPP*

- <https://www.3gpp.org/DynaReport/27007.htm>

DISTRIBUTED SWITCH ARCHITECTURE

7.1 Architecture

This document describes the **Distributed Switch Architecture (DSA)** subsystem design principles, limitations, interactions with other subsystems, and how to develop drivers for this subsystem as well as a TODO for developers interested in joining the effort.

7.1.1 Design principles

The Distributed Switch Architecture subsystem was primarily designed to support Marvell Ethernet switches (MV88E6xxx, a.k.a. Link Street product line) using Linux, but has since evolved to support other vendors as well.

The original philosophy behind this design was to be able to use unmodified Linux tools such as bridge, iproute2, ifconfig to work transparently whether they configured/queried a switch port network device or a regular network device.

An Ethernet switch typically comprises multiple front-panel ports and one or more CPU or management ports. The DSA subsystem currently relies on the presence of a management port connected to an Ethernet controller capable of receiving Ethernet frames from the switch. This is a very common setup for all kinds of Ethernet switches found in Small Home and Office products: routers, gateways, or even top-of-rack switches. This host Ethernet controller will be later referred to as "conduit" and "cpu" in DSA terminology and code.

The D in DSA stands for Distributed, because the subsystem has been designed with the ability to configure and manage cascaded switches on top of each other using upstream and downstream Ethernet links between switches. These specific ports are referred to as "dsa" ports in DSA terminology and code. A collection of multiple switches connected to each other is called a "switch tree".

For each front-panel port, DSA creates specialized network devices which are used as controlling and data-flowing endpoints for use by the Linux networking stack. These specialized network interfaces are referred to as "user" network interfaces in DSA terminology and code.

The ideal case for using DSA is when an Ethernet switch supports a "switch tag" which is a hardware feature making the switch insert a specific tag for each Ethernet frame it receives to/from specific ports to help the management interface figure out:

- what port is this frame coming from
- what was the reason why this frame got forwarded
- how to send CPU originated traffic to specific ports

The subsystem does support switches not capable of inserting/stripping tags, but the features might be slightly limited in that case (traffic separation relies on Port-based VLAN IDs).

Note that DSA does not currently create network interfaces for the "cpu" and "dsa" ports because:

- the "cpu" port is the Ethernet switch facing side of the management controller, and as such, would create a duplication of feature, since you would get two interfaces for the same conduit: conduit netdev, and "cpu" netdev
- the "dsa" port(s) are just conduits between two or more switches, and as such cannot really be used as proper network interfaces either, only the downstream, or the top-most upstream interface makes sense with that model

NB: for the past 15 years, the DSA subsystem had been making use of the terms "master" (rather than "conduit") and "slave" (rather than "user"). These terms have been removed from the DSA codebase and phased out of the uAPI.

Switch tagging protocols

DSA supports many vendor-specific tagging protocols, one software-defined tagging protocol, and a tag-less mode as well (`DSA_TAG_PROTO_NONE`).

The exact format of the tag protocol is vendor specific, but in general, they all contain something which:

- identifies which port the Ethernet frame came from/should be sent to
- provides a reason why this frame was forwarded to the management interface

All tagging protocols are in `net/dsa/tag_*.c` files and implement the methods of the `struct dsa_device_ops` structure, which are detailed below.

Tagging protocols generally fall in one of three categories:

1. The switch-specific frame header is located before the Ethernet header, shifting to the right (from the perspective of the DSA conduit's frame parser) the MAC DA, MAC SA, EtherType and the entire L2 payload.
2. The switch-specific frame header is located before the EtherType, keeping the MAC DA and MAC SA in place from the DSA conduit's perspective, but shifting the 'real' EtherType and L2 payload to the right.
3. The switch-specific frame header is located at the tail of the packet, keeping all frame headers in place and not altering the view of the packet that the DSA conduit's frame parser has.

A tagging protocol may tag all packets with switch tags of the same length, or the tag length might vary (for example packets with PTP timestamps might require an extended switch tag, or there might be one tag length on TX and a different one on RX). Either way, the tagging protocol driver must populate the `struct dsa_device_ops::needed_headroom` and/or `struct dsa_device_ops::needed_tailroom` with the length in octets of the longest switch frame header/trailer. The DSA framework will automatically adjust the MTU of the conduit interface to accommodate for this extra size in order for DSA user ports to support the standard MTU (L2 payload length) of 1500 octets. The `needed_headroom` and `needed_tailroom` properties are also used to request from the network stack, on a best-effort basis, the allocation of

packets with enough extra space such that the act of pushing the switch tag on transmission of a packet does not cause it to reallocate due to lack of memory.

Even though applications are not expected to parse DSA-specific frame headers, the format on the wire of the tagging protocol represents an Application Binary Interface exposed by the kernel towards user space, for decoders such as `libpcap`. The tagging protocol driver must populate the `proto` member of `struct dsa_device_ops` with a value that uniquely describes the characteristics of the interaction required between the switch hardware and the data path driver: the offset of each bit field within the frame header and any stateful processing required to deal with the frames (as may be required for PTP timestamping).

From the perspective of the network stack, all switches within the same DSA switch tree use the same tagging protocol. In case of a packet transiting a fabric with more than one switch, the switch-specific frame header is inserted by the first switch in the fabric that the packet was received on. This header typically contains information regarding its type (whether it is a control frame that must be trapped to the CPU, or a data frame to be forwarded). Control frames should be decapsulated only by the software data path, whereas data frames might also be autonomously forwarded towards other user ports of other switches from the same fabric, and in this case, the outermost switch ports must decapsulate the packet.

Note that in certain cases, it might be the case that the tagging format used by a leaf switch (not connected directly to the CPU) is not the same as what the network stack sees. This can be seen with Marvell switch trees, where the CPU port can be configured to use either the DSA or the Ethertype DSA (EDSA) format, but the DSA links are configured to use the shorter (without Ethertype) DSA frame header, in order to reduce the autonomous packet forwarding overhead. It still remains the case that, if the DSA switch tree is configured for the EDSA tagging protocol, the operating system sees EDSA-tagged packets from the leaf switches that tagged them with the shorter DSA header. This can be done because the Marvell switch connected directly to the CPU is configured to perform tag translation between DSA and EDSA (which is simply the operation of adding or removing the `ETH_P_EDSA` EtherType and some padding octets).

It is possible to construct cascaded setups of DSA switches even if their tagging protocols are not compatible with one another. In this case, there are no DSA links in this fabric, and each switch constitutes a disjoint DSA switch tree. The DSA links are viewed as simply a pair of a DSA conduit (the out-facing port of the upstream DSA switch) and a CPU port (the in-facing port of the downstream DSA switch).

The tagging protocol of the attached DSA switch tree can be viewed through the `dsa/tagging` sysfs attribute of the DSA conduit:

```
cat /sys/class/net/eth0/dsa/tagging
```

If the hardware and driver are capable, the tagging protocol of the DSA switch tree can be changed at runtime. This is done by writing the new tagging protocol name to the same sysfs device attribute as above (the DSA conduit and all attached switch ports must be down while doing this).

It is desirable that all tagging protocols are testable with the `dsa_loop` mockup driver, which can be attached to any network interface. The goal is that any network interface should be capable of transmitting the same packet in the same way, and the tagger should decode the same received packet in the same way regardless of the driver used for the switch control path, and the driver used for the DSA conduit.

The transmission of a packet goes through the tagger's `xmit` function. The passed `struct sk_buff *skb` has `skb->data` pointing at `skb_mac_header(skb)`, i.e. at the destination MAC

address, and the passed `struct net_device *dev` represents the virtual DSA user network interface whose hardware counterpart the packet must be steered to (i.e. `swp0`). The job of this method is to prepare the `skb` in a way that the switch will understand what egress port the packet is for (and not deliver it towards other ports). Typically this is fulfilled by pushing a frame header. Checking for insufficient size in the `skb` headroom or tailroom is unnecessary provided that the `needed_headroom` and `needed_tailroom` properties were filled out properly, because DSA ensures there is enough space before calling this method.

The reception of a packet goes through the tagger's `recv` function. The passed `struct sk_buff *skb` has `skb->data` pointing at `skb_mac_header(skb) + ETH_ALEN` octets, i.e. to where the first octet after the EtherType would have been, were this frame not tagged. The role of this method is to consume the frame header, adjust `skb->data` to really point at the first octet after the EtherType, and to change `skb->dev` to point to the virtual DSA user network interface corresponding to the physical front-facing switch port that the packet was received on.

Since tagging protocols in category 1 and 2 break software (and most often also hardware) packet dissection on the DSA conduit, features such as RPS (Receive Packet Steering) on the DSA conduit would be broken. The DSA framework deals with this by hooking into the flow dissector and shifting the offset at which the IP header is to be found in the tagged frame as seen by the DSA conduit. This behavior is automatic based on the overhead value of the tagging protocol. If not all packets are of equal size, the tagger can implement the `flow_dissect` method of the `struct dsa_device_ops` and override this default behavior by specifying the correct offset incurred by each individual RX packet. Tail taggers do not cause issues to the flow dissector.

Checksum offload should work with category 1 and 2 taggers when the DSA conduit driver declares `NETIF_F_HW_CSUM` in `vlan_features` and looks at `csum_start` and `csum_offset`. For those cases, DSA will shift the checksum start and offset by the tag size. If the DSA conduit driver still uses the legacy `NETIF_F_IP_CSUM` or `NETIF_F_IPV6_CSUM` in `vlan_features`, the offload might only work if the offload hardware already expects that specific tag (perhaps due to matching vendors). DSA user ports inherit those flags from the conduit, and it is up to the driver to correctly fall back to software checksum when the IP header is not where the hardware expects. If that check is ineffective, the packets might go to the network without a proper checksum (the checksum field will have the pseudo IP header sum). For category 3, when the offload hardware does not already expect the switch tag in use, the checksum must be calculated before any tag is inserted (i.e. inside the tagger). Otherwise, the DSA conduit would include the tail tag in the (software or hardware) checksum calculation. Then, when the tag gets stripped by the switch during transmission, it will leave an incorrect IP checksum in place.

Due to various reasons (most common being category 1 taggers being associated with DSA-unaware conduits, mangling what the conduit perceives as MAC DA), the tagging protocol may require the DSA conduit to operate in promiscuous mode, to receive all frames regardless of the value of the MAC DA. This can be done by setting the `promisc_on_conduit` property of the `struct dsa_device_ops`. Note that this assumes a DSA-unaware conduit driver, which is the norm.

Conduit network devices

Conduit network devices are regular, unmodified Linux network device drivers for the CPU/management Ethernet interface. Such a driver might occasionally need to know whether DSA is enabled (e.g.: to enable/disable specific offload features), but the DSA subsystem has been proven to work with industry standard drivers: `e1000e`, `mv643xx_eth` etc. without having to introduce modifications to these drivers. Such network devices are also often referred to as conduit network devices since they act as a pipe between the host processor and the hardware Ethernet switch.

Networking stack hooks

When a conduit netdev is used with DSA, a small hook is placed in the networking stack in order to have the DSA subsystem process the Ethernet switch specific tagging protocol. DSA accomplishes this by registering a specific (and fake) Ethernet type (later becoming `skb->protocol`) with the networking stack, this is also known as a `ptype` or `packet_type`. A typical Ethernet Frame receive sequence looks like this:

Conduit network device (e.g.: `e1000e`):

1. Receive interrupt fires:
 - receive function is invoked
 - basic packet processing is done: getting length, status etc.
 - packet is prepared to be processed by the Ethernet layer by calling `eth_type_trans`
2. net/ethernet/eth.c:

```
eth_type_trans(skb, dev)
    if (dev->dsa_ptr != NULL)
        -> skb->protocol = ETH_P_XDSA
```

3. drivers/net/ethernet/*:

```
netif_receive_skb(skb)
    -> iterate over registered packet_type
        -> invoke handler for ETH_P_XDSA, calls dsa_switch_rcv()
```

4. net/dsa/dsa.c:

```
-> dsa_switch_rcv()
    -> invoke switch tag specific protocol handler in 'net/dsa/tag_*.c'
```

5. net/dsa/tag_*.c:

- inspect and strip switch tag protocol to determine originating port
- locate per-port network device
- invoke `eth_type_trans()` with the DSA user network device
- invoked `netif_receive_skb()`

Past this point, the DSA user network devices get delivered regular Ethernet frames that can be processed by the networking stack.

User network devices

User network devices created by DSA are stacked on top of their conduit network device, each of these network interfaces will be responsible for being a controlling and data-flowing endpoint for each front-panel port of the switch. These interfaces are specialized in order to:

- insert/remove the switch tag protocol (if it exists) when sending traffic to/from specific switch ports
- query the switch for ethtool operations: statistics, link state, Wake-on-LAN, register dumps...
- manage external/internal PHY: link, auto-negotiation, etc.

These user network devices have custom `net_device_ops` and `ethtool_ops` function pointers which allow DSA to introduce a level of layering between the networking stack/ethtool and the switch driver implementation.

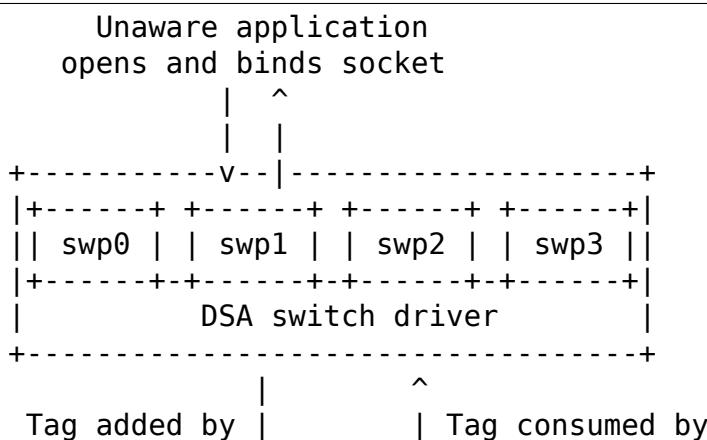
Upon frame transmission from these user network devices, DSA will look up which switch tagging protocol is currently registered with these network devices and invoke a specific transmit routine which takes care of adding the relevant switch tag in the Ethernet frames.

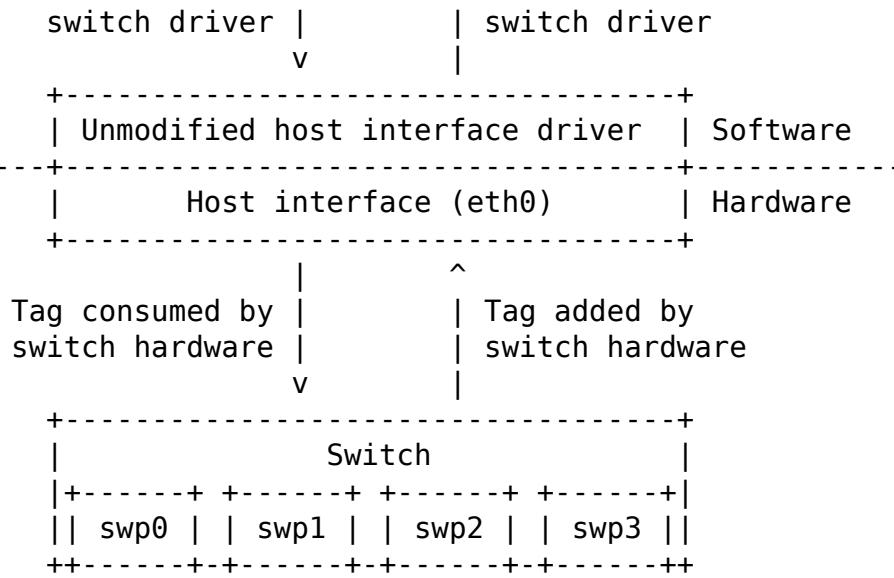
These frames are then queued for transmission using the conduit network device `ndo_start_xmit()` function. Since they contain the appropriate switch tag, the Ethernet switch will be able to process these incoming frames from the management interface and deliver them to the physical switch port.

When using multiple CPU ports, it is possible to stack a LAG (bonding/team) device between the DSA user devices and the physical DSA conduits. The LAG device is thus also a DSA conduit, but the LAG slave devices continue to be DSA conduits as well (just with no user port assigned to them; this is needed for recovery in case the LAG DSA conduit disappears). Thus, the data path of the LAG DSA conduit is used asymmetrically. On RX, the `ETH_P_XDSA` handler, which calls `dsa_switch_rcv()`, is invoked early (on the physical DSA conduit; LAG slave). Therefore, the RX data path of the LAG DSA conduit is not used. On the other hand, TX takes place linearly: `dsa_user_xmit` calls `dsa_enqueue_skb`, which calls `dev_queue_xmit` towards the LAG DSA conduit. The latter calls `dev_queue_xmit` towards one physical DSA conduit or the other, and in both cases, the packet exits the system through a hardware path towards the switch.

Graphical representation

Summarized, this is basically how DSA looks like from a network device perspective:





User MDIO bus

In order to be able to read to/from a switch PHY built into it, DSA creates an user MDIO bus which allows a specific switch driver to divert and intercept MDIO reads/writes towards specific PHY addresses. In most MDIO-connected switches, these functions would utilize direct or indirect PHY addressing mode to return standard MII registers from the switch builtin PHYs, allowing the PHY library and/or to return link status, link partner pages, auto-negotiation results, etc.

For Ethernet switches which have both external and internal MDIO buses, the user MII bus can be utilized to mux/demux MDIO reads and writes towards either internal or external MDIO devices this switch might be connected to: internal PHYs, external PHYs, or even external switches.

Data structures

DSA data structures are defined in `include/net/dsa.h` as well as `net/dsa/dsa_priv.h`:

- `dsa_chip_data`: platform data configuration for a given switch device, this structure describes a switch device's parent device, its address, as well as various properties of its ports: names/labels, and finally a routing table indication (when cascading switches)
- `dsa_platform_data`: platform device configuration data which can reference a collection of `dsa_chip_data` structures if multiple switches are cascaded, the conduit network device this switch tree is attached to needs to be referenced
- `dsa_switch_tree`: structure assigned to the conduit network device under `dsa_ptr`, this structure references a `dsa_platform_data` structure as well as the tagging protocol supported by the switch tree, and which receive/transmit function hooks should be invoked, information about the directly attached switch is also provided: CPU port. Finally, a collection of `dsa_switch` are referenced to address individual switches in the tree.
- `dsa_switch`: structure describing a switch device in the tree, referencing a `dsa_switch_tree` as a backpointer, user network devices, conduit network device, and

- a reference to the backing ```dsa_switch_ops```
- `dsa_switch_ops`: structure referencing function pointers, see below for a full description.

7.1.2 Design limitations

Lack of CPU/DSA network devices

DSA does not currently create user network devices for the CPU or DSA ports, as described before. This might be an issue in the following cases:

- inability to fetch switch CPU port statistics counters using ethtool, which can make it harder to debug MDIO switch connected using xMII interfaces
- inability to configure the CPU port link parameters based on the Ethernet controller capabilities attached to it: <http://patchwork.ozlabs.org/patch/509806/>
- inability to configure specific VLAN IDs / trunking VLANs between switches when using a cascaded setup

Common pitfalls using DSA setups

Once a conduit network device is configured to use DSA (`dev->dsa_ptr` becomes non-NUL), and the switch behind it expects a tagging protocol, this network interface can only exclusively be used as a conduit interface. Sending packets directly through this interface (e.g.: opening a socket using this interface) will not make us go through the switch tagging protocol transmit function, so the Ethernet switch on the other end, expecting a tag will typically drop this frame.

7.1.3 Interactions with other subsystems

DSA currently leverages the following subsystems:

- MDIO/PHY library: `drivers/net/phy/phy.c`, `mdio_bus.c`
- Switchdev: `net/switchdev/*`
- Device Tree for various of `_*` functions
- Devlink: `net/core/devlink.c`

MDIO/PHY library

User network devices exposed by DSA may or may not be interfacing with PHY devices (`struct phy_device` as defined in `include/linux/phy.h`), but the DSA subsystem deals with all possible combinations:

- internal PHY devices, built into the Ethernet switch hardware
- external PHY devices, connected via an internal or external MDIO bus
- internal PHY devices, connected via an internal MDIO bus
- special, non-autonegotiated or non MDIO-managed PHY devices: SFPs, MoCA; a.k.a fixed PHYs

The PHY configuration is done by the `dsa_user_phy_setup()` function and the logic basically looks like this:

- if Device Tree is used, the PHY device is looked up using the standard "phy-handle" property, if found, this PHY device is created and registered using `of_phy_connect()`
- if Device Tree is used and the PHY device is "fixed", that is, conforms to the definition of a non-MDIO managed PHY as defined in `Documentation/devicetree/bindings/net/fixed-link.txt`, the PHY is registered and connected transparently using the special fixed MDIO bus driver
- finally, if the PHY is built into the switch, as is very common with standalone switch packages, the PHY is probed using the user MII bus created by DSA

SWITCHDEV

DSA directly utilizes SWITCHDEV when interfacing with the bridge layer, and more specifically with its VLAN filtering portion when configuring VLANs on top of per-port user network devices. As of today, the only SWITCHDEV objects supported by DSA are the FDB and VLAN objects.

Devlink

DSA registers one devlink device per physical switch in the fabric. For each devlink device, every physical port (i.e. user ports, CPU ports, DSA links or unused ports) is exposed as a devlink port.

DSA drivers can make use of the following devlink features:

- Regions: debugging feature which allows user space to dump driver-defined areas of hardware information in a low-level, binary format. Both global regions as well as per-port regions are supported. It is possible to export devlink regions even for pieces of data that are already exposed in some way to the standard iproute2 user space programs (ip-link, bridge), like address tables and VLAN tables. For example, this might be useful if the tables contain additional hardware-specific details which are not visible through the iproute2 abstraction, or it might be useful to inspect these tables on the non-user ports too, which are invisible to iproute2 because no network interface is registered for them.
- Params: a feature which enables user to configure certain low-level tunable knobs pertaining to the device. Drivers may implement applicable generic devlink params, or may add new device-specific devlink params.
- Resources: a monitoring feature which enables users to see the degree of utilization of certain hardware tables in the device, such as FDB, VLAN, etc.
- Shared buffers: a QoS feature for adjusting and partitioning memory and frame reservations per port and per traffic class, in the ingress and egress directions, such that low-priority bulk traffic does not impede the processing of high-priority critical traffic.

For more details, consult [Documentation/networking/devlink/](#).

Device Tree

DSA features a standardized binding which is documented in [Documentation/devicetree/bindings/net/dsa/dsa.txt](#). PHY/MDIO library helper functions such as `of_get_phy_mode()`, `of_phy_connect()` are also used to query per-port PHY specific details: interface connection, MDIO bus location, etc.

7.1.4 Driver development

DSA switch drivers need to implement a `dsa_switch_ops` structure which will contain the various members described below.

Probing, registration and device lifetime

DSA switches are regular device structures on buses (be they platform, SPI, I2C, MDIO or otherwise). The DSA framework is not involved in their probing with the device core.

Switch registration from the perspective of a driver means passing a valid `struct dsa_switch` pointer to `dsa_register_switch()`, usually from the switch driver's probing function. The following members must be valid in the provided structure:

- `ds->dev`: will be used to parse the switch's OF node or platform data.
- `ds->num_ports`: will be used to create the port list for this switch, and to validate the port indices provided in the OF node.
- `ds->ops`: a pointer to the `dsa_switch_ops` structure holding the DSA method implementations.
- `ds->priv`: backpointer to a driver-private data structure which can be retrieved in all further DSA method callbacks.

In addition, the following flags in the `dsa_switch` structure may optionally be configured to obtain driver-specific behavior from the DSA core. Their behavior when set is documented through comments in `include/net/dsa.h`.

- `ds->vlan_filtering_is_global`
- `ds->needs_standalone_vlan_filtering`
- `ds->configure_vlan_while_not_filtering`
- `ds->untag_bridge_pvid`
- `ds->assisted_learning_on_cpu_port`
- `ds->mtu_enforcement_ingress`
- `ds->fdb_isolation`

Internally, DSA keeps an array of switch trees (group of switches) global to the kernel, and attaches a `dsa_switch` structure to a tree on registration. The tree ID to which the switch is attached is determined by the first u32 number of the `dsa_member` property of the switch's OF node (0 if missing). The switch ID within the tree is determined by the second u32 number of the same OF property (0 if missing). Registering multiple switches with the same switch ID and tree ID is illegal and will cause an error. Using platform data, a single switch and a single switch tree is permitted.

In case of a tree with multiple switches, probing takes place asymmetrically. The first N-1 callers of `dsa_register_switch()` only add their ports to the port list of the tree (`dst->ports`), each port having a backpointer to its associated switch (`dp->ds`). Then, these switches exit their `dsa_register_switch()` call early, because `dsa_tree_setup_routing_table()` has determined that the tree is not yet complete (not all ports referenced by DSA links are present in the tree's port list). The tree becomes complete when the last switch calls `dsa_register_switch()`, and this triggers the effective continuation of initialization (including the call to `ds->ops->setup()` for all switches within that tree, all as part of the calling context of the last switch's probe function).

The opposite of registration takes place when calling `dsa_unregister_switch()`, which removes a switch's ports from the port list of the tree. The entire tree is torn down when the first switch unregisters.

It is mandatory for DSA switch drivers to implement the `shutdown()` callback of their respective bus, and call `dsa_switch_shutdown()` from it (a minimal version of the full teardown performed by `dsa_unregister_switch()`). The reason is that DSA keeps a reference on the conduit net device, and if the driver for the conduit device decides to unbind on shutdown, DSA's reference will block that operation from finalizing.

Either `dsa_switch_shutdown()` or `dsa_unregister_switch()` must be called, but not both, and the device driver model permits the bus' `remove()` method to be called even if `shutdown()` was already called. Therefore, drivers are expected to implement a mutual exclusion method between `remove()` and `shutdown()` by setting their `drvdata` to `NULL` after any of these has run, and checking whether the `drvdata` is `NULL` before proceeding to take any action.

After `dsa_switch_shutdown()` or `dsa_unregister_switch()` was called, no further callbacks via the provided `dsa_switch_ops` may take place, and the driver may free the data structures associated with the `dsa_switch`.

Switch configuration

- `get_tag_protocol`: this is to indicate what kind of tagging protocol is supported, should be a valid value from the `dsa_tag_protocol` enum. The returned information does not have to be static; the driver is passed the CPU port number, as well as the tagging protocol of a possibly stacked upstream switch, in case there are hardware limitations in terms of supported tag formats.
- `change_tag_protocol`: when the default tagging protocol has compatibility problems with the conduit or other issues, the driver may support changing it at runtime, either through a device tree property or through sysfs. In that case, further calls to `get_tag_protocol` should report the protocol in current use.
- `setup`: setup function for the switch, this function is responsible for setting up the `dsa_switch_ops` private structure with all it needs: register maps, interrupts, mutexes, locks, etc. This function is also expected to properly configure the switch to separate all network interfaces from each other, that is, they should be isolated by the switch hardware itself, typically by creating a Port-based VLAN ID for each port and allowing only the CPU port and the specific port to be in the forwarding vector. Ports that are unused by the platform should be disabled. Past this function, the switch is expected to be fully configured and ready to serve any kind of request. It is recommended to issue a software reset of the switch during this setup function in order to avoid relying on what a previous software agent such as a bootloader/firmware may have previously configured. The method responsible for undoing any applicable allocations or operations done here is `teardown`.

- `port_setup` and `port_teardown`: methods for initialization and destruction of per-port data structures. It is mandatory for some operations such as registering and unregistering devlink port regions to be done from these methods, otherwise they are optional. A port will be torn down only if it has been previously set up. It is possible for a port to be set up during probing only to be torn down immediately afterwards, for example in case its PHY cannot be found. In this case, probing of the DSA switch continues without that particular port.
- `port_change_conduit`: method through which the affinity (association used for traffic termination purposes) between a user port and a CPU port can be changed. By default all user ports from a tree are assigned to the first available CPU port that makes sense for them (most of the times this means the user ports of a tree are all assigned to the same CPU port, except for H topologies as described in [commit 2c0b03258b8b](#)). The `port` argument represents the index of the user port, and the `conduit` argument represents the new DSA conduit `net_device`. The CPU port associated with the new conduit can be retrieved by looking at `struct dsa_port *cpu_dp = conduit->dsa_ptr`. Additionally, the conduit can also be a LAG device where all the slave devices are physical DSA conduits. LAG DSA also have a valid `conduit->dsa_ptr` pointer, however this is not unique, but rather a duplicate of the first physical DSA conduit's (LAG slave) `dsa_ptr`. In case of a LAG DSA conduit, a further call to `port_lag_join` will be emitted separately for the physical CPU ports associated with the physical DSA conduits, requesting them to create a hardware LAG associated with the LAG interface.

PHY devices and link management

- `get_phy_flags`: Some switches are interfaced to various kinds of Ethernet PHYs, if the PHY library PHY driver needs to know about information it cannot obtain on its own (e.g.: coming from switch memory mapped registers), this function should return a 32-bit bitmask of "flags" that is private between the switch driver and the Ethernet PHY driver in `drivers/net/phy/*`.
- `phy_read`: Function invoked by the DSA user MDIO bus when attempting to read the switch port MDIO registers. If unavailable, return 0xffff for each read. For builtin switch Ethernet PHYs, this function should allow reading the link status, auto-negotiation results, link partner pages, etc.
- `phy_write`: Function invoked by the DSA user MDIO bus when attempting to write to the switch port MDIO registers. If unavailable return a negative error code.
- `adjust_link`: Function invoked by the PHY library when a user network device is attached to a PHY device. This function is responsible for appropriately configuring the switch port link parameters: speed, duplex, pause based on what the `phy_device` is providing.
- `fixed_link_update`: Function invoked by the PHY library, and specifically by the fixed PHY driver asking the switch driver for link parameters that could not be auto-negotiated, or obtained by reading the PHY registers through MDIO. This is particularly useful for specific kinds of hardware such as QSGMII, MoCA or other kinds of non-MDIO managed PHYs where out of band link information is obtained

Ethtool operations

- `get_strings`: ethtool function used to query the driver's strings, will typically return statistics strings, private flags strings, etc.
- `get_ethtool_stats`: ethtool function used to query per-port statistics and return their values. DSA overlays user network devices general statistics: RX/TX counters from the network device, with switch driver specific statistics per port
- `get_sset_count`: ethtool function used to query the number of statistics items
- `get_wol`: ethtool function used to obtain Wake-on-LAN settings per-port, this function may for certain implementations also query the conduit network device Wake-on-LAN settings if this interface needs to participate in Wake-on-LAN
- `set_wol`: ethtool function used to configure Wake-on-LAN settings per-port, direct counterpart to `set_wol` with similar restrictions
- `set_eee`: ethtool function which is used to configure a switch port EEE (Green Ethernet) settings, can optionally invoke the PHY library to enable EEE at the PHY level if relevant. This function should enable EEE at the switch port MAC controller and data-processing logic
- `get_eee`: ethtool function which is used to query a switch port EEE settings, this function should return the EEE state of the switch port MAC controller and data-processing logic as well as query the PHY for its currently configured EEE settings
- `get_eeprom_len`: ethtool function returning for a given switch the EEPROM length/size in bytes
- `get_eeprom`: ethtool function returning for a given switch the EEPROM contents
- `set_eeprom`: ethtool function writing specified data to a given switch EEPROM
- `get_regs_len`: ethtool function returning the register length for a given switch
- `get_regs`: ethtool function returning the Ethernet switch internal register contents. This function might require user-land code in ethtool to pretty-print register values and registers

Power management

- `suspend`: function invoked by the DSA platform device when the system goes to suspend, should quiesce all Ethernet switch activities, but keep ports participating in Wake-on-LAN active as well as additional wake-up logic if supported
- `resume`: function invoked by the DSA platform device when the system resumes, should resume all Ethernet switch activities and re-configure the switch to be in a fully active state
- `port_enable`: function invoked by the DSA user network device `ndo_open` function when a port is administratively brought up, this function should fully enable a given switch port. DSA takes care of marking the port with `BR_STATE_BLOCKING` if the port is a bridge member, or `BR_STATE_FORWARDING` if it was not, and propagating these changes down to the hardware
- `port_disable`: function invoked by the DSA user network device `ndo_close` function when a port is administratively brought down, this function should fully disable a given switch

port. DSA takes care of marking the port with `BR_STATE_DISABLED` and propagating changes to the hardware if this port is disabled while being a bridge member

Address databases

Switching hardware is expected to have a table for FDB entries, however not all of them are active at the same time. An address database is the subset (partition) of FDB entries that is active (can be matched by address learning on RX, or FDB lookup on TX) depending on the state of the port. An address database may occasionally be called "FID" (Filtering ID) in this document, although the underlying implementation may choose whatever is available to the hardware.

For example, all ports that belong to a VLAN-unaware bridge (which is *currently* VLAN-unaware) are expected to learn source addresses in the database associated by the driver with that bridge (and not with other VLAN-unaware bridges). During forwarding and FDB lookup, a packet received on a VLAN-unaware bridge port should be able to find a VLAN-unaware FDB entry having the same MAC DA as the packet, which is present on another port member of the same bridge. At the same time, the FDB lookup process must be able to not find an FDB entry having the same MAC DA as the packet, if that entry points towards a port which is a member of a different VLAN-unaware bridge (and is therefore associated with a different address database).

Similarly, each VLAN of each offloaded VLAN-aware bridge should have an associated address database, which is shared by all ports which are members of that VLAN, but not shared by ports belonging to different bridges that are members of the same VID.

In this context, a VLAN-unaware database means that all packets are expected to match on it irrespective of VLAN ID (only MAC address lookup), whereas a VLAN-aware database means that packets are supposed to match based on the VLAN ID from the classified 802.1Q header (or the pvid if untagged).

At the bridge layer, VLAN-unaware FDB entries have the special VID value of 0, whereas VLAN-aware FDB entries have non-zero VID values. Note that a VLAN-unaware bridge may have VLAN-aware (non-zero VID) FDB entries, and a VLAN-aware bridge may have VLAN-unaware FDB entries. As in hardware, the software bridge keeps separate address databases, and offloads to hardware the FDB entries belonging to these databases, through switchdev, asynchronously relative to the moment when the databases become active or inactive.

When a user port operates in standalone mode, its driver should configure it to use a separate database called a port private database. This is different from the databases described above, and should impede operation as standalone port (packet in, packet out to the CPU port) as little as possible. For example, on ingress, it should not attempt to learn the MAC SA of ingress traffic, since learning is a bridging layer service and this is a standalone port, therefore it would consume useless space. With no address learning, the port private database should be empty in a naive implementation, and in this case, all received packets should be trivially flooded to the CPU port.

DSA (cascade) and CPU ports are also called "shared" ports because they service multiple address databases, and the database that a packet should be associated to is usually embedded in the DSA tag. This means that the CPU port may simultaneously transport packets coming from a standalone port (which were classified by hardware in one address database), and from a bridge port (which were classified to a different address database).

Switch drivers which satisfy certain criteria are able to optimize the naive configuration by

removing the CPU port from the flooding domain of the switch, and just program the hardware with FDB entries pointing towards the CPU port for which it is known that software is interested in those MAC addresses. Packets which do not match a known FDB entry will not be delivered to the CPU, which will save CPU cycles required for creating an skb just to drop it.

DSA is able to perform host address filtering for the following kinds of addresses:

- Primary unicast MAC addresses of ports (`dev->dev_addr`). These are associated with the port private database of the respective user port, and the driver is notified to install them through `port_fdb_add` towards the CPU port.
- Secondary unicast and multicast MAC addresses of ports (addresses added through `dev_uc_add()` and `dev_mc_add()`). These are also associated with the port private database of the respective user port.
- Local/permanent bridge FDB entries (`BR_FDB_LOCAL`). These are the MAC addresses of the bridge ports, for which packets must be terminated locally and not forwarded. They are associated with the address database for that bridge.
- Static bridge FDB entries installed towards foreign (non-DSA) interfaces present in the same bridge as some DSA switch ports. These are also associated with the address database for that bridge.
- Dynamically learned FDB entries on foreign interfaces present in the same bridge as some DSA switch ports, only if `ds->assisted_learning_on_cpu_port` is set to true by the driver. These are associated with the address database for that bridge.

For various operations detailed below, DSA provides a `dsa_db` structure which can be of the following types:

- `DSA_DB_PORT`: the FDB (or MDB) entry to be installed or deleted belongs to the port private database of user port `db->dp`.
- `DSA_DB_BRIDGE`: the entry belongs to one of the address databases of bridge `db->bridge`. Separation between the VLAN-unaware database and the per-VID databases of this bridge is expected to be done by the driver.
- `DSA_DB_LAG`: the entry belongs to the address database of LAG `db->lag`. Note: `DSA_DB_LAG` is currently unused and may be removed in the future.

The drivers which act upon the `dsa_db` argument in `port_fdb_add`, `port_mdb_add` etc should declare `ds->fdb_isolation` as true.

DSA associates each offloaded bridge and each offloaded LAG with a one-based ID (`struct dsa_bridge :: num`, `struct dsa_lag :: id`) for the purposes of refcounting addresses on shared ports. Drivers may piggyback on DSA's numbering scheme (the ID is readable through `db->bridge.num` and `db->lag.id` or may implement their own).

Only the drivers which declare support for FDB isolation are notified of FDB entries on the CPU port belonging to `DSA_DB_PORT` databases. For compatibility/legacy reasons, `DSA_DB_BRIDGE` addresses are notified to drivers even if they do not support FDB isolation. However, `db->bridge.num` and `db->lag.id` are always set to 0 in that case (to denote the lack of isolation, for refcounting purposes).

Note that it is not mandatory for a switch driver to implement physically separate address databases for each standalone user port. Since FDB entries in the port private databases will always point to the CPU port, there is no risk for incorrect forwarding decisions. In this

case, all standalone ports may share the same database, but the reference counting of host-filtered addresses (not deleting the FDB entry for a port's MAC address if it's still in use by another port) becomes the responsibility of the driver, because DSA is unaware that the port databases are in fact shared. This can be achieved by calling `dsa_fdb_present_in_other_db()` and `dsa_mdb_present_in_other_db()`. The down side is that the RX filtering lists of each user port are in fact shared, which means that user port A may accept a packet with a MAC DA it shouldn't have, only because that MAC address was in the RX filtering list of user port B. These packets will still be dropped in software, however.

Bridge layer

Offloading the bridge forwarding plane is optional and handled by the methods below. They may be absent, return -EOPNOTSUPP, or `ds->max_num_bridges` may be non-zero and exceeded, and in this case, joining a bridge port is still possible, but the packet forwarding will take place in software, and the ports under a software bridge must remain configured in the same way as for standalone operation, i.e. have all bridging service functions (address learning etc) disabled, and send all received packets to the CPU port only.

Concretely, a port starts offloading the forwarding plane of a bridge once it returns success to the `port_bridge_join` method, and stops doing so after `port_bridge_leave` has been called. Offloading the bridge means autonomously learning FDB entries in accordance with the software bridge port's state, and autonomously forwarding (or flooding) received packets without CPU intervention. This is optional even when offloading a bridge port. Tagging protocol drivers are expected to call `dsa_default_offload_fwd_mark(skb)` for packets which have already been autonomously forwarded in the forwarding domain of the ingress switch port. DSA, through `dsa_port_devlink_setup()`, considers all switch ports part of the same tree ID to be part of the same bridge forwarding domain (capable of autonomous forwarding to each other).

Offloading the TX forwarding process of a bridge is a distinct concept from simply offloading its forwarding plane, and refers to the ability of certain driver and tag protocol combinations to transmit a single skb coming from the bridge device's transmit function to potentially multiple egress ports (and thereby avoid its cloning in software).

Packets for which the bridge requests this behavior are called data plane packets and have `skb->offload_fwd_mark` set to true in the tag protocol driver's `xmit` function. Data plane packets are subject to FDB lookup, hardware learning on the CPU port, and do not override the port STP state. Additionally, replication of data plane packets (multicast, flooding) is handled in hardware and the bridge driver will transmit a single skb for each packet that may or may not need replication.

When the TX forwarding offload is enabled, the tag protocol driver is responsible to inject packets into the data plane of the hardware towards the correct bridging domain (FID) that the port is a part of. The port may be VLAN-unaware, and in this case the FID must be equal to the FID used by the driver for its VLAN-unaware address database associated with that bridge. Alternatively, the bridge may be VLAN-aware, and in that case, it is guaranteed that the packet is also VLAN-tagged with the VLAN ID that the bridge processed this packet in. It is the responsibility of the hardware to untag the VID on the egress-untagged ports, or keep the tag on the egress-tagged ones.

- `port_bridge_join`: bridge layer function invoked when a given switch port is added to a bridge, this function should do what's necessary at the switch level to permit the joining port to be added to the relevant logical domain for it to ingress/egress traffic with

other members of the bridge. By setting the `tx_fwd_offload` argument to true, the TX forwarding process of this bridge is also offloaded.

- `port_bridge_leave`: bridge layer function invoked when a given switch port is removed from a bridge, this function should do what's necessary at the switch level to deny the leaving port from ingress/egress traffic from the remaining bridge members.
- `port_stp_state_set`: bridge layer function invoked when a given switch port STP state is computed by the bridge layer and should be propagated to switch hardware to forward/block/learn traffic.
- `port_bridge_flags`: bridge layer function invoked when a port must configure its settings for e.g. flooding of unknown traffic or source address learning. The switch driver is responsible for initial setup of the standalone ports with address learning disabled and egress flooding of all types of traffic, then the DSA core notifies of any change to the bridge port flags when the port joins and leaves a bridge. DSA does not currently manage the bridge port flags for the CPU port. The assumption is that address learning should be statically enabled (if supported by the hardware) on the CPU port, and flooding towards the CPU port should also be enabled, due to a lack of an explicit address filtering mechanism in the DSA core.
- `port_fast_age`: bridge layer function invoked when flushing the dynamically learned FDB entries on the port is necessary. This is called when transitioning from an STP state where learning should take place to an STP state where it shouldn't, or when leaving a bridge, or when address learning is turned off via `port_bridge_flags`.

Bridge VLAN filtering

- `port_vlan_filtering`: bridge layer function invoked when the bridge gets configured for turning on or off VLAN filtering. If nothing specific needs to be done at the hardware level, this callback does not need to be implemented. When VLAN filtering is turned on, the hardware must be programmed with rejecting 802.1Q frames which have VLAN IDs outside of the programmed allowed VLAN ID map/rules. If there is no PVID programmed into the switch port, untagged frames must be rejected as well. When turned off the switch must accept any 802.1Q frames irrespective of their VLAN ID, and untagged frames are allowed.
- `port_vlan_add`: bridge layer function invoked when a VLAN is configured (tagged or untagged) for the given switch port. The CPU port becomes a member of a VLAN only if a foreign bridge port is also a member of it (and forwarding needs to take place in software), or the VLAN is installed to the VLAN group of the bridge device itself, for termination purposes (`bridge vlan add dev br0 vid 100 self`). VLANs on shared ports are reference counted and removed when there is no user left. Drivers do not need to manually install a VLAN on the CPU port.
- `port_vlan_del`: bridge layer function invoked when a VLAN is removed from the given switch port
- `port_fdb_add`: bridge layer function invoked when the bridge wants to install a Forwarding Database entry, the switch hardware should be programmed with the specified address in the specified VLAN Id in the forwarding database associated with this VLAN ID.
- `port_fdb_del`: bridge layer function invoked when the bridge wants to remove a Forwarding Database entry, the switch hardware should be programmed to delete the speci-

fied MAC address from the specified VLAN ID if it was mapped into this port forwarding database

- `port_fdb_dump`: bridge bypass function invoked by `ndo_fdb_dump` on the physical DSA port interfaces. Since DSA does not attempt to keep in sync its hardware FDB entries with the software bridge, this method is implemented as a means to view the entries visible on user ports in the hardware database. The entries reported by this function have the `self` flag in the output of the `bridge fdb show` command.
- `port_mdb_add`: bridge layer function invoked when the bridge wants to install a multicast database entry. The switch hardware should be programmed with the specified address in the specified VLAN ID in the forwarding database associated with this VLAN ID.
- `port_mdb_del`: bridge layer function invoked when the bridge wants to remove a multicast database entry, the switch hardware should be programmed to delete the specified MAC address from the specified VLAN ID if it was mapped into this port forwarding database.

Link aggregation

Link aggregation is implemented in the Linux networking stack by the bonding and team drivers, which are modeled as virtual, stackable network interfaces. DSA is capable of offloading a link aggregation group (LAG) to hardware that supports the feature, and supports bridging between physical ports and LAGs, as well as between LAGs. A bonding/team interface which holds multiple physical ports constitutes a logical port, although DSA has no explicit concept of a logical port at the moment. Due to this, events where a LAG joins/leaves a bridge are treated as if all individual physical ports that are members of that LAG join/leave the bridge. Switchdev port attributes (VLAN filtering, STP state, etc) and objects (VLANs, MDB entries) offloaded to a LAG as bridge port are treated similarly: DSA offloads the same switchdev object / port attribute on all members of the LAG. Static bridge FDB entries on a LAG are not yet supported, since the DSA driver API does not have the concept of a logical port ID.

- `port_lag_join`: function invoked when a given switch port is added to a LAG. The driver may return `-EOPNOTSUPP`, and in this case, DSA will fall back to a software implementation where all traffic from this port is sent to the CPU.
- `port_lag_leave`: function invoked when a given switch port leaves a LAG and returns to operation as a standalone port.
- `port_lag_change`: function invoked when the link state of any member of the LAG changes, and the hashing function needs rebalancing to only make use of the subset of physical LAG member ports that are up.

Drivers that benefit from having an ID associated with each offloaded LAG can optionally populate `ds->num_lag_ids` from the `dsa_switch_ops::setup` method. The LAG ID associated with a bonding/team interface can then be retrieved by a DSA switch driver using the `dsa_lag_id` function.

IEC 62439-2 (MRP)

The Media Redundancy Protocol is a topology management protocol optimized for fast fault recovery time for ring networks, which has some components implemented as a function of the bridge driver. MRP uses management PDUs (Test, Topology, LinkDown/Up, Option) sent at a multicast destination MAC address range of 01:15:4e:00:00:0x and with an EtherType of 0x88e3. Depending on the node's role in the ring (MRM: Media Redundancy Manager, MRC: Media Redundancy Client, MRA: Media Redundancy Automanager), certain MRP PDUs might need to be terminated locally and others might need to be forwarded. An MRM might also benefit from offloading to hardware the creation and transmission of certain MRP PDUs (Test).

Normally an MRP instance can be created on top of any network interface, however in the case of a device with an offloaded data path such as DSA, it is necessary for the hardware, even if it is not MRP-aware, to be able to extract the MRP PDUs from the fabric before the driver can proceed with the software implementation. DSA today has no driver which is MRP-aware, therefore it only listens for the bare minimum switchdev objects required for the software assist to work properly. The operations are detailed below.

- `port_mrp_add` and `port_mrp_del`: notifies driver when an MRP instance with a certain ring ID, priority, primary port and secondary port is created/deleted.
- `port_mrp_add_ring_role` and `port_mrp_del_ring_role`: function invoked when an MRP instance changes ring roles between MRM or MRC. This affects which MRP PDUs should be trapped to software and which should be autonomously forwarded.

IEC 62439-3 (HSR/PRP)

The Parallel Redundancy Protocol (PRP) is a network redundancy protocol which works by duplicating and sequence numbering packets through two independent L2 networks (which are unaware of the PRP tail tags carried in the packets), and eliminating the duplicates at the receiver. The High-availability Seamless Redundancy (HSR) protocol is similar in concept, except all nodes that carry the redundant traffic are aware of the fact that it is HSR-tagged (because HSR uses a header with an EtherType of 0x892f) and are physically connected in a ring topology. Both HSR and PRP use supervision frames for monitoring the health of the network and for discovery of other nodes.

In Linux, both HSR and PRP are implemented in the `hsr` driver, which instantiates a virtual, stackable network interface with two member ports. The driver only implements the basic roles of DANH (Doubly Attached Node implementing HSR) and DANP (Doubly Attached Node implementing PRP); the roles of RedBox and QuadBox are not implemented (therefore, bridging a `hsr` network interface with a physical switch port does not produce the expected result).

A driver which is able of offloading certain functions of a DANP or DANH should declare the corresponding netdev features as indicated by the documentation at [Documentation/networking/netdev-features.rst](#). Additionally, the following methods must be implemented:

- `port_hsr_join`: function invoked when a given switch port is added to a DANP/DANH. The driver may return `-EOPNOTSUPP` and in this case, DSA will fall back to a software implementation where all traffic from this port is sent to the CPU.
- `port_hsr_leave`: function invoked when a given switch port leaves a DANP/DANH and returns to normal operation as a standalone port.

7.1.5 TODO

Making SWITCHDEV and DSA converge towards an unified codebase

SWITCHDEV properly takes care of abstracting the networking stack with offload capable hardware, but does not enforce a strict switch device driver model. On the other DSA enforces a fairly strict device driver model, and deals with most of the switch specific. At some point we should envision a merger between these two subsystems and get the best of both worlds.

7.2 Broadcom RoboSwitch Ethernet switch driver

The Broadcom RoboSwitch Ethernet switch family is used in quite a range of xDSL router, cable modems and other multimedia devices.

The actual implementation supports the devices BCM5325E, BCM5365, BCM539x, BCM53115 and BCM53125 as well as BCM63XX.

7.2.1 Implementation details

The driver is located in `drivers/net/dsa/b53/` and is implemented as a DSA driver; see `Documentation/networking/dsa/dsa.rst` for details on the subsystem and what it provides.

The switch is, if possible, configured to enable a Broadcom specific 4-bytes switch tag which gets inserted by the switch for every packet forwarded to the CPU interface, conversely, the CPU network interface should insert a similar tag for packets entering the CPU port. The tag format is described in `net/dsa/tag_brcm.c`.

The configuration of the device depends on whether or not tagging is supported.

The interface names and example network configuration are used according the configuration described in the *Configuration showcases*.

Configuration with tagging support

The tagging based configuration is desired. It is not specific to the b53 DSA driver and will work like all DSA drivers which supports tagging.

See *Configuration with tagging support*.

Configuration without tagging support

Older models (5325, 5365) support a different tag format that is not supported yet. 539x and 531x5 require managed mode and some special handling, which is also not yet supported. The tagging support is disabled in these cases and the switch need a different configuration.

The configuration slightly differ from the *Configuration without tagging support*.

The b53 tags the CPU port in all VLANs, since otherwise any PVID untagged VLAN programming would basically change the CPU port's default PVID and make it untagged, undesirable.

In difference to the configuration described in *Configuration without tagging support* the default VLAN 1 has to be removed from the user interface configuration in single port and gateway

configuration, while there is no need to add an extra VLAN configuration in the bridge showcase.

single port

The configuration can only be set up via VLAN tagging and bridge setup. By default packages are tagged with vid 1:

```
# tag traffic on CPU port
ip link add link eth0 name eth0.1 type vlan id 1
ip link add link eth0 name eth0.2 type vlan id 2
ip link add link eth0 name eth0.3 type vlan id 3

# The conduit interface needs to be brought up before the user ports.
ip link set eth0 up
ip link set eth0.1 up
ip link set eth0.2 up
ip link set eth0.3 up

# bring up the user interfaces
ip link set wan up
ip link set lan1 up
ip link set lan2 up

# create bridge
ip link add name br0 type bridge

# activate VLAN filtering
ip link set dev br0 type bridge vlan_filtering 1

# add ports to bridges
ip link set dev wan master br0
ip link set dev lan1 master br0
ip link set dev lan2 master br0

# tag traffic on ports
bridge vlan add dev lan1 vid 2 pvid untagged
bridge vlan del dev lan1 vid 1
bridge vlan add dev lan2 vid 3 pvid untagged
bridge vlan del dev lan2 vid 1

# configure the VLANs
ip addr add 192.0.2.1/30 dev eth0.1
ip addr add 192.0.2.5/30 dev eth0.2
ip addr add 192.0.2.9/30 dev eth0.3

# bring up the bridge devices
ip link set br0 up
```

bridge

```

# tag traffic on CPU port
ip link add link eth0 name eth0.1 type vlan id 1

# The conduit interface needs to be brought up before the user ports.
ip link set eth0 up
ip link set eth0.1 up

# bring up the user interfaces
ip link set wan up
ip link set lan1 up
ip link set lan2 up

# create bridge
ip link add name br0 type bridge

# activate VLAN filtering
ip link set dev br0 type bridge vlan_filtering 1

# add ports to bridge
ip link set dev wan master br0
ip link set dev lan1 master br0
ip link set dev lan2 master br0
ip link set eth0.1 master br0

# configure the bridge
ip addr add 192.0.2.129/25 dev br0

# bring up the bridge
ip link set dev br0 up

```

gateway

```

# tag traffic on CPU port
ip link add link eth0 name eth0.1 type vlan id 1
ip link add link eth0 name eth0.2 type vlan id 2

# The conduit interface needs to be brought up before the user ports.
ip link set eth0 up
ip link set eth0.1 up
ip link set eth0.2 up

# bring up the user interfaces
ip link set wan up
ip link set lan1 up
ip link set lan2 up

# create bridge

```

```

ip link add name br0 type bridge

# activate VLAN filtering
ip link set dev br0 type bridge vlan_filtering 1

# add ports to bridges
ip link set dev wan master br0
ip link set eth0.1 master br0
ip link set dev lan1 master br0
ip link set dev lan2 master br0

# tag traffic on ports
bridge vlan add dev wan vid 2 pvid untagged
bridge vlan del dev wan vid 1

# configure the VLANs
ip addr add 192.0.2.1/30 dev eth0.2
ip addr add 192.0.2.129/25 dev br0

# bring up the bridge devices
ip link set br0 up

```

7.3 Broadcom Starfighter 2 Ethernet switch driver

Broadcom's Starfighter 2 Ethernet switch hardware block is commonly found and deployed in the following products:

- xDSL gateways such as BCM63138
- streaming/multimedia Set Top Box such as BCM7445
- Cable Modem/residential gateways such as BCM7145/BCM3390

The switch is typically deployed in a configuration involving between 5 to 13 ports, offering a range of built-in and customizable interfaces:

- single integrated Gigabit PHY
- quad integrated Gigabit PHY
- quad external Gigabit PHY w/ MDIO multiplexer
- integrated MoCA PHY
- several external MII/RevMII/GMII/RGMII interfaces

The switch also supports specific congestion control features which allow MoCA fail-over not to lose packets during a MoCA role re-election, as well as out of band back-pressure to the host CPU network interface when downstream interfaces are connected at a lower speed.

The switch hardware block is typically interfaced using MMIO accesses and contains a bunch of sub-blocks/register:

- SWITCH_CORE: common switch registers

- SWITCH_REG: external interfaces switch register
- SWITCH_MDIO: external MDIO bus controller (there is another one in SWITCH_CORE, which is used for indirect PHY accesses)
- SWITCH_INDIR_RW: 64-bits wide register helper block
- SWITCH_INTRL2_0/1: Level-2 interrupt controllers
- SWITCH_ACB: Admission control block
- SWITCH_FCB: Fail-over control block

7.3.1 Implementation details

The driver is located in `drivers/net/dsa/bcm_sf2.c` and is implemented as a DSA driver; see `Documentation/networking/dsa/dsa.rst` for details on the subsystem and what it provides.

The SF2 switch is configured to enable a Broadcom specific 4-bytes switch tag which gets inserted by the switch for every packet forwarded to the CPU interface, conversely, the CPU network interface should insert a similar tag for packets entering the CPU port. The tag format is described in `net/dsa/tag_brcm.c`.

Overall, the SF2 driver is a fairly regular DSA driver; there are a few specifics covered below.

Device Tree probing

The DSA platform device driver is probed using a specific compatible string provided in `net/dsa/dsa.c`. The reason for that is because the DSA subsystem gets registered as a platform device driver currently. DSA will provide the needed `device_node` pointers which are then accessible by the switch driver setup function to setup resources such as register ranges and interrupts. This currently works very well because none of the `of_*` functions utilized by the driver require a struct `device` to be bound to a struct `device_node`, but things may change in the future.

MDIO indirect accesses

Due to a limitation in how Broadcom switches have been designed, external Broadcom switches connected to a SF2 require the use of the DSA user MDIO bus in order to properly configure them. By default, the SF2 pseudo-PHY address, and an external switch pseudo-PHY address will both be snooping for incoming MDIO transactions, since they are at the same address (30), resulting in some kind of "double" programming. Using DSA, and setting `ds->phys_mii_mask` accordingly, we selectively divert reads and writes towards external Broadcom switches pseudo-PHY addresses. Newer revisions of the SF2 hardware have introduced a configurable pseudo-PHY address which circumvents the initial design limitation.

Multimedia over CoAxial (MoCA) interfaces

MoCA interfaces are fairly specific and require the use of a firmware blob which gets loaded onto the MoCA processor(s) for packet processing. The switch hardware contains logic which will assert/de-assert link states accordingly for the MoCA interface whenever the MoCA coaxial cable gets disconnected or the firmware gets reloaded. The SF2 driver relies on such events to properly set its MoCA interface carrier state and properly report this to the networking stack.

The MoCA interfaces are supported using the PHY library's fixed PHY/emulated PHY device and the switch driver registers a `fixed_link_update` callback for such PHYs which reflects the link state obtained from the interrupt handler.

Power Management

Whenever possible, the SF2 driver tries to minimize the overall switch power consumption by applying a combination of:

- turning off internal buffers/memories
- disabling packet processing logic
- putting integrated PHYs in IDDQ/low-power
- reducing the switch core clock based on the active port count
- enabling and advertising EEE
- turning off RGMII data processing logic when the link goes down

Wake-on-LAN

Wake-on-LAN is currently implemented by utilizing the host processor Ethernet MAC controller wake-on logic. Whenever Wake-on-LAN is requested, an intersection between the user request and the supported host Ethernet interface WoL capabilities is done and the intersection result gets configured. During system-wide suspend/resume, only ports not participating in Wake-on-LAN are disabled.

7.4 LAN9303 Ethernet switch driver

The LAN9303 is a three port 10/100 Mbps ethernet switch with integrated phys for the two external ethernet ports. The third port is an RMII/MII interface to a host conduit network interface (e.g. fixed link).

7.4.1 Driver details

The driver is implemented as a DSA driver, see [Documentation/networking/dsa/dsa.rst](#).

See [Documentation/devicetree/bindings/net/dsa/lan9303.txt](#) for device tree binding.

The LAN9303 can be managed both via MDIO and I2C, both supported by this driver.

At startup the driver configures the device to provide two separate network interfaces (which is the default state of a DSA device). Due to HW limitations, no HW MAC learning takes place in this mode.

When both user ports are joined to the same bridge, the normal HW MAC learning is enabled. This means that unicast traffic is forwarded in HW. Broadcast and multicast is flooded in HW. STP is also supported in this mode. The driver support fdb/mdb operations as well, meaning IGMP snooping is supported.

If one of the user ports leave the bridge, the ports goes back to the initial separated operation.

7.4.2 Driver limitations

- Support for VLAN filtering is not implemented
- The HW does not support VLAN-specific fdb entries

7.5 NXP SJA1105 switch driver

7.5.1 Overview

The NXP SJA1105 is a family of 10 SPI-managed automotive switches:

- SJA1105E: First generation, no TTEthernet
- SJA1105T: First generation, TTEthernet
- SJA1105P: Second generation, no TTEthernet, no SGMII
- SJA1105Q: Second generation, TTEthernet, no SGMII
- SJA1105R: Second generation, no TTEthernet, SGMII
- SJA1105S: Second generation, TTEthernet, SGMII
- SJA1110A: Third generation, TTEthernet, SGMII, integrated 100base-T1 and 100base-TX PHYs
- SJA1110B: Third generation, TTEthernet, SGMII, 100base-T1, 100base-TX
- SJA1110C: Third generation, TTEthernet, SGMII, 100base-T1, 100base-TX
- SJA1110D: Third generation, TTEthernet, SGMII, 100base-T1

Being automotive parts, their configuration interface is geared towards set-and-forget use, with minimal dynamic interaction at runtime. They require a static configuration to be composed by software and packed with CRC and table headers, and sent over SPI.

The static configuration is composed of several configuration tables. Each table takes a number of entries. Some configuration tables can be (partially) reconfigured at runtime, some not. Some tables are mandatory, some not:

Table	Mandatory	Reconfigurable
Schedule	no	no
Schedule entry points	if Scheduling	no
VL Lookup	no	no
VL Policing	if VL Lookup	no
VL Forwarding	if VL Lookup	no
L2 Lookup	no	no
L2 Policing	yes	no
VLAN Lookup	yes	yes
L2 Forwarding	yes	partially (fully on P/Q/R/S)
MAC Config	yes	partially (fully on P/Q/R/S)
Schedule Params	if Scheduling	no
Schedule Entry Points Params	if Scheduling	no
VL Forwarding Params	if VL Forwarding	no
L2 Lookup Params	no	partially (fully on P/Q/R/S)
L2 Forwarding Params	yes	no
Clock Sync Params	no	no
AVB Params	no	no
General Params	yes	partially
Retagging	no	yes
xMII Params	yes	no
SGMII	no	yes

Also the configuration is write-only (software cannot read it back from the switch except for very few exceptions).

The driver creates a static configuration at probe time, and keeps it at all times in memory, as a shadow for the hardware state. When required to change a hardware setting, the static configuration is also updated. If that changed setting can be transmitted to the switch through the dynamic reconfiguration interface, it is; otherwise the switch is reset and reprogrammed with the updated static configuration.

7.5.2 Switching features

The driver supports the configuration of L2 forwarding rules in hardware for port bridging. The forwarding, broadcast and flooding domain between ports can be restricted through two methods: either at the L2 forwarding level (isolate one bridge's ports from another's) or at the VLAN port membership level (isolate ports within the same bridge). The final forwarding decision taken by the hardware is a logical AND of these two sets of rules.

The hardware tags all traffic internally with a port-based VLAN (pvid), or it decodes the VLAN information from the 802.1Q tag. Advanced VLAN classification is not possible. Once attributed a VLAN tag, frames are checked against the port's membership rules and dropped at ingress if they don't match any VLAN. This behavior is available when switch ports join a bridge with `vlan_filtering 1`.

Normally the hardware is not configurable with respect to VLAN awareness, but by changing

what TPID the switch searches 802.1Q tags for, the semantics of a bridge with `vlan_filtering=0` can be kept (accept all traffic, tagged or untagged), and therefore this mode is also supported.

Segregating the switch ports in multiple bridges is supported (e.g. 2 + 2), but all bridges should have the same level of VLAN awareness (either both have `vlan_filtering=0`, or both 1).

Topology and loop detection through STP is supported.

7.5.3 Offloads

Time-aware scheduling

The switch supports a variation of the enhancements for scheduled traffic specified in IEEE 802.1Q-2018 (formerly 802.1Qbv). This means it can be used to ensure deterministic latency for priority traffic that is sent in-band with its gate-open event in the network schedule.

This capability can be managed through the tc-taprio offload ('flags 2'). The difference compared to the software implementation of taprio is that the latter would only be able to shape traffic originated from the CPU, but not autonomously forwarded flows.

The device has 8 traffic classes, and maps incoming frames to one of them based on the VLAN PCP bits (if no VLAN is present, the port-based default is used). As described in the previous sections, depending on the value of `vlan_filtering`, the EtherType recognized by the switch as being VLAN can either be the typical 0x8100 or a custom value used internally by the driver for tagging. Therefore, the switch ignores the VLAN PCP if used in standalone or bridge mode with `vlan_filtering=0`, as it will not recognize the 0x8100 EtherType. In these modes, injecting into a particular TX queue can only be done by the DSA net devices, which populate the PCP field of the tagging header on egress. Using `vlan_filtering=1`, the behavior is the other way around: offloaded flows can be steered to TX queues based on the VLAN PCP, but the DSA net devices are no longer able to do that. To inject frames into a hardware TX queue with VLAN awareness active, it is necessary to create a VLAN sub-interface on the DSA conduit port, and send normal (0x8100) VLAN-tagged towards the switch, with the VLAN PCP bits set appropriately.

Management traffic (having DMAC 01-80-C2-xx-xx-xx or 01-19-1B-xx-xx-xx) is the notable exception: the switch always treats it with a fixed priority and disregards any VLAN PCP bits even if present. The traffic class for management traffic has a value of 7 (highest priority) at the moment, which is not configurable in the driver.

Below is an example of configuring a 500 us cyclic schedule on egress port `swp5`. The traffic class gate for management traffic (7) is open for 100 us, and the gates for all other traffic classes are open for 400 us:

```
#!/bin/bash

set -e -u -o pipefail

NSEC_PER_SEC="1000000000"

gatemask() {
    local tc_list="$1"
    local mask=0
```

```

        for tc in ${tc_list}; do
                mask=$(( ${mask} | (1 << $tc) ))
        done

        printf "%02x" ${mask}
}

if ! systemctl is-active --quiet ptp4l; then
        echo "Please start the ptp4l service"
        exit
fi

now=$(phc_ctl /dev/ptp1 get | gawk '/clock time is/ { print $5; }')
# Phase-align the base time to the start of the next second.
sec=$(echo "${now}" | gawk -F. '{ print $1; }')
base_time="$((( ${sec} ) + 1) * ${NSEC_PER_SEC}))"

tc qdisc add dev swp5 parent root handle 100 taprio \
    num_tc 8 \
    map 0 1 2 3 5 6 7 \
    queues 1@0 1@1 1@2 1@3 1@4 1@5 1@6 1@7 \
    base-time ${base_time} \
    sched-entry S $(gatemask 7) 100000 \
    sched-entry S $(gatemask "0 1 2 3 4 5 6") 400000 \
    flags 2

```

It is possible to apply the tc-taprio offload on multiple egress ports. There are hardware restrictions related to the fact that no gate event may trigger simultaneously on two ports. The driver checks the consistency of the schedules against this restriction and errors out when appropriate. Schedule analysis is needed to avoid this, which is outside the scope of the document.

Routing actions (redirect, trap, drop)

The switch is able to offload flow-based redirection of packets to a set of destination ports specified by the user. Internally, this is implemented by making use of Virtual Links, a TTEthernet concept.

The driver supports 2 types of keys for Virtual Links:

- VLAN-aware virtual links: these match on destination MAC address, VLAN ID and VLAN PCP.
- VLAN-unaware virtual links: these match on destination MAC address only.

The VLAN awareness state of the bridge (vlan_filtering) cannot be changed while there are virtual link rules installed.

Composing multiple actions inside the same rule is supported. When only routing actions are requested, the driver creates a "non-critical" virtual link. When the action list also contains tc-gate (more details below), the virtual link becomes "time-critical" (draws frame buffers from a reserved memory partition, etc).

The 3 routing actions that are supported are "trap", "drop" and "redirect".

Example 1: send frames received on swp2 with a DA of 42:be:24:9b:76:20 to the CPU and to swp3. This type of key (DA only) when the port's VLAN awareness state is off:

```
tc qdisc add dev swp2 clsact
tc filter add dev swp2 ingress flower skip_sw dst_mac 42:be:24:9b:76:20 \
    action mirred egress redirect dev swp3 \
    action trap
```

Example 2: drop frames received on swp2 with a DA of 42:be:24:9b:76:20, a VID of 100 and a PCP of 0:

```
tc filter add dev swp2 ingress protocol 802.1Q flower skip_sw \
    dst_mac 42:be:24:9b:76:20 vlan_id 100 vlan_prio 0 action drop
```

Time-based ingress policing

The TTEthernet hardware abilities of the switch can be constrained to act similarly to the Per-Stream Filtering and Policing (PSFP) clause specified in IEEE 802.1Q-2018 (formerly 802.1Qci). This means it can be used to perform tight timing-based admission control for up to 1024 flows (identified by a tuple composed of destination MAC address, VLAN ID and VLAN PCP). Packets which are received outside their expected reception window are dropped.

This capability can be managed through the offload of the tc-gate action. As routing actions are intrinsic to virtual links in TTEthernet (which performs explicit routing of time-critical traffic and does not leave that in the hands of the FDB, flooding etc), the tc-gate action may never appear alone when asking sja1105 to offload it. One (or more) redirect or trap actions must also follow along.

Example: create a tc-taprio schedule that is phase-aligned with a tc-gate schedule (the clocks must be synchronized by a 1588 application stack, which is outside the scope of this document). No packet delivered by the sender will be dropped. Note that the reception window is larger than the transmission window (and much more so, in this example) to compensate for the packet propagation delay of the link (which can be determined by the 1588 application stack).

Receiver (sja1105):

```
tc qdisc add dev swp2 clsact
now=$(phc_ctl /dev/ptp1 get | awk '/clock time is/ {print $5}') && \
    sec=$(echo $now | awk -F. '{print $1}') && \
    base_time="$(((sec + 2) * 1000000000))" && \
    echo "base time ${base_time}"
tc filter add dev swp2 ingress flower skip_sw \
    dst_mac 42:be:24:9b:76:20 \
    action gate base-time ${base_time} \
    sched-entry OPEN 60000 -1 -1 \
    sched-entry CLOSE 40000 -1 -1 \
    action trap
```

Sender:

```
now=$(phc_ctl /dev/ptp0 get | awk '/clock time is/ {print $5}') && \
    sec=$(echo $now | awk -F. '{print $1}') && \
```

```

base_time="$(((sec + 2) * 1000000000))" && \
echo "base time ${base_time}"
tc qdisc add dev eno0 parent root taprio \
num_tc 8 \
map 0 1 2 3 4 5 6 7 \
queues 1@0 1@1 1@2 1@3 1@4 1@5 1@6 1@7 \
base-time ${base_time} \
sched-entry S 01 50000 \
sched-entry S 00 50000 \
flags 2

```

The engine used to schedule the ingress gate operations is the same that the one used for the tc-taprio offload. Therefore, the restrictions regarding the fact that no two gate actions (either tc-gate or tc-taprio gates) may fire at the same time (during the same 200 ns slot) still apply.

To come in handy, it is possible to share time-triggered virtual links across more than 1 ingress port, via flow blocks. In this case, the restriction of firing at the same time does not apply because there is a single schedule in the system, that of the shared virtual link:

```

tc qdisc add dev swp2 ingress_block 1 clsact
tc qdisc add dev swp3 ingress_block 1 clsact
tc filter add block 1 flower skip_sw dst_mac 42:be:24:9b:76:20 \
action gate index 2 \
base-time 0 \
sched-entry OPEN 50000000 -1 -1 \
sched-entry CLOSE 50000000 -1 -1 \
action trap

```

Hardware statistics for each flow are also available ("pkts" counts the number of dropped frames, which is a sum of frames dropped due to timing violations, lack of destination ports and MTU enforcement checks). Byte-level counters are not available.

7.5.4 Limitations

The SJA1105 switch family always performs VLAN processing. When configured as VLAN-unaware, frames carry a different VLAN tag internally, depending on whether the port is standalone or under a VLAN-unaware bridge.

The virtual link keys are always fixed at {MAC DA, VLAN ID, VLAN PCP}, but the driver asks for the VLAN ID and VLAN PCP when the port is under a VLAN-aware bridge. Otherwise, it fills in the VLAN ID and PCP automatically, based on whether the port is standalone or in a VLAN-unaware bridge, and accepts only "VLAN-unaware" tc-flower keys (MAC DA).

The existing tc-flower keys that are offloaded using virtual links are no longer operational after one of the following happens:

- port was standalone and joins a bridge (VLAN-aware or VLAN-unaware)
- port is part of a bridge whose VLAN awareness state changes
- port was part of a bridge and becomes standalone
- port was standalone, but another port joins a VLAN-aware bridge and this changes the global VLAN awareness state of the bridge

The driver cannot veto all these operations, and it cannot update/remove the existing tc-flower filters either. So for proper operation, the tc-flower filters should be installed only after the forwarding configuration of the port has been made, and removed by user space before making any changes to it.

7.5.5 Device Tree bindings and board design

This section references Documentation/devicetree/bindings/net/dsa/nxp,sja1105.yaml and aims to showcase some potential switch caveats.

RMII PHY role and out-of-band signaling

In the RMII spec, the 50 MHz clock signals are either driven by the MAC or by an external oscillator (but not by the PHY). But the spec is rather loose and devices go outside it in several ways. Some PHYs go against the spec and may provide an output pin where they source the 50 MHz clock themselves, in an attempt to be helpful. On the other hand, the SJA1105 is only binary configurable - when in the RMII MAC role it will also attempt to drive the clock signal. To prevent this from happening it must be put in RMII PHY role. But doing so has some unintended consequences. In the RMII spec, the PHY can transmit extra out-of-band signals via RXD[1:0]. These are practically some extra code words (/J/ and /K/) sent prior to the preamble of each frame. The MAC does not have this out-of-band signaling mechanism defined by the RMII spec. So when the SJA1105 port is put in PHY role to avoid having 2 drivers on the clock signal, inevitably an RMII PHY-to-PHY connection is created. The SJA1105 emulates a PHY interface fully and generates the /J/ and /K/ symbols prior to frame preambles, which the real PHY is not expected to understand. So the PHY simply encodes the extra symbols received from the SJA1105-as-PHY onto the 100Base-Tx wire. On the other side of the wire, some link partners might discard these extra symbols, while others might choke on them and discard the entire Ethernet frames that follow along. This looks like packet loss with some link partners but not with others. The take-away is that in RMII mode, the SJA1105 must be let to drive the reference clock if connected to a PHY.

RGMII fixed-link and internal delays

As mentioned in the bindings document, the second generation of devices has tunable delay lines as part of the MAC, which can be used to establish the correct RGMII timing budget. When powered up, these can shift the Rx and Tx clocks with a phase difference between 73.8 and 101.7 degrees. The catch is that the delay lines need to lock onto a clock signal with a stable frequency. This means that there must be at least 2 microseconds of silence between the clock at the old vs at the new frequency. Otherwise the lock is lost and the delay lines must be reset (powered down and back up). In RGMII the clock frequency changes with link speed (125 MHz at 1000 Mbps, 25 MHz at 100 Mbps and 2.5 MHz at 10 Mbps), and link speed might change during the AN process. In the situation where the switch port is connected through an RGMII fixed-link to a link partner whose link state life cycle is outside the control of Linux (such as a different SoC), then the delay lines would remain unlocked (and inactive) until there is manual intervention (ifdown/ifup on the switch port). The take-away is that in RGMII mode, the switch's internal delays are only reliable if the link partner never changes link speeds, or if it does, it does so in a way that is coordinated with the switch port (practically, both ends of the fixed-link are under control of the same Linux system). As to why would a fixed-link interface ever change link speeds: there are Ethernet controllers out there which come out of reset in

100 Mbps mode, and their driver inevitably needs to change the speed and clock frequency if it's required to work at gigabit.

MDIO bus and PHY management

The SJA1105 does not have an MDIO bus and does not perform in-band AN either. Therefore there is no link state notification coming from the switch device. A board would need to hook up the PHYs connected to the switch to any other MDIO bus available to Linux within the system (e.g. to the DSA conduit's MDIO bus). Link state management then works by the driver manually keeping in sync (over SPI commands) the MAC link speed with the settings negotiated by the PHY.

By comparison, the SJA1110 supports an MDIO slave access point over which its internal 100base-T1 PHYs can be accessed from the host. This is, however, not used by the driver, instead the internal 100base-T1 and 100base-TX PHYs are accessed through SPI commands, modeled in Linux as virtual MDIO buses.

The microcontroller attached to the SJA1110 port 0 also has an MDIO controller operating in master mode, however the driver does not support this either, since the microcontroller gets disabled when the Linux driver operates. Discrete PHYs connected to the switch ports should have their MDIO interface attached to an MDIO controller from the host system and not to the switch, similar to SJA1105.

Port compatibility matrix

The SJA1105 port compatibility matrix is:

Port	SJA1105E/T	SJA1105P/Q	SJA1105R/S
0	xMII	xMII	xMII
1	xMII	xMII	xMII
2	xMII	xMII	xMII
3	xMII	xMII	xMII
4	xMII	xMII	SGMII

The SJA1110 port compatibility matrix is:

Port	SJA1110A	SJA1110B	SJA1110C	SJA1110D
0	RevMII (uC)	RevMII (uC)	RevMII (uC)	RevMII (uC)
1	100base-TX or SGMII	100base-TX	100base-TX	SGMII
2	xMII or SGMII	xMII	xMII	xMII or SGMII
3	xMII or SGMII or 2500base-X	xMII or SGMII or 2500base-X	xMII	SGMII or 2500base-X
4	SGMII or 2500base-X	SGMII or 2500base-X	SGMII or 2500base-X	SGMII or 2500base-X
5	100base-T1	100base-T1	100base-T1	100base-T1
6	100base-T1	100base-T1	100base-T1	100base-T1
7	100base-T1	100base-T1	100base-T1	100base-T1
8	100base-T1	100base-T1	n/a	n/a
9	100base-T1	100base-T1	n/a	n/a
10	100base-T1	n/a	n/a	n/a

7.6 DSA switch configuration from userspace

The DSA switch configuration is not integrated into the main userspace network configuration suites by now and has to be performed manually.

7.6.1 Configuration showcases

To configure a DSA switch a couple of commands need to be executed. In this documentation some common configuration scenarios are handled as showcases:

single port

Every switch port acts as a different configurable Ethernet port

bridge

Every switch port is part of one configurable Ethernet bridge

gateway

Every switch port except one upstream port is part of a configurable Ethernet bridge. The upstream port acts as different configurable Ethernet port.

All configurations are performed with tools from iproute2, which is available at <https://www.kernel.org/pub/linux/utils/net/iproute2/>

Through DSA every port of a switch is handled like a normal linux Ethernet interface. The CPU port is the switch port connected to an Ethernet MAC chip. The corresponding linux Ethernet interface is called the conduit interface. All other corresponding linux interfaces are called user interfaces.

The user interfaces depend on the conduit interface being up in order for them to send or receive traffic. Prior to kernel v5.12, the state of the conduit interface had to be managed explicitly by the user. Starting with kernel v5.12, the behavior is as follows:

- when a DSA user interface is brought up, the conduit interface is automatically brought up.
- when the conduit interface is brought down, all DSA user interfaces are automatically brought down.

In this documentation the following Ethernet interfaces are used:

eth0

the conduit interface

eth1

another conduit interface

lan1

a user interface

lan2

another user interface

lan3

a third user interface

wan

A user interface dedicated for upstream traffic

Further Ethernet interfaces can be configured similar. The configured IPs and networks are:

single port

- lan1: 192.0.2.1/30 (192.0.2.0 - 192.0.2.3)
- lan2: 192.0.2.5/30 (192.0.2.4 - 192.0.2.7)
- lan3: 192.0.2.9/30 (192.0.2.8 - 192.0.2.11)

bridge

- br0: 192.0.2.129/25 (192.0.2.128 - 192.0.2.255)

gateway

- br0: 192.0.2.129/25 (192.0.2.128 - 192.0.2.255)
- wan: 192.0.2.1/30 (192.0.2.0 - 192.0.2.3)

7.6.2 Configuration with tagging support

The tagging based configuration is desired and supported by the majority of DSA switches. These switches are capable to tag incoming and outgoing traffic without using a VLAN based configuration.

single port

```
# configure each interface
ip addr add 192.0.2.1/30 dev lan1
ip addr add 192.0.2.5/30 dev lan2
ip addr add 192.0.2.9/30 dev lan3

# For kernels earlier than v5.12, the conduit interface needs to be
# brought up manually before the user ports.
ip link set eth0 up

# bring up the user interfaces
ip link set lan1 up
ip link set lan2 up
ip link set lan3 up
```

bridge

```
# For kernels earlier than v5.12, the conduit interface needs to be
# brought up manually before the user ports.
ip link set eth0 up

# bring up the user interfaces
ip link set lan1 up
ip link set lan2 up
ip link set lan3 up

# create bridge
ip link add name br0 type bridge
```

```
# add ports to bridge
ip link set dev lan1 master br0
ip link set dev lan2 master br0
ip link set dev lan3 master br0

# configure the bridge
ip addr add 192.0.2.129/25 dev br0

# bring up the bridge
ip link set dev br0 up
```

gateway

```
# For kernels earlier than v5.12, the conduit interface needs to be
# brought up manually before the user ports.
ip link set eth0 up

# bring up the user interfaces
ip link set wan up
ip link set lan1 up
ip link set lan2 up

# configure the upstream port
ip addr add 192.0.2.1/30 dev wan

# create bridge
ip link add name br0 type bridge

# add ports to bridge
ip link set dev lan1 master br0
ip link set dev lan2 master br0

# configure the bridge
ip addr add 192.0.2.129/25 dev br0

# bring up the bridge
ip link set dev br0 up
```

7.6.3 Configuration without tagging support

A minority of switches are not capable to use a tagging protocol (DSA_TAG_PROTO_NONE). These switches can be configured by a VLAN based configuration.

single port

The configuration can only be set up via VLAN tagging and bridge setup.

```
# tag traffic on CPU port
ip link add link eth0 name eth0.1 type vlan id 1
ip link add link eth0 name eth0.2 type vlan id 2
ip link add link eth0 name eth0.3 type vlan id 3
```

```

# For kernels earlier than v5.12, the conduit interface needs to be
# brought up manually before the user ports.
ip link set eth0 up
ip link set eth0.1 up
ip link set eth0.2 up
ip link set eth0.3 up

# bring up the user interfaces
ip link set lan1 up
ip link set lan2 up
ip link set lan3 up

# create bridge
ip link add name br0 type bridge

# activate VLAN filtering
ip link set dev br0 type bridge vlan_filtering 1

# add ports to bridges
ip link set dev lan1 master br0
ip link set dev lan2 master br0
ip link set dev lan3 master br0

# tag traffic on ports
bridge vlan add dev lan1 vid 1 pvid untagged
bridge vlan add dev lan2 vid 2 pvid untagged
bridge vlan add dev lan3 vid 3 pvid untagged

# configure the VLANs
ip addr add 192.0.2.1/30 dev eth0.1
ip addr add 192.0.2.5/30 dev eth0.2
ip addr add 192.0.2.9/30 dev eth0.3

# bring up the bridge devices
ip link set br0 up

```

bridge

```

# tag traffic on CPU port
ip link add link eth0 name eth0.1 type vlan id 1

# For kernels earlier than v5.12, the conduit interface needs to be
# brought up manually before the user ports.
ip link set eth0 up
ip link set eth0.1 up

# bring up the user interfaces
ip link set lan1 up
ip link set lan2 up
ip link set lan3 up

```

```

# create bridge
ip link add name br0 type bridge

# activate VLAN filtering
ip link set dev br0 type bridge vlan_filtering 1

# add ports to bridge
ip link set dev lan1 master br0
ip link set dev lan2 master br0
ip link set dev lan3 master br0
ip link set eth0.1 master br0

# tag traffic on ports
bridge vlan add dev lan1 vid 1 pvid untagged
bridge vlan add dev lan2 vid 1 pvid untagged
bridge vlan add dev lan3 vid 1 pvid untagged

# configure the bridge
ip addr add 192.0.2.129/25 dev br0

# bring up the bridge
ip link set dev br0 up

```

gateway

```

# tag traffic on CPU port
ip link add link eth0 name eth0.1 type vlan id 1
ip link add link eth0 name eth0.2 type vlan id 2

# For kernels earlier than v5.12, the conduit interface needs to be
# brought up manually before the user ports.
ip link set eth0 up
ip link set eth0.1 up
ip link set eth0.2 up

# bring up the user interfaces
ip link set wan up
ip link set lan1 up
ip link set lan2 up

# create bridge
ip link add name br0 type bridge

# activate VLAN filtering
ip link set dev br0 type bridge vlan_filtering 1

# add ports to bridges
ip link set dev wan master br0
ip link set dev eth0.1 master br0
ip link set dev lan1 master br0

```

```

ip link set dev lan2 master br0

# tag traffic on ports
bridge vlan add dev lan1 vid 1 pvid untagged
bridge vlan add dev lan2 vid 1 pvid untagged
bridge vlan add dev wan vid 2 pvid untagged

# configure the VLANs
ip addr add 192.0.2.1/30 dev eth0.2
ip addr add 192.0.2.129/25 dev br0

# bring up the bridge devices
ip link set br0 up

```

7.6.4 Forwarding database (FDB) management

The existing DSA switches do not have the necessary hardware support to keep the software FDB of the bridge in sync with the hardware tables, so the two tables are managed separately (`bridge fdb show` queries both, and depending on whether the `self` or `master` flags are being used, a `bridge fdb add` or `bridge fdb del` command acts upon entries from one or both tables).

Up until kernel v4.14, DSA only supported user space management of bridge FDB entries using the bridge bypass operations (which do not update the software FDB, just the hardware one) using the `self` flag (which is optional and can be omitted).

```

bridge fdb add dev swp0 00:01:02:03:04:05 self static
# or shorthand
bridge fdb add dev swp0 00:01:02:03:04:05 static

```

Due to a bug, the bridge bypass FDB implementation provided by DSA did not distinguish between `static` and `local` FDB entries (`static` are meant to be forwarded, while `local` are meant to be locally terminated, i.e. sent to the host port). Instead, all FDB entries with the `self` flag (implicit or explicit) are treated by DSA as `static` even if they are `local`.

```

# This command:
bridge fdb add dev swp0 00:01:02:03:04:05 static
# behaves the same for DSA as this command:
bridge fdb add dev swp0 00:01:02:03:04:05 local
# or shorthand, because the 'local' flag is implicit if 'static' is not
# specified, it also behaves the same as:
bridge fdb add dev swp0 00:01:02:03:04:05

```

The last command is an incorrect way of adding a static bridge FDB entry to a DSA switch using the bridge bypass operations, and works by mistake. Other drivers will treat an FDB entry added by the same command as `local` and as such, will not forward it, as opposed to DSA.

Between kernel v4.14 and v5.14, DSA has supported in parallel two modes of adding a bridge FDB entry to the switch: the bridge bypass discussed above, as well as a new mode using the `master` flag which installs FDB entries in the software bridge too.

```
bridge fdb add dev swp0 00:01:02:03:04:05 master static
```

Since kernel v5.14, DSA has gained stronger integration with the bridge's software FDB, and the support for its bridge bypass FDB implementation (using the `self` flag) has been removed. This results in the following changes:

```
# This is the only valid way of adding an FDB entry that is supported,
# compatible with v4.14 kernels and later:
bridge fdb add dev swp0 00:01:02:03:04:05 master static
# This command is no longer buggy and the entry is properly treated as
# 'local' instead of being forwarded:
bridge fdb add dev swp0 00:01:02:03:04:05
# This command no longer installs a static FDB entry to hardware:
bridge fdb add dev swp0 00:01:02:03:04:05 static
```

Script writers are therefore encouraged to use the `master static` set of flags when working with bridge FDB entries on DSA switch interfaces.

7.6.5 Affinity of user ports to CPU ports

Typically, DSA switches are attached to the host via a single Ethernet interface, but in cases where the switch chip is discrete, the hardware design may permit the use of 2 or more ports connected to the host, for an increase in termination throughput.

DSA can make use of multiple CPU ports in two ways. First, it is possible to statically assign the termination traffic associated with a certain user port to be processed by a certain CPU port. This way, user space can implement custom policies of static load balancing between user ports, by spreading the affinities according to the available CPU ports.

Secondly, it is possible to perform load balancing between CPU ports on a per packet basis, rather than statically assigning user ports to CPU ports. This can be achieved by placing the DSA conduits under a LAG interface (bonding or team). DSA monitors this operation and creates a mirror of this software LAG on the CPU ports facing the physical DSA conduits that constitute the LAG slave devices.

To make use of multiple CPU ports, the firmware (device tree) description of the switch must mark all the links between CPU ports and their DSA conduits using the `ether` reference/phandle. At startup, only a single CPU port and DSA conduit will be used - the numerically first port from the firmware description which has an `ether` property. It is up to the user to configure the system for the switch to use other conduits.

DSA uses the `rtwl_link_ops` mechanism (with a "dsa" kind) to allow changing the DSA conduit of a user port. The `IFLA_DSA_CONDUIT` u32 netlink attribute contains the ifindex of the conduit device that handles each user device. The DSA conduit must be a valid candidate based on firmware node information, or a LAG interface which contains only slaves which are valid candidates.

Using iproute2, the following manipulations are possible:

```
# See the DSA conduit in current use
ip -d link show dev swp0
(...)
dsa master eth0
```

```

# Static CPU port distribution
ip link set swp0 type dsa master eth1
ip link set swp1 type dsa master eth0
ip link set swp2 type dsa master eth1
ip link set swp3 type dsa master eth0

# CPU ports in LAG, using explicit assignment of the DSA conduit
ip link add bond0 type bond mode balance-xor && ip link set bond0 up
ip link set eth1 down && ip link set eth1 master bond0
ip link set swp0 type dsa master bond0
ip link set swp1 type dsa master bond0
ip link set swp2 type dsa master bond0
ip link set swp3 type dsa master bond0
ip link set eth0 down && ip link set eth0 master bond0
ip -d link show dev swp0
(...)
    dsa master bond0

# CPU ports in LAG, relying on implicit migration of the DSA conduit
ip link add bond0 type bond mode balance-xor && ip link set bond0 up
ip link set eth0 down && ip link set eth0 master bond0
ip link set eth1 down && ip link set eth1 master bond0
ip -d link show dev swp0
(...
    dsa master bond0

```

Notice that in the case of CPU ports under a LAG, the use of the `IFLA_DSA_CONDUIT` netlink attribute is not strictly needed, but rather, DSA reacts to the `IFLA_MASTER` attribute change of its present conduit (`eth0`) and migrates all user ports to the new upper of `eth0`, `bond0`. Similarly, when `bond0` is destroyed using `RTM_DELLINK`, DSA migrates the user ports that were assigned to this interface to the first physical DSA conduit which is eligible, based on the firmware description (it effectively reverts to the startup configuration).

In a setup with more than 2 physical CPU ports, it is therefore possible to mix static user to CPU port assignment with LAG between DSA conduits. It is not possible to statically assign a user port towards a DSA conduit that has any upper interfaces (this includes LAG devices - the conduit must always be the LAG in this case).

Live changing of the DSA conduit (and thus CPU port) affinity of a user port is permitted, in order to allow dynamic redistribution in response to traffic.

Physical DSA conduits are allowed to join and leave at any time a LAG interface used as a DSA conduit; however, DSA will reject a LAG interface as a valid candidate for being a DSA conduit unless it has at least one physical DSA conduit as a slave device.

LINUX DEVLINK DOCUMENTATION

devlink is an API to expose device information and resources not directly related to any device class, such as chip-wide/switch-ASIC-wide configuration.

8.1 Locking

Driver facing APIs are currently transitioning to allow more explicit locking. Drivers can use the existing `devlink_*` set of APIs, or new APIs prefixed by `devl_*`. The older APIs handle all the locking in devlink core, but don't allow registration of most sub-objects once the main devlink object is itself registered. The newer `devl_*` APIs assume the devlink instance lock is already held. Drivers can take the instance lock by calling `devl_lock()`. It is also held all callbacks of devlink netlink commands.

Drivers are encouraged to use the devlink instance lock for their own needs.

Drivers need to be cautious when taking devlink instance lock and taking RTNL lock at the same time. Devlink instance lock needs to be taken first, only after that RTNL lock could be taken.

8.2 Nested instances

Some objects, like linecards or port functions, could have another devlink instances created underneath. In that case, drivers should make sure to respect following rules:

- Lock ordering should be maintained. If driver needs to take instance lock of both nested and parent instances at the same time, devlink instance lock of the parent instance should be taken first, only then instance lock of the nested instance could be taken.
- Driver should use object-specific helpers to setup the nested relationship:
 - `devl_nested_devlink_set()` - called to setup devlink -> nested devlink relationship (could be user for multiple nested instances).
 - `devl_port_fn_devlink_set()` - called to setup port function -> nested devlink relationship.
 - `devlink_linecard_nested_dl_set()` - called to setup linecard -> nested devlink relationship.

The nested devlink info is exposed to the userspace over object-specific attributes of devlink netlink.

8.3 Interface documentation

The following pages describe various interfaces available through devlink in general.

8.3.1 Devlink DPipe

Background

While performing the hardware offloading process, much of the hardware specifics cannot be presented. These details are useful for debugging, and devlink-dpipe provides a standardized way to provide visibility into the offloading process.

For example, the routing longest prefix match (LPM) algorithm used by the Linux kernel may differ from the hardware implementation. The pipeline debug API (DPipe) is aimed at providing the user visibility into the ASIC's pipeline in a generic way.

The hardware offload process is expected to be done in a way that the user should not be able to distinguish between the hardware vs. software implementation. In this process, hardware specifics are neglected. In reality those details can have lots of meaning and should be exposed in some standard way.

This problem is made even more complex when one wishes to offload the control path of the whole networking stack to a switch ASIC. Due to differences in the hardware and software models some processes cannot be represented correctly.

One example is the kernel's LPM algorithm which in many cases differs greatly to the hardware implementation. The configuration API is the same, but one cannot rely on the Forward Information Base (FIB) to look like the Level Path Compression trie (LPC-trie) in hardware.

In many situations trying to analyze systems failure solely based on the kernel's dump may not be enough. By combining this data with complementary information about the underlying hardware, this debugging can be made easier; additionally, the information can be useful when debugging performance issues.

Overview

The devlink-dpipe interface closes this gap. The hardware's pipeline is modeled as a graph of match/action tables. Each table represents a specific hardware block. This model is not new, first being used by the P4 language.

Traditionally it has been used as an alternative model for hardware configuration, but the devlink-dpipe interface uses it for visibility purposes as a standard complementary tool. The system's view from devlink-dpipe should change according to the changes done by the standard configuration tools.

For example, it's quite common to implement Access Control Lists (ACL) using Ternary Content Addressable Memory (TCAM). The TCAM memory can be divided into TCAM regions. Complex TC filters can have multiple rules with different priorities and different lookup keys. On the other hand hardware TCAM regions have a predefined lookup key. Offloading the TC filter rules using TCAM engine can result in multiple TCAM regions being interconnected in a chain (which may affect the data path latency). In response to a new TC filter new tables should be created describing those regions.

Model

The DPIPE model introduces several objects:

- headers
- tables
- entries

A header describes packet formats and provides names for fields within the packet. A table describes hardware blocks. An entry describes the actual content of a specific table.

The hardware pipeline is not port specific, but rather describes the whole ASIC. Thus it is tied to the top of the devlink infrastructure.

Drivers can register and unregister tables at run time, in order to support dynamic behavior. This dynamic behavior is mandatory for describing hardware blocks like TCAM regions which can be allocated and freed dynamically.

`devlink-dpipe` generally is not intended for configuration. The exception is hardware counting for a specific table.

The following commands are used to obtain the `dpipe` objects from userspace:

- `table_get`: Receive a table's description.
- `headers_get`: Receive a device's supported headers.
- `entries_get`: Receive a table's current entries.
- `counters_set`: Enable or disable counters on a table.

Table

The driver should implement the following operations for each table:

- `matches_dump`: Dump the supported matches.
- `actions_dump`: Dump the supported actions.
- `entries_dump`: Dump the actual content of the table.
- `counters_set_update`: Synchronize hardware with counters enabled or disabled.

Header/Field

In a similar way to P4 headers and fields are used to describe a table's behavior. There is a slight difference between the standard protocol headers and specific ASIC metadata. The protocol headers should be declared in the devlink core API. On the other hand ASIC meta data is driver specific and should be defined in the driver. Additionally, each driver-specific devlink documentation file should document the driver-specific `dpipe` headers it implements. The headers and fields are identified by enumeration.

In order to provide further visibility some ASIC metadata fields could be mapped to kernel objects. For example, internal router interface indexes can be directly mapped to the net device ifindex. FIB table indexes used by different Virtual Routing and Forwarding (VRF) tables can be mapped to internal routing table indexes.

Match

Matches are kept primitive and close to hardware operation. Match types like LPM are not supported due to the fact that this is exactly a process we wish to describe in full detail. Example of matches:

- `field_exact`: Exact match on a specific field.
- `field_exact_mask`: Exact match on a specific field after masking.
- `field_range`: Match on a specific range.

The id's of the header and the field should be specified in order to identify the specific field. Furthermore, the header index should be specified in order to distinguish multiple headers of the same type in a packet (tunneling).

Action

Similar to match, the actions are kept primitive and close to hardware operation. For example:

- `field_modify`: Modify the field value.
- `field_inc`: Increment the field value.
- `push_header`: Add a header.
- `pop_header`: Remove a header.

Entry

Entries of a specific table can be dumped on demand. Each entry is identified with an index and its properties are described by a list of match/action values and specific counter. By dumping the tables content the interactions between tables can be resolved.

Abstraction Example

The following is an example of the abstraction model of the L3 part of Mellanox Spectrum ASIC. The blocks are described in the order they appear in the pipeline. The table sizes in the following examples are not real hardware sizes and are provided for demonstration purposes.

LPM

The LPM algorithm can be implemented as a list of hash tables. Each hash table contains routes with the same prefix length. The root of the list is /32, and in case of a miss the hardware will continue to the next hash table. The depth of the search will affect the data path latency.

In case of a hit the entry contains information about the next stage of the pipeline which resolves the MAC address. The next stage can be either local host table for directly connected routes, or adjacency table for next-hops. The `meta.lpm_prefix` field is used to connect two LPM tables.

```
table lpm_prefix_16 {
    size: 4096,
    counters_enabled: true,
    match: { meta.vr_id: exact,
              ipv4.dst_addr: exact_mask,
              ipv6.dst_addr: exact_mask,
              meta.lpm_prefix: exact },
    action: { meta.adj_index: set,
              meta.adj_group_size: set,
              meta.rif_port: set,
              meta.lpm_prefix: set },
}
```

Local Host

In the case of local routes the LPM lookup already resolves the egress router interface (RIF), yet the exact MAC address is not known. The local host table is a hash table combining the output interface id with destination IP address as a key. The result is the MAC address.

```
table local_host {
    size: 4096,
    counters_enabled: true,
    match: { meta.rif_port: exact,
              ipv4.dst_addr: exact },
    action: { ethernet.daddr: set }
}
```

Adjacency

In case of remote routes this table does the ECMP. The LPM lookup results in ECMP group size and index that serves as a global offset into this table. Concurrently a hash of the packet is generated. Based on the ECMP group size and the packet's hash a local offset is generated. Multiple LPM entries can point to the same adjacency group.

```
table adjacency {
    size: 4096,
    counters_enabled: true,
    match: { meta.adj_index: exact,
              meta.adj_group_size: exact,
              meta.packet_hash_index: exact },
    action: { ethernet.daddr: set,
              meta.erif: set }
}
```

ERIF

In case the egress RIF and destination MAC have been resolved by previous tables this table does multiple operations like TTL decrease and MTU check. Then the decision of forward/drop is taken and the port L3 statistics are updated based on the packet's type (broadcast, unicast, multicast).

```
table erif {
    size: 800,
    counters_enabled: true,
    match: { meta.rif_port: exact,
              meta.is_l3_unicast: exact,
              meta.is_l3_broadcast: exact,
              meta.is_l3_multicast, exact },
    action: { meta.l3_drop: set,
              meta.l3_forward: set }
}
```

8.3.2 Devlink Health

Background

The devlink health mechanism is targeted for Real Time Alerting, in order to know when something bad happened to a PCI device.

- Provide alert debug information.
- Self healing.
- If problem needs vendor support, provide a way to gather all needed debugging information.

Overview

The main idea is to unify and centralize driver health reports in the generic devlink instance and allow the user to set different attributes of the health reporting and recovery procedures.

The devlink health reporter: Device driver creates a "health reporter" per each error/health type. Error/Health type can be a known/generic (e.g. PCI error, fw error, rx/tx error) or unknown (driver specific). For each registered health reporter a driver can issue error/health reports asynchronously. All health reports handling is done by devlink. Device driver can provide specific callbacks for each "health reporter", e.g.:

- Recovery procedures
- Diagnostics procedures
- Object dump procedures
- Out Of Box initial parameters

Different parts of the driver can register different types of health reporters with different handlers.

Actions

Once an error is reported, devlink health will perform the following actions:

- A log is being send to the kernel trace events buffer
- Health status and statistics are being updated for the reporter instance
- Object dump is being taken and saved at the reporter instance (as long as auto-dump is set and there is no other dump which is already stored)
- Auto recovery attempt is being done. Depends on:
 - Auto-recovery configuration
 - Grace period vs. time passed since last recover

Devlink formatted message

To handle devlink health diagnose and health dump requests, devlink creates a formatted message structure `devlink_fmsg` and send it to the driver's callback to fill the data in using the devlink fmsg API.

Devlink fmsg is a mechanism to pass descriptors between drivers and devlink, in json-like format. The API allows the driver to add nested attributes such as object, object pair and value array, in addition to attributes such as name and value.

Driver should use this API to fill the fmsg context in a format which will be translated by the devlink to the netlink message later. When it needs to send the data using SKBs to the netlink layer, it fragments the data between different SKBs. In order to do this fragmentation, it uses virtual nests attributes, to avoid actual nesting use which cannot be divided between different SKBs.

User Interface

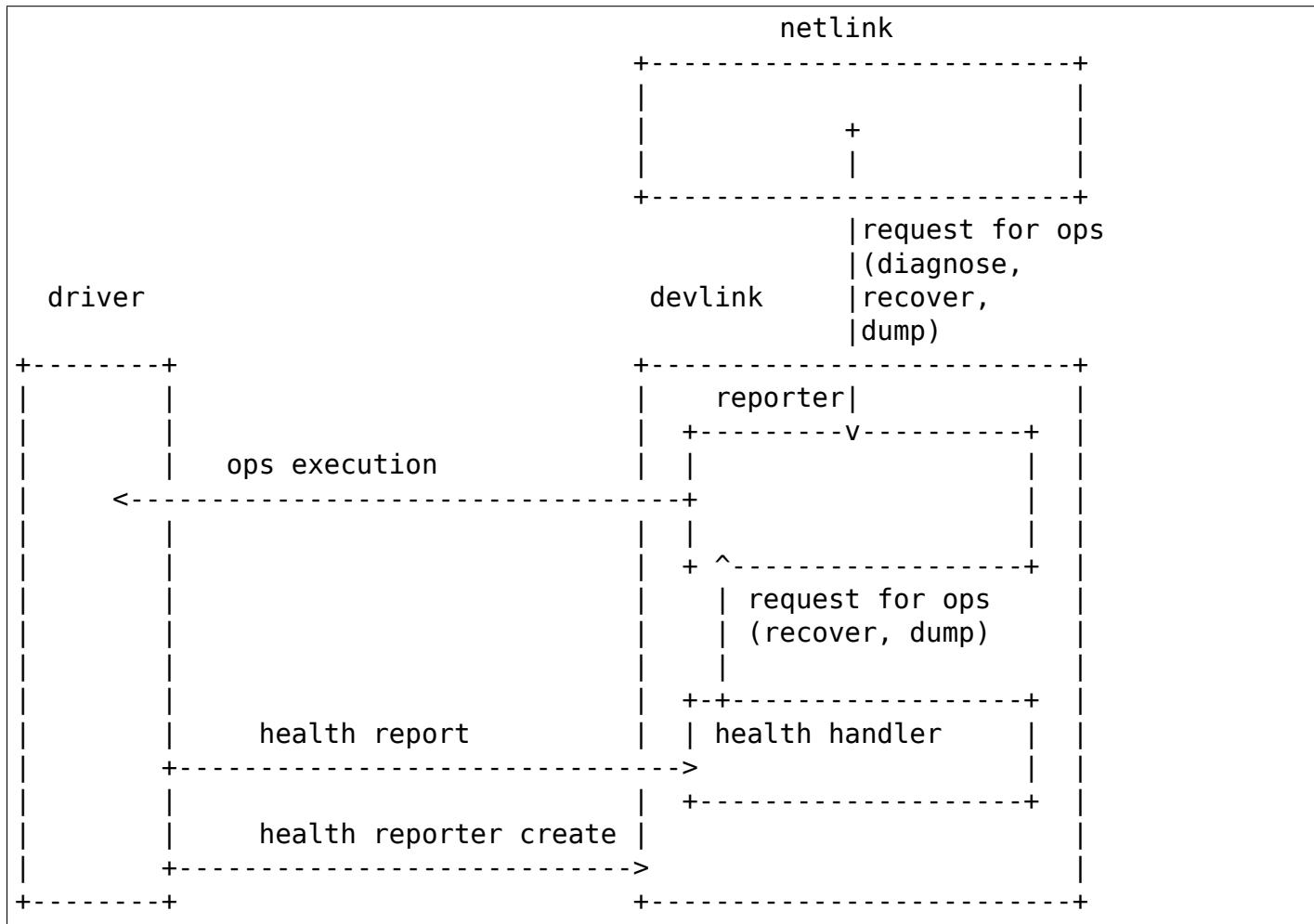
User can access/change each reporter's parameters and driver specific callbacks via devlink, e.g per error type (per health reporter):

- Configure reporter's generic parameters (like: disable/enable auto recovery)
- Invoke recovery procedure
- Run diagnostics
- Object dump

Table 1: List of devlink health interfaces

Name	Description
DEVLINK_RMDRIVENLTHSHAREPORTERDGET	Configuration info per DEV and reporter.
DEVLINK_AMDWLTHAPTRREPORTERDSET	Configuration setting.
DEVLINK_TMDLTHREPORTERDRECOVER	Recovery procedure.
DEVLINK_TMDLTHREPORTERDTEST	Test on the reporter. The effects of the test event in terms of recovery flow should follow closely that of a real event.
DEVLINK_RMDRIVENLTHREPORTERDIAGNOSE	Related to the reporter.
DEVLINK_RMDRIVENLTHREPORTERDUMPGET	Devlink health saves a single dump. If no dump is already stored by devlink for this reporter, devlink generates a new dump. Dump output is defined by the reporter.
DEVLINK_CMDLTHREPORTERDUMFILECLR	Clear health reporter dump file for the specified reporter.

The following diagram provides a general overview of devlink-health:



8.3.3 Devlink Info

The devlink-info mechanism enables device drivers to report device (hardware and firmware) information in a standard, extensible fashion.

The original motivation for the devlink-info API was twofold:

- making it possible to automate device and firmware management in a fleet of machines in a vendor-independent fashion (see also [Documentation/networking/devlink/devlink-flash.rst](#));
- name the per component FW versions (as opposed to the crowded ethtool version string).

devlink-info supports reporting multiple types of objects. Reporting driver versions is generally discouraged - here, and via any other Linux API.

Table 2: List of top level info objects

Name	Description
driver	Name of the currently used device driver, also available through sysfs.
serial	Serial member of the device. This is usually the serial number of the ASIC, also often available in PCI config space of the device in the <i>Device Serial Number</i> capability. The serial number should be unique per physical device. Sometimes the serial number of the device is only 48 bits long (the length of the Ethernet MAC address), and since PCI DSN is 64 bits long devices pad or encode additional information into the serial number. One example is adding port ID or PCI interface ID in the extra two bytes. Drivers should make sure to strip or normalize any such padding or interface ID, and report only the part of the serial number which uniquely identifies the hardware. In other words serial number reported for two ports of the same device or on two hosts of a multi-host device should be identical.
board	Board serial number of the device.
series	This is usually the serial number of the board, often available in PCI <i>Vital Product Data</i> .
fixed	Group for hardware identifiers, and versions of components which are not field-updatable. Versions in this section identify the device design. For example, component identifiers or the board version reported in the PCI VPD. Data in devlink-info should be broken into the smallest logical components, e.g. PCI VPD may concatenate various information to form the Part Number string, while in devlink-info all parts should be reported as separate items. This group must not contain any frequently changing identifiers, such as serial numbers. See Documentation/networking/devlink/devlink-flash.rst to understand why.
running	Group for information about currently running software/firmware. These versions often only update after a reboot, sometimes device reset.
stored	Group for software/firmware versions in device flash. Stored values must update to reflect changes in the flash even if reboot has not yet occurred. If device is not capable of updating stored versions when new software is flashed, it must not report them.

Each version can be reported at most once in each version group. Firmware components stored on the flash should feature in both the running and stored sections, if device is capable of reporting stored versions (see [Documentation/networking/devlink/devlink-flash.rst](#)). In case software/firmware components are loaded from the disk (e.g. /lib/firmware) only the running

version should be reported via the kernel API.

Generic Versions

It is expected that drivers use the following generic names for exporting version information. If a generic name for a given component doesn't exist yet, driver authors should consult existing driver-specific versions and attempt reuse. As last resort, if a component is truly unique, using driver-specific names is allowed, but these should be documented in the driver-specific file.

All versions should try to use the following terminology:

Table 3: List of common version suffixes

Name	Description
id, revision	Identifiers of designs and revision, mostly used for hardware versions.
api	Version of API between components. API items are usually of limited value to the user, and can be inferred from other versions by the vendor, so adding API versions is generally discouraged as noise.
bundle	Identifier of a distribution package which was flashed onto the device. This is an attribute of a firmware package which covers multiple versions for ease of managing firmware images (see Documentation/networking/devlink/devlink-flash.rst). bundle_id can appear in both running and stored versions, but it must not be reported if any of the components covered by the bundle_id was changed and no longer matches the version from the bundle.

board.id

Unique identifier of the board design.

board.rev

Board design revision.

asic.id

ASIC design identifier.

asic.rev

ASIC design revision/stepping.

board.manufacture

An identifier of the company or the facility which produced the part.

fw

Overall firmware version, often representing the collection of fw_mgmt, fw_app, etc.

fw_mgmt

Control unit firmware version. This firmware is responsible for house keeping tasks, PHY control etc. but not the packet-by-packet data path operation.

fw_mgmt_api

Firmware interface specification version of the software interfaces between driver and firmware.

fw_app

Data path microcode controlling high-speed packet processing.

fw_undi

UNDI software, may include the UEFI driver, firmware or both.

fw_ncsi

Version of the software responsible for supporting/handling the Network Controller Sideband Interface.

fw_psid

Unique identifier of the firmware parameter set. These are usually parameters of a particular board, defined at manufacturing time.

fw.roce

RoCE firmware version which is responsible for handling roce management.

fw.bundle_id

Unique identifier of the entire firmware bundle.

fw.bootloader

Version of the bootloader.

Future work

The following extensions could be useful:

- on-disk firmware file names - drivers list the file names of firmware they may need to load onto devices via the `MODULE_FIRMWARE()` macro. These, however, are per module, rather than per device. It'd be useful to list the names of firmware files the driver will try to load for a given device, in order of priority.

8.3.4 Devlink Flash

The devlink-flash API allows updating device firmware. It replaces the older ethtool-flash mechanism, and doesn't require taking any networking locks in the kernel to perform the flash update. Example use:

```
$ devlink dev flash pci/0000:05:00.0 file flash-boot.bin
```

Note that the file name is a path relative to the firmware loading path (usually `/lib/firmware/`). Drivers may send status updates to inform user space about the progress of the update operation.

Overwrite Mask

The devlink-flash command allows optionally specifying a mask indicating how the device should handle subsections of flash components when updating. This mask indicates the set of sections which are allowed to be overwritten.

Table 4: List of overwrite mask bits

Name	Description
<code>DEVLINK_FLASH_HOVERWRITE_SETTINGS</code>	overwrite settings in the components being updated with the settings found in the provided image.
<code>DEVLINK_FLASH_HOVERWRITE_IDENTIFIERS</code>	overwrite identifiers in the components being updated with the identifiers found in the provided image. This includes MAC addresses, serial IDs, and similar device identifiers.

Multiple overwrite bits may be combined and requested together. If no bits are provided, it is expected that the device only update firmware binaries in the components being updated. Settings and identifiers are expected to be preserved across the update. A device may not support every combination and the driver for such a device must reject any combination which cannot be faithfully implemented.

Firmware Loading

Devices which require firmware to operate usually store it in non-volatile memory on the board, e.g. flash. Some devices store only basic firmware on the board, and the driver loads the rest from disk during probing. devlink-info allows users to query firmware information (loaded components and versions).

In other cases the device can both store the image on the board, load from disk, or automatically flash a new image from disk. The fw_load_policy devlink parameter can be used to control this behavior ([Documentation/networking/devlink/devlink-params.rst](#)).

On-disk firmware files are usually stored in /lib/firmware/.

Firmware Version Management

Drivers are expected to implement devlink-flash and devlink-info functionality, which together allow for implementing vendor-independent automated firmware update facilities.

devlink-info exposes the driver name and three version groups (fixed, running, stored).

The driver attribute and fixed group identify the specific device design, e.g. for looking up applicable firmware updates. This is why serial_number is not part of the fixed versions (even though it is fixed) - fixed versions should identify the design, not a single device.

running and stored firmware versions identify the firmware running on the device, and firmware which will be activated after reboot or device reset.

The firmware update agent is supposed to be able to follow this simple algorithm to update firmware contents, regardless of the device vendor:

```
# Get unique HW design identifier
$hw_id = devlink-dev-info['fixed']

# Find out which FW flash we want to use for this NIC
$want_flash_vers = some-db-backed.lookup($hw_id, 'flash')

# Update flash if necessary
if $want_flash_vers != devlink-dev-info['stored']:
    $file = some-db-backed.download($hw_id, 'flash')
    devlink-dev-flash($file)

# Find out the expected overall firmware versions
$want_fw_vers = some-db-backed.lookup($hw_id, 'all')

# Update on-disk file if necessary
if $want_fw_vers != devlink-dev-info['running']:
    $file = some-db-backed.download($hw_id, 'disk')
```

```

write($file, '/lib/firmware/')

# Try device reset, if available
if $want_fw_ver != devlink-dev-info['running']:
    devlink-reset()

# Reboot, if reset wasn't enough
if $want_fw_ver != devlink-dev-info['running']:
    reboot()

```

Note that each reference to devlink-dev-info in this pseudo-code is expected to fetch up-to-date information from the kernel.

For the convenience of identifying firmware files some vendors add bundle_id information to the firmware versions. This meta-version covers multiple per-component versions and can be used e.g. in firmware file names (all component versions could get rather long.)

8.3.5 Devlink Params

devlink provides capability for a driver to expose device parameters for low level device functionality. Since devlink can operate at the device-wide level, it can be used to provide configuration that may affect multiple ports on a single device.

This document describes a number of generic parameters that are supported across multiple drivers. Each driver is also free to add their own parameters. Each driver must document the specific parameters they support, whether generic or not.

Configuration modes

Parameters may be set in different configuration modes.

Table 5: Possible configuration modes

Name	Description
run	set while the driver is running, and takes effect immediately. No reset is required.
drive	applied while the driver initializes. Requires the user to restart the driver using the devlink reload command.
permanent	written to the device's non-volatile memory. A hard reset is required for it to take effect.

Reloading

In order for driverinit parameters to take effect, the driver must support reloading via the devlink-reload command. This command will request a reload of the device driver.

Generic configuration parameters

The following is a list of generic configuration parameters that drivers may add. Use of generic parameters is preferred over each driver creating their own name.

Table 6: List of generic parameters

Name	Type	Description
enable_sriov	Bool	Enable Single Root I/O Virtualization (SRIOV) in the device.
ignore_ari	Bool	Ignore Alternative Routing-ID Interpretation (ARI) capability. If enabled, the adapter will ignore ARI capability even when the platform has support enabled. The device will create the same number of partitions as when the platform does not support ARI.
msix_x8c_max	U32	Provides the maximum number of MSI-X interrupts that a device can create. Value is the same across all physical functions (PFs) in the device.
msix_x8c_min	U32	Provides the minimum number of MSI-X interrupts required for the device to initialize. Value is the same across all physical functions (PFs) in the device.
fw_load_policy	U8	Control the device's firmware loading policy. <ul style="list-style-type: none"> DEVLINK_PARAM_FW_LOAD_POLICY_VALUE_DRIVER (0) Load firmware version preferred by the driver. DEVLINK_PARAM_FW_LOAD_POLICY_VALUE_FLASH (1) Load firmware currently stored in flash. DEVLINK_PARAM_FW_LOAD_POLICY_VALUE_DISK (2) Load firmware currently available on host's disk.
reset_on_drv_probe	U8	Controls the device's reset policy on driver probe. <ul style="list-style-type: none"> DEVLINK_PARAM_RESET_DEV_ON_DRV_PROBE_VALUE_UNKNOWN (0) Unknown or invalid value. DEVLINK_PARAM_RESET_DEV_ON_DRV_PROBE_VALUE_ALWAYS (1) Always reset device on driver probe. DEVLINK_PARAM_RESET_DEV_ON_DRV_PROBE_VALUE_NEVER (2) Never reset device on driver probe. DEVLINK_PARAM_RESET_DEV_ON_DRV_PROBE_VALUE_DISK (3) Reset the device only if firmware can be found in the filesystem.
enable_roce	Bool	Handle of RoCE traffic in the device.
enable_ebpf	Bool	When enabled, the device driver will instantiate Ethernet specific auxiliary device of the devlink device.
enable_rdma	Bool	When enabled, the device driver will instantiate RDMA specific auxiliary device of the devlink device.
enable_vdpa	Bool	When enabled, the device driver will instantiate VDPA networking specific auxiliary device of the devlink device.
enable_iwarp	Bool	Handle of iWARP traffic in the device.
internal_error_reset	Bool	When enabled, the device driver will reset the device on internal errors.
max_macs	U32	Typically macvlan, vlan net devices mac are also programmed in their parent net-device's Function rx filter. This parameter limit the maximum number of unicast mac address filters to receive traffic from per ethernet port of this device.
region_snapshot_table	Bool	Snapshot table of devlink-region snapshots.
remote_device_reset	Bool	Handle device reset by remote host. When cleared, the device driver will NACK any attempt of other host to reset the device. This parameter is useful for setups where a device is shared by different hosts, such as multi-host setup.
io_eq_size	U32	Control the size of I/O completion EQs.
event_size	U32	Control the size of asynchronous control events EQ.

8.3.6 Devlink Port

`devlink-port` is a port that exists on the device. It has a logically separate ingress/egress point of the device. A devlink port can be any one of many flavours. A devlink port flavour along with port attributes describe what a port represents.

A device driver that intends to publish a devlink port sets the devlink port attributes and registers the devlink port.

Devlink port flavours are described below.

Table 7: List of devlink port flavours

Flavour	Description
<code>DEVLINK_PORT_FLAVOUR_PHYSICAL</code>	<code>PHYSICAL</code> indicates physical port. This can be an eswitch physical port or any other physical port on the device.
<code>DEVLINK_PORT_FLAVOUR_DSA</code>	<code>DSA</code> indicates a DSA interconnect port.
<code>DEVLINK_PORT_FLAVOUR_CPU</code>	<code>CPU</code> indicates a CPU port applicable only to DSA.
<code>DEVLINK_PORT_FLAVOUR_PFS</code>	<code>PFS</code> indicates an eswitch port representing a port of PCI physical function (PF).
<code>DEVLINK_PORT_FLAVOUR_VFS</code>	<code>VFS</code> indicates an eswitch port representing a port of PCI virtual function (VF).
<code>DEVLINK_PORT_FLAVOUR_SF</code>	<code>SF</code> indicates an eswitch port representing a port of PCI subfunction (SF).
<code>DEVLINK_PORT_FLAVOUR_VIRTUAL</code>	<code>VIRTUAL</code> indicates a virtual port for the PCI virtual function.

Devlink port can have a different type based on the link layer described below.

Table 8: List of devlink port types

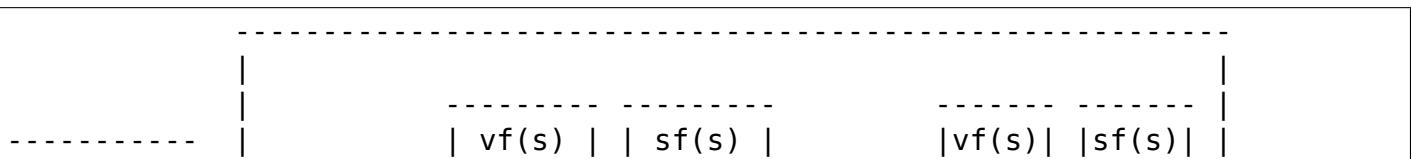
Type	Description
<code>DEVLINK_PORT_TYPE_Ethernet</code>	Driver should set this port type when a link layer of the port is Ethernet.
<code>DEVLINK_PORT_TYPE_IB</code>	Driver should set this port type when a link layer of the port is InfiniBand.
<code>DEVLINK_PORT_TYPE_AUTO</code>	Type is indicated by the user when driver should detect the port type automatically.

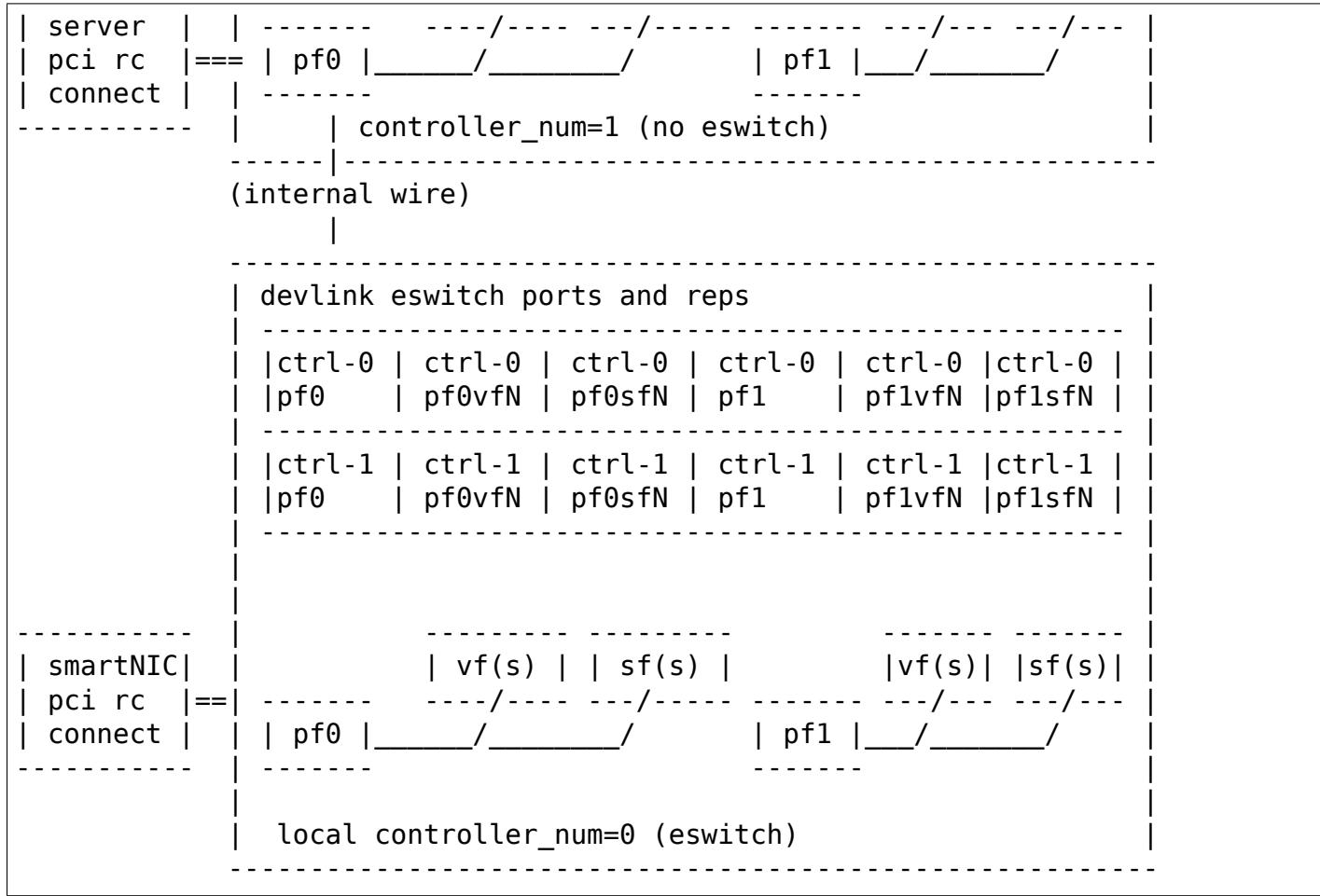
PCI controllers

In most cases a PCI device has only one controller. A controller consists of potentially multiple physical, virtual functions and subfunctions. A function consists of one or more ports. This port is represented by the devlink eswitch port.

A PCI device connected to multiple CPUs or multiple PCI root complexes or a SmartNIC, however, may have multiple controllers. For a device with multiple controllers, each controller is distinguished by a unique controller number. An eswitch is on the PCI device which supports ports of multiple controllers.

An example view of a system with two controllers:





In the above example, the external controller (identified by controller number = 1) doesn't have the eswitch. Local controller (identified by controller number = 0) has the eswitch. The Devlink instance on the local controller has eswitch devlink ports for both the controllers.

Function configuration

Users can configure one or more function attributes before enumerating the PCI function. Usually it means, user should configure function attribute before a bus specific device for the function is created. However, when SRIOV is enabled, virtual function devices are created on the PCI bus. Hence, function attribute should be configured before binding virtual function device to the driver. For subfunctions, this means user should configure port function attribute before activating the port function.

A user may set the hardware address of the function using *devlink port function set hw_addr* command. For Ethernet port function this means a MAC address.

Users may also set the RoCE capability of the function using *devlink port function set roce* command.

Users may also set the function as migratable using *devlink port function set migratable* command.

Users may also set the IPsec crypto capability of the function using *devlink port function set ipsec crypto* command.

Users may also set the IPsec packet capability of the function using *devlink port function set ipsec_packet* command.

Function attributes

MAC address setup

The configured MAC address of the PCI VF/SF will be used by netdevice and rdma device created for the PCI VF/SF.

- Get the MAC address of the VF identified by its unique devlink port index:

```
$ devlink port show pci/0000:06:00.0/2
pci/0000:06:00.0/2: type eth netdev enp6s0pf0vf1 flavour pcivf pfnum 0
 ↪vfn 1
   function:
     hw_addr 00:00:00:00:00:00
```

- Set the MAC address of the VF identified by its unique devlink port index:

```
$ devlink port function set pci/0000:06:00.0/2 hw_addr 00:11:22:33:44:55

$ devlink port show pci/0000:06:00.0/2
pci/0000:06:00.0/2: type eth netdev enp6s0pf0vf1 flavour pcivf pfnum 0
 ↪vfn 1
   function:
     hw_addr 00:11:22:33:44:55
```

- Get the MAC address of the SF identified by its unique devlink port index:

```
$ devlink port show pci/0000:06:00.0/32768
pci/0000:06:00.0/32768: type eth netdev enp6s0pf0sf88 flavour pcisf pfnum 0
 ↪0 sfnum 88
   function:
     hw_addr 00:00:00:00:00:00
```

- Set the MAC address of the SF identified by its unique devlink port index:

```
$ devlink port function set pci/0000:06:00.0/32768 hw_addr 00:00:00:00:88:88

$ devlink port show pci/0000:06:00.0/32768
pci/0000:06:00.0/32768: type eth netdev enp6s0pf0sf88 flavour pcisf pfnum 0
 ↪0 sfnum 88
   function:
     hw_addr 00:00:00:00:88:88
```

RoCE capability setup

Not all PCI VFs/SFs require RoCE capability.

When RoCE capability is disabled, it saves system memory per PCI VF/SF.

When user disables RoCE capability for a VF/SF, user application cannot send or receive any RoCE packets through this VF/SF and RoCE GID table for this PCI will be empty.

When RoCE capability is disabled in the device using port function attribute, VF/SF driver cannot override it.

- Get RoCE capability of the VF device:

```
$ devlink port show pci/0000:06:00.0/2
pci/0000:06:00.0/2: type eth netdev enp6s0pf0vf1 flavour pcivf pfnum 0
  ↪vfn 1
    function:
      hw_addr 00:00:00:00:00:00 roce enable
```

- Set RoCE capability of the VF device:

```
$ devlink port function set pci/0000:06:00.0/2 roce disable

$ devlink port show pci/0000:06:00.0/2
pci/0000:06:00.0/2: type eth netdev enp6s0pf0vf1 flavour pcivf pfnum 0
  ↪vfn 1
    function:
      hw_addr 00:00:00:00:00:00 roce disable
```

migratable capability setup

Live migration is the process of transferring a live virtual machine from one physical host to another without disrupting its normal operation.

User who want PCI VFs to be able to perform live migration need to explicitly enable the VF migratable capability.

When user enables migratable capability for a VF, and the HV binds the VF to VFIO driver with migration support, the user can migrate the VM with this VF from one HV to a different one.

However, when migratable capability is enable, device will disable features which cannot be migrated. Thus migratable cap can impose limitations on a VF so let the user decide.

Example of LM with migratable function configuration:

- Get migratable capability of the VF device:

```
$ devlink port show pci/0000:06:00.0/2
pci/0000:06:00.0/2: type eth netdev enp6s0pf0vf1 flavour pcivf pfnum 0 vfn 1
  function:
    hw_addr 00:00:00:00:00:00 migratable disable
```

- Set migratable capability of the VF device:

```
$ devlink port function set pci/0000:06:00.0/2 migratable enable
$ devlink port show pci/0000:06:00.0/2
pci/0000:06:00.0/2: type eth netdev enp6s0pf0vf1 flavour pcivf pfnum 0
    ↪vfn 1
    function:
        hw_addr 00:00:00:00:00:00 migratable enable
```

- Bind VF to VFIO driver with migration support:

```
$ echo <pci_id> > /sys/bus/pci/devices/0000:08:00.0/driver/unbind
$ echo mlx5_vfio_pci > /sys/bus/pci/devices/0000:08:00.0/driver_override
$ echo <pci_id> > /sys/bus/pci/devices/0000:08:00.0/driver/bind
```

Attach VF to the VM. Start the VM. Perform live migration.

IPsec crypto capability setup

When user enables IPsec crypto capability for a VF, user application can offload XFRM state crypto operation (Encrypt/Decrypt) to this VF.

When IPsec crypto capability is disabled (default) for a VF, the XFRM state is processed in software by the kernel.

- Get IPsec crypto capability of the VF device:

```
$ devlink port show pci/0000:06:00.0/2
pci/0000:06:00.0/2: type eth netdev enp6s0pf0vf1 flavour pcivf pfnum 0
    ↪vfn 1
    function:
        hw_addr 00:00:00:00:00:00 ipsec_crypto disabled
```

- Set IPsec crypto capability of the VF device:

```
$ devlink port function set pci/0000:06:00.0/2 ipsec_crypto enable
$ devlink port show pci/0000:06:00.0/2
pci/0000:06:00.0/2: type eth netdev enp6s0pf0vf1 flavour pcivf pfnum 0
    ↪vfn 1
    function:
        hw_addr 00:00:00:00:00:00 ipsec_crypto enabled
```

IPsec packet capability setup

When user enables IPsec packet capability for a VF, user application can offload XFRM state and policy crypto operation (Encrypt/Decrypt) to this VF, as well as IPsec encapsulation.

When IPsec packet capability is disabled (default) for a VF, the XFRM state and policy is processed in software by the kernel.

- Get IPsec packet capability of the VF device:

```
$ devlink port show pci/0000:06:00.0/2
pci/0000:06:00.0/2: type eth netdev enp6s0pf0vf1 flavour pcivf pfnum 0
  ↳vfn 1
    function:
      hw_addr 00:00:00:00:00:00 ipsec_packet disabled
```

- Set IPsec packet capability of the VF device:

```
$ devlink port function set pci/0000:06:00.0/2 ipsec_packet enable
$ devlink port show pci/0000:06:00.0/2
pci/0000:06:00.0/2: type eth netdev enp6s0pf0vf1 flavour pcivf pfnum 0
  ↳vfn 1
    function:
      hw_addr 00:00:00:00:00:00 ipsec_packet enabled
```

Subfunction

Subfunction is a lightweight function that has a parent PCI function on which it is deployed. Subfunction is created and deployed in unit of 1. Unlike SRIOV VFs, a subfunction doesn't require its own PCI virtual function. A subfunction communicates with the hardware through the parent PCI function.

To use a subfunction, 3 steps setup sequence is followed:

- 1) create - create a subfunction;
- 2) configure - configure subfunction attributes;
- 3) deploy - deploy the subfunction;

Subfunction management is done using devlink port user interface. User performs setup on the subfunction management device.

(1) Create

A subfunction is created using a devlink port interface. A user adds the subfunction by adding a devlink port of subfunction flavour. The devlink kernel code calls down to subfunction management driver (devlink ops) and asks it to create a subfunction devlink port. Driver then instantiates the subfunction port and any associated objects such as health reporters and representor netdevice.

(2) Configure

A subfunction devlink port is created but it is not active yet. That means the entities are created on devlink side, the e-switch port representor is created, but the subfunction device itself is not created. A user might use e-switch port representor to do settings, putting it into bridge, adding TC rules, etc. A user might as well configure the hardware address (such as MAC address) of the subfunction while subfunction is inactive.

(3) Deploy

Once a subfunction is configured, user must activate it to use it. Upon activation, subfunction management driver asks the subfunction management device to instantiate the subfunction device on particular PCI function. A subfunction device is created on the Documentation/driver-api/auxiliary_bus.rst. At this point a matching subfunction driver binds to the subfunction's auxiliary device.

Rate object management

Devlink provides API to manage tx rates of single devlink port or a group. This is done through rate objects, which can be one of the two types:

leaf

Represents a single devlink port; created/destroyed by the driver. Since leaf have 1to1 mapping to its devlink port, in user space it is referred as `pci/<bus_addr>/<port_index>`;

node

Represents a group of rate objects (leafs and/or nodes); created/deleted by request from the userspace; initially empty (no rate objects added). In userspace it is referred as `pci/<bus_addr>/<node_name>`, where `node_name` can be any identifier, except decimal number, to avoid collisions with leafs.

API allows to configure following rate object's parameters:

tx_share

Minimum TX rate value shared among all other rate objects, or rate objects that parts of the parent group, if it is a part of the same group.

tx_max

Maximum TX rate value.

tx_priority

Allows for usage of strict priority arbiter among siblings. This arbitration scheme attempts

to schedule nodes based on their priority as long as the nodes remain within their bandwidth limit. The higher the priority the higher the probability that the node will get selected for scheduling.

tx_weight

Allows for usage of Weighted Fair Queuing arbitration scheme among siblings. This arbitration scheme can be used simultaneously with the strict priority. As a node is configured with a higher rate it gets more BW relative to its siblings. Values are relative like a percentage points, they basically tell how much BW should node take relative to its siblings.

parent

Parent node name. Parent node rate limits are considered as additional limits to all node children limits. `tx_max` is an upper limit for children. `tx_share` is a total bandwidth distributed among children.

`tx_priority` and `tx_weight` can be used simultaneously. In that case nodes with the same priority form a WFQ subgroup in the sibling group and arbitration among them is based on assigned weights.

Arbitration flow from the high level:

1. Choose a node, or group of nodes with the highest priority that stays within the BW limit and are not blocked. Use `tx_priority` as a parameter for this arbitration.
2. If group of nodes have the same priority perform WFQ arbitration on that subgroup. Use `tx_weight` as a parameter for this arbitration.
3. Select the winner node, and continue arbitration flow among its children, until leaf node is reached, and the winner is established.
4. If all the nodes from the highest priority sub-group are satisfied, or overused their assigned BW, move to the lower priority nodes.

Driver implementations are allowed to support both or either rate object types and setting methods of their parameters. Additionally driver implementation may export nodes/leafs and their child-parent relationships.

Terms and Definitions

Table 9: Terms and Definitions

Term	Definitions
PCI device	A physical PCI device having one or more PCI buses consists of one or more PCI controllers.
PCI controller	A controller consists of potentially multiple physical functions, virtual functions and subfunctions.
Port function	An object to manage the function of a port.
Subfunction	A lightweight function that has parent PCI function on which it is deployed.
Subfunction device	A bus device of the subfunction, usually on a auxiliary bus.
Subfunction driver	A device driver for the subfunction auxiliary device.
Subfunction management device	A PCI physical function that supports subfunction management.
Subfunction management driver	A device driver for PCI physical function that supports subfunction management using devlink port interface.
Subfunction host driver	A device driver for PCI physical function that hosts subfunction devices. In most cases it is same as subfunction management driver. When subfunction is used on external controller, subfunction management and host drivers are different.

8.3.7 Devlink Region

devlink regions enable access to driver defined address regions using devlink.

Each device can create and register its own supported address regions. The region can then be accessed via the devlink region interface.

Region snapshots are collected by the driver, and can be accessed via read or dump commands. This allows future analysis on the created snapshots. Regions may optionally support triggering snapshots on demand.

Snapshot identifiers are scoped to the devlink instance, not a region. All snapshots with the same snapshot id within a devlink instance correspond to the same event.

The major benefit to creating a region is to provide access to internal address regions that are otherwise inaccessible to the user.

Regions may also be used to provide an additional way to debug complex error states, but see also [Devlink Health](#)

Regions may optionally support capturing a snapshot on demand via the `DEVLINK_CMD_REGION_NEW` netlink message. A driver wishing to allow requested snapshots must implement the `.snapshot` callback for the region in its `devlink_region_ops` structure. If snapshot id is not set in the `DEVLINK_CMD_REGION_NEW` request kernel will allocate one and send the snapshot information to user space.

Regions may optionally allow directly reading from their contents without a snapshot. Direct

read requests are not atomic. In particular a read request of size 256 bytes or larger will be split into multiple chunks. If atomic access is required, use a snapshot. A driver wishing to enable this for a region should implement the .read callback in the devlink_region_ops structure. User space can request a direct read by using the DEVLINK_ATTR_REGION_DIRECT attribute instead of specifying a snapshot id.

example usage

```
$ devlink region help
$ devlink region show [ DEV/REGION ]
$ devlink region del DEV/REGION snapshot SNAPSHOT_ID
$ devlink region dump DEV/REGION [ snapshot SNAPSHOT_ID ]
$ devlink region read DEV/REGION [ snapshot SNAPSHOT_ID ] address ADDRESS
  ↳length length

# Show all of the exposed regions with region sizes:
$ devlink region show
pci/0000:00:05.0/cr-space: size 1048576 snapshot [1 2] max 8
pci/0000:00:05.0/fw-health: size 64 snapshot [1 2] max 8

# Delete a snapshot using:
$ devlink region del pci/0000:00:05.0/cr-space snapshot 1

# Request an immediate snapshot, if supported by the region
$ devlink region new pci/0000:00:05.0/cr-space
pci/0000:00:05.0/cr-space: snapshot 5

# Dump a snapshot:
$ devlink region dump pci/0000:00:05.0/fw-health snapshot 1
0000000000000000 0014 95dc 0014 9514 0035 1670 0034 db30
0000000000000010 0000 0000 ffff ff04 0029 8c00 0028 8cc8
0000000000000020 0016 0bb8 0016 1720 0000 0000 c00f 3ffc
0000000000000030 bada cce5 bada cce5 bada cce5

# Read a specific part of a snapshot:
$ devlink region read pci/0000:00:05.0/fw-health snapshot 1 address 0 length 16
0000000000000000 0014 95dc 0014 9514 0035 1670 0034 db30

# Read from the region without a snapshot
$ devlink region read pci/0000:00:05.0/fw-health address 16 length 16
0000000000000010 0000 0000 ffff ff04 0029 8c00 0028 8cc8
```

As regions are likely very device or driver specific, no generic regions are defined. See the driver-specific documentation files for information on the specific regions a driver supports.

8.3.8 Devlink Resource

devlink provides the ability for drivers to register resources, which can allow administrators to see the device restrictions for a given resource, as well as how much of the given resource is currently in use. Additionally, these resources can optionally have configurable size. This could enable the administrator to limit the number of resources that are used.

For example, the `netdevsim` driver enables `/IPv4/fib` and `/IPv4/fib-rules` as resources to limit the number of IPv4 FIB entries and rules for a given device.

Resource Ids

Each resource is represented by an id, and contains information about its current size and related sub resources. To access a sub resource, you specify the path of the resource. For example `/IPv4/fib` is the id for the `fib` sub-resource under the `IPv4` resource.

Generic Resources

Generic resources are used to describe resources that can be shared by multiple device drivers and their description must be added to the following table:

Table 10: List of Generic Resources

Name	Description
<code>physical_ports</code>	capacity of physical ports that the switch ASIC can support

example usage

The resources exposed by the driver can be observed, for example:

```
$devlink resource show pci/0000:03:00.0
pci/0000:03:00.0:
    name kvd size 245760 unit entry
    resources:
        name linear size 98304 occ 0 unit entry size_min 0 size_max 147456 size_
        ↵gran 128
        name hash_double size 60416 unit entry size_min 32768 size_max 180224
        ↵size_gran 128
        name hash_single size 87040 unit entry size_min 65536 size_max 212992
        ↵size_gran 128
```

Some resource's size can be changed. Examples:

```
$devlink resource set pci/0000:03:00.0 path /kvd/hash_single size 73088
$devlink resource set pci/0000:03:00.0 path /kvd/hash_double size 74368
```

The changes do not apply immediately, this can be validated by the 'size_new' attribute, which represents the pending change in size. For example:

```
$devlink resource show pci/0000:03:00.0
pci/0000:03:00.0:
  name kvd size 245760 unit entry size_valid false
  resources:
    name linear size 98304 size_new 147456 occ 0 unit entry size_min 0 size_
    ↵max 147456 size_gran 128
    name hash_double size 60416 unit entry size_min 32768 size_max 180224 size_
    ↵gran 128
    name hash_single size 87040 unit entry size_min 65536 size_max 212992 size_
    ↵gran 128
```

Note that changes in resource size may require a device reload to properly take effect.

8.3.9 Devlink Reload

`devlink-reload` provides mechanism to reinit driver entities, applying `devlink-params` and `devlink-resources` new values. It also provides mechanism to activate firmware.

Reload Actions

User may select a reload action. By default `driver_reinit` action is selected.

Table 11: Possible reload actions

Name	Description
<code>driver_reinit</code>	Driver entities re-initialization, including applying new values to devlink entities which are used during driver load which are: <ul style="list-style-type: none"> • <code>devlink-params</code> in configuration mode <code>driverinit</code> • <code>devlink-resources</code> Other devlink entities may stay over the re-initialization: <ul style="list-style-type: none"> • <code>devlink-health-reporter</code> • <code>devlink-region</code> The rest of the devlink entities have to be removed and readded.
<code>fw_activate</code>	Firmware activate. Activates new firmware if such image is stored and pending activation. If no limitation specified this action may involve firmware reset. If no new image pending this action will reload current firmware image.

Note that even though user asks for a specific action, the driver implementation might require to perform another action alongside with it. For example, some driver do not support driver reinitialization being performed without fw activation. Therefore, the `devlink reload` command returns the list of actions which were actually performed.

Reload Limits

By default reload actions are not limited and driver implementation may include reset or down-time as needed to perform the actions.

However, some drivers support action limits, which limit the action implementation to specific constraints.

Table 12: Possible reload limits

Name	Description
no_reset	reset allowed, no down time allowed, no link flap and no configuration is lost.

Change Namespace

The netns option allows user to be able to move devlink instances into namespaces during devlink reload operation. By default all devlink instances are created in init_net and stay there.

example usage

```
$ devlink dev reload help
$ devlink dev reload DEV [ netns { PID | NAME | ID } ] [ action { driver_
    ↵reinit | fw_activate } ] [ limit no_reset ]

# Run reload command for devlink driver entities re-initialization:
$ devlink dev reload pci/0000:82:00.0 action driver_reinit
reload_actions_performed:
    driver_reinit

# Run reload command to activate firmware:
# Note that mlx5 driver reloads the driver while activating firmware
$ devlink dev reload pci/0000:82:00.0 action fw_activate
reload_actions_performed:
    driver_reinit fw_activate
```

8.3.10 Devlink Selftests

The devlink-selftests API allows executing selftests on the device.

Tests Mask

The devlink-selftests command should be run with a mask indicating the tests to be executed.

Tests Description

The following is a list of tests that drivers may execute.

Table 13: List of tests

Name	Description
DEVLINK_SET_TESTS_FLASH	The firmware on non-volatile memory on the board, e.g. flash. This particular test helps to run a flash selftest on the device. Implementation of the test is left to the driver/firmware.

example usage

```
# Query selftests supported on the devlink device
$ devlink dev selftests show DEV
# Query selftests supported on all devlink devices
$ devlink dev selftests show
# Executes selftests on the device
$ devlink dev selftests run DEV id flash
```

8.3.11 Devlink Trap

Background

Devices capable of offloading the kernel's datapath and perform functions such as bridging and routing must also be able to send specific packets to the kernel (i.e., the CPU) for processing.

For example, a device acting as a multicast-aware bridge must be able to send IGMP membership reports to the kernel for processing by the bridge module. Without processing such packets, the bridge module could never populate its MDB.

As another example, consider a device acting as router which has received an IP packet with a TTL of 1. Upon routing the packet the device must send it to the kernel so that it will route it as well and generate an ICMP Time Exceeded error datagram. Without letting the kernel route such packets itself, utilities such as traceroute could never work.

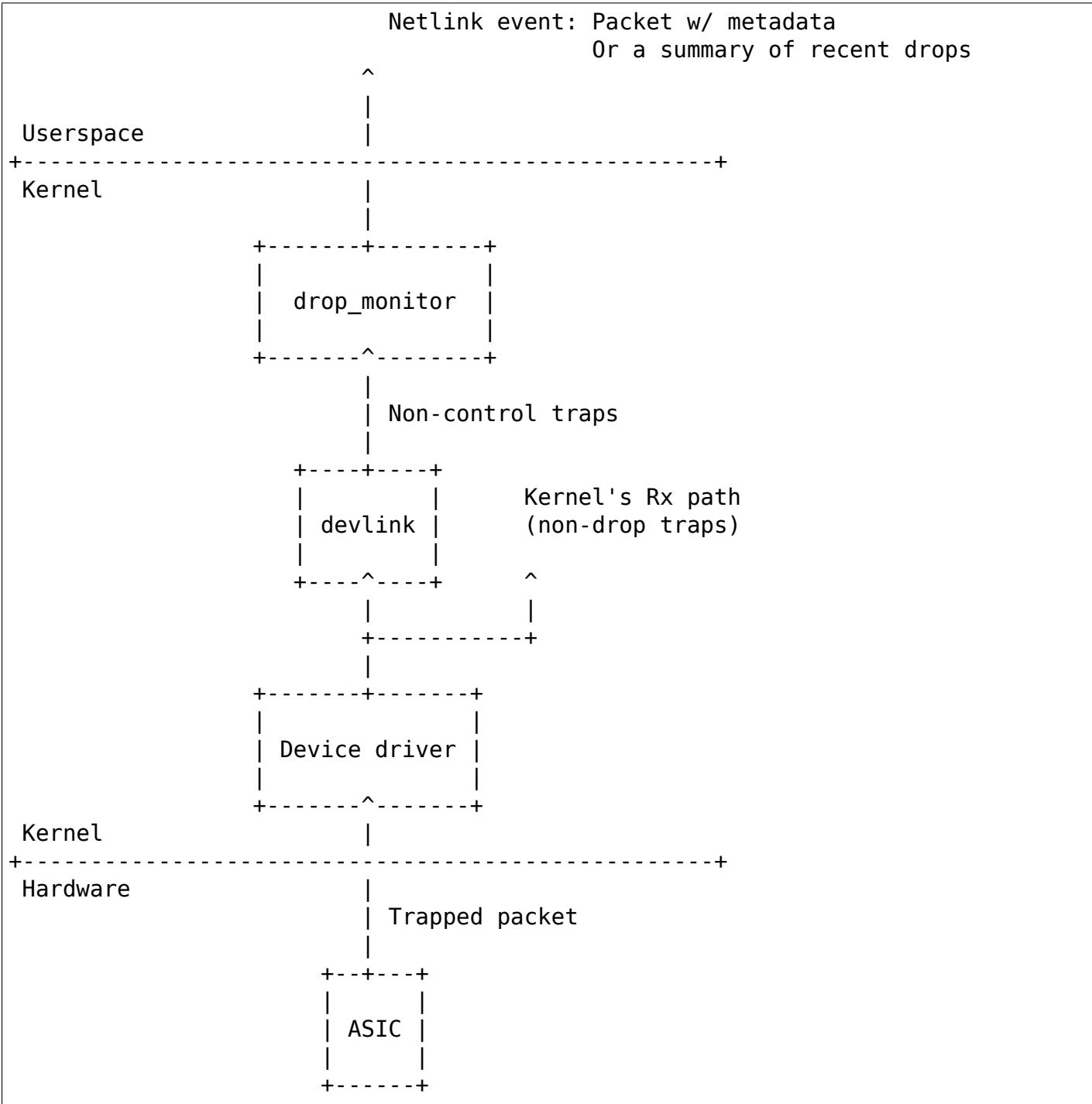
The fundamental ability of sending certain packets to the kernel for processing is called "packet trapping".

Overview

The devlink-trap mechanism allows capable device drivers to register their supported packet traps with devlink and report trapped packets to devlink for further analysis.

Upon receiving trapped packets, devlink will perform a per-trap packets and bytes accounting and potentially report the packet to user space via a netlink event along with all the provided metadata (e.g., trap reason, timestamp, input port). This is especially useful for drop traps (see [Trap Types](#)) as it allows users to obtain further visibility into packet drops that would otherwise be invisible.

The following diagram provides a general overview of devlink-trap:



Trap Types

The devlink-trap mechanism supports the following packet trap types:

- **drop:** Trapped packets were dropped by the underlying device. Packets are only processed by devlink and not injected to the kernel's Rx path. The trap action (see [Trap Actions](#)) can be changed.
- **exception:** Trapped packets were not forwarded as intended by the underlying device due to an exception (e.g., TTL error, missing neighbour entry) and trapped to the control plane for resolution. Packets are processed by devlink and injected to the kernel's Rx path.

path. Changing the action of such traps is not allowed, as it can easily break the control plane.

- control: Trapped packets were trapped by the device because these are control packets required for the correct functioning of the control plane. For example, ARP request and IGMP query packets. Packets are injected to the kernel's Rx path, but not reported to the kernel's drop monitor. Changing the action of such traps is not allowed, as it can easily break the control plane.

Trap Actions

The devlink-trap mechanism supports the following packet trap actions:

- trap: The sole copy of the packet is sent to the CPU.
- drop: The packet is dropped by the underlying device and a copy is not sent to the CPU.
- mirror: The packet is forwarded by the underlying device and a copy is sent to the CPU.

Generic Packet Traps

Generic packet traps are used to describe traps that trap well-defined packets or packets that are trapped due to well-defined conditions (e.g., TTL error). Such traps can be shared by multiple device drivers and their description must be added to the following table:

Table 14: List of Generic Packet Traps

Name	Type	Description
source_trap	multicast	packets that the device decided to drop because of a multicast source MAC
vlandtag_trap	missmatch	packets that the device decided to drop in case of VLAN tag mismatch: The ingress bridge port is not configured with a PVID and the packet is untagged or prio-tagged
ingress_trap	missmatch	packets that the device decided to drop in case they are tagged with a VLAN that is not configured on the ingress bridge port
ingress_stp_trap	ingoing_packets	packets that the device decided to drop in case the STP state of the ingress bridge port is not "forwarding"
port_dkbp_trap	unknow	packets that the device decided to drop in case they need to be flooded (e.g., unknown unicast, unregistered multicast) and there are no ports the packets should be flooded to
port_dkbp_trap	pkts	packets that the device decided to drop in case after layer 2 forwarding the only port from which they should be transmitted through is the port from which they were received
blackhole_trap	pkts	packets that the device decided to drop in case they hit a blackhole route
ttl_expired_trap	too_cst	packets that should be forwarded by the device whose TTL was decremented to 0 or less
tail_drop_trap	pkts	packets that the device decided to drop because they could not be enqueued to a transmission queue which is full
non_ip_drop_trap	pkts	packets that the device decided to drop because they need to undergo a layer 3 lookup, but are not IP or MPLS packets

continues on next page

Table 14 – continued from previous page

uc_ddrop	Traps packets that the device decided to drop because they need to be routed and they have a unicast destination IP and a multicast destination MAC
dip_dsopt	Traps packets that the device decided to drop because they need to be routed and their destination IP is the loopback address (i.e., 127.0.0.0/8 and ::1/128)
sip_dsopt	Traps packets that the device decided to drop because they need to be routed and their source IP is multicast (i.e., 224.0.0.0/8 and ff::/8)
sip_dsopt	Traps packets that the device decided to drop because they need to be routed and their source IP is the loopback address (i.e., 127.0.0.0/8 and ::1/128)
ip_header	Traps packets that the device decided to drop because they need to be routed and their IP header is corrupted: wrong checksum, wrong IP version or too short Internet Header Length (IHL)
ipv4dsopt	Traps packets that the device decided to drop because they need to be routed and their source IP is limited broadcast (i.e., 255.255.255.255/32)
ipv6dmopt	Traps IPv6 packets that the device decided to drop because they need to be routed and their IPv6 multicast destination IP has a reserved scope (i.e., ffx0::/16)
ipv6dmopt	Traps IPv6 packets that the device decided to drop because they need to be routed and their IPv6 multicast destination IP has an interface-local scope (i.e., ffx1::/16)
mtu_exceed	Traps packets that should have been routed by the device, but were bigger than the MTU of the egress interface
unreachable	Traps packets that did not have a matching IP neighbour after routing
mc_reject	Traps multicast IP packets that failed reverse-path forwarding (RPF) check during multicast routing
reject	Traps packets that hit reject routes (i.e., "unreachable", "prohibit")
ipv4exp	Traps unicast IPv4 packets that did not match any route
ipv6exp	Traps unicast IPv6 packets that did not match any route
non_droppable	Traps packets that the device decided to drop because they are not supposed to be routed. For example, IGMP queries can be flooded by the device in layer 2 and reach the router. Such packets should not be routed and instead dropped
decap_error	Traps NVE and IPinIP packets that the device decided to drop because of failure during decapsulation (e.g., packet being too short, reserved bits set in VXLAN header)
overlays	Traps NVIF packets that the device decided to drop because their overlay source MAC is multicast
ingressdrop	Traps packets dropped during processing of ingress flow action drop
egressdrop	Traps packets dropped during processing of egress flow action drop
stp_ctrl	Traps STP packets
lacpctrl	Traps LACP packets
lldpctrl	Traps LLDP packets
igmpcq	Traps IGMP Membership Query packets
igmpv1t	Traps IGMP Version 1 Membership Report packets
igmpv2t	Traps IGMP Version 2 Membership Report packets
igmpv3t	Traps IGMP Version 3 Membership Report packets
igmpv2l	Traps IGMP Version 2 Leave Group packets
mld_query	Traps MLD Multicast Listener Query packets
mld_v1r	Traps MLD Version 1 Multicast Listener Report packets
mld_v2r	Traps MLD Version 2 Multicast Listener Report packets
mld_v1d	Traps MLD Version 1 Multicast Listener Done packets
ipv4cdht	Traps IPv4 DHCP packets
ipv6cdht	Traps IPv6 DHCP packets

continues on next page

Table 14 – continued from previous page

<code>arp_ceq</code>	Traps ARP request packets
<code>arp_cres</code>	Traps ARP response packets
<code>arp_over</code>	Traps NVE-decapsulated ARP packets that reached the overlay network. This is required, for example, when the address that needs to be resolved is a local address
<code>ipv6coet</code>	Traps Neighbour Solicitation packets
<code>ipv6coif</code>	Traps IPv6 Neighbour Advertisement packets
<code>ipv4cbfd</code>	Traps IPv4 BFD packets
<code>ipv6cbfd</code>	Traps IPv6 BFD packets
<code>ipv4cosp</code>	Traps IPv4 OSPF packets
<code>ipv6cosp</code>	Traps IPv6 OSPF packets
<code>ipv4cbgp</code>	Traps IPv4 BGP packets
<code>ipv6cbgp</code>	Traps IPv6 BGP packets
<code>ipv4cont</code>	Traps IPv4 VRRP packets
<code>ipv6cont</code>	Traps IPv6 VRRP packets
<code>ipv4cpim</code>	Traps IPv4 PIM packets
<code>ipv6cpim</code>	Traps IPv6 PIM packets
<code>uc_loopback</code>	Traps unicast packets that need to be routed through the same layer 3 interface from which they were received. Such packets are routed by the kernel, but also cause it to potentially generate ICMP redirect packets
<code>locatonto</code>	Traps unicast packets that hit a local route and need to be locally delivered
<code>extecon</code>	Traps packets that should be routed through an external interface (e.g., management interface) that does not belong to the same device (e.g., switch ASIC) as the ingress interface
<code>ipv6contlinklocal</code>	Traps link-local IPv6 packets that need to be routed and have a destination IP address with a link-local scope (i.e., fe80::/10). The trap allows device drivers to avoid programming link-local routes, but still receive packets for local delivery
<code>ipv6cdmpt</code>	Traps IPv6 packets that their destination IP address is the "All Nodes Address" (i.e., ff02::1)
<code>ipv6cdmpt</code>	Traps IPv6 packets that their destination IP address is the "All Routers Address" (i.e., ff02::2)
<code>ipv6conttrap</code>	Traps IPv6 Router Solicitation packets
<code>ipv6conttrapadv</code>	Traps IPv6 Router Advertisement packets
<code>ipv6conttrap</code>	Traps IPv6 Redirect Message packets
<code>ipv4conttrap</code>	Traps IPv4 packets that need to be routed and include the Router Alert option. Such packets need to be locally delivered to raw sockets that have the IP_ROUTER_ALERT socket option set
<code>ipv6conttrap</code>	Traps IPv6 packets that need to be routed and include the Router Alert option in their Hop-by-Hop extension header. Such packets need to be locally delivered to raw sockets that have the IPV6_ROUTER_ALERT socket option set
<code>ptp_event</code>	Traps PTP time-critical event messages (Sync, Delay_req, Pdelay_Req and Pdelay_Resp)
<code>ptp_genet</code>	Traps PTP general messages (Announce, Follow_Up, Delay_Resp, Pdelay_Resp_Follow_Up, management and signaling)
<code>flowcenttrap</code>	Traps packets sampled during processing of flow action sample (e.g., via tc's sample action)
<code>flowcenttrap</code>	Traps packets logged during processing of flow action trap (e.g., via tc's trap action)
<code>earlydrop</code>	Traps packets dropped due to the RED (Random Early Detection) algorithm (i.e., early drops)

continues on next page

Table 14 – continued from previous page

<code>vxladrpt\$rops</code>	packets dropped due to an error in the VXLAN header parsing which might be because of packet truncation or the I flag is not set.
<code>llcsnapTrops</code>	packets dropped due to an error in the LLC+SNAP header parsing
<code>vlandpap\$rops</code>	packets dropped due to an error in the VLAN header parsing. Could include unexpected packet truncation.
<code>pppoedpppTrops</code>	packets dropped due to an error in the PPPoE+PPP header parsing. This could include finding a session ID of 0xFFFF (which is reserved and not for use), a PPPoE length which is larger than the frame received or any common error on this type of header
<code>mplsdpap\$rops</code>	packets dropped due to an error in the MPLS header parsing which could include unexpected header truncation
<code>arp_drops</code>	packets dropped due to an error in the ARP header parsing
<code>ip_1dpap\$rops</code>	packets dropped due to an error in the first IP header parsing. This packet trap could include packets which do not pass an IP checksum check, a header length check (a minimum of 20 bytes), which might suffer from packet truncation thus the total length field exceeds the received packet length etc
<code>ip_ndpap\$rops</code>	packets dropped due to an error in the parsing of the last IP header (the inner one in case of an IP over IP tunnel). The same common error checking is performed here as for the ip_1_parsing trap
<code>gre_drops</code>	packets dropped due to an error in the GRE header parsing
<code>udp_drops</code>	packets dropped due to an error in the UDP header parsing. This packet trap could include checksum errors, an improper UDP length detected (smaller than 8 bytes) or detection of header truncation.
<code>tcp_drops</code>	packets dropped due to an error in the TCP header parsing. This could include TCP checksum errors, improper combination of SYN, FIN and/or RESET etc.
<code>ipsecdrops</code>	packets dropped due to an error in the IPSEC header parsing
<code>sctpdrops</code>	packets dropped due to an error in the SCTP header parsing. This would mean that port number 0 was used or that the header is truncated.
<code>dccpdrops</code>	packets dropped due to an error in the DCCP header parsing
<code>gtp_drops</code>	packets dropped due to an error in the GTP header parsing
<code>esp_drops</code>	packets dropped due to an error in the ESP header parsing
<code>blackhole\$rops</code>	packets that the device decided to drop in case they hit a blackhole nexthop
<code>dmacdf0\$rops</code>	incoming packets that the device decided to drop because the destination MAC is not configured in the MAC table and the interface is not in promiscuous mode
<code>eapolcont\$rops</code>	"Extensible Authentication Protocol over LAN" (EAPOL) packets specified in IEEE 802.1X
<code>lockedport\$rops</code>	packets that the device decided to drop because they failed the locked bridge port check. That is, packets that were received via a locked port and whose {SMAC, VID} does not correspond to an FDB entry pointing to the port

Driver-specific Packet Traps

Device drivers can register driver-specific packet traps, but these must be clearly documented. Such traps can correspond to device-specific exceptions and help debug packet drops caused by these exceptions. The following list includes links to the description of driver-specific traps registered by various device drivers:

- [*netdevsim devlink support*](#)
- [*mlxsw devlink support*](#)
- [*prestera devlink support*](#)

Generic Packet Trap Groups

Generic packet trap groups are used to aggregate logically related packet traps. These groups allow the user to batch operations such as setting the trap action of all member traps. In addition, `devlink-trap` can report aggregated per-group packets and bytes statistics, in case per-trap statistics are too narrow. The description of these groups must be added to the following table:

Table 15: List of Generic Packet Trap Groups

Name	Description
l2_drop	Contains packet traps for packets that were dropped by the device during layer 2 forwarding (i.e., bridge)
l3_drop	Contains packet traps for packets that were dropped by the device during layer 3 forwarding
l3_exception	Contains packet traps for packets that hit an exception (e.g., TTL error) during layer 3 forwarding
buffer_decision	Contains packet traps for packets that were dropped by the device due to an enqueue decision
tunnel_decision	Contains packet traps for packets that were dropped by the device during tunnel encapsulation / decapsulation
acl_drop	Contains packet traps for packets that were dropped by the device during ACL processing
stp	Contains packet traps for STP packets
lacp	Contains packet traps for LACP packets
lldp	Contains packet traps for LLDP packets
mc_snooping	Contains packet traps for IGMP and MLD packets required for multicast snooping
dhcp	Contains packet traps for DHCP packets
neigh_discovers	Contains packet traps for neighbour discovery packets (e.g., ARP, IPv6 ND)
bfd	Contains packet traps for BFD packets
ospf	Contains packet traps for OSPF packets
bgp	Contains packet traps for BGP packets
vrrp	Contains packet traps for VRRP packets
pim	Contains packet traps for PIM packets
uc_loopback	Contains a packet trap for unicast loopback packets (i.e., uc_loopback). This trap is singled-out because in cases such as one-armed router it will be constantly triggered. To limit the impact on the CPU usage, a packet trap policer with a low rate can be bound to the group without affecting other traps
local_delivery	Contains packet traps for packets that should be locally delivered after routing, but do not match more specific packet traps (e.g., ipv4_bgp)
external	Contains packet traps for packets that should be routed through an external interface (e.g., management interface) that does not belong to the same device (e.g., switch ASIC) as the ingress interface
ipv6	Contains packet traps for various IPv6 control packets (e.g., Router Advertisements)
ptp_event	Contains packet traps for PTP time-critical event messages (Sync, Delay_req, Pdelay_Req and Pdelay_Resp)
ptp_general	Contains packet traps for PTP general messages (Announce, Follow_Up, Delay_Resp, Pdelay_Resp_Follow_Up, management and signaling)
acl_sample	Contains packet traps for packets that were sampled by the device during ACL processing
acl_trap	Contains packet traps for packets that were trapped (logged) by the device during ACL processing
parser_error	Contains packet traps for packets that were marked by the device during parsing as erroneous
eapol	Contains packet traps for "Extensible Authentication Protocol over LAN" (EAPOL) packets specified in IEEE 802.1X

Packet Trap Policers

As previously explained, the underlying device can trap certain packets to the CPU for processing. In most cases, the underlying device is capable of handling packet rates that are several orders of magnitude higher compared to those that can be handled by the CPU.

Therefore, in order to prevent the underlying device from overwhelming the CPU, devices usually include packet trap policers that are able to police the trapped packets to rates that can be handled by the CPU.

The devlink-trap mechanism allows capable device drivers to register their supported packet trap policers with devlink. The device driver can choose to associate these policers with supported packet trap groups (see [Generic Packet Trap Groups](#)) during its initialization, thereby exposing its default control plane policy to user space.

Device drivers should allow user space to change the parameters of the policers (e.g., rate, burst size) as well as the association between the policers and trap groups by implementing the relevant callbacks.

If possible, device drivers should implement a callback that allows user space to retrieve the number of packets that were dropped by the policer because its configured policy was violated.

Testing

See `tools/testing/selftests/drivers/net/netdevsim/devlink_trap.sh` for a test covering the core infrastructure. Test cases should be added for any new functionality.

Device drivers should focus their tests on device-specific functionality, such as the triggering of supported packet traps.

8.3.12 Devlink Line card

Background

The devlink-linecard mechanism is targeted for manipulation of line cards that serve as a detachable PHY modules for modular switch system. Following operations are provided:

- Get a list of supported line card types.
- Provision of a slot with specific line card type.
- Get and monitor of line card state and its change.

Line card according to the type may contain one or more gearboxes to mux the lanes with certain speed to multiple ports with lanes of different speed. Line card ensures N:M mapping between the switch ASIC modules and physical front panel ports.

Overview

Each line card devlink object is created by device driver, according to the physical line card slots available on the device.

Similar to splitter cable, where the device might have no way of detection of the splitter cable geometry, the device might not have a way to detect line card type. For that devices, concept of provisioning is introduced. It allows the user to:

- Provision a line card slot with certain line card type
 - Device driver would instruct the ASIC to prepare all resources accordingly. The device driver would create all instances, namely devlink port and netdevices that reside on the line card, according to the line card type
- Manipulate of line card entities even without line card being physically connected or powered-up
- Setup splitter cable on line card ports
 - As on the ordinary ports, user may provision a splitter cable of a certain type, without the need to be physically connected to the port
- Configure devlink ports and netdevices

Netdevice carrier is decided as follows:

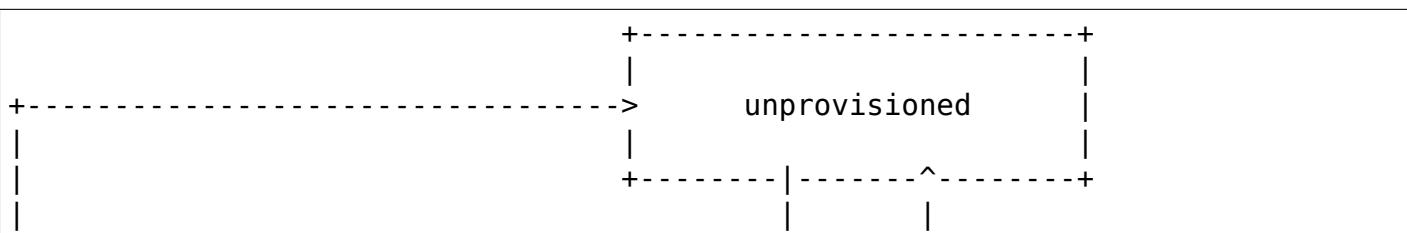
- Line card is not inserted or powered-down
 - The carrier is always down
- Line card is inserted and powered up
 - The carrier is decided as for ordinary port netdevice

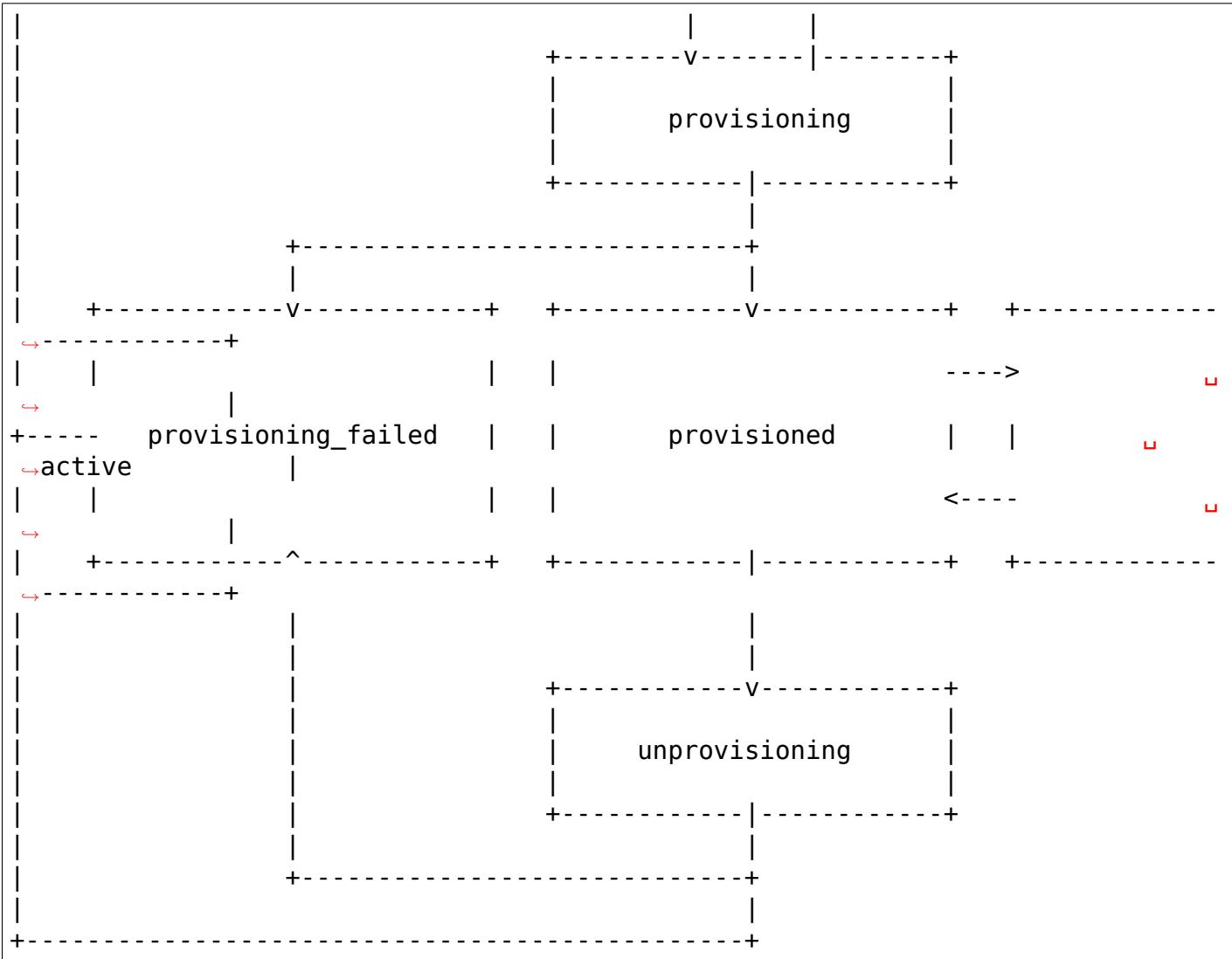
Line card state

The devlink-linecard mechanism supports the following line card states:

- **unprovisioned**: Line card is not provisioned on the slot.
- **unprovisioning**: Line card slot is currently being unprovisioned.
- **provisioning**: Line card slot is currently in a process of being provisioned with a line card type.
- **provisioning_failed**: Provisioning was not successful.
- **provisioned**: Line card slot is provisioned with a type.
- **active**: Line card is powered-up and active.

The following diagram provides a general overview of devlink-linecard state transitions:





Example usage

```

$ devlink lc show [ DEV [ lc LC_INDEX ] ]
$ devlink lc set DEV lc LC_INDEX [ { type LC_TYPE | notype } ]

# Show current line card configuration and status for all slots:
$ devlink lc

# Set slot 8 to be provisioned with type "16x100G":
$ devlink lc set pci/0000:01:00.0 lc 8 type 16x100G

# Set slot 8 to be unprovisioned:
$ devlink lc set pci/0000:01:00.0 lc 8 notype

```

8.4 Driver-specific documentation

Each driver that implements devlink is expected to document what parameters, info versions, and other features it supports.

8.4.1 bnxt devlink support

This document describes the devlink features implemented by the bnxt device driver.

Parameters

Table 16: Generic parameters implemented

Name	Mode
enable_sriov	Permanent
ignore_ari	Permanent
msix_vec_per_pf_max	Permanent
msix_vec_per_pf_min	Permanent
enable_remote_dev_reset	Runtime

The bnxt driver also implements the following driver-specific parameters.

Table 17: Driver-specific parameters implemented

Name	Type	Mode	Description
gre_version_check	Bool	Per Device	Generic Routing Encapsulation (GRE) version check will be enabled in the device. If disabled, the device will skip the version check for incoming packets.

Info versions

The bnxt_en driver reports the following versions

Table 18: devlink info versions implemented :widths: 5 5 90

Name	Type	Description
board.id	fixed	Part number identifying the board design
asic.id	fixed	ASIC design identifier
asic.rev	fixed	ASIC design revision
fw.psid	stored, running	Firmware parameter set version of the board
fw	stored, running	Overall board firmware version
fw.mgmt	stored, running	NIC hardware resource management firmware version
fw.mgmt.api	running	Minimum firmware interface spec version supported between driver and firmware
fw.nsci	stored, running	General platform management firmware version
fw.roce	stored, running	RoCE management firmware version

8.4.2 etas_es58x devlink support

This document describes the devlink features implemented by the `etas_es58x` device driver.

Info versions

The `etas_es58x` driver reports the following versions

Table 19: devlink info versions implemented

Name	Type	Description
fw	running	Version of the firmware running on the device. Also available through <code>ethtool -i</code> as the first member of the <code>firmware-version</code> .
fw.	running	Version of the bootloader running on the device. Also available through <code>ethtool -i</code> as the second member of the <code>firmware-version</code> .
board	fixed	The hardware revision of the device.
rev		
serial	fixed	The USB serial number. Also available through <code>lsusb -v</code> .

8.4.3 hns3 devlink support

This document describes the devlink features implemented by the `hns3` device driver.

The `hns3` driver supports reloading via `DEVLINK_CMD_RELOAD`.

Info versions

The `hns3` driver reports the following versions

Table 20: devlink info versions implemented

Name	Type	Description
fw	running	Used to represent the firmware version.

8.4.4 i40e devlink support

This document describes the devlink features implemented by the `i40e` device driver.

Info versions

The `i40e` driver reports the following versions

Table 21: devlink info versions implemented

Name	Type	Example	Description
board_id	fixed	K151900	Product Board Assembly (PBA) identifier of the board.
fw. mgmtning	run	9.130	10-digit version number of the management firmware that controls the PHY, link, etc.
fw. mgmtning api	run	1.152	2-digit version number of the API exported over the AdminQ by the management firmware. Used by the driver to identify what commands are supported.
fw. mgmtning build	run	7361	Build number of the source for the management firmware.
fw. undining	run	1.34	Version of the Option ROM containing the UEFI driver. The version is reported in major.minor.patch format. The major version is incremented whenever a major breaking change occurs, or when the minor version would overflow. The minor version is incremented for non-breaking changes and reset to 1 when the major version is incremented. The patch version is normally 0 but is incremented when a fix is delivered as a patch against an older base Option ROM.
fw. psidning api	run	9.30	Version defining the format of the flash contents.
fw. bundling_id	run	0x8000000000000000	Identifier of the firmware image file that was loaded onto the device. Also referred to as the EETRACK identifier of the NVM.

8.4.5 ionic devlink support

This document describes the devlink features implemented by the `ionic` device driver.

Info versions

The `ionic` driver reports the following versions

Table 22: devlink info versions implemented

Name	Type	Description
fw	run	Version of firmware running on the device
asic_id	fixed	The ASIC type for this device
asic_rev	fixed	The revision of the ASIC for this device

8.4.6 ice devlink support

This document describes the devlink features implemented by the `ice` device driver.

Parameters

Table 23: Generic parameters implemented

Name	Mode	Notes
<code>enable_roce</code>	runtime	mutually exclusive with <code>enable_iwarp</code>
<code>enable_iwarp</code>	runtime	mutually exclusive with <code>enable_roce</code>

Info versions

The `ice` driver reports the following versions

Table 24: devlink info versions implemented

Name	Type	Example	Description
board_id	fixed	K653900	Product Board Assembly (PBA) identifier of the board.
cgu_id	fixed	36	The Clock Generation Unit (CGU) hardware revision identifier.
fw_mgmtning	run	2.1	3-digit version number of the management firmware running on the Embedded Management Processor of the device. It controls the PHY, link, access to device resources, etc. Intel documentation refers to this as the EMP firmware.
fw_mgmtning	run	1.5	13-digit version number (major.minor.patch) of the API exported over the AdminQ by the management firmware. Used by the driver to identify what commands are supported. Historical versions of the kernel only displayed a 2-digit version number (major.minor).
fw_mgmtning	run	0x305d955f	identifier of the source for the management firmware.
fw_undining	run	1.25	Version of the Option ROM containing the UEFI driver. The version is reported in major.minor.patch format. The major version is incremented whenever a major breaking change occurs, or when the minor version would overflow. The minor version is incremented for non-breaking changes and reset to 1 when the major version is incremented. The patch version is normally 0 but is incremented when a fix is delivered as a patch against an older base Option ROM.
fw_psidning	run	0.80	Version defining the format of the flash contents.
fw_bundling	run	0x80002000	identifier of the firmware image file that was loaded onto the device. Also referred to as the EETRACK identifier of the NVM.
fw_app.name	run	ICE	The name of the DDP package that is active in the device. The DDP package is loaded by the driver during initialization. Each variation of the DDP package has a unique name.
fw_app.name	run	fault	
fw_app.name	run	Package	
fw_app.name	run	1.3	The version of the DDP package that is active in the device. Note that both the name (as reported by fw.app.name) and version are required to uniquely identify the package.
fw_app.name	run	0xc0000001	identifier for the DDP package loaded in the device. Also referred to as the DDP Track ID. Can be used to uniquely identify the specific DDP package.
fw_netlist	run	2000	version of the netlist module. This module defines the device's Ethernet capabilities and default settings, and is used by the management firmware as part of managing link and device connectivity.
fw_netlist	run	7	The first 4 bytes of the hash of the netlist module contents.
fw_cgu_type	run	803216973825160210	Clock Generation Unit (CGU). Format: <CGU type>.<configuration version>.<firmware version>.

Flash Update

The ice driver implements support for flash update using the devlink-flash interface. It supports updating the device flash using a combined flash image that contains the fw.mgmt, fw.undi, and fw.netlist components.

Table 25: List of supported overwrite modes

Bits	Behavior
DEVLINK_FLASH_OVERWRITE_SETTINGS	In the flash components being updated. This includes overwriting the port configuration that determines the number of physical functions the device will initialize with.
DEVLINK_FLASH_OVERWRITE_SETTINGS and DEVLINK_FLASH_OVERWRITE_IDENTIFIER	Overwrite everything in the flash with the contents from the provided image, without performing any preservation. This includes identifiers such as the MAC address, VPD area, and device serial number. It is expected that this combination be used with an image customized for the specific device.
DEVLINK_FLASH_OVERWRITE_IDENTIFIER	Identifiers such as the MAC address, VPD area, and device serial number. It is expected that this combination be used with an image customized for the specific device.

The ice hardware does not support overwriting only identifiers while preserving settings, and thus DEVLINK_FLASH_OVERWRITE_IDENTIFIER on its own will be rejected. If no overwrite mask is provided, the firmware will be instructed to preserve all settings and identifying fields when updating.

Reload

The ice driver supports activating new firmware after a flash update using DEVLINK_CMD_RELOAD with the DEVLINK_RELOAD_ACTION_FW_ACTIVATE action.

```
$ devlink dev reload pci/0000:01:00.0 reload action fw_activate
```

The new firmware is activated by issuing a device specific Embedded Management Processor reset which requests the device to reset and reload the EMP firmware image.

The driver does not currently support reloading the driver via DEVLINK_RELOAD_ACTION_DRIVER_REINIT.

Port split

The ice driver supports port splitting only for port 0, as the FW has a predefined set of available port split options for the whole device.

A system reboot is required for port split to be applied.

The following command will select the port split option with 4 ports:

```
$ devlink port split pci/0000:16:00.0/0 count 4
```

The list of all available port options will be printed to dynamic debug after each split and unsplit command. The first option is the default.

```
ice 0000:16:00.0: Available port split options and max port speeds (Gbps):
ice 0000:16:00.0: Status Split Quad 0 Quad 1
```

ice 0000:16:00.0:	count	L0	L1	L2	L3	L4	L5	L6	L7
ice 0000:16:00.0: Active	2	100	-	-	-	100	-	-	-
ice 0000:16:00.0:	2	50	-	50	-	-	-	-	-
ice 0000:16:00.0: Pending	4	25	25	25	25	-	-	-	-
ice 0000:16:00.0:	4	25	25	-	-	25	25	-	-
ice 0000:16:00.0:	8	10	10	10	10	10	10	10	10
ice 0000:16:00.0:	1	100	-	-	-	-	-	-	-

There could be multiple FW port options with the same port split count. When the same port split count request is issued again, the next FW port option with the same port split count will be selected.

`devlink port unsplit` will select the option with a split count of 1. If there is no FW option available with split count 1, you will receive an error.

Regions

The `ice` driver implements the following regions for accessing internal device data.

Table 26: regions implemented

Name	Description
<code>nvm-flash</code>	The contents of the entire flash chip, sometimes referred to as the device's Non Volatile Memory.
<code>shadow-ram</code>	The contents of the Shadow RAM, which is loaded from the beginning of the flash. Although the contents are primarily from the flash, this area also contains data generated during device boot which is not stored in flash.
<code>device-caps</code>	The contents of the device firmware's capabilities buffer. Useful to determine the current state and configuration of the device.

Both the `nvm-flash` and `shadow-ram` regions can be accessed without a snapshot. The `device-caps` region requires a snapshot as the contents are sent by firmware and can't be split into separate reads.

Users can request an immediate capture of a snapshot for all three regions via the `DEVLINK_CMD_REGION_NEW` command.

```
$ devlink region show
pci/0000:01:00.0/nvm-flash: size 10485760 snapshot [] max 1
pci/0000:01:00.0/device-caps: size 4096 snapshot [] max 10

$ devlink region new pci/0000:01:00.0/nvm-flash snapshot 1
$ devlink region dump pci/0000:01:00.0/nvm-flash snapshot 1

$ devlink region dump pci/0000:01:00.0/nvm-flash snapshot 1
0000000000000000 0014 95dc 0014 9514 0035 1670 0034 db30
0000000000000010 0000 0000 ffff ff04 0029 8c00 0028 8cc8
0000000000000020 0016 0bb8 0016 1720 0000 0000 c00f 3ffc
0000000000000030 bada cce5 bada cce5 bada cce5

$ devlink region read pci/0000:01:00.0/nvm-flash snapshot 1 address 0 length 16
```

```
0000000000000000 0014 95dc 0014 9514 0035 1670 0034 db30

$ devlink region delete pci/0000:01:00.0/nvm-flash snapshot 1

$ devlink region new pci/0000:01:00.0/device-caps snapshot 1
$ devlink region dump pci/0000:01:00.0/device-caps snapshot 1
0000000000000000 01 00 01 00 00 00 00 00 01 00 00 00 00 00 00 00 00
0000000000000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000000000000020 02 00 02 01 32 03 00 00 0a 00 00 00 25 00 00 00 00
0000000000000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000000000000040 04 00 01 00 01 00 00 00 00 00 00 00 00 00 00 00 00
0000000000000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000000000000060 05 00 01 00 03 00 00 00 00 00 00 00 00 00 00 00 00
0000000000000070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000000000000080 06 00 01 00 01 00 00 00 00 00 00 00 00 00 00 00 00
0000000000000090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000000000a0 08 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000000000b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000000000c0 12 00 01 00 01 00 00 00 01 00 01 00 00 00 00 00 00
00000000000000d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000000000e0 13 00 01 00 00 01 00 00 00 00 00 00 00 00 00 00 00
00000000000000f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000000000100 14 00 01 00 01 00 00 00 00 00 00 00 00 00 00 00 00
000000000000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000000000120 15 00 01 00 01 00 00 00 00 00 00 00 00 00 00 00 00
000000000000130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000000000140 16 00 01 00 01 00 00 00 00 00 00 00 00 00 00 00 00
000000000000150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000000000160 17 00 01 00 06 00 00 00 00 00 00 00 00 00 00 00 00
000000000000170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000000000180 18 00 01 00 01 00 00 00 01 00 00 00 08 00 00 00 00
000000000000190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000000000001a0 22 00 01 00 01 00 00 00 00 00 00 00 00 00 00 00 00
0000000000001b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000000000001c0 40 00 01 00 00 08 00 00 08 00 00 00 00 00 00 00 00
0000000000001d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000000000001e0 41 00 01 00 00 08 00 00 00 00 00 00 00 00 00 00 00
0000000000001f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000000000200 42 00 01 00 00 08 00 00 00 00 00 00 00 00 00 00 00
000000000000210 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```



```
$ devlink region delete pci/0000:01:00.0/device-caps snapshot 1
```

Devlink Rate

The `ice` driver implements devlink-rate API. It allows for offload of the Hierarchical QoS to the hardware. It enables user to group Virtual Functions in a tree structure and assign supported parameters: `tx_share`, `tx_max`, `tx_priority` and `tx_weight` to each node in a tree. So effectively user gains an ability to control how much bandwidth is allocated for each VF group. This is later enforced by the HW.

It is assumed that this feature is mutually exclusive with DCB performed in FW and ADQ, or any driver feature that would trigger changes in QoS, for example creation of the new traffic class. The driver will prevent DCB or ADQ configuration if user started making any changes to the nodes using devlink-rate API. To configure those features a driver reload is necessary. Correspondingly if ADQ or DCB will get configured the driver won't export hierarchy at all, or will remove the untouched hierarchy if those features are enabled after the hierarchy is exported, but before any changes are made.

This feature is also dependent on switchdev being enabled in the system. It's required because devlink-rate requires devlink-port objects to be present, and those objects are only created in switchdev mode.

If the driver is set to the switchdev mode, it will export internal hierarchy the moment VF's are created. Root of the tree is always represented by the `node_0`. This node can't be deleted by the user. Leaf nodes and nodes with children also can't be deleted.

Table 27: Attributes supported

Name	Description
<code>tx_max</code>	maximum bandwidth to be consumed by the tree Node. Rate Limit is an absolute number specifying a maximum amount of bytes a Node may consume during the course of one second. Rate limit guarantees that a link will not oversaturate the receiver on the remote end and also enforces an SLA between the subscriber and network provider.
<code>tx_share</code>	minimum bandwidth allocated to a tree node when it is not blocked. It specifies an absolute BW. While <code>tx_max</code> defines the maximum bandwidth the node may consume, the <code>tx_share</code> marks committed BW for the Node.
<code>tx_priority</code>	allows for usage of strict priority arbiter among siblings. This arbitration scheme attempts to schedule nodes based on their priority as long as the nodes remain within their bandwidth limit. Range 0-7. Nodes with priority 7 have the highest priority and are selected first, while nodes with priority 0 have the lowest priority. Nodes that have the same priority are treated equally.
<code>tx_weight</code>	allows for usage of Weighted Fair Queuing arbitration scheme among siblings. This arbitration scheme can be used simultaneously with the strict priority. Range 1-200. Only relative values matter for arbitration.

`tx_priority` and `tx_weight` can be used simultaneously. In that case nodes with the same priority form a WFQ subgroup in the sibling group and arbitration among them is based on assigned weights.

```
# enable switchdev
$ devlink dev eswitch set pci/0000:4b:00.0 mode switchdev

# at this point driver should export internal hierarchy
$ echo 2 > /sys/class/net/ens785np0/device/sriov_numvfs
```

```
$ devlink port function rate show
pci/0000:4b:00.0/node_25: type node parent node_24
pci/0000:4b:00.0/node_24: type node parent node_0
pci/0000:4b:00.0/node_32: type node parent node_31
pci/0000:4b:00.0/node_31: type node parent node_30
pci/0000:4b:00.0/node_30: type node parent node_16
pci/0000:4b:00.0/node_19: type node parent node_18
pci/0000:4b:00.0/node_18: type node parent node_17
pci/0000:4b:00.0/node_17: type node parent node_16
pci/0000:4b:00.0/node_14: type node parent node_5
pci/0000:4b:00.0/node_5: type node parent node_3
pci/0000:4b:00.0/node_13: type node parent node_4
pci/0000:4b:00.0/node_12: type node parent node_4
pci/0000:4b:00.0/node_11: type node parent node_4
pci/0000:4b:00.0/node_10: type node parent node_4
pci/0000:4b:00.0/node_9: type node parent node_4
pci/0000:4b:00.0/node_8: type node parent node_4
pci/0000:4b:00.0/node_7: type node parent node_4
pci/0000:4b:00.0/node_6: type node parent node_4
pci/0000:4b:00.0/node_4: type node parent node_3
pci/0000:4b:00.0/node_3: type node parent node_16
pci/0000:4b:00.0/node_16: type node parent node_15
pci/0000:4b:00.0/node_15: type node parent node_0
pci/0000:4b:00.0/node_2: type node parent node_1
pci/0000:4b:00.0/node_1: type node parent node_0
pci/0000:4b:00.0/node_0: type node
pci/0000:4b:00.0/1: type leaf parent node_25
pci/0000:4b:00.0/2: type leaf parent node_25

# let's create some custom node
$ devlink port function rate add pci/0000:4b:00.0/node_custom parent node_0

# second custom node
$ devlink port function rate add pci/0000:4b:00.0/node_custom_1 parent node_
custom

# reassign second VF to newly created branch
$ devlink port function rate set pci/0000:4b:00.0/2 parent node_custom_1

# assign tx_weight to the VF
$ devlink port function rate set pci/0000:4b:00.0/2 tx_weight 5

# assign tx_share to the VF
$ devlink port function rate set pci/0000:4b:00.0/2 tx_share 500Mbps
```

8.4.7 mlx4 devlink support

This document describes the devlink features implemented by the `mlx4` device driver.

Parameters

Table 28: Generic parameters implemented

Name	Mode
<code>internal_err_reset</code>	driverinit, runtime
<code>max_macs</code>	driverinit
<code>region_snapshot_enable</code>	driverinit, runtime

The `mlx4` driver also implements the following driver-specific parameters.

Table 29: Driver-specific parameters implemented

Name	Type	Mode	Description
<code>enable_64qes</code>	Bool	Driver	Enable 64 byte CQEs/EQEs, if the FW supports it.
<code>enable_4kuar</code>	Bool	Driver	Enable using the 4k UAR.

The `mlx4` driver supports reloading via `DEVLINK_CMD_RELOAD`

Regions

The `mlx4` driver supports dumping the firmware PCI crspace and health buffer during a critical firmware issue.

In case a firmware command times out, firmware getting stuck, or a non zero value on the catastrophic buffer, a snapshot will be taken by the driver.

The `cr-space` region will contain the firmware PCI crspace contents. The `fw-health` region will contain the device firmware's health buffer. Snapshots for both of these regions are taken on the same event triggers.

8.4.8 mlx5 devlink support

This document describes the devlink features implemented by the `mlx5` device driver.

Parameters

Table 30: Generic parameters implemented

Name	Mode	Validation
enable_roce	driverinit	Type: Boolean If the device supports RoCE disablement, RoCE enablement state controls device support for RoCE capability. Otherwise, the control occurs in the driver stack. When RoCE is disabled at the driver level, only raw ethernet QPs are supported.
io_eq_size	driverinit	The range is between 64 and 4096.
event_eq_size	driverinit	The range is between 64 and 4096.
max_macs	driverinit	The range is between 1 and 2^{31} . Only power of 2 values are supported.

The mlx5 driver also implements the following driver-specific parameters.

Table 31: Driver-specific parameters implemented

Name	Type	Mode	Description
flows_steering_time	steering	control	<p>Controls the flow steering mode of the driver</p> <ul style="list-style-type: none"> • dmfs Device managed flow steering. In DMFS mode, the HW steering entities are created and managed through firmware. • smfs Software managed flow steering. In SMFS mode, the HW steering entities are created and manage through the driver without firmware intervention. <p>SMFS mode is faster and provides better rule insertion rate compared to default DMFS mode.</p>
fdb_lag_driver_time	lag	control	<p>Control the number of large groups (size > 1) in the FDB table.</p> <ul style="list-style-type: none"> • The default value is 15, and the range is between 1 and 1024.
esw_Boot_time	Booting	control	<p>Control MultiPort E-Switch shared fdb mode.</p> <p>An experimental mode where a single E-Switch is used and all the vports and physical ports on the NIC are connected to it.</p> <p>An example is to send traffic from a VF that is created on PF0 to an uplink that is natively associated with the uplink of PF1</p> <p>Note: Future devices, ConnectX-8 and onward, will eventually have this as the default to allow forwarding between all NIC ports in a single E-switch environment and the dual E-switch mode will likely get deprecated.</p> <p>Default: disabled</p>
esw_Boot_metadata_time	Booting	Metadata	<p>When applicable, disabling eswitch metadata can increase packet rate up to 20% depending on the use case and packet sizes.</p> <p>Eswitch port metadata state controls whether to internally tag packets with metadata. Metadata tagging must be enabled for multi-port RoCE, failover between representors and stacked devices. By default metadata is enabled on the supported devices in E-switch. Metadata is applicable only for E-switch in switchdev mode and users may disable it when NONE of the below use cases will be in use: 1. HCA is in Dual/multi-port RoCE mode. 2. VF/SF representor bonding (Usually used for Live migration) 3. Stacked devices</p> <p>When metadata is disabled, the above use cases will fail to initialize if users try to enable them.</p>
hairpin_driver_time	Driver	Writers	<p>Refers to a TC NIC rule that involves forwarding as "hairpin". Hairpin queues are mlx5 hardware specific implementation for hardware forwarding of such packets.</p> <p>Control the number of hairpin queues.</p>
hairpin_driver_size	Driver	Control	Control the size (in packets) of the hairpin queues.

The mlx5 driver supports reloading via DEVLINK_CMD_RELOAD

Info versions

The mlx5 driver reports the following versions

Table 32: devlink info versions implemented

Name	Type	Description
fw_psid	fixed	Used to represent the board id of the device.
fw_version	store	Three digit major.minor.subminor firmware version number.

Health reporters

tx reporter

The tx reporter is responsible for reporting and recovering of the following three error scenarios:

- **tx timeout**
Report on kernel tx timeout detection. Recover by searching lost interrupts.
- **tx error completion**
Report on error tx completion. Recover by flushing the tx queue and reset it.
- **tx PTP port timestamping CQ unhealthy**
Report too many CQEs never delivered on port ts CQ. Recover by flushing and re-creating all PTP channels.

tx reporter also support on demand diagnose callback, on which it provides real time information of its send queues status.

User commands examples:

- Diagnose send queues status:

```
$ devlink health diagnose pci/0000:82:00.0 reporter tx
```

Note: This command has valid output only when interface is up, otherwise the command has empty output.

- Show number of tx errors indicated, number of recover flows ended successfully, is autorecover enabled and graceful period from last recover:

```
$ devlink health show pci/0000:82:00.0 reporter tx
```

rx reporter

The rx reporter is responsible for reporting and recovering of the following two error scenarios:

- **rx queues' initialization (population) timeout**

Population of rx queues' descriptors on ring initialization is done in napi context via triggering an irq. In case of a failure to get the minimum amount of descriptors, a timeout would occur, and descriptors could be recovered by polling the EQ (Event Queue).

- **rx completions with errors (reported by HW on interrupt context)**

Report on rx completion error. Recover (if needed) by flushing the related queue and reset it.

rx reporter also supports on demand diagnose callback, on which it provides real time information of its receive queues' status.

- Diagnose rx queues' status and corresponding completion queue:

```
$ devlink health diagnose pci/0000:82:00.0 reporter rx
```

Note: This command has valid output only when interface is up. Otherwise, the command has empty output.

- Show number of rx errors indicated, number of recover flows ended successfully, is autorecover enabled, and graceful period from last recover:

```
$ devlink health show pci/0000:82:00.0 reporter rx
```

fw reporter

The fw reporter implements *diagnose* and *dump* callbacks. It follows symptoms of fw error such as fw syndrome by triggering fw core dump and storing it into the dump buffer. The fw reporter diagnose command can be triggered any time by the user to check current fw status.

User commands examples:

- Check fw heath status:

```
$ devlink health diagnose pci/0000:82:00.0 reporter fw
```

- Read FW core dump if already stored or trigger new one:

```
$ devlink health dump show pci/0000:82:00.0 reporter fw
```

Note: This command can run only on the PF which has fw tracer ownership, running it on other PF or any VF will return "Operation not permitted".

fw fatal reporter

The fw fatal reporter implements *dump* and *recover* callbacks. It follows fatal errors indications by CR-space dump and recover flow. The CR-space dump uses vsc interface which is valid even if the FW command interface is not functional, which is the case in most FW fatal errors. The recover function runs recover flow which reloads the driver and triggers fw reset if needed. On firmware error, the health buffer is dumped into the dmesg. The log level is derived from the error's severity (given in health buffer).

User commands examples:

- Run fw recover flow manually:

```
$ devlink health recover pci/0000:82:00.0 reporter fw_fatal
```

- Read FW CR-space dump if already stored or trigger new one:

```
$ devlink health dump show pci/0000:82:00.1 reporter fw_fatal
```

Note: This command can run only on PF.

vnic reporter

The vnic reporter implements only the *diagnose* callback. It is responsible for querying the vnic diagnostic counters from fw and displaying them in realtime.

Description of the vnic counters:

- **total_q_under_processor_handle**
number of queues in an error state due to an async error or errored command.
- **send_queue_priority_update_flow**
number of QP/SQ priority/SL update events.
- **cq_overrun**
number of times CQ entered an error state due to an overflow.
- **async_eq_overrun**
number of times an EQ mapped to async events was overrun. comp_eq_overrun number of times an EQ mapped to completion events was overrun.
- **quota_exceeded_command**
number of commands issued and failed due to quota exceeded.
- **invalid_command**
number of commands issued and failed due to any reason other than quota exceeded.
- **nic_receive_steering_discard**
number of packets that completed RX flow steering but were discarded due to a mismatch in flow table.
- **generated_pkt_steering_fail**
number of packets generated by the VNIC experiencing unexpected steering failure (at any point in steering flow).

- **handled_pkt_steering_fail**

number of packets handled by the VNIC experiencing unexpected steering failure (at any point in steering flow owned by the VNIC, including the FDB for the eswitch owner).

User commands examples:

- Diagnose PF/VF vnic counters:

```
$ devlink health diagnose pci/0000:82:00.1 reporter vnic
```

- Diagnose representor vnic counters (performed by supplying devlink port of the representor, which can be obtained via devlink port command):

```
$ devlink health diagnose pci/0000:82:00.1/65537 reporter vnic
```

Note: This command can run over all interfaces such as PF/VF and representor ports.

8.4.9 mlxsw devlink support

This document describes the devlink features implemented by the `mlxsw` device driver.

Parameters

Table 33: Generic parameters implemented

Name	Mode
<code>fw_load_policy</code>	driverinit

The `mlxsw` driver also implements the following driver-specific parameters.

Table 34: Driver-specific parameters implemented

Name	Type	Mode	Description
<code>acl_region_rehash_interval</code>	time	driverinit	for periodic ACL region rehashes. The value is specified in time milliseconds, with a minimum of 3000. The value of 0 disables periodic work entirely. The first rehash will be run immediately after the value is set.

The `mlxsw` driver supports reloading via `DEVLINK_CMD_RELOAD`

Info versions

The `mlxsw` driver reports the following versions

Table 35: devlink info versions implemented

Name	Type	Description
hw. revision	fixed	The hardware revision for this board
fw. psid	fixed	Firmware PSID
fw. version	run-time	Three digit firmware version

Line card auxiliary device info versions

The `mlxsw` driver reports the following versions for line card auxiliary device

Table 36: devlink info versions implemented

Name	Type	Description
hw. revision	fixed	The hardware revision for this line card
ini. version	run-time	Version of line cardINI loaded
fw. psid	fixed	Line card device PSID
fw. version	run-time	Three digit firmware version of line card device

Driver-specific Traps

Table 37: List of Driver-specific Traps Registered by `mlxsw`

Name	Type	Description
irif ddrop trap	packet	packets that the device decided to drop because they need to be routed from a disabled router interface (RIF). This can happen during RIF dismantle, when the RIF is first disabled before being removed completely
erif ddrop trap	packet	packets that the device decided to drop because they need to be routed through a disabled router interface (RIF). This can happen during RIF dismantle, when the RIF is first disabled before being removed completely

8.4.10 mv88e6xxx devlink support

This document describes the devlink features implemented by the `mv88e6xxx` device driver.

Parameters

The `mv88e6xxx` driver implements the following driver-specific parameters.

Table 38: Driver-specific parameters implemented

Name	Type	Mode	Description
<code>ATU_b8shrun</code>	Boolean	Driverinit	Select one of four possible hashing algorithms for MAC addresses in the Address Translation Unit. A value of 3 may work better than the default of 1 when many MAC addresses have the same OUI. Only the values 0 to 3 are valid for this parameter.

8.4.11 netdevsim devlink support

This document describes the devlink features supported by the `netdevsim` device driver.

Parameters

Table 39: Generic parameters implemented

Name	Mode
<code>max_macs</code>	Driverinit

The `netdevsim` driver also implements the following driver-specific parameters.

Table 40: Driver-specific parameters implemented

Name	Type	Mode	Description
<code>test</code>	Boolean	Driverinit	Test parameter used to show how a driver-specific devlink parameter can be implemented.

The `netdevsim` driver supports reloading via `DEVLINK_CMD_RELOAD`

Regions

The `netdevsim` driver exposes a dummy region as an example of how the devlink-region interfaces work. A snapshot is taken whenever the `take_snapshot` debugfs file is written to.

Resources

The `netdevsim` driver exposes resources to control the number of FIB entries, FIB rule entries and nexthops that the driver will allow.

```
$ devlink resource set netdevsim/netdevsim0 path /IPv4/fib size 96
$ devlink resource set netdevsim/netdevsim0 path /IPv4/fib-rules size 16
$ devlink resource set netdevsim/netdevsim0 path /IPv6/fib size 64
$ devlink resource set netdevsim/netdevsim0 path /IPv6/fib-rules size 16
$ devlink resource set netdevsim/netdevsim0 path /nexthops size 16
$ devlink dev reload netdevsim/netdevsim0
```

Rate objects

The `netdevsim` driver supports rate objects management, which includes:

- registering/unregistering leaf rate objects per VF devlink port;
- creation/deletion node rate objects;
- setting tx_share and tx_max rate values for any rate object type;
- setting parent node for any rate object type.

Rate nodes and their parameters are exposed in `netdevsim` debugfs in RO mode. For example created rate node with name `some_group`:

```
$ ls /sys/kernel/debug/netdevsim/netdevsim0/rate_groups/some_group
rate_parent tx_max tx_share
```

Same parameters are exposed for leaf objects in corresponding ports directories. For ex.:

```
$ ls /sys/kernel/debug/netdevsim/netdevsim0/ports/1
dev ethtool rate_parent tx_max tx_share
```

Driver-specific Traps

Table 41: List of Driver-specific Traps Registered by `netdevsim`

Name	Type	Description
<code>fid_miss</code>	When a packet enters the device it is classified to a filtering identifier (FID) based on the ingress port and VLAN. This trap is used to trap packets for which a FID could not be found	

8.4.12 nfp devlink support

This document describes the devlink features implemented by the nfp device driver.

Parameters

Table 42: Generic parameters implemented

Name	Mode
fw_load_policy	permanent
reset_dev_on_drv_probe	permanent

Info versions

The nfp driver reports the following versions

Table 43: devlink info versions implemented

Name	Type	Description
board_id	fixed	Part number identifying the board design
board_rev	fixed	Revision of the board design
board_manufacturer	fixed	Vendor of the board design
board_model	fixed	Model name of the board design
fw_bundle_id	stored	Firmware bundle id
fw_mgmtrunning	running	Version of the management firmware
fw_cpldrunning	running	The CPLD firmware component version
fw_apprunning	running	The APP firmware component version
fw_undirunning	running	The UNDI firmware component version
fw_ncsirunning	running	The NSCI firmware component version
chipinitrunning	running	The CFGR firmware component version

8.4.13 qed devlink support

This document describes the devlink features implemented by the `qed` core device driver.

Parameters

The `qed` driver implements the following driver-specific parameters.

Table 44: Driver-specific parameters implemented

Name	Type	Mode	Description
<code>iwarp</code>	Boolean		- Enable iWARP functionality for 100g devices. Note that this impacts L2 performance, and is therefore not enabled by default.

8.4.14 ti-cpsw-switch devlink support

This document describes the devlink features implemented by the `ti-cpsw-switch` device driver.

Parameters

The `ti-cpsw-switch` driver implements the following driver-specific parameters.

Table 45: Driver-specific parameters implemented

Name	Type	Mode	Description
<code>ale_Bypass</code>	Boolean		- Enables ALE_CONTROL(4).BYPASS mode for debugging purposes. In this time mode, all packets will be sent to the host port only.
<code>switch</code>	Boolean		- Enable switch mode

8.4.15 am65-cpsw-nuss devlink support

This document describes the devlink features implemented by the `am65-cpsw-nuss` device driver.

Parameters

The `am65-cpsw-nuss` driver implements the following driver-specific parameters.

Table 46: Driver-specific parameters implemented

Name	Type	Mode	Description
<code>switch</code>	Boolean		- Enable switch mode

8.4.16 prestera devlink support

This document describes the devlink features implemented by the `prestera` device driver.

Driver-specific Traps

Table 47: List of Driver-specific Traps Registered by `prestera`

Name	Type	Description
------	------	-------------

Table 48: List of Driver-specific Traps Registered by `prestera`

Name	Type	Description
<code>arp_bcap</code>	Trap	Traps ARP broadcast packets (both requests/responses)
<code>is_istrap</code>	Trap	Traps IS-IS packets
<code>ospftrap</code>	Trap	Traps OSPF packets
<code>ip_btrap</code>	Trap	Traps IPv4 packets with broadcast DA Mac address
<code>stp_trap</code>	Trap	Traps STP BPDU
<code>lacptrap</code>	Trap	Traps LACP packets
<code>lldptrap</code>	Trap	Traps LLDP packets
<code>routertrap</code>	Trap	Traps multicast packets
<code>vrrptrap</code>	Trap	Traps VRRP packets
<code>dhcptrap</code>	Trap	Traps DHCP packets
<code>mtuertrap</code>	Trap	Traps (exception) packets that exceeded port's MTU
<code>mac_toapertrap</code>	Trap	Traps packets with switch-port's DA Mac address
<code>ttl_ertrap</code>	Trap	Traps (exception) IPv4 packets whose TTL exceeded
<code>ipv4toptrap</code>	Trap	Traps (exception) packets due to the malformed IPV4 header options
<code>ip_defapTrap</code>	Trap	packets that have no specific IP interface (IP to me) and no forwarding prefix
<code>locatrap</code>	Trap	Traps packets that have been send to one of switch IP interfaces addresses
<code>ipv4trapptrap</code>	Trap	(exception) IPV4 ICMP redirect packets
<code>arp_teapTrap</code>	Trap	ARP replies packets that have switch-port's DA Mac address
<code>acl_t0epT0aps</code>	Trap	packets that have ACL priority set to 0 (tc pref 0)
<code>acl_t0epT1aps</code>	Trap	packets that have ACL priority set to 1 (tc pref 1)
<code>acl_t0epT2aps</code>	Trap	packets that have ACL priority set to 2 (tc pref 2)
<code>acl_t0epT3aps</code>	Trap	packets that have ACL priority set to 3 (tc pref 3)
<code>acl_t0epT4aps</code>	Trap	packets that have ACL priority set to 4 (tc pref 4)
<code>acl_t0epT5aps</code>	Trap	packets that have ACL priority set to 5 (tc pref 5)
<code>acl_t0epT6aps</code>	Trap	packets that have ACL priority set to 6 (tc pref 6)
<code>acl_t0epT7aps</code>	Trap	packets that have ACL priority set to 7 (tc pref 7)
<code>ipv4tbgp</code>	Trap	Traps IPv4 BGP packets
<code>ssh_trap</code>	Trap	Traps SSH packets
<code>telnettrap</code>	Trap	Traps Telnet packets
<code>icmptrap</code>	Trap	Traps ICMP packets
<code>rxdmddrop</code>	Drop	Drops packets (RxDMA) due to the lack of ingress buffers etc.
<code>portdndrop</code>	Drop	Drops packets due to faulty-configured network or due to internal bug (config issue).

continues on next page

Table 48 – continued from previous page

<code>localaddrdrop</code>	Drops packets whose decision (FDB entry) is to bridge packet back to the incoming port/trunk.
<code>invalidaddrdrop</code>	Drops packets with multicast source MAC address.
<code>illegaladdrdrop</code>	Drops packets with illegal SIP/DIP multicast/unicast addresses.
<code>illegalipdrop</code>	Drops packets with illegal IPV4 header.
<code>ip_udrddrop</code>	Drops packets with destination MAC being unicast, but destination IP address being multicast.
<code>ip_sdrop</code>	Drops packets with zero (0) IPV4 source address.
<code>met_deddrop</code>	Drops non-conforming packets (dropped by Ingress policer, metering drop), e.g. packet rate exceeded configured bandwidth.

8.4.17 iosm devlink support

This document describes the devlink features implemented by the `iosm` device driver.

Parameters

The `iosm` driver implements the following driver-specific parameters.

Table 49: Driver-specific parameters implemented

Name	Type	Mode	Description
<code>erase8fullinfase</code>			<code>erase_full_flash</code> parameter is used to check if full erase is required for the device during firmware flashing. If set, Full nand erase command will be sent to the device. By default, only conditional erase support is enabled.

Flash Update

The `iosm` driver implements support for flash update using the `devlink-flash` interface.

It supports updating the device flash using a combined flash image which contains the Bootloader images and other modem software images.

The driver uses `DEVLINK_SUPPORT_FLASH_UPDATE_COMPONENT` to identify type of firmware image that need to be flashed as requested by user space application. Supported firmware image types.

Table 50: Firmware Image types

Name	Description
<code>PSI RAM</code>	Primary Signed Image
<code>EBL</code>	External Bootloader
<code>FLS</code>	Modem Software Image

`PSI RAM` and `EBL` are the RAM images which are injected to the device when the device is in BOOT ROM stage. Once this is successful, the actual modem firmware image is flashed to the device. The modem software image contains multiple files each having one secure bin file and at least one Loadmap/Region file. For flashing these files, appropriate commands are sent to

the modem device along with the data required for flashing. The data like region count and address of each region has to be passed to the driver using the devlink param command.

If the device has to be fully erased before firmware flashing, user application need to set the `erase_full_flash` parameter using devlink param command. By default, conditional erase feature is supported.

Flash Commands:

1) When modem is in Boot ROM stage, user can use below command to inject PSI RAM image using devlink flash command.

```
$ devlink dev flash pci/0000:02:00.0 file <PSI_RAM_File_name>
```

2) If user want to do a full erase, below command need to be issued to set the `erase_full_flash` param (To be set only if full erase required).

```
$ devlink dev param set pci/0000:02:00.0 name erase_full_flash value true cmode runtime
```

3) Inject EBL after the modem is in PSI stage.

```
$ devlink dev flash pci/0000:02:00.0 file <EBL_File_name>
```

4) Once EBL is injected successfully, then the actual firmware flashing takes place. Below is the sequence of commands used for each of the firmware images.

a) Flash secure bin file.

```
$ devlink dev flash pci/0000:02:00.0 file <Secure_bin_file_name>
```

b) Flashing the Loadmap/Region file

```
$ devlink dev flash pci/0000:02:00.0 file <Load_map_file_name>
```

Regions

The `iosm` driver supports dumping the coredump logs.

In case a firmware encounters an exception, a snapshot will be taken by the driver. Following regions are accessed for device internal data.

Table 51: Regions implemented

Name	Description
<code>report.json</code>	The summary of exception details logged as part of this region.
<code>coredump.fcd</code>	This region contains the details related to the exception occurred in the device (RAM dump).
<code>cdd.log</code>	This region contains the logs related to the modem CDD driver.
<code>eeprom.bin</code>	This region contains the eeprom logs.
<code>bootcore_trace.bin</code>	This region contains the current instance of bootloader logs.
<code>bootcore_prev_trace.bin</code>	This region contains the previous instance of bootloader logs.

Region commands

```
$ devlink region show
$ devlink region new pci/0000:02:00.0/report.json
$ devlink region dump pci/0000:02:00.0/report.json snapshot 0
$ devlink region del pci/0000:02:00.0/report.json snapshot 0
$ devlink region new pci/0000:02:00.0/coredump.fcd
$ devlink region dump pci/0000:02:00.0/coredump.fcd snapshot 1
$ devlink region del pci/0000:02:00.0/coredump.fcd snapshot 1
$ devlink region new pci/0000:02:00.0/cdd.log
$ devlink region dump pci/0000:02:00.0/cdd.log snapshot 2
$ devlink region del pci/0000:02:00.0/cdd.log snapshot 2
$ devlink region new pci/0000:02:00.0/eeprom.bin
$ devlink region dump pci/0000:02:00.0/eeprom.bin snapshot 3
$ devlink region del pci/0000:02:00.0/eeprom.bin snapshot 3
$ devlink region new pci/0000:02:00.0/bootcore_trace.bin
$ devlink region dump pci/0000:02:00.0/bootcore_trace.bin snapshot 4
$ devlink region del pci/0000:02:00.0/bootcore_trace.bin snapshot 4
$ devlink region new pci/0000:02:00.0/bootcore_prev_trace.bin
$ devlink region dump pci/0000:02:00.0/bootcore_prev_trace.bin snapshot 5
$ devlink region del pci/0000:02:00.0/bootcore_prev_trace.bin snapshot 5
```

8.4.18 octeontx2 devlink support

This document describes the devlink features implemented by the octeontx2 AF, PF and VF device drivers.

Parameters

The octeontx2 PF and VF drivers implement the following driver-specific parameters.

Table 52: Driver-specific parameters implemented

Name	Type	Mode	Description
mcamu6untn	-	-	Select number of match CAM entries to be allocated for an interface. The same time is used for ntuple filters of the interface. Supported by PF and VF drivers.

The octeontx2 AF driver implements the following driver-specific parameters.

Table 53: Driver-specific parameters implemented

Name	Type	Mode	Description
dwr	run	run	Use to set the quantum which hardware uses for scheduling among transmit queues. Hardware uses weighted DWRR algorithm to schedule among all transmit queues.

8.4.19 sfc devlink support

This document describes the devlink features implemented by the `sfc` device driver for the `ef100` device.

Info versions

The `sfc` driver reports the following versions

Table 54: devlink info versions implemented

Name	Type	Description
fw.	run	For boards where the management function is split between multiple control units, this is the SUC control unit's firmware version.
mgmt	run	For boards where the management function is split between multiple control units, this is the CMC control unit's firmware version.
suc	run	FPGA design revision.
fw.	rev	Datapath programmable logic version.
app	ning	Datapath software/microcode/firmware version.
fw.	app	SmartNIC application co-processor (APU) first stage boot loader version.
boot	ning	SmartNIC application co-processor (APU) co-operating system loader version.
coproc	boot	SmartNIC application co-processor (APU) main operating system version.
main	ning	SmartNIC application co-processor (APU) recovery operating system version.
coproc	recovery	Expansion ROM version. For boards where the expansion ROM is split between multiple images (e.g. PXE and UEFI), this is the specifically the PXE boot ROM version.
fw.	uefi	UEFI driver version (No UNDI support).
uefi	ning	

Contents:

9.1 Linux CAIF

Copyright © ST-Ericsson AB 2010

Author

Sjur Brendeland/ sjur.brandeland@stericsson.com

License terms

GNU General Public License (GPL) version 2

9.1.1 Introduction

CAIF is a MUX protocol used by ST-Ericsson cellular modems for communication between Modem and host. The host processes can open virtual AT channels, initiate GPRS Data connections, Video channels and Utility Channels. The Utility Channels are general purpose pipes between modem and host.

ST-Ericsson modems support a number of transports between modem and host. Currently, UART and Loopback are available for Linux.

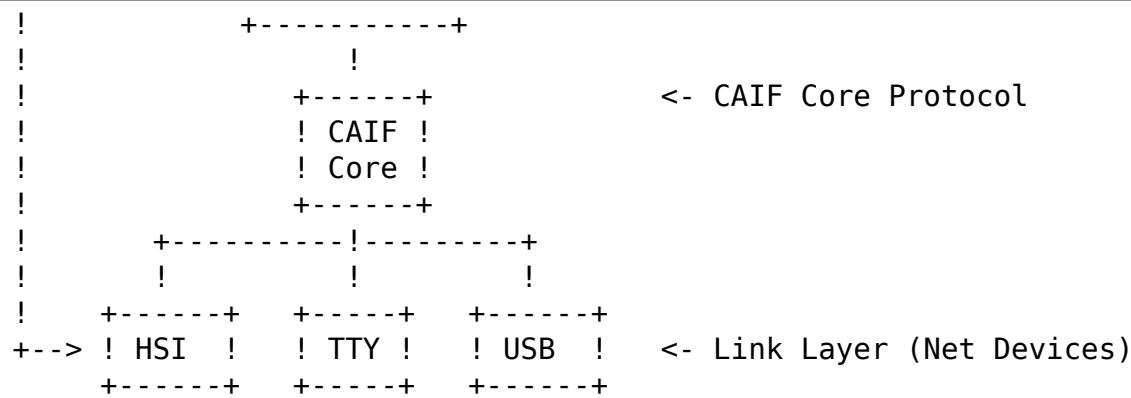
9.1.2 Architecture

The implementation of CAIF is divided into:

- CAIF Socket Layer and GPRS IP Interface.
- CAIF Core Protocol Implementation
- CAIF Link Layer, implemented as NET devices.

RTNL

```
!
!      +----+  +----+
!      +----+!  +----+!
!      !  IP  !!  !Socket!!
+----> !interf!+  ! API  !+      <- CAIF Client APIs
!      +----+  +----+!
!          !          !
```



9.1.3 Implementation

CAIF Core Protocol Layer

CAIF Core layer implements the CAIF protocol as defined by ST-Ericsson. It implements the CAIF protocol stack in a layered approach, where each layer described in the specification is implemented as a separate layer. The architecture is inspired by the design patterns "Protocol Layer" and "Protocol Packet".

CAIF structure

The Core CAIF implementation contains:

- Simple implementation of CAIF.
- Layered architecture (a la Streams), each layer in the CAIF specification is implemented in a separate c-file.
- Clients must call configuration function to add PHY layer.
- Clients must implement CAIF layer to consume/produce CAIF payload with receive and transmit functions.
- Clients must call configuration function to add and connect the Client layer.
- When receiving / transmitting CAIF Packets (cfpkt), ownership is passed to the called function (except for framing layers' receive function)

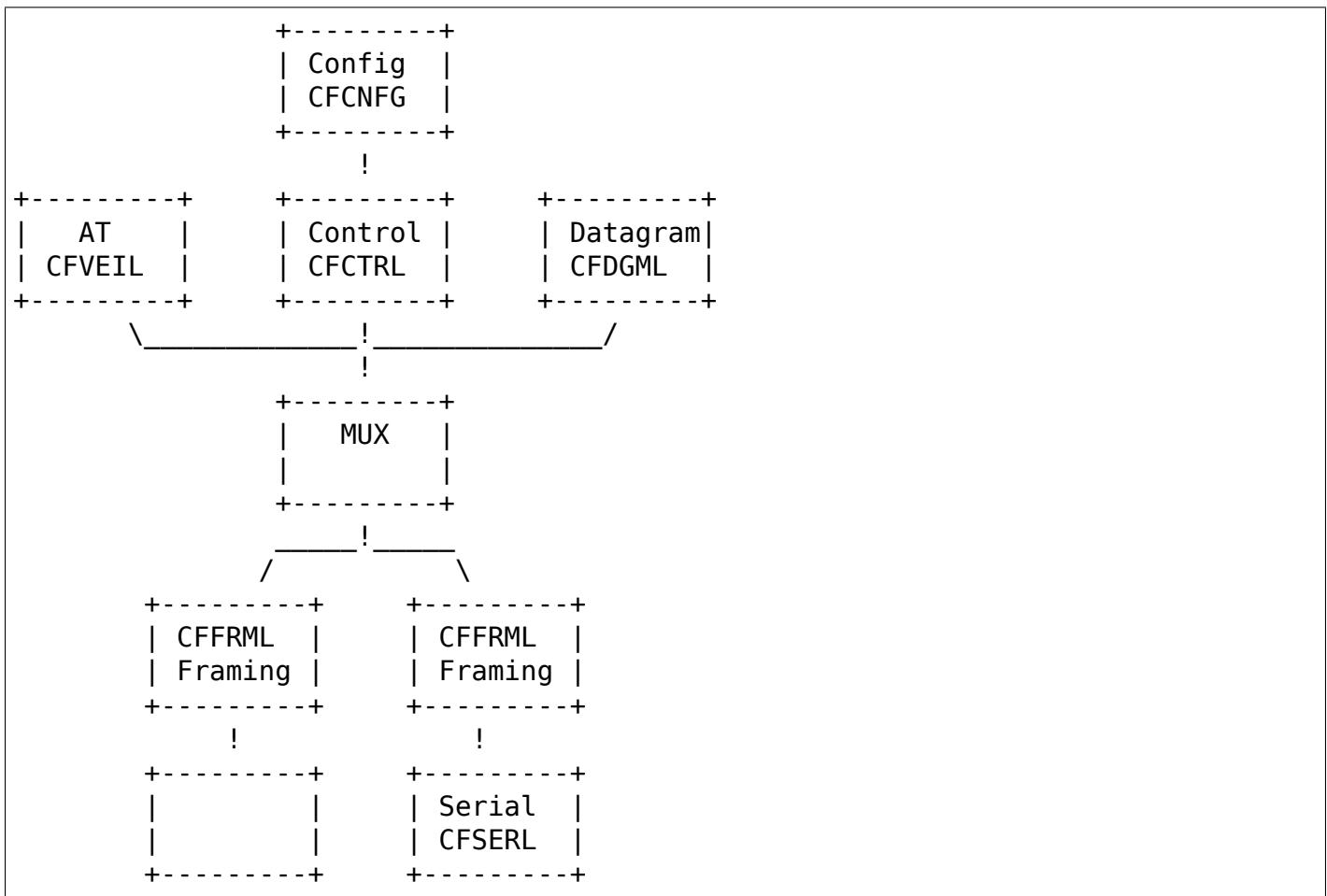
9.1.4 Layered Architecture

The CAIF protocol can be divided into two parts: Support functions and Protocol Implementation. The support functions include:

- CFPKT CAIF Packet. Implementation of CAIF Protocol Packet. The CAIF Packet has functions for creating, destroying and adding content and for adding/extracting header and trailers to protocol packets.

The CAIF Protocol implementation contains:

- CFCNFG CAIF Configuration layer. Configures the CAIF Protocol Stack and provides a Client interface for adding Link-Layer and Driver interfaces on top of the CAIF Stack.
- CFCTRL CAIF Control layer. Encodes and Decodes control messages such as enumeration and channel setup. Also matches request and response messages.
- CFSERVL General CAIF Service Layer functionality; handles flow control and remote shutdown requests.
- CFVEI CAIF VEI layer. Handles CAIF AT Channels on VEI (Virtual External Interface). This layer encodes/decodes VEI frames.
- CFDGML CAIF Datagram layer. Handles CAIF Datagram layer (IP traffic), encodes/decodes Datagram frames.
- CFMUX CAIF Mux layer. Handles multiplexing between multiple physical bearers and multiple channels such as VEI, Datagram, etc. The MUX keeps track of the existing CAIF Channels and Physical Instances and selects the appropriate instance based on Channel-Id and Physical-ID.
- CFFRML CAIF Framing layer. Handles Framing i.e. Frame length and frame checksum.
- CFSERL CAIF Serial layer. Handles concatenation/split of frames into CAIF Frames with correct length.



In this layered approach the following "rules" apply.

- All layers embed the same structure "struct cflayer"

- A layer does not depend on any other layer's private data.
- Layers are stacked by setting the pointers:

```
layer->up , layer->dn
```

- In order to send data upwards, each layer should do:

```
layer->up->receive(layer->up, packet);
```

- In order to send data downwards, each layer should do:

```
layer->dn->transmit(layer->dn, packet);
```

9.1.5 CAIF Socket and IP interface

The IP interface and CAIF socket API are implemented on top of the CAIF Core protocol. The IP Interface and CAIF socket have an instance of 'struct cflayer', just like the CAIF Core protocol stack. Net device and Socket implement the 'receive()' function defined by 'struct cflayer', just like the rest of the CAIF stack. In this way, transmit and receive of packets is handled as by the rest of the layers: the 'dn->transmit()' function is called in order to transmit data.

Configuration of Link Layer

The Link Layer is implemented as Linux network devices (*struct net_device*). Payload handling and registration is done using standard Linux mechanisms.

The CAIF Protocol relies on a loss-less link layer without implementing retransmission. This implies that packet drops must not happen. Therefore a flow-control mechanism is implemented where the physical interface can initiate flow stop for all CAIF Channels.

9.2 Using Linux CAIF

Copyright

© ST-Ericsson AB 2010

Author

Sjur Brendeland/ sjur.brandeland@stericsson.com

9.2.1 Start

If you have compiled CAIF for modules do:

```
$modprobe crc_ccitt  
$modprobe caif  
$modprobe caif_socket  
$modprobe chnl_net
```

9.2.2 Preparing the setup with a STE modem

If you are working on integration of CAIF you should make sure that the kernel is built with module support.

There are some things that need to be tweaked to get the host TTY correctly set up to talk to the modem. Since the CAIF stack is running in the kernel and we want to use the existing TTY, we are installing our physical serial driver as a line discipline above the TTY device.

To achieve this we need to install the N_CAIF ldisc from user space. The benefit is that we can hook up to any TTY.

The use of Start-of-frame-extension (STX) must also be set as module parameter "ser_use_stx".

Normally Frame Checksum is always used on UART, but this is also provided as a module parameter "ser_use_fcs".

```
$ modprobe caif_serial ser_ttyname=/dev/ttyS0 ser_use_stx=yes
$ ifconfig caif_ttyS0 up
```

PLEASE NOTE:

There is a limitation in Android shell. It only accepts one argument to insmod/modprobe!

9.2.3 Trouble shooting

There are debugfs parameters provided for serial communication.
/sys/kernel/debug/caif_serial/<tty-name>/

- ser_state: Prints the bit-mask status where
 - 0x02 means SENDING, this is a transient state.
 - 0x10 means FLOW_OFF_SENT, i.e. the previous frame has not been sent and is blocking further send operation. Flow OFF has been propagated to all CAIF Channels using this TTY.
- tty_status: Prints the bit-mask tty status information
 - 0x01 - tty->warned is on.
 - 0x04 - tty->packed is on.
 - 0x08 - tty->flow.tco_stopped is on.
 - 0x10 - tty->hw_stopped is on.
 - 0x20 - tty->flow.stopped is on.
- last_tx_msg: Binary blob Prints the last transmitted frame.

This can be printed with:

```
$od --format=x1 /sys/kernel/debug/caif_serial/<tty>/last_rx_msg.
```

The first two tx messages sent look like this. Note: The initial byte 02 is start of frame extension (STX) used for re-syncing upon errors.

- Enumeration:

00000000	02	05	00	00	03	01	d2	02
STX(1)								
Length(2)								
	Control	Channel(1)						
	Command:	Enumeration(1)						
	Link-ID(1)							
	Checksum(2)							

- Channel Setup:

00000000	02	07	00	00	00	21	a1	00	48	df
STX(1)										
Length(2)										
	Control	Channel(1)								
	Command:	Channel Setup(1)								
	Channel	Type(1)								
	Priority	and Link-ID(1)								
	Endpoint(1)									
	Checksum(2)									

- last_rx_msg: Prints the last transmitted frame.

The RX messages for LinkSetup look almost identical but they have the bit 0x20 set in the command bit, and Channel Setup has added one byte before Checksum containing Channel ID.

NOTE:

Several CAIF Messages might be concatenated. The maximum debug buffer size is 128 bytes.

9.2.4 Error Scenarios

- last_tx_msg contains channel setup message and last_rx_msg is empty -> The host seems to be able to send over the UART, at least the CAIF ldisc get notified that sending is completed.
- last_tx_msg contains enumeration message and last_rx_msg is empty -> The host is not able to send the message from UART, the tty has not been able to complete the transmit operation.
- if /sys/kernel/debug/caif_serial/<tty>/tty_status is non-zero there might be problems transmitting over UART.

E.g. host and modem wiring is not correct you will typically see tty_status = 0x10 (hw_stopped) and ser_state = 0x10 (FLOW_OFF_SENT).

You will probably see the enumeration message in last_tx_message and empty last_rx_message.

NETLINK INTERFACE FOR ETHTOOL

10.1 Basic information

Netlink interface for ethtool uses generic netlink family `ethtool` (userspace application should use macros `ETHTOOL_GENL_NAME` and `ETHTOOL_GENL_VERSION` defined in `<linux/ethtool_netlink.h>` uapi header). This family does not use a specific header, all information in requests and replies is passed using netlink attributes.

The ethtool netlink interface uses extended ACK for error and warning reporting, userspace application developers are encouraged to make these messages available to user in a suitable way.

Requests can be divided into three categories: "get" (retrieving information), "set" (setting parameters) and "action" (invoking an action).

All "set" and "action" type requests require admin privileges (`CAP_NET_ADMIN` in the namespace). Most "get" type requests are allowed for anyone but there are exceptions (where the response contains sensitive information). In some cases, the request as such is allowed for anyone but unprivileged users have attributes with sensitive information (e.g. wake-on-lan password) omitted.

10.2 Conventions

Attributes which represent a boolean value usually use `NLA_U8` type so that we can distinguish three states: "on", "off" and "not present" (meaning the information is not available in "get" requests or value is not to be changed in "set" requests). For these attributes, the "true" value should be passed as number 1 but any non-zero value should be understood as "true" by recipient. In the tables below, "bool" denotes `NLA_U8` attributes interpreted in this way.

In the message structure descriptions below, if an attribute name is suffixed with "+", parent nest can contain multiple attributes of the same type. This implements an array of entries.

Attributes that need to be filled-in by device drivers and that are dumped to user space based on whether they are valid or not should not use zero as a valid value. This avoids the need to explicitly signal the validity of the attribute in the device driver API.

10.3 Request header

Each request or reply message contains a nested attribute with common header. Structure of this header is

ETHTOOL_A_HEADER_DEV_INDEX	u32	device ifindex
ETHTOOL_A_HEADER_DEV_NAME	string	device name
ETHTOOL_A_HEADER_FLAGS	u32	flags common for all requests

ETHTOOL_A_HEADER_DEV_INDEX and ETHTOOL_A_HEADER_DEV_NAME identify the device message relates to. One of them is sufficient in requests, if both are used, they must identify the same device. Some requests, e.g. global string sets, do not require device identification. Most GET requests also allow dump requests without device identification to query the same information for all devices providing it (each device in a separate message).

ETHTOOL_A_HEADER_FLAGS is a bitmap of request flags common for all request types. The interpretation of these flags is the same for all request types but the flags may not apply to requests. Recognized flags are:

ETHTOOL_FLAG_COMPACT_BITSETS	use compact format bitsets in reply
ETHTOOL_FLAG OMIT_REPLY	omit optional reply (_SET and _ACT)
ETHTOOL_FLAG_STATS	include optional device statistics

New request flags should follow the general idea that if the flag is not set, the behaviour is backward compatible, i.e. requests from old clients not aware of the flag should be interpreted the way the client expects. A client must not set flags it does not understand.

10.4 Bit sets

For short bitmaps of (reasonably) fixed length, standard `NLA_BITFIELD32` type is used. For arbitrary length bitmaps, ethtool netlink uses a nested attribute with contents of one of two forms: compact (two binary bitmaps representing bit values and mask of affected bits) and bit-by-bit (list of bits identified by either index or name).

Verbose (bit-by-bit) bitsets allow sending symbolic names for bits together with their values which saves a round trip (when the bitset is passed in a request) or at least a second request (when the bitset is in a reply). This is useful for one shot applications like traditional ethtool command. On the other hand, long running applications like ethtool monitor (displaying notifications) or network management daemons may prefer fetching the names only once and using compact form to save message size. Notifications from ethtool netlink interface always use compact form for bitsets.

A bitset can represent either a value/mask pair (ETHTOOL_A_BITSET_NOMASK not set) or a single bitmap (ETHTOOL_A_BITSET_NOMASK set). In requests modifying a bitmap, the former changes the bit set in mask to values set in value and preserves the rest; the latter sets the bits set in the bitmap and clears the rest.

Compact form: nested (bitset) attribute contents:

<code>ETHTOOL_A_BITSET_NOMASK</code>	flag	no mask, only a list
<code>ETHTOOL_A_BITSET_SIZE</code>	u32	number of significant bits
<code>ETHTOOL_A_BITSET_VALUE</code>	binary	bitmap of bit values
<code>ETHTOOL_A_BITSET_MASK</code>	binary	bitmap of valid bits

Value and mask must have length at least `ETHTOOL_A_BITSET_SIZE` bits rounded up to a multiple of 32 bits. They consist of 32-bit words in host byte order, words ordered from least significant to most significant (i.e. the same way as bitmaps are passed with ioctl interface).

For compact form, `ETHTOOL_A_BITSET_SIZE` and `ETHTOOL_A_BITSET_VALUE` are mandatory. `ETHTOOL_A_BITSET_MASK` attribute is mandatory if `ETHTOOL_A_BITSET_NOMASK` is not set (bitset represents a value/mask pair); if `ETHTOOL_A_BITSET_NOMASK` is not set, `ETHTOOL_A_BITSET_MASK` is not allowed (bitset represents a single bitmap).

Kernel bit set length may differ from userspace length if older application is used on newer kernel or vice versa. If userspace bitmap is longer, an error is issued only if the request actually tries to set values of some bits not recognized by kernel.

Bit-by-bit form: nested (bitset) attribute contents:

<code>ETHTOOL_A_BITSET_NOMASK</code>	flag	no mask, only a list
<code>ETHTOOL_A_BITSET_SIZE</code>	u32	number of significant bits
<code>ETHTOOL_A_BITSET_BITS</code>	nested	array of bits
<code>ETHTOOL_A_BITSET_BITS_BIT+</code>	nested	one bit
	<code>ETHTOOL_A_BITSET_BIT_INDEX</code>	bit index (0 for LSB)
	<code>ETHTOOL_A_BITSET_BIT_NAME</code>	bit name
	<code>ETHTOOL_A_BITSET_BIT_VALUE</code>	present if bit is set

Bit size is optional for bit-by-bit form. `ETHTOOL_A_BITSET_BITS` nest can only contain `ETHTOOL_A_BITSET_BITS_BIT` attributes but there can be an arbitrary number of them. A bit may be identified by its index or by its name. When used in requests, listed bits are set to 0 or 1 according to `ETHTOOL_A_BITSET_BIT_VALUE`, the rest is preserved. A request fails if index exceeds kernel bit length or if name is not recognized.

When `ETHTOOL_A_BITSET_NOMASK` flag is present, bitset is interpreted as a simple bitmap. `ETHTOOL_A_BITSET_BIT_VALUE` attributes are not used in such case. Such bitset represents a bitmap with listed bits set and the rest zero.

In requests, application can use either form. Form used by kernel in reply is determined by `ETHTOOL_FLAG_COMPACT_BITSETS` flag in flags field of request header. Semantics of value and mask depends on the attribute.

10.5 List of message types

All constants identifying message types use `ETHTOOL_CMD_` prefix and suffix according to message purpose:

<code>_GET</code>	userspace request to retrieve data
<code>_SET</code>	userspace request to set data
<code>_ACT</code>	userspace request to perform an action
<code>_GET_REPLY</code>	kernel reply to a GET request
<code>_SET_REPLY</code>	kernel reply to a SET request
<code>_ACT_REPLY</code>	kernel reply to an ACT request
<code>_NTF</code>	kernel notification

Userspace to kernel:

<code>ETHTOOL_MSG_STRSET_GET</code>	get string set
<code>ETHTOOL_MSG_LINKINFO_GET</code>	get link settings
<code>ETHTOOL_MSG_LINKINFO_SET</code>	set link settings
<code>ETHTOOL_MSG_LINKMODES_GET</code>	get link modes info
<code>ETHTOOL_MSG_LINKMODES_SET</code>	set link modes info
<code>ETHTOOL_MSG_LINKSTATE_GET</code>	get link state
<code>ETHTOOL_MSG_DEBUG_GET</code>	get debugging settings
<code>ETHTOOL_MSG_DEBUG_SET</code>	set debugging settings
<code>ETHTOOL_MSG_WOL_GET</code>	get wake-on-lan settings
<code>ETHTOOL_MSG_WOL_SET</code>	set wake-on-lan settings
<code>ETHTOOL_MSG_FEATURES_GET</code>	get device features
<code>ETHTOOL_MSG_FEATURES_SET</code>	set device features
<code>ETHTOOL_MSG_PRIVFLAGS_GET</code>	get private flags
<code>ETHTOOL_MSG_PRIVFLAGS_SET</code>	set private flags
<code>ETHTOOL_MSG_RINGS_GET</code>	get ring sizes
<code>ETHTOOL_MSG_RINGS_SET</code>	set ring sizes
<code>ETHTOOL_MSG_CHANNELS_GET</code>	get channel counts
<code>ETHTOOL_MSG_CHANNELS_SET</code>	set channel counts
<code>ETHTOOL_MSG_COALESCE_GET</code>	get coalescing parameters
<code>ETHTOOL_MSG_COALESCE_SET</code>	set coalescing parameters
<code>ETHTOOL_MSG_PAUSE_GET</code>	get pause parameters
<code>ETHTOOL_MSG_PAUSE_SET</code>	set pause parameters
<code>ETHTOOL_MSG_EEE_GET</code>	get EEE settings
<code>ETHTOOL_MSG_EEE_SET</code>	set EEE settings
<code>ETHTOOL_MSG_TSINFO_GET</code>	get timestamping info
<code>ETHTOOL_MSG_CABLE_TEST_ACT</code>	action start cable test
<code>ETHTOOL_MSG_CABLE_TEST_TDR_ACT</code>	action start raw TDR cable test
<code>ETHTOOL_MSG_TUNNEL_INFO_GET</code>	get tunnel offload info
<code>ETHTOOL_MSG_FEC_GET</code>	get FEC settings
<code>ETHTOOL_MSG_FEC_SET</code>	set FEC settings
<code>ETHTOOL_MSG_MODULE_EEPROM_GET</code>	read SFP module EEPROM
<code>ETHTOOL_MSG_STATS_GET</code>	get standard statistics
<code>ETHTOOL_MSG_PHC_VCLOCKS_GET</code>	get PHC virtual clocks info

continues on next page

Table 1 - continued from previous page

ETHTOOL_MSG_MODULE_SET	set transceiver module parameters
ETHTOOL_MSG_MODULE_GET	get transceiver module parameters
ETHTOOL_MSG_PSE_SET	set PSE parameters
ETHTOOL_MSG_PSE_GET	get PSE parameters
ETHTOOL_MSG_RSS_GET	get RSS settings
ETHTOOL_MSG_PLCA_GET_CFG	get PLCA RS parameters
ETHTOOL_MSG_PLCA_SET_CFG	set PLCA RS parameters
ETHTOOL_MSG_PLCA_GET_STATUS	get PLCA RS status
ETHTOOL_MSG_MM_GET	get MAC merge layer state
ETHTOOL_MSG_MM_SET	set MAC merge layer parameters

Kernel to userspace:

ETHTOOL_MSG_STRSET_GET_REPLY	string set contents
ETHTOOL_MSG_LINKINFO_GET_REPLY	link settings
ETHTOOL_MSG_LINKINFO_NTF	link settings notification
ETHTOOL_MSG_LINKMODES_GET_REPLY	link modes info
ETHTOOL_MSG_LINKMODES_NTF	link modes notification
ETHTOOL_MSG_LINKSTATE_GET_REPLY	link state info
ETHTOOL_MSG_DEBUG_GET_REPLY	debugging settings
ETHTOOL_MSG_DEBUG_NTF	debugging settings notification
ETHTOOL_MSG_WOL_GET_REPLY	wake-on-lan settings
ETHTOOL_MSG_WOL_NTF	wake-on-lan settings notification
ETHTOOL_MSG_FEATURES_GET_REPLY	device features
ETHTOOL_MSG_FEATURES_SET_REPLY	optional reply to FEATURES_SET
ETHTOOL_MSG_FEATURES_NTF	netdev features notification
ETHTOOL_MSG_PRIVFLAGS_GET_REPLY	private flags
ETHTOOL_MSG_PRIVFLAGS_NTF	private flags
ETHTOOL_MSG_RINGS_GET_REPLY	ring sizes
ETHTOOL_MSG_RINGS_NTF	ring sizes
ETHTOOL_MSG_CHANNELS_GET_REPLY	channel counts
ETHTOOL_MSG_CHANNELS_NTF	channel counts
ETHTOOL_MSG_COALESCE_GET_REPLY	coalescing parameters
ETHTOOL_MSG_COALESCE_NTF	coalescing parameters
ETHTOOL_MSG_PAUSE_GET_REPLY	pause parameters
ETHTOOL_MSG_PAUSE_NTF	pause parameters
ETHTOOL_MSG_EEE_GET_REPLY	EEE settings
ETHTOOL_MSG_EEE_NTF	EEE settings
ETHTOOL_MSG_TSINFO_GET_REPLY	timestamping info
ETHTOOL_MSG_CABLE_TEST_NTF	Cable test results
ETHTOOL_MSG_CABLE_TEST_TDR_NTF	Cable test TDR results
ETHTOOL_MSG_TUNNEL_INFO_GET_REPLY	tunnel offload info
ETHTOOL_MSG_FEC_GET_REPLY	FEC settings
ETHTOOL_MSG_FEC_NTF	FEC settings
ETHTOOL_MSG_MODULE_EEPROM_GET_REPLY	read SFP module EEPROM
ETHTOOL_MSG_STATS_GET_REPLY	standard statistics

continues on next page

Table 2 – continued from previous page

ETHTOOL_MSG_PHC_VCLOCKS_GET_REPLY	PHC virtual clocks info
ETHTOOL_MSG_MODULE_GET_REPLY	transceiver module parameters
ETHTOOL_MSG_PSE_GET_REPLY	PSE parameters
ETHTOOL_MSG_RSS_GET_REPLY	RSS settings
ETHTOOL_MSG_PLCA_GET_CFG_REPLY	PLCA RS parameters
ETHTOOL_MSG_PLCA_GET_STATUS_REPLY	PLCA RS status
ETHTOOL_MSG_PLCA_NTF	PLCA RS parameters
ETHTOOL_MSG_MM_GET_REPLY	MAC merge layer status

GET requests are sent by userspace applications to retrieve device information. They usually do not contain any message specific attributes. Kernel replies with corresponding "GET_REPLY" message. For most types, GET request with NLM_F_DUMP and no device identification can be used to query the information for all devices supporting the request.

If the data can be also modified, corresponding SET message with the same layout as corresponding GET_REPLY is used to request changes. Only attributes where a change is requested are included in such request (also, not all attributes may be changed). Replies to most SET request consist only of error code and extack; if kernel provides additional data, it is sent in the form of corresponding SET_REPLY message which can be suppressed by setting ETHTOOL_FLAG OMIT_REPLY flag in request header.

Data modification also triggers sending a NTF message with a notification. These usually bear only a subset of attributes which was affected by the change. The same notification is issued if the data is modified using other means (mostly ioctl ethtool interface). Unlike notifications from ethtool netlink code which are only sent if something actually changed, notifications triggered by ioctl interface may be sent even if the request did not actually change any data.

ACT messages request kernel (driver) to perform a specific action. If some information is reported by kernel (which can be suppressed by setting ETHTOOL_FLAG OMIT_REPLY flag in request header), the reply takes form of an ACT_REPLY message. Performing an action also triggers a notification (NTF message).

Later sections describe the format and semantics of these messages.

10.6 STRSET_GET

Requests contents of a string set as provided by ioctl commands ETHTOOL_GSSET_INFO and ETHTOOL_GSTRINGS. String sets are not user writeable so that the corresponding STRSET_SET message is only used in kernel replies. There are two types of string sets: global (independent of a device, e.g. device feature names) and device specific (e.g. device private flags).

Request contents:

ETHTOOL_A_STRSET_HEADER	nested	request header
ETHTOOL_A_STRSET_STRINGSETS	nested	string set to request
ETHTOOL_A_STRINGSETS_STRINGSET+	nested	one string set
ETHTOOL_A_STRINGSET_ID32		set id

Kernel response contents:

ETHTOOL_A_STRSET_HEADER				nested	reply header
ETHTOOL_A_STRSET_STRINGSETS				nested	array of string sets
	ETHTOOL_A_STRINGSETS_STRINGSET+			nested	one string set
	ETHTOOL_A_STRINGSET_ID		u32	set id	
	ETHTOOL_A_STRINGSET_COUNT		u32	number of strings	
	ETHTOOL_A_STRINGSET_STRINGS			nested	array of strings
	ETHTOOL_A_STRINGS_STRING+nested			one string	
	ETHTOOL_A_STRING3INDEX		string index		
	ETHTOOL_A_STRING_VALUE			string value	
ETHTOOL_A_STRSET_COUNTS_ONLY				flag	return only counts

Device identification in request header is optional. Depending on its presence a and NLM_F_DUMP flag, there are three type of STRSET_GET requests:

- no NLM_F_DUMP, no device: get "global" stringsets
- no NLM_F_DUMP, with device: get string sets related to the device
- NLM_F_DUMP, no device: get device related string sets for all devices

If there is no ETHTOOL_A_STRSET_STRINGSETS array, all string sets of requested type are returned, otherwise only those specified in the request. Flag ETHTOOL_A_STRSET_COUNTS_ONLY tells kernel to only return string counts of the sets, not the actual strings.

10.7 LINKINFO_GET

Requests link settings as provided by ETHTOOL_GLINKSETTINGS except for link modes and autonegotiation related information. The request does not use any attributes.

Request contents:

ETHTOOL_A_LINKINFO_HEADER	nested	request header
---------------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_LINKINFO_HEADER	nested	reply header
ETHTOOL_A_LINKINFO_PORT	u8	physical port
ETHTOOL_A_LINKINFO_PHYADDR	u8	phy MDIO address
ETHTOOL_A_LINKINFO_TP_MDIX	u8	MDI(-X) status
ETHTOOL_A_LINKINFO_TP_MDIX_CTRL	u8	MDI(-X) control
ETHTOOL_A_LINKINFO_TRANSCEIVER	u8	transceiver

Attributes and their values have the same meaning as matching members of the corresponding ioctl structures.

`LINKINFO_GET` allows dump requests (kernel returns reply message for all devices supporting the request).

10.8 LINKINFO_SET

`LINKINFO_SET` request allows setting some of the attributes reported by `LINKINFO_GET`.

Request contents:

ETHTOOL_A_LINKINFO_HEADER	nested	request header
ETHTOOL_A_LINKINFO_PORT	u8	physical port
ETHTOOL_A_LINKINFO_PHYADDR	u8	phy MDIO address
ETHTOOL_A_LINKINFO_TP_MDIX_CTRL	u8	MDI(-X) control

MDI(-X) status and transceiver cannot be set, request with the corresponding attributes is rejected.

10.9 LINKMODES_GET

Requests link modes (supported, advertised and peer advertised) and related information (autonegotiation status, link speed and duplex) as provided by `ETHTOOL_GLINKSETTINGS`. The request does not use any attributes.

Request contents:

ETHTOOL_A_LINKMODES_HEADER	nested	request header
----------------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_LINKMODES_HEADER	nested	reply header
ETHTOOL_A_LINKMODES_AUTONEG	u8	autonegotiation status
ETHTOOL_A_LINKMODES_OURS	bitset	advertised link modes
ETHTOOL_A_LINKMODES_PEER	bitset	partner link modes
ETHTOOL_A_LINKMODES_SPEED	u32	link speed (Mb/s)
ETHTOOL_A_LINKMODES_DUPLEX	u8	duplex mode
ETHTOOL_A_LINKMODES_MASTER_SLAVE_CFG	u8	Master/slave port mode
ETHTOOL_A_LINKMODES_MASTER_SLAVE_STATE	u8	Master/slave port state
ETHTOOL_A_LINKMODES_RATE_MATCHING	u8	PHY rate matching

For ETHTOOL_A_LINKMODES_OURS, value represents advertised modes and mask represents supported modes. ETHTOOL_A_LINKMODES_PEER in the reply is a bit list.

LINKMODES_GET allows dump requests (kernel returns reply messages for all devices supporting the request).

10.10 LINKMODES_SET

Request contents:

ETHTOOL_A_LINKMODES_HEADER	nested	request header
ETHTOOL_A_LINKMODES_AUTONEG	u8	autonegotiation status
ETHTOOL_A_LINKMODES_OURS	bitset	advertised link modes
ETHTOOL_A_LINKMODES_PEER	bitset	partner link modes
ETHTOOL_A_LINKMODES_SPEED	u32	link speed (Mb/s)
ETHTOOL_A_LINKMODES_DUPLEX	u8	duplex mode
ETHTOOL_A_LINKMODES_MASTER_SLAVE_CFG	u8	Master/slave port mode
ETHTOOL_A_LINKMODES_RATE_MATCHING	u8	PHY rate matching
ETHTOOL_A_LINKMODES_LANES	u32	lanes

ETHTOOL_A_LINKMODES_OURS bit set allows setting advertised link modes. If autonegotiation is on (either set now or kept from before), advertised modes are not changed (no ETHTOOL_A_LINKMODES_OURS attribute) and at least one of speed, duplex and lanes is specified, kernel adjusts advertised modes to all supported modes matching speed, duplex, lanes or all (whatever is specified). This autoselection is done on ethtool side with ioctl interface, netlink interface is supposed to allow requesting changes without knowing what exactly kernel supports.

10.11 LINKSTATE_GET

Requests link state information. Link up/down flag (as provided by ETHTOOL_GLINK ioctl command) is provided. Optionally, extended state might be provided as well. In general, extended state describes reasons for why a port is down, or why it operates in some non-obvious mode. This request does not have any attributes.

Request contents:

ETHTOOL_A_LINKSTATE_HEADER	nested	request header
----------------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_LINKSTATE_HEADER	nested	reply header
ETHTOOL_A_LINKSTATE_LINK	bool	link state (up/down)
ETHTOOL_A_LINKSTATE_SQI	u32	Current Signal Quality Index
ETHTOOL_A_LINKSTATE_SQI_MAX	u32	Max support SQI value
ETHTOOL_A_LINKSTATE_EXT_STATE	u8	link extended state
ETHTOOL_A_LINKSTATE_EXT_SUBSTATE	u8	link extended substate
ETHTOOL_A_LINKSTATE_EXT_DOWN_CNT	u32	count of link down events

For most NIC drivers, the value of `ETHTOOL_A_LINKSTATE_LINK` returns carrier flag provided by `netif_carrier_ok()` but there are drivers which define their own handler.

`ETHTOOL_A_LINKSTATE_EXT_STATE` and `ETHTOOL_A_LINKSTATE_EXT_SUBSTATE` are optional values. ethtool core can provide either both `ETHTOOL_A_LINKSTATE_EXT_STATE` and `ETHTOOL_A_LINKSTATE_EXT_SUBSTATE`, or only `ETHTOOL_A_LINKSTATE_EXT_STATE`, or none of them.

`LINKSTATE_GET` allows dump requests (kernel returns reply messages for all devices supporting the request).

Link extended states:

<code>ETHTOOL_LINK_EXT_STATE_AUTONEG</code>	States relating to the autonegotiation or issues therein
<code>ETHTOOL_LINK_EXT_STATE_LINK_TRAINING_FAILURE</code>	Failure during link training
<code>ETHTOOL_LINK_EXT_STATE_LINK_LOGICALISMATCH</code>	Logical mismatch in physical coding sub-layer or forward error correction sub-layer
<code>ETHTOOL_LINK_EXT_STATE_BAD_SIGNAL_INTEGRITY</code>	Integrity issues
<code>ETHTOOL_LINK_EXT_STATE_NO_CABLE</code>	No cable connected
<code>ETHTOOL_LINK_EXT_STATE_CABLE_ISSUE</code>	Failure is related to cable, e.g., unsupported cable
<code>ETHTOOL_LINK_EXT_STATE_EEPROM_ISSUE</code>	Failure is related to EEPROM, e.g., failure during reading or parsing the data
<code>ETHTOOL_LINK_EXT_STATE_CALIBRATION_FAILURE</code>	Failure during calibration algorithm
<code>ETHTOOL_LINK_EXT_STATE_POWER_BUDGET_EXCEEDED</code>	Hardware is not able to provide the power required from cable or module
<code>ETHTOOL_LINK_EXT_STATE_OVERHEAT</code>	The module is overheated
<code>ETHTOOL_LINK_EXT_STATE_MODULE</code>	Transceiver module issue

Link extended substates:

Autoneg substates:

<code>ETHTOOL_LINK_EXT_SUBSTATE_AN_NO_PARTNER_SELECTED</code>	Partner selected down
<code>ETHTOOL_LINK_EXT_SUBSTATE_AN_ACK_NOT_RECEIVED</code>	Ack received from peer side
<code>ETHTOOL_LINK_EXT_SUBSTATE_AN_NEXT_PAGE_EXCHANGE_FAILED</code>	Next exchange failed
<code>ETHTOOL_LINK_EXT_SUBSTATE_AN_NO_PARTNER_SELECTED_FORCED_MODE</code>	Force mode or there is no agreement of speed
<code>ETHTOOL_LINK_EXT_SUBSTATE_AN_FEC_MISMATCH_DURING_OVERRIDE</code>	mismatch modes in both sides are mismatched
<code>ETHTOOL_LINK_EXT_SUBSTATE_AN_NO_HCD</code>	No Highest Common Denominator

Link training substates:

ETHTOOL_LINK_EXT_SUBSTATE_LT_KR_FRAMELOCKS_NOREQUIRED	Frame locks were not recognized, the lock failed
ETHTOOL_LINK_EXT_SUBSTATE_LT_KR_LINKTIMEOUT	The link did not occur before timeout
ETHTOOL_LINK_EXT_SUBSTATE_LT_KR_LINKPARTNER_DIDNOTSENTERREADY	Partner did not sense receiver ready
ETHTOOL_LINK_EXT_SUBSTATE_LT_REMOTE	Remote side is not ready yet

Link logical mismatch substates:

ETHTOOL_LINK_EXT_SUBSTATE_LLM_PCS_BLOCKLOCK	Did not acquire block lock not locked in first phase - block lock
ETHTOOL_LINK_EXT_SUBSTATE_LLM_PCS_ALIGNMENTLOCK	Did not acquire alignment lock was not locked in second phase - alignment markers lock
ETHTOOL_LINK_EXT_SUBSTATE_LLM_PCS_FORWARDLOCKED	Did not get alignment status
ETHTOOL_LINK_EXT_SUBSTATE_LLM_FC_FORWARDLOCKED	FC forward lock correction is not locked
ETHTOOL_LINK_EXT_SUBSTATE_LLM_RS_FORWARDLOCKED	RS forward lock correction is not locked

Bad signal integrity substates:

ETHTOOL_LINK_EXT_SUBSTATE_BSI_LARGEERRORS	Number of physical errors
ETHTOOL_LINK_EXT_SUBSTATE_BSI_UNSUPPORTEDRATE	Attempted to operate the cable at a rate that is not formally supported, which led to signal integrity issues
ETHTOOL_LINK_EXT_SUBSTATE_BSI_SERDESCLOCK	The reference clock signal for SerDes is too weak or unavailable.
ETHTOOL_LINK_EXT_SUBSTATE_BSI_SELDLOSS	The received signal for SerDes is too weak because analog loss of signal.

Cable issue substates:

ETHTOOL_LINK_EXT_SUBSTATE_CI_UNSUPPORTED_CABLE	Unsupported cable
ETHTOOL_LINK_EXT_SUBSTATE_CI_CABLE_TEST_FAILURE	Cable test failure

Transceiver module issue substates:

ETHTOOL_LINK_EXT_SUBSTATE_MODULE_CMISNOMREADY	Module State Machine did not reach the ModuleReady state. For example, if the module is stuck at ModuleFault state
---	--

10.12 DEBUG_GET

Requests debugging settings of a device. At the moment, only message mask is provided.

Request contents:

ETHTOOL_A_DEBUG_HEADER	nested	request header
------------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_DEBUG_HEADER	nested	reply header
ETHTOOL_A_DEBUG_MSGMASK	bitset	message mask

The message mask (ETHTOOL_A_DEBUG_MSGMASK) is equal to message level as provided by ETHTOOL_GMSGLEVEL and set by ETHTOOL_SMSGLVL in ioctl interface. While it is called message level there for historical reasons, most drivers and almost all newer drivers use it as a mask of enabled message classes (represented by NETIF_MSG_* constants); therefore netlink interface follows its actual use in practice.

DEBUG_GET allows dump requests (kernel returns reply messages for all devices supporting the request).

10.13 DEBUG_SET

Set or update debugging settings of a device. At the moment, only message mask is supported.

Request contents:

ETHTOOL_A_DEBUG_HEADER	nested	request header
ETHTOOL_A_DEBUG_MSGMASK	bitset	message mask

ETHTOOL_A_DEBUG_MSGMASK bit set allows setting or modifying mask of enabled debugging message types for the device.

10.14 WOL_GET

Query device wake-on-lan settings. Unlike most "GET" type requests, ETHTOOL_MSG_WOL_GET requires (netns) CAP_NET_ADMIN privileges as it (potentially) provides SecureOn(tm) password which is confidential.

Request contents:

ETHTOOL_A_WOL_HEADER	nested	request header
----------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_WOL_HEADER	nested	reply header
ETHTOOL_A_WOL_MODES	bitset	mask of enabled WoL modes
ETHTOOL_A_WOL_SOPASS	binary	SecureOn(tm) password

In reply, ETHTOOL_A_WOL_MODES mask consists of modes supported by the device, value of modes which are enabled. ETHTOOL_A_WOL_SOPASS is only included in reply if WAKE_MAGICSECURE mode is supported.

10.15 WOL_SET

Set or update wake-on-lan settings.

Request contents:

ETHTOOL_A_WOL_HEADER	nested	request header
ETHTOOL_A_WOL_MODES	bitset	enabled WoL modes
ETHTOOL_A_WOL_SOPASS	binary	SecureOn(tm) password

ETHTOOL_A_WOL_SOPASS is only allowed for devices supporting WAKE_MAGICSECURE mode.

10.16 FEATURES_GET

Gets netdev features like ETHTOOL_GFEATURES ioctl request.

Request contents:

ETHTOOL_A_FEATURES_HEADER	nested	request header
---------------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_FEATURES_HEADER	nested	reply header
ETHTOOL_A_FEATURES_HW	bitset	dev->hw_features
ETHTOOL_A_FEATURES_WANTED	bitset	dev->wanted_features
ETHTOOL_A_FEATURES_ACTIVE	bitset	dev->features
ETHTOOL_A_FEATURES_NOCHANGE	bitset	NETIF_F_NEVER_CHANGE

Bitmaps in kernel response have the same meaning as bitmaps used in ioctl interference but attribute names are different (they are based on corresponding members of *struct net_device*). Legacy "flags" are not provided, if userspace needs them (most likely only ethtool for backward compatibility), it can calculate their values from related feature bits itself. ETHTOOL_FEATURES_HW uses mask consisting of all features recognized by kernel (to provide all names when using verbose bitmap format), the other three use no mask (simple bit lists).

10.17 FEATURES_SET

Request to set netdev features like ETHTOOL_SFEATURES ioctl request.

Request contents:

ETHTOOL_A_FEATURES_HEADER	nested	request header
ETHTOOL_A_FEATURES_WANTED	bitset	requested features

Kernel response contents:

ETHTOOL_A_FEATURES_HEADER	nested	reply header
ETHTOOL_A_FEATURES_WANTED	bitset	diff wanted vs. result
ETHTOOL_A_FEATURES_ACTIVE	bitset	diff old vs. new active

Request contains only one bitset which can be either value/mask pair (request to change specific feature bits and leave the rest) or only a value (request to set all features to specified set).

As request is subject to [*netdev_change_features\(\)*](#) sanity checks, optional kernel reply (can be suppressed by ETHTOOL_FLAG OMIT_REPLY flag in request header) informs client about the actual result. ETHTOOL_A_FEATURES_WANTED reports the difference between client request and actual result: mask consists of bits which differ between requested features and result (dev->features after the operation), value consists of values of these bits in the request (i.e. negated values from resulting features). ETHTOOL_A_FEATURES_ACTIVE reports the difference between old and new dev->features: mask consists of bits which have changed, values are their values in new dev->features (after the operation).

ETHTOOL_MSG_FEATURES_NTF notification is sent not only if device features are modified using ETHTOOL_MSG_FEATURES_SET request or on of ethtool ioctl request but also each time features are modified with [*netdev_update_features\(\)*](#) or [*netdev_change_features\(\)*](#).

10.18 PRIVFLAGS_GET

Gets private flags like ETHTOOL_GPFLAGS ioctl request.

Request contents:

ETHTOOL_A_PRIVFLAGS_HEADER	nested	request header
----------------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_PRIVFLAGS_HEADER	nested	reply header
ETHTOOL_A_PRIVFLAGS_FLAGS	bitset	private flags

ETHTOOL_A_PRIVFLAGS_FLAGS is a bitset with values of device private flags. These flags are defined by driver, their number and names (and also meaning) are device dependent. For compact bitset format, names can be retrieved as ETH_SS_PRIV_FLAGS string set. If verbose bitset format is requested, response uses all private flags supported by the device as mask so that client gets the full information without having to fetch the string set with names.

10.19 PRIVFLAGS_SET

Sets or modifies values of device private flags like ETHTOOL_SPFLAGS ioctl request.

Request contents:

ETHTOOL_A_PRIVFLAGS_HEADER	nested	request header
ETHTOOL_A_PRIVFLAGS_FLAGS	bitset	private flags

ETHTOOL_A_PRIVFLAGS_FLAGS can either set the whole set of private flags or modify only values of some of them.

10.20 RINGS_GET

Gets ring sizes like ETHTOOL_GRINGPARAM ioctl request.

Request contents:

ETHTOOL_A_RINGS_HEADER	nested	request header
------------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_RINGS_HEADER	nested	reply header
ETHTOOL_A_RINGS_RX_MAX	u32	max size of RX ring
ETHTOOL_A_RINGS_RX_MINI_MAX	u32	max size of RX mini ring
ETHTOOL_A_RINGS_RX_JUMBO_MAX	u32	max size of RX jumbo ring
ETHTOOL_A_RINGS_TX_MAX	u32	max size of TX ring
ETHTOOL_A_RINGS_RX	u32	size of RX ring
ETHTOOL_A_RINGS_RX_MINI	u32	size of RX mini ring
ETHTOOL_A_RINGS_RX_JUMBO	u32	size of RX jumbo ring
ETHTOOL_A_RINGS_TX	u32	size of TX ring
ETHTOOL_A_RINGS_RX_BUF_LEN	u32	size of buffers on the ring
ETHTOOL_A_RINGS_TCP_DATA_SPLIT	u8	TCP header / data split
ETHTOOL_A_RINGS_CQE_SIZE	u32	Size of TX/RX CQE
ETHTOOL_A_RINGS_TX_PUSH	u8	flag of TX Push mode
ETHTOOL_A_RINGS_RX_PUSH	u8	flag of RX Push mode
ETHTOOL_A_RINGS_TX_PUSH_BUF_LEN	u32	size of TX push buffer
ETHTOOL_A_RINGS_TX_PUSH_BUF_LEN_MAX	u32	max size of TX push buffer

ETHTOOL_A_RINGS_TCP_DATA_SPLIT indicates whether the device is usable with page-flipping TCP zero-copy receive (getsockopt(TCP_ZEROCOPY_RECEIVE)). If enabled the device is configured to place frame headers and data into separate buffers. The device configuration must make it possible to receive full memory pages of data, for example because MTU is high enough or through HW-GRO.

ETHTOOL_A_RINGS_[RX|TX]_PUSH flag is used to enable descriptor fast path to send or receive packets. In ordinary path, driver fills descriptors in DRAM and notifies NIC hardware. In fast path, driver pushes descriptors to the device through MMIO writes, thus reducing the latency.

However, enabling this feature may increase the CPU cost. Drivers may enforce additional per-packet eligibility checks (e.g. on packet size).

`ETHTOOL_A_RINGS_TX_PUSH_BUF_LEN` specifies the maximum number of bytes of a transmitted packet a driver can push directly to the underlying device ('push' mode). Pushing some of the payload bytes to the device has the advantages of reducing latency for small packets by avoiding DMA mapping (same as `ETHTOOL_A_RINGS_TX_PUSH` parameter) as well as allowing the underlying device to process packet headers ahead of fetching its payload. This can help the device to make fast actions based on the packet's headers. This is similar to the "tx-copybreak" parameter, which copies the packet to a preallocated DMA memory area instead of mapping new memory. However, `tx-push-buff` parameter copies the packet directly to the device to allow the device to take faster actions on the packet.

10.21 RINGS_SET

Sets ring sizes like `ETHTOOL_SRINGPARAM` ioctl request.

Request contents:

<code>ETHTOOL_A_RINGS_HEADER</code>	nested	reply header
<code>ETHTOOL_A_RINGS_RX</code>	u32	size of RX ring
<code>ETHTOOL_A_RINGS_RX_MINI</code>	u32	size of RX mini ring
<code>ETHTOOL_A_RINGS_RX_JUMBO</code>	u32	size of RX jumbo ring
<code>ETHTOOL_A_RINGS_TX</code>	u32	size of TX ring
<code>ETHTOOL_A_RINGS_RX_BUF_LEN</code>	u32	size of buffers on the ring
<code>ETHTOOL_A_RINGS_CQE_SIZE</code>	u32	Size of TX/RX CQE
<code>ETHTOOL_A_RINGS_TX_PUSH</code>	u8	flag of TX Push mode
<code>ETHTOOL_A_RINGS_RX_PUSH</code>	u8	flag of RX Push mode
<code>ETHTOOL_A_RINGS_TX_PUSH_BUF_LEN</code>	u32	size of TX push buffer

Kernel checks that requested ring sizes do not exceed limits reported by driver. Driver may impose additional constraints and may not support all attributes.

`ETHTOOL_A_RINGS_CQE_SIZE` specifies the completion queue event size. Completion queue events(CQE) are the events posted by NIC to indicate the completion status of a packet when the packet is sent(like send success or error) or received(like pointers to packet fragments). The CQE size parameter enables to modify the CQE size other than default size if NIC supports it. A bigger CQE can have more receive buffer pointers in turn NIC can transfer a bigger frame from wire. Based on the NIC hardware, the overall completion queue size can be adjusted in the driver if CQE size is modified.

10.22 CHANNELS_GET

Gets channel counts like ETHTOOL_GCHANNELS ioctl request.

Request contents:

ETHTOOL_A_CHANNELS_HEADER	nested	request header
---------------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_CHANNELS_HEADER	nested	reply header
ETHTOOL_A_CHANNELS_RX_MAX	u32	max receive channels
ETHTOOL_A_CHANNELS_TX_MAX	u32	max transmit channels
ETHTOOL_A_CHANNELS_OTHER_MAX	u32	max other channels
ETHTOOL_A_CHANNELS_COMBINED_MAX	u32	max combined channels
ETHTOOL_A_CHANNELS_RX_COUNT	u32	receive channel count
ETHTOOL_A_CHANNELS_TX_COUNT	u32	transmit channel count
ETHTOOL_A_CHANNELS_OTHER_COUNT	u32	other channel count
ETHTOOL_A_CHANNELS_COMBINED_COUNT	u32	combined channel count

10.23 CHANNELS_SET

Sets channel counts like ETHTOOL_SCHANNELS ioctl request.

Request contents:

ETHTOOL_A_CHANNELS_HEADER	nested	request header
ETHTOOL_A_CHANNELS_RX_COUNT	u32	receive channel count
ETHTOOL_A_CHANNELS_TX_COUNT	u32	transmit channel count
ETHTOOL_A_CHANNELS_OTHER_COUNT	u32	other channel count
ETHTOOL_A_CHANNELS_COMBINED_COUNT	u32	combined channel count

Kernel checks that requested channel counts do not exceed limits reported by driver. Driver may impose additional constraints and may not support all attributes.

10.24 COALESCE_GET

Gets coalescing parameters like ETHTOOL_GCOALESCE ioctl request.

Request contents:

ETHTOOL_A_COALESCE_HEADER	nested	request header
---------------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_COALESCE_HEADER	nested	reply header
ETHTOOL_A_COALESCE_RX_USECS	u32	delay (us), normal Rx
ETHTOOL_A_COALESCE_RX_MAX_FRAMES	u32	max packets, normal Rx
ETHTOOL_A_COALESCE_RX_USECS_IRQ	u32	delay (us), Rx in IRQ
ETHTOOL_A_COALESCE_RX_MAX_FRAMES_IRQ	u32	max packets, Rx in IRQ
ETHTOOL_A_COALESCE_TX_USECS	u32	delay (us), normal Tx
ETHTOOL_A_COALESCE_TX_MAX_FRAMES	u32	max packets, normal Tx
ETHTOOL_A_COALESCE_TX_USECS_IRQ	u32	delay (us), Tx in IRQ
ETHTOOL_A_COALESCE_TX_MAX_FRAMES_IRQ	u32	IRQ packets, Tx in IRQ
ETHTOOL_A_COALESCE_STATS_BLOCK_USECS	u32	delay of stats update
ETHTOOL_A_COALESCE_USE_ADAPTIVE_RX	bool	adaptive Rx coalesce
ETHTOOL_A_COALESCE_USE_ADAPTIVE_TX	bool	adaptive Tx coalesce
ETHTOOL_A_COALESCE_PKT_RATE_LOW	u32	threshold for low rate
ETHTOOL_A_COALESCE_RX_USECS_LOW	u32	delay (us), low Rx
ETHTOOL_A_COALESCE_RX_MAX_FRAMES_LOW	u32	max packets, low Rx
ETHTOOL_A_COALESCE_TX_USECS_LOW	u32	delay (us), low Tx
ETHTOOL_A_COALESCE_TX_MAX_FRAMES_LOW	u32	max packets, low Tx
ETHTOOL_A_COALESCE_PKT_RATE_HIGH	u32	threshold for high rate
ETHTOOL_A_COALESCE_RX_USECS_HIGH	u32	delay (us), high Rx
ETHTOOL_A_COALESCE_RX_MAX_FRAMES_HIGH	u32	max packets, high Rx
ETHTOOL_A_COALESCE_TX_USECS_HIGH	u32	delay (us), high Tx
ETHTOOL_A_COALESCE_TX_MAX_FRAMES_HIGH	u32	max packets, high Tx
ETHTOOL_A_COALESCE_RATE_SAMPLE_INTERVAL	u32	rate sampling interval
ETHTOOL_A_COALESCE_USE_CQE_TX	bool	timer reset mode, Tx
ETHTOOL_A_COALESCE_USE_CQE_RX	bool	timer reset mode, Rx
ETHTOOL_A_COALESCE_TX_AGGR_MAX_BYTES	u32	max aggr size, Tx
ETHTOOL_A_COALESCE_TX_AGGR_MAX_FRAMES	u32	max aggr packets, Tx
ETHTOOL_A_COALESCE_TX_AGGR_TIME_USECS	u32	time (us), agrgr, Tx

Attributes are only included in reply if their value is not zero or the corresponding bit in `ethtool_ops::supported_coalesce_params` is set (i.e. they are declared as supported by driver).

Timer reset mode (`ETHTOOL_A_COALESCE_USE_CQE_TX` and `ETHTOOL_A_COALESCE_USE_CQE_RX`) controls the interaction between packet arrival and the various time based delay parameters. By default timers are expected to limit the max delay between any packet arrival/departure and a corresponding interrupt. In this mode timer should be started by packet arrival (sometimes delivery of previous interrupt) and reset when interrupt is delivered. Setting the appropriate attribute to 1 will enable CQE mode, where each packet event resets the timer. In this mode timer is used to force the interrupt if queue goes idle, while busy queues depend on the packet limit to trigger interrupts.

Tx aggregation consists of copying frames into a contiguous buffer so that they can be submitted as a single IO operation. `ETHTOOL_A_COALESCE_TX_AGGR_MAX_BYTES` describes the maximum size in bytes for the submitted buffer. `ETHTOOL_A_COALESCE_TX_AGGR_MAX_FRAMES` describes the maximum number of frames that can be aggregated into a single buffer. `ETHTOOL_A_COALESCE_TX_AGGR_TIME_USECS` describes the amount of time in usecs, counted since the first packet arrival in an aggregated block, after which the block should be sent. This feature is mainly of interest for specific USB devices which does not cope well with frequent small-sized URBs transmissions.

10.25 COALESCE_SET

Sets coalescing parameters like ETHTOOL_SCOALESCE ioctl request.

Request contents:

ETHTOOL_A_COALESCE_HEADER	nested	request header
ETHTOOL_A_COALESCE_RX_USECS	u32	delay (us), normal Rx
ETHTOOL_A_COALESCE_RX_MAX_FRAMES	u32	max packets, normal Rx
ETHTOOL_A_COALESCE_RX_USECS_IRQ	u32	delay (us), Rx in IRQ
ETHTOOL_A_COALESCE_RX_MAX_FRAMES_IRQ	u32	max packets, Rx in IRQ
ETHTOOL_A_COALESCE_TX_USECS	u32	delay (us), normal Tx
ETHTOOL_A_COALESCE_TX_MAX_FRAMES	u32	max packets, normal Tx
ETHTOOL_A_COALESCE_TX_USECS_IRQ	u32	delay (us), Tx in IRQ
ETHTOOL_A_COALESCE_TX_MAX_FRAMES_IRQ	u32	IRQ packets, Tx in IRQ
ETHTOOL_A_COALESCE_STATS_BLOCK_USECS	u32	delay of stats update
ETHTOOL_A_COALESCE_USE_ADAPTIVE_RX	bool	adaptive Rx coalesce
ETHTOOL_A_COALESCE_USE_ADAPTIVE_TX	bool	adaptive Tx coalesce
ETHTOOL_A_COALESCE_PKT_RATE_LOW	u32	threshold for low rate
ETHTOOL_A_COALESCE_RX_USECS_LOW	u32	delay (us), low Rx
ETHTOOL_A_COALESCE_RX_MAX_FRAMES_LOW	u32	max packets, low Rx
ETHTOOL_A_COALESCE_TX_USECS_LOW	u32	delay (us), low Tx
ETHTOOL_A_COALESCE_TX_MAX_FRAMES_LOW	u32	max packets, low Tx
ETHTOOL_A_COALESCE_PKT_RATE_HIGH	u32	threshold for high rate
ETHTOOL_A_COALESCE_RX_USECS_HIGH	u32	delay (us), high Rx
ETHTOOL_A_COALESCE_RX_MAX_FRAMES_HIGH	u32	max packets, high Rx
ETHTOOL_A_COALESCE_TX_USECS_HIGH	u32	delay (us), high Tx
ETHTOOL_A_COALESCE_TX_MAX_FRAMES_HIGH	u32	max packets, high Tx
ETHTOOL_A_COALESCE_RATE_SAMPLE_INTERVAL	u32	rate sampling interval
ETHTOOL_A_COALESCE_USE_CQE_TX	bool	timer reset mode, Tx
ETHTOOL_A_COALESCE_USE_CQE_RX	bool	timer reset mode, Rx
ETHTOOL_A_COALESCE_TX_AGGR_MAX_BYTES	u32	max aggr size, Tx
ETHTOOL_A_COALESCE_TX_AGGR_MAX_FRAMES	u32	max aggr packets, Tx
ETHTOOL_A_COALESCE_TX_AGGR_TIME_USECS	u32	time (us), aggr, Tx

Request is rejected if it attributes declared as unsupported by driver (i.e. such that the corresponding bit in `ethtool_ops::supported_coalesce_params` is not set), regardless of their values. Driver may impose additional constraints on coalescing parameters and their values.

Compared to requests issued via the `ioctl()` netlink version of this request will try harder to make sure that values specified by the user have been applied and may call the driver twice.

10.26 PAUSE_GET

Gets pause frame settings like ETHTOOL_GPAUSEPARAM ioctl request.

Request contents:

ETHTOOL_A_PAUSE_HEADER	nested	request header
ETHTOOL_A_PAUSE_STATS_SRC	u32	source of statistics

ETHTOOL_A_PAUSE_STATS_SRC is optional. It takes values from:

```
enum ethtool_mac_stats_src
    source of ethtool MAC statistics
```

Constants

ETHTOOL_MAC_STATS_SRC_AGGREGATE

if device supports a MAC merge layer, this retrieves the aggregate statistics of the eMAC and pMAC. Otherwise, it retrieves just the statistics of the single (express) MAC.

ETHTOOL_MAC_STATS_SRC_EMAC

if device supports a MM layer, this retrieves the eMAC statistics. Otherwise, it retrieves the statistics of the single (express) MAC.

ETHTOOL_MAC_STATS_SRC_PMAC

if device supports a MM layer, this retrieves the pMAC statistics.

If absent from the request, stats will be provided with an ETHTOOL_A_PAUSE_STATS_SRC attribute in the response equal to ETHTOOL_MAC_STATS_SRC_AGGREGATE.

Kernel response contents:

ETHTOOL_A_PAUSE_HEADER	nested	request header
ETHTOOL_A_PAUSE_AUTONEG	bool	pause autonegotiation
ETHTOOL_A_PAUSE_RX	bool	receive pause frames
ETHTOOL_A_PAUSE_TX	bool	transmit pause frames
ETHTOOL_A_PAUSE_STATS	nested	pause statistics

ETHTOOL_A_PAUSE_STATS are reported if ETHTOOL_FLAG_STATS was set in ETHTOOL_A_HEADER_FLAGS. It will be empty if driver did not report any statistics. Drivers fill in the statistics in the following structure:

```
struct ethtool_pause_stats
    statistics for IEEE 802.3x pause frames
```

Definition:

```
struct ethtool_pause_stats {
    enum ethtool_mac_stats_src src;
    u64 tx_pause_frames;
    u64 rx_pause_frames;
};
```

Members

src

input field denoting whether stats should be queried from the eMAC or pMAC (if the MM layer is supported). To be ignored otherwise.

tx_pause_frames

transmitted pause frame count. Reported to user space as ETHTOOL_A_PAUSE_STAT_TX_FRAMES.

Equivalent to 30.3.4.2 *aPAUSEMACCtrlFramesTransmitted* from the standard.

rx_pause_frames

received pause frame count. Reported to user space as ETHTOOL_A_PAUSE_STAT_RX_FRAMES. Equivalent to:

Equivalent to 30.3.4.3 *aPAUSEMACCtrlFramesReceived* from the standard.

Each member has a corresponding attribute defined.

10.27 PAUSE_SET

Sets pause parameters like ETHTOOL_GPAUSEPARAM ioctl request.

Request contents:

ETHTOOL_A_PAUSE_HEADER	nested	request header
ETHTOOL_A_PAUSE_AUTONEG	bool	pause autonegotiation
ETHTOOL_A_PAUSE_RX	bool	receive pause frames
ETHTOOL_A_PAUSE_TX	bool	transmit pause frames

10.28 EEE_GET

Gets Energy Efficient Ethernet settings like ETHTOOL_GEEE ioctl request.

Request contents:

ETHTOOL_A_EEE_HEADER	nested	request header
----------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_EEE_HEADER	nested	request header
ETHTOOL_A_EEE_MODES_OURS	bool	supported/advertised modes
ETHTOOL_A_EEE_MODES_PEER	bool	peer advertised link modes
ETHTOOL_A_EEE_ACTIVE	bool	EEE is actively used
ETHTOOL_A_EEE_ENABLED	bool	EEE is enabled
ETHTOOL_A_EEE_TX_LPI_ENABLED	bool	Tx lpi enabled
ETHTOOL_A_EEE_TX_LPI_TIMER	u32	Tx lpi timeout (in us)

In ETHTOOL_A_EEE_MODES_OURS, mask consists of link modes for which EEE is enabled, value of link modes for which EEE is advertised. Link modes for which peer advertises EEE are listed in

`ETHTOOL_A_EEE_MODES_PEER` (no mask). The netlink interface allows reporting EEE status for all link modes but only first 32 are provided by the `ethtool_ops` callback.

10.29 EEE_SET

Sets Energy Efficient Ethernet parameters like `ETHTOOL_SEEE` ioctl request.

Request contents:

<code>ETHTOOL_A_EEE_HEADER</code>	nested	request header
<code>ETHTOOL_A_EEE_MODES_OURS</code>	bool	advertised modes
<code>ETHTOOL_A_EEE_ENABLED</code>	bool	EEE is enabled
<code>ETHTOOL_A_EEE_TX_LPI_ENABLED</code>	bool	Tx lpi enabled
<code>ETHTOOL_A_EEE_TX_LPI_TIMER</code>	u32	Tx lpi timeout (in us)

`ETHTOOL_A_EEE_MODES_OURS` is used to either list link modes to advertise EEE for (if there is no mask) or specify changes to the list (if there is a mask). The netlink interface allows reporting EEE status for all link modes but only first 32 can be set at the moment as that is what the `ethtool_ops` callback supports.

10.30 TSINFO_GET

Gets timestamping information like `ETHTOOL_GET_TS_INFO` ioctl request.

Request contents:

<code>ETHTOOL_A_TSINFO_HEADER</code>	nested	request header
--------------------------------------	--------	----------------

Kernel response contents:

<code>ETHTOOL_A_TSINFO_HEADER</code>	nested	request header
<code>ETHTOOL_A_TSINFO_TIMESTAMPING</code>	bitset	SO_TIMESTAMPING flags
<code>ETHTOOL_A_TSINFO_TX_TYPES</code>	bitset	supported Tx types
<code>ETHTOOL_A_TSINFO_RX_FILTERS</code>	bitset	supported Rx filters
<code>ETHTOOL_A_TSINFO_PHC_INDEX</code>	u32	PTP hw clock index

`ETHTOOL_A_TSINFO_PHC_INDEX` is absent if there is no associated PHC (there is no special value for this case). The bitset attributes are omitted if they would be empty (no bit set).

10.31 CABLE_TEST

Start a cable test.

Request contents:

ETHTOOL_A_CABLE_TEST_HEADER	nested	request header
-----------------------------	--------	----------------

Notification contents:

An Ethernet cable typically contains 1, 2 or 4 pairs. The length of the pair can only be measured when there is a fault in the pair and hence a reflection. Information about the fault may not be available, depending on the specific hardware. Hence the contents of the notify message are mostly optional. The attributes can be repeated an arbitrary number of times, in an arbitrary order, for an arbitrary number of pairs.

The example shows the notification sent when the test is completed for a T2 cable, i.e. two pairs. One pair is OK and hence has no length information. The second pair has a fault and does have length information.

ETHTOOL_A_CABLE_TEST_HEADER	nested	reply header
ETHTOOL_A_CABLE_TEST_STATUS	u8	completed
ETHTOOL_A_CABLE_TEST_NTF_NEST	nested	all the results
ETHTOOL_A_CABLE_NEST_RESULT	nested	cable test result
ETHTOOL_A_CABLE_RESULTS_PAIR	PAIR	pair number
ETHTOOL_A_CABLE_RESULTS_CODE	CODE	result code
ETHTOOL_A_CABLE_NEST_RESULT	nested	cable test results
ETHTOOL_A_CABLE_RESULTS_PAIR	PAIR	pair number
ETHTOOL_A_CABLE_RESULTS_CODE	CODE	result code
ETHTOOL_A_CABLE_NEST_FAULT_LENGTH	nested	cable length
ETHTOOL_A_CABLE_FAULT_LENGTH_PAIR	PAT	pair number
ETHTOOL_A_CABLE_FAULT_LENGTH_CM	CM	length in cm

10.32 CABLE_TEST_TDR

Start a cable test and report raw TDR data

Request contents:

ETHTOOL_A_CABLE_TEST_TDR_HEADER	nested	reply header
ETHTOOL_A_CABLE_TEST_TDR_CFG	nested	test configuration
ETHTOOL_A_CABLE_STEP_FIRST_DISTANCE	u32	first data distance
ETHTOOL_A_CABLE_STEP_LAST_DISTANCE	u32	last data distance
ETHTOOL_A_CABLE_STEP_STEP_DISTANCE	u32	distance of each step
ETHTOOL_A_CABLE_TEST_TDR_CFG_PAIR	u8	pair to test

The ETHTOOL_A_CABLE_TEST_TDR_CFG is optional, as well as all members of the nest. All distances are expressed in centimeters. The PHY takes the distances as a guide, and rounds to the nearest distance it actually supports. If a pair is passed, only that one pair will be tested. Otherwise all pairs are tested.

Notification contents:

Raw TDR data is gathered by sending a pulse down the cable and recording the amplitude of the reflected pulse for a given distance.

It can take a number of seconds to collect TDR data, especial if the full 100 meters is probed at 1 meter intervals. When the test is started a notification will be sent containing just ETHTOOL_A_CABLE_TEST_TDR_STATUS with the value ETHTOOL_A_CABLE_TEST_NTF_STATUS_STARTED.

When the test has completed a second notification will be sent containing ETHTOOL_A_CABLE_TEST_TDR_STATUS with the value ETHTOOL_A_CABLE_TEST_NTF_STATUS_COMPLETED and the TDR data.

The message may optionally contain the amplitude of the pulse send down the cable. This is measured in mV. A reflection should not be bigger than transmitted pulse.

Before the raw TDR data should be an ETHTOOL_A_CABLE_TDR_NEST_STEP nest containing information about the distance along the cable for the first reading, the last reading, and the step between each reading. Distances are measured in centimeters. These should be the exact values the PHY used. These may be different to what the user requested, if the native measurement resolution is greater than 1 cm.

For each step along the cable, a ETHTOOL_A_CABLE_TDR_NEST_AMPLITUDE is used to report the amplitude of the reflection for a given pair.

ETHTOOL_A_CABLE_TEST_TDR_HEADER		nested	reply header
ETHTOOL_A_CABLE_TEST_TDR_STATUS		u8	completed
ETHTOOL_A_CABLE_TEST_TDR_NTF_NEST		nested	all the results
	ETHTOOL_A_CABLE_TDR_NEST_PULSE	nested	TX Pulse amplitude
		ETHTOOL_A_CABLE_PULSE	mV
	ETHTOOL_A_CABLE_NEST_STEP	nested	TDR step info
		ETHTOOL_A_CABLE_STEP_FIRST_DIST	SIANCE data distance
		ETHTOOL_A_CABLE_STEP_LAST_DIST	SIANCE data distance
		ETHTOOL_A_CABLE_STEP_STEP_DIST	SIANCE of each step
	ETHTOOL_A_CABLE_TDR_NEST_AMPLITUDE	nested	Reflection amplitude
		ETHTOOL_A_CABLE_RESULTS18PAIR	pair number
		ETHTOOL_A_CABLE_AMPLITUDE16mV	Reflection amplitude
	ETHTOOL_A_CABLE_TDR_NEST_AMPLITUDE	nested	Reflection amplitude
		ETHTOOL_A_CABLE_RESULTS18PAIR	pair number
		ETHTOOL_A_CABLE_AMPLITUDE16mV	Reflection amplitude
	ETHTOOL_A_CABLE_TDR_NEST_AMPLITUDE	nested	Reflection amplitude
		ETHTOOL_A_CABLE_RESULTS18PAIR	pair number
		ETHTOOL_A_CABLE_AMPLITUDE16mV	Reflection amplitude

10.33 TUNNEL_INFO

Gets information about the tunnel state NIC is aware of.

Request contents:

ETHTOOL_A_TUNNEL_INFO_HEADER	nested	request header
------------------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_TUNNEL_INFO_HEADER	nested	reply header
ETHTOOL_A_TUNNEL_INFO_UDP_PORTS	nested	all UDP port tables
ETHTOOL_A_TUNNEL_UDP_TABLE	nested	one UDP port table
ETHTOOL_A_TUNNEL_UDP_TABLE_SIZE		max size of the table
ETHTOOL_A_TUNNEL_UDP_TABLE_TYPES		tunnel types which table can hold
ETHTOOL_A_TUNNEL_UDP_TABLE_ENTRIES		offloaded UDP port
	ETHTOOL_A_TUNNEL_UDP_ENTRY_TYPE	IP6_ENTRYPORt
	ETHTOOL_A_TUNNEL_UDP_ENTRY_TYPE	UDP_ENTRYnTYPE

For UDP tunnel table empty ETHTOOL_A_TUNNEL_UDP_TABLE_TYPES indicates that the table contains static entries, hard-coded by the NIC.

10.34 FEC_GET

Gets FEC configuration and state like ETHTOOL_GFECPARAM ioctl request.

Request contents:

ETHTOOL_A_FEC_HEADER	nested	request header
----------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_FEC_HEADER	nested	request header
ETHTOOL_A_FEC_MODES	bitset	configured modes
ETHTOOL_A_FEC_AUTO	bool	FEC mode auto selection
ETHTOOL_A_FEC_ACTIVE	u32	index of active FEC mode
ETHTOOL_A_FEC_STATS	nested	FEC statistics

ETHTOOL_A_FEC_ACTIVE is the bit index of the FEC link mode currently active on the interface. This attribute may not be present if device does not support FEC.

ETHTOOL_A_FEC_MODES and ETHTOOL_A_FEC_AUTO are only meaningful when autonegotiation is disabled. If ETHTOOL_A_FEC_AUTO is non-zero driver will select the FEC mode automatically based on the parameters of the SFP module. This is equivalent to the ETHTOOL_FEC_AUTO bit of the ioctl interface. ETHTOOL_A_FEC_MODES carry the current FEC configuration using link mode bits (rather than old ETHTOOL_FEC_* bits).

ETHTOOL_A_FEC_STATS are reported if ETHTOOL_FLAG_STATS was set in ETHTOOL_A_HEADER_FLAGS. Each attribute carries an array of 64bit statistics. First entry in the array contains the total number of events on the port, while the following entries are counters corresponding to lanes/PCS instances. The number of entries in the array will be:

<i>0</i>	device does not support FEC statistics
<i>1</i>	device does not support per-lane break down
<i>1 + #lanes</i>	device has full support for FEC stats

Drivers fill in the statistics in the following structure:

```
struct ethtool_fec_stats
    statistics for IEEE 802.3 FEC
```

Definition:

```
struct ethtool_fec_stats {
    struct ethtool_fec_stat {
        u64 total;
        u64 lanes[ETHTOOL_MAX_LANES];
    } corrected_blocks, uncorrectable_blocks, corrected_bits;
};
```

Members

corrected_blocks

number of received blocks corrected by FEC Reported to user space as ETHTOOL_A_FEC_STAT_CORRECTED.

Equivalent to 30.5.1.1.17 *aFECCorrectedBlocks* from the standard.

uncorrectable_blocks

number of received blocks FEC was not able to correct Reported to user space as ETHTOOL_A_FEC_STAT_UNCORR.

Equivalent to 30.5.1.1.18 *aFECUncorrectableBlocks* from the standard.

corrected_bits

number of bits corrected by FEC Similar to **corrected_blocks** but counts individual bit changes, not entire FEC data blocks. This is a non-standard statistic. Reported to user space as ETHTOOL_A_FEC_STAT_CORR_BITS.

Description

For each of the above fields, the two substructure members are:

- **lanes**: per-lane/PCS-instance counts as defined by the standard
- **total: error counts for the entire port, for drivers incapable of reporting per-lane stats**

Drivers should fill in either only total or per-lane statistics, core will take care of adding lane values up to produce the total.

10.35 FEC_SET

Sets FEC parameters like ETHTOOL_SFECPARAM ioctl request.

Request contents:

ETHTOOL_A_FEC_HEADER	nested	request header
ETHTOOL_A_FEC_MODES	bitset	configured modes
ETHTOOL_A_FEC_AUTO	bool	FEC mode auto selection

FEC_SET is only meaningful when autonegotiation is disabled. Otherwise FEC mode is selected as part of autonegotiation.

ETHTOOL_A_FEC_MODES selects which FEC mode should be used. It's recommended to set only one bit, if multiple bits are set driver may choose between them in an implementation specific way.

ETHTOOL_A_FEC_AUTO requests the driver to choose FEC mode based on SFP module parameters. This does not mean autonegotiation.

10.36 MODULE_EEPROM_GET

Fetch module EEPROM data dump. This interface is designed to allow dumps of at most 1/2 page at once. This means only dumps of 128 (or less) bytes are allowed, without crossing half page boundary located at offset 128. For pages other than 0 only high 128 bytes are accessible.

Request contents:

ETHTOOL_A_MODULE_EEPROM_HEADER	nested	request header
ETHTOOL_A_MODULE_EEPROM_OFFSET	u32	offset within a page
ETHTOOL_A_MODULE_EEPROM_LENGTH	u32	amount of bytes to read
ETHTOOL_A_MODULE_EEPROM_PAGE	u8	page number
ETHTOOL_A_MODULE_EEPROM_BANK	u8	bank number
ETHTOOL_A_MODULE_EEPROM_I2C_ADDRESS	u8	page I2C address

If ETHTOOL_A_MODULE_EEPROM_BANK is not specified, bank 0 is assumed.

Kernel response contents:

ETHTOOL_A_MODULE_EEPROM_HEADER	nested	reply header
ETHTOOL_A_MODULE_EEPROM_DATA	binary	array of bytes from module EEPROM

ETHTOOL_A_MODULE_EEPROM_DATA has an attribute length equal to the amount of bytes driver actually read.

10.37 STATS_GET

Get standard statistics for the interface. Note that this is not a re-implementation of ETHTOOL_GSTATS which exposed driver-defined stats.

Request contents:

ETHTOOL_A_STATS_HEADER	nested	request header
ETHTOOL_A_STATS_SRC	u32	source of statistics
ETHTOOL_A_STATS_GROUPS	bitset	requested groups of stats

Kernel response contents:

ETHTOOL_A_STATS_HEADER	nested	reply header
ETHTOOL_A_STATS_SRC	u32	source of statistics
ETHTOOL_A_STATS_GRP	nested	one or more group of stats
	ETHTOOL_A_STATS_GRP_ID	group ID - ETHTOOL_STATS_*
	ETHTOOL_A_STATS_GRP_SS_ID	string set ID for names
	ETHTOOL_A_STATS_GRP_STAT	nest containing a statistic
	ETHTOOL_A_STATS_GRP_HIST_RXnested	histogram statistic (Rx)
	ETHTOOL_A_STATS_GRP_HIST_TXnested	histogram statistic (Tx)

Users specify which groups of statistics they are requesting via the ETHTOOL_A_STATS_GROUPS bitset. Currently defined values are:

ETH-TOOL_STATS_ETH_MAC	eth-mac	Basic IEEE 802.3 MAC statistics (30.3.1.1.*)
ETH-TOOL_STATS_ETH_PHY	eth-phy	Basic IEEE 802.3 PHY statistics (30.3.2.1.*)
ETH-TOOL_STATS_ETH_CTRL	eth-ctrl	Basic IEEE 802.3 MAC Ctrl statistics (30.3.3.*)
ETHTOOL_STATS_RMON	rmon	RMON (RFC 2819) statistics

Each group should have a corresponding ETHTOOL_A_STATS_GRP in the reply. ETHTOOL_A_STATS_GRP_ID identifies which group's statistics nest contains. ETHTOOL_A_STATS_GRP_SS_ID identifies the string set ID for the names of the statistics in the group, if available.

Statistics are added to the ETHTOOL_A_STATS_GRP nest under ETHTOOL_A_STATS_GRP_STAT. ETHTOOL_A_STATS_GRP_STAT should contain single 8 byte (u64) attribute inside - the type of that attribute is the statistic ID and the value is the value of the statistic. Each group has its own interpretation of statistic IDs. Attribute IDs correspond to strings from the string set identified by ETHTOOL_A_STATS_GRP_SS_ID. Complex statistics (such as RMON histogram entries) are also listed inside ETHTOOL_A_STATS_GRP and do not have a string defined in the string set.

RMON "histogram" counters count number of packets within given size range. Because RFC does not specify the ranges beyond the standard 1518 MTU devices differ in definition of buckets. For this reason the definition of packet ranges is left to each driver.

ETHTOOL_A_STATS_GRP_HIST_RX and ETHTOOL_A_STATS_GRP_HIST_TX nests contain the following attributes:

ETHTOOL_A_STATS_RMON_HIST_BKT_LOW	u32	low bound of the packet size bucket
ETHTOOL_A_STATS_RMON_HIST_BKT_HI	u32	high bound of the bucket
ETHTOOL_A_STATS_RMON_HIST_VAL	u64	packet counter

Low and high bounds are inclusive, for example:

RFC statistic	low	high
etherStatsPkts64Octets	0	64
etherStatsPkts512to1023Octets	512	1023

ETHTOOL_A_STATS_SRC is optional. Similar to PAUSE_GET, it takes values from enum ethtool_mac_stats_src. If absent from the request, stats will be provided with an ETHTOOL_A_STATS_SRC attribute in the response equal to ETHTOOL_MAC_STATS_SRC_AGGREGATE.

10.38 PHC_VCLOCKS_GET

Query device PHC virtual clocks information.

Request contents:

ETHTOOL_A_PHC_VCLOCKS_HEADER	nested	request header
------------------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_PHC_VCLOCKS_HEADER	nested	reply header
ETHTOOL_A_PHC_VCLOCKS_NUM	u32	PHC virtual clocks number
ETHTOOL_A_PHC_VCLOCKS_INDEX	s32	PHC index array

10.39 MODULE_GET

Gets transceiver module parameters.

Request contents:

ETHTOOL_A_MODULE_HEADER	nested	request header
-------------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_MODULE_HEADER	nested	reply header
ETHTOOL_A_MODULE_POWER_MODE_POLICY	u8	power mode policy
ETHTOOL_A_MODULE_POWER_MODE	u8	operational power mode

The optional `ETHTOOL_A_MODULE_POWER_MODE_POLICY` attribute encodes the transceiver module power mode policy enforced by the host. The default policy is driver-dependent, but "auto" is the recommended default and it should be implemented by new drivers and drivers where conformance to a legacy behavior is not critical.

The optional `ETHTH00L_A_MODULE_POWER_MODE` attribute encodes the operational power mode policy of the transceiver module. It is only reported when a module is plugged-in. Possible values are:

enum **`ethtool_module_power_mode`**

 plug-in module power mode

Constants

`ETHTOOL_MODULE_POWER_MODE_LOW`

 Module is in low power mode.

`ETHTOOL_MODULE_POWER_MODE_HIGH`

 Module is in high power mode.

10.40 MODULE_SET

Sets transceiver module parameters.

Request contents:

<code>ETHTOOL_A_MODULE_HEADER</code>	nested	request header
<code>ETHTOOL_A_MODULE_POWER_MODE_POLICY</code>	u8	power mode policy

When set, the optional `ETHTOOL_A_MODULE_POWER_MODE_POLICY` attribute is used to set the transceiver module power policy enforced by the host. Possible values are:

enum **`ethtool_module_power_mode_policy`**

 plug-in module power mode policy

Constants

`ETHTOOL_MODULE_POWER_MODE_POLICY_HIGH`

 Module is always in high power mode.

`ETHTOOL_MODULE_POWER_MODE_POLICY_AUTO`

 Module is transitioned by the host to high power mode when the first port using it is put administratively up and to low power mode when the last port using it is put administratively down.

For SFF-8636 modules, low power mode is forced by the host according to table 6-10 in revision 2.10a of the specification.

For CMIS modules, low power mode is forced by the host according to table 6-12 in revision 5.0 of the specification.

10.41 PSE_GET

Gets PSE attributes.

Request contents:

ETHTOOL_A_PSE_HEADER	nested	request header
----------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_PSE_HEADER	nested	reply header
ETHTOOL_A_PODL_PSE_ADMIN_STATE	IEEE32	Operational state of the PoDL PSE functions
ETHTOOL_A_PODL_PSE_PW_D_STATUS	US32	power detection status of the PoDL PSE.

When set, the optional ETHTOOL_A_PODL_PSE_ADMIN_STATE attribute identifies the operational state of the PoDL PSE functions. The operational state of the PSE function can be changed using the ETHTOOL_A_PODL_PSE_ADMIN_CONTROL action. This option is corresponding to IEEE 802.3-2018 30.15.1.1.2 aPoDLPSEAdminState. Possible values are:

enum **ethtool_podl_pse_admin_state**

operational state of the PoDL PSE functions. IEEE 802.3-2018 30.15.1.1.2 aPoDLPSEAdminState

Constants

ETHTOOL_PODL_PSE_ADMIN_STATE_UNKNOWN

PoDL PSE functions are unknown

ETHTOOL_PODL_PSE_ADMIN_STATE_DISABLED

PoDL PSE functions are disabled

ETHTOOL_PODL_PSE_ADMIN_STATE_ENABLED

PoDL PSE functions are enabled

When set, the optional ETHTOOL_A_PODL_PSE_PW_D_STATUS attribute identifies the power detection status of the PoDL PSE. The status depend on internal PSE state machine and automatic PD classification support. This option is corresponding to IEEE 802.3-2018 30.15.1.1.3 aPoDLPSEPowerDetectionStatus. Possible values are:

enum **ethtool_podl_pse_pw_d_status**

power detection status of the PoDL PSE. IEEE 802.3-2018 30.15.1.1.3 aPoDLPSEPowerDetectionStatus:

Constants

ETHTOOL_PODL_PSE_PW_D_STATUS_UNKNOWN

PoDL PSE

ETHTOOL_PODL_PSE_PW_D_STATUS_DISABLED

"The enumeration "disabled" is asserted true when the PoDL PSE state diagram variable mr_pse_enable is false"

ETHTOOL_PODL_PSE_PW_D_STATUS_SEARCHING

"The enumeration "searching" is asserted true when either of the PSE state diagram variables pi_detecting or pi_classifying is true."

ETHTOOL_PODL_PSE_PW_D_STATUS_DELIVERING

"The enumeration "deliveringPower" is asserted true when the PoDL PSE state diagram variable pi_powered is true."

ETHTOOL_PODL_PSE_PW_D_STATUS_SLEEP

"The enumeration "sleep" is asserted true when the PoDL PSE state diagram variable pi_sleeping is true."

ETHTOOL_PODL_PSE_PW_D_STATUS_IDLE

"The enumeration "idle" is asserted true when the logical combination of the PoDL PSE state diagram variables pi_preambled*!pi_sleeping is true."

ETHTOOL_PODL_PSE_PW_D_STATUS_ERROR

"The enumeration "error" is asserted true when the PoDL PSE state diagram variable overload_held is true."

10.42 PSE_SET

Sets PSE parameters.

Request contents:

ETHTOOL_A_PSE_HEADER	nested	request header
ETHTOOL_A_PODL_PSE_ADMIN_CONTROL	u32	Control PoDL PSE Admin state

When set, the optional ETHTOOL_A_PODL_PSE_ADMIN_CONTROL attribute is used to control PoDL PSE Admin functions. This option is implementing IEEE 802.3-2018 30.15.1.2.1 acPoDLPSEAdminControl. See ETHTOOL_A_PODL_PSE_ADMIN_STATE for supported values.

10.43 RSS_GET

Get indirection table, hash key and hash function info associated with a RSS context of an interface similar to ETHTOOL_GRSSH ioctl request.

Request contents:

ETHTOOL_A_RSS_HEADER	nested	request header
ETHTOOL_A_RSS_CONTEXT	u32	context number

Kernel response contents:

ETHTOOL_A_RSS_HEADER	nested	reply header
ETHTOOL_A_RSS_HFUNC	u32	RSS hash func
ETHTOOL_A_RSS_INDIR	binary	Indir table bytes
ETHTOOL_A_RSS_HKEY	binary	Hash key bytes
ETHTOOL_A_RSS_INPUT_XFRM	u32	RSS input data transformation

ETHTOOL_A_RSS_HFUNC attribute is bitmap indicating the hash function being used. Current supported options are toeplitz, xor or crc32. ETHTOOL_A_RSS_INDIR attribute returns RSS indirection table where each byte indicates queue number. ETHTOOL_A_RSS_INPUT_XFRM attribute is a bitmap indicating the type of transformation applied to the input protocol fields before given to the RSS hfunc. Current supported option is symmetric-xor.

10.44 PLCA_GET_CFG

Gets the IEEE 802.3cg-2019 Clause 148 Physical Layer Collision Avoidance (PLCA) Reconciliation Sublayer (RS) attributes.

Request contents:

ETHTOOL_A_PLCA_HEADER	nested	request header
-----------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_PLCA_HEADER	nested	reply header
ETHTOOL_A_PLCA_VERSION	u16	Supported PLCA management interface standard/version
ETHTOOL_A_PLCA_ENABLED	u8	PLCA Admin State
ETHTOOL_A_PLCA_NODE_ID	u32	PLCA unique local node ID
ETHTOOL_A_PLCA_NODE_CNT	u32	Number of PLCA nodes on the network, including the coordinator
ETHTOOL_A_PLCA_TO_TMR	u32	Transmit Opportunity Timer value in bit-times (BT)
ETHTOOL_A_PLCA_BURST_CNT	Tu32	Number of additional packets the node is allowed to send within a single TO
ETHTOOL_A_PLCA_BURST_TMR	Tu32	Time to wait for the MAC to transmit a new frame before terminating the burst

When set, the optional ETHTOOL_A_PLCA_VERSION attribute indicates which standard and version the PLCA management interface complies to. When not set, the interface is vendor-specific and (possibly) supplied by the driver. The OPEN Alliance SIG specifies a standard register map for 10BASE-T1S PHYs embedding the PLCA Reconciliation Sublayer. See "10BASE-T1S PLCA Management Registers" at <https://www.opensig.org/about/specifications/>.

When set, the optional ETHTOOL_A_PLCA_ENABLED attribute indicates the administrative state of the PLCA RS. When not set, the node operates in "plain" CSMA/CD mode. This option is corresponding to IEEE 802.3cg-2019 30.16.1.1.1 aPLCAAdminState / 30.16.1.2.1 acPLCAAdminControl.

When set, the optional ETHTOOL_A_PLCA_NODE_ID attribute indicates the configured local node ID of the PHY. This ID determines which transmit opportunity (TO) is reserved for the node to transmit into. This option is corresponding to IEEE 802.3cg-2019 30.16.1.1.4 aPLCALocalNodeID. The valid range for this attribute is [0 .. 255] where 255 means "not configured".

When set, the optional ETHTOOL_A_PLCA_NODE_CNT attribute indicates the configured maximum number of PLCA nodes on the mixing-segment. This number determines the total number of transmit opportunities generated during a PLCA cycle. This attribute is relevant only for the

PLCA coordinator, which is the node with aPLCALocalNodeID set to 0. Follower nodes ignore this setting. This option is corresponding to IEEE 802.3cg-2019 30.16.1.1.3 aPLCANodeCount. The valid range for this attribute is [1 .. 255].

When set, the optional ETHTOOL_A_PLCA_TO_TMR attribute indicates the configured value of the transmit opportunity timer in bit-times. This value must be set equal across all nodes sharing the medium for PLCA to work correctly. This option is corresponding to IEEE 802.3cg-2019 30.16.1.1.5 aPLCATransmitOpportunityTimer. The valid range for this attribute is [0 .. 255].

When set, the optional ETHTOOL_A_PLCA_BURST_CNT attribute indicates the configured number of extra packets that the node is allowed to send during a single transmit opportunity. By default, this attribute is 0, meaning that the node can only send a single frame per TO. When greater than 0, the PLCA RS keeps the TO after any transmission, waiting for the MAC to send a new frame for up to aPLCABurstTimer BTs. This can only happen a number of times per PLCA cycle up to the value of this parameter. After that, the burst is over and the normal counting of TOs resumes. This option is corresponding to IEEE 802.3cg-2019 30.16.1.1.6 aPLCAMaxBurstCount. The valid range for this attribute is [0 .. 255].

When set, the optional ETHTOOL_A_PLCA_BURST_TMR attribute indicates how many bit-times the PLCA RS waits for the MAC to initiate a new transmission when aPLCAMaxBurstCount is greater than 0. If the MAC fails to send a new frame within this time, the burst ends and the counting of TOs resumes. Otherwise, the new frame is sent as part of the current burst. This option is corresponding to IEEE 802.3cg-2019 30.16.1.1.7 aPLCABurstTimer. The valid range for this attribute is [0 .. 255]. Although, the value should be set greater than the Inter-Frame-Gap (IFG) time of the MAC (plus some margin) for PLCA burst mode to work as intended.

10.45 PLCA_SET_CFG

Sets PLCA RS parameters.

Request contents:

ETHTOOL_A_PLCA_HEADER	nested	request header
ETHTOOL_A_PLCA_ENABLED	u8	PLCA Admin State
ETHTOOL_A_PLCA_NODE_ID	u8	PLCA unique local node ID
ETHTOOL_A_PLCA_NODE_CNT	u8	Number of PLCA nodes on the netkork, including the coordinator
ETHTOOL_A_PLCA_TO_TMR	u8	Transmit Opportunity Timer value in bit-times (BT)
ETHTOOL_A_PLCA_BURST_CNT	u8	Number of additional packets the node is allowed to send within a single TO
ETHTOOL_A_PLCA_BURST_TMR	u8	Time to wait for the MAC to transmit a new frame before terminating the burst

For a description of each attribute, see [PLCA_GET_CFG](#).

10.46 PLCA_GET_STATUS

Gets PLCA RS status information.

Request contents:

ETHTOOL_A_PLCA_HEADER	nested	request header
-----------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_PLCA_HEADER	nested	reply header
ETHTOOL_A_PLCA_STATUS	u8	PLCA RS operational status

When set, the ETHTOOL_A_PLCA_STATUS attribute indicates whether the node is detecting the presence of the BEACON on the network. This flag is corresponding to IEEE 802.3cg-2019 30.16.1.1.2 aPLCAStatus.

10.47 MM_GET

Retrieve 802.3 MAC Merge parameters.

Request contents:

ETHTOOL_A_MM_HEADER	nested	request header
---------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_MM_HEADER	nested	request header
ETHTOOL_A_MM_PMAC_ENABLED	bool	set if RX of preemptible and SMD-V frames is enabled
ETHTOOL_A_MM_TX_ENABLED	bool	set if TX of preemptible frames is administratively enabled (might be inactive if verification failed)
ETHTOOL_A_MM_TX_ACTIVE	bool	set if TX of preemptible frames is operationally enabled
ETHTOOL_A_MM_TX_MIN_FRAG_SIZE	u32	minimum size of transmitted non-final fragments, in octets
ETHTOOL_A_MM_RX_MIN_FRAG_SIZE	u32	minimum size of received non-final fragments, in octets
ETHTOOL_A_MM_VERIFY_ENABLED	bool	set if TX of SMD-V frames is administratively enabled
ETHTOOL_A_MM_VERIFY_STATUS	u8	state of the verification function
ETHTOOL_A_MM_VERIFY_TIME	u32	delay between verification attempts
ETHTOOL_A_MM_MAX_VERIFY_TIME	u32	maximum verification interval supported by device
ETHTOOL_A_MM_STATS	nested	IEEE 802.3-2018 subclause 30.14.1 oMACMergeEntity statistics counters

The attributes are populated by the device driver through the following structure:

```
struct ethtool_mm_state
    802.3 MAC merge layer state
```

Definition:

```
struct ethtool_mm_state {
    u32 verify_time;
    u32 max_verify_time;
    enum ethtool_mm_verify_status verify_status;
    bool tx_enabled;
    bool tx_active;
    bool pmac_enabled;
    bool verify_enabled;
    u32 tx_min_frag_size;
    u32 rx_min_frag_size;
};
```

Members

verify_time

wait time between verification attempts in ms (according to clause 30.14.1.6 aMACMergeVerifyTime)

max_verify_time

maximum accepted value for the **verify_time** variable in set requests

verify_status

state of the verification state machine of the MM layer (according to clause 30.14.1.2 aMACMergeStatusVerify)

tx_enabled

set if the MM layer is administratively enabled in the TX direction (according to clause 30.14.1.3 aMACMergeEnableTx)

tx_active

set if the MM layer is enabled in the TX direction, which makes FP possible (according to 30.14.1.5 aMACMergeStatusTx). This should be true if MM is enabled, and the verification status is either verified, or disabled.

pmac_enabled

set if the preemptible MAC is powered on and is able to receive preemptible packets and respond to verification frames.

verify_enabled

set if the Verify function of the MM layer (which sends SMD-V verification requests) is administratively enabled (regardless of whether it is currently in the ETH_TOOL_MM_VERIFY_STATUS_DISABLED state or not), according to clause 30.14.1.4 aMACMergeVerifyDisableTx (but using positive rather than negative logic). The device should always respond to received SMD-V requests as long as **pmac_enabled** is set.

`tx_min_frag_size`

the minimum size of non-final mPacket fragments that the link partner supports receiving, expressed in octets. Compared to the definition from clause 30.14.1.7 aMACMergeAddFragSize which is expressed in the range 0 to 3 (requiring a translation to the size in octets according to the formula $64 * (1 + \text{addFragSize}) - 4$), a value in a continuous and unbounded range can be specified here.

`rx_min_frag_size`

the minimum size of non-final mPacket fragments that this device supports receiving, expressed in octets.

The `ETHTOOL_A_MM_VERIFY_STATUS` will report one of the values from

enum `ethtool_mm_verify_status`

status of MAC Merge Verify function

Constants

`ETHTOOL_MM_VERIFY_STATUS_UNKNOWN`

verification status is unknown

`ETHTOOL_MM_VERIFY_STATUS_INITIAL`

the 802.3 Verify State diagram is in the state INIT_VERIFICATION

`ETHTOOL_MM_VERIFY_STATUS VERIFYING`

the Verify State diagram is in the state VERIFICATION_IDLE, SEND_VERIFY or WAIT_FOR_RESPONSE

`ETHTOOL_MM_VERIFY_STATUS_SUCCEEDED`

indicates that the Verify State diagram is in the state VERIFIED

`ETHTOOL_MM_VERIFY_STATUS FAILED`

the Verify State diagram is in the state VERIFY_FAIL

`ETHTOOL_MM_VERIFY_STATUS_DISABLED`

verification of preemption operation is disabled

If `ETHTOOL_A_MM_VERIFY_ENABLED` was passed as false in the `MM_SET` command, `ETHTOOL_A_MM_VERIFY_STATUS` will report either `ETHTOOL_MM_VERIFY_STATUS_INITIAL` or `ETHTOOL_MM_VERIFY_STATUS_DISABLED`, otherwise it should report one of the other states.

It is recommended that drivers start with the pMAC disabled, and enable it upon user space request. It is also recommended that user space does not depend upon the default values from `ETHTOOL_MSG_MM_GET` requests.

`ETHTOOL_A_MM_STATS` are reported if `ETHTOOL_FLAG_STATS` was set in `ETHTOOL_A_HEADER_FLAGS`. The attribute will be empty if driver did not report any statistics. Drivers fill in the statistics in the following structure:

struct `ethtool_mm_stats`

802.3 MAC merge layer statistics

Definition:

```
struct ethtool_mm_stats {
    u64 MACMergeFrameAssErrorCount;
    u64 MACMergeFrameSmdErrorCount;
    u64 MACMergeFrameAssOkCount;
    u64 MACMergeFragCountRx;
    u64 MACMergeFragCountTx;
    u64 MACMergeHoldCount;
};
```

Members**MACMergeFrameAssErrorCount**

received MAC frames with reassembly errors

MACMergeFrameSmdErrorCount

received MAC frames/fragments rejected due to unknown or incorrect SMD

MACMergeFrameAssOkCount

received MAC frames that were successfully reassembled and passed up

MACMergeFragCountRx

number of additional correct SMD-C mPackets received due to preemption

MACMergeFragCountTx

number of additional mPackets sent due to preemption

MACMergeHoldCount

number of times the MM layer entered the HOLD state, which blocks transmission of preemptible traffic

10.48 MM_SET

Modifies the configuration of the 802.3 MAC Merge layer.

Request contents:

ETHTOOL_A_MM_VERIFY_TIME	u32	see MM_GET description
ETHTOOL_A_MM_VERIFY_ENABLED	bool	see MM_GET description
ETHTOOL_A_MM_TX_ENABLED	bool	see MM_GET description
ETHTOOL_A_MM_PMAC_ENABLED	bool	see MM_GET description
ETHTOOL_A_MM_TX_MIN_FRAG_SIZE	u32	see MM_GET description

The attributes are propagated to the driver through the following structure:

struct ethtool_mm_cfg

802.3 MAC merge layer configuration

Definition:

```
struct ethtool_mm_cfg {
    u32 verify_time;
    bool verify_enabled;
    bool tx_enabled;
    bool pmac_enabled;
    u32 tx_min_frag_size;
};
```

Members

verify_time
 see *struct ethtool_mm_state*

verify_enabled
 see *struct ethtool_mm_state*

tx_enabled
 see *struct ethtool_mm_state*

pmac_enabled
 see *struct ethtool_mm_state*

tx_min_frag_size
 see *struct ethtool_mm_state*

10.49 Request translation

The following table maps ioctl commands to netlink commands providing their functionality. Entries with "n/a" in right column are commands which do not have their netlink replacement yet. Entries which "n/a" in the left column are netlink only.

ioctl command	netlink command
ETHTOOL_GSET	ETHTOOL_MSG_LINKINFO_GET ETHTOOL_MSG_LINKMODES_GET
ETHTOOL_SSET	ETHTOOL_MSG_LINKINFO_SET ETHTOOL_MSG_LINKMODES_SET
ETHTOOL_GDRVINFO	n/a
ETHTOOL_GREGS	n/a
ETHTOOL_GWOL	ETHTOOL_MSG_WOL_GET
ETHTOOL_SWOL	ETHTOOL_MSG_WOL_SET
ETHTOOL_GMSGLVL	ETHTOOL_MSG_DEBUG_GET
ETHTOOL_SMSGLVL	ETHTOOL_MSG_DEBUG_SET
ETHTOOL_NWAY_RST	n/a
ETHTOOL_GLINK	ETHTOOL_MSG_LINKSTATE_GET
ETHTOOL_GEEPROM	n/a
ETHTOOL_SEEPROM	n/a
ETHTOOL_GCOALESCE	ETHTOOL_MSG_COALESCE_GET
ETHTOOL_SCOALESCE	ETHTOOL_MSG_COALESCE_SET
ETHTOOL_GRINGPARAM	ETHTOOL_MSG_RINGS_GET
ETHTOOL_SRINGPARAM	ETHTOOL_MSG_RINGS_SET
ETHTOOL_GPAUSEPARAM	ETHTOOL_MSG_PAUSE_GET
ETHTOOL_SPAUSEPARAM	ETHTOOL_MSG_PAUSE_SET

continues on next page

Table 3 – continued from previous page

ioctl command	netlink command
ETHTOOL_GRXCSUM	ETHTOOL_MSG_FEATURES_GET
ETHTOOL_SRXCSUM	ETHTOOL_MSG_FEATURES_SET
ETHTOOL_GTXCSUM	ETHTOOL_MSG_FEATURES_GET
ETHTOOL_STXCSUM	ETHTOOL_MSG_FEATURES_SET
ETHTOOL_GSG	ETHTOOL_MSG_FEATURES_GET
ETHTOOL_SSG	ETHTOOL_MSG_FEATURES_SET
ETHTOOL_TEST	n/a
ETHTOOL_GSTRINGS	ETHTOOL_MSG_STRSET_GET
ETHTOOL_PHYS_ID	n/a
ETHTOOL_GSTATS	n/a
ETHTOOL_GTSO	ETHTOOL_MSG_FEATURES_GET
ETHTOOL_STSO	ETHTOOL_MSG_FEATURES_SET
ETHTOOL_GPERMADDR	rtnetlink RTM_GETLINK
ETHTOOL_GUFO	ETHTOOL_MSG_FEATURES_GET
ETHTOOL_SUFO	ETHTOOL_MSG_FEATURES_SET
ETHTOOL_GGSO	ETHTOOL_MSG_FEATURES_GET
ETHTOOL_SGSO	ETHTOOL_MSG_FEATURES_SET
ETHTOOL_GFLAGS	ETHTOOL_MSG_FEATURES_GET
ETHTOOL_SFLAGS	ETHTOOL_MSG_FEATURES_SET
ETHTOOL_GPFLAGS	ETHTOOL_MSG_PRIVFLAGS_GET
ETHTOOL_SPFLAGS	ETHTOOL_MSG_PRIVFLAGS_SET
ETHTOOL_GRXFH	n/a
ETHTOOL_SRXFH	n/a
ETHTOOL_GGRO	ETHTOOL_MSG_FEATURES_GET
ETHTOOL_SGRO	ETHTOOL_MSG_FEATURES_SET
ETHTOOL_GRXRINGS	n/a
ETHTOOL_GRXCLSLCNT	n/a
ETHTOOL_GRXCLSRULE	n/a
ETHTOOL_GRXCLSLALL	n/a
ETHTOOL_SRXCLSLDEL	n/a
ETHTOOL_SRXCLSLINS	n/a
ETHTOOL_FLASHDEV	n/a
ETHTOOL_RESET	n/a
ETHTOOL_SRXNTUPLE	n/a
ETHTOOL_GRXNTUPLE	n/a
ETHTOOL_GSSET_INFO	ETHTOOL_MSG_STRSET_GET
ETHTOOL_GRXFHINDIR	n/a
ETHTOOL_SRXFHINDIR	n/a
ETHTOOL_GFEATURES	ETHTOOL_MSG_FEATURES_GET
ETHTOOL_SFEATURES	ETHTOOL_MSG_FEATURES_SET
ETHTOOL_GCHANNELS	ETHTOOL_MSG_CHANNELS_GET
ETHTOOL_SCHANNELS	ETHTOOL_MSG_CHANNELS_SET
ETHTOOL_SET_DUMP	n/a
ETHTOOL_GET_DUMP_FLAG	n/a
ETHTOOL_GET_DUMP_DATA	n/a
ETHTOOL_GET_TS_INFO	ETHTOOL_MSG_TSINFO_GET
ETHTOOL_GMODULEINFO	ETHTOOL_MSG_MODULE_EEPROM_GET

continues on next page

Table 3 – continued from previous page

ioctl command	netlink command
ETHTOOL_GMODULEEEPROM	ETHTOOL_MSG_MODULE_EEPROM_GET
ETHTOOL_GEEE	ETHTOOL_MSG_EEE_GET
ETHTOOL_SEEE	ETHTOOL_MSG_EEE_SET
ETHTOOL_GRSSH	ETHTOOL_MSG_RSS_GET
ETHTOOL_SRSSH	n/a
ETHTOOL_GTUNABLE	n/a
ETHTOOL_STUNABLE	n/a
ETHTOOL_GPHYSTATS	n/a
ETHTOOL_PERQUEUE	n/a
ETHTOOL_GLINKSETTINGS	ETHTOOL_MSG_LINKINFO_GET ETHTOOL_MSG_LINKMODES_GET
ETHTOOL_SLINKSETTINGS	ETHTOOL_MSG_LINKINFO_SET ETHTOOL_MSG_LINKMODES_SET
ETHTOOL_PHY_GTUNABLE	n/a
ETHTOOL_PHY_STUNABLE	n/a
ETHTOOL_GFECPARAM	ETHTOOL_MSG_FEC_GET
ETHTOOL_SFECPARAM	ETHTOOL_MSG_FEC_SET
n/a	ETHTOOL_MSG_CABLE_TEST_ACT
n/a	ETHTOOL_MSG_CABLE_TEST_TDR_ACT
n/a	ETHTOOL_MSG_TUNNEL_INFO_GET
n/a	ETHTOOL_MSG_PHC_VCLOCKS_GET
n/a	ETHTOOL_MSG_MODULE_GET
n/a	ETHTOOL_MSG_MODULE_SET
n/a	ETHTOOL_MSG_PLCA_GET_CFG
n/a	ETHTOOL_MSG_PLCA_SET_CFG
n/a	ETHTOOL_MSG_PLCA_GET_STATUS
n/a	ETHTOOL_MSG_MM_GET
n/a	ETHTOOL_MSG_MM_SET

IEEE 802.15.4 DEVELOPER'S GUIDE

11.1 Introduction

The IEEE 802.15.4 working group focuses on standardization of the bottom two layers: Medium Access Control (MAC) and Physical access (PHY). And there are mainly two options available for upper layers:

- ZigBee - proprietary protocol from the ZigBee Alliance
- 6LoWPAN - IPv6 networking over low rate personal area networks

The goal of the Linux-wpan is to provide a complete implementation of the IEEE 802.15.4 and 6LoWPAN protocols. IEEE 802.15.4 is a stack of protocols for organizing Low-Rate Wireless Personal Area Networks.

The stack is composed of three main parts:

- IEEE 802.15.4 layer; We have chosen to use plain Berkeley socket API, the generic Linux networking stack to transfer IEEE 802.15.4 data messages and a special protocol over netlink for configuration/management
- MAC - provides access to shared channel and reliable data delivery
- PHY - represents device drivers

11.2 Socket API

```
int sd = socket(PF_IEEE802154, SOCK_DGRAM, 0);
```

The address family, socket addresses etc. are defined in the include/net/af_ieee802154.h header or in the special header in the userspace package (see either <https://linux-wpan.org/wpan-tools.html> or the git tree at <https://github.com/linux-wpan/wpan-tools>).

11.3 6LoWPAN Linux implementation

The IEEE 802.15.4 standard specifies an MTU of 127 bytes, yielding about 80 octets of actual MAC payload once security is turned on, on a wireless link with a link throughput of 250 kbps or less. The 6LoWPAN adaptation format [RFC4944] was specified to carry IPv6 datagrams over such constrained links, taking into account limited bandwidth, memory, or energy resources that are expected in applications such as wireless Sensor Networks. [RFC4944] defines a Mesh Addressing header to support sub-IP forwarding, a Fragmentation header to support the IPv6 minimum MTU requirement [RFC2460], and stateless header compression for IPv6 datagrams (LOWPAN_HC1 and LOWPAN_HC2) to reduce the relatively large IPv6 and UDP headers down to (in the best case) several bytes.

In September 2011 the standard update was published - [RFC6282]. It deprecates HC1 and HC2 compression and defines IPHC encoding format which is used in this Linux implementation.

All the code related to 6lowpan you may find in files: net/6lowpan/* and net/ieee802154/6lowpan/*

To setup a 6LoWPAN interface you need: 1. Add IEEE802.15.4 interface and set channel and PAN ID; 2. Add 6lowpan interface by command like: # ip link add link wpan0 name lowpan0 type lowpan 3. Bring up 'lowpan0' interface

11.4 Drivers

Like with WiFi, there are several types of devices implementing IEEE 802.15.4. 1) 'Hard-MAC'. The MAC layer is implemented in the device itself, the device exports a management (e.g. MLME) and data API. 2) 'SoftMAC' or just radio. These types of devices are just radio transceivers possibly with some kinds of acceleration like automatic CRC computation and comparison, automagic ACK handling, address matching, etc.

Those types of devices require different approach to be hooked into Linux kernel.

11.4.1 HardMAC

See the header include/net/ieee802154_netdev.h. You have to implement Linux net_device, with .type = ARPHRD_IEEE802154. Data is exchanged with socket family code via plain sk_buffs. On skb reception skb->cb must contain additional info as described in the struct ieee802154_mac_cb. During packet transmission the skb->cb is used to provide additional data to device's header_ops->create function. Be aware that this data can be overridden later (when socket code submits skb to qdisc), so if you need something from that cb later, you should store info in the skb->data on your own.

To hook the MLME interface you have to populate the ml_priv field of your net_device with a pointer to struct ieee802154_mlme_ops instance. The fields assoc_req, assoc_resp, disassoc_req, start_req, and scan_req are optional. All other fields are required.

11.4.2 SoftMAC

The MAC is the middle layer in the IEEE 802.15.4 Linux stack. This moment it provides interface for drivers registration and management of slave interfaces.

NOTE: Currently the only monitor device type is supported - it's IEEE 802.15.4 stack interface for network sniffers (e.g. Wireshark).

This layer is going to be extended soon.

See header include/net/mac802154.h and several drivers in drivers/net/ieee802154/.

11.4.3 Fake drivers

In addition there is a driver available which simulates a real device with SoftMAC (fakelb - IEEE 802.15.4 loopback driver) interface. This option provides a possibility to test and debug the stack without usage of real hardware.

11.5 Device drivers API

The include/net/mac802154.h defines following functions:

```
struct ieee802154_dev *ieee802154_alloc_device(size_t priv_size, struct ieee802154_ops
                                              *ops)
```

Allocation of IEEE 802.15.4 compatible device.

```
void ieee802154_free_device(struct ieee802154_dev *dev)
```

Freeing allocated device.

```
int ieee802154_register_device(struct ieee802154_dev *dev)
```

Register PHY in the system.

```
void ieee802154_unregister_device(struct ieee802154_dev *dev)
```

Freeing registered PHY.

```
void ieee802154_rx_irqsafe(struct ieee802154_hw *hw, struct sk_buff *skb, u8 lqi)
```

Telling 802.15.4 module there is a new received frame in the skb with the RF Link Quality Indicator (LQI) from the hardware device.

```
void ieee802154_xmit_complete(struct ieee802154_hw *hw, struct sk_buff *skb, bool
                               ifs_handling)
```

Telling 802.15.4 module the frame in the skb is or going to be transmitted through the hardware device

The device driver must implement the following callbacks in the IEEE 802.15.4 operations structure at least:

```
struct ieee802154_ops {  
    ...  
    int     (*start)(struct ieee802154_hw *hw);  
    void    (*stop)(struct ieee802154_hw *hw);  
    ...  
    int     (*xmit_async)(struct ieee802154_hw *hw, struct sk_buff *skb);  
    int     (*ed)(struct ieee802154_hw *hw, u8 *level);  
    int     (*set_channel)(struct ieee802154_hw *hw, u8 page, u8 channel);  
    ...  
};
```

int **start**(struct ieee802154_hw *hw)

Handler that 802.15.4 module calls for the hardware device initialization.

void **stop**(struct ieee802154_hw *hw)

Handler that 802.15.4 module calls for the hardware device cleanup.

int **xmit_async**(struct ieee802154_hw *hw, struct *sk_buff* *skb)

Handler that 802.15.4 module calls for each frame in the skb going to be transmitted through the hardware device.

int **ed**(struct ieee802154_hw *hw, u8 *level)

Handler that 802.15.4 module calls for Energy Detection from the hardware device.

int **set_channel**(struct ieee802154_hw *hw, u8 page, u8 channel)

Set radio for listening on specific channel of the hardware device.

Moreover IEEE 802.15.4 device operations structure should be filled.

J1939 DOCUMENTATION

12.1 Overview / What Is J1939

SAE J1939 defines a higher layer protocol on CAN. It implements a more sophisticated addressing scheme and extends the maximum packet size above 8 bytes. Several derived specifications exist, which differ from the original J1939 on the application level, like MilCAN A, NMEA2000, and especially ISO-11783 (ISOBUS). This last one specifies the so-called ETP (Extended Transport Protocol), which has been included in this implementation. This results in a maximum packet size of $((2^{24} - 1) * 7)$ bytes == 111 MiB.

12.1.1 Specifications used

- SAE J1939-21 : data link layer
- SAE J1939-81 : network management
- ISO 11783-6 : Virtual Terminal (Extended Transport Protocol)

12.2 Motivation

Given the fact there's something like SocketCAN with an API similar to BSD sockets, we found some reasons to justify a kernel implementation for the addressing and transport methods used by J1939.

- **Addressing:** when a process on an ECU communicates via J1939, it should not necessarily know its source address. Although, at least one process per ECU should know the source address. Other processes should be able to reuse that address. This way, address parameters for different processes cooperating for the same ECU, are not duplicated. This way of working is closely related to the UNIX concept, where programs do just one thing and do it well.
- **Dynamic addressing:** Address Claiming in J1939 is time critical. Furthermore, data transport should be handled properly during the address negotiation. Putting this functionality in the kernel eliminates it as a requirement for every user space process that communicates via J1939. This results in a consistent J1939 bus with proper addressing.
- **Transport:** both TP & ETP reuse some PGNs to relay big packets over them. Different processes may thus use the same TP & ETP PGNs without actually knowing it. The individual TP & ETP sessions must be serialized (synchronized) between different processes.

The kernel solves this problem properly and eliminates the serialization (synchronization) as a requirement for *_every_* user space process that communicates via J1939.

J1939 defines some other features (relaying, gateway, fast packet transport, ...). In-kernel code for these would not contribute to protocol stability. Therefore, these parts are left to user space.

The J1939 sockets operate on CAN network devices (see SocketCAN). Any J1939 user space library operating on CAN raw sockets will still operate properly. Since such a library does not communicate with the in-kernel implementation, care must be taken that these two do not interfere. In practice, this means they cannot share ECU addresses. A single ECU (or virtual ECU) address is used by the library exclusively, or by the in-kernel system exclusively.

12.3 J1939 concepts

12.3.1 PGN

The J1939 protocol uses the 29-bit CAN identifier with the following structure:

29 bit CAN-ID		
Bit positions within the CAN-ID		
28 ... 26	25 ... 8	7 ... 0
Priority	PGN	SA (Source Address)

The PGN (Parameter Group Number) is a number to identify a packet. The PGN is composed as follows:

PGN			
Bit positions within the CAN-ID			
25	24	23 ... 16	15 ... 8
R (Reserved)	DP (Data Page)	PF (PDU Format)	PS (PDU Specific)

In J1939-21 distinction is made between PDU1 format (where PF < 240) and PDU2 format (where PF >= 240). Furthermore, when using the PDU2 format, the PS-field contains a so-called Group Extension, which is part of the PGN. When using PDU2 format, the Group Extension is set in the PS-field.

PDU1 Format (specific) (peer to peer)	
Bit positions within the CAN-ID	
23 ... 16	15 ... 8
00h ... EFh	DA (Destination address)

PDU2 Format (global) (broadcast)	
Bit positions within the CAN-ID	
23 ... 16	15 ... 8
F0h ... FFh	GE (Group Extension)

On the other hand, when using PDU1 format, the PS-field contains a so-called Destination Address, which is *_not_* part of the PGN. When communicating a PGN from user space to kernel

(or vice versa) and PDU2 format is used, the PS-field of the PGN shall be set to zero. The Destination Address shall be set elsewhere.

Regarding PGN mapping to 29-bit CAN identifier, the Destination Address shall be get/set from/to the appropriate bits of the identifier by the kernel.

12.3.2 Addressing

Both static and dynamic addressing methods can be used.

For static addresses, no extra checks are made by the kernel and provided addresses are considered right. This responsibility is for the OEM or system integrator.

For dynamic addressing, so-called Address Claiming, extra support is foreseen in the kernel. In J1939 any ECU is known by its 64-bit NAME. At the moment of a successful address claim, the kernel keeps track of both NAME and source address being claimed. This serves as a base for filter schemes. By default, packets with a destination that is not locally will be rejected.

Mixed mode packets (from a static to a dynamic address or vice versa) are allowed. The BSD sockets define separate API calls for getting/setting the local & remote address and are applicable for J1939 sockets.

12.3.3 Filtering

J1939 defines white list filters per socket that a user can set in order to receive a subset of the J1939 traffic. Filtering can be based on:

- SA
- SOURCE_NAME
- PGN

When multiple filters are in place for a single socket, and a packet comes in that matches several of those filters, the packet is only received once for that socket.

12.4 How to Use J1939

12.4.1 API Calls

On CAN, you first need to open a socket for communicating over a CAN network. To use J1939, `#include <linux/can/j1939.h>`. From there, `<linux/can.h>` will be included too. To open a socket, use:

```
s = socket(PF_CAN, SOCK_DGRAM, CAN_J1939);
```

J1939 does use SOCK_DGRAM sockets. In the J1939 specification, connections are mentioned in the context of transport protocol sessions. These still deliver packets to the other end (using several CAN packets). SOCK_STREAM is not supported.

After the successful creation of the socket, you would normally use the `bind(2)` and/or `connect(2)` system call to bind the socket to a CAN interface. After binding and/or connecting the socket, you can `read(2)` and `write(2)` from/to the socket or use `send(2)`, `sendto(2)`,

`sendmsg(2)` and the `recv*`(*i*) counterpart operations on the socket as usual. There are also J1939 specific socket options described below.

In order to send data, a `bind(2)` must have been successful. `bind(2)` assigns a local address to a socket.

Different from CAN is that the payload data is just the data that get sends, without its header info. The header info is derived from the `sockaddr` supplied to `bind(2)`, `connect(2)`, `sendto(2)` and `recvfrom(2)`. A `write(2)` with size 4 will result in a packet with 4 bytes.

The `sockaddr` structure has extensions for use with J1939 as specified below:

```
struct sockaddr_can {
    sa_family_t can_family;
    int         can_ifindex;
    union {
        struct {
            __u64 name;
            /* pgn:
             * 8 bit: PS in PDU2 case, else 0
             * 8 bit: PF
             * 1 bit: DP
             * 1 bit: reserved
             */
            __u32 pgn;
            __u8  addr;
        } j1939;
    } can_addr;
}
```

`can_family` & `can_ifindex` serve the same purpose as for other SocketCAN sockets.

`can_addr.j1939.pgn` specifies the PGN (max 0x3ffff). Individual bits are specified above.

`can_addr.j1939.name` contains the 64-bit J1939 NAME.

`can_addr.j1939.addr` contains the address.

The `bind(2)` system call assigns the local address, i.e. the source address when sending packages. If a PGN during `bind(2)` is set, it's used as a RX filter. I.e. only packets with a matching PGN are received. If an ADDR or NAME is set it is used as a receive filter, too. It will match the destination NAME or ADDR of the incoming packet. The NAME filter will work only if appropriate Address Claiming for this name was done on the CAN bus and registered/cached by the kernel.

On the other hand `connect(2)` assigns the remote address, i.e. the destination address. The PGN from `connect(2)` is used as the default PGN when sending packets. If ADDR or NAME is set it will be used as the default destination ADDR or NAME. Further a set ADDR or NAME during `connect(2)` is used as a receive filter. It will match the source NAME or ADDR of the incoming packet.

Both `write(2)` and `send(2)` will send a packet with local address from `bind(2)` and the remote address from `connect(2)`. Use `sendto(2)` to overwrite the destination address.

If `can_addr.j1939.name` is set ($\neq 0$) the NAME is looked up by the kernel and the corresponding ADDR is used. If `can_addr.j1939.name` is not set ($\neq 0$), `can_addr.j1939.addr` is used.

When creating a socket, reasonable defaults are set. Some options can be modified with `setsockopt(2)` & `getsockopt(2)`.

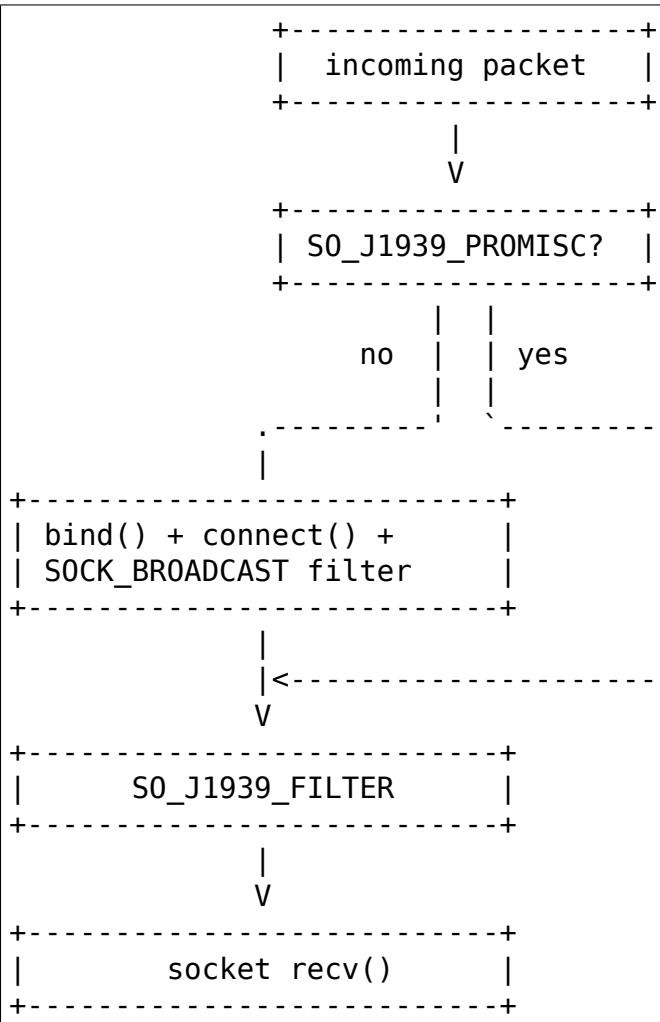
RX path related options:

- `SO_J1939_FILTER` - configure array of filters
- `SO_J1939_PROMISC` - disable filters set by `bind(2)` and `connect(2)`

By default no broadcast packets can be send or received. To enable sending or receiving broadcast packets use the socket option `SO_BROADCAST`:

```
int value = 1;
setsockopt(sock, SOL_SOCKET, SO_BROADCAST, &value, sizeof(value));
```

The following diagram illustrates the RX path:



TX path related options: `SO_J1939_SEND_PRIO` - change default send priority for the socket

Message Flags during send() and Related System Calls

send(2), sendto(2) and sendmsg(2) take a 'flags' argument. Currently supported flags are:

- MSG_DONTWAIT, i.e. non-blocking operation.

recvmsg(2)

In most cases recvmsg(2) is needed if you want to extract more information than recvfrom(2) can provide. For example package priority and timestamp. The Destination Address, name and packet priority (if applicable) are attached to the msghdr in the recvmsg(2) call. They can be extracted using cmsg(3) macros, with `cmsg_level == SOL_J1939 && cmsg_type == SCM_J1939_DEST_ADDR`, `SCM_J1939_DEST_NAME` or `SCM_J1939_PRIO`. The returned data is a `uint8_t` for priority and `dst_addr`, and `uint64_t` for `dst_name`.

```
uint8_t priority, dst_addr;
uint64_t dst_name;

for (cmsg = CMSG_FIRSTHDR(&msg); cmsg; cmsg = CMSG_NXTHDR(&msg, cmsg)) {
    switch (cmsg->cmsg_level) {
        case SOL_CAN_J1939:
            if (cmsg->cmsg_type == SCM_J1939_DEST_ADDR)
                dst_addr = *CMSG_DATA(cmsg);
            else if (cmsg->cmsg_type == SCM_J1939_DEST_NAME)
                memcpy(&dst_name, CMSG_DATA(cmsg), cmsg->cmsg_len - ↵
                    CMSG_LEN(0));
            else if (cmsg->cmsg_type == SCM_J1939_PRIO)
                priority = *CMSG_DATA(cmsg);
            break;
    }
}
```

12.4.2 Dynamic Addressing

Distinction has to be made between using the claimed address and doing an address claim. To use an already claimed address, one has to fill in the `j1939.name` member and provide it to bind(2). If the name had claimed an address earlier, all further messages being sent will use that address. And the `j1939.addr` member will be ignored.

An exception on this is PGN 0x0ee00. This is the "Address Claim/Cannot Claim Address" message and the kernel will use the `j1939.addr` member for that PGN if necessary.

To claim an address following code example can be used:

```
struct sockaddr_can baddr = {
    .can_family = AF_CAN,
    .can_addr.j1939 = {
        .name = name,
        .addr = J1939_IDLE_ADDR,
        .pgn = J1939_NO_PGN,      /* to disable bind() rx filter for PGN */
    }
}
```

```

        },
        .can_ifindex = if_nametoindex("can0"),
};

bind(sock, (struct sockaddr *)&baddr, sizeof(baddr));

/* for Address Claiming broadcast must be allowed */
int value = 1;
setsockopt(sock, SOL_SOCKET, SO_BROADCAST, &value, sizeof(value));

/* configured advanced RX filter with PGN needed for Address Claiming */
const struct j1939_filter filt[] = {
{
    .pgn = J1939_PGN_ADDRESS CLAIMED,
    .pgn_mask = J1939_PGN_PDU1_MAX,
},
{
    .pgn = J1939_PGN_REQUEST,
    .pgn_mask = J1939_PGN_PDU1_MAX,
},
{
    .pgn = J1939_PGN_ADDRESS_COMMANDED,
    .pgn_mask = J1939_PGN_MAX,
},
};

setsockopt(sock, SOL_CAN_J1939, SO_J1939_FILTER, &filt, sizeof(filt));

uint64_t dat = htole64(name);
const struct sockaddr_can saddr = {
    .can_family = AF_CAN,
    .can_addr.j1939 = {
        .pgn = J1939_PGN_ADDRESS CLAIMED,
        .addr = J1939_NO_ADDR,
    },
};

/* Afterwards do a sendto(2) with data set to the NAME (LittleEndian). If the
 * NAME provided, does not match the j1939.name provided to bind(2), EPROTO
 * will be returned.
 */
sendto(sock, dat, sizeof(dat), 0, (const struct sockaddr *)&saddr, sizeof(saddr));

```

If no-one else contests the address claim within 250ms after transmission, the kernel marks the NAME-SA assignment as valid. The valid assignment will be kept among other valid NAME-SA assignments. From that point, any socket bound to the NAME can send packets.

If another ECU claims the address, the kernel will mark the NAME-SA expired. No socket bound to the NAME can send packets (other than address claims). To claim another address, some socket bound to NAME, must `bind(2)` again, but with only `j1939.addr` changed to the new SA, and must then send a valid address claim packet. This restarts the state machine in the kernel (and any other participant on the bus) for this NAME.

can-utils also include the j1939acd tool, so it can be used as code example or as default Address Claiming daemon.

12.4.3 Send Examples

Static Addressing

This example will send a PGN (0x12300) from SA 0x20 to DA 0x30.

Bind:

```
struct sockaddr_can baddr = {
    .can_family = AF_CAN,
    .can_addr.j1939 = {
        .name = J1939_NO_NAME,
        .addr = 0x20,
        .pgn = J1939_NO_PGN,
    },
    .can_ifindex = if_nametoindex("can0"),
};

bind(sock, (struct sockaddr *)&baddr, sizeof(baddr));
```

Now, the socket 'sock' is bound to the SA 0x20. Since no `connect(2)` was called, at this point we can use only `sendto(2)` or `sendmsg(2)`.

Send:

```
const struct sockaddr_can saddr = {
    .can_family = AF_CAN,
    .can_addr.j1939 = {
        .name = J1939_NO_NAME;
        .addr = 0x30,
        .pgn = 0x12300,
    },
};

sendto(sock, dat, sizeof(dat), 0, (const struct sockaddr *)&saddr, sizeof(saddr));
```

LINUX NETWORKING AND NETWORK DEVICES APIs

13.1 Linux Networking

13.1.1 Networking Base Types

enum **sock_type**

 Socket types

Constants

SOCK_STREAM

 stream (connection) socket

SOCK_DGRAM

 datagram (conn.less) socket

SOCK_RAW

 raw socket

SOCK_RDM

 reliably-delivered message

SOCK_SEQPACKET

 sequential packet socket

SOCK_DCCP

 Datagram Congestion Control Protocol socket

SOCK_PACKET

 linux specific way of getting packets at the dev level. For writing rarp and other similar things on the user level.

Description

When adding some new socket type please grep ARCH_HAS_SOCKET_TYPE include/asm-*/socket.h, at least MIPS overrides this enum for binary compat reasons.

enum **sock_shutdown_cmd**

 Shutdown types

Constants

SHUT_RD

 shutdown receptions

SHUT_WR

shutdown transmissions

SHUT_RDWR

shutdown receptions/transmissions

struct socket

general BSD socket

Definition:

```
struct socket {  
    socket_state state;  
    short type;  
    unsigned long          flags;  
    struct file           *file;  
    struct sock            *sk;  
    const struct proto_ops *ops;  
    struct socket_wq      wq;  
};
```

Members

state

socket state (SS_CONNECTED, etc)

type

socket type (SOCK_STREAM, etc)

flags

socket flags (SOCK_NOSPACE, etc)

file

File back pointer for gc

sk

internal networking protocol agnostic socket representation

ops

protocol specific socket operations

wq

wait queue for several uses

13.1.2 Socket Buffer Functions

unsigned int **skb_frag_size**(const skb_frag_t *frag)

Returns the size of a skb fragment

Parameters

const skb_frag_t *frag

skb fragment

void **skb_frag_size_set**(skb_frag_t *frag, unsigned int size)

Sets the size of a skb fragment

Parameters

skb_frag_t *frag
 skb fragment

unsigned int size
 size of fragment

void skb_frag_size_add(skb_frag_t *frag, int delta)
 Increments the size of a skb fragment by **delta**

Parameters

skb_frag_t *frag
 skb fragment

int delta
 value to add

void skb_frag_size_sub(skb_frag_t *frag, int delta)
 Decrement the size of a skb fragment by **delta**

Parameters

skb_frag_t *frag
 skb fragment

int delta
 value to subtract

bool skb_frag_must_loop(struct page *p)
 Test if p is a high memory page

Parameters

struct page *p
 fragment's page

skb_frag_foreach_page

skb_frag_foreach_page (f, f_off, f_len, p, p_off, p_len, copied)
 loop over pages in a fragment

Parameters

f
 skb frag to operate on

f_off
 offset from start of f->bv_page

f_len
 length from f_off to loop over

p
 (temp var) current page

p_off
 (temp var) offset from start of current page, non-zero only on first page.

p_len

(temp var) length in current page, < PAGE_SIZE only on first and last page.

copied

(temp var) length so far, excluding current p_len.

A fragment can hold a compound page, in which case per-page operations, notably kmap_atomic, must be called for each regular page.

struct skb_shared_hwtstamps

hardware time stamps

Definition:

```
struct skb_shared_hwtstamps {
    union {
        ktime_t hwtstamp;
        void *netdev_data;
    };
};
```

Members**{unnamed_union}**

anonymous

hwtstamp

hardware time stamp transformed into duration since arbitrary point in time

netdev_data

address/cookie of network device driver used as reference to actual hardware time stamp

Description

Software time stamps generated by ktime_get_real() are stored in skb->tstamp.

hwtstamps can only be compared against other hwtstamps from the same device.

This structure is attached to packets as part of the skb_shared_info. Use skb_hwtstamps() to get a pointer.

struct sk_buff

socket buffer

Definition:

```
struct sk_buff {
    union {
        struct {
            struct sk_buff      *next;
            struct sk_buff      *prev;
            union {
                struct net_device *dev;
                unsigned long     dev_scratch;
            };
        };
        struct rb_node       rbnоде;
        struct list_head     list;
    };
};
```

```

        struct llist_node          ll_node;
};

union {
    struct sock              *sk;
    int ip_defrag_offset;
};

union {
    ktime_t tstamp;
    u64 skb_mstamp_ns;
};

char cb[48] ;
union {
    struct {
        unsigned long _skb_refdst;
        void (*destructor)(struct sk_buff *skb);
    };
    struct list_head          tcp_tsorited_anchor;
#endif CONFIG_NET_SOCK_MSG;
    unsigned long             _sk_redir;
#endif;
};

#if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE);
    unsigned long             _nfct;
#endif;
    unsigned int               len, data_len;
    __u16 mac_len, hdr_len;
    __u16 queue_mapping;
#endifdef __BIG_ENDIAN_BITFIELD;
#define CLONED_MASK      (1 << 7);
#else;
#define CLONED_MASK      1;
#endif;
#define CLONED_OFFSET      offsetof(struct sk_buff, __cloned_offset);
    __u8 cloned:1,nohdr:1,fclone:2,peeked:1,head_frag:1,pfmemalloc:1, pp_
    →recycle:1;
#endifdef CONFIG_SKB_EXTENSIONS;
    __u8 active_extensions;
#endif;
    __u8 pkt_type:3;
    __u8 ignore_df:1;
    __u8 dst_pending_confirm:1;
    __u8 ip_summed:2;
    __u8 ooo_okay:1;
    __u8 mono_delivery_time:1;
#endifdef CONFIG_NET_XGRESS;
    __u8 tc_at_ingress:1;
    __u8 tc_skip_classify:1;
#endif;
    __u8 remcsum_offload:1;
    __u8 csum_complete_sw:1;

```

```

__u8 csum_level:2;
__u8 inner_protocol_type:1;
__u8 l4_hash:1;
__u8 sw_hash:1;
#endif CONFIG_WIRELESS;
__u8 wifi_acked_valid:1;
__u8 wifi_acked:1;
#endif;
__u8 no_fcs:1;
__u8 encapsulation:1;
__u8 encap_hdr_csum:1;
__u8 csum_valid:1;
#endif CONFIG_IPV6_NDISC_NODETYPE;
__u8 ndisc_nodetype:2;
#endif;
#if IS_ENABLED(CONFIG_IP_VS);
__u8 ipvs_property:1;
#endif;
#if IS_ENABLED(CONFIG_NETFILTER_XT_TARGET_TRACE) || IS_ENABLED(CONFIG_NF_TABLES);
__u8 nf_trace:1;
#endif;
#endif CONFIG_NET_SWITCHDEV;
__u8 offload_fwd_mark:1;
__u8 offload_l3_fwd_mark:1;
#endif;
__u8 redirected:1;
#endif CONFIG_NET_REDIRECT;
__u8 from_ingress:1;
#endif;
#endif CONFIG_NETFILTER_SKIP_EGRESS;
__u8 nf_skip_egress:1;
#endif;
#endif CONFIG_TLS_DEVICE;
__u8 decrypted:1;
#endif;
__u8 slow_gro:1;
#endif IS_ENABLED(CONFIG_IP_SCTP);
__u8 csum_not_inet:1;
#endif;
#endif defined(CONFIG_NET_SCHED) || defined(CONFIG_NET_XGRESS);
__u16 tc_index;
#endif;
u16 alloc_cpu;
union {
    __wsum csum;
    struct {
        __u16 csum_start;
        __u16 csum_offset;
    };
};

```

```

};

__u32 priority;
int skb_iif;
__u32 hash;
union {
    u32 vlan_all;
    struct {
        __be16 vlan_proto;
        __u16 vlan_tci;
    };
};

#if defined(CONFIG_NET_RX_BUSY_POLL) || defined(CONFIG_XPS);
union {
    unsigned int      napi_id;
    unsigned int      sender_cpu;
};
#endif;
#ifndef CONFIG_NETWORK_SECMARK;
    __u32 secmark;
#endif;
union {
    __u32 mark;
    __u32 reserved_tailroom;
};
union {
    __be16 inner_protocol;
    __u8 inner_ipproto;
};
__u16 inner_transport_header;
__u16 inner_network_header;
__u16 inner_mac_header;
__be16 protocol;
__u16 transport_header;
__u16 network_header;
__u16 mac_header;
#endif;
#ifdef CONFIG_KCOV;
    u64 kcov_handle;
#endif;
sk_buff_data_t tail;
sk_buff_data_t end;
unsigned char          *head, *data;
unsigned int           truesize;
refcount_t users;
#endif;
#ifdef CONFIG_SKB_EXTENSIONS;
    struct skb_ext      *extensions;
#endif;
};

}

```

Members

{unnamed_union}

anonymous

{unnamed_struct}

anonymous

next

Next buffer in list

prev

Previous buffer in list

{unnamed_union}

anonymous

dev

Device we arrived on/are leaving by

dev_scratch

(aka **dev**) alternate use of **dev** when **dev** would be NULL

rbnode

RB tree node, alternative to next/prev for netem/tcp

list

queue head

ll_node

anchor in an llist (eg socket defer_list)

{unnamed_union}

anonymous

sk

Socket we are owned by

ip_defrag_offset

(aka **sk**) alternate use of **sk**, used in fragmentation management

{unnamed_union}

anonymous

tstamp

Time we arrived/left

skb_mstamp_ns

(aka **tstamp**) earliest departure time; start point for retransmit timer

cb

Control buffer. Free for use by every layer. Put private vars here

{unnamed_union}

anonymous

{unnamed_struct}

anonymous

_skb_refdst

destination entry (with norefcount bit)

destructor

Destruct function

tcp_tsorted_anchor

list structure for TCP (tp->tsorted_sent_queue)

_sk_redir

socket redirection information for skmsg

_nfct

Associated connection, if any (with nfctinfo bits)

len

Length of actual data

data_len

Data length

mac_len

Length of link layer header

hdr_len

writable header length of cloned skb

queue_mapping

Queue mapping for multiqueue devices

cloned

Head may be cloned (check refcnt to be sure)

nohdr

Payload reference only, must not modify header

fclone

skbuff clone status

peeked

this packet has been seen already, so stats have been done for it, don't do them again

head_frag

skb was allocated from page fragments, not allocated by kmalloc() or vmalloc().

pfmemalloc

skbuff was allocated from PFMEMALLOC reserves

pp_recycle

mark the packet for recycling instead of freeing (implies page_pool support on driver)

active_extensions

active extensions (skb_ext_id types)

pkt_type

Packet class

ignore_df

allow local fragmentation

dst_pending_confirm

need to confirm neighbour

ip_summed

Driver fed us an IP checksum

ooo_okay

allow the mapping of a socket to a queue to be changed

mono_delivery_time

When set, skb->tstamp has the delivery_time in mono clock base (i.e. EDT). Otherwise, the skb->tstamp has the (rcv) timestamp at ingress and delivery_time at egress.

tc_at_ingress

used within tc_classify to distinguish in/egress

tc_skip_classify

do not classify packet. set by IFB device

remcsum_offload

remote checksum offload is enabled

csum_complete_sw

checksum was completed by software

csum_level

indicates the number of consecutive checksums found in the packet minus one that have been verified as CHECKSUM_UNNECESSARY (max 3)

inner_protocol_type

whether the inner protocol is ENCAP_TYPE_ETHER or ENCAP_TYPE_IPPROTO

l4_hash

indicate hash is a canonical 4-tuple hash over transport ports.

sw_hash

indicates hash was computed in software stack

wifi_acked_valid

wifi_acked was set

wifi_acked

whether frame was acked on wifi or not

no_fcs

Request NIC to treat last 4 bytes as Ethernet FCS

encapsulation

indicates the inner headers in the skbuff are valid

encap_hdr_csum

software checksum is needed

csum_valid

checksum is already valid

ndisc_nodetype

router type (from link layer)

ipvs_property

skbuff is owned by ipvs

nf_trace

netfilter packet trace flag

offload_fwd_mark

Packet was L2-forwarded in hardware

offload_l3_fwd_mark

Packet was L3-forwarded in hardware

redirected

packet was redirected by packet classifier

from_ingress

packet was redirected from the ingress path

nf_skip_egress

packet shall skip nf egress - see netfilter_netdev.h

decrypted

Decrypted SKB

slow_gro

state present at GRO time, slower prepare step required

csum_not_inet

use CRC32c to resolve CHECKSUM_PARTIAL

tc_index

Traffic control index

alloc_cpu

CPU which did the skb allocation.

{unnamed_union}

anonymous

csum

Checksum (must include start/offset pair)

{unnamed_struct}

anonymous

csum_start

Offset from skb->head where checksumming should start

csum_offset

Offset from csum_start where checksum should be stored

priority

Packet queueing priority

skb_iif

ifindex of device we arrived on

hash

the packet hash

{unnamed_union}

anonymous

vlan_all

vlan fields (proto & tci)

{unnamed_struct}

anonymous

vlan_proto
 vlan encapsulation protocol

vlan_tci
 vlan tag control information

{unnamed_union}
 anonymous

napi_id
 id of the NAPI struct this skb came from

sender_cpu
 (aka **napi_id**) source CPU in XPS

secmark
 security marking

{unnamed_union}
 anonymous

mark
 Generic packet mark

reserved_tailroom
 (aka **mark**) number of bytes of free space available at the tail of an `sk_buff`

{unnamed_union}
 anonymous

inner_protocol
 Protocol (encapsulation)

inner_ipproto
 (aka **inner_protocol**) stores ipproto when `skb->inner_protocol_type == EN-
CAP_TYPE IPPROTO;`

inner_transport_header
 Inner transport layer header (encapsulation)

inner_network_header
 Network layer header (encapsulation)

inner_mac_header
 Link layer header (encapsulation)

protocol
 Packet protocol from driver

transport_header
 Transport layer header

network_header
 Network layer header

mac_header
 Link layer header

kcov_handle
 KCOV remote handle for remote coverage collection

tail
 Tail pointer
end
 End pointer
head
 Head of buffer
data
 Data head pointer

truesize
 Buffer size

users
 User count - see {datagram,tcp}.c

extensions
 allocated extensions, valid if active_extensions is nonzero
bool skb_pfmemalloc(const struct sk_buff *skb)
 Test if the skb was allocated from PFMEMALLOC reserves

Parameters

const struct sk_buff *skb
 buffer
struct dst_entry *skb_dst(const struct sk_buff *skb)
 returns skb dst_entry

Parameters

const struct sk_buff *skb
 buffer

Description

Returns skb dst_entry, regardless of reference taken or not.

void skb_dst_set(struct sk_buff *skb, struct dst_entry *dst)
 sets skb dst

Parameters

struct sk_buff *skb
 buffer

struct dst_entry *dst
 dst entry

Description

Sets skb dst, assuming a reference was taken on dst and should be released by skb_dst_drop()

void skb_dst_set_noref(struct sk_buff *skb, struct dst_entry *dst)
 sets skb dst, hopefully, without taking reference

Parameters

```
struct sk_buff *skb
    buffer

struct dst_entry *dst
    dst entry
```

Description

Sets skb dst, assuming a reference was not taken on dst. If dst entry is cached, we do not take reference and dst_release will be avoided by refdst_drop. If dst entry is not cached, we take reference, so that last dst_release can destroy the dst immediately.

```
bool skb_dst_is_noref(const struct sk_buff *skb)
```

Test if skb dst isn't refcounted

Parameters

```
const struct sk_buff *skb
    buffer
```

```
struct rtable *skb_rtable(const struct sk_buff *skb)
```

Returns the skb rtable

Parameters

```
const struct sk_buff *skb
    buffer
```

```
unsigned int skb_napi_id(const struct sk_buff *skb)
```

Returns the skb's NAPI id

Parameters

```
const struct sk_buff *skb
    buffer
```

```
bool skb_unref(struct sk_buff *skb)
```

decrement the skb's reference count

Parameters

```
struct sk_buff *skb
    buffer
```

Description

Returns true if we can free the skb.

```
void kfree_skb(struct sk_buff *skb)
```

free an sk_buff with 'NOT_SPECIFIED' reason

Parameters

```
struct sk_buff *skb
    buffer to free
```

```
struct sk_buff *alloc_skb(unsigned int size, gfp_t priority)
```

allocate a network buffer

Parameters

unsigned int size
size to allocate

gfp_t priority
allocation mask

Description

This function is a convenient wrapper around `_alloc_skb()`.

bool skb_fclone_busy(const struct sock *sk, const struct sk_buff *skb)
check if fclone is busy

Parameters

const struct sock *sk
socket

const struct sk_buff *skb
buffer

Description

Returns true if skb is a fast clone, and its clone is not freed. Some drivers call `skb_orphan()` in their `ndo_start_xmit()`, so we also check that didn't happen.

struct sk_buff *alloc_skb_fclone(unsigned int size, gfp_t priority)
allocate a network buffer from fclone cache

Parameters

unsigned int size
size to allocate

gfp_t priority
allocation mask

Description

This function is a convenient wrapper around `_alloc_skb()`.

int skb_pad(struct sk_buff *skb, int pad)
zero pad the tail of an skb

Parameters

struct sk_buff *skb
buffer to pad

int pad
space to pad

Ensure that a buffer is followed by a padding area that is zero filled. Used by network drivers which may DMA or transfer data beyond the buffer end onto the wire.

May return error in out of memory cases. The skb is freed on error.

int skb_queue_empty(const struct sk_buff_head *list)
check if a queue is empty

Parameters

const struct sk_buff_head *list
queue head

Returns true if the queue is empty, false otherwise.

bool skb_queue_empty_lockless(const struct sk_buff_head *list)
check if a queue is empty

Parameters

const struct sk_buff_head *list
queue head

Returns true if the queue is empty, false otherwise. This variant can be used in lockless contexts.

bool skb_queue_is_last(const struct sk_buff_head *list, const struct sk_buff *skb)
check if skb is the last entry in the queue

Parameters

const struct sk_buff_head *list
queue head

const struct sk_buff *skb
buffer

Returns true if **skb** is the last buffer on the list.

bool skb_queue_is_first(const struct sk_buff_head *list, const struct sk_buff *skb)
check if skb is the first entry in the queue

Parameters

const struct sk_buff_head *list
queue head

const struct sk_buff *skb
buffer

Returns true if **skb** is the first buffer on the list.

struct sk_buff *skb_queue_next(const struct sk_buff_head *list, const struct sk_buff *skb)
return the next packet in the queue

Parameters

const struct sk_buff_head *list
queue head

const struct sk_buff *skb
current buffer

Return the next packet in **list** after **skb**. It is only valid to call this if **skb_queue_is_last()** evaluates to false.

struct sk_buff *skb_queue_prev(const struct sk_buff_head *list, const struct sk_buff *skb)
return the prev packet in the queue

Parameters

const struct sk_buff_head *list

queue head

const struct sk_buff *skb

current buffer

Return the prev packet in **list** before **skb**. It is only valid to call this if [*skb_queue_is_first\(\)*](#) evaluates to false.

struct sk_buff *skb_get(struct sk_buff *skb)

reference buffer

Parameters

struct sk_buff *skb

buffer to reference

Makes another reference to a socket buffer and returns a pointer to the buffer.

int skb_cloned(const struct sk_buff *skb)

is the buffer a clone

Parameters

const struct sk_buff *skb

buffer to check

Returns true if the buffer was generated with [*skb_clone\(\)*](#) and is one of multiple shared copies of the buffer. Cloned buffers are shared data so must not be written to under normal circumstances.

int skb_header_cloned(const struct sk_buff *skb)

is the header a clone

Parameters

const struct sk_buff *skb

buffer to check

Returns true if modifying the header part of the buffer requires the data to be copied.

void __skb_header_release(struct sk_buff *skb)

allow clones to use the headroom

Parameters

struct sk_buff *skb

buffer to operate on

Description

See "DOC: dataref and headerless skbs".

int skb_shared(const struct sk_buff *skb)

is the buffer shared

Parameters

const struct sk_buff *skb

buffer to check

Returns true if more than one person has a reference to this buffer.

`struct sk_buff *skb_share_check(struct sk_buff *skb, gfp_t pri)`

check if buffer is shared and if so clone it

Parameters

struct sk_buff *skb

buffer to check

gfp_t pri

priority for memory allocation

If the buffer is shared the buffer is cloned and the old copy drops a reference. A new clone with a single reference is returned. If the buffer is not shared the original buffer is returned. When being called from interrupt status or with spinlocks held **pri** must be GFP_ATOMIC.

NULL is returned on a memory allocation failure.

`struct sk_buff *skb_unshare(struct sk_buff *skb, gfp_t pri)`

make a copy of a shared buffer

Parameters

struct sk_buff *skb

buffer to check

gfp_t pri

priority for memory allocation

If the socket buffer is a clone then this function creates a new copy of the data, drops a reference count on the old copy and returns the new copy with the reference count at 1. If the buffer is not a clone the original buffer is returned. When called with a spinlock held or from interrupt state **pri** must be GFP_ATOMIC

NULL is returned on a memory allocation failure.

`struct sk_buff *skb_peek(const struct sk_buff_head *list_)`

peek at the head of an `sk_buff_head`

Parameters

const struct sk_buff_head *list_

list to peek at

Peek an `sk_buff`. Unlike most other operations you MUST be careful with this one. A peek leaves the buffer on the list and someone else may run off with it. You must hold the appropriate locks or have a private queue to do this.

Returns NULL for an empty list or a pointer to the head element. The reference count is not incremented and the reference is therefore volatile. Use with caution.

`struct sk_buff *__skb_peek(const struct sk_buff_head *list_)`

peek at the head of a non-empty `sk_buff_head`

Parameters

const struct sk_buff_head *list_

list to peek at

Like `skb_peek()`, but the caller knows that the list is not empty.

`struct sk_buff *skb_peek_next(struct sk_buff *skb, const struct sk_buff_head *list_)`
 peek skb following the given one from a queue

Parameters

struct sk_buff *skb
 skb to start from

const struct sk_buff_head *list_
 list to peek at

Returns NULL when the end of the list is met or a pointer to the next element. The reference count is not incremented and the reference is therefore volatile. Use with caution.

`struct sk_buff *skb_peek_tail(const struct sk_buff_head *list_)`
 peek at the tail of an `sk_buff_head`

Parameters

const struct sk_buff_head *list_
 list to peek at

Peek an `sk_buff`. Unlike most other operations you `_MUST_` be careful with this one. A peek leaves the buffer on the list and someone else may run off with it. You must hold the appropriate locks or have a private queue to do this.

Returns NULL for an empty list or a pointer to the tail element. The reference count is not incremented and the reference is therefore volatile. Use with caution.

`_u32 skb_queue_len(const struct sk_buff_head *list_)`
 get queue length

Parameters

const struct sk_buff_head *list_
 list to measure

Return the length of an `sk_buff` queue.

`_u32 skb_queue_len_lockless(const struct sk_buff_head *list_)`
 get queue length

Parameters

const struct sk_buff_head *list_
 list to measure

Return the length of an `sk_buff` queue. This variant can be used in lockless contexts.

`void __skb_queue_head_init(struct sk_buff_head *list)`
 initialize non-spinlock portions of `sk_buff_head`

Parameters

struct sk_buff_head *list
 queue to initialize

This initializes only the list and queue length aspects of an `sk_buff_head` object. This allows to initialize the list aspects of an `sk_buff_head` without reinitializing things like the spinlock. It can also be used for on-stack `sk_buff_head` objects where the spinlock is known to not be used.

void `skb_queue_splice`(const struct sk_buff_head *list, struct sk_buff_head *head)
join two skb lists, this is designed for stacks

Parameters

const struct sk_buff_head *list
the new list to add

struct sk_buff_head *head
the place to add it in the first list

void `skb_queue_splice_init`(struct sk_buff_head *list, struct sk_buff_head *head)
join two skb lists and reinitialise the emptied list

Parameters

struct sk_buff_head *list
the new list to add

struct sk_buff_head *head
the place to add it in the first list

The list at **list** is reinitialised

void `skb_queue_splice_tail`(const struct sk_buff_head *list, struct sk_buff_head *head)
join two skb lists, each list being a queue

Parameters

const struct sk_buff_head *list
the new list to add

struct sk_buff_head *head
the place to add it in the first list

void `skb_queue_splice_tail_init`(struct sk_buff_head *list, struct sk_buff_head *head)
join two skb lists and reinitialise the emptied list

Parameters

struct sk_buff_head *list
the new list to add

struct sk_buff_head *head
the place to add it in the first list

Each of the lists is a queue. The list at **list** is reinitialised

void `__skb_queue_after`(struct sk_buff_head *list, struct *sk_buff* *prev, struct *sk_buff* *newsk)
queue a buffer at the list head

Parameters

struct sk_buff_head *list
list to use

struct sk_buff *prev
place after this buffer

struct sk_buff *newsk
buffer to queue

Queue a buffer int the middle of a list. This function takes no locks and you must therefore hold required locks before calling it.

A buffer cannot be placed on two lists at the same time.

void __skb_queue_head(struct sk_buff_head *list, struct sk_buff *news)
queue a buffer at the list head

Parameters

struct sk_buff_head *list
list to use

struct sk_buff *news
buffer to queue

Queue a buffer at the start of a list. This function takes no locks and you must therefore hold required locks before calling it.

A buffer cannot be placed on two lists at the same time.

void __skb_queue_tail(struct sk_buff_head *list, struct sk_buff *news)
queue a buffer at the list tail

Parameters

struct sk_buff_head *list
list to use

struct sk_buff *news
buffer to queue

Queue a buffer at the end of a list. This function takes no locks and you must therefore hold required locks before calling it.

A buffer cannot be placed on two lists at the same time.

struct sk_buff *__skb_dequeue(struct sk_buff_head *list)
remove from the head of the queue

Parameters

struct sk_buff_head *list
list to dequeue from

Remove the head of the list. This function does not take any locks so must be used with appropriate locks held only. The head item is returned or NULL if the list is empty.

struct sk_buff *__skb_dequeue_tail(struct sk_buff_head *list)
remove from the tail of the queue

Parameters

struct sk_buff_head *list
list to dequeue from

Remove the tail of the list. This function does not take any locks so must be used with appropriate locks held only. The tail item is returned or NULL if the list is empty.

```
void skb_len_add(struct sk_buff *skb, int delta)
    adds a number to len fields of skb
```

Parameters

struct sk_buff *skb
buffer to add len to

int delta
number of bytes to add

```
void _skb_fill_page_desc(struct sk_buff *skb, int i, struct page *page, int off, int size)
    initialise a paged fragment in an skb
```

Parameters

struct sk_buff *skb
buffer containing fragment to be initialised

int i
paged fragment index to initialise

struct page *page
the page to use for this fragment

int off
the offset to the data with **page**

int size
the length of the data

Description

Initialises the **i**'th fragment of **skb** to point to **size** bytes at offset **off** within **page**.

Does not take any additional reference on the fragment.

```
void skb_fill_page_desc(struct sk_buff *skb, int i, struct page *page, int off, int size)
    initialise a paged fragment in an skb
```

Parameters

struct sk_buff *skb
buffer containing fragment to be initialised

int i
paged fragment index to initialise

struct page *page
the page to use for this fragment

int off
the offset to the data with **page**

int size
the length of the data

Description

As per `_skb_fill_page_desc()` -- initialises the **i**'th fragment of **skb** to point to **size** bytes at offset **off** within **page**. In addition updates **skb** such that **i** is the last fragment.

Does not take any additional reference on the fragment.

```
void skb_fill_page_desc_noacc(struct sk_buff *skb, int i, struct page *page, int off, int size)
    initialise a paged fragment in an skb
```

Parameters

struct sk_buff *skb
buffer containing fragment to be initialised

int i
paged fragment index to initialise

struct page *page
the page to use for this fragment

int off
the offset to the data with **page**

int size
the length of the data

Description

Variant of *skb_fill_page_desc()* which does not deal with pfmemalloc, if page is not owned by us.

```
unsigned int skb_headroom(const struct sk_buff *skb)
    bytes at buffer head
```

Parameters

const struct sk_buff *skb
buffer to check
Return the number of bytes of free space at the head of an *sk_buff*.

```
int skb_tailroom(const struct sk_buff *skb)
    bytes at buffer end
```

Parameters

const struct sk_buff *skb
buffer to check
Return the number of bytes of free space at the tail of an *sk_buff*

```
int skb_availroom(const struct sk_buff *skb)
    bytes at buffer end
```

Parameters

const struct sk_buff *skb
buffer to check
Return the number of bytes of free space at the tail of an *sk_buff* allocated by *sk_stream_alloc()*

```
void skb_reserve(struct sk_buff *skb, int len)
    adjust headroom
```

Parameters

struct sk_buff *skb
buffer to alter

int len
bytes to move

Increase the headroom of an empty *sk_buff* by reducing the tail room. This is only allowed for an empty buffer.

void skb_tailroom_reserve(struct sk_buff *skb, unsigned int mtu, unsigned int needed_tailroom)
adjust reserved_tailroom

Parameters

struct sk_buff *skb
buffer to alter

unsigned int mtu
maximum amount of headlen permitted

unsigned int needed_tailroom
minimum amount of reserved_tailroom

Set reserved_tailroom so that headlen can be as large as possible but not larger than mtu and tailroom cannot be smaller than needed_tailroom. The required headroom should already have been reserved before using this function.

void pskb_trim_unique(struct sk_buff *skb, unsigned int len)
remove end from a paged unique (not cloned) buffer

Parameters

struct sk_buff *skb
buffer to alter

unsigned int len
new length

This is identical to pskb_trim except that the caller knows that the skb is not cloned so we should never get an error due to out-of-memory.

void skb_orphan(struct sk_buff *skb)
orphan a buffer

Parameters

struct sk_buff *skb
buffer to orphan

If a buffer currently has an owner then we call the owner's destructor function and make the **skb** unowned. The buffer continues to exist but is no longer charged to its former owner.

int skb_orphan frags(struct sk_buff *skb, gfp_t gfp_mask)
orphan the frags contained in a buffer

Parameters

struct sk_buff *skb
 buffer to orphan frags from

gfp_t gfp_mask
 allocation mask for replacement pages

For each frag in the SKB which needs a destructor (i.e. has an owner) create a copy of that frag and release the original page by calling the destructor.

void __skb_queue_purge_reason(struct sk_buff_head *list, enum skb_drop_reason reason)
 empty a list

Parameters

struct sk_buff_head *list
 list to empty

enum skb_drop_reason reason
 drop reason

Delete all buffers on an *sk_buff* list. Each buffer is removed from the list and one reference dropped. This function does not take the list lock and the caller must hold the relevant locks to use it.

void *netdev_alloc_frag(unsigned int fragsz)
 allocate a page fragment

Parameters

unsigned int fragsz
 fragment size

Description

Allocates a frag from a page for receive buffer. Uses GFP_ATOMIC allocations.

struct sk_buff *netdev_alloc_skb(struct net_device *dev, unsigned int length)
 allocate an skbuff for rx on a specific device

Parameters

struct net_device *dev
 network device to receive on

unsigned int length
 length to allocate

Allocate a new *sk_buff* and assign it a usage count of one. The buffer has unspecified headroom built in. Users should allocate the headroom they think they need without accounting for the built in space. The built in space is used for optimisations.

NULL is returned if there is no free memory. Although this function allocates memory it can be called from an interrupt.

struct page *__dev_alloc_pages(gfp_t gfp_mask, unsigned int order)
 allocate page for network Rx

Parameters

gfp_t gfp_mask
 allocation priority. Set __GFP_NOMEMALLOC if not for network Rx

unsigned int order
size of the allocation

Description

Allocate a new page.

NULL is returned if there is no free memory.

struct page *__dev_alloc_page(gfp_t gfp_mask)
allocate a page for network Rx

Parameters

gfp_t gfp_mask
allocation priority. Set __GFP_NOMEMALLOC if not for network Rx

Description

Allocate a new page.

NULL is returned if there is no free memory.

bool dev_page_is_reusable(const struct page *page)
check whether a page can be reused for network Rx

Parameters

const struct page *page
the page to test

Description

A page shouldn't be considered for reusing/recycling if it was allocated under memory pressure or at a distant memory node.

Returns false if this page should be returned to page allocator, true otherwise.

void skb_propagate_pfmemalloc(const struct page *page, struct sk_buff *skb)
Propagate pfmemalloc if skb is allocated after RX page

Parameters

const struct page *page
The page that was allocated from skb_alloc_page

struct sk_buff *skb
The skb that may need pfmemalloc set

unsigned int skb_frag_off(const skb_frag_t *frag)
Returns the offset of a skb fragment

Parameters

const skb_frag_t *frag
the paged fragment

void skb_frag_off_add(skb_frag_t *frag, int delta)
Increments the offset of a skb fragment by **delta**

Parameters

skb_frag_t *frag
 skb fragment

int delta
 value to add

void skb_frag_off_set(skb_frag_t *frag, unsigned int offset)
 Sets the offset of a skb fragment

Parameters

skb_frag_t *frag
 skb fragment

unsigned int offset
 offset of fragment

void skb_frag_off_copy(skb_frag_t *fragto, const skb_frag_t *fragfrom)
 Sets the offset of a skb fragment from another fragment

Parameters

skb_frag_t *fragto
 skb fragment where offset is set

const skb_frag_t *fragfrom
 skb fragment offset is copied from

struct page *skb_frag_page(const skb_frag_t *frag)
 retrieve the page referred to by a paged fragment

Parameters

const skb_frag_t *frag
 the paged fragment

Description

Returns the struct page associated with **frag**.

void __skb_frag_ref(skb_frag_t *frag)
 take an addition reference on a paged fragment.

Parameters

skb_frag_t *frag
 the paged fragment

Description

Takes an additional reference on the paged fragment **frag**.

void skb_frag_ref(struct sk_buff *skb, int f)
 take an addition reference on a paged fragment of an skb.

Parameters

struct sk_buff *skb
 the buffer

int f
 the fragment offset.

Description

Takes an additional reference on the **f**'th paged fragment of **skb**.

```
void __skb_frag_unref(skb_frag_t *frag, bool recycle)
    release a reference on a paged fragment.
```

Parameters

skb_frag_t *frag
the paged fragment

bool recycle
recycle the page if allocated via page_pool

Description

Releases a reference on the paged fragment **frag** or recycles the page via the page_pool API.

```
void skb_frag_unref(struct sk_buff *skb, int f)
    release a reference on a paged fragment of an skb.
```

Parameters

struct sk_buff *skb
the buffer

int f
the fragment offset

Description

Releases a reference on the **f**'th paged fragment of **skb**.

```
void *skb_frag_address(const skb_frag_t *frag)
    gets the address of the data contained in a paged fragment
```

Parameters

const skb_frag_t *frag
the paged fragment buffer

Description

Returns the address of the data within **frag**. The page must already be mapped.

```
void *skb_frag_address_safe(const skb_frag_t *frag)
    gets the address of the data contained in a paged fragment
```

Parameters

const skb_frag_t *frag
the paged fragment buffer

Description

Returns the address of the data within **frag**. Checks that the page is mapped and returns NULL otherwise.

```
void skb_frag_page_copy(skb_frag_t *fragto, const skb_frag_t *fragfrom)
    sets the page in a fragment from another fragment
```

Parameters

skb_frag_t *fragto

skb fragment where page is set

const skb_frag_t *fragfrom

skb fragment page is copied from

dma_addr_t **skb_frag_dma_map**(struct device *dev, const skb_frag_t *frag, size_t offset, size_t size, enum dma_data_direction dir)

maps a paged fragment via the DMA API

Parameters**struct device *dev**

the device to map the fragment to

const skb_frag_t *frag

the paged fragment to map

size_t offset

the offset within the fragment (starting at the fragment's own offset)

size_t size

the number of bytes to map

enum dma_data_direction dir

the direction of the mapping (PCI_DMA_*)

Description

Maps the page associated with **frag** to **device**.

int **skb_clone_writable**(const struct *sk_buff* *skb, unsigned int len)

is the header of a clone writable

Parameters**const struct sk_buff *skb**

buffer to check

unsigned int len

length up to which to write

Returns true if modifying the header part of the cloned buffer does not require the data to be copied.

int **skb_cow**(struct *sk_buff* *skb, unsigned int headroom)

copy header of skb when it is required

Parameters**struct sk_buff *skb**

buffer to cow

unsigned int headroom

needed headroom

If the skb passed lacks sufficient headroom or its data part is shared, data is reallocated. If reallocation fails, an error is returned and original skb is not changed.

The result is skb with writable area skb->head...skb->tail and at least **headroom** of space at head.

```
int skb_cow_head(struct sk_buff *skb, unsigned int headroom)
    skb_cow but only making the head writable
```

Parameters

struct sk_buff *skb
buffer to cow

unsigned int headroom
needed headroom

This function is identical to `skb_cow` except that we replace the `skb_cloned` check by `skb_header_cloned`. It should be used when you only need to push on some header and do not need to modify the data.

```
int skb_padto(struct sk_buff *skb, unsigned int len)
    pad an skbuff up to a minimal size
```

Parameters

struct sk_buff *skb
buffer to pad

unsigned int len
minimal length

Pads up a buffer to ensure the trailing bytes exist and are blanked. If the buffer already contains sufficient data it is untouched. Otherwise it is extended. Returns zero on success. The skb is freed on error.

```
int _skb_put_padto(struct sk_buff *skb, unsigned int len, bool free_on_error)
    increase size and pad an skbuff up to a minimal size
```

Parameters

struct sk_buff *skb
buffer to pad

unsigned int len
minimal length

bool free_on_error
free buffer on error

Pads up a buffer to ensure the trailing bytes exist and are blanked. If the buffer already contains sufficient data it is untouched. Otherwise it is extended. Returns zero on success. The skb is freed on error if `free_on_error` is true.

```
int skb_put_padto(struct sk_buff *skb, unsigned int len)
    increase size and pad an skbuff up to a minimal size
```

Parameters

struct sk_buff *skb
buffer to pad

unsigned int len
minimal length

Pads up a buffer to ensure the trailing bytes exist and are blanked. If the buffer already contains sufficient data it is untouched. Otherwise it is extended. Returns zero on success. The skb is freed on error.

```
int skb_linearize(struct sk_buff *skb)
    convert paged skb to linear one
```

Parameters

struct sk_buff *skb
buffer to linearize

If there is no free memory -ENOMEM is returned, otherwise zero is returned and the old skb data released.

```
bool skb_has_shared_frag(const struct sk_buff *skb)
    can any frag be overwritten
```

Parameters

const struct sk_buff *skb
buffer to test

Description

Return true if the skb has at least one frag that might be modified by an external entity (as in vmsplice()/sendfile())

```
int skb_linearize_cow(struct sk_buff *skb)
    make sure skb is linear and writable
```

Parameters

struct sk_buff *skb
buffer to process

If there is no free memory -ENOMEM is returned, otherwise zero is returned and the old skb data released.

```
void skb_postpull_rcsum(struct sk_buff *skb, const void *start, unsigned int len)
    update checksum for received skb after pull
```

Parameters

struct sk_buff *skb
buffer to update

const void *start
start of data before pull

unsigned int len
length of data pulled

After doing a pull on a received packet, you need to call this to update the CHECKSUM_COMPLETE checksum, or set ip_summed to CHECKSUM_NONE so that it can be recomputed from scratch.

```
void skb_postpush_rcsum(struct sk_buff *skb, const void *start, unsigned int len)
    update checksum for received skb after push
```

Parameters

```
struct sk_buff *skb
    buffer to update

const void *start
    start of data after push

unsigned int len
    length of data pushed

After doing a push on a received packet, you need to call this to update the CHECKSUM_COMPLETE checksum.
```

```
void *skb_push_rcsum(struct sk_buff *skb, unsigned int len)
    push skb and update receive checksum
```

Parameters

```
struct sk_buff *skb
    buffer to update

unsigned int len
    length of data pulled

This function performs an skb_push on the packet and updates the CHECKSUM_COMPLETE checksum. It should be used on receive path processing instead of skb_push unless you know that the checksum difference is zero (e.g., a valid IP header) or you are setting ip_summed to CHECKSUM_NONE.
```

```
int pskb_trim_rcsum(struct sk_buff *skb, unsigned int len)
    trim received skb and update checksum
```

Parameters

```
struct sk_buff *skb
    buffer to trim

unsigned int len
    new length

This is exactly the same as pskb_trim except that it ensures the checksum of received packets are still valid after the operation. It can change skb pointers.
```

```
bool skb_needs_linearize(struct sk_buff *skb, netdev_features_t features)
    check if we need to linearize a given skb depending on the given device features.
```

Parameters

```
struct sk_buff *skb
    socket buffer to check

netdev_features_t features
    net device features

Returns true if either: 1. skb has frag_list and the device doesn't support FRAGLIST, or 2. skb is fragmented and the device does not support SG.
```

```
void skb_get_timestamp(const struct sk_buff *skb, struct __kernel_old_timeval *stamp)
    get timestamp from a skb
```

Parameters

const struct sk_buff *skb

skb to get stamp from

struct __kernel_old_timeval *stamp

pointer to struct __kernel_old_timeval to store stamp in

Timestamps are stored in the skb as offsets to a base timestamp. This function converts the offset back to a struct timeval and stores it in stamp.

void skb_complete_tx_timestamp(struct sk_buff *skb, struct skb_shared_hwtstamps *hwtstamps)

deliver cloned skb with tx timestamps

Parameters

struct sk_buff *skb

clone of the original outgoing packet

struct skb_shared_hwtstamps *hwtstamps

hardware time stamps

Description

PHY drivers may accept clones of transmitted packets for timestamping via their phy_driver.txtstamp method. These drivers must call this function to return the skb back to the stack with a timestamp.

void skb_tstamp_tx(struct sk_buff *orig_skb, struct skb_shared_hwtstamps *hwtstamps)

queue clone of skb with send time stamps

Parameters

struct sk_buff *orig_skb

the original outgoing packet

struct skb_shared_hwtstamps *hwtstamps

hardware time stamps, may be NULL if not available

Description

If the skb has a socket associated, then this function clones the skb (thus sharing the actual data and optional structures), stores the optional hardware time stamping information (if non NULL) or generates a software time stamp (otherwise), then queues the clone to the error queue of the socket. Errors are silently ignored.

void skb_tx_timestamp(struct sk_buff *skb)

Driver hook for transmit timestamping

Parameters

struct sk_buff *skb

A socket buffer.

Description

Ethernet MAC Drivers should call this function in their hard_xmit() function immediately before giving the sk_buff to the MAC hardware.

Specifically, one should make absolutely sure that this function is called before TX completion of this packet can trigger. Otherwise the packet could potentially already be freed.

```
void skb_complete_wifi_ack(struct sk_buff *skb, bool acked)
    deliver skb with wifi status
```

Parameters

struct sk_buff *skb
the original outgoing packet

bool acked
ack status

```
_sum16 skb_checksum_complete(struct sk_buff *skb)
    Calculate checksum of an entire packet
```

Parameters

struct sk_buff *skb
packet to process

This function calculates the checksum over the entire packet plus the value of skb->csum. The latter can be used to supply the checksum of a pseudo header as used by TCP/UDP. It returns the checksum.

For protocols that contain complete checksums such as ICMP/TCP/UDP, this function can be used to verify that checksum on received packets. In that case the function should return zero if the checksum is correct. In particular, this function will return zero if skb->ip_summed is CHECKSUM_UNNECESSARY which indicates that the hardware has already verified the correctness of the checksum.

struct skb_ext
sk_buff extensions

Definition:

```
struct skb_ext {
    refcount_t refcnt;
    u8 offset[SKB_EXT_NUM];
    u8 chunks;
    char data[] ;
};
```

Members

refcnt
1 on allocation, deallocated on 0

offset
offset to add to **data** to obtain extension address

chunks
size currently allocated, stored in SKB_EXT_ALIGN_SHIFT units

data
start of extension data, variable sized

Note

offsets/lengths are stored in chunks of 8 bytes, this allows
to use 'u8' types while allowing up to 2kb worth of extension data.

```
void skb_checksum_none_assert(const struct sk_buff *skb)
    make sure skb ip_summed is CHECKSUM_NONE
```

Parameters

const struct sk_buff *skb
skb to check

Description

fresh skbs have their ip_summed set to CHECKSUM_NONE. Instead of forcing ip_summed to CHECKSUM_NONE, we can use this helper, to document places where we make this assertion.

```
bool skb_head_is_locked(const struct sk_buff *skb)
```

Determine if the skb->head is locked down

Parameters

const struct sk_buff *skb
skb to check

Description

The head on skbs build around a head frag can be removed if they are not cloned. This function returns true if the skb head is locked down due to either being allocated via kmalloc, or by being a clone with multiple references to the head.

struct sock_common

minimal network layer representation of sockets

Definition:

```
struct sock_common {
    union {
        __addrpair skc_addrpair;
        struct {
            __be32 skc_daddr;
            __be32 skc_rcv_saddr;
        };
    };
    union {
        unsigned int skc_hash;
        __u16 skc_u16hashes[2];
    };
    union {
        __portpair skc_portpair;
        struct {
            __be16 skc_dport;
            __u16 skc_num;
        };
    };
    unsigned short skc_family;
    volatile unsigned char skc_state;
    unsigned char skc_reuse:4;
    unsigned char skc_reuseport:1;
    unsigned char skc_ipv6only:1;
```

```

    unsigned char          skc_net_refcnt:1;
    int skc_bound_dev_if;
    union {
        struct hlist_node      skc_bind_node;
        struct hlist_node      skc_portaddr_node;
    };
    struct proto           *skc_prot;
    possible_net_t          skc_net;
#ifndef IS_ENABLED(CONFIG_IPV6)
    struct in6_addr         skc_v6_daddr;
    struct in6_addr         skc_v6_rcv_saddr;
#endif;
    atomic64_t              skc_cookie;
    union {
        unsigned long         skc_flags;
        struct sock           *skc_listener;
        struct inet_timewait_death_row *skc_tw_dr;
    };
    union {
        struct hlist_node      skc_node;
        struct hlist_nulls_node skc_nulls_node;
    };
    unsigned short          skc_tx_queue_mapping;
#ifndef CONFIG_SOCK_RX_QUEUE_MAPPING
    unsigned short          skc_rx_queue_mapping;
#endif;
    union {
        int skc_incoming_cpu;
        u32 skc_rcv_wnd;
        u32 skc_tw_rcv_nxt;
    };
    refcount_t               skc_refcnt;
};

```

Members

{unnamed_union}

anonymous

skc_addrpair

8-byte-aligned __u64 union of **skc_daddr** & **skc_rcv_saddr**

{unnamed_struct}

anonymous

skc_daddr

Foreign IPv4 addr

skc_rcv_saddr

Bound local IPv4 addr

{unnamed_union}

anonymous

skc_hash
hash value used with various protocol lookup tables

skc_u16hashes
two u16 hash values used by UDP lookup tables

{unnamed_union}
anonymous

skc_portpair
_u32 union of **skc_dport** & **skc_num**

{unnamed_struct}
anonymous

skc_dport
placeholder for inet_dport/tw_dport

skc_num
placeholder for inet_num/tw_num

skc_family
network address family

skc_state
Connection state

skc_reuse
SO_REUSEADDR setting

skc_reuseport
SO_REUSEPORT setting

skc_ipv6only
socket is IPV6 only

skc_net_refcnt
socket is using net ref counting

skc_bound_dev_if
bound device index if != 0

{unnamed_union}
anonymous

skc_bind_node
bind hash linkage for various protocol lookup tables

skc_portaddr_node
second hash linkage for UDP/UDP-Lite protocol

skc_prot
protocol handlers inside a network family

skc_net
reference to the network namespace of this socket

skc_v6_daddr
IPV6 destination address

skc_v6_rcv_saddr
IPV6 source address

skc_cookie
socket's cookie value

{unnamed_union}
anonymous

skc_flags
place holder for sk_flags SO_LINGER (l_onoff), SO_BROADCAST, SO_KEEPALIVE, SO_OOBINLINE settings, SO_TIMESTAMPING settings

skc_listener
connection request listener socket (aka rsk_listener) [union with **skc_flags**]

skc_tw_dr
(aka tw_dr) ptr to struct inet_timewait_death_row [union with **skc_flags**]

{unnamed_union}
anonymous

skc_node
main hash linkage for various protocol lookup tables

skc_nulls_node
main hash linkage for TCP/UDP/UDP-Lite protocol

skc_tx_queue_mapping
tx queue number for this connection

skc_rx_queue_mapping
rx queue number for this connection

{unnamed_union}
anonymous

skc_incoming_cpu
record/match cpu processing incoming packets

skc_rcv_wnd
(aka rsk_rcv_wnd) TCP receive window size (possibly scaled) [union with **skc_incoming_cpu**]

skc_tw_rcv_nxt
(aka tw_rcv_nxt) TCP window next expected seq number [union with **skc_incoming_cpu**]

skc_refcnt
reference count

This is the minimal network layer representation of sockets, the header for *struct sock* and *struct inet_timewait_sock*.

struct sock
network layer representation of sockets

Definition:

```
struct sock {  
    struct sock_common      __sk_common;
```

```

#define sk_node           __sk_common.skc_node;
#define sk_nulls_node    __sk_common.skc_nulls_node;
#define sk_refcnt         __sk_common.skc_refcnt;
#define sk_tx_queue_mapping __sk_common.skc_tx_queue_mapping;
#ifndef CONFIG_SOCK_RX_QUEUE_MAPPING;
#define sk_rx_queue_mapping __sk_common.skc_rx_queue_mapping;
#endif;
#define sk_dontcopy_begin __sk_common.skc_dontcopy_begin;
#define sk_dontcopy_end   __sk_common.skc_dontcopy_end;
#define sk_hash            __sk_common.skc_hash;
#define sk_portpair        __sk_common.skc_portpair;
#define sk_num             __sk_common.skc_num;
#define sk_dport            __sk_common.skc_dport;
#define sk_addrpair        __sk_common.skc_addrpair;
#define sk_daddr            __sk_common.skc_daddr;
#define sk_rcv_saddr       __sk_common.skc_rcv_saddr;
#define sk_family           __sk_common.skc_family;
#define sk_state            __sk_common.skc_state;
#define sk_reuse            __sk_common.skc_reuse;
#define sk_reuseport        __sk_common.skc_reuseport;
#define sk_ipv6only         __sk_common.skc_ipv6only;
#define sk_net_refcnt      __sk_common.skc_net_refcnt;
#define sk_bound_dev_if    __sk_common.skc_bound_dev_if;
#define sk_bind_node        __sk_common.skc_bind_node;
#define sk_prot             __sk_common.skc_prot;
#define sk_net              __sk_common.skc_net;
#define sk_v6_daddr         __sk_common.skc_v6_daddr;
#define sk_v6_rcv_saddr    __sk_common.skc_v6_rcv_saddr;
#define sk_cookie           __sk_common.skc_cookie;
#define sk_incoming_cpu     __sk_common.skc_incoming_cpu;
#define sk_flags             __sk_common.skc_flags;
#define sk_rxhash           __sk_common.skc_rxhash;

    struct dst_entry __rcu *sk_rx_dst;
    int sk_rx_dst_ifindex;
    u32 sk_rx_dst_cookie;
    socket_lock_t sk_lock;
    atomic_t sk_drops;
    int sk_rcvlowat;
    struct sk_buff_head    sk_error_queue;
    struct sk_buff_head    sk_receive_queue;
    struct {
        atomic_t rmem_alloc;
        int len;
        struct sk_buff *head;
        struct sk_buff *tail;
    } sk_backlog;
#define sk_rmem_alloc sk_backlog.rmem_alloc;
    int sk_forward_alloc;
    u32 sk_reserved_mem;
#endif CONFIG_NET_RX_BUSY_POLL;

```

```

        unsigned int          sk_ll_usec;
        unsigned int          sk_napi_id;
#endif;
        int sk_rcvbuf;
        int sk_disconnects;
        struct sk_filter __rcu *sk_filter;
        union {
            struct socket_wq __rcu *sk_wq;
        };
#endif CONFIG_XFRM;
        struct xfrm_policy __rcu *sk_policy[2];
#endif;
        struct dst_entry __rcu *sk_dst_cache;
        atomic_t sk_omem_alloc;
        int sk_sndbuf;
        int sk_wmem_queued;
        refcount_t sk_wmem_alloc;
        unsigned long          sk_tsq_flags;
        union {
            struct sk_buff *sk_send_head;
            struct rb_root tcp_rtx_queue;
        };
        struct sk_buff_head    sk_write_queue;
        __s32 sk_peek_off;
        int sk_write_pending;
        __u32 sk_dst_pending_confirm;
        u32 sk_pacing_status;
        long sk_sndtimeo;
        struct timer_list      sk_timer;
        __u32 sk_priority;
        __u32 sk_mark;
        unsigned long          sk_pacing_rate;
        unsigned long          sk_max_pacing_rate;
        struct page_frag       sk_frag;
        netdev_features_t sk_route_caps;
        int sk_gso_type;
        unsigned int           sk_gso_max_size;
        gfp_t sk_allocation;
        __u32 sk_txhash;
        u8 sk_gso_disabled : 1,sk_kern_sock : 1,sk_no_check_tx : 1,sk_no_check_rx : 1, sk_userlocks : 4;
        u8 sk_pacing_shift;
        u16 sk_type;
        u16 sk_protocol;
        u16 sk_gso_max_segs;
        unsigned long          sk_lingertime;
        struct proto           *sk_prot_creator;
        rwlock_t sk_callback_lock;
        int sk_err, sk_err_soft;
        u32 sk_ack_backlog;

```

```

u32 sk_max_ack_backlog;
kuid_t sk_uid;
u8 sk_txrehash;
#ifndef CONFIG_NET_RX_BUSY_POLL;
    u8 sk_prefer_busy_poll;
    u16 sk_busy_poll_budget;
#endif;
    spinlock_t sk_peer_lock;
    int sk_bind_phc;
    struct pid             *sk_peer_pid;
    const struct cred       *sk_peer_cred;
    long sk_rcvtimeo;
    ktime_t sk_stamp;
#if BITS_PER_LONG==32;
    seqlock_t sk_stamp_seq;
#endif;
    atomic_t sk_tskey;
    atomic_t sk_zckey;
    u32 sk_tsflags;
    u8 sk_shutdown;
    u8 sk_clockid;
    u8 sk_txtime_deadline_mode : 1,sk_txtime_report_errors : 1, sk_txtime_
→unused : 6;
    bool sk_use_task_frag;
    struct socket           *sk_socket;
    void *sk_user_data;
#ifndef CONFIG_SECURITY;
    void *sk_security;
#endif;
    struct sock_cgroup_data sk_cgrp_data;
    struct mem_cgroup        *sk_memcg;
    void (*sk_state_change)(struct sock *sk);
    void (*sk_data_ready)(struct sock *sk);
    void (*sk_write_space)(struct sock *sk);
    void (*sk_error_report)(struct sock *sk);
    int (*sk_backlog_rcv)(struct sock *sk, struct sk_buff *skb);
#ifndef CONFIG_SOCK_VALIDATE_XMIT;
    struct sk_buff* (*sk_validate_xmit_skb)(struct sock *sk, struct net_
→device *dev, struct sk_buff *skb);
#endif;
    void (*sk_destruct)(struct sock *sk);
    struct sock_reuseport __rcu     *sk_reuseport_cb;
#ifndef CONFIG_BPF_SYSCALL;
    struct bpf_local_storage __rcu  *sk_bpf_storage;
#endif;
    struct rcu_head           sk_rcu;
    netns_tracker ns_tracker;
};


```

Members

__sk_common
shared layout with inet_timewait_sock

sk_rx_dst
receive input route used by early demux

sk_rx_dst_ifindex
ifindex for **sk_rx_dst**

sk_rx_dst_cookie
cookie for **sk_rx_dst**

sk_lock
synchronizer

sk_drops
raw/udp drops counter

sk_rcvlowat
SO_RCVLOWAT setting

sk_error_queue
rarely used

sk_receive_queue
incoming packets

sk_backlog
always used with the per-socket spinlock held

sk_forward_alloc
space allocated forward

sk_reserved_mem
space reserved and non-reclaimable for the socket

sk_ll_usec
usecs to busypoll when there is no data

sk_napi_id
id of the last napi context to receive data for sk

sk_rcvbuf
size of receive buffer in bytes

sk_disconnects
number of disconnect operations performed on this sock

sk_filter
socket filtering instructions

{unnamed_union}
anonymous

sk_wq
sock wait queue and async head

sk_policy
flow policy

sk_dst_cache

destination cache

sk_omem_alloc

"o" is "option" or "other"

sk_sndbuf

size of send buffer in bytes

sk_wmem_queued

persistent queue size

sk_wmem_alloc

transmit queue bytes committed

sk_tsq_flags

TCP Small Queues flags

{unnamed_union}

anonymous

sk_send_head

front of stuff to transmit

tcp_rtx_queue

TCP re-transmit queue [union with **sk_send_head**]

sk_write_queue

Packet sending queue

sk_peek_offset

current peek_offset value

sk_write_pending

a write to stream socket waits to start

sk_dst_pending_confirm

need to confirm neighbour

sk_pacing_status

Pacing status (requested, handled by sch_fq)

sk_sndtimeo

S0_SNDTIMEO setting

sk_timer

sock cleanup timer

sk_priority

S0_PRIORITY setting

sk_mark

generic packet mark

sk_pacing_rate

Pacing rate (if supported by transport/packet scheduler)

sk_max_pacing_rate

Maximum pacing rate (S0_MAX_PACING_RATE)

sk_frag
cached page frag

sk_route_caps
route capabilities (e.g. NETIF_F_TSO)

sk_gso_type
GSO type (e.g. SKB_GSO_TCPV4)

sk_gso_max_size
Maximum GSO segment size to build

sk_allocation
allocation mode

sk_txhash
computed flow hash for use on transmit

sk_gso_disabled
if set, NETIF_F_GSO_MASK is forbidden.

sk_kern_sock
True if sock is using kernel lock classes

sk_no_check_tx
SO_NO_CHECK setting, set checksum in TX packets

sk_no_check_rx
allow zero checksum in RX packets

sk_userlocks
SO_SNDBUF and SO_RCVBUF settings

sk_pacing_shift
scaling factor for TCP Small Queues

sk_type
socket type (SOCK_STREAM, etc)

sk_protocol
which protocol this socket belongs in this network family

sk_gso_max_segs
Maximum number of GSO segments

sk_lingertime
SO_LINGER l_linger setting

sk_prot_creator
sk_prot of original sock creator (see ipv6_setsockopt, IPV6_ADDRFORM for instance)

sk_callback_lock
used with the callbacks in the end of this struct

sk_err
last error

sk_err_soft
errors that don't cause failure but are the cause of a persistent failure not just 'timed out'

sk_ack_backlog

current listen backlog

sk_max_ack_backlog

listen backlog set in listen()

sk_uid

user id of owner

sk_txrehash

enable TX hash rethink

sk_prefer_busy_poll

prefer busypolling over softirq processing

sk_busy_poll_budget

napi processing budget when busypolling

sk_peer_lock

lock protecting **sk_peer_pid** and **sk_peer_cred**

sk_bind_phc

SO_TIMESTAMPING bind PHC index of PTP virtual clock for timestamping

sk_peer_pid

struct pid for this socket's peer

sk_peer_cred

SO_PEERCRED setting

sk_rcvtimeo

SO_RCVTIMEO setting

sk_stamp

time stamp of last packet received

sk_stamp_seq

lock for accessing sk_stamp on 32 bit architectures only

sk_tskey

counter to disambiguate concurrent tstamp requests

sk_zckey

counter to order MSG_ZEROCOPY notifications

sk_tsflags

SO_TIMESTAMPING flags

sk_shutdown

mask of SEND_SHUTDOWN and/or RCV_SHUTDOWN

sk_clockid

clockid used by time-based scheduling (SO_TXTIME)

sk_txtime_deadline_mode

set deadline mode for SO_TXTIME

sk_txtime_report_errors

set report errors mode for SO_TXTIME

sk_txtime_unused
unused txtime flags

sk_use_task_frag
allow `sk_page_frag()` to use current->task_frag. Sockets that can be used under memory reclaim should set this to false.

sk_socket
Identd and reporting IO signals

sk_user_data
RPC layer private data. Write-protected by **sk_callback_lock**.

sk_security
used by security modules

sk_cgrp_data
cgroup data for this cgroup

sk_memcg
this socket's memory cgroup association

sk_state_change
callback to indicate change in the state of the sock

sk_data_ready
callback to indicate there is data to be processed

sk_write_space
callback to indicate there is bf sending space available

sk_error_report
callback to indicate errors (e.g. MSG_ERRQUEUE)

sk_backlog_rcv
callback to process the backlog

sk_validate_xmit_skb
ptr to an optional validate function

sk_destruct
called at sock freeing time, i.e. when all refcnt == 0

sk_reuseport_cb
reuseport group container

sk_bpf_storage
ptr to cache and control for bpf_sk_storage

sk_rcu
used during RCU grace period

ns_tracker
tracker for netns reference

bool **sk_user_data_is_nocopy**(const struct *sock* *sk)
Test if sk_user_data pointer must not be copied

Parameters

```
const struct sock *sk
    socket

void *__locked_read_sk_user_data_with_flags(const struct sock *sk, uintptr_t flags)
    return the pointer only if argument flags all has been set in sk_user_data. Otherwise return NULL
```

Parameters

const struct sock *sk
socket

uintptr_t flags
flag bits

Description

The caller must be holding sk->sk_callback_lock.

```
void *__rcu_dereference_sk_user_data_with_flags(const struct sock *sk, uintptr_t flags)
    return the pointer only if argument flags all has been set in sk_user_data. Otherwise return NULL
```

Parameters

const struct sock *sk
socket

uintptr_t flags
flag bits

sk_for_each_entry_offset_rcu

sk_for_each_entry_offset_rcu (tpos, pos, head, offset)

iterate over a list at a given struct offset

Parameters

tpos
the type * to use as a loop cursor.

pos
the struct hlist_node to use as a loop cursor.

head
the head for your list.

offset
offset of hlist_node within the struct.

bool lock_sock_fast(struct *sock* *sk)
fast version of lock_sock

Parameters

struct sock *sk
socket

Description

This version should be used for very small section, where process wont block return false if fast path is taken:

sk_lock.slock locked, owned = 0, BH disabled

return true if slow path is taken:

sk_lock.slock unlocked, owned = 1, BH enabled

void unlock_sock_fast(struct sock *sk, bool slow)
complement of lock_sock_fast

Parameters

struct sock *sk

socket

bool slow

slow mode

Description

fast unlock socket for user context. If slow mode is on, we call regular release_sock()

int sk_wmem_alloc_get(const struct sock *sk)
returns write allocations

Parameters

const struct sock *sk

socket

Return

sk_wmem_alloc minus initial offset of one

int sk_rmem_alloc_get(const struct sock *sk)
returns read allocations

Parameters

const struct sock *sk

socket

Return

sk_rmem_alloc

bool sk_has_allocations(const struct sock *sk)
check if allocations are outstanding

Parameters

const struct sock *sk

socket

Return

true if socket has write or read allocations

bool skwq_has_sleeper(struct socket_wq *wq)
check if there are any waiting processes

Parameters

struct socket_wq *wq
 struct socket_wq

Return

true if socket_wq has waiting processes

Description

The purpose of the skwq_has_sleeper and sock_poll_wait is to wrap the memory barrier call. They were added due to the race found within the tcp code.

Consider following tcp code paths:

CPU1	CPU2
sys_select	receive packet
...	...
__add_wait_queue	update tp->recv_nxt
...	...
tp->recv_nxt check	sock_def_readable
...	{
schedule	rcu_read_lock();
	wq = rcu_dereference(sk->sk_wq);
	if (wq && waitqueue_active(&wq->wait))
	wake_up_interruptible(&wq->wait)
	...
	}

The race for tcp fires when the __add_wait_queue changes done by CPU1 stay in its cache, and so does the tp->recv_nxt update on CPU2 side. The CPU1 could then endup calling schedule and sleep forever if there are no more data on the socket.

void sock_poll_wait(struct file *filp, struct socket *sock, poll_table *p)
 place memory barrier behind the poll_wait call.

Parameters

struct file *filp
 file
struct socket *sock
 socket to wait on
poll_table *p
 poll_table

Description

See the comments in the wq_has_sleeper function.

struct page_frag *sk_page_frag(struct sock *sk)
 return an appropriate page_frag

Parameters

struct sock *sk
 socket

Description

Use the per task page_frag instead of the per socket one for optimization when we know that we're in process context and own everything that's associated with current.

Both direct reclaim and page faults can nest inside other socket operations and end up recursing into `sk_page_frag()` while it's already in use: explicitly avoid task page_frag when users disable sk_use_task_frag.

Return

a per task page_frag if context allows that, otherwise a per socket one.

```
void __sock_tx_timestamp(struct sock *sk, __u16 tsflags, __u8 *tx_flags, __u32 *tskey)  
    checks whether the outgoing packet is to be time stamped
```

Parameters

struct sock *sk

socket sending this packet

__u16 tsflags

timestampling flags to use

__u8 *tx_flags

completed with instructions for time stamping

__u32 *tskey

filled in with next sk_tskey (not for TCP, which uses seqno)

Note

callers should take care of initial *tx_flags value (usually 0)

```
void sk_eat_skb(struct sock *sk, struct sk_buff *skb)  
    Release a skb if it is no longer needed
```

Parameters

struct sock *sk

socket to eat this skb from

struct sk_buff *skb

socket buffer to eat

Description

This routine must be called with interrupts disabled or with the socket locked so that the sk_buff queue operation is ok.

```
struct sock *skb_stole_sock(struct sk_buff *skb, bool *refcounted, bool *prefetched)  
    steal a socket from an sk_buff
```

Parameters

struct sk_buff *skb

sk_buff to steal the socket from

bool *refcounted

is set to true if the socket is reference-counted

bool *prefetched

is set to true if the socket was assigned from bpf

struct file *sock_alloc_file(struct *socket* *sock, int flags, const char *dname)

Bind a *socket* to a *file*

Parameters**struct socket *sock**

socket

int flags

file status flags

const char *dname

protocol name

Returns the *file* bound with **sock**, implicitly storing it in *sock*->*file*. If *dname* is NULL, sets to "".

On failure **sock** is released, and an ERR pointer is returned.

This function uses GFP_KERNEL internally.

struct *socket* *sock_from_file(struct *file* *file)

Return the *socket* bounded to *file*.

Parameters**struct file *file**

file

On failure returns NULL.

struct *socket* *sockfd_lookup(int fd, int *err)

Go from a file number to its socket slot

Parameters**int fd**

file handle

int *err

pointer to an error code return

The file handle passed in is locked and the socket it is bound to is returned. If an error occurs the err pointer is overwritten with a negative errno code and NULL is returned. The function checks for both invalid handles and passing a handle which is not a socket.

On a success the socket object pointer is returned.

struct *socket* *sock_alloc(void)

allocate a socket

Parameters**void**

no arguments

Description

Allocate a new inode and socket object. The two are bound together and initialised. The socket is then returned. If we are out of inodes NULL is returned. This function uses GFP_KERNEL internally.

```
void sock_release(struct socket *sock)
    close a socket
```

Parameters

struct socket *sock
socket to close

The socket is released from the protocol stack if it has a release callback, and the inode is then released if the socket is bound to an inode not a file.

```
int sock_sendmsg(struct socket *sock, struct msghdr *msg)
    send a message through sock
```

Parameters

struct socket *sock
socket

struct msghdr *msg
message to send

Sends **msg** through **sock**, passing through LSM. Returns the number of bytes sent, or an error code.

```
int kernel_sendmsg(struct socket *sock, struct msghdr *msg, struct kvec *vec, size_t num,
                    size_t size)
    send a message through sock (kernel-space)
```

Parameters

struct socket *sock
socket

struct msghdr *msg
message header

struct kvec *vec
kernel vec

size_t num
vec array length

size_t size
total message data size

Builds the message data with **vec** and sends it through **sock**. Returns the number of bytes sent, or an error code.

```
int kernel_sendmsg_locked(struct socket *sk, struct msghdr *msg, struct kvec *vec, size_t
                           num, size_t size)
    send a message through sock (kernel-space)
```

Parameters

```
struct sock *sk
    sock

struct msghdr *msg
    message header

struct kvec *vec
    output s/g array

size_t num
    output s/g array length

size_t size
    total message data size
```

Builds the message data with **vec** and sends it through **sock**. Returns the number of bytes sent, or an error code. Caller must hold **sk**.

```
int sock_recvmsg(struct socket *sock, struct msghdr *msg, int flags)
    receive a message from sock
```

Parameters

struct socket *sock

socket

struct msghdr *msg

message to receive

int flags

message flags

Receives **msg** from **sock**, passing through LSM. Returns the total number of bytes received, or an error.

```
int kernel_recvmsg(struct socket *sock, struct msghdr *msg, struct kvec *vec, size_t num,
                    size_t size, int flags)
```

Receive a message from a socket (kernel space)

Parameters

struct socket *sock

The socket to receive the message from

struct msghdr *msg

Received message

struct kvec *vec

Input s/g array for message data

size_t num

Size of input s/g array

size_t size

Number of bytes to read

int flags

Message flags (MSG_DONTWAIT, etc...)

On return the msg structure contains the scatter/gather array passed in the vec argument. The array is modified so that it consists of the unfilled portion of the original array.

The returned value is the total number of bytes received, or an error.

```
int sock_create_lite(int family, int type, int protocol, struct socket **res)  
    creates a socket
```

Parameters

int family
protocol family (AF_INET, ...)

int type
communication type (SOCK_STREAM, ...)

int protocol
protocol (0, ...)

struct socket **res
new socket

Creates a new socket and assigns it to **res**, passing through LSM. The new socket initialization is not complete, see [*kernel_accept\(\)*](#). Returns 0 or an error. On failure **res** is set to NULL. This function internally uses GFP_KERNEL.

```
int _sock_create(struct net *net, int family, int type, int protocol, struct socket **res, int  
    kern)  
    creates a socket
```

Parameters

struct net *net
net namespace

int family
protocol family (AF_INET, ...)

int type
communication type (SOCK_STREAM, ...)

int protocol
protocol (0, ...)

struct socket **res
new socket

int kern
boolean for kernel space sockets

Creates a new socket and assigns it to **res**, passing through LSM. Returns 0 or an error. On failure **res** is set to NULL. **kern** must be set to true if the socket resides in kernel space. This function internally uses GFP_KERNEL.

```
int sock_create(int family, int type, int protocol, struct socket **res)  
    creates a socket
```

Parameters

int family
protocol family (AF_INET, ...)

int type
communication type (SOCK_STREAM, ...)

```
int protocol
    protocol (0, ...)
```

```
struct socket **res
    new socket
```

A wrapper around `__sock_create()`. Returns 0 or an error. This function internally uses GFP_KERNEL.

```
int sock_create_kern(struct net *net, int family, int type, int protocol, struct socket **res)
    creates a socket (kernel space)
```

Parameters

```
struct net *net
    net namespace
```

```
int family
    protocol family (AF_INET, ...)
```

```
int type
    communication type (SOCK_STREAM, ...)
```

```
int protocol
    protocol (0, ...)
```

```
struct socket **res
    new socket
```

A wrapper around `__sock_create()`. Returns 0 or an error. This function internally uses GFP_KERNEL.

```
int sock_register(const struct net_proto_family *ops)
    add a socket protocol handler
```

Parameters

```
const struct net_proto_family *ops
    description of protocol
```

This function is called by a protocol handler that wants to advertise its address family, and have it linked into the socket interface. The value ops->family corresponds to the socket system call protocol family.

```
void sock_unregister(int family)
    remove a protocol handler
```

Parameters

```
int family
    protocol family to remove
```

This function is called by a protocol handler that wants to remove its address family, and have it unlinked from the new socket creation.

If protocol handler is a module, then it can use module reference counts to protect against new references. If protocol handler is not a module then it needs to provide its own protection in the ops->create routine.

int kernel_bind(struct *socket* *sock, struct sockaddr *addr, int addrlen)
bind an address to a socket (kernel space)

Parameters

struct socket *sock
socket

struct sockaddr *addr
address

int addrlen
length of address

Returns 0 or an error.

int kernel_listen(struct *socket* *sock, int backlog)
move socket to listening state (kernel space)

Parameters

struct socket *sock
socket

int backlog
pending connections queue size

Returns 0 or an error.

int kernel_accept(struct *socket* *sock, struct *socket* **newsock, int flags)
accept a connection (kernel space)

Parameters

struct socket *sock
listening socket

struct socket **newsock
new connected socket

int flags
flags

flags must be SOCK_CLOEXEC, SOCK_NONBLOCK or 0. If it fails, **newsock** is guaranteed to be NULL. Returns 0 or an error.

int kernel_connect(struct *socket* *sock, struct sockaddr *addr, int addrlen, int flags)
connect a socket (kernel space)

Parameters

struct socket *sock
socket

struct sockaddr *addr
address

int addrlen
address length

int flags

flags (O_NONBLOCK, ...)

For datagram sockets, **addr** is the address to which datagrams are sent by default, and the only address from which datagrams are received. For stream sockets, attempts to connect to **addr**. Returns 0 or an error code.

int kernel_getsockname(struct *socket* *sock, struct sockaddr *addr)

get the address which the socket is bound (kernel space)

Parameters**struct socket *sock**

socket

struct sockaddr *addr

address holder

Fills the **addr** pointer with the address which the socket is bound. Returns the length of the address in bytes or an error code.

int kernel_getpeername(struct *socket* *sock, struct sockaddr *addr)

get the address which the socket is connected (kernel space)

Parameters**struct socket *sock**

socket

struct sockaddr *addr

address holder

Fills the **addr** pointer with the address which the socket is connected. Returns the length of the address in bytes or an error code.

int kernel_sock_shutdown(struct *socket* *sock, enum *sock_shutdown_cmd* how)

shut down part of a full-duplex connection (kernel space)

Parameters**struct socket *sock**

socket

enum sock_shutdown_cmd how

connection part

Returns 0 or an error.

u32 kernel_sock_ip_overhead(struct *sock* *sk)

returns the IP overhead imposed by a socket

Parameters**struct sock *sk**

socket

This routine returns the IP overhead imposed by a socket i.e. the length of the underlying IP header, depending on whether this is an IPv4 or IPv6 socket and the length from IP options turned on at the socket. Assumes that the caller has a lock on the socket.

```
void drop_reasons_register_subsys(enum skb_drop_reason_subsys subsys, const struct
                                  drop_reason_list *list)
```

register another drop reason subsystem

Parameters

enum skb_drop_reason_subsys subsys

the subsystem to register, must not be the core

const struct drop_reason_list *list

the list of drop reasons within the subsystem, must point to a statically initialized list

```
void drop_reasons_unregister_subsys(enum skb_drop_reason_subsys subsys)
```

unregister a drop reason subsystem

Parameters

enum skb_drop_reason_subsys subsys

the subsystem to remove, must not be the core

Note

This will synchronize_rcu() to ensure no users when it returns.

```
struct sk_buff *build_skb_around(struct sk_buff *skb, void *data, unsigned int frag_size)
```

build a network buffer around provided skb

Parameters

struct sk_buff *skb

sk_buff provide by caller, must be memset cleared

void *data

data buffer provided by caller

unsigned int frag_size

size of data

```
struct sk_buff *napi_build_skb(void *data, unsigned int frag_size)
```

build a network buffer

Parameters

void *data

data buffer provided by caller

unsigned int frag_size

size of data

Description

Version of __napi_build_skb() that takes care of skb->head_frag and skb->pfrag_size when the data is a page or page fragment.

Returns a new *sk_buff* on success, NULL on allocation failure.

```
struct sk_buff *__alloc_skb(unsigned int size, gfp_t gfp_mask, int flags, int node)
```

allocate a network buffer

Parameters

unsigned int size
size to allocate

gfp_t gfp_mask
allocation mask

int flags

If SKB_ALLOC_FCLONE is set, allocate from fclone cache instead of head cache and allocate a cloned (child) skb. If SKB_ALLOC_RX is set, __GFP_MEMALLOC will be used for allocations in case the data is required for writeback

int node

numa node to allocate memory on

Allocate a new *sk_buff*. The returned buffer has no headroom and a tail room of at least size bytes. The object has a reference count of one. The return is the buffer. On a failure the return is NULL.

Buffers may only be allocated from interrupts using a **gfp mask** of GFP_ATOMIC.

struct sk_buff * __netdev_alloc_skb(struct net_device *dev, unsigned int len, gfp_t gfp_mask)

allocate an skbuff for rx on a specific device

Parameters

struct net_device *dev
network device to receive on

unsigned int len
length to allocate

gfp_t gfp_mask
get_free_pages mask, passed to alloc_skb

Allocate a new *sk_buff* and assign it a usage count of one. The buffer has NET_SKB_PAD headroom built in. Users should allocate the headroom they think they need without accounting for the built in space. The built in space is used for optimisations.

NULL is returned if there is no free memory.

struct sk_buff * __napi_alloc_skb(struct napi_struct *napi, unsigned int len, gfp_t gfp_mask)
allocate skbuff for rx in a specific NAPI instance

Parameters

struct napi_struct *napi
napi instance this buffer was allocated for

unsigned int len
length to allocate

gfp_t gfp_mask
get_free_pages mask, passed to alloc_skb and alloc_pages

Allocate a new *sk_buff* for use in NAPI receive. This buffer will attempt to allocate the head from a special reserved region used only for NAPI Rx allocation. By doing this we can save several CPU cycles by avoiding having to disable and re-enable IRQs.

NULL is returned if there is no free memory.

```
void __kfree_skb(struct sk_buff *skb)
    private function
```

Parameters

```
struct sk_buff *skb
    buffer
```

Free an sk_buff. Release anything attached to the buffer. Clean the state. This is an internal helper function. Users should always call kfree_skb

```
void __fix_address kfree_skb_reason(struct sk_buff *skb, enum skb_drop_reason reason)
    free an sk_buff with special reason
```

Parameters

```
struct sk_buff *skb
    buffer to free
```

```
enum skb_drop_reason reason
    reason why this skb is dropped
```

Drop a reference to the buffer and free it if the usage count has hit zero. Meanwhile, pass the drop reason to 'kfree_skb' tracepoint.

```
void skb_tx_error(struct sk_buff *skb)
    report an sk_buff xmit error
```

Parameters

```
struct sk_buff *skb
    buffer that triggered an error
```

Report xmit error if a device callback is tracking this skb. skb must be freed afterwards.

```
void consume_skb(struct sk_buff *skb)
    free an skbuff
```

Parameters

```
struct sk_buff *skb
    buffer to free
```

Drop a ref to the buffer and free it if the usage count has hit zero Functions identically to kfree_skb, but kfree_skb assumes that the frame is being dropped after a failure and notes that

```
struct sk_buff *alloc_skb_for_msg(struct sk_buff *first)
    allocate sk_buff to wrap frag list forming a msg
```

Parameters

```
struct sk_buff *first
    first sk_buff of the msg
```

```
struct sk_buff *skb_morph(struct sk_buff *dst, struct sk_buff *src)
    morph one skb into another
```

Parameters

struct sk_buff *dst
the skb to receive the contents

struct sk_buff *src
the skb to supply the contents

This is identical to skb_clone except that the target skb is supplied by the user.

The target skb is returned upon exit.

int **skb_copy_ubufs**(struct *sk_buff* *skb, gfp_t gfp_mask)
copy userspace skb frags buffers to kernel

Parameters

struct sk_buff *skb
the skb to modify

gfp_t gfp_mask
allocation priority

This must be called on skb with SKBFL_ZEROCPY_ENABLE. It will copy all frags into kernel and drop the reference to userspace pages.

If this function is called from an interrupt gfp_mask() must be GFP_ATOMIC.

Returns 0 on success or a negative error code on failure to allocate kernel memory to copy to.

struct *sk_buff* ***skb_clone**(struct *sk_buff* *skb, gfp_t gfp_mask)
duplicate an sk_buff

Parameters

struct sk_buff *skb
buffer to clone

gfp_t gfp_mask
allocation priority

Duplicate an *sk_buff*. The new one is not owned by a socket. Both copies share the same packet data but not structure. The new buffer has a reference count of 1. If the allocation fails the function returns NULL otherwise the new buffer is returned.

If this function is called from an interrupt gfp_mask() must be GFP_ATOMIC.

struct *sk_buff* ***skb_copy**(const struct *sk_buff* *skb, gfp_t gfp_mask)
create private copy of an sk_buff

Parameters

const struct sk_buff *skb
buffer to copy

gfp_t gfp_mask
allocation priority

Make a copy of both an *sk_buff* and its data. This is used when the caller wishes to modify the data and needs a private copy of the data to alter. Returns NULL on failure or the pointer to the buffer on success. The returned buffer has a reference count of 1.

As by-product this function converts non-linear *sk_buff* to linear one, so that *sk_buff* becomes completely private and caller is allowed to modify all the data of returned buffer. This means that this function is not recommended for use in circumstances when only header is going to be modified. Use pskb_copy() instead.

```
struct sk_buff *__pskb_copy_fclone(struct sk_buff *skb, int headroom, gfp_t gfp_mask, bool fclone)
```

create copy of an *sk_buff* with private head.

Parameters

struct sk_buff *skb

buffer to copy

int headroom

headroom of new skb

gfp_t gfp_mask

allocation priority

bool fclone

if true allocate the copy of the skb from the fclone cache instead of the head cache; it is recommended to set this to true for the cases where the copy will likely be cloned

Make a copy of both an *sk_buff* and part of its data, located in header. Fragmented data remain shared. This is used when the caller wishes to modify only header of *sk_buff* and needs private copy of the header to alter. Returns NULL on failure or the pointer to the buffer on success. The returned buffer has a reference count of 1.

```
int pskb_expand_head(struct sk_buff *skb, int nhead, int ntail, gfp_t gfp_mask)
```

reallocates header of *sk_buff*

Parameters

struct sk_buff *skb

buffer to reallocate

int nhead

room to add at head

int ntail

room to add at tail

gfp_t gfp_mask

allocation priority

Expands (or creates identical copy, if **nhead** and **ntail** are zero) header of **skb**. *sk_buff* itself is not changed. *sk_buff* MUST have reference count of 1. Returns zero in the case of success or error, if expansion failed. In the last case, *sk_buff* is not changed.

All the pointers pointing into skb header may change and must be reloaded after call to this function.

```
struct sk_buff *skb_expand_head(struct sk_buff *skb, unsigned int headroom)
```

reallocates header of *sk_buff*

Parameters

struct sk_buff *skb

buffer to reallocate

unsigned int headroom

needed headroom

Unlike skb_realloc_headroom, this one does not allocate a new skb if possible; copies skb->sk to new skb as needed and frees original skb in case of failures.

It expects increased headroom and generates warning otherwise.

```
struct sk_buff *skb_copy_expand(const struct sk_buff *skb, int newheadroom, int  
newtailroom, gfp_t gfp_mask)
```

copy and expand sk_buff

Parameters**const struct sk_buff *skb**

buffer to copy

int newheadroom

new free bytes at head

int newtailroom

new free bytes at tail

gfp_t gfp_mask

allocation priority

Make a copy of both an *sk_buff* and its data and while doing so allocate additional space.

This is used when the caller wishes to modify the data and needs a private copy of the data to alter as well as more space for new fields. Returns NULL on failure or the pointer to the buffer on success. The returned buffer has a reference count of 1.

You must pass GFP_ATOMIC as the allocation priority if this function is called from an interrupt.

```
int __skb_pad(struct sk_buff *skb, int pad, bool free_on_error)
```

zero pad the tail of an skb

Parameters**struct sk_buff *skb**

buffer to pad

int pad

space to pad

bool free_on_error

free buffer on error

Ensure that a buffer is followed by a padding area that is zero filled. Used by network drivers which may DMA or transfer data beyond the buffer end onto the wire.

May return error in out of memory cases. The skb is freed on error if **free_on_error** is true.

```
void *pskb_put(struct sk_buff *skb, struct sk_buff *tail, int len)
```

add data to the tail of a potentially fragmented buffer

Parameters

struct sk_buff *skb
start of the buffer to use

struct sk_buff *tail
tail fragment of the buffer to use

int len
amount of data to add

This function extends the used data area of the potentially fragmented buffer. **tail** must be the last fragment of **skb** -- or **skb** itself. If this would exceed the total buffer size the kernel will panic. A pointer to the first byte of the extra data is returned.

void *skb_put(struct sk_buff *skb, unsigned int len)
add data to a buffer

Parameters

struct sk_buff *skb
buffer to use

unsigned int len
amount of data to add

This function extends the used data area of the buffer. If this would exceed the total buffer size the kernel will panic. A pointer to the first byte of the extra data is returned.

void *skb_push(struct sk_buff *skb, unsigned int len)
add data to the start of a buffer

Parameters

struct sk_buff *skb
buffer to use

unsigned int len
amount of data to add

This function extends the used data area of the buffer at the buffer start. If this would exceed the total buffer headroom the kernel will panic. A pointer to the first byte of the extra data is returned.

void *skb_pull(struct sk_buff *skb, unsigned int len)
remove data from the start of a buffer

Parameters

struct sk_buff *skb
buffer to use

unsigned int len
amount of data to remove

This function removes data from the start of a buffer, returning the memory to the headroom. A pointer to the next data in the buffer is returned. Once the data has been pulled future pushes will overwrite the old data.

void *skb_pull_data(struct sk_buff *skb, size_t len)
remove data from the start of a buffer returning its original position.

Parameters**struct sk_buff *skb**

buffer to use

size_t len

amount of data to remove

This function removes data from the start of a buffer, returning the memory to the head-room. A pointer to the original data in the buffer is returned after checking if there is enough data to pull. Once the data has been pulled future pushes will overwrite the old data.

void **skb_trim**(struct *sk_buff* *skb, unsigned int len)

remove end from a buffer

Parameters**struct sk_buff *skb**

buffer to alter

unsigned int len

new length

Cut the length of a buffer down by removing data from the tail. If the buffer is already under the length specified it is not modified. The skb must be linear.

void * **__pskb_pull_tail**(struct *sk_buff* *skb, int delta)

advance tail of skb header

Parameters**struct sk_buff *skb**

buffer to reallocate

int delta

number of bytes to advance tail

The function makes a sense only on a fragmented *sk_buff*, it expands header moving its tail forward and copying necessary data from fragmented part.

sk_buff MUST have reference count of 1.

Returns NULL (and *sk_buff* does not change) if pull failed or value of new tail of skb in the case of success.

All the pointers pointing into skb header may change and must be reloaded after call to this function.

int **skb_copy_bits**(const struct *sk_buff* *skb, int offset, void *to, int len)

copy bits from skb to kernel buffer

Parameters**const struct sk_buff *skb**

source skb

int offset

offset in source

```
void *to
      destination buffer

int len
      number of bytes to copy

Copy the specified number of bytes from the source skb to the destination buffer.

CAUTION ! :
      If its prototype is ever changed, check arch/{*}/net/{*}.S files, since it is called from
      BPF assembly code.
```

```
int skb_store_bits(struct sk_buff *skb, int offset, const void *from, int len)
      store bits from kernel buffer to skb
```

Parameters

```
struct sk_buff *skb
      destination buffer

int offset
      offset in destination

const void *from
      source buffer

int len
      number of bytes to copy

Copy the specified number of bytes from the source buffer to the destination skb. This
function handles all the messy bits of traversing fragment lists and such.
```

```
int skb_zerocopy(struct sk_buff *to, struct sk_buff *from, int len, int hlen)
      Zero copy skb to skb
```

Parameters

```
struct sk_buff *to
      destination buffer

struct sk_buff *from
      source buffer

int len
      number of bytes to copy from source buffer

int hlen
      size of linear headroom in destination buffer

Copies up to len bytes from from to to by creating references to the frags in the source
buffer.
```

The *hlen* as calculated by *skb_zerocopy_headlen()* specifies the headroom in the *to* buffer.

Return value: 0: everything is OK -ENOMEM: couldn't orphan frags of **from** due to lack
of memory -EFAULT: *skb_copy_bits()* found some problem with skb geometry

```
struct sk_buff *skb_dequeue(struct sk_buff_head *list)
      remove from the head of the queue
```

Parameters

struct sk_buff_head *list
list to dequeue from

Remove the head of the list. The list lock is taken so the function may be used safely with other locking list functions. The head item is returned or NULL if the list is empty.

struct sk_buff *skb_dequeue_tail(struct sk_buff_head *list)
remove from the tail of the queue

Parameters

struct sk_buff_head *list
list to dequeue from

Remove the tail of the list. The list lock is taken so the function may be used safely with other locking list functions. The tail item is returned or NULL if the list is empty.

void skb_queue_purge_reason(struct sk_buff_head *list, enum skb_drop_reason reason)
empty a list

Parameters

struct sk_buff_head *list
list to empty

enum skb_drop_reason reason
drop reason

Delete all buffers on an *sk_buff* list. Each buffer is removed from the list and one reference dropped. This function takes the list lock and is atomic with respect to other list locking functions.

void skb_queue_head(struct sk_buff_head *list, struct sk_buff *newsk)
queue a buffer at the list head

Parameters

struct sk_buff_head *list
list to use

struct sk_buff *newsk
buffer to queue

Queue a buffer at the start of the list. This function takes the list lock and can be used safely with other locking *sk_buff* functions safely.

A buffer cannot be placed on two lists at the same time.

void skb_queue_tail(struct sk_buff_head *list, struct sk_buff *newsk)
queue a buffer at the list tail

Parameters

struct sk_buff_head *list
list to use

struct sk_buff *newsk
buffer to queue

Queue a buffer at the tail of the list. This function takes the list lock and can be used safely with other locking *sk_buff* functions safely.

A buffer cannot be placed on two lists at the same time.

```
void skb_unlink(struct sk_buff *skb, struct sk_buff_head *list)  
    remove a buffer from a list
```

Parameters

struct sk_buff *skb
buffer to remove

struct sk_buff_head *list
list to use

Remove a packet from a list. The list locks are taken and this function is atomic with respect to other list locked calls

You must know what list the SKB is on.

```
void skb_append(struct sk_buff *old, struct sk_buff *newsk, struct sk_buff_head *list)  
    append a buffer
```

Parameters

struct sk_buff *old
buffer to insert after

struct sk_buff *newsk
buffer to insert

struct sk_buff_head *list
list to use

Place a packet after a given packet in a list. The list locks are taken and this function is atomic with respect to other list locked calls. A buffer cannot be placed on two lists at the same time.

```
void skb_split(struct sk_buff *skb, struct sk_buff *skb1, const u32 len)
```

Split fragmented skb to two parts at length len.

Parameters

struct sk_buff *skb
the buffer to split

struct sk_buff *skb1
the buffer to receive the second part

const u32 len
new length for skb

```
void skb_prepare_seq_read(struct sk_buff *skb, unsigned int from, unsigned int to, struct  
                           skb_seq_state *st)
```

Prepare a sequential read of skb data

Parameters

struct sk_buff *skb
the buffer to read

unsigned int from
lower offset of data to be read

```
unsigned int to
    upper offset of data to be read
struct skb_seq_state *st
    state variable
```

Description

Initializes the specified state variable. Must be called before invoking `skb_seq_read()` for the first time.

```
unsigned int skb_seq_read(unsigned int consumed, const u8 **data, struct skb_seq_state *st)
    Sequentially read skb data
```

Parameters

```
unsigned int consumed
    number of bytes consumed by the caller so far
const u8 **data
    destination pointer for data to be returned
struct skb_seq_state *st
    state variable
```

Description

Reads a block of skb data at **consumed** relative to the lower offset specified to `skb_prepare_seq_read()`. Assigns the head of the data block to **data** and returns the length of the block or 0 if the end of the skb data or the upper offset has been reached.

The caller is not required to consume all of the data returned, i.e. **consumed** is typically set to the number of bytes already consumed and the next call to `skb_seq_read()` will return the remaining part of the block.

Note 1: The size of each block of data returned can be arbitrary,
this limitation is the cost for zero-copy sequential reads of potentially non linear data.

Note 2: Fragment lists within fragments are not implemented
at the moment, state->root_skb could be replaced with a stack for this purpose.

```
void skb_abort_seq_read(struct skb_seq_state *st)
    Abort a sequential read of skb data
```

Parameters

```
struct skb_seq_state *st
    state variable
```

Description

Must be called if `skb_seq_read()` was not called until it returned 0.

```
unsigned int skb_find_text(struct sk_buff *skb, unsigned int from, unsigned int to, struct ts_config *config)
```

Find a text pattern in skb data

Parameters

```
struct sk_buff *skb
    the buffer to look in
```

```
unsigned int from
    search offset

unsigned int to
    search limit

struct ts_config *config
    textsearch configuration
```

Description

Finds a pattern in the skb data according to the specified textsearch configuration. Use textsearch_next() to retrieve subsequent occurrences of the pattern. Returns the offset to the first occurrence or UINT_MAX if no match was found.

```
void *skb_pull_rcsum(struct sk_buff *skb, unsigned int len)
    pull skb and update receive checksum
```

Parameters

struct sk_buff *skb
buffer to update

unsigned int len
length of data pulled

This function performs an skb_pull on the packet and updates the CHECKSUM_COMPLETE checksum. It should be used on receive path processing instead of skb_pull unless you know that the checksum difference is zero (e.g., a valid IP header) or you are setting ip_summed to CHECKSUM_NONE.

```
struct sk_buff *skb_segment(struct sk_buff *head_skb, netdev_features_t features)
    Perform protocol segmentation on skb.
```

Parameters

struct sk_buff *head_skb
buffer to segment

netdev_features_t features
features for the output path (see dev->features)

This function performs segmentation on the given skb. It returns a pointer to the first in a list of new skbs for the segments. In case of error it returns ERR_PTR(err).

```
int skb_to_sgvec(struct sk_buff *skb, struct scatterlist *sg, int offset, int len)
    Fill a scatter-gather list from a socket buffer
```

Parameters

struct sk_buff *skb
Socket buffer containing the buffers to be mapped

struct scatterlist *sg
The scatter-gather list to map into

int offset
The offset into the buffer's contents to start mapping

int len
Length of buffer space to be mapped

Fill the specified scatter-gather list with mappings/pointers into a region of the buffer space attached to a socket buffer. Returns either the number of scatterlist items used, or -EMSGSIZE if the contents could not fit.

int skb_cow_data(struct sk_buff *skb, int tailbits, struct sk_buff **trailer)

Check that a socket buffer's data buffers are writable

Parameters

struct sk_buff *skb

The socket buffer to check.

int tailbits

Amount of trailing space to be added

struct sk_buff **trailer

Returned pointer to the skb where the **tailbits** space begins

Make sure that the data buffers attached to a socket buffer are writable. If they are not, private copies are made of the data buffers and the socket buffer is set to use these instead.

If **tailbits** is given, make sure that there is space to write **tailbits** bytes of data beyond current end of socket buffer. **trailer** will be set to point to the skb in which this space begins.

The number of scatterlist elements required to completely map the COW'd and extended socket buffer will be returned.

struct sk_buff *skb_clone_sk(struct sk_buff *skb)

create clone of skb, and take reference to socket

Parameters

struct sk_buff *skb

the skb to clone

Description

This function creates a clone of a buffer that holds a reference on sk_refcnt. Buffers created via this function are meant to be returned using sock_queue_err_skb, or free via kfree_skb.

When passing buffers allocated with this function to sock_queue_err_skb it is necessary to wrap the call with sock_hold/sock_put in order to prevent the socket from being released prior to being enqueued on the sk_error_queue.

bool skb_partial_csum_set(struct sk_buff *skb, u16 start, u16 off)

set up and verify partial csum values for packet

Parameters

struct sk_buff *skb

the skb to set

u16 start

the number of bytes after skb->data to start checksumming.

u16 off

the offset from start to place the checksum.

Description

For untrusted partially-checksummed packets, we need to make sure the values for skb->csum_start and skb->csum_offset are valid so we don't oops.

This function checks and sets those values and skb->ip_summed: if this returns false you should drop the packet.

int `skb_checksum_setup`(struct *sk_buff* *skb, bool recalculate)

 set up partial checksum offset

Parameters

struct *sk_buff* *skb

 the skb to set up

bool recalculate

 if true the pseudo-header checksum will be recalculated

**struct *sk_buff* *`skb_checksum_trimmed`(struct *sk_buff* *skb, unsigned int transport_len,
 __sum16 (*skb_chkf)(struct *sk_buff* *skb))**

 validate checksum of an skb

Parameters

struct *sk_buff* *skb

 the skb to check

unsigned int transport_len

 the data length beyond the network header

__sum16(*skb_chkf)(struct *sk_buff* *skb)

 checksum function to use

Description

Applies the given checksum function skb_chkf to the provided skb. Returns a checked and maybe trimmed skb. Returns NULL on error.

If the skb has data beyond the given transport length, then a trimmed & cloned skb is checked and returned.

Caller needs to set the skb transport header and free any returned skb if it differs from the provided skb.

**bool `skb_try_coalesce`(struct *sk_buff* *to, struct *sk_buff* *from, bool *fragstolen, int
 *delta_truesize)**

 try to merge skb to prior one

Parameters

struct *sk_buff* *to

 prior buffer

struct *sk_buff* *from

 buffer to add

bool *fragstolen

 pointer to boolean

int *delta_truesize

 how much more was allocated than was requested

```
void skb_scrub_packet(struct sk_buff *skb, bool xnet)
    scrub an skb
```

Parameters

struct sk_buff *skb
buffer to clean

bool xnet
packet is crossing netns

Description

`skb_scrub_packet` can be used after encapsulating or decapsulating a packet into/from a tunnel. Some information have to be cleared during these operations. `skb_scrub_packet` can also be used to clean a `skb` before injecting it in another namespace (`xnet == true`). We have to clear all information in the `skb` that could impact namespace isolation.

```
int skb_eth_pop(struct sk_buff *skb)
    Drop the Ethernet header at the head of a packet
```

Parameters

struct sk_buff *skb
Socket buffer to modify

Description

Drop the Ethernet header of **skb**.

Expects that `skb->data` points to the mac header and that no VLAN tags are present.

Returns 0 on success, -errno otherwise.

```
int skb_eth_push(struct sk_buff *skb, const unsigned char *dst, const unsigned char *src)
    Add a new Ethernet header at the head of a packet
```

Parameters

struct sk_buff *skb
Socket buffer to modify

const unsigned char *dst
Destination MAC address of the new header

const unsigned char *src
Source MAC address of the new header

Description

Prepend **skb** with a new Ethernet header.

Expects that `skb->data` points to the mac header, which must be empty.

Returns 0 on success, -errno otherwise.

```
int skb_mpls_push(struct sk_buff *skb, __be32 mpls_lse, __be16 mpls_proto, int mac_len, bool ethernet)
    push a new MPLS header after mac_len bytes from start of the packet
```

Parameters

```
struct sk_buff *skb
    buffer

__be32 mpls_lse
    MPLS label stack entry to push

__be16 mpls_proto
    ethertype of the new MPLS header (expects 0x8847 or 0x8848)

int mac_len
    length of the MAC header

bool ethernet
    flag to indicate if the resulting packet after skb_mpls_push is ethernet
```

Description

Expects skb->data at mac header.

Returns 0 on success, -errno otherwise.

```
int skb_mpls_pop(struct sk_buff *skb, __be16 next_proto, int mac_len, bool ethernet)
    pop the outermost MPLS header
```

Parameters

```
struct sk_buff *skb
    buffer

__be16 next_proto
    ethertype of header after popped MPLS header

int mac_len
    length of the MAC header

bool ethernet
    flag to indicate if the packet is ethernet
```

Description

Expects skb->data at mac header.

Returns 0 on success, -errno otherwise.

```
int skb_mpls_update_lse(struct sk_buff *skb, __be32 mpls_lse)
    modify outermost MPLS header and update csum
```

Parameters

```
struct sk_buff *skb
    buffer

__be32 mpls_lse
    new MPLS label stack entry to update to
```

Description

Expects skb->data at mac header.

Returns 0 on success, -errno otherwise.

```
int skb_mpls_dec_ttl(struct sk_buff *skb)
    decrement the TTL of the outermost MPLS header
```

Parameters

struct sk_buff *skb
buffer

Description

Expects skb->data at mac header.

Returns 0 on success, -errno otherwise.

```
struct sk_buff *alloc_skb_with frags(unsigned long header_len, unsigned long data_len, int
order, int *errcode, gfp_t gfp_mask)
    allocate skb with page frags
```

Parameters

unsigned long header_len
size of linear part

unsigned long data_len
needed length in frags

int order
max page order desired.

int *errcode
pointer to error code if any

gfp_t gfp_mask
allocation mask

Description

This can be used to allocate a paged skb, given a maximal order for frags.

```
void skb_condense(struct sk_buff *skb)
    try to get rid of fragments/frag_list if possible
```

Parameters

struct sk_buff *skb
buffer

Description

Can be used to save memory before skb is added to a busy queue. If packet has bytes in frags and enough tail room in skb->head, pull all of them, so that we can free the frags right now and adjust truesize.

Notes

We do not reallocate skb->head thus can not fail. Caller must re-evaluate skb->truesize if needed.

```
void *skb_ext_add(struct sk_buff *skb, enum skb_ext_id id)
    allocate space for given extension, COW if needed
```

Parameters

```
struct sk_buff *skb
    buffer

enum skb_ext_id id
    extension to allocate space for
```

Description

Allocates enough space for the given extension. If the extension is already present, a pointer to that extension is returned.

If the skb was cloned, COW applies and the returned memory can be modified without changing the extension space of clones buffers.

Returns pointer to the extension or NULL on allocation failure.

```
ssize_t skb_splice_from_iter(struct sk_buff *skb, struct iov_iter *iter, ssize_t maxsize, gfp_t gfp)
```

Splice (or copy) pages to skbuff

Parameters

```
struct sk_buff *skb
    The buffer to add pages to
```

```
struct iov_iter *iter
    Iterator representing the pages to be added
```

```
ssize_t maxsize
    Maximum amount of pages to be added
```

```
gfp_t gfp
    Allocation flags
```

Description

This is a common helper function for supporting MSG_SPLICE_PAGES. It extracts pages from an iterator and adds them to the socket buffer if possible, copying them to fragments if not possible (such as if they're slab pages).

Returns the amount of data spliced/copied or -EMSGSIZE if there's insufficient space in the buffer to transfer anything.

```
bool sk_ns_capable(const struct sock *sk, struct user_namespace *user_ns, int cap)
    General socket capability test
```

Parameters

```
const struct sock *sk
    Socket to use a capability on or through
```

```
struct user_namespace *user_ns
    The user namespace of the capability to use
```

```
int cap
    The capability to use
```

Description

Test to see if the opener of the socket had when the socket was created and the current process has the capability **cap** in the user namespace **user_ns**.

bool sk_capable(const struct *sock* *sk, int cap)

Socket global capability test

Parameters

const struct sock *sk

Socket to use a capability on or through

int cap

The global capability to use

Description

Test to see if the opener of the socket had when the socket was created and the current process has the capability **cap** in all user namespaces.

bool sk_net_capable(const struct *sock* *sk, int cap)

Network namespace socket capability test

Parameters

const struct sock *sk

Socket to use a capability on or through

int cap

The capability to use

Description

Test to see if the opener of the socket had when the socket was created and the current process has the capability **cap** over the network namespace the socket is a member of.

void sk_set_memalloc(struct *sock* *sk)

sets SOCK_MEMALLOC

Parameters

struct sock *sk

socket to set it on

Description

Set SOCK_MEMALLOC on a socket for access to emergency reserves. It's the responsibility of the admin to adjust min_free_kbytes to meet the requirements

struct *sock* *sk_alloc(struct *net* *net, int family, gfp_t priority, struct proto *prot, int kern)

All socket objects are allocated here

Parameters

struct net *net

the applicable net namespace

int family

protocol family

gfp_t priority

for allocation (GFP_KERNEL, GFP_ATOMIC, etc)

struct proto *prot

struct proto associated with this new sock instance

int kern

is this to be a kernel socket?

struct *sock* *sk_clone_lock(const struct *sock* *sk, const gfp_t priority)

clone a socket, and lock its clone

Parameters

const struct sock *sk

the socket to clone

const gfp_t priority

for allocation (GFP_KERNEL, GFP_ATOMIC, etc)

Caller must unlock socket even in error path (bh_unlock_sock(newsk))

bool skb_page_frag_refill(unsigned int sz, struct page_frag *pfrag, gfp_t gfp)

check that a page_frag contains enough room

Parameters

unsigned int sz

minimum size of the fragment we want to get

struct page_frag *pfrag

pointer to page_frag

gfp_t gfp

priority for memory allocation

Note

While this allocator tries to use high order pages, there is no guarantee that allocations succeed. Therefore, **sz** MUST be less or equal than PAGE_SIZE.

int sk_wait_data(struct *sock* *sk, long *timeo, const struct *sk_buff* *skb)

wait for data to arrive at sk_receive_queue

Parameters

struct sock *sk

sock to wait on

long *timeo

for how long

const struct sk_buff *skb

last skb seen on sk_receive_queue

Description

Now socket state including sk->sk_err is changed only under lock, hence we may omit checks after joining wait queue. We check receive queue before schedule() only as optimization; it is very likely that release_sock() added new data.

int __sk_mem_schedule(struct *sock* *sk, int size, int kind)

increase sk_forward_alloc and memory_allocated

Parameters

struct sock *sk

socket

int size
memory size to allocate

int kind
allocation type

If kind is SK_MEM_SEND, it means wmem allocation. Otherwise it means rmem allocation. This function assumes that protocols which have memory_pressure use sk_wmem_queued as write buffer accounting.

void __sk_mem_reclaim(struct sock *sk, int amount)
reclaim sk_forward_alloc and memory_allocated

Parameters

struct sock *sk
socket

int amount
number of bytes (rounded down to a PAGE_SIZE multiple)

struct sk_buff *__skb_try_recv_datagram(struct sock *sk, struct sk_buff_head *queue, unsigned int flags, int *off, int *err, struct sk_buff **last)

Receive a datagram skbuff

Parameters

struct sock *sk
socket

struct sk_buff_head *queue
socket queue from which to receive

unsigned int flags
MSG_flags

int *off
an offset in bytes to peek skb from. Returns an offset within an skb where data actually starts

int *err
error code returned

struct sk_buff **last
set to last peeked message to inform the wait function what to look for when peeking

Get a datagram skbuff, understands the peeking, nonblocking wakeups and possible races. This replaces identical code in packet, raw and udp, as well as the IPX AX.25 and Appletalk. It also finally fixes the long standing peek and read race for datagram sockets. If you alter this routine remember it must be re-entrant.

This function will lock the socket if a skb is returned, so the caller needs to unlock the socket in that case (usually by calling skb_free_datagram). Returns NULL with err set to -EAGAIN if no data was available or to some other value if an error was detected.

- It does not lock socket since today. This function is
- free of race conditions. This measure should/can improve

- significantly datagram socket latencies at high loads,
- when data copying to user space takes lots of time.
- (BTW I've just killed the last cli() in IP/IPv6/core/netlink/packet
- 8) Great win.)
- --ANK (980729)

The order of the tests when we find no data waiting are specified quite explicitly by POSIX 1003.1g, don't change them without having the standard around please.

```
int skb_kill_datagram(struct sock *sk, struct sk_buff *skb, unsigned int flags)
```

Free a datagram skbuff forcibly

Parameters

struct sock *sk
socket

struct sk_buff *skb
datagram skbuff

unsigned int flags
MSG_flags

This function frees a datagram skbuff that was received by skb_recv_datagram. The flags argument must match the one used for skb_recv_datagram.

If the MSG_PEEK flag is set, and the packet is still on the receive queue of the socket, it will be taken off the queue before it is freed.

This function currently only disables BH when acquiring the sk_receive_queue lock. Therefore it must not be used in a context where that lock is acquired in an IRQ context.

It returns 0 if the packet was removed by us.

```
int skb_copy_and_hash_datagram_iter(const struct sk_buff *skb, int offset, struct iov_iter *to, int len, struct ahash_request *hash)
```

Copy datagram to an iovec iterator and update a hash.

Parameters

const struct sk_buff *skb
buffer to copy

int offset
offset in the buffer to start copying from

struct iov_iter *to
iovec iterator to copy to

int len
amount of data to copy from buffer to iovec

struct ahash_request *hash
hash request to update

```
int skb_copy_datagram_iter(const struct sk_buff *skb, int offset, struct iov_iter *to, int len)
```

Copy a datagram to an iovec iterator.

Parameters**const struct sk_buff *skb**

buffer to copy

int offset

offset in the buffer to start copying from

struct iov_iter *to

iovec iterator to copy to

int len

amount of data to copy from buffer to iovec

int **skb_copy_datagram_from_iter**(struct *sk_buff* *skb, int offset, struct iov_iter *from, int len)

Copy a datagram from an iov_iter.

Parameters**struct sk_buff *skb**

buffer to copy

int offset

offset in the buffer to start copying to

struct iov_iter *from

the copy source

int len

amount of data to copy to buffer from iovec

Returns 0 or -EFAULT.

int **zerocopy_sg_from_iter**(struct *sk_buff* *skb, struct iov_iter *from)

Build a zerocopy datagram from an iov_iter

Parameters**struct sk_buff *skb**

buffer to copy

struct iov_iter *from

the source to copy from

The function will first copy up to headlen, and then pin the userspace pages and build frags through them.

Returns 0, -EFAULT or -EMSGSIZE.

int **skb_copy_and_csum_datagram_msg**(struct *sk_buff* *skb, int hlen, struct msghdr *msg)

Copy and checksum skb to user iovec.

Parameters**struct sk_buff *skb**

skbuff

int hlen

hardware length

struct msghdr *msg
destination
Caller _must_ check that skb will fit to this iovec.

Return

0 - success.

-EINVAL - checksum failure. -EFAULT - fault during copy.

_poll_t datagram_poll(struct file *file, struct socket *sock, poll_table *wait)
generic datagram poll

Parameters

struct file *file
file struct

struct socket *sock
socket

poll_table *wait
poll table

Datagram poll: Again totally generic. This also handles sequenced packet sockets providing the socket receive queue is only ever holding data ready to receive.

Note

when you **don't** use this routine for this protocol,

and you use a different write policy from sock_writeable() then please supply your own write_space callback.

int sk_stream_wait_connect(struct sock *sk, long *timeo_p)
Wait for a socket to get into the connected state

Parameters

struct sock *sk
sock to wait on

long *timeo_p
for how long to wait

Description

Must be called with the socket locked.

int sk_stream_wait_memory(struct sock *sk, long *timeo_p)
Wait for more memory for a socket

Parameters

struct sock *sk
socket to wait for memory

long *timeo_p
for how long

13.1.3 Socket Filter

```
int sk_filter_trim_cap(struct sock *sk, struct sk_buff *skb, unsigned int cap)
    run a packet through a socket filter
```

Parameters

struct sock *sk
sock associated with *sk_buff*

struct sk_buff *skb
buffer to filter

unsigned int cap
limit on how short the eBPF program may trim the packet

Description

Run the eBPF program and then cut skb->data to correct size returned by the program. If pkt_len is 0 we toss packet. If skb->len is smaller than pkt_len we keep whole skb->data. This is the socket level wrapper to bpf_prog_run. It returns 0 if the packet should be accepted or -EPERM if the packet should be tossed.

```
int bpf_prog_create(struct bpf_prog **pfp, struct sock_fprog_kern *fprog)
    create an unattached filter
```

Parameters

struct bpf_prog **pfp
the unattached filter that is created

struct sock_fprog_kern *fprog
the filter program

Description

Create a filter independent of any socket. We first run some sanity checks on it to make sure it does not explode on us later. If an error occurs or there is insufficient memory for the filter a negative errno code is returned. On success the return is zero.

```
int bpf_prog_create_from_user(struct bpf_prog **pfp, struct sock_fprog *fprog,
                           bpf_aux_classic_check_t trans, bool save_orig)
    create an unattached filter from user buffer
```

Parameters

struct bpf_prog **pfp
the unattached filter that is created

struct sock_fprog *fprog
the filter program

bpf_aux_classic_check_t trans
post-classic verifier transformation handler

bool save_orig
save classic BPF program

Description

This function effectively does the same as [*bpf_prog_create\(\)*](#), only that it builds up its insns buffer from user space provided buffer. It also allows for passing a bpf_aux_classic_check_t handler.

```
int sk_attach_filter(struct sock_fprog *fprog, struct sock *sk)
    attach a socket filter
```

Parameters

struct sock_fprog *fprog
the filter program

struct sock *sk
the socket to use

Description

Attach the user's filter code. We first run some sanity checks on it to make sure it does not explode on us later. If an error occurs or there is insufficient memory for the filter a negative errno code is returned. On success the return is zero.

13.1.4 Generic Network Statistics

```
struct gnet_stats_basic
    byte/packet throughput statistics
```

Definition:

```
struct gnet_stats_basic {
    __u64 bytes;
    __u32 packets;
};
```

Members

bytes
number of seen bytes

packets
number of seen packets

struct gnet_stats_rate_est
rate estimator

Definition:

```
struct gnet_stats_rate_est {
    __u32 bps;
    __u32 pps;
};
```

Members

bps
current byte rate

pps
current packet rate

struct gnet_stats_rate_est64
rate estimator

Definition:

```
struct gnet_stats_rate_est64 {
    __u64 bps;
    __u64 pps;
};
```

Members

bps
current byte rate

pps
current packet rate

struct gnet_stats_queue
queuing statistics

Definition:

```
struct gnet_stats_queue {
    __u32 qlen;
    __u32 backlog;
    __u32 drops;
    __u32 requeues;
    __u32 overlimits;
};
```

Members

qlen
queue length

backlog
backlog size of queue

drops
number of dropped packets

requeues
number of requeues

overlimits
number of enqueues over the limit

struct gnet_estimator
rate estimator configuration

Definition:

```
struct gnet_estimator {
    signed char      interval;
    unsigned char    ewma_log;
};
```

Members**interval**

sampling period

ewma_log

the log of measurement window weight

```
int gnet_stats_start_copy_compat(struct sk_buff *skb, int type, int tc_stats_type, int
                                  xstats_type, spinlock_t *lock, struct gnet_dump *d, int
                                  padattr)
```

start dumping procedure in compatibility mode

Parameters**struct sk_buff *skb**

socket buffer to put statistics TLVs into

int type

TLV type for top level statistic TLV

int tc_stats_type

TLV type for backward compatibility struct tc_stats TLV

int xstats_type

TLV type for backward compatibility xstats TLV

spinlock_t *lock

statistics lock

struct gnet_dump *d

dumping handle

int padattr

padding attribute

Description

Initializes the dumping handle, grabs the statistic lock and appends an empty TLV header to the socket buffer for use a container for all other statistic TLVs.

The dumping handle is marked to be in backward compatibility mode telling all gnet_stats_copy_XXX() functions to fill a local copy of struct tc_stats.

Returns 0 on success or -1 if the room in the socket buffer was not sufficient.

```
int gnet_stats_start_copy(struct sk_buff *skb, int type, spinlock_t *lock, struct gnet_dump
                           *d, int padattr)
```

start dumping procedure in compatibility mode

Parameters**struct sk_buff *skb**

socket buffer to put statistics TLVs into

```
int type
    TLV type for top level statistic TLV

spinlock_t *lock
    statistics lock

struct gnet_dump *d
    dumping handle

int padattr
    padding attribute
```

Description

Initializes the dumping handle, grabs the statistic lock and appends an empty TLV header to the socket buffer for use a container for all other statistic TLVs.

Returns 0 on success or -1 if the room in the socket buffer was not sufficient.

```
int gnet_stats_copy_basic(struct gnet_dump *d, struct gnet_stats_basic_sync __percpu
                           *cpu, struct gnet_stats_basic_sync *b, bool running)
    copy basic statistics into statistic TLV
```

Parameters

```
struct gnet_dump *d
    dumping handle

struct gnet_stats_basic_sync __percpu *cpu
    copy statistic per cpu

struct gnet_stats_basic_sync *b
    basic statistics

bool running
    true if b represents a running qdisc, thus b's internal values might change during basic reads. Only used if cpu is NULL
```

Context

task; must not be run from IRQ or BH contexts

Description

Appends the basic statistics to the top level TLV created by [*gnet_stats_start_copy\(\)*](#).

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

```
int gnet_stats_copy_basic_hw(struct gnet_dump *d, struct gnet_stats_basic_sync __percpu
                            *cpu, struct gnet_stats_basic_sync *b, bool running)
    copy basic hw statistics into statistic TLV
```

Parameters

```
struct gnet_dump *d
    dumping handle

struct gnet_stats_basic_sync __percpu *cpu
    copy statistic per cpu
```

```
struct gnet_stats_basic_sync *b
    basic statistics

bool running
    true if b represents a running qdisc, thus b's internal values might change during basic
    reads. Only used if cpu is NULL
```

Context

task; must not be run from IRQ or BH contexts

Description

Appends the basic statistics to the top level TLV created by [*gnet_stats_start_copy\(\)*](#).

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

```
int gnet_stats_copy_rate_est(struct gnet_dump *d, struct net_rate_estimator __rcu
    **rate_est)
```

copy rate estimator statistics into statistics TLV

Parameters

```
struct gnet_dump *d
    dumping handle

struct net_rate_estimator __rcu **rate_est
    rate estimator
```

Description

Appends the rate estimator statistics to the top level TLV created by [*gnet_stats_start_copy\(\)*](#).

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

```
int gnet_stats_copy_queue(struct gnet_dump *d, struct gnet_stats_queue __percpu *cpu_q,
    struct gnet_stats_queue *q, __u32 qlen)
    copy queue statistics into statistics TLV
```

Parameters

```
struct gnet_dump *d
    dumping handle

struct gnet_stats_queue __percpu *cpu_q
    per cpu queue statistics

struct gnet_stats_queue *q
    queue statistics

__u32 qlen
    queue length statistics
```

Description

Appends the queue statistics to the top level TLV created by [*gnet_stats_start_copy\(\)*](#). Using per cpu queue statistics if they are available.

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

```
int gnet_stats_copy_app(struct gnet_dump *d, void *st, int len)
    copy application specific statistics into statistics TLV
```

Parameters

struct gnet_dump *d
dumping handle

void *st
application specific statistics data

int len
length of data

Description

Appends the application specific statistics to the top level TLV created by [gnet_stats_start_copy\(\)](#) and remembers the data for XSTATS if the dumping handle is in backward compatibility mode.

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

```
int gnet_stats_finish_copy(struct gnet_dump *d)
    finish dumping procedure
```

Parameters

struct gnet_dump *d
dumping handle

Description

Corrects the length of the top level TLV to include all TLVs added by gnet_stats_copy_XXX() calls. Adds the backward compatibility TLVs if [gnet_stats_start_copy_compat\(\)](#) was used and releases the statistics lock.

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

```
int gen_new_estimator(struct gnet_stats_basic_sync *bstats, struct gnet_stats_basic_sync __percpu *cpu_bstats, struct net_rate_estimator __rcu **rate_est, spinlock_t *lock, bool running, struct nlattr *opt)
    create a new rate estimator
```

Parameters

struct gnet_stats_basic_sync *bstats
basic statistics

struct gnet_stats_basic_sync __percpu *cpu_bstats
bstats per cpu

struct net_rate_estimator __rcu **rate_est
rate estimator statistics

spinlock_t *lock
lock for statistics and control path

bool running

true if **bstats** represents a running qdisc, thus **bstats**' internal values might change during basic reads. Only used if **bstats_cpu** is NULL

struct nlastr *opt

rate estimator configuration TLV

Description

Creates a new rate estimator with **bstats** as source and **rate_est** as destination. A new timer with the interval specified in the configuration TLV is created. Upon each interval, the latest statistics will be read from **bstats** and the estimated rate will be stored in **rate_est** with the statistics lock grabbed during this period.

Returns 0 on success or a negative error code.

```
void gen_kill_estimator(struct net_rate_estimator __rcu **rate_est)
```

remove a rate estimator

Parameters**struct net_rate_estimator __rcu **rate_est**

rate estimator

Description

Removes the rate estimator.

```
int gen_replace_estimator(struct gnet_stats_basic_sync *bstats, struct  
                         gnet_stats_basic_sync __percpu *cpu_bstats, struct  
                         net_rate_estimator __rcu **rate_est, spinlock_t *lock, bool  
                         running, struct nlastr *opt)
```

replace rate estimator configuration

Parameters**struct gnet_stats_basic_sync *bstats**

basic statistics

struct gnet_stats_basic_sync __percpu *cpu_bstats

bstats per cpu

struct net_rate_estimator __rcu **rate_est

rate estimator statistics

spinlock_t *lock

lock for statistics and control path

bool running

true if **bstats** represents a running qdisc, thus **bstats**' internal values might change during basic reads. Only used if **cpu_bstats** is NULL

struct nlastr *opt

rate estimator configuration TLV

Description

Replaces the configuration of a rate estimator by calling [*gen_kill_estimator\(\)*](#) and [*gen_new_estimator\(\)*](#).

Returns 0 on success or a negative error code.

```
bool gen_estimator_active(struct net_rate_estimator __rcu **rate_est)
    test if estimator is currently in use
```

Parameters

struct net_rate_estimator __rcu **rate_est
rate estimator

Description

Returns true if estimator is active, and false if not.

13.1.5 SUN RPC subsystem

```
_be32 *xdr_encode_opaque_fixed(_be32 *p, const void *ptr, unsigned int nbytes)
    Encode fixed length opaque data
```

Parameters

_be32 *p
pointer to current position in XDR buffer.
const void *ptr
pointer to data to encode (or NULL)
unsigned int nbytes
size of data.

Description

Copy the array of data of length nbytes at ptr to the XDR buffer at position p, then align to the next 32-bit boundary by padding with zero bytes (see RFC1832). Returns the updated current XDR buffer position

Note

if ptr is NULL, only the padding is performed.

```
_be32 *xdr_encode_opaque(_be32 *p, const void *ptr, unsigned int nbytes)
    Encode variable length opaque data
```

Parameters

_be32 *p
pointer to current position in XDR buffer.
const void *ptr
pointer to data to encode (or NULL)
unsigned int nbytes
size of data.

Description

Returns the updated current XDR buffer position

```
void xdr_terminate_string(const struct xdr_buf *buf, const u32 len)
    '0'-terminate a string residing in an xdr_buf
```

Parameters

const struct xdr_buf *buf

XDR buffer where string resides

const u32 len

length of string, in bytes

void xdr_inline_pages(struct xdr_buf *xdr, unsigned int offset, struct page **pages,

unsigned int base, unsigned int len)

Prepare receive buffer for a large reply

Parameters

struct xdr_buf *xdr

xdr_buf into which reply will be placed

unsigned int offset

expected offset where data payload will start, in bytes

struct page **pages

vector of struct page pointers

unsigned int base

offset in first page where receive should start, in bytes

unsigned int len

expected size of the upper layer data payload, in bytes

void _copy_from_pages(char *p, struct page **pages, size_t pgbase, size_t len)

Parameters

char *p

pointer to destination

struct page **pages

array of pages

size_t pgbase

offset of source data

size_t len

length

Description

Copies data into an arbitrary memory location from an array of pages. The copy is assumed to be non-overlapping.

unsigned int xdr_stream_pos(const struct xdr_stream *xdr)

Return the current offset from the start of the xdr_stream

Parameters

const struct xdr_stream *xdr

pointer to struct xdr_stream

unsigned int xdr_page_pos(const struct xdr_stream *xdr)

Return the current offset from the start of the xdr pages

Parameters

```
const struct xdr_stream *xdr
    pointer to struct xdr_stream

void xdr_init_encode(struct xdr_stream *xdr, struct xdr_buf *buf, __be32 *p, struct rpc_rqst *rqst)
```

Initialize a struct xdr_stream for sending data.

Parameters

```
struct xdr_stream *xdr
    pointer to xdr_stream struct

struct xdr_buf *buf
    pointer to XDR buffer in which to encode data

__be32 *p
    current pointer inside XDR buffer

struct rpc_rqst *rqst
    pointer to controlling rpc_rqst, for debugging
```

Note

at the moment the RPC client only passes the length of our

scratch buffer in the xdr_buf's header kvec. Previously this meant we needed to call xdr_adjust_iovec() after encoding the data. With the new scheme, the xdr_stream manages the details of the buffer length, and takes care of adjusting the kvec length for us.

```
void xdr_init_encode_pages(struct xdr_stream *xdr, struct xdr_buf *buf, struct page **pages, struct rpc_rqst *rqst)
```

Initialize an xdr_stream for encoding into pages

Parameters

```
struct xdr_stream *xdr
    pointer to xdr_stream struct

struct xdr_buf *buf
    pointer to XDR buffer into which to encode data

struct page **pages
    list of pages to decode into

struct rpc_rqst *rqst
    pointer to controlling rpc_rqst, for debugging

void __xdr_commit_encode(struct xdr_stream *xdr)
```

Ensure all data is written to buffer

Parameters

```
struct xdr_stream *xdr
    pointer to xdr_stream
```

Description

We handle encoding across page boundaries by giving the caller a temporary location to write to, then later copying the data into place; xdr_commit_encode does that copying.

Normally the caller doesn't need to call this directly, as the following `xdr_reserve_space` will do it. But an explicit call may be required at the end of encoding, or any other time when the `xdr_buf` data might be read.

`_be32 *xdr_reserve_space(struct xdr_stream *xdr, size_t nbytes)`

Reserve buffer space for sending

Parameters

`struct xdr_stream *xdr`

pointer to `xdr_stream`

`size_t nbytes`

number of bytes to reserve

Description

Checks that we have enough buffer space to encode 'nbytes' more bytes of data. If so, update the total `xdr_buf` length, and adjust the length of the current kvec.

`int xdr_reserve_space_vec(struct xdr_stream *xdr, size_t nbytes)`

Reserves a large amount of buffer space for sending

Parameters

`struct xdr_stream *xdr`

pointer to `xdr_stream`

`size_t nbytes`

number of bytes to reserve

Description

The size argument passed to `xdr_reserve_space()` is determined based on the number of bytes remaining in the current page to avoid invalidating `iov_base` pointers when `xdr_commit_encode()` is called.

Return values:

0: success -EMSGSIZE: not enough space is available in `xdr`

`void xdr_truncate_encode(struct xdr_stream *xdr, size_t len)`

truncate an encode buffer

Parameters

`struct xdr_stream *xdr`

pointer to `xdr_stream`

`size_t len`

new length of buffer

Description

Truncates the `xdr` stream, so that `xdr->buf->len == len`, and `xdr->p` points at offset `len` from the start of the buffer, and head, tail, and page lengths are adjusted to correspond.

If this means moving `xdr->p` to a different buffer, we assume that the end pointer should be set to the end of the current page, except in the case of the head buffer when we assume the head buffer's current length represents the end of the available buffer.

This is *not* safe to use on a buffer that already has inlined page cache pages (as in a zero-copy server read reply), except for the simple case of truncating from one position in the tail to another.

void `xdr_truncate_decode`(struct xdr_stream *xdr, size_t len)

Truncate a decoding stream

Parameters

struct xdr_stream *xdr

pointer to struct xdr_stream

size_t len

Number of bytes to remove

int `xdr_restrict buflen`(struct xdr_stream *xdr, int newbuflen)

decrease available buffer space

Parameters

struct xdr_stream *xdr

pointer to xdr_stream

int newbuflen

new maximum number of bytes available

Description

Adjust our idea of how much space is available in the buffer. If we've already used too much space in the buffer, returns -1. If the available space is already smaller than newbuflen, returns 0 and does nothing. Otherwise, adjusts xdr->buf->bulen to newbuflen and ensures xdr->end is set at most offset newbuflen from the start of the buffer.

void `xdr_write_pages`(struct xdr_stream *xdr, struct page **pages, unsigned int base, unsigned int len)

Insert a list of pages into an XDR buffer for sending

Parameters

struct xdr_stream *xdr

pointer to xdr_stream

struct page **pages

array of pages to insert

unsigned int base

starting offset of first data byte in **pages**

unsigned int len

number of data bytes in **pages** to insert

Description

After the **pages** are added, the tail iovec is instantiated pointing to end of the head buffer, and the stream is set up to encode subsequent items into the tail.

void `xdr_init_decode`(struct xdr_stream *xdr, struct xdr_buf *buf, __be32 *p, struct rpc_rqst *rqst)

Initialize an xdr_stream for decoding data.

Parameters

struct xdr_stream *xdr
pointer to xdr_stream struct

struct xdr_buf *buf
pointer to XDR buffer from which to decode data

_be32 *p
current pointer inside XDR buffer

struct rpc_rqst *rqst
pointer to controlling rpc_rqst, for debugging

void xdr_init_decode_pages(struct xdr_stream *xdr, struct xdr_buf *buf, struct page **pages, unsigned int len)

Initialize an xdr_stream for decoding into pages

Parameters

struct xdr_stream *xdr
pointer to xdr_stream struct

struct xdr_buf *buf
pointer to XDR buffer from which to decode data

struct page **pages
list of pages to decode into

unsigned int len
length in bytes of buffer in pages

void xdr_finish_decode(struct xdr_stream *xdr)
Clean up the xdr_stream after decoding data.

Parameters

struct xdr_stream *xdr
pointer to xdr_stream struct

_be32 *xdr_inline_decode(struct xdr_stream *xdr, size_t nbytes)
Retrieve XDR data to decode

Parameters

struct xdr_stream *xdr
pointer to xdr_stream struct

size_t nbytes
number of bytes of data to decode

Description

Check if the input buffer is long enough to enable us to decode 'nbytes' more bytes of data starting at the current position. If so return the current pointer, then update the current pointer position.

unsigned int xdr_read_pages(struct xdr_stream *xdr, unsigned int len)
align page-based XDR data to current pointer position

Parameters

struct xdr_stream *xdr
pointer to xdr_stream struct

unsigned int len
number of bytes of page data

Description

Moves data beyond the current pointer position from the XDR head[] buffer into the page list. Any data that lies beyond current position + **len** bytes is moved into the XDR tail[]. The xdr_stream current position is then advanced past that data to align to the next XDR object in the tail.

Returns the number of XDR encoded bytes now contained in the pages

void xdr_set_pagelen(struct xdr_stream *xdr, unsigned int len)
Sets the length of the XDR pages

Parameters

struct xdr_stream *xdr
pointer to xdr_stream struct

unsigned int len
new length of the XDR page data

Description

Either grows or shrinks the length of the xdr pages by setting pagelen to **len** bytes. When shrinking, any extra data is moved into buf->tail, whereas when growing any data beyond the current pointer is moved into the tail.

Returns True if the operation was successful, and False otherwise.

void xdr_enter_page(struct xdr_stream *xdr, unsigned int len)
decode data from the XDR page

Parameters

struct xdr_stream *xdr
pointer to xdr_stream struct

unsigned int len
number of bytes of page data

Description

Moves data beyond the current pointer position from the XDR head[] buffer into the page list. Any data that lies beyond current position + "len" bytes is moved into the XDR tail[]. The current pointer is then repositioned at the beginning of the first XDR page.

int xdr_buf_subsegment(const struct xdr_buf *buf, struct xdr_buf *subbuf, unsigned int base,
unsigned int len)
set subbuf to a portion of buf

Parameters

const struct xdr_buf *buf
an xdr buffer

struct xdr_buf *subbuf
the result buffer

unsigned int base
beginning of range in bytes

unsigned int len
length of range in bytes

Description

sets **subbuf** to an xdr buffer representing the portion of **buf** of length **len** starting at offset **base**.

buf and **subbuf** may be pointers to the same struct xdr_buf.

Returns -1 if base or length are out of bounds.

bool xdr_stream_subsegment(struct xdr_stream *xdr, struct xdr_buf *subbuf, unsigned int nbytes)

set **subbuf** to a portion of **xdr**

Parameters

struct xdr_stream *xdr
an xdr_stream set up for decoding

struct xdr_buf *subbuf
the result buffer

unsigned int nbytes
length of **xdr** to extract, in bytes

Description

Sets up **subbuf** to represent a portion of **xdr**. The portion starts at the current offset in **xdr**, and extends for a length of **nbytes**. If this is successful, **xdr** is advanced to the next XDR data item following that portion.

Return values:

true: **subbuf** has been initialized, and **xdr** has been advanced. false: a bounds error has occurred

unsigned int xdr_stream_move_subsegment(struct xdr_stream *xdr, unsigned int offset, unsigned int target, unsigned int length)

Move part of a stream to another position

Parameters

struct xdr_stream *xdr
the source xdr_stream

unsigned int offset
the source offset of the segment

unsigned int target
the target offset of the segment

unsigned int length
the number of bytes to move

Description

Moves **length** bytes from **offset** to **target** in the **xdr_stream**, overwriting anything in its space. Returns the number of bytes in the segment.

```
unsigned int xdr_stream_zero(struct xdr_stream *xdr, unsigned int offset, unsigned int length)
```

zero out a portion of an **xdr_stream**

Parameters

struct xdr_stream *xdr
an **xdr_stream** to zero out

unsigned int offset
the starting point in the stream

unsigned int length
the number of bytes to zero

```
void xdr_buf_trim(struct xdr_buf *buf, unsigned int len)
```

lop at most "len" bytes off the end of "buf"

Parameters

struct xdr_buf *buf
buf to be trimmed

unsigned int len
number of bytes to reduce "buf" by

Description

Trim an **xdr_buf** by the given number of bytes by fixing up the lengths. Note that it's possible that we'll trim less than that amount if the **xdr_buf** is too small, or if (for instance) it's all in the head and the parser has already read too far into it.

```
ssize_t xdr_stream_decode_opaque(struct xdr_stream *xdr, void *ptr, size_t size)
```

Decode variable length opaque

Parameters

struct xdr_stream *xdr
pointer to **xdr_stream**

void *ptr
location to store opaque data

size_t size
size of storage buffer **ptr**

Description

Return values:

On success, returns size of object stored in ***ptr** -EBADMSG on XDR buffer overflow
-EMSGSIZE on overflow of storage buffer **ptr**

```
ssize_t xdr_stream_decode_opaque_dup(struct xdr_stream *xdr, void **ptr, size_t maxlen,  
gfp_t gfp_flags)
```

Decode and duplicate variable length opaque

Parameters

struct xdr_stream *xdr
pointer to xdr_stream

void **ptr
location to store pointer to opaque data

size_t maxlen
maximum acceptable object size

gfp_t gfp_flags
GFP mask to use

Description

Return values:

On success, returns size of object stored in ***ptr** -EBADMSG on XDR buffer overflow
-EMSGSIZE if the size of the object would exceed **maxlen** -ENOMEM on memory allocation failure

ssize_t xdr_stream_decode_string(struct xdr_stream *xdr, char *str, size_t size)

Decode variable length string

Parameters

struct xdr_stream *xdr
pointer to xdr_stream

char *str
location to store string

size_t size
size of storage buffer **str**

Description

Return values:

On success, returns length of NUL-terminated string stored in ***str** -EBADMSG on XDR buffer overflow
-EMSGSIZE on overflow of storage buffer **str**

ssize_t xdr_stream_decode_string_dup(struct xdr_stream *xdr, char **str, size_t maxlen, gfp_t gfp_flags)

Decode and duplicate variable length string

Parameters

struct xdr_stream *xdr
pointer to xdr_stream

char **str
location to store pointer to string

size_t maxlen
maximum acceptable string length

gfp_t gfp_flags
GFP mask to use

Description

Return values:

On success, returns length of NUL-terminated string stored in *ptr - EBADMSG on XDR buffer overflow - EMSGSIZE if the size of the string would exceed maxlen - ENOMEM on memory allocation failure

`ssize_t xdr_stream_decode_opaque_auth(struct xdr_stream *xdr, u32 *flavor, void **body, unsigned int *body_len)`

Decode struct opaque_auth (RFC5531 S8.2)

Parameters

struct xdr_stream *xdr
pointer to xdr_stream

u32 *flavor
location to store decoded flavor

void **body
location to store decode body

unsigned int *body_len
location to store length of decoded body

Description**Return values:**

On success, returns the number of buffer bytes consumed - EBADMSG on XDR buffer overflow - EMSGSIZE if the decoded size of the body field exceeds 400 octets

`ssize_t xdr_stream_encode_opaque_auth(struct xdr_stream *xdr, u32 flavor, void *body, unsigned int body_len)`

Encode struct opaque_auth (RFC5531 S8.2)

Parameters

struct xdr_stream *xdr
pointer to xdr_stream

u32 flavor
verifier flavor to encode

void *body
content of body to encode

unsigned int body_len
length of body to encode

Description**Return values:**

On success, returns length in bytes of XDR buffer consumed - EBADMSG on XDR buffer overflow - EMSGSIZE if the size of **body** exceeds 400 octets

`int svc_reg_xprt_class(struct svc_xprt_class *xcl)`

Register a server-side RPC transport class

Parameters

struct svc_xprt_class *xcl
New transport class to be registered

Description

Returns zero on success; otherwise a negative errno is returned.

void svc_unreg_xprt_class(struct svc_xprt_class *xcl)

 Unregister a server-side RPC transport class

Parameters

struct svc_xprt_class *xcl

 Transport class to be unregistered

void svc_xprt_deferred_close(struct svc_xprt *xprt)

 Close a transport

Parameters

struct svc_xprt *xprt

 transport instance

Description

Used in contexts that need to defer the work of shutting down the transport to an nfsd thread.

void svc_xprt_received(struct svc_xprt *xprt)

 start next receiver thread

Parameters

struct svc_xprt *xprt

 controlling transport

Description

The caller must hold the XPT_BUSY bit and must not thereafter touch transport data.

Note

XPT_DATA only gets cleared when a read-attempt finds no (or insufficient) data.

int svc_xprt_create(struct svc_serv *serv, const char *xprt_name, struct net *net, const int family, const unsigned short port, int flags, const struct cred *cred)

 Add a new listener to **serv**

Parameters

struct svc_serv *serv

 target RPC service

const char *xprt_name

 transport class name

struct net *net

 network namespace

const int family

 network address family

const unsigned short port

 listener port

```
int flags
    SVC_SOCK flags

const struct cred *cred
    credential to bind to this transport
```

Description**Return values:**

0: New listener added successfully -EPROTONOSUPPORT: Requested transport type not supported

```
char *svc_print_addr(struct svc_rqst *rqstp, char *buf, size_t len)
    Format rq_addr field for printing
```

Parameters

```
struct svc_rqst *rqstp
    svc_rqst struct containing address to print
```

```
char *buf
    target buffer for formatted address
```

```
size_t len
    length of target buffer
```

```
void svc_xprt_enqueue(struct svc_xprt *xprt)
    Queue a transport on an idle nfsd thread
```

Parameters

```
struct svc_xprt *xprt
    transport with data pending
```

```
void svc_reserve(struct svc_rqst *rqstp, int space)
    change the space reserved for the reply to a request.
```

Parameters

```
struct svc_rqst *rqstp
    The request in question
```

```
int space
    new max space to reserve
```

Description

Each request reserves some space on the output queue of the transport to make sure the reply fits. This function reduces that reserved space to be the amount of space used already, plus **space**.

```
void svc_wake_up(struct svc_serv *serv)
    Wake up a service thread for non-transport work
```

Parameters

```
struct svc_serv *serv
    RPC service
```

Description

Some svc_serv's will have occasional work to do, even when a xprt is not waiting to be serviced. This function is there to "kick" a task in one of those services so that it can wake up and do that work. Note that we only bother with pool 0 as we don't need to wake up more than one thread for this purpose.

void **svc_recv**(struct svc_rqst *rqstp)

Receive and process the next request on any transport

Parameters

struct svc_rqst *rqstp

an idle RPC service thread

Description

This code is carefully organised not to touch any cachelines in the shared svc_serv structure, only cachelines in the local svc_pool.

void **svc_xprt_close**(struct svc_xprt *xprt)

Close a client connection

Parameters

struct svc_xprt *xprt

transport to disconnect

void **svc_xprt_destroy_all**(struct svc_serv *serv, struct net *net)

Destroy transports associated with **serv**

Parameters

struct svc_serv *serv

RPC service to be shut down

struct net *net

target network namespace

Description

Server threads may still be running (especially in the case where the service is still running in other network namespaces).

So we shut down sockets the same way we would on a running server, by setting XPT_CLOSE, enqueueing, and letting a thread pick it up to do the close. In the case there are no such other threads, threads running, svc_clean_up_xprts() does a simple version of a server's main event loop, and in the case where there are other threads, we may need to wait a little while and then check again to see if they're done.

struct svc_xprt ***svc_find_xprt**(struct svc_serv *serv, const char *xcl_name, struct net *net,
const sa_family_t af, const unsigned short port)

find an RPC transport instance

Parameters

struct svc_serv *serv

pointer to svc_serv to search

const char *xcl_name

C string containing transport's class name

```
struct net *net
    owner net pointer

const sa_family_t af
    Address family of transport's local address

const unsigned short port
    transport's IP port number
```

Description

Return the transport instance pointer for the endpoint accepting connections/peer traffic from the specified transport class, address family and port.

Specifying 0 for the address family or port is effectively a wild-card, and will result in matching the first transport in the service's list that has a matching class name.

```
int svc_xprt_names(struct svc_serv *serv, char *buf, const int buflen)
    format a buffer with a list of transport names
```

Parameters

```
struct svc_serv *serv
    pointer to an RPC service

char *buf
    pointer to a buffer to be filled in

const int buflen
    length of buffer to be filled in
```

Description

Fills in **buf** with a string containing a list of transport names, each name terminated with '\n'.

Returns positive length of the filled-in string on success; otherwise a negative errno value is returned if an error occurs.

```
int xprt_register_transport(struct xprt_class *transport)
    register a transport implementation
```

Parameters

```
struct xprt_class *transport
    transport to register
```

Description

If a transport implementation is loaded as a kernel module, it can call this interface to make itself known to the RPC client.

Return

0: transport successfully registered -EEXIST: transport already registered -EINVAL: transport module being unloaded

```
int xprt_unregister_transport(struct xprt_class *transport)
    unregister a transport implementation
```

Parameters

struct xpvt_class *transport
transport to unregister

Return

0: transport successfully unregistered -ENOENT: transport never registered

int xpvt_find_transport_ident(const char *netid)
convert a netid into a transport identifier

Parameters

const char *netid
transport to load

Return

> 0: transport identifier -ENOENT: transport module not available

int xpvt_reserve_xprt(struct rpc_xprt *xprt, struct rpc_task *task)
serialize write access to transports

Parameters

struct rpc_xprt *xprt
pointer to the target transport

struct rpc_task *task
task that is requesting access to the transport

Description

This prevents mixing the payload of separate requests, and prevents transport connects from colliding with writes. No congestion control is provided.

void xpvt_release_xprt(struct rpc_xprt *xprt, struct rpc_task *task)
allow other requests to use a transport

Parameters

struct rpc_xprt *xprt
transport with other tasks potentially waiting

struct rpc_task *task
task that is releasing access to the transport

Description

Note that "task" can be NULL. No congestion control is provided.

void xpvt_release_xprt_cong(struct rpc_xprt *xprt, struct rpc_task *task)
allow other requests to use a transport

Parameters

struct rpc_xprt *xprt
transport with other tasks potentially waiting

struct rpc_task *task
task that is releasing access to the transport

Description

Note that "task" can be NULL. Another task is awoken to use the transport if the transport's congestion window allows it.

bool xprt_request_get_cong(struct rpc_xprt *xprt, struct rpc_rqst *req)
Request congestion control credits

Parameters

struct rpc_xprt *xprt
pointer to transport

struct rpc_rqst *req
pointer to RPC request

Description

Useful for transports that require congestion control.

void xprt_release_rqst_cong(struct rpc_task *task)
housekeeping when request is complete

Parameters

struct rpc_task *task
RPC request that recently completed

Description

Useful for transports that require congestion control.

void xprt_adjust_cwnd(struct rpc_xprt *xprt, struct rpc_task *task, int result)
adjust transport congestion window

Parameters

struct rpc_xprt *xprt
pointer to xprt

struct rpc_task *task
recently completed RPC request used to adjust window

int result
result code of completed RPC request

Description

The transport code maintains an estimate on the maximum number of out-standing RPC requests, using a smoothed version of the congestion avoidance implemented in 44BSD. This is basically the Van Jacobson congestion algorithm: If a retransmit occurs, the congestion window is halved; otherwise, it is incremented by 1/cwnd when

- a reply is received and
- a full number of requests are outstanding and
- the congestion window hasn't been updated recently.

void xprt_wake_pending_tasks(struct rpc_xprt *xprt, int status)
wake all tasks on a transport's pending queue

Parameters

struct rpc_xprt *xprt
transport with waiting tasks

int status
result code to plant in each task before waking it

void xprt_wait_for_buffer_space(struct rpc_xprt *xprt)
wait for transport output buffer to clear

Parameters

struct rpc_xprt *xprt
transport

Description

Note that we only set the timer for the case of RPC_IS_SOFT(), since we don't in general want to force a socket disconnection due to an incomplete RPC call transmission.

bool xprt_write_space(struct rpc_xprt *xprt)
wake the task waiting for transport output buffer space

Parameters

struct rpc_xprt *xprt
transport with waiting tasks

Description

Can be called in a soft IRQ context, so xprt_write_space never sleeps.

void xprt_disconnect_done(struct rpc_xprt *xprt)
mark a transport as disconnected

Parameters

struct rpc_xprt *xprt
transport to flag for disconnect

void xprt_force_disconnect(struct rpc_xprt *xprt)
force a transport to disconnect

Parameters

struct rpc_xprt *xprt
transport to disconnect

unsigned long xprt_reconnect_delay(const struct rpc_xprt *xprt)
compute the wait before scheduling a connect

Parameters

const struct rpc_xprt *xprt
transport instance

void xprt_reconnect_backoff(struct rpc_xprt *xprt, unsigned long init_to)
compute the new re-establish timeout

Parameters

```
struct rpc_xprt *xprt
    transport instance

unsigned long init_to
    initial reestablish timeout

struct rpc_rqst *xprt_lookup_rqst(struct rpc_xprt *xprt, __be32 xid)
    find an RPC request corresponding to an XID
```

Parameters

```
struct rpc_xprt *xprt
    transport on which the original request was transmitted

__be32 xid
    RPC XID of incoming reply
```

Description

Caller holds xprt->queue_lock.

```
void xprt_pin_rqst(struct rpc_rqst *req)
    Pin a request on the transport receive list
```

Parameters

```
struct rpc_rqst *req
    Request to pin
```

Description

Caller must ensure this is atomic with the call to [*xprt_lookup_rqst\(\)*](#) so should be holding xprt->queue_lock.

```
void xprt_unpin_rqst(struct rpc_rqst *req)
    Unpin a request on the transport receive list
```

Parameters

```
struct rpc_rqst *req
    Request to pin
```

Description

Caller should be holding xprt->queue_lock.

```
void xprt_update_rtt(struct rpc_task *task)
    Update RPC RTT statistics
```

Parameters

```
struct rpc_task *task
    RPC request that recently completed
```

Description

Caller holds xprt->queue_lock.

```
void xprt_complete_rqst(struct rpc_task *task, int copied)
    called when reply processing is complete
```

Parameters

struct rpc_task *task
RPC request that recently completed

int copied
actual number of bytes received from the transport

Description

Caller holds xprt->queue_lock.

void xprt_wait_for_reply_request_def(struct rpc_task *task)
wait for reply

Parameters

struct rpc_task *task
pointer to rpc_task

Description

Set a request's retransmit timeout based on the transport's default timeout parameters. Used by transports that don't adjust the retransmit timeout based on round-trip time estimation, and put the task to sleep on the pending queue.

void xprt_wait_for_reply_request_rtt(struct rpc_task *task)
wait for reply using RTT estimator

Parameters

struct rpc_task *task
pointer to rpc_task

Description

Set a request's retransmit timeout using the RTT estimator, and put the task to sleep on the pending queue.

struct rpc_xprt *xprt_get(struct rpc_xprt *xprt)
return a reference to an RPC transport.

Parameters

struct rpc_xprt *xprt
pointer to the transport

void xprt_put(struct rpc_xprt *xprt)
release a reference to an RPC transport.

Parameters

struct rpc_xprt *xprt
pointer to the transport

void rpc_wake_up(struct rpc_wait_queue *queue)
wake up all rpc_tasks

Parameters

struct rpc_wait_queue *queue
rpc_wait_queue on which the tasks are sleeping

Description

Grabs queue->lock

```
void rpc_wake_up_status(struct rpc_wait_queue *queue, int status)
    wake up all rpc_tasks and set their status value.
```

Parameters

struct rpc_wait_queue *queue

rpc_wait_queue on which the tasks are sleeping

int status

status value to set

Description

Grabs queue->lock

```
int rpc_malloc(struct rpc_task *task)
    allocate RPC buffer resources
```

Parameters

struct rpc_task *task

RPC task

Description

A single memory region is allocated, which is split between the RPC call and RPC reply that this task is being used for. When this RPC is retired, the memory is released by calling `rpc_free`.

To prevent `rpciod` from hanging, this allocator never sleeps, returning `-ENOMEM` and suppressing warning if the request cannot be serviced immediately. The caller can arrange to sleep in a way that is safe for `rpciod`.

Most requests are 'small' (under 2KiB) and can be serviced from a mempool, ensuring that NFS reads and writes can always proceed, and that there is good locality of reference for these buffers.

`void rpc_free(struct rpc_task *task)`

free RPC buffer resources allocated via `rpc_malloc`

Parameters

struct rpc_task *task

RPC task

```
int csum_partial_copy_to_xdr(struct xdr_buf *xdr, struct sk_buff *skb)
    checksum and copy data
```

Parameters

struct xdr_buf *xdr

target XDR buffer

struct sk_buff *skb

source skb

Description

We have set things up such that we perform the checksum of the UDP packet in parallel with the copies into the RPC client iovec. -DaveM

```
struct rpc_iostats *rpc_alloc_iostats(struct rpc_clnt *clnt)
    allocate an rpc_iostats structure
```

Parameters

```
struct rpc_clnt *clnt
    RPC program, version, and xprt
```

```
void rpc_free_iostats(struct rpc_iostats *stats)
    release an rpc_iostats structure
```

Parameters

```
struct rpc_iostats *stats
    doomed rpc_iostats structure
```

```
void rpc_count_iostats_metrics(const struct rpc_task *task, struct rpc_iostats *op_metrics)
    tally up per-task stats
```

Parameters

```
const struct rpc_task *task
    completed rpc_task
```

```
struct rpc_iostats *op_metrics
    stat structure for OP that will accumulate stats from task
```

```
void rpc_count_iostats(const struct rpc_task *task, struct rpc_iostats *stats)
    tally up per-task stats
```

Parameters

```
const struct rpc_task *task
    completed rpc_task
```

```
struct rpc_iostats *stats
    array of stat structures
```

Description

Uses the statidx from **task**

```
int rpc_queue_upcall(struct rpc_pipe *pipe, struct rpc_pipe_msg *msg)
    queue an upcall message to userspace
```

Parameters

```
struct rpc_pipe *pipe
    upcall pipe on which to queue given message
```

```
struct rpc_pipe_msg *msg
    message to queue
```

Description

Call with an **inode** created by `rpc_mkpipe()` to queue an upcall. A userspace process may then later read the upcall by performing a read on an open file for this inode. It is up to the caller to initialize the fields of **msg** (other than **msg->list**) appropriately.

```
struct dentry *rpc_mkpipe_dentry(struct dentry *parent, const char *name, void *private,
                                 struct rpc_pipe *pipe)
```

make an rpc_pipefs file for kernel<->userspace communication

Parameters

struct dentry *parent

dentry of directory to create new "pipe" in

const char *name

name of pipe

void *private

private data to associate with the pipe, for the caller's use

struct rpc_pipe *pipe

rpc_pipe containing input parameters

Description

Data is made available for userspace to read by calls to [rpc_queue_upcall\(\)](#). The actual reads will result in calls to **ops->upcall**, which will be called with the file pointer, message, and userspace buffer to copy to.

Writes can come at any time, and do not necessarily have to be responses to upcalls. They will result in calls to **msg->downcall**.

The **private** argument passed here will be available to all these methods from the file pointer, via RPC_I(file_inode(file))->private.

```
int rpc_unlink(struct dentry *dentry)
```

remove a pipe

Parameters

struct dentry *dentry

dentry for the pipe, as returned from rpc_mkpipe

Description

After this call, lookups will no longer find the pipe, and any attempts to read or write using preexisting opens of the pipe will return -EPIPE.

```
void rpc_init_pipe_dir_head(struct rpc_pipe_dir_head *pdh)
```

initialise a struct rpc_pipe_dir_head

Parameters

struct rpc_pipe_dir_head *pdh

pointer to struct rpc_pipe_dir_head

```
void rpc_init_pipe_dir_object(struct rpc_pipe_dir_object *pdo, const struct
                               rpc_pipe_dir_object_ops *pdo_ops, void *pdo_data)
```

initialise a struct rpc_pipe_dir_object

Parameters

struct rpc_pipe_dir_object *pdo

pointer to struct rpc_pipe_dir_object

```
const struct rpc_pipe_dir_object_ops *pdo_ops
pointer to const struct rpc_pipe_dir_object_ops

void *pdo_data
pointer to caller-defined data

int rpc_add_pipe_dir_object(struct net *net, struct rpc_pipe_dir_head *pdh, struct
                           rpc_pipe_dir_object *pdo)
associate a rpc_pipe_dir_object to a directory
```

Parameters

```
struct net *net
pointer to struct net

struct rpc_pipe_dir_head *pdh
pointer to struct rpc_pipe_dir_head

struct rpc_pipe_dir_object *pdo
pointer to struct rpc_pipe_dir_object

void rpc_remove_pipe_dir_object(struct net *net, struct rpc_pipe_dir_head *pdh, struct
                               rpc_pipe_dir_object *pdo)
remove a rpc_pipe_dir_object from a directory
```

Parameters

```
struct net *net
pointer to struct net

struct rpc_pipe_dir_head *pdh
pointer to struct rpc_pipe_dir_head

struct rpc_pipe_dir_object *pdo
pointer to struct rpc_pipe_dir_object

struct rpc_pipe_dir_object *rpc_find_or_alloc_pipe_dir_object(struct net *net, struct
                                                               rpc_pipe_dir_head *pdh,
                                                               int (*match)(struct
                                                               rpc_pipe_dir_object*, void*),
                                                               struct
                                                               rpc_pipe_dir_object
                                                               *(*alloc)(void*), void
                                                               *data)
```

Parameters

```
struct net *net
pointer to struct net

struct rpc_pipe_dir_head *pdh
pointer to struct rpc_pipe_dir_head

int (*match)(struct rpc_pipe_dir_object *, void *)
match struct rpc_pipe_dir_object to data

struct rpc_pipe_dir_object *(*alloc)(void *)
allocate a new struct rpc_pipe_dir_object
```

```
void *data
    user defined data for match() and alloc()

void rpcb_getport_async(struct rpc_task *task)
    obtain the port for a given RPC service on a given host
```

Parameters

```
struct rpc_task *task
    task that is waiting for portmapper request
```

Description

This one can be called for an ongoing RPC request, and can be used in an async (rpciod) context.

```
struct rpc_clnt *rpc_create(struct rpc_create_args *args)
    create an RPC client and transport with one call
```

Parameters

```
struct rpc_create_args *args
    rpc_clnt create argument structure
```

Description

Creates and initializes an RPC transport and an RPC client.

It can ping the server in order to determine if it is up, and to see if it supports this program and version. RPC_CLNT_CREATE_NOPING disables this behavior so asynchronous tasks can also use rpc_create.

```
struct rpc_clnt *rpc_clone_client(struct rpc_clnt *clnt)
    Clone an RPC client structure
```

Parameters

```
struct rpc_clnt *clnt
    RPC client whose parameters are copied
```

Description

Returns a fresh RPC client or an ERR_PTR.

```
struct rpc_clnt *rpc_clone_client_set_auth(struct rpc_clnt *clnt, rpc_authflavor_t flavor)
    Clone an RPC client structure and set its auth
```

Parameters

```
struct rpc_clnt *clnt
    RPC client whose parameters are copied
```

```
rpc_authflavor_t flavor
    security flavor for new client
```

Description

Returns a fresh RPC client or an ERR_PTR.

```
int rpc_switch_client_transport(struct rpc_clnt *clnt, struct xprt_create *args, const
                                struct rpc_timeout *timeout)
    switch the RPC transport on the fly
```

Parameters

struct rpc_clnt *clnt
pointer to a struct rpc_clnt

struct xprt_create *args
pointer to the new transport arguments

const struct rpc_timeout *timeout
pointer to the new timeout parameters

Description

This function allows the caller to switch the RPC transport for the rpc_clnt structure 'clnt' to allow it to connect to a mirrored NFS server, for instance. It assumes that the caller has ensured that there are no active RPC tasks by using some form of locking.

Returns zero if "clnt" is now using the new xprt. Otherwise a negative errno is returned, and "clnt" continues to use the old xprt.

```
int rpc_clnt_iterate_for_each_xprt(struct rpc_clnt *clnt, int (*fn)(struct rpc_clnt*, struct rpc_xprt*, void*), void *data)
```

Apply a function to all transports

Parameters

struct rpc_clnt *clnt
pointer to client

int (*fn)(struct rpc_clnt *, struct rpc_xprt *, void *)
function to apply

void *data
void pointer to function data

Description

Iterates through the list of RPC transports currently attached to the client and applies the function fn(clnt, xprt, data).

On error, the iteration stops, and the function returns the error value.

```
unsigned long rpc_cancel_tasks(struct rpc_clnt *clnt, int error, bool (*fnmatch)(const struct rpc_task*, const void*), const void *data)
```

try to cancel a set of RPC tasks

Parameters

struct rpc_clnt *clnt
Pointer to RPC client

int error
RPC task error value to set

bool (*fnmatch)(const struct rpc_task *, const void *)
Pointer to selector function

const void *data
User data

Description

Uses **fnmatch** to define a set of RPC tasks that are to be cancelled. The argument **error** must be a negative error value.

```
struct rpc_clnt *rpc_bind_new_program(struct rpc_clnt *old, const struct rpc_program
                                      *program, u32 vers)
```

bind a new RPC program to an existing client

Parameters

struct rpc_clnt *old
old rpc_client

const struct rpc_program *program
rpc program to set

u32 vers
rpc program version

Description

Clones the rpc client and sets up a new RPC program. This is mainly of use for enabling different RPC programs to share the same transport. The Sun NFSv2/v3 ACL protocol can do this.

```
struct rpc_task *rpc_run_task(const struct rpc_task_setup *task_setup_data)
```

Allocate a new RPC task, then run rpc_execute against it

Parameters

const struct rpc_task_setup *task_setup_data
pointer to task initialisation data

```
int rpc_call_sync(struct rpc_clnt *clnt, const struct rpc_message *msg, int flags)
```

Perform a synchronous RPC call

Parameters

struct rpc_clnt *clnt
pointer to RPC client

const struct rpc_message *msg
RPC call parameters

int flags
RPC call flags

```
int rpc_call_async(struct rpc_clnt *clnt, const struct rpc_message *msg, int flags, const
                   struct rpc_call_ops *tk_ops, void *data)
```

Perform an asynchronous RPC call

Parameters

struct rpc_clnt *clnt
pointer to RPC client

const struct rpc_message *msg
RPC call parameters

int flags
RPC call flags

```
const struct rpc_call_ops *tk_ops
    RPC call ops

void *data
    user call data

void rpc_prepare_reply_pages(struct rpc_rqst *req, struct page **pages, unsigned int base,
                             unsigned int len, unsigned int hdrsize)
    Prepare to receive a reply data payload into pages
```

Parameters

struct rpc_rqst *req
RPC request to prepare

struct page **pages
vector of struct page pointers

unsigned int base
offset in first page where receive should start, in bytes

unsigned int len
expected size of the upper layer data payload, in bytes

unsigned int hdrsize
expected size of upper layer reply header, in XDR words

```
size_t rpc_peeraddr(struct rpc_clnt *clnt, struct sockaddr *buf, size_t bufsize)
    extract remote peer address from clnt's xprt
```

Parameters

struct rpc_clnt *clnt
RPC client structure

struct sockaddr *buf
target buffer

size_t bufsize
length of target buffer

Description

Returns the number of bytes that are actually in the stored address.

```
const char *rpc_peeraddr2str(struct rpc_clnt *clnt, enum rpc_display_format_t format)
    return remote peer address in printable format
```

Parameters

struct rpc_clnt *clnt
RPC client structure

enum rpc_display_format_t format
address format

Description

NB: the lifetime of the memory referenced by the returned pointer is the same as the rpc_xprt itself. As long as the caller uses this pointer, it must hold the RCU read lock.

```
int rpc_localaddr(struct rpc_clnt *clnt, struct sockaddr *buf, size_t buflen)
    discover local endpoint address for an RPC client
```

Parameters

struct rpc_clnt *clnt
RPC client structure

struct sockaddr *buf
target buffer

size_t buflen
size of target buffer, in bytes

Description

Returns zero and fills in "buf" and "buflen" if successful; otherwise, a negative errno is returned.

This works even if the underlying transport is not currently connected, or if the upper layer never previously provided a source address.

The result of this function call is transient: multiple calls in succession may give different results, depending on how local networking configuration changes over time.

struct net *rpc_net_ns(struct rpc_clnt *clnt)
Get the network namespace for this RPC client

Parameters

struct rpc_clnt *clnt
RPC client to query

size_t rpc_max_payload(struct rpc_clnt *clnt)
Get maximum payload size for a transport, in bytes

Parameters

struct rpc_clnt *clnt
RPC client to query

Description

For stream transports, this is one RPC record fragment (see RFC 1831), as we don't support multi-record requests yet. For datagram transports, this is the size of an IP packet minus the IP, UDP, and RPC header sizes.

size_t rpc_max_bc_payload(struct rpc_clnt *clnt)
Get maximum backchannel payload size, in bytes

Parameters

struct rpc_clnt *clnt
RPC client to query

void rpc_force_rebind(struct rpc_clnt *clnt)
force transport to check that remote port is unchanged

Parameters

struct rpc_clnt *clnt
client to rebind

```
int rpc_clnt_test_and_add_xprt(struct rpc_clnt *clnt, struct rpc_xprt_switch *xps, struct
                                rpc_xprt *xprt, void *in_max_connect)
```

Test and add a new transport to a rpc_clnt

Parameters

struct rpc_clnt *clnt

pointer to struct rpc_clnt

struct rpc_xprt_switch *xps

pointer to struct rpc_xprt_switch,

struct rpc_xprt *xprt

pointer struct rpc_xprt

void *in_max_connect

pointer to the max_connect value for the passed in xprt transport

```
int rpc_clnt_setup_test_and_add_xprt(struct rpc_clnt *clnt, struct rpc_xprt_switch *xps,
                                      struct rpc_xprt *xprt, void *data)
```

Parameters

struct rpc_clnt *clnt

struct rpc_clnt to get the new transport

struct rpc_xprt_switch *xps

the rpc_xprt_switch to hold the new transport

struct rpc_xprt *xprt

the rpc_xprt to test

void *data

a struct rpc_add_xprt_test pointer that holds the test function and test function call data

Description

This is an rpc_clnt_add_xprt setup() function which returns 1 so:

1) caller of the test function must dereference the rpc_xprt_switch and the rpc_xprt. 2) test function must call rpc_xprt_switch_add_xprt, usually in the rpc_call_done routine.

Upon success (return of 1), the test function adds the new transport to the rpc_clnt xprt switch

```
int rpc_clnt_add_xprt(struct rpc_clnt *clnt, struct xprt_create *xprtargs, int (*setup)(struct
                                         rpc_clnt*, struct rpc_xprt_switch*, struct rpc_xprt*, void*),
                                         void*)
```

Add a new transport to a rpc_clnt

Parameters

struct rpc_clnt *clnt

pointer to struct rpc_clnt

struct xprt_create *xprtargs

pointer to struct xprt_create

```
int (*setup)(struct rpc_clnt *, struct rpc_xprt_switch *, struct rpc_xprt *,
            void *)
```

callback to test and/or set up the connection

```
void *data
pointer to setup function data
```

Description

Creates a new transport using the parameters set in args and adds it to clnt. If ping is set, then test that connectivity succeeds before adding the new transport.

13.2 Network device support

13.2.1 Driver Support

```
void dev_add_pack(struct packet_type *pt)
add packet handler
```

Parameters

```
struct packet_type *pt
packet type declaration
```

Add a protocol handler to the networking stack. The passed `packet_type` is linked into kernel lists and may not be freed until it has been removed from the kernel lists.

This call does not sleep therefore it can not guarantee all CPU's that are in middle of receiving packets will see the new packet type (until the next received packet).

```
void __dev_remove_pack(struct packet_type *pt)
remove packet handler
```

Parameters

```
struct packet_type *pt
packet type declaration
```

Remove a protocol handler that was previously added to the kernel protocol handlers by `dev_add_pack()`. The passed `packet_type` is removed from the kernel lists and can be freed or reused once this function returns.

The packet type might still be in use by receivers and must not be freed until after all the CPU's have gone through a quiescent state.

```
void dev_remove_pack(struct packet_type *pt)
remove packet handler
```

Parameters

```
struct packet_type *pt
packet type declaration
```

Remove a protocol handler that was previously added to the kernel protocol handlers by `dev_add_pack()`. The passed `packet_type` is removed from the kernel lists and can be freed or reused once this function returns.

This call sleeps to guarantee that no CPU is looking at the packet type after return.

int **dev_get_iflink**(const struct *net_device* *dev)
 get 'iflink' value of a interface

Parameters

const struct net_device *dev
 targeted interface

Indicates the ifindex the interface is linked to. Physical interfaces have the same 'ifindex' and 'iflink' values.

int **dev_fill_metadata_dst**(struct *net_device* *dev, struct *sk_buff* *skb)
 Retrieve tunnel egress information.

Parameters

struct net_device *dev
 targeted interface

struct sk_buff *skb
 The packet.

For better visibility of tunnel traffic OVS needs to retrieve egress tunnel information for a packet. Following API allows user to get this info.

struct *net_device* ***__dev_get_by_name**(struct *net* *net, const char *name)
 find a device by its name

Parameters

struct net *net
 the applicable net namespace

const char *name
 name to find

Find an interface by name. Must be called under RTNL semaphore or **dev_base_lock**. If the name is found a pointer to the device is returned. If the name is not found then NULL is returned. The reference counters are not incremented so the caller must be careful with locks.

struct *net_device* ***dev_get_by_name_rcu**(struct *net* *net, const char *name)
 find a device by its name

Parameters

struct net *net
 the applicable net namespace

const char *name
 name to find

Description

Find an interface by name. If the name is found a pointer to the device is returned. If the name is not found then NULL is returned. The reference counters are not incremented so the caller must be careful with locks. The caller must hold RCU lock.

struct *net_device* ***netdev_get_by_name**(struct *net* *net, const char *name, netdevice_tracker *tracker, gfp_t gfp)

find a device by its name

Parameters

struct net *net

the applicable net namespace

const char *name

name to find

netdevice_tracker *tracker

tracking object for the acquired reference

gfp_t gfp

allocation flags for the tracker

Find an interface by name. This can be called from any context and does its own locking. The returned handle has the usage count incremented and the caller must use `netdev_put()` to release it when it is no longer needed. `NULL` is returned if no matching device is found.

struct `net_device *__dev_get_by_index`(`struct net *net`, `int ifindex`)

find a device by its ifindex

Parameters

struct net *net

the applicable net namespace

int ifindex

index of device

Search for an interface by index. Returns `NULL` if the device is not found or a pointer to the device. The device has not had its reference counter increased so the caller must be careful about locking. The caller must hold either the RTNL semaphore or **dev_base_lock**.

struct `net_device *dev_get_by_index_rcu`(`struct net *net`, `int ifindex`)

find a device by its ifindex

Parameters

struct net *net

the applicable net namespace

int ifindex

index of device

Search for an interface by index. Returns `NULL` if the device is not found or a pointer to the device. The device has not had its reference counter increased so the caller must be careful about locking. The caller must hold RCU lock.

struct `net_device *netdev_get_by_index`(`struct net *net`, `int ifindex`, `netdevice_tracker *tracker`, `gfp_t gfp`)

find a device by its ifindex

Parameters

struct net *net

the applicable net namespace

int ifindex

index of device

netdevice_tracker *tracker

tracking object for the acquired reference

gfp_t gfp

allocation flags for the tracker

Search for an interface by index. Returns NULL if the device is not found or a pointer to the device. The device returned has had a reference added and the pointer is safe until the user calls netdev_put() to indicate they have finished with it.

struct *net_device* ***dev_get_by_napi_id**(unsigned int napi_id)

find a device by napi_id

Parameters**unsigned int napi_id**

ID of the NAPI struct

Search for an interface by NAPI ID. Returns NULL if the device is not found or a pointer to the device. The device has not had its reference counter increased so the caller must be careful about locking. The caller must hold RCU lock.

struct *net_device* ***dev_getbyhwaddr_rcu**(struct *net* *net, unsigned short type, const char *ha)

find a device by its hardware address

Parameters**struct net *net**

the applicable net namespace

unsigned short type

media type of device

const char *ha

hardware address

Search for an interface by MAC address. Returns NULL if the device is not found or a pointer to the device. The caller must hold RCU or RTNL. The returned device has not had its ref count increased and the caller must therefore be careful about locking

struct *net_device* ***__dev_get_by_flags**(struct *net* *net, unsigned short if_flags, unsigned short mask)

find any device with given flags

Parameters**struct net *net**

the applicable net namespace

unsigned short if_flags

IFF_* values

unsigned short mask

bitmask of bits in if_flags to check

Search for any interface with the given flags. Returns NULL if a device is not found or a pointer to the device. Must be called inside rtnl_lock(), and result refcount is unchanged.

bool dev_valid_name(const char *name)
check if name is okay for network device

Parameters

const char *name
name string

Network device names need to be valid file names to allow sysfs to work. We also disallow any kind of whitespace.

int dev_alloc_name(struct net_device *dev, const char *name)
allocate a name for a device

Parameters

struct net_device *dev
device

const char *name
name format string

Passed a format string - eg "lt`d`" it will try and find a suitable id. It scans list of devices to build up a free map, then chooses the first empty slot. The caller must hold the dev_base or rtnl lock while allocating the name and adding the device in order to avoid duplicates. Limited to bits_per_byte * page size devices (ie 32K on most platforms). Returns the number of the unit assigned or a negative errno code.

int dev_set_alias(struct net_device *dev, const char *alias, size_t len)
change ifalias of a device

Parameters

struct net_device *dev
device

const char *alias
name up to IFALIASZ

size_t len
limit of bytes to copy from info

Set ifalias for a device,

void netdev_features_change(struct net_device *dev)
device changes features

Parameters

struct net_device *dev
device to cause notification

Called to indicate a device has changed features.

void netdev_state_change(struct net_device *dev)
device changes state

Parameters

struct net_device *dev
device to cause notification

Called to indicate a device has changed state. This function calls the notifier chains for netdev_chain and sends a NEWLINK message to the routing socket.

void __netdev_notify_peers(struct net_device *dev)

notify network peers about existence of **dev**, to be called when rtnl lock is already held.

Parameters

struct net_device *dev
network device

Description

Generate traffic such that interested network peers are aware of **dev**, such as by generating a gratuitous ARP. This may be used when a device wants to inform the rest of the network about some sort of reconfiguration such as a failover event or virtual machine migration.

void netdev_notify_peers(struct net_device *dev)

notify network peers about existence of **dev**

Parameters

struct net_device *dev
network device

Description

Generate traffic such that interested network peers are aware of **dev**, such as by generating a gratuitous ARP. This may be used when a device wants to inform the rest of the network about some sort of reconfiguration such as a failover event or virtual machine migration.

int dev_open(struct net_device *dev, struct netlink_ext_ack *extack)

prepare an interface for use.

Parameters

struct net_device *dev
device to open
struct netlink_ext_ack *extack
netlink extended ack

Takes a device from down to up state. The device's private open function is invoked and then the multicast lists are loaded. Finally the device is moved into the up state and a NETDEV_UP message is sent to the netdev notifier chain.

Calling this function on an active interface is a nop. On a failure a negative errno code is returned.

void dev_close(struct net_device *dev)

shutdown an interface.

Parameters

struct net_device *dev
device to shutdown

This function moves an active device into down state. A NETDEV_GOING_DOWN is sent to the netdev notifier chain. The device is then deactivated and finally a NETDEV_DOWN is sent to the notifier chain.

```
void dev_disable_lro(struct net_device *dev)
    disable Large Receive Offload on a device
```

Parameters

```
struct net_device *dev
    device
```

Disable Large Receive Offload (LRO) on a net device. Must be called under RTNL. This is needed if received packets may be forwarded to another interface.

```
int register_netdevice_notifier(struct notifier_block *nb)
    register a network notifier block
```

Parameters

```
struct notifier_block *nb
    notifier
```

Description

Register a notifier to be called when network device events occur. The notifier passed is linked into the kernel structures and must not be reused until it has been unregistered. A negative errno code is returned on a failure.

When registered all registration and up events are replayed to the new notifier to allow device to have a race free view of the network device list.

```
int unregister_netdevice_notifier(struct notifier_block *nb)
    unregister a network notifier block
```

Parameters

```
struct notifier_block *nb
    notifier
```

Description

Unregister a notifier previously registered by [*register_netdevice_notifier\(\)*](#). The notifier is unlinked into the kernel structures and may then be reused. A negative errno code is returned on a failure.

After unregistering unregister and down device events are synthesized for all devices on the device list to the removed notifier to remove the need for special case cleanup code.

```
int register_netdevice_notifier_net(struct net *net, struct notifier_block *nb)
    register a per-netns network notifier block
```

Parameters

```
struct net *net
    network namespace
```

```
struct notifier_block *nb
    notifier
```

Description

Register a notifier to be called when network device events occur. The notifier passed is linked into the kernel structures and must not be reused until it has been unregistered. A negative errno code is returned on a failure.

When registered all registration and up events are replayed to the new notifier to allow device to have a race free view of the network device list.

```
int unregister_netdevice_notifier_net(struct net *net, struct notifier_block *nb)  
    unregister a per-netns network notifier block
```

Parameters

struct net *net
network namespace
struct notifier_block *nb
notifier

Description

Unregister a notifier previously registered by *register_netdevice_notifier_net()*. The notifier is unlinked from the kernel structures and may then be reused. A negative errno code is returned on a failure.

After unregistering unregister and down device events are synthesized for all devices on the device list to the removed notifier to remove the need for special case cleanup code.

```
int call_netdevice_notifiers(unsigned long val, struct net_device *dev)  
    call all network notifier blocks
```

Parameters

unsigned long val
value passed unmodified to notifier function

struct net_device *dev
net_device pointer passed unmodified to notifier function
Call all network notifier blocks. Parameters and return value are as for raw_notifier_call_chain().

```
int dev_forward_skb(struct net_device *dev, struct sk_buff *skb)  
    loopback an skb to another netif
```

Parameters

struct net_device *dev
destination network device
struct sk_buff *skb
buffer to forward

Description

return values:

NET_RX_SUCCESS (no congestion) NET_RX_DROP (packet was dropped, but freed)
dev_forward_skb can be used for injecting an skb from the start_xmit function of one device into the receive queue of another device.

The receiving device may be in another namespace, so we have to clear all information in the skb that could impact namespace isolation.

```
bool dev_nit_active(struct net_device *dev)
    return true if any network interface taps are in use
```

Parameters

struct net_device *dev
network device to check for the presence of taps

```
int netif_set_real_num_rx_queues(struct net_device *dev, unsigned int rxq)
    set actual number of RX queues used
```

Parameters

struct net_device *dev
Network device

unsigned int rxq
Actual number of RX queues

This must be called either with the rtnl_lock held or before registration of the net device. Returns 0 on success, or a negative error code. If called before registration, it always succeeds.

```
int netif_set_real_num_queues(struct net_device *dev, unsigned int txq, unsigned int rxq)
    set actual number of RX and TX queues used
```

Parameters

struct net_device *dev
Network device

unsigned int txq
Actual number of TX queues

unsigned int rxq
Actual number of RX queues

Set the real number of both TX and RX queues. Does nothing if the number of queues is already correct.

```
void netif_set_tso_max_size(struct net_device *dev, unsigned int size)
    set the max size of TSO frames supported
```

Parameters

struct net_device *dev
netdev to update

unsigned int size
max skb->len of a TSO frame

Description

Set the limit on the size of TSO super-frames the device can handle. Unless explicitly set the stack will assume the value of GSO_LEGACY_MAX_SIZE.

void **netif_set_tso_max_segs**(struct *net_device* *dev, unsigned int segs)
 set the max number of segs supported for TSO

Parameters

struct net_device *dev
 netdev to update
unsigned int segs
 max number of TCP segments

Description

Set the limit on the number of TCP segments the device can generate from a single TSO super-frame. Unless explicitly set the stack will assume the value of GSO_MAX_SEGS.

void **netif_inherit_tso_max**(struct *net_device* *to, const struct *net_device* *from)
 copy all TSO limits from a lower device to an upper

Parameters

struct net_device *to
 netdev to update
const struct net_device *from
 netdev from which to copy the limits
int **netif_get_num_default_rss_queues**(void)
 default number of RSS queues

Parameters

void
 no arguments

Description

Default value is the number of physical cores if there are only 1 or 2, or divided by 2 if there are more.

void **netif_device_detach**(struct *net_device* *dev)
 mark device as removed

Parameters

struct net_device *dev
 network device

Description

Mark device as removed from system and therefore no longer available.

void **netif_device_attach**(struct *net_device* *dev)
 mark device as attached

Parameters

struct net_device *dev
 network device

Description

Mark device as attached from system and restart if needed.

```
int dev_loopback_xmit(struct net *net, struct sock *sk, struct sk_buff *skb)
    loop back skb
```

Parameters

struct net *net

network namespace this loopback is happening in

struct sock *sk

sk needed to be a netfilter okfn

struct sk_buff *skb

buffer to transmit

```
int _dev_queue_xmit(struct sk_buff *skb, struct net_device *sb_dev)
```

transmit a buffer

Parameters

struct sk_buff *skb

buffer to transmit

struct net_device *sb_dev

subordinate device used for L2 forwarding offload

Description

Queue a buffer for transmission to a network device. The caller must have set the device and priority and built the buffer before calling this function. The function can be called from an interrupt.

When calling this method, interrupts MUST be enabled. This is because the BH enable code must have IRQs enabled so that it will not deadlock.

Regardless of the return value, the skb is consumed, so it is currently difficult to retry a send to this method. (You can bump the ref count before sending to hold a reference for retry if you are careful.)

Return

- 0 - buffer successfully transmitted
- positive qdisc return code - NET_XMIT_DROP etc.
- negative errno - other errors

```
bool rps_may_expire_flow(struct net_device *dev, u16 rxq_index, u32 flow_id, u16 filter_id)
    check whether an RFS hardware filter may be removed
```

Parameters

struct net_device *dev

Device on which the filter was set

u16 rxq_index

RX queue index

u32 flow_id

Flow ID passed to ndo_rx_flow_steer()

u16 filter_id

Filter ID returned by ndo_rx_flow_steer()

Description

Drivers that implement ndo_rx_flow_steer() should periodically call this function for each installed filter and remove the filters for which it returns true.

int __netif_rx(struct sk_buff *skb)

Slightly optimized version of netif_rx

Parameters

struct sk_buff *skb

buffer to post

This behaves as netif_rx except that it does not disable bottom halves. As a result this function may only be invoked from the interrupt context (either hard or soft interrupt).

int netif_rx(struct sk_buff *skb)

post buffer to the network code

Parameters

struct sk_buff *skb

buffer to post

This function receives a packet from a device driver and queues it for the upper (protocol) levels to process via the backlog NAPI device. It always succeeds. The buffer may be dropped during processing for congestion control or by the protocol layers. The network buffer is passed via the backlog NAPI device. Modern NIC driver should use NAPI and GRO. This function can be used from interrupt and from process context. The caller from process context must not disable interrupts before invoking this function.

return values: NET_RX_SUCCESS (no congestion) NET_RX_DROP (packet was dropped)

bool netdev_is_rx_handler_busy(struct net_device *dev)

check if receive handler is registered

Parameters

struct net_device *dev

device to check

Check if a receive handler is already registered for a given device. Return true if there is one.

The caller must hold the rtnl_mutex.

int netdev_rx_handler_register(struct net_device *dev, rx_handler_func_t *rx_handler, void *rx_handler_data)

register receive handler

Parameters

struct net_device *dev

device to register a handler for

rx_handler_func_t *rx_handler

receive handler to register

void *rx_handler_data

data pointer that is used by rx handler

Register a receive handler for a device. This handler will then be called from `_netif_receive_skb`. A negative errno code is returned on a failure.

The caller must hold the `rtnl_mutex`.

For a general description of `rx_handler`, see enum `rx_handler_result`.

void netdev_rx_handler_unregister(struct net_device *dev)

unregister receive handler

Parameters**struct net_device *dev**

device to unregister a handler from

Unregister a receive handler from a device.

The caller must hold the `rtnl_mutex`.

int netif_receive_skb_core(struct sk_buff *skb)special purpose version of `netif_receive_skb`**Parameters****struct sk_buff *skb**

buffer to process

More direct receive version of `netif_receive_skb()`. It should only be used by callers that have a need to skip RPS and Generic XDP. Caller must also take care of handling if `(page_is_)pfmemalloc`.

This function may only be called from softirq context and interrupts should be enabled.

Return values (usually ignored): NET_RX_SUCCESS: no congestion NET_RX_DROP: packet was dropped

int netif_receive_skb(struct sk_buff *skb)

process receive buffer from network

Parameters**struct sk_buff *skb**

buffer to process

`netif_receive_skb()` is the main receive data processing function. It always succeeds. The buffer may be dropped during processing for congestion control or by the protocol layers.

This function may only be called from softirq context and interrupts should be enabled.

Return values (usually ignored): NET_RX_SUCCESS: no congestion NET_RX_DROP: packet was dropped

void netif_receive_skb_list(struct list_head *head)

process many receive buffers from network

Parameters

struct list_head *head

list of skbs to process.

Since return value of [*netif_receive_skb\(\)*](#) is normally ignored, and wouldn't be meaningful for a list, this function returns void.

This function may only be called from softirq context and interrupts should be enabled.

void __napi_schedule(struct napi_struct *n)

schedule for receive

Parameters

struct napi_struct *n

entry to schedule

Description

The entry's receive function will be scheduled to run. Consider using [*__napi_schedule_irqoff\(\)*](#) if hard irqs are masked.

bool napi_schedule_prep(struct napi_struct *n)

check if napi can be scheduled

Parameters

struct napi_struct *n

napi context

Description

Test if NAPI routine is already running, and if not mark it as running. This is used as a condition variable to insure only one NAPI poll instance runs. We also make sure there is no pending NAPI disable.

void __napi_schedule_irqoff(struct napi_struct *n)

schedule for receive

Parameters

struct napi_struct *n

entry to schedule

Description

Variant of [*__napi_schedule\(\)*](#) assuming hard irqs are masked.

On PREEMPT_RT enabled kernels this maps to [*__napi_schedule\(\)*](#) because the interrupt disabled assumption might not be true due to force-threaded interrupts and spinlock substitution.

void netif_queue_set_napi(struct net_device *dev, unsigned int queue_index, enum netdev_queue_type type, struct napi_struct *napi)

Associate queue with the napi

Parameters

struct net_device *dev

device to which NAPI and queue belong

unsigned int queue_index

Index of queue

enum netdev_queue_type type

queue type as RX or TX

struct napi_struct *napi

NAPI context, pass NULL to clear previously set NAPI

Description

Set queue with its corresponding napi context. This should be done after registering the NAPI handler for the queue-vector and the queues have been mapped to the corresponding interrupt vector.

void napi_enable(struct napi_struct *n)

enable NAPI scheduling

Parameters

struct napi_struct *n

NAPI context

Description

Resume NAPI from being scheduled on this context. Must be paired with napi_disable.

bool netdev_has_upper_dev(struct net_device *dev, struct net_device *upper_dev)

Check if device is linked to an upper device

Parameters

struct net_device *dev

device

struct net_device *upper_dev

upper device to check

Description

Find out if a device is linked to specified upper device and return true in case it is. Note that this checks only immediate upper device, not through a complete stack of devices. The caller must hold the RTNL lock.

bool netdev_has_upper_dev_all_rcu(struct net_device *dev, struct net_device *upper_dev)

Check if device is linked to an upper device

Parameters

struct net_device *dev

device

struct net_device *upper_dev

upper device to check

Description

Find out if a device is linked to specified upper device and return true in case it is. Note that this checks the entire upper device chain. The caller must hold rcu lock.

`bool netdev_has_any_upper_dev(struct net_device *dev)`

Check if device is linked to some device

Parameters

`struct net_device *dev`

device

Description

Find out if a device is linked to an upper device and return true in case it is. The caller must hold the RTNL lock.

`struct net_device *netdev_master_upper_dev_get(struct net_device *dev)`

Get master upper device

Parameters

`struct net_device *dev`

device

Description

Find a master upper device and return pointer to it or NULL in case it's not there. The caller must hold the RTNL lock.

`struct net_device *netdev_upper_get_next_dev_rcu(struct net_device *dev, struct list_head **iter)`

Get the next dev from upper list

Parameters

`struct net_device *dev`

device

`struct list_head **iter`

list_head ** of the current position

Description

Gets the next device from the dev's upper list, starting from iter position. The caller must hold RCU read lock.

`void *netdev_lower_get_next_private(struct net_device *dev, struct list_head **iter)`

Get the next ->private from the lower neighbour list

Parameters

`struct net_device *dev`

device

`struct list_head **iter`

list_head ** of the current position

Description

Gets the next netdev_adjacent->private from the dev's lower neighbour list, starting from iter position. The caller must hold either hold the RTNL lock or its own locking that guarantees that the neighbour lower list will remain unchanged.

```
void *netdev_lower_get_next_private_rcu(struct net_device *dev, struct list_head **iter)
```

Get the next ->private from the lower neighbour list, RCU variant

Parameters

struct net_device *dev
device

struct list_head **iter
list_head ** of the current position

Description

Gets the next netdev_adjacent->private from the dev's lower neighbour list, starting from iter position. The caller must hold RCU read lock.

```
void *netdev_lower_get_next(struct net_device *dev, struct list_head **iter)
```

Get the next device from the lower neighbour list

Parameters

struct net_device *dev
device

struct list_head **iter
list_head ** of the current position

Description

Gets the next netdev_adjacent from the dev's lower neighbour list, starting from iter position. The caller must hold RTNL lock or its own locking that guarantees that the neighbour lower list will remain unchanged.

```
void *netdev_lower_get_first_private_rcu(struct net_device *dev)
```

Get the first ->private from the lower neighbour list, RCU variant

Parameters

struct net_device *dev
device

Description

Gets the first netdev_adjacent->private from the dev's lower neighbour list. The caller must hold RCU read lock.

```
struct net_device *netdev_master_upper_dev_get_rcu(struct net_device *dev)
```

Get master upper device

Parameters

struct net_device *dev
device

Description

Find a master upper device and return pointer to it or NULL in case it's not there. The caller must hold the RCU read lock.

```
int netdev_upper_dev_link(struct net_device *dev, struct net_device *upper_dev, struct  
                           netlink_ext_ack *extack)
```

Add a link to the upper device

Parameters

```
struct net_device *dev  
    device  
struct net_device *upper_dev  
    new upper device  
struct netlink_ext_ack *extack  
    netlink extended ack
```

Description

Adds a link to device which is upper to this one. The caller must hold the RTNL lock. On a failure a negative errno code is returned. On success the reference counts are adjusted and the function returns zero.

```
int netdev_master_upper_dev_link(struct net_device *dev, struct net_device *upper_dev,  
                                 void *upper_priv, void *upper_info, struct  
                                 netlink_ext_ack *extack)
```

Add a master link to the upper device

Parameters

```
struct net_device *dev  
    device  
struct net_device *upper_dev  
    new upper device  
void *upper_priv  
    upper device private  
void *upper_info  
    upper info to be passed down via notifier  
struct netlink_ext_ack *extack  
    netlink extended ack
```

Description

Adds a link to device which is upper to this one. In this case, only one master upper device can be linked, although other non-master devices might be linked as well. The caller must hold the RTNL lock. On a failure a negative errno code is returned. On success the reference counts are adjusted and the function returns zero.

```
void netdev_upper_dev_unlink(struct net_device *dev, struct net_device *upper_dev)
```

Removes a link to upper device

Parameters

```
struct net_device *dev  
    device  
struct net_device *upper_dev  
    new upper device
```

Description

Removes a link to device which is upper to this one. The caller must hold the RTNL lock.

```
void netdev_bonding_info_change(struct net_device *dev, struct netdev_bonding_info
                                *bonding_info)
```

Dispatch event about slave change

Parameters

struct net_device *dev
device

struct netdev_bonding_info *bonding_info
info to dispatch

Description

Send NETDEV_BONDING_INFO to netdev notifiers with info. The caller must hold the RTNL lock.

```
struct net_device *netdev_get_xmit_slave(struct net_device *dev, struct sk_buff *skb, bool
                                         all_slaves)
```

Get the xmit slave of master device

Parameters

struct net_device *dev
device

struct sk_buff *skb
The packet

bool all_slaves
assume all the slaves are active

Description

The reference counters are not incremented so the caller must be careful with locks. The caller must hold RCU lock. NULL is returned if no slave is found.

```
struct net_device *netdev_sk_get_lowest_dev(struct net_device *dev, struct sock *sk)
```

Get the lowest device in chain given device and socket

Parameters

struct net_device *dev
device

struct sock *sk
the socket

Description

NULL is returned if no lower device is found.

```
void netdev_lower_state_changed(struct net_device *lower_dev, void *lower_state_info)
```

Dispatch event about lower device state change

Parameters

struct net_device *lower_dev

device

void *lower_state_info

state to dispatch

Description

Send NETDEV_CHANGELOWERSTATE to netdev notifiers with info. The caller must hold the RTNL lock.

int dev_set_promiscuity(struct net_device *dev, int inc)

update promiscuity count on a device

Parameters

struct net_device *dev

device

int inc

modifier

Add or remove promiscuity from a device. While the count in the device remains above zero the interface remains promiscuous. Once it hits zero the device reverts back to normal filtering operation. A negative inc value is used to drop promiscuity on the device. Return 0 if successful or a negative errno code on error.

int dev_set_allmulti(struct net_device *dev, int inc)

update allmulti count on a device

Parameters

struct net_device *dev

device

int inc

modifier

Add or remove reception of all multicast frames to a device. While the count in the device remains above zero the interface remains listening to all interfaces. Once it hits zero the device reverts back to normal filtering operation. A negative inc value is used to drop the counter when releasing a resource needing all multicasts. Return 0 if successful or a negative errno code on error.

unsigned int dev_get_flags(const struct net_device *dev)

get flags reported to userspace

Parameters

const struct net_device *dev

device

Get the combination of flag bits exported through APIs to userspace.

int dev_change_flags(struct net_device *dev, unsigned int flags, struct netlink_ext_ack *extack)

change device settings

Parameters

```
struct net_device *dev
    device
unsigned int flags
    device state flags
struct netlink_ext_ack *extack
    netlink extended ack
```

Change settings on device based state flags. The flags are in the userspace exported format.

```
int dev_pre_changeaddr_notify(struct net_device *dev, const char *addr, struct netlink_ext_ack *extack)
```

Call NETDEV_PRE_CHANGEADDR.

Parameters

```
struct net_device *dev
    device
const char *addr
    new address
struct netlink_ext_ack *extack
    netlink extended ack
```

```
int dev_set_mac_address(struct net_device *dev, struct sockaddr *sa, struct netlink_ext_ack *extack)
```

Change Media Access Control Address

Parameters

```
struct net_device *dev
    device
struct sockaddr *sa
    new address
struct netlink_ext_ack *extack
    netlink extended ack
```

Change the hardware (MAC) address of the device

```
int dev_get_port_parent_id(struct net_device *dev, struct netdev_phys_item_id *ppid, bool recurse)
```

Get the device's port parent identifier

Parameters

```
struct net_device *dev
    network device
struct netdev_phys_item_id *ppid
    pointer to a storage for the port's parent identifier
bool recurse
    allow/disallow recursion to lower devices
```

Get the devices's port parent identifier

`bool netdev_port_same_id(struct net_device *a, struct net_device *b)`

Indicate if two network devices have the same port parent identifier

Parameters

struct net_device *a

first network device

struct net_device *b

second network device

`void netdev_update_features(struct net_device *dev)`

recalculate device features

Parameters

struct net_device *dev

the device to check

Recalculate dev->features set and send notifications if it has changed. Should be called after driver or hardware dependent conditions might have changed that influence the features.

`void netdev_change_features(struct net_device *dev)`

recalculate device features

Parameters

struct net_device *dev

the device to check

Recalculate dev->features set and send notifications even if they have not changed. Should be called instead of `netdev_update_features()` if also dev->vlan_features might have changed to allow the changes to be propagated to stacked VLAN devices.

`void netif_stacked_transfer_operstate(const struct net_device *rootdev, struct net_device *dev)`

transfer operstate

Parameters

const struct net_device *rootdev

the root or lower level device to transfer state from

struct net_device *dev

the device to transfer operstate to

Transfer operational state from root to device. This is normally called when a stacking relationship exists between the root device and the device(a leaf device).

`int register_netdevice(struct net_device *dev)`

register a network device

Parameters

struct net_device *dev

device to register

Description

Take a prepared network device structure and make it externally accessible. A NETDEV_REGISTER message is sent to the netdev notifier chain. Callers must hold the rtnl lock - you may want `register_netdev()` instead of this.

```
int init_dummy_netdev(struct net_device *dev)
    init a dummy network device for NAPI
```

Parameters

```
struct net_device *dev
    device to init
```

This takes a network device structure and initialize the minimum amount of fields so it can be used to schedule NAPI polls without registering a full blown interface. This is to be used by drivers that need to tie several hardware interfaces to a single NAPI poll scheduler due to HW limitations.

```
int register_netdev(struct net_device *dev)
    register a network device
```

Parameters

```
struct net_device *dev
    device to register
```

Take a completed network device structure and add it to the kernel interfaces. A NETDEV_REGISTER message is sent to the netdev notifier chain. 0 is returned on success. A negative errno code is returned on a failure to set up the device, or if the name is a duplicate.

This is a wrapper around `register_netdevice` that takes the rtnl semaphore and expands the device name if you passed a format string to `alloc_netdev`.

```
struct rtnl_link_stats64 *dev_get_stats(struct net_device *dev, struct rtnl_link_stats64 *storage)
    get network device statistics
```

Parameters

```
struct net_device *dev
    device to get statistics from
```

```
struct rtnl_link_stats64 *storage
    place to store stats
```

Get network statistics from device. Return **storage**. The device driver may provide its own method by setting `dev->netdev_ops->get_stats64` or `dev->netdev_ops->get_stats`; otherwise the internal statistics structure is used.

```
void dev_fetch_sw_netstats(struct rtnl_link_stats64 *s, const struct pcpu_sw_netstats __percpu *netstats)
    get per-cpu network device statistics
```

Parameters

```
struct rtnl_link_stats64 *s
    place to store stats
```

```
const struct pcpu_sw_netstats __percpu *netstats
    per-cpu network stats to read from

    Read per-cpu network statistics and populate the related fields in s.

void dev_get_tstats64(struct net_device *dev, struct rtnl_link_stats64 *s)
    ndo_get_stats64 implementation
```

Parameters

struct net_device *dev
device to get statistics from

struct rtnl_link_stats64 *s
place to store stats

Populate **s** from dev->stats and dev->tstats. Can be used as ndo_get_stats64() callback.

```
void netdev_sw_irq_coalesce_default_on(struct net_device *dev)
    enable SW IRQ coalescing by default
```

Parameters

struct net_device *dev
netdev to enable the IRQ coalescing on

Description

Sets a conservative default for SW IRQ coalescing. Users can use sysfs attributes to override the default values.

```
struct net_device *alloc_netdev_mqs(int sizeof_priv, const char *name, unsigned char
                                    name_assign_type, void (*setup)(struct net_device *),
                                    unsigned int txqs, unsigned int rxqs)

    allocate network device
```

Parameters

int sizeof_priv
size of private data to allocate space for

const char *name
device name format string

unsigned char name_assign_type
origin of device name

void (*setup)(struct net_device *)
callback to initialize device

unsigned int txqs
the number of TX subqueues to allocate

unsigned int rxqs
the number of RX subqueues to allocate

Description

Allocates a *struct net_device* with private data area for driver use and performs basic initialization. Also allocates subqueue structs for each queue on the device.

```
void free_netdev(struct net_device *dev)
    free network device
```

Parameters

struct net_device *dev
device

Description

This function does the last stage of destroying an allocated device interface. The reference to the device object is released. If this is the last reference then it will be freed. Must be called in process context.

```
void synchronize_net(void)
    Synchronize with packet receive processing
```

Parameters

void
no arguments

Description

Wait for packets currently being received to be done. Does not block later packets from starting.

```
void unregister_netdevice_queue(struct net_device *dev, struct list_head *head)
    remove device from the kernel
```

Parameters

struct net_device *dev
device

struct list_head *head
list

This function shuts down a device interface and removes it from the kernel tables. If head not NULL, device is queued to be unregistered later.

Callers must hold the rtnl semaphore. You may want [*unregister_netdev\(\)*](#) instead of this.

```
void unregister_netdevice_many(struct list_head *head)
    unregister many devices
```

Parameters

struct list_head *head
list of devices

Note**As most callers use a stack allocated list_head,**

we force a list_del() to make sure stack wont be corrupted later.

```
void unregister_netdev(struct net_device *dev)
    remove device from the kernel
```

Parameters

```
struct net_device *dev  
device
```

This function shuts down a device interface and removes it from the kernel tables.

This is just a wrapper for unregister_netdevice that takes the rtnl semaphore. In general you want to use this and not unregister_netdevice.

```
int __dev_change_net_namespace(struct net_device *dev, struct net *net, const char *pat, int  
new_ifindex)
```

move device to different nethost namespace

Parameters

```
struct net_device *dev  
device
```

```
struct net *net  
network namespace
```

```
const char *pat
```

If not NULL name pattern to try if the current device name is already taken in the destination network namespace.

```
int new_ifindex
```

If not zero, specifies device index in the target namespace.

This function shuts down a device interface and moves it to a new network namespace. On success 0 is returned, on a failure a negative errno code is returned.

Callers must hold the rtnl semaphore.

```
netdev_features_t netdev_increment_features(netdev_features_t all, netdev_features_t one,  
netdev_features_t mask)
```

increment feature set by one

Parameters

```
netdev_features_t all  
current feature set
```

```
netdev_features_t one  
new feature set
```

```
netdev_features_t mask  
mask feature set
```

Computes a new feature set after adding a device with feature set **one** to the master device with current feature set **all**. Will not enable anything that is off in **mask**. Returns the new feature set.

```
int eth_header(struct sk_buff *skb, struct net_device *dev, unsigned short type, const void  
*daddr, const void *saddr, unsigned int len)
```

create the Ethernet header

Parameters

```
struct sk_buff *skb  
buffer to alter
```

```

struct net_device *dev
    source device

unsigned short type
    Ethernet type field

const void *daddr
    destination address (NULL leave destination address)

const void *saddr
    source address (NULL use device source address)

unsigned int len
    packet length (<= skb->len)

```

Description

Set the protocol type. For a packet of type ETH_P_802_3/2 we put the length in here instead.

```

32 eth_get_headlen(const struct net_device *dev, const void *data, u32 len)
    determine the length of header for an ethernet frame

```

Parameters

```

const struct net_device *dev
    pointer to network device

const void *data
    pointer to start of frame

u32 len
    total length of frame

```

Description

Make a best effort attempt to pull the length for all of the headers for a given frame in a linear buffer.

```

__be16 eth_type_trans(struct sk_buff *skb, struct net_device *dev)
    determine the packet's protocol ID.

```

Parameters

```

struct sk_buff *skb
    received socket data

struct net_device *dev
    receiving network device

```

Description

The rule here is that we assume 802.3 if the type field is short enough to be a length. This is normal practice and works for any 'now in use' protocol.

```

int eth_header_parse(const struct sk_buff *skb, unsigned char *haddr)
    extract hardware address from packet

```

Parameters

```

const struct sk_buff *skb
    packet to extract header from

```

```
unsigned char *haddr
destination buffer

int eth_header_cache(const struct neighbour *neigh, struct hh_cache *hh, __be16 type)
fill cache entry from neighbour
```

Parameters

const struct neighbour *neigh

source neighbour

struct hh_cache *hh

destination cache entry

__be16 type

Ethernet type field

Description

Create an Ethernet header template from the neighbour.

```
void eth_header_cache_update(struct hh_cache *hh, const struct net_device *dev, const
                             unsigned char *haddr)
update cache entry
```

Parameters

struct hh_cache *hh

destination cache entry

const struct net_device *dev

network device

const unsigned char *haddr

new hardware address

Description

Called by Address Resolution module to notify changes in address.

```
__be16 eth_header_parse_protocol(const struct sk_buff *skb)
extract protocol from L2 header
```

Parameters

const struct sk_buff *skb

packet to extract protocol from

```
int eth_prepare_mac_addr_change(struct net_device *dev, void *p)
prepare for mac change
```

Parameters

struct net_device *dev

network device

void *p

socket address

```
void eth_commit_mac_addr_change(struct net_device *dev, void *p)
    commit mac change
```

Parameters

```
struct net_device *dev
    network device
```

```
void *p
    socket address
```

```
int eth_mac_addr(struct net_device *dev, void *p)
    set new Ethernet hardware address
```

Parameters

```
struct net_device *dev
    network device
```

```
void *p
    socket address
```

Description

Change hardware address of device.

This doesn't change hardware matching, so needs to be overridden for most real devices.

```
void ether_setup(struct net_device *dev)
    setup Ethernet network device
```

Parameters

```
struct net_device *dev
    network device
```

Description

Fill in the fields of the device structure with Ethernet-generic values.

```
struct net_device *alloc_etherdev_mqs(int sizeof_priv, unsigned int txqs, unsigned int rxqs)
    Allocates and sets up an Ethernet device
```

Parameters

```
int sizeof_priv
    Size of additional driver-private structure to be allocated for this Ethernet device
```

```
unsigned int txqs
    The number of TX queues this device has.
```

```
unsigned int rxqs
    The number of RX queues this device has.
```

Description

Fill in the fields of the device structure with Ethernet-generic values. Basically does everything except registering the device.

Constructs a new net device, complete with a private data area of size (sizeof_priv). A 32-byte (not bit) alignment is enforced for this private data area.

```
int platform_get_ethdev_address(struct device *dev, struct net_device *netdev)
```

Set netdev's MAC address from a given device

Parameters

struct device *dev

Pointer to the device

struct net_device *netdev

Pointer to netdev to write the address to

Description

Wrapper around eth_platform_get_mac_address() which writes the address directly to netdev->dev_addr.

```
int fwnode_get_mac_address(struct fwnode_handle *fwnode, char *addr)
```

Get the MAC from the firmware node

Parameters

struct fwnode_handle *fwnode

Pointer to the firmware node

char *addr

Address of buffer to store the MAC in

Description

Search the firmware node for the best MAC address to use. 'mac-address' is checked first, because that is supposed to contain the "most recent" MAC address. If that isn't set, then 'local-mac-address' is checked next, because that is the default address. If that isn't set, then the obsolete 'address' is checked, just in case we're using an old device tree.

Note that the 'address' property is supposed to contain a virtual address of the register set, but some DTS files have redefined that property to be the MAC address.

All-zero MAC addresses are rejected, because those could be properties that exist in the firmware tables, but were not updated by the firmware. For example, the DTS could define 'mac-address' and 'local-mac-address', with zero MAC addresses. Some older U-Boots only initialized 'local-mac-address'. In this case, the real MAC is in 'local-mac-address', and 'mac-address' exists but is all zeros.

```
int device_get_mac_address(struct device *dev, char *addr)
```

Get the MAC for a given device

Parameters

struct device *dev

Pointer to the device

char *addr

Address of buffer to store the MAC in

```
int device_get_ethdev_address(struct device *dev, struct net_device *netdev)
```

Set netdev's MAC address from a given device

Parameters

struct device *dev

Pointer to the device

struct net_device *netdev

Pointer to netdev to write the address to

DescriptionWrapper around `device_get_mac_address()` which writes the address directly to netdev->dev_addr.**void netif_carrier_on(struct net_device *dev)**

set carrier

Parameters**struct net_device *dev**

network device

Description

Device has detected acquisition of carrier.

void netif_carrier_off(struct net_device *dev)

clear carrier

Parameters**struct net_device *dev**

network device

Description

Device has detected loss of carrier.

void netif_carrier_event(struct net_device *dev)

report carrier state event

Parameters**struct net_device *dev**

network device

Description

Device has detected a carrier event but the carrier state wasn't changed. Use in drivers when querying carrier state asynchronously, to avoid missing events (link flaps) if link recovers before it's queried.

bool is_link_local_ether_addr(const u8 *addr)

Determine if given Ethernet address is link-local

Parameters**const u8 *addr**

Pointer to a six-byte array containing the Ethernet address

Description

Return true if address is link local reserved addr (01:80:c2:00:00:0X) per IEEE 802.1Q 8.6.3 Frame filtering.

Please note: addr must be aligned to u16.

bool `is_zero_ether_addr`(const u8 *addr)

Determine if give Ethernet address is all zeros.

Parameters

const u8 *addr

Pointer to a six-byte array containing the Ethernet address

Description

Return true if the address is all zeroes.

Please note: addr must be aligned to u16.

bool `is_multicast_ether_addr`(const u8 *addr)

Determine if the Ethernet address is a multicast.

Parameters

const u8 *addr

Pointer to a six-byte array containing the Ethernet address

Description

Return true if the address is a multicast address. By definition the broadcast address is also a multicast address.

bool `is_local_ether_addr`(const u8 *addr)

Determine if the Ethernet address is locally-assigned one (IEEE 802).

Parameters

const u8 *addr

Pointer to a six-byte array containing the Ethernet address

Description

Return true if the address is a local address.

bool `is_broadcast_ether_addr`(const u8 *addr)

Determine if the Ethernet address is broadcast

Parameters

const u8 *addr

Pointer to a six-byte array containing the Ethernet address

Description

Return true if the address is the broadcast address.

Please note: addr must be aligned to u16.

bool `is_unicast_ether_addr`(const u8 *addr)

Determine if the Ethernet address is unicast

Parameters

const u8 *addr

Pointer to a six-byte array containing the Ethernet address

Description

Return true if the address is a unicast address.

bool `is_valid_ether_addr`(const u8 *addr)

Determine if the given Ethernet address is valid

Parameters

const u8 *addr

Pointer to a six-byte array containing the Ethernet address

Description

Check that the Ethernet address (MAC) is not 00:00:00:00:00:00, is not a multicast address, and is not FF:FF:FF:FF:FF:FF.

Return true if the address is valid.

Please note: addr must be aligned to u16.

bool `eth_proto_is_802_3`(__be16 proto)

Determine if a given Ethertype/length is a protocol

Parameters

__be16 proto

Ethertype/length value to be tested

Description

Check that the value from the Ethertype/length field is a valid Ethertype.

Return true if the valid is an 802.3 supported Ethertype.

void `eth_random_addr`(u8 *addr)

Generate software assigned random Ethernet address

Parameters

u8 *addr

Pointer to a six-byte array containing the Ethernet address

Description

Generate a random Ethernet address (MAC) that is not multicast and has the local assigned bit set.

void `eth_broadcast_addr`(u8 *addr)

Assign broadcast address

Parameters

u8 *addr

Pointer to a six-byte array containing the Ethernet address

Description

Assign the broadcast address to the given address array.

`void eth_zero_addr(u8 *addr)`

Assign zero address

Parameters

`u8 *addr`

Pointer to a six-byte array containing the Ethernet address

Description

Assign the zero address to the given address array.

`void eth_hw_addr_random(struct net_device *dev)`

Generate software assigned random Ethernet and set device flag

Parameters

`struct net_device *dev`

pointer to net_device structure

Description

Generate a random Ethernet address (MAC) to be used by a net device and set addr_assign_type so the state can be read by sysfs and be used by userspace.

`u32 eth_hw_addr_crc(struct netdev_hw_addr *ha)`

Calculate CRC from netdev_hw_addr

Parameters

`struct netdev_hw_addr *ha`

pointer to hardware address

Description

Calculate CRC from a hardware address as basis for filter hashes.

`void ether_addr_copy(u8 *dst, const u8 *src)`

Copy an Ethernet address

Parameters

`u8 *dst`

Pointer to a six-byte array Ethernet address destination

`const u8 *src`

Pointer to a six-byte array Ethernet address source

Description

Please note: dst & src must both be aligned to u16.

`void eth_hw_addr_set(struct net_device *dev, const u8 *addr)`

Assign Ethernet address to a net_device

Parameters

`struct net_device *dev`

pointer to net_device structure

`const u8 *addr`

address to assign

Description

Assign given address to the net_device, addr_assign_type is not changed.

void eth_hw_addr_inherit(struct net_device *dst, struct net_device *src)

Copy dev_addr from another net_device

Parameters

struct net_device *dst

pointer to net_device to copy dev_addr to

struct net_device *src

pointer to net_device to copy dev_addr from

Description

Copy the Ethernet address from one net_device to another along with the address attributes (addr_assign_type).

bool ether_addr_equal(const u8 *addr1, const u8 *addr2)

Compare two Ethernet addresses

Parameters

const u8 *addr1

Pointer to a six-byte array containing the Ethernet address

const u8 *addr2

Pointer other six-byte array containing the Ethernet address

Description

Compare two Ethernet addresses, returns true if equal

Please note: addr1 & addr2 must both be aligned to u16.

bool ether_addr_equal_64bits(const u8 *addr1, const u8 *addr2)

Compare two Ethernet addresses

Parameters

const u8 *addr1

Pointer to an array of 8 bytes

const u8 *addr2

Pointer to an other array of 8 bytes

Description

Compare two Ethernet addresses, returns true if equal, false otherwise.

The function doesn't need any conditional branches and possibly uses word memory accesses on CPU allowing cheap unaligned memory reads. arrays = { byte1, byte2, byte3, byte4, byte5, byte6, pad1, pad2 }

Please note that alignment of addr1 & addr2 are only guaranteed to be 16 bits.

bool ether_addr_equal_unaligned(const u8 *addr1, const u8 *addr2)

Compare two not u16 aligned Ethernet addresses

Parameters

const u8 *addr1

Pointer to a six-byte array containing the Ethernet address

const u8 *addr2

Pointer other six-byte array containing the Ethernet address

Description

Compare two Ethernet addresses, returns true if equal

Please note: Use only when any Ethernet address may not be u16 aligned.

bool ether_addr_equal_masked(const u8 *addr1, const u8 *addr2, const u8 *mask)

Compare two Ethernet addresses with a mask

Parameters

const u8 *addr1

Pointer to a six-byte array containing the 1st Ethernet address

const u8 *addr2

Pointer to a six-byte array containing the 2nd Ethernet address

const u8 *mask

Pointer to a six-byte array containing the Ethernet address bitmask

Description

Compare two Ethernet addresses with a mask, returns true if for every bit set in the bitmask the equivalent bits in the ethernet addresses are equal. Using a mask with all bits set is a slower ether_addr_equal.

u64 ether_addr_to_u64(const u8 *addr)

Convert an Ethernet address into a u64 value.

Parameters

const u8 *addr

Pointer to a six-byte array containing the Ethernet address

Description

Return a u64 value of the address

void u64_to_ether_addr(u64 u, u8 *addr)

Convert a u64 to an Ethernet address.

Parameters

u64 u

u64 to convert to an Ethernet MAC address

u8 *addr

Pointer to a six-byte array to contain the Ethernet address

void eth_addr_dec(u8 *addr)

Decrement the given MAC address

Parameters

u8 *addr

Pointer to a six-byte array containing Ethernet address to decrement

void eth_addr_inc(u8 *addr)
Increment the given MAC address.

Parameters

u8 *addr
Pointer to a six-byte array containing Ethernet address to increment.

void eth_addr_add(u8 *addr, long offset)
Add (or subtract) an offset to/from the given MAC address.

Parameters

u8 *addr
Pointer to a six-byte array containing Ethernet address to increment.

long offset
Offset to add.

bool is_etherdev_addr(const struct net_device *dev, const u8 addr[6 + 2])
Tell if given Ethernet address belongs to the device.

Parameters

const struct net_device *dev
Pointer to a device structure

const u8 addr[6 + 2]
Pointer to a six-byte array containing the Ethernet address

Description

Compare passed address with all addresses of the device. Return true if the address if one of the device addresses.

Note that this function calls [ether_addr_equal_64bits\(\)](#) so take care of the right padding.

unsigned long compare_ether_header(const void *a, const void *b)
Compare two Ethernet headers

Parameters

const void *a
Pointer to Ethernet header

const void *b
Pointer to Ethernet header

Description

Compare two Ethernet headers, returns 0 if equal. This assumes that the network header (i.e., IP header) is 4-byte aligned OR the platform can handle unaligned access. This is the case for all packets coming into netif_receive_skb or similar entry points.

void eth_hw_addr_gen(struct net_device *dev, const u8 *base_addr, unsigned int id)
Generate and assign Ethernet address to a port

Parameters

struct net_device *dev
pointer to port's net_device structure

const u8 *base_addr
base Ethernet address

unsigned int id
offset to add to the base address

Description

Generate a MAC address using a base address and an offset and assign it to a net_device. Commonly used by switch drivers which need to compute addresses for all their ports. addr_assign_type is not changed.

int **eth_skb_pad**(struct *sk_buff* *skb)
Pad buffer to minimum number of octets for Ethernet frame

Parameters

struct sk_buff *skb
Buffer to pad

Description

An Ethernet frame should have a minimum size of 60 bytes. This function takes short frames and pads them with zeros up to the 60 byte limit.

bool **napi_is_scheduled**(struct napi_struct *n)
test if NAPI is scheduled

Parameters

struct napi_struct *n
NAPI context

Description

This check is "best-effort". With no locking implemented, a NAPI can be scheduled or terminate right after this check and produce not precise results.

NAPI_STATE_SCHED is an internal state, napi_is_scheduled should not be used normally and napi_schedule should be used instead.

Use only if the driver really needs to check if a NAPI is scheduled for example in the context of delayed timer that can be skipped if a NAPI is already scheduled.

Return True if NAPI is scheduled, False otherwise.

bool **napi_schedule**(struct napi_struct *n)
schedule NAPI poll

Parameters

struct napi_struct *n
NAPI context

Description

Schedule NAPI poll routine to be called if it is not already running. Return true if we schedule a NAPI or false if not. Refer to [*napi_schedule_prep\(\)*](#) for additional reason on why a NAPI might not be scheduled.

```
void napi_schedule_irqoff(struct napi_struct *n)
    schedule NAPI poll
```

Parameters

struct napi_struct *n
NAPI context

Description

Variant of [*napi_schedule\(\)*](#), assuming hard irqs are masked.

```
bool napi_complete_done(struct napi_struct *n, int work_done)
    NAPI processing complete
```

Parameters

struct napi_struct *n
NAPI context

int work_done
number of packets processed

Description

Mark NAPI processing as complete. Should only be called if poll budget has not been completely consumed. Prefer over [*napi_complete\(\)*](#). Return false if device should avoid rearming interrupts.

```
void napi_disable(struct napi_struct *n)
    prevent NAPI from scheduling
```

Parameters

struct napi_struct *n
NAPI context

Description

Stop NAPI from being scheduled on this context. Waits till any outstanding processing completes.

```
void napi_synchronize(const struct napi_struct *n)
    wait until NAPI is not running
```

Parameters

const struct napi_struct *n
NAPI context

Description

Wait until NAPI is done being scheduled on this context. Waits till any outstanding processing completes but does not disable future activations.

```
bool napi_if_scheduled_mark_missed(struct napi_struct *n)
    if napi is running, set the NAPIF_STATE_MISSED
```

Parameters

struct napi_struct *n
NAPI context

Description

If napi is running, set the NAPIF_STATE_MISSED, and return true if NAPI is scheduled.

enum `netdev_priv_flags`

`struct net_device` priv_flags

Constants

IFF_802_1Q_VLAN

802.1Q VLAN device

IFF_EBRIDGE

Ethernet bridging device

IFF_BONDING

bonding master or slave

IFF_ISATAP

ISATAP interface (RFC4214)

IFF_WAN_HDLC

WAN HDLC device

IFF_XMIT_DST_RELEASE

dev_hard_start_xmit() is allowed to release skb->dst

IFF_DONT_BRIDGE

disallow bridging this ether dev

IFF_DISABLE_NETPOLL

disable netpoll at run-time

IFF_MACVLAN_PORT

device used as macvlan port

IFF_BRIDGE_PORT

device used as bridge port

IFF_OVS_DATAPATH

device used as Open vSwitch datapath port

IFF_TX_SKB_SHARING

The interface supports sharing skbs on transmit

IFF_UNICAST_FLT

Supports unicast filtering

IFF_TEAM_PORT

device used as team port

IFF_SUPP_NOFC

device supports sending custom FCS

IFF_LIVE_ADDR_CHANGE

device supports hardware address change when it's running

IFF_MACVLAN

Macvlan device

IFF_XMIT_DST_RELEASE_PERM

IFF_XMIT_DST_RELEASE not taking into account underlying stacked devices

IFF_L3MDEV_MASTER

device is an L3 master device

IFF_NO_QUEUE

device can run without qdisc attached

IFF_OPENVSWITCH

device is a Open vSwitch master

IFF_L3MDEV_SLAVE

device is enslaved to an L3 master device

IFF_TEAM

device is a team device

IFF_RXFH_CONFIGURED

device has had Rx Flow indirection table configured

IFF_PHONY_HEADROOM

the headroom value is controlled by an external entity (i.e. the master device for bridged veth)

IFF_MACSEC

device is a MACsec device

IFF_NO_RX_HANDLER

device doesn't support the rx_handler hook

IFF_FAILOVER

device is a failover master device

IFF_FAILOVER_SLAVE

device is lower dev of a failover master device

IFF_L3MDEV_RX_HANDLER

only invoke the rx handler of L3 master device

IFF_NO_ADDRCONF

prevent ipv6 addrconf

IFF_TX_SKB_NO_LINEAR

device/driver is capable of xmitting frames with skb_headlen(skb) == 0 (data starts from frag0)

IFF_CHANGE_PROTO_DOWN

device supports setting carrier via IFLA_PROTO_DOWN

IFF_SEE_ALL_HWTSTAMP_REQUESTS

device wants to see calls to ndo_hwtstamp_set() for all timestamp requests regardless of source, even if those aren't HWTSTAMP_SOURCE_NETDEV.

Description

These are the *struct net_device*, they are only set internally by drivers and used in the kernel. These flags are invisible to userspace; this means that the order of these flags can change during any kernel release.

You should have a pretty good reason to be extending these flags.

struct net_device

The DEVICE structure.

Definition:

```
struct net_device {
    __cacheline_group_begin(net_device_read_tx);
    unsigned long long      priv_flags;
    const struct net_device_ops *netdev_ops;
    const struct header_ops *header_ops;
    struct netdev_queue     *_tx;
    netdev_features_t gso_partial_features;
    unsigned int            real_num_tx_queues;
    unsigned int            gso_max_size;
    unsigned int            gso_ipv4_max_size;
    u16 gso_max_segs;
    s16 num_tc;
    unsigned int            mtu;
    unsigned short          needed_headroom;
    struct netdev_tc_txq   tc_to_txq[TC_MAX_QUEUE];
#ifndef CONFIG_XPS;
    struct xps_dev_maps __rcu *xps_maps[XPS_MAPS_MAX];
#endif;
#ifndef CONFIG_NETFILTER_EGRESS;
    struct nf_hook_entries __rcu *nf_hooks_egress;
#endif;
#ifndef CONFIG_NET_XGREGS;
    struct bpf_mprog_entry __rcu *tcx_egress;
#endif;
    __cacheline_group_end(net_device_read_tx);
    __cacheline_group_begin(net_device_read_txrx);
    union {
        struct pcpu_lstats __percpu           *lstats;
        struct pcpu_sw_netstats __percpu       *tstats;
        struct pcpu_dstats __percpu           *dstats;
    };
    unsigned int            flags;
    unsigned short          hard_header_len;
    netdev_features_t       features;
    struct inet6_dev __rcu *ip6_ptr;
    __cacheline_group_end(net_device_read_txrx);
    __cacheline_group_begin(net_device_read_rx);
    struct bpf_prog __rcu  *xdp_prog;
    struct list_head        ptype_specific;
    int ifindex;
    unsigned int            real_num_rx_queues;
    struct netdev_rx_queue *_rx;
    unsigned long           gro_flush_timeout;
    int napi_defer_hard_irqs;
    unsigned int            gro_max_size;
    unsigned int            gro_ipv4_max_size;
    rx_handler_func_t __rcu *rx_handler;
```

```

void __rcu *rx_handler_data;
possible_net_t nd_net;
#endif CONFIG_NETPOLL;
struct netpoll_info __rcu *npinfo;
#endif;
#ifndef CONFIG_NET_XGRESS;
struct bpf_mprog_entry __rcu *tcx_ingress;
#endif;
__cacheline_group_end(net_device_read_rx);
char name[IFNAMSIZ];
struct netdev_name_node *name_node;
struct dev_ifalias __rcu *ifalias;
unsigned long mem_end;
unsigned long mem_start;
unsigned long base_addr;
unsigned long state;
struct list_head dev_list;
struct list_head napi_list;
struct list_head unreg_list;
struct list_head close_list;
struct list_head ptype_all;
struct {
    struct list_head upper;
    struct list_head lower;
} adj_list;
xdp_features_t xdp_features;
const struct xdp_metadata_ops *xdp_metadata_ops;
const struct xsk_tx_metadata_ops *xsk_tx_metadata_ops;
unsigned short gflags;
unsigned short needed_tailroom;
netdev_features_t hw_features;
netdev_features_t wanted_features;
netdev_features_t vlan_features;
netdev_features_t hw_enc_features;
netdev_features_t mpls_features;
unsigned int min_mtu;
unsigned int max_mtu;
unsigned short type;
unsigned char min_header_len;
unsigned char name_assign_type;
int group;
struct net_device_stats stats;
struct net_device_core_stats __percpu *core_stats;
atomic_t carrier_up_count;
atomic_t carrier_down_count;
#endif CONFIG_WIRELESS_EXT;
const struct iw_handler_def *wireless_handlers;
struct iw_public_data *wireless_data;
#endif;
const struct ethtool_ops *ethtool_ops;

```

```

#define CONFIG_NET_L3_MASTER_DEV;
    const struct l3mdev_ops *l3mdev_ops;
#endif;
#if IS_ENABLED(CONFIG_IPV6);
    const struct ndisc_ops *ndisc_ops;
#endif;
#ifdef CONFIG_XFRM_OFFLOAD;
    const struct xfrmdev_ops *xfrmdev_ops;
#endif;
#if IS_ENABLED(CONFIG_TLS_DEVICE);
    const struct tlsdev_ops *tlsdev_ops;
#endif;
    unsigned char          operstate;
    unsigned char          link_mode;
    unsigned char          if_port;
    unsigned char          dma;
    unsigned char          perm_addr[MAX_ADDR_LEN];
    unsigned char          addr_assign_type;
    unsigned char          addr_len;
    unsigned char          upper_level;
    unsigned char          lower_level;
    unsigned short         neigh_priv_len;
    unsigned short         dev_id;
    unsigned short         dev_port;
    unsigned short         padded;
    spinlock_t             addr_list_lock;
    int                   irq;
    struct netdev_hw_addr_list      uc;
    struct netdev_hw_addr_list      mc;
    struct netdev_hw_addr_list      dev_addrs;
#endif;
#ifdef CONFIG_SYSFS;
    struct kset              *queues_kset;
#endif;
#ifdef CONFIG_LOCKDEP;
    struct list_head          unlink_list;
#endif;
    unsigned int              promiscuity;
    unsigned int              allmulti;
    bool uc_promisc;
#endif;
#ifdef CONFIG_LOCKDEP;
    unsigned char              nested_level;
#endif;
    struct in_device __rcu *ip_ptr;
#endif;
#ifdef IS_ENABLED(CONFIG_VLAN_8021Q);
    struct vlan_info __rcu *vlan_info;
#endif;
#ifdef IS_ENABLED(CONFIG_NET_DSA);
    struct dsa_port           *dsa_ptr;
#endif;
#ifdef IS_ENABLED(CONFIG_TIPC);

```

```

    struct tipc_bearer __rcu *tipc_ptr;
#endif;
#if IS_ENABLED(CONFIG_ATALK);
    void *atalk_ptr;
#endif;
#if IS_ENABLED(CONFIG_AX25);
    void *ax25_ptr;
#endif;
#if IS_ENABLED(CONFIG_CFG80211);
    struct wireless_dev      *ieee80211_ptr;
#endif;
#if IS_ENABLED(CONFIG_IEEE802154) || IS_ENABLED(CONFIG_6LOWPAN);
    struct wpan_dev           *ieee802154_ptr;
#endif;
#if IS_ENABLED(CONFIG_MPLS_ROUTING);
    struct mpls_dev __rcu     *mpls_ptr;
#endif;
#if IS_ENABLED(CONFIG_MCTP);
    struct mctp_dev __rcu     *mctp_ptr;
#endif;
    const unsigned char        *dev_addr;
    unsigned int                num_rx_queues;
#define GRO_LEGACY_MAX_SIZE      65536u;
#define GRO_MAX_SIZE             (8 * 65535u);
    unsigned int                xdp_zc_max_segs;
    struct netdev_queue __rcu *ingress_queue;
#endif CONFIG_NETFILTER_INGRESS;
    struct nf_hook_entries __rcu *nf_hooks_ingress;
#endif;
    unsigned char                broadcast[MAX_ADDR_LEN];
#endif CONFIG_RFS_ACCEL;
    struct cpu_rmap              *rx_cpu_rmap;
#endif;
    struct hlist_node            index_hlist;
    unsigned int                 num_tx_queues;
    struct Qdisc __rcu          *qdisc;
    unsigned int                 tx_queue_len;
    spinlock_t tx_global_lock;
    struct xdp_dev_bulk_queue __percpu *xdp_bulkq;
#endif CONFIG_NET_SCHED;
    unsigned long qdisc_hash[1 << ((4) - 1)];
#endif;
    struct timer_list             watchdog_timer;
    int watchdog_timeo;
    u32 proto_down_reason;
    struct list_head              todo_list;
#endif CONFIG_PCPU_DEV_REFCNT;
    int __percpu                  *pcpu_refcnt;
#else;
    refcount_t dev_refcnt;

```

```

#endif;
    struct ref_tracker_dir  refcnt_tracker;
    struct list_head         link_watch_list;
    enum {
        NETREG_UNINITIALIZED=0,
        NETREG_REGISTERED,
        NETREG_UNREGISTERING,
        NETREG_UNREGISTERED,
        NETREG_RELEASED,
        NETREG_DUMMY,
    } reg_state:8;
    bool dismantle;
    enum {
        RTNL_LINK_INITIALIZED,
        RTNL_LINK_INITIALIZING,
    } rtnl_link_state:16;
    bool needs_free_netdev;
    void (*priv_destructor)(struct net_device *dev);
    void *ml_priv;
    enum netdev_ml_priv_type      ml_priv_type;
    enum netdev_stat_type        pcpu_stat_type:8;
#ifndef IS_ENABLED(CONFIG_GARP);
    struct garp_port __rcu *garp_port;
#endif;
#ifndef IS_ENABLED(CONFIG_MRP);
    struct mrp_port __rcu *mrp_port;
#endif;
#ifndef IS_ENABLED(CONFIG_NET_DROP_MONITOR);
    struct dm_hw_stat_delta __rcu *dm_private;
#endif;
    struct device               dev;
    const struct attribute_group *sysfs_groups[4];
    const struct attribute_group *sysfs_rx_queue_group;
    const struct rtnl_link_ops *rtnl_link_ops;
#define GSO_MAX_SEGS           65535u;
#define GSO_LEGACY_MAX_SIZE     65536u;
#define GSO_MAX_SIZE            (8 * GSO_MAX_SEGS);
#define TSO_LEGACY_MAX_SIZE     65536;
#define TSO_MAX_SIZE             UINT_MAX;
    unsigned int                tso_max_size;
#define TSO_MAX_SEGS            U16_MAX;
    u16 tso_max_segs;
#ifndef CONFIG_DCB;
    const struct dcbnl_rtnl_ops *dcbnl_ops;
#endif;
    u8 prio_tc_map[TC_BITMASK + 1];
#ifndef IS_ENABLED(CONFIG_FCOE);
    unsigned int                fcoe_ddp_xid;
#endif;
#ifndef IS_ENABLED(CONFIG_CGROUP_NET_PRIO);

```

```

    struct netprio_map __rcu *priomap;
#endif;
    struct phy_device      *phydev;
    struct sfp_bus         *sfp_bus;
    struct lock_class_key  *qdisc_tx_busylock;
    bool proto_down;
    unsigned wol_enabled:1;
    unsigned threaded:1;
    struct list_head        net_notifier_list;
#ifndef IS_ENABLED(CONFIG_MACSEC)
    const struct macsec_ops *macsec_ops;
#endif;
    const struct udp_tunnel_nic_info      *udp_tunnel_nic_info;
    struct udp_tunnel_nic   *udp_tunnel_nic;
    struct bpf_xdp_entity  xdp_state[__MAX_XDP_MODE];
    u8 dev_addr_shadow[MAX_ADDR_LEN];
    netdevice_tracker linkwatch_dev_tracker;
    netdevice_tracker watchdog_dev_tracker;
    netdevice_tracker dev_registered_tracker;
    struct rtnl_hw_stats64 *offload_xstats_l3;
    struct devlink_port     *devlink_port;
#ifndef IS_ENABLED(CONFIG_DPLL)
    struct dpll_pin __rcu  *dpll_pin;
#endif;
#ifndef IS_ENABLED(CONFIG_PAGE_POOL)
    struct hlist_head       page_pools;
#endif;
};

```

Members

priv_flags

Like 'flags' but invisible to userspace, see if.h for the definitions

netdev_ops

Includes several pointers to callbacks, if one wants to override the ndo_*() functions

header_ops

Includes callbacks for creating,parsing,caching,etc of Layer 2 headers.

_tx

Array of TX queues

gso_partial_features

value(s) from NETIF_F_GSO*

real_num_tx_queues

Number of TX queues currently active in device

gso_max_size

Maximum size of generic segmentation offload

gso_ipv4_max_size

Maximum size of generic segmentation offload, for IPv4.

gso_max_segs

Maximum number of segments that can be passed to the NIC for GSO

num_tc

Number of traffic classes in the net device

mtu

Interface MTU value

needed_headroom

Extra headroom the hardware may need, but not in all cases can this be guaranteed

tc_to_txq

XXX: need comments on this one

xps_maps

XXX: need comments on this one

nf_hooks_egress

netfilter hooks executed for egress packets

tcx_egress

BPF & clsact qdisc specific data for egress processing

{unnamed_union}

anonymous

lstats

Loopback statistics: packets, bytes

tstats

Tunnel statistics: RX/TX packets, RX/TX bytes

dstats

Dummy statistics: RX/TX/drop packets, RX/TX bytes

flags

Interface flags (a la BSD)

hard_header_len

Maximum hardware header length.

features

Currently active device features

ip6_ptr

IPv6 specific data

xdp_prog

XDP sockets filter program pointer

ptype_specific

Device-specific, protocol-specific packet handlers

ifindex

interface index

real_num_rx_queues

Number of RX queues currently active in device

-rx
Array of RX queues

gro_flush_timeout
timeout for GRO layer in NAPI

napi_defer_hard_irqs
If not zero, provides a counter that would allow to avoid NIC hard IRQ, on busy queues.

gro_max_size
Maximum size of aggregated packet in generic receive offload (GRO)

gro_ipv4_max_size
Maximum size of aggregated packet in generic receive offload (GRO), for IPv4.

rx_handler
handler for received packets

rx_handler_data
XXX: need comments on this one

nd_net
Network namespace this network device is inside

npinfo
XXX: need comments on this one

tcx_ingress
BPF & clsact qdisc specific data for ingress processing

name
This is the first field of the "visible" part of this structure (i.e. as seen by users in the "Space.c" file). It is the name of the interface.

name_node
Name hashlist node

ifalias
SNMP alias

mem_end
Shared memory end

mem_start
Shared memory start

base_addr
Device I/O address

state
Generic network queuing layer state, see netdev_state_t

dev_list
The global list of network devices

napi_list
List entry used for polling NAPI devices

unreg_list
List entry when we are unregistering the device; see the function unregister_netdev

close_list

List entry used when we are closing the device

ptype_all

Device-specific packet handlers for all protocols

adj_list

Directly linked devices, like slaves for bonding

xdp_features

XDP capability supported by the device

xdp_metadata_ops

Includes pointers to XDP metadata callbacks.

xsk_tx_metadata_ops

Includes pointers to AF_XDP TX metadata callbacks.

gflags

Global flags (kept as legacy)

needed_tailroom

Extra tailroom the hardware may need, but not in all cases can this be guaranteed. Some cases also use LL_MAX_HEADER instead to allocate the skb

interface address info:

hw_features

User-changeable features

wanted_features

User-requested features

vlan_features

Mask of features inheritable by VLAN devices

hw_enc_features

Mask of features inherited by encapsulating devices This field indicates what encapsulation offloads the hardware is capable of doing, and drivers will need to set them appropriately.

mpls_features

Mask of features inheritable by MPLS

min_mtu

Interface Minimum MTU value

max_mtu

Interface Maximum MTU value

type

Interface hardware type

min_header_len

Minimum hardware header length

name_assign_type

network interface name assignment type

group

The group the device belongs to

stats

Statistics struct, which was left as a legacy, use rtnl_link_stats64 instead

core_stats

core networking counters, do not use this in drivers

carrier_up_count

Number of times the carrier has been up

carrier_down_count

Number of times the carrier has been down

wireless_handlers

List of functions to handle Wireless Extensions, instead of ioctl, see <net/iw_handler.h> for details.

wireless_data

Instance data managed by the core of wireless extensions

ethtool_ops

Management operations

l3mdev_ops

Layer 3 master device operations

ndisc_ops

Includes callbacks for different IPv6 neighbour discovery handling. Necessary for e.g. 6LoWPAN.

xfrmdev_ops

Transformation offload operations

tlsdev_ops

Transport Layer Security offload operations

operstate

RFC2863 operstate

link_mode

Mapping policy to operstate

if_port

Selectable AUI, TP, ...

dma

DMA channel

perm_addr

Permanent hw address

addr_assign_type

Hw address assignment type

addr_len

Hardware address length

upper_level

Maximum depth level of upper devices.

lower_level

Maximum depth level of lower devices.

neigh_priv_len

Used in neigh_alloc()

dev_id

Used to differentiate devices that share the same link layer address

dev_port

Used to differentiate devices that share the same function

padded

How much padding added by alloc_netdev()

addr_list_lock

XXX: need comments on this one

irq

Device IRQ number

uc

unicast mac addresses

mc

multicast mac addresses

dev_addrs

list of device hw addresses

queues_kset

Group of all Kobjects in the Tx and RX queues

unlink_list

As netif_addr_lock() can be called recursively, keep a list of interfaces to be deleted.

promiscuity

Number of times the NIC is told to work in promiscuous mode; if it becomes 0 the NIC will exit promiscuous mode

allmulti

Counter, enables or disables allmulticast mode

uc_promisc

Counter that indicates promiscuous mode has been enabled due to the need to listen to additional unicast addresses in a device that does not implement ndo_set_rx_mode()

nested_level

Used as a parameter of spin_lock_nested() of dev->addr_list_lock.

ip_ptr

IPv4 specific data

vlan_info

VLAN info

dsa_ptr

dsa specific data

tipc_ptr

TIPC specific data

atalk_ptr

AppleTalk link

ax25_ptr

AX.25 specific data

ieee80211_ptr

IEEE 802.11 specific data, assign before registering

ieee802154_ptr

IEEE 802.15.4 low-rate Wireless Personal Area Network device struct

mpls_ptr

mpls_dev struct pointer

mctp_ptr

MCTP specific data

dev_addr

Hw address (before bcast, because most packets are unicast)

num_rx_queues

Number of RX queues allocated at *register_netdev()* time

xdp_zc_max_segs

Maximum number of segments supported by AF_XDP zero copy driver

ingress_queue

XXX: need comments on this one

nf_hooks_ingress

netfilter hooks executed for ingress packets

broadcast

hw bcast address

rx_cpu_rmap

CPU reverse-mapping for RX completion interrupts, indexed by RX queue number. Assigned by driver. This must only be set if the ndo_rx_flow_steer operation is defined

index_hlist

Device index hash chain

num_tx_queues

Number of TX queues allocated at alloc_netdev_mq() time

qdisc

Root qdisc from userspace point of view

tx_queue_len

Max frames per queue allowed

tx_global_lock

XXX: need comments on this one

xdp_bulkq

XDP device bulk queue

qdisc_hash

qdisc hash table

watchdog_timer

List of timers

watchdog_timeo

Represents the timeout that is used by the watchdog (see dev_watchdog())

proto_down_reason

reason a netdev interface is held down

todo_list

Delayed register/unregister

pcpu_refcnt

Number of references to this device

dev_refcnt

Number of references to this device

refcnt_tracker

Tracker directory for tracked references to this device

link_watch_list

XXX: need comments on this one

reg_state

Register/unregister state machine

dismantle

Device is going to be freed

rtnl_link_state

This enum represents the phases of creating a new link

needs_free_netdev

Should unregister perform free_netdev?

priv_destructor

Called from unregister

ml_priv

Mid-layer private

ml_priv_type

Mid-layer private type

pcpu_stat_type

Type of device statistics which the core should allocate/free: none, lstats, tstats, dstats.
none means the driver is handling statistics allocation/ freeing internally.

garp_port

GARP

mrp_port

MRP

dm_private

Drop monitor private

dev

Class/net/name entry

sysfs_groups

Space for optional device, statistics and wireless sysfs groups

sysfs_rx_queue_group

Space for optional per-rx queue attributes

rtnl_link_ops

Rtnl_link_ops

tso_max_size

Device (as in HW) limit on the max TSO request size

tso_max_segs

Device (as in HW) limit on the max TSO segment count

dcbnl_ops

Data Center Bridging netlink ops

prio_tc_map

XXX: need comments on this one

fcoe_ddp_xid

Max exchange id for FCoE LRO by ddp

priomap

XXX: need comments on this one

phydev

Physical device may attach itself for hardware timestamping

sfp_bus

attached *struct sfp_bus* structure.

qdisc_tx_busylock

lockdep class annotating Qdisc->busylock spinlock

proto_down

protocol port state information can be sent to the switch driver and used to set the phys state of the switch port.

wol_enabled

Wake-on-LAN is enabled

threaded

napi threaded mode is enabled

net_notifier_list

List of per-net netdev notifier block that follow this device when it is moved to another network namespace.

macsec_ops

MACsec offloading ops

udp_tunnel_nic_info

static structure describing the UDP tunnel offload capabilities of the device

udp_tunnel_nic

UDP tunnel offload state

xdp_state

stores info on attached XDP BPF programs

dev_addr_shadow

Copy of **dev_addr** to catch direct writes.

linkwatch_dev_tracker

refcount tracker used by linkwatch.

watchdog_dev_tracker

refcount tracker used by watchdog.

dev_registered_tracker

tracker for reference held while registered

offload_xstats_l3

L3 HW stats for this netdevice.

devlink_port

Pointer to related devlink port structure. Assigned by a driver before netdev registration using SET_NETDEV_DEVLINK_PORT macro. This pointer is static during the time netdevice is registered.

dpll_pin

Pointer to the SyncE source pin of a DPLL subsystem, where the clock is recovered.

FIXME: cleanup *struct net_device* such that network protocol info moves out.

page_pools

page pools created for this netdevice

Description

Actually, this whole structure is a big mistake. It mixes I/O data with strictly "high-level" data, and it has to know about almost every data structure used in the INET module.

void *netdev_priv(const struct net_device *dev)

access network device private data

Parameters

const struct net_device *dev

network device

Description

Get network device private data

void netif_napi_add(struct net_device *dev, struct napi_struct *napi, int (*poll)(struct napi_struct*, int))

initialize a NAPI context

Parameters

struct net_device *dev

network device

struct napi_struct *napi

NAPI context

int (*poll)(struct napi_struct *, int)

polling function

Description

`netif_napi_add()` must be used to initialize a NAPI context prior to calling *any* of the other NAPI-related functions.

```
void netif_napi_add_tx(struct net_device *dev, struct napi_struct *napi, int (*poll)(struct napi_struct*, int))
```

initialize a NAPI context to be used for Tx only

Parameters

struct net_device *dev
network device

struct napi_struct *napi
NAPI context

int (*poll)(struct napi_struct *, int)
polling function

Description

This variant of `netif_napi_add()` should be used from drivers using NAPI to exclusively poll a TX queue. This will avoid we add it into napi_hash[], thus polluting this hash table.

```
void __netif_napi_del(struct napi_struct *napi)
```

remove a NAPI context

Parameters

struct napi_struct *napi
NAPI context

Description

Warning: caller must observe RCU grace period before freeing memory containing **napi**. Drivers might want to call this helper to combine all the needed RCU grace periods into a single one.

```
void netif_napi_del(struct napi_struct *napi)
```

remove a NAPI context

Parameters

struct napi_struct *napi
NAPI context

`netif_napi_del()` removes a NAPI context from the network device NAPI list

```
void netif_start_queue(struct net_device *dev)
```

allow transmit

Parameters

struct net_device *dev
network device

Allow upper layers to call the device hard_start_xmit routine.

```
void netif_wake_queue(struct net_device *dev)
```

restart transmit

Parameters

struct net_device *dev
network device

Allow upper layers to call the device hard_start_xmit routine. Used for flow control when transmit resources are available.

void netif_stop_queue(struct net_device *dev)
stop transmitted packets

Parameters

struct net_device *dev
network device

Stop upper layers calling the device hard_start_xmit routine. Used for flow control when transmit resources are unavailable.

bool netif_queue_stopped(const struct net_device *dev)
test if transmit queue is flowblocked

Parameters

const struct net_device *dev
network device

Test if transmit queue on device is currently unable to send.

void netdev_queue_set_dql_min_limit(struct netdev_queue *dev_queue, unsigned int min_limit)
set dql minimum limit

Parameters

struct netdev_queue *dev_queue
pointer to transmit queue

unsigned int min_limit
dql minimum limit

Description

Forces xmit_more() to return true until the minimum threshold defined by **min_limit** is reached (or until the tx queue is empty). Warning: to be use with care, misuse will impact the latency.

void netdev_txq_bql_enqueue_prefetchw(struct netdev_queue *dev_queue)
prefetch bql data for write

Parameters

struct netdev_queue *dev_queue
pointer to transmit queue

Description

BQL enabled drivers might use this helper in their ndo_start_xmit(), to give appropriate hint to the CPU.

void netdev_txq_bql_complete_prefetchw(struct netdev_queue *dev_queue)
prefetch bql data for write

Parameters

struct netdev_queue *dev_queue
pointer to transmit queue

Description

BQL enabled drivers might use this helper in their TX completion path, to give appropriate hint to the CPU.

```
void netdev_tx_sent_queue(struct netdev_queue *dev_queue, unsigned int bytes)
    report the number of bytes queued to a given tx queue
```

Parameters

struct netdev_queue *dev_queue
network device queue

unsigned int bytes

number of bytes queued to the device queue

Report the number of bytes queued for sending/completion to the network device hardware queue. **bytes** should be a good approximation and should exactly match `netdev_completed_queue()` **bytes**. This is typically called once per packet, from `ndo_start_xmit()`.

void **netdev_sent_queue**(struct *net_device* *dev, unsigned int bytes)
 report the number of bytes queued to hardware

Parameters

struct net_device *dev
network device

unsigned int bytes

number of bytes queued to the hardware device queue

Report the number of bytes queued for sending/completion to the network device hardware queue#0. **bytes** should be a good approximation and should exactly match `netdev_completed_queue()` **bytes**. This is typically called once per packet, from `ndo_start_xmit()`.

```
void netdev_tx_completed_queue(struct netdev_queue *dev_queue, unsigned int pkts,
                               unsigned int bytes)
```

report number of packets/bytes at TX completion.

Parameters

struct netdev_queue *dev_queue
network device queue

unsigned int pkts
number of packets (currently ignored)

unsigned int bytes
number of bytes dequeued from the device queue

Must be called at most once per TX completion round (and not per individual packet), so that ROI can adjust its limits appropriately

void **netdev_completed_queue**(struct *net_device* *dev, unsigned int pkts, unsigned int bytes)
 report bytes and packets completed by device

Parameters

struct net_device *dev
 network device

unsigned int pkts
 actual number of packets sent over the medium

unsigned int bytes
 actual number of bytes sent over the medium

Report the number of bytes and packets transmitted by the network device hardware queue over the physical medium, **bytes** must exactly match the **bytes** amount passed to *netdev_sent_queue()*

void **netdev_reset_queue**(struct *net_device* *dev_queue)
 reset the packets and bytes count of a network device

Parameters

struct net_device *dev_queue
 network device

Reset the bytes and packet count of a network device and clear the software flow control OFF bit for this network device

u16 **netdev_cap_txqueue**(struct *net_device* *dev, u16 queue_index)
 check if selected tx queue exceeds device queues

Parameters

struct net_device *dev
 network device

u16 queue_index
 given tx queue index

Returns 0 if given tx queue index \geq number of device tx queues, otherwise returns the originally passed tx queue index.

bool **netif_running**(const struct *net_device* *dev)
 test if up

Parameters

const struct net_device *dev
 network device

Test if the device has been brought up.

void **netif_start_subqueue**(struct *net_device* *dev, u16 queue_index)
 allow sending packets on subqueue

Parameters

struct net_device *dev
 network device

u16 queue_index
sub queue index

Description

Start individual transmit queue of a device with multiple transmit queues.

void netif_stop_subqueue(struct *net_device* *dev, u16 queue_index)
stop sending packets on subqueue

Parameters

struct net_device *dev
network device

u16 queue_index
sub queue index

Description

Stop individual transmit queue of a device with multiple transmit queues.

bool __netif_subqueue_stopped(const struct *net_device* *dev, u16 queue_index)
test status of subqueue

Parameters

const struct net_device *dev
network device

u16 queue_index
sub queue index

Description

Check individual transmit queue of a device with multiple transmit queues.

bool netif_subqueue_stopped(const struct *net_device* *dev, struct *sk_buff* *skb)
test status of subqueue

Parameters

const struct net_device *dev
network device

struct sk_buff *skb
sub queue buffer pointer

Description

Check individual transmit queue of a device with multiple transmit queues.

void netif_wake_subqueue(struct *net_device* *dev, u16 queue_index)
allow sending packets on subqueue

Parameters

struct net_device *dev
network device

u16 queue_index
sub queue index

Description

Resume individual transmit queue of a device with multiple transmit queues.

`bool netif_attr_test_mask(unsigned long j, const unsigned long *mask, unsigned int nr_bits)`

Test a CPU or Rx queue set in a mask

Parameters

unsigned long j

CPU/Rx queue index

const unsigned long *mask

bitmask of all cpus/rx queues

unsigned int nr_bits

number of bits in the bitmask

Description

Test if a CPU or Rx queue index is set in a mask of all CPU/Rx queues.

`bool netif_attr_test_online(unsigned long j, const unsigned long *online_mask, unsigned int nr_bits)`

Test for online CPU/Rx queue

Parameters

unsigned long j

CPU/Rx queue index

const unsigned long *online_mask

bitmask for CPUs/Rx queues that are online

unsigned int nr_bits

number of bits in the bitmask

Description

Returns true if a CPU/Rx queue is online.

`unsigned int netif_attrmask_next(int n, const unsigned long *srcp, unsigned int nr_bits)`
get the next CPU/Rx queue in a cpu/Rx queues mask

Parameters

int n

CPU/Rx queue index

const unsigned long *srcp

the cpumask/Rx queue mask pointer

unsigned int nr_bits

number of bits in the bitmask

Description

Returns \geq nr_bits if no further CPUs/Rx queues set.

```
int netif_attrmask_next_and(int n, const unsigned long *src1p, const unsigned long *src2p,
                           unsigned int nr_bits)
```

get the next CPU/Rx queue in *src1p & *src2p

Parameters

int n

CPU/Rx queue index

const unsigned long *src1p

the first CPUs/Rx queues mask pointer

const unsigned long *src2p

the second CPUs/Rx queues mask pointer

unsigned int nr_bits

number of bits in the bitmask

Description

Returns >= nr_bits if no further CPUs/Rx queues set in both.

```
bool netif_is_multiqueue(const struct net_device *dev)
```

test if device has multiple transmit queues

Parameters

const struct net_device *dev

network device

Description

Check if device has multiple transmit queues

```
void dev_hold(struct net_device *dev)
```

get reference to device

Parameters

struct net_device *dev

network device

Description

Hold reference to device to keep it from being freed. Try using netdev_hold() instead.

```
void dev_put(struct net_device *dev)
```

release reference to device

Parameters

struct net_device *dev

network device

Description

Release reference to device to allow it to be freed. Try using netdev_put() instead.

```
void linkwatch_sync_dev(struct net_device *dev)
```

sync linkwatch for the given device

Parameters

```
struct net_device *dev
    network device to sync linkwatch for
```

Description

Sync linkwatch for the given device, removing it from the pending work list (if queued).

```
bool netif_carrier_ok(const struct net_device *dev)
    test if carrier present
```

Parameters

```
const struct net_device *dev
    network device
```

Description

Check if carrier is present on device

```
void netif_dormant_on(struct net_device *dev)
    mark device as dormant.
```

Parameters

```
struct net_device *dev
    network device
```

Description

Mark device as dormant (as per RFC2863).

The dormant state indicates that the relevant interface is not actually in a condition to pass packets (i.e., it is not 'up') but is in a "pending" state, waiting for some external event. For "on-demand" interfaces, this new state identifies the situation where the interface is waiting for events to place it in the up state.

```
void netif_dormant_off(struct net_device *dev)
    set device as not dormant.
```

Parameters

```
struct net_device *dev
    network device
```

Description

Device is not in dormant state.

```
bool netif_dormant(const struct net_device *dev)
    test if device is dormant
```

Parameters

```
const struct net_device *dev
    network device
```

Description

Check if device is dormant.

```
void netif_testing_on(struct net_device *dev)
    mark device as under test.
```

Parameters

```
struct net_device *dev
    network device
```

Description

Mark device as under test (as per RFC2863).

The testing state indicates that some test(s) must be performed on the interface. After completion, of the test, the interface state will change to up, dormant, or down, as appropriate.

```
void netif_testing_off(struct net_device *dev)
    set device as not under test.
```

Parameters

```
struct net_device *dev
    network device
```

Description

Device is not in testing state.

```
bool netif_testing(const struct net_device *dev)
    test if device is under test
```

Parameters

```
const struct net_device *dev
    network device
```

Description

Check if device is under test

```
bool netif_oper_up(const struct net_device *dev)
    test if device is operational
```

Parameters

```
const struct net_device *dev
    network device
```

Description

Check if carrier is operational

```
bool netif_device_present(const struct net_device *dev)
    is device available or removed
```

Parameters

```
const struct net_device *dev
    network device
```

Description

Check if device has not been removed from system.

```
void netif_tx_lock(struct net_device *dev)
    grab network device transmit lock
```

Parameters

```
struct net_device *dev
    network device
```

Description

Get network device transmit lock

```
int __dev_uc_sync(struct net_device *dev, int (*sync)(struct net_device*, const unsigned
    char*), int (*unsync)(struct net_device*, const unsigned char*))
```

Synchonize device's unicast list

Parameters

```
struct net_device *dev
    device to sync
```

```
int (*sync)(struct net_device *, const unsigned char *)
    function to call if address should be added
```

```
int (*unsync)(struct net_device *, const unsigned char *)
    function to call if address should be removed
```

Add newly added addresses to the interface, and release addresses that have been deleted.

```
void __dev_uc_unsync(struct net_device *dev, int (*unsync)(struct net_device*, const
    unsigned char*))
```

Remove synchronized addresses from device

Parameters

```
struct net_device *dev
    device to sync
```

```
int (*unsync)(struct net_device *, const unsigned char *)
    function to call if address should be removed
```

Remove all addresses that were added to the device by dev_uc_sync().

```
int __dev_mc_sync(struct net_device *dev, int (*sync)(struct net_device*, const unsigned
    char*), int (*unsync)(struct net_device*, const unsigned char*))
```

Synchonize device's multicast list

Parameters

```
struct net_device *dev
    device to sync
```

```
int (*sync)(struct net_device *, const unsigned char *)
    function to call if address should be added
```

```
int (*unsync)(struct net_device *, const unsigned char *)
    function to call if address should be removed
```

Add newly added addresses to the interface, and release addresses that have been deleted.

```
void __dev_mc_unsync(struct net_device *dev, int (*unsync)(struct net_device*, const
                           unsigned char*))
```

Remove synchronized addresses from device

Parameters

struct net_device *dev

device to sync

int (*unsync)(struct net_device *, const unsigned char *)

function to call if address should be removed

Remove all addresses that were added to the device by dev_mc_sync().

13.2.2 PHY Support

```
void phy_print_status(struct phy_device *phydev)
```

Convenience function to print out the current phy status

Parameters

struct phy_device *phydev

the phy_device struct

int phy_get_rate_matching(struct phy_device *phydev, phy_interface_t iface)

determine if rate matching is supported

Parameters

struct phy_device *phydev

The phy device to return rate matching for

phy_interface_t iface

The interface mode to use

Description

This determines the type of rate matching (if any) that **phy** supports using **iface**. **iface** may be **PHY_INTERFACE_MODE_NA** to determine if any interface supports rate matching.

Return

The type of rate matching phy supports for iface, or

RATE_MATCH_NONE.

```
int phy_restart_aneg(struct phy_device *phydev)
```

restart auto-negotiation

Parameters

struct phy_device *phydev

target phy_device struct

Description

Restart the autonegotiation on **phydev**. Returns ≥ 0 on success or negative errno on error.

```
int phy_aneg_done(struct phy_device *phydev)
    return auto-negotiation status
```

Parameters

struct phy_device *phydev
target phy_device struct

Description

Return the auto-negotiation status from this **phydev**. Returns > 0 on success or < 0 on error. 0 means that auto-negotiation is still pending.

```
bool phy_check_valid(int speed, int duplex, unsigned long *features)
    check if there is a valid PHY setting which matches speed, duplex, and feature mask
```

Parameters

int speed
speed to match

int duplex
duplex to match

unsigned long *features
A mask of the valid settings

Description

Returns true if there is a valid setting, false otherwise.

```
int phy_mii_ioctl(struct phy_device *phydev, struct ifreq *ifr, int cmd)
    generic PHY MII ioctl interface
```

Parameters

struct phy_device *phydev
the phy_device struct

struct ifreq *ifr
struct ifreq for socket ioctl's

int cmd
ioctl cmd to execute

Description

Note that this function is currently incompatible with the PHYCONTROL layer. It changes registers without regard to current state. Use at own risk.

```
int phy_do_ioctl(struct net_device *dev, struct ifreq *ifr, int cmd)
    generic ndo_eth_ioctl implementation
```

Parameters

struct net_device *dev
the net_device struct

struct ifreq *ifr
struct ifreq for socket ioctl's

```
int cmd
    ioctl cmd to execute

int phy_do_ioctl_running(struct net_device *dev, struct ifreq *ifr, int cmd)
    generic ndo_eth_ioctl implementation but test first
```

Parameters

```
struct net_device *dev
    the net_device struct

struct ifreq *ifr
    struct ifreq for socket ioctl's
```

```
int cmd
    ioctl cmd to execute
```

Description

Same as phy_do_ioctl, but ensures that net_device is running before handling the ioctl.

```
void phy_queue_state_machine(struct phy_device *phydev, unsigned long jiffies)
    Trigger the state machine to run soon
```

Parameters

```
struct phy_device *phydev
    the phy_device struct

unsigned long jiffies
    Run the state machine after these jiffies
```

```
void phy_trigger_machine(struct phy_device *phydev)
    Trigger the state machine to run now
```

Parameters

```
struct phy_device *phydev
    the phy_device struct

int phy_ethtool_get_strings(struct phy_device *phydev, u8 *data)
    Get the statistic counter names
```

Parameters

```
struct phy_device *phydev
    the phy_device struct

u8 *data
    Where to put the strings

int phy_ethtool_get_sset_count(struct phy_device *phydev)
    Get the number of statistic counters
```

Parameters

```
struct phy_device *phydev
    the phy_device struct
```

```
int phy_ethtool_get_stats(struct phy_device *phydev, struct ethtool_stats *stats, u64 *data)
    Get the statistic counters
```

Parameters

```
struct phy_device *phydev
    the phy_device struct
```

```
struct ethtool_stats *stats
    What counters to get
```

```
u64 *data
    Where to store the counters
```

```
int phy_start_cable_test(struct phy_device *phydev, struct netlink_ext_ack *extack)
    Start a cable test
```

Parameters

```
struct phy_device *phydev
    the phy_device struct
```

```
struct netlink_ext_ack *extack
    extack for reporting useful error messages
```

```
int phy_start_cable_test_tdr(struct phy_device *phydev, struct netlink_ext_ack *extack,
                             const struct phy_tdr_config *config)
```

Start a raw TDR cable test

Parameters

```
struct phy_device *phydev
    the phy_device struct
```

```
struct netlink_ext_ack *extack
    extack for reporting useful error messages
```

```
const struct phy_tdr_config *config
    Configuration of the test to run
```

```
int _phy_start_aneg(struct phy_device *phydev)
    start auto-negotiation for this PHY device
```

Parameters

```
struct phy_device *phydev
    the phy_device struct
```

Description**Sanitizes the settings (if we're not autonegotiating)**

them), and then calls the driver's config_aneg function. If the PHYCONTROL Layer is operating, we change the state to reflect the beginning of Auto-negotiation or forcing.

```
int phy_start_aneg(struct phy_device *phydev)
    start auto-negotiation for this PHY device
```

Parameters

```
struct phy_device *phydev
    the phy_device struct
```

Description

Sanitizes the settings (if we're not autonegotiating)

them), and then calls the driver's config_aneg function. If the PHYCONTROL Layer is operating, we change the state to reflect the beginning of Auto-negotiation or forcing.

```
int phy_speed_down(struct phy_device *phydev, bool sync)
    set speed to lowest speed supported by both link partners
```

Parameters

struct phy_device *phydev
the phy_device struct

bool sync
perform action synchronously

Description

Typically used to save energy when waiting for a WoL packet

WARNING: Setting sync to false may cause the system being unable to suspend in case the PHY generates an interrupt when finishing the autonegotiation. This interrupt may wake up the system immediately after suspend. Therefore use sync = false only if you're sure it's safe with the respective network chip.

```
int phy_speed_up(struct phy_device *phydev)
    (re)set advertised speeds to all supported speeds
```

Parameters

struct phy_device *phydev
the phy_device struct

Description

Used to revert the effect of phy_speed_down

```
void phy_start_machine(struct phy_device *phydev)
    start PHY state machine tracking
```

Parameters

struct phy_device *phydev
the phy_device struct

Description

The PHY infrastructure can run a state machine

which tracks whether the PHY is starting up, negotiating, etc. This function starts the delayed workqueue which tracks the state of the PHY. If you want to maintain your own state machine, do not call this function.

```
void phy_error(struct phy_device *phydev)
    enter ERROR state for this PHY device
```

Parameters

struct phy_device *phydev
target phy_device struct

Description

Moves the PHY to the ERROR state in response to a read or write error, and tells the controller the link is down. Must be called with phydev->lock held.

```
void phy_request_interrupt(struct phy_device *phydev)
    request and enable interrupt for a PHY device
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

Description

Request and enable the interrupt for the given PHY.

If this fails, then we set irq to PHY_POLL. This should only be called with a valid IRQ number.

```
void phy_free_interrupt(struct phy_device *phydev)
    disable and free interrupt for a PHY device
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

Description

Disable and free the interrupt for the given PHY.

This should only be called with a valid IRQ number.

```
void phy_stop(struct phy_device *phydev)
    Bring down the PHY link, and stop checking the status
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

```
void phy_start(struct phy_device *phydev)
    start or restart a PHY device
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

Description

Indicates the attached device's readiness to

handle PHY-related work. Used during startup to start the PHY, and after a call to `phy_stop()` to resume operation. Also used to indicate the MDIO bus has cleared an error condition.

```
void phy_mac_interrupt(struct phy_device *phydev)
    MAC says the link has changed
```

Parameters

```
struct phy_device *phydev
    phy_device struct with changed link
```

Description

The MAC layer is able to indicate there has been a change in the PHY link status. Trigger the state machine and work a work queue.

```
int phy_init_eee(struct phy_device *phydev, bool clk_stop_enable)
    init and check the EEE feature
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

```
bool clk_stop_enable
    PHY may stop the clock during LPI
```

Description

it checks if the Energy-Efficient Ethernet (EEE) is supported by looking at the MMD registers 3.20 and 7.60/61 and it programs the MMD register 3.0 setting the "Clock stop enable" bit if required.

```
int phy_get_eee_err(struct phy_device *phydev)
    report the EEE wake error count
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

Description

it is to report the number of time where the PHY failed to complete its normal wake sequence.

```
int phy_ethtool_get_eee(struct phy_device *phydev, struct ethtool_eee *data)
    get EEE supported and status
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

```
struct ethtool_eee *data
    ethtool_eee data
```

Description

it reports the Supported/Advertisement/LP Advertisement capabilities.

```
int phy_ethtool_set_eee(struct phy_device *phydev, struct ethtool_eee *data)
    set EEE supported and status
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

```
struct ethtool_eee *data
    ethtool_eee data
```

Description

it is to program the Advertisement EEE register.

```
int phy_ethtool_set_wol(struct phy_device *phydev, struct ethtool_wolinfo *wol)
```

Configure Wake On LAN

Parameters

```
struct phy_device *phydev
```

target phy_device struct

```
struct ethtool_wolinfo *wol
```

Configuration requested

```
void phy_ethtool_get_wol(struct phy_device *phydev, struct ethtool_wolinfo *wol)
```

Get the current Wake On LAN configuration

Parameters

```
struct phy_device *phydev
```

target phy_device struct

```
struct ethtool_wolinfo *wol
```

Store the current configuration here

```
int phy_ethtool_nway_reset(struct net_device *ndev)
```

Restart auto negotiation

Parameters

```
struct net_device *ndev
```

Network device to restart autoneg for

```
int phy_config_interrupt(struct phy_device *phydev, bool interrupts)
```

configure the PHY device for the requested interrupts

Parameters

```
struct phy_device *phydev
```

the phy_device struct

```
bool interrupts
```

interrupt flags to configure for this **phydev**

Description

Returns 0 on success or < 0 on error.

```
const struct phy_setting *phy_find_valid(int speed, int duplex, unsigned long *supported)
```

find a PHY setting that matches the requested parameters

Parameters

```
int speed
```

desired speed

```
int duplex
```

desired duplex

```
unsigned long *supported
```

mask of supported link modes

Description

Locate a supported phy setting that is, in priority order: - an exact match for the specified speed and duplex mode - a match for the specified speed, or slower speed - the slowest supported speed Returns the matched phy_setting entry, or NULL if no supported phy settings were found.

```
unsigned int phy_supported_speeds(struct phy_device *phy, unsigned int *speeds, unsigned int size)
```

return all speeds currently supported by a phy device

Parameters

struct phy_device *phy

The phy device to return supported speeds of.

unsigned int *speeds

buffer to store supported speeds in.

unsigned int size

size of speeds buffer.

Description

Returns the number of supported speeds, and fills the speeds buffer with the supported speeds. If speeds buffer is too small to contain all currently supported speeds, will return as many speeds as can fit.

```
void phy_sanitize_settings(struct phy_device *phydev)
```

make sure the PHY is set to supported speed and duplex

Parameters

struct phy_device *phydev

the target phy_device struct

Description

Make sure the PHY is set to supported speeds and

duplexes. Drop down by one in this order: 1000/FULL, 1000/HALF, 100/FULL, 100/HALF, 10/FULL, 10/HALF.

```
int _phy_hwtstamp_get(struct phy_device *phydev, struct kernel_hwtstamp_config *config)
```

Get hardware timestamping configuration from PHY

Parameters

struct phy_device *phydev

the PHY device structure

struct kernel_hwtstamp_config *config

structure holding the timestamping configuration

Description

Query the PHY device for its current hardware timestamping configuration.

```
int _phy_hwtstamp_set(struct phy_device *phydev, struct kernel_hwtstamp_config *config,  
                          struct netlink_ext_ack *extack)
```

Modify PHY hardware timestamping configuration

Parameters

```
struct phy_device *phydev
    the PHY device structure

struct kernel_hwtstamp_config *config
    structure holding the timestamping configuration

struct netlink_ext_ack *extack
    netlink extended ack structure, for error reporting

int phy_ethtool_get_plca_cfg(struct phy_device *phydev, struct phy_plca_cfg *plca_cfg)
    Get PLCA RS configuration
```

Parameters

```
struct phy_device *phydev
    the phy_device struct

struct phy_plca_cfg *plca_cfg
    where to store the retrieved configuration
```

Description

Retrieve the PLCA configuration from the PHY. Return 0 on success or a negative value if an error occurred.

```
int plca_check_valid(struct phy_device *phydev, const struct phy_plca_cfg *plca_cfg, struct
                      netlink_ext_ack *extack)
    Check PLCA configuration before enabling
```

Parameters

```
struct phy_device *phydev
    the phy_device struct

const struct phy_plca_cfg *plca_cfg
    current PLCA configuration

struct netlink_ext_ack *extack
    extack for reporting useful error messages
```

Description

Checks whether the PLCA and PHY configuration are consistent and it is safe to enable PLCA. Returns 0 on success or a negative value if the PLCA or PHY configuration is not consistent.

```
int phy_ethtool_set_plca_cfg(struct phy_device *phydev, const struct phy_plca_cfg
                               *plca_cfg, struct netlink_ext_ack *extack)
    Set PLCA RS configuration
```

Parameters

```
struct phy_device *phydev
    the phy_device struct

const struct phy_plca_cfg *plca_cfg
    new PLCA configuration to apply

struct netlink_ext_ack *extack
    extack for reporting useful error messages
```

Description

Sets the PLCA configuration in the PHY. Return 0 on success or a negative value if an error occurred.

```
int phy_ethtool_get_plca_status(struct phy_device *phydev, struct phy_plca_status
                                *plca_st)
```

Get PLCA RS status information

Parameters

struct phy_device *phydev
the phy_device struct

struct phy_plca_status *plca_st
where to store the retrieved status information

Description

Retrieve the PLCA status information from the PHY. Return 0 on success or a negative value if an error occurred.

```
int phy_check_link_status(struct phy_device *phydev)
                           check link status and set state accordingly
```

Parameters

struct phy_device *phydev
the phy_device struct

Description

Check for link and whether autoneg was triggered / is running and set state accordingly

```
void phy_stop_machine(struct phy_device *phydev)
                      stop the PHY state machine tracking
```

Parameters

struct phy_device *phydev
target phy_device struct

Description

Stops the state machine delayed workqueue, sets the

state to UP (unless it wasn't up yet). This function must be called BEFORE phy_detach.

```
int phy_disable_interrupts(struct phy_device *phydev)
```

Disable the PHY interrupts from the PHY side

Parameters

struct phy_device *phydev
target phy_device struct

irqreturn_t **phy_interrupt**(int irq, void *phy_dat)
PHY interrupt handler

Parameters

int irq
interrupt line

void *phy_dat
 phy_device pointer

Description

Handle PHY interrupt

int phy_enable_interrupts(struct phy_device *phydev)
 Enable the interrupts from the PHY side

Parameters

struct phy_device *phydev
 target phy_device struct

void phy_state_machine(struct work_struct *work)
 Handle the state machine

Parameters

struct work_struct *work
 work_struct that describes the work to be done

const char *phy_speed_to_str(int speed)
 Return a string representing the PHY link speed

Parameters

int speed
 Speed of the link

const char *phy_duplex_to_str(unsigned int duplex)
 Return string describing the duplex

Parameters

unsigned int duplex
 Duplex setting to describe

const char *phy_rate_matching_to_str(int rate_matching)
 Return a string describing the rate matching

Parameters

int rate_matching
 Type of rate matching to describe

int phy_interface_num_ports(phy_interface_t interface)

 Return the number of links that can be carried by a given MAC-PHY physical link. Returns 0 if this is unknown, the number of links else.

Parameters

phy_interface_t interface
 The interface mode we want to get the number of ports

const struct phy_setting *phy_lookup_setting(int speed, int duplex, const unsigned long *mask, bool exact)
 lookup a PHY setting

Parameters

```
int speed
    speed to match

int duplex
    duplex to match

const unsigned long *mask
    allowed link modes

bool exact
    an exact match is required
```

Description

Search the settings array for a setting that matches the speed and duplex, and which is supported.

If **exact** is unset, either an exact match or NULL for no match will be returned.

If **exact** is set, an exact match, the fastest supported setting at or below the specified speed, the slowest supported setting, or if they all fail, NULL will be returned.

```
void phy_set_max_speed(struct phy_device *phydev, u32 max_speed)
    Set the maximum speed the PHY should support
```

Parameters

```
struct phy_device *phydev
    The phy_device struct

u32 max_speed
    Maximum speed
```

Description

The PHY might be more capable than the MAC. For example a Fast Ethernet is connected to a 1G PHY. This function allows the MAC to indicate its maximum speed, and so limit what the PHY will advertise.

```
void phy_resolve_aneg_pause(struct phy_device *phydev)
    Determine pause autoneg results
```

Parameters

```
struct phy_device *phydev
    The phy_device struct
```

Description

Once autoneg has completed the local pause settings can be resolved. Determine if pause and asymmetric pause should be used by the MAC.

```
void phy_resolve_aneg_linkmode(struct phy_device *phydev)
    resolve the advertisements into PHY settings
```

Parameters

```
struct phy_device *phydev
    The phy_device struct
```

Description

Resolve our and the link partner advertisements into their corresponding speed and duplex. If full duplex was negotiated, extract the pause mode from the link partner mask.

```
void phy_check_downshift(struct phy_device *phydev)
    check whether downshift occurred
```

Parameters

struct phy_device *phydev
The phy_device struct

Description

Check whether a downshift to a lower speed occurred. If this should be the case warn the user. Prerequisite for detecting downshift is that PHY driver implements the read_status callback and sets phydev->speed to the actual link speed.

```
int __phy_read_mmd(struct phy_device *phydev, int devad, u32 regnum)
    Convenience function for reading a register from an MMD on a given PHY.
```

Parameters

struct phy_device *phydev
The phy_device struct

int devad
The MMD to read from (0..31)

u32 regnum
The register on the MMD to read (0..65535)

Description

Same rules as for [_phy_read\(\)](#);

```
int phy_read_mmd(struct phy_device *phydev, int devad, u32 regnum)
    Convenience function for reading a register from an MMD on a given PHY.
```

Parameters

struct phy_device *phydev
The phy_device struct

int devad
The MMD to read from

u32 regnum
The register on the MMD to read

Description

Same rules as for [phy_read\(\)](#);

```
int __phy_write_mmd(struct phy_device *phydev, int devad, u32 regnum, u16 val)
    Convenience function for writing a register on an MMD on a given PHY.
```

Parameters

struct phy_device *phydev
The phy_device struct

int devad

The MMD to read from

u32 regnum

The register on the MMD to read

u16 val

value to write to **regnum**

Description

Same rules as for [phy_write\(\)](#);

```
int phy_write_mmd(struct phy_device *phydev, int devad, u32 regnum, u16 val)
```

Convenience function for writing a register on an MMD on a given PHY.

Parameters**struct phy_device *phydev**

The phy_device struct

int devad

The MMD to read from

u32 regnum

The register on the MMD to read

u16 val

value to write to **regnum**

Description

Same rules as for [phy_write\(\)](#);

```
int __phy_package_read_mmd(struct phy_device *phydev, unsigned int addr_offset, int devad,
                           u32 regnum)
```

read MMD reg relative to PHY package base addr

Parameters**struct phy_device *phydev**

The phy_device struct

unsigned int addr_offset

The offset to be added to PHY package base_addr

int devad

The MMD to read from

u32 regnum

The register on the MMD to read

Description

Convenience helper for reading a register of an MMD on a given PHY using the PHY package base address. The base address is added to the addr_offset value.

Same calling rules as for [phy_read\(\)](#);

NOTE

It's assumed that the entire PHY package is either C22 or C45.

```
int phy_package_read_mmd(struct phy_device *phydev, unsigned int addr_offset, int devad,
                           u32 regnum)
```

read MMD reg relative to PHY package base addr

Parameters

struct phy_device *phydev

The phy_device struct

unsigned int addr_offset

The offset to be added to PHY package base_addr

int devad

The MMD to read from

u32 regnum

The register on the MMD to read

Description

Convenience helper for reading a register of an MMD on a given PHY using the PHY package base address. The base address is added to the addr_offset value.

Same calling rules as for [*phy_read\(\)*](#);

NOTE

It's assumed that the entire PHY package is either C22 or C45.

```
int __phy_package_write_mmd(struct phy_device *phydev, unsigned int addr_offset, int devad,
                             u32 regnum, u16 val)
```

write MMD reg relative to PHY package base addr

Parameters

struct phy_device *phydev

The phy_device struct

unsigned int addr_offset

The offset to be added to PHY package base_addr

int devad

The MMD to write to

u32 regnum

The register on the MMD to write

u16 val

value to write to **regnum**

Description

Convenience helper for writing a register of an MMD on a given PHY using the PHY package base address. The base address is added to the addr_offset value.

Same calling rules as for [*__phy_write\(\)*](#);

NOTE

It's assumed that the entire PHY package is either C22 or C45.

```
int phy_package_write_mmd(struct phy_device *phydev, unsigned int addr_offset, int devad,
                           u32 regnum, u16 val)
```

write MMD reg relative to PHY package base addr

Parameters

struct phy_device *phydev

The phy_device struct

unsigned int addr_offset

The offset to be added to PHY package base_addr

int devad

The MMD to write to

u32 regnum

The register on the MMD to write

u16 val

value to write to **regnum**

Description

Convenience helper for writing a register of an MMD on a given PHY using the PHY package base address. The base address is added to the addr_offset value.

Same calling rules as for [*phy_write\(\)*](#);

NOTE

It's assumed that the entire PHY package is either C22 or C45.

```
int phy_modify_changed(struct phy_device *phydev, u32 regnum, u16 mask, u16 set)
```

Function for modifying a PHY register

Parameters

struct phy_device *phydev

the phy_device struct

u32 regnum

register number to modify

u16 mask

bit mask of bits to clear

u16 set

new value of bits set in mask to write to **regnum**

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

Description

Returns negative errno, 0 if there was no change, and 1 in case of change

```
int __phy_modify(struct phy_device *phydev, u32 regnum, u16 mask, u16 set)
```

Convenience function for modifying a PHY register

Parameters

```
struct phy_device *phydev
    the phy_device struct

u32 regnum
    register number to modify

u16 mask
    bit mask of bits to clear

u16 set
    new value of bits set in mask to write to regnum
```

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

```
int phy_modify(struct phy_device *phydev, u32 regnum, u16 mask, u16 set)
    Convenience function for modifying a given PHY register
```

Parameters

```
struct phy_device *phydev
    the phy_device struct

u32 regnum
    register number to write

u16 mask
    bit mask of bits to clear

u16 set
    new value of bits set in mask to write to regnum
```

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

```
int __phy_modify_mmd_changed(struct phy_device *phydev, int devad, u32 regnum, u16 mask,
                             u16 set)
```

Function for modifying a register on MMD

Parameters

```
struct phy_device *phydev
    the phy_device struct

int devad
    the MMD containing register to modify

u32 regnum
    register number to modify

u16 mask
    bit mask of bits to clear

u16 set
    new value of bits set in mask to write to regnum
```

Description

Unlocked helper function which allows a MMD register to be modified as new register value = (old register value & ~mask) | set

Returns negative errno, 0 if there was no change, and 1 in case of change

```
int phy_modify_mmd_changed(struct phy_device *phydev, int devad, u32 regnum, u16 mask,
                           u16 set)
```

Function for modifying a register on MMD

Parameters

struct phy_device *phydev

the phy_device struct

int devad

the MMD containing register to modify

u32 regnum

register number to modify

u16 mask

bit mask of bits to clear

u16 set

new value of bits set in mask to write to **regnum**

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

Description

Returns negative errno, 0 if there was no change, and 1 in case of change

```
int __phy_modify_mmd(struct phy_device *phydev, int devad, u32 regnum, u16 mask, u16 set)
```

Convenience function for modifying a register on MMD

Parameters

struct phy_device *phydev

the phy_device struct

int devad

the MMD containing register to modify

u32 regnum

register number to modify

u16 mask

bit mask of bits to clear

u16 set

new value of bits set in mask to write to **regnum**

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

```
int phy_modify_mmd(struct phy_device *phydev, int devad, u32 regnum, u16 mask, u16 set)
    Convenience function for modifying a register on MMD
```

Parameters

struct phy_device *phydev
the phy_device struct

int devad
the MMD containing register to modify

u32 regnum
register number to modify

u16 mask
bit mask of bits to clear

u16 set
new value of bits set in mask to write to **regnum**

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

```
int phy_save_page(struct phy_device *phydev)
    take the bus lock and save the current page
```

Parameters

struct phy_device *phydev
a pointer to a *struct phy_device*

Description

Take the MDIO bus lock, and return the current page number. On error, returns a negative errno. *phy_restore_page()* must always be called after this, irrespective of success or failure of this call.

```
int phy_select_page(struct phy_device *phydev, int page)
    take the bus lock, save the current page, and set a page
```

Parameters

struct phy_device *phydev
a pointer to a *struct phy_device*

int page
desired page

Description

Take the MDIO bus lock to protect against concurrent access, save the current PHY page, and set the current page. On error, returns a negative errno, otherwise returns the previous page number. *phy_restore_page()* must always be called after this, irrespective of success or failure of this call.

```
int phy_restore_page(struct phy_device *phydev, int oldpage, int ret)
    restore the page register and release the bus lock
```

Parameters

```
struct phy_device *phydev
    a pointer to a struct phy_device

int oldpage
    the old page, return value from phy_save_page() or phy_select_page()

int ret
    operation's return code
```

Description

Release the MDIO bus lock, restoring **oldpage** if it is a valid page. This function propagates the earliest error code from the group of operations.

Return

oldpage if it was a negative value, otherwise **ret** if it was a negative errno value, otherwise *phy_write_page()*'s negative value if it were in error, otherwise **ret**.

```
int phy_read_paged(struct phy_device *phydev, int page, u32 regnum)
    Convenience function for reading a paged register
```

Parameters

```
struct phy_device *phydev
    a pointer to a struct phy_device

int page
    the page for the phy

u32 regnum
    register number
```

Description

Same rules as for *phy_read()*.

```
int phy_write_paged(struct phy_device *phydev, int page, u32 regnum, u16 val)
    Convenience function for writing a paged register
```

Parameters

```
struct phy_device *phydev
    a pointer to a struct phy_device

int page
    the page for the phy

u32 regnum
    register number

u16 val
    value to write
```

Description

Same rules as for *phy_write()*.

```
int phy_modify_paged_changed(struct phy_device *phydev, int page, u32 regnum, u16 mask,
                             u16 set)
    Function for modifying a paged register
```

Parameters

struct phy_device *phydev
a pointer to a *struct phy_device*

int page
the page for the phy

u32 regnum
register number

u16 mask
bit mask of bits to clear

u16 set
bit mask of bits to set

Description

Returns negative errno, 0 if there was no change, and 1 in case of change

int phy_modify_paged(struct phy_device *phydev, int page, u32 regnum, u16 mask, u16 set)
Convenience function for modifying a paged register

Parameters

struct phy_device *phydev
a pointer to a *struct phy_device*

int page
the page for the phy

u32 regnum
register number

u16 mask
bit mask of bits to clear

u16 set
bit mask of bits to set

Description

Same rules as for *phy_read()* and *phy_write()*.

int genphy_c45_pma_resume(struct phy_device *phydev)
wakes up the PMA module

Parameters

struct phy_device *phydev
target phy_device struct

int genphy_c45_pma_suspend(struct phy_device *phydev)
susPENDs the PMA module

Parameters

struct phy_device *phydev
target phy_device struct

int genphy_c45_pma_baset1_setup_master_slave(struct *phy_device* *phydev)
configures forced master/slave role of BaseT1 devices.

Parameters

struct phy_device *phydev
target phy_device struct

int genphy_c45_pma_setup_forced(struct *phy_device* *phydev)
configures a forced speed

Parameters

struct phy_device *phydev
target phy_device struct

int genphy_c45_an_config_aneg(struct *phy_device* *phydev)
configure advertisement registers

Parameters

struct phy_device *phydev
target phy_device struct

Description

Configure advertisement registers based on modes set in phydev->advertising

Returns negative errno code on failure, 0 if advertisement didn't change, or 1 if advertised modes changed.

int genphy_c45_an_disable_aneg(struct *phy_device* *phydev)
disable auto-negotiation

Parameters

struct phy_device *phydev
target phy_device struct

Description

Disable auto-negotiation in the Clause 45 PHY. The link parameters are controlled through the PMA/PMD MMD registers.

Returns zero on success, negative errno code on failure.

int genphy_c45_restart_aneg(struct *phy_device* *phydev)
Enable and restart auto-negotiation

Parameters

struct phy_device *phydev
target phy_device struct

Description

This assumes that the auto-negotiation MMD is present.

Enable and restart auto-negotiation.

```
int genphy_c45_check_and_restart_aneg(struct phy_device *phydev, bool restart)
```

Enable and restart auto-negotiation

Parameters

```
struct phy_device *phydev
```

target phy_device struct

```
bool restart
```

whether aneg restart is requested

Description

This assumes that the auto-negotiation MMD is present.

Check, and restart auto-negotiation if needed.

```
int genphy_c45_aneg_done(struct phy_device *phydev)
```

return auto-negotiation complete status

Parameters

```
struct phy_device *phydev
```

target phy_device struct

Description

This assumes that the auto-negotiation MMD is present.

Reads the status register from the auto-negotiation MMD, returning:
- positive if auto-negotiation is complete
- negative errno code on error
- zero otherwise

```
int genphy_c45_read_link(struct phy_device *phydev)
```

read the overall link status from the MMDs

Parameters

```
struct phy_device *phydev
```

target phy_device struct

Description

Read the link status from the specified MMDs, and if they all indicate that the link is up, set phydev->link to 1. If an error is encountered, a negative errno will be returned, otherwise zero.

```
int genphy_c45_read_lpa(struct phy_device *phydev)
```

read the link partner advertisement and pause

Parameters

```
struct phy_device *phydev
```

target phy_device struct

Description

Read the Clause 45 defined base (7.19) and 10G (7.33) status registers, filling in the link partner advertisement, pause and asym_pause members in **phydev**. This assumes that the auto-negotiation MMD is present, and the backplane bit (7.48.0) is clear. Clause 45 PHY drivers are expected to fill in the remainder of the link partner advert from vendor registers.

```
int genphy_c45_pma_baset1_read_master_slave(struct phy_device *phydev)
    read forced master/slave configuration
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

```
int genphy_c45_read_pma(struct phy_device *phydev)
    read link speed etc from PMA
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

```
int genphy_c45_read_mdix(struct phy_device *phydev)
    read mdix status from PMA
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

```
int genphy_c45_read_eee_abilities(struct phy_device *phydev)
    read supported EEE link modes
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

```
int genphy_c45_pma_baset1_read_abilities(struct phy_device *phydev)
    read supported baset1 link modes from PMA
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

Description

Read the supported link modes from the extended BASE-T1 ability register

```
int genphy_c45_pma_read_ext_abilities(struct phy_device *phydev)
    read supported link modes from PMA
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

Description

Read the supported link modes from the PMA/PMD extended ability register (Register 1.11).

```
int genphy_c45_pma_read_abilities(struct phy_device *phydev)
    read supported link modes from PMA
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

Description

Read the supported link modes from the PMA Status 2 (1.8) register. If bit 1.8.9 is set, the list of supported modes is build using the values in the PMA Extended Abilities (1.11) register, indicating 1000BASET and 10G related modes. If bit 1.11.14 is set, then the list is also extended with the modes in the 2.5G/5G PMA Extended register (1.21), indicating if 2.5GBASET and 5GBASET are supported.

```
int genphy_c45_read_status(struct phy_device *phydev)
    read PHY status
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

Description

Reads status from PHY and sets phy_device members accordingly.

```
int genphy_c45_config_aneg(struct phy_device *phydev)
    restart auto-negotiation or forced setup
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

Description

If auto-negotiation is enabled, we configure the

advertising, and then restart auto-negotiation. If it is not enabled, then we force a configuration.

```
int genphy_c45_fast_retrain(struct phy_device *phydev, bool enable)
    configure fast retrain registers
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

```
bool enable
    enable fast retrain or not
```

Description

If fast-retrain is enabled, we configure PHY as

advertising fast retrain capable and THP Bypass Request, then enable fast retrain. If it is not enabled, we configure fast retrain disabled.

```
int genphy_c45_plca_get_cfg(struct phy_device *phydev, struct phy_plca_cfg *plca_cfg)
    get PLCA configuration from standard registers
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

struct phy_plca_cfg *plca_cfg
output structure to store the PLCA configuration

Description**if the PHY complies to the Open Alliance TC14 10BASE-T1S PLCA**

Management Registers specifications, this function can be used to retrieve the current PLCA configuration from the standard registers in MMD 31.

int genphy_c45_plca_set_cfg(struct phy_device *phydev, const struct phy_plca_cfg *plca_cfg)
set PLCA configuration using standard registers

Parameters

struct phy_device *phydev
target phy_device struct

const struct phy_plca_cfg *plca_cfg
structure containing the PLCA configuration. Fields set to -1 are not to be changed.

Description**if the PHY complies to the Open Alliance TC14 10BASE-T1S PLCA**

Management Registers specifications, this function can be used to modify the PLCA configuration using the standard registers in MMD 31.

int genphy_c45_plca_get_status(struct phy_device *phydev, struct phy_plca_status *plca_st)
get PLCA status from standard registers

Parameters

struct phy_device *phydev
target phy_device struct

struct phy_plca_status *plca_st
output structure to store the PLCA status

Description**if the PHY complies to the Open Alliance TC14 10BASE-T1S PLCA**

Management Registers specifications, this function can be used to retrieve the current PLCA status information from the standard registers in MMD 31.

int genphy_c45_eee_is_active(struct phy_device *phydev, unsigned long *adv, unsigned long *lp, bool *is_enabled)
get EEE status

Parameters

struct phy_device *phydev
target phy_device struct

unsigned long *adv
variable to store advertised linkmodes

unsigned long *lp
variable to store LP advertised linkmodes

bool *is_enabled

variable to store EEE enabled/disabled configuration value

Description

this function will read local and link partner PHY advertisements. Compare them return current EEE state.

int **genphy_c45_ethtool_get_eee**(struct *phy_device* *phydev, struct ethtool_eee *data)

 get EEE supported and status

Parameters**struct phy_device *phydev**

 target phy_device struct

struct ethtool_eee *data

 ethtool_eee data

Description

it reports the Supported/Advertisement/LP Advertisement capabilities.

int **genphy_c45_ethtool_set_eee**(struct *phy_device* *phydev, struct ethtool_eee *data)

 set EEE supported and status

Parameters**struct phy_device *phydev**

 target phy_device struct

struct ethtool_eee *data

 ethtool_eee data

Description

sets the Supported/Advertisement/LP Advertisement capabilities. If eee_enabled is false, no links modes are advertised, but the previously advertised link modes are retained. This allows EEE to be enabled/disabled in a non-destructive way.

enum phy_interface_t

 Interface Mode definitions

Constants**PHY_INTERFACE_MODE_NA**

 Not Applicable - don't touch

PHY_INTERFACE_MODE_INTERNAL

 No interface, MAC and PHY combined

PHY_INTERFACE_MODE_MII

 Media-independent interface

PHY_INTERFACE_MODE_GMII

 Gigabit media-independent interface

PHY_INTERFACE_MODE_SGMII

 Serial gigabit media-independent interface

PHY_INTERFACE_MODE_TBI

 Ten Bit Interface

PHY_INTERFACE_MODE_REVMI

Reverse Media Independent Interface

PHY_INTERFACE_MODE_RMII

Reduced Media Independent Interface

PHY_INTERFACE_MODE_REVRMII

Reduced Media Independent Interface in PHY role

PHY_INTERFACE_MODE_RGMII

Reduced gigabit media-independent interface

PHY_INTERFACE_MODE_RGMII_ID

RGMII with Internal RX+TX delay

PHY_INTERFACE_MODE_RGMII_RXID

RGMII with Internal RX delay

PHY_INTERFACE_MODE_RGMII_TXID

RGMII with Internal RX delay

PHY_INTERFACE_MODE_RTBI

Reduced TBI

PHY_INTERFACE_MODE_SMII

Serial MII

PHY_INTERFACE_MODE_XGMII

10 gigabit media-independent interface

PHY_INTERFACE_MODE_XLGMII

40 gigabit media-independent interface

PHY_INTERFACE_MODE_MOCA

Multimedia over Coax

PHY_INTERFACE_MODE_PSGMII

Penta SGMII

PHY_INTERFACE_MODE_QSGMII

Quad SGMII

PHY_INTERFACE_MODE_TRGMII

Turbo RGMII

PHY_INTERFACE_MODE_100BASEX

100 BaseX

PHY_INTERFACE_MODE_1000BASEX

1000 BaseX

PHY_INTERFACE_MODE_2500BASEX

2500 BaseX

PHY_INTERFACE_MODE_5GBASER

5G BaseR

PHY_INTERFACE_MODE_RXAUI

Reduced XAUI

PHY_INTERFACE_MODE_XAUI

10 Gigabit Attachment Unit Interface

PHY_INTERFACE_MODE_10GBASER

10G BaseR

PHY_INTERFACE_MODE_25GBASER

25G BaseR

PHY_INTERFACE_MODE_USXGMII

Universal Serial 10GE MII

PHY_INTERFACE_MODE_10GKR

10GBASE-KR - with Clause 73 AN

PHY_INTERFACE_MODE_QUSGMII

Quad Universal SGMII

PHY_INTERFACE_MODE_1000BASEKX

1000Base-KX - with Clause 73 AN

PHY_INTERFACE_MODE_MAX

Book keeping

Description

Describes the interface between the MAC and PHY.

```
const char *phy_modes(phy_interface_t interface)
```

map phy_interface_t enum to device tree binding of phy-mode

Parameters***phy_interface_t* interface**

enum phy_interface_t value

Description

maps enum *phy_interface_t* defined in this file into the device tree binding of 'phy-mode', so that Ethernet device driver can get PHY interface from device tree.

struct mdio_bus_stats

Statistics counters for MDIO busses

Definition:

```
struct mdio_bus_stats {
    u64_stats_t transfers;
    u64_stats_t errors;
    u64_stats_t writes;
    u64_stats_t reads;
    struct u64_stats_sync syncp;
};
```

Members**transfers**

Total number of transfers, i.e. **writes + reads**

errors

Number of MDIO transfers that returned an error

writes

Number of write transfers

reads

Number of read transfers

syncp

Synchronisation for incrementing statistics

struct phy_package_shared

Shared information in PHY packages

Definition:

```
struct phy_package_shared {
    u8 base_addr;
    refcount_t refcnt;
    unsigned long flags;
    size_t priv_size;
    void *priv;
};
```

Members**base_addr**

Base PHY address of PHY package used to combine PHYs in one package and for offset calculation of phy_package_read/write

refcnt

Number of PHYs connected to this shared data

flags

Initialization of PHY package

priv_size

Size of the shared private data **priv**

priv

Driver private data shared across a PHY package

Description

Represents a shared structure between different phydev's in the same package, for example a quad PHY. See [phy_package_join\(\)](#) and [phy_package_leave\(\)](#).

struct mii_bus

Represents an MDIO bus

Definition:

```
struct mii_bus {
    struct module *owner;
    const char *name;
    char id[MII_BUS_ID_SIZE];
    void *priv;
```

```

int (*read)(struct mii_bus *bus, int addr, int regnum);
int (*write)(struct mii_bus *bus, int addr, int regnum, u16 val);
int (*read_c45)(struct mii_bus *bus, int addr, int devnum, int regnum);
int (*write_c45)(struct mii_bus *bus, int addr, int devnum, int regnum, u16 val);
int (*reset)(struct mii_bus *bus);
struct mdio_bus_stats stats[PHY_MAX_ADDR];
struct mutex mdio_lock;
struct device *parent;
enum {
    MDIOPORT_ALLOCATED = 1,
    MDIOPORT_REGISTERED,
    MDIOPORT_UNREGISTERED,
    MDIOPORT_RELEASED,
} state;
struct device dev;
struct mdio_device *mdio_map[PHY_MAX_ADDR];
u32 phy_mask;
u32 phy_ignore_ta_mask;
int irq[PHY_MAX_ADDR];
int reset_delay_us;
int reset_post_delay_us;
struct gpio_desc *reset_gpiod;
struct mutex shared_lock;
struct phy_package_shared *shared[PHY_MAX_ADDR];
};


```

Members

owner

Who owns this device

name

User friendly name for this MDIO device, or driver name

id

Unique identifier for this bus, typical from bus hierarchy

priv

Driver private data

read

Perform a read transfer on the bus

write

Perform a write transfer on the bus

read_c45

Perform a C45 read transfer on the bus

write_c45

Perform a C45 write transfer on the bus

reset

Perform a reset of the bus

stats

Statistic counters per device on the bus

mdio_lock

A lock to ensure that only one thing can read/write the MDIO bus at a time

parent

Parent device of this bus

state

State of bus structure

dev

Kernel device representation

mdio_map

list of all MDIO devices on bus

phy_mask

PHY addresses to be ignored when probing

phy_ignore_ta_mask

PHY addresses to ignore the TA/read failure

irq

An array of interrupts, each PHY's interrupt at the index matching its address

reset_delay_us

GPIO reset pulse width in microseconds

reset_post_delay_us

GPIO reset deassert delay in microseconds

reset_gpiod

Reset GPIO descriptor pointer

shared_lock

protect access to the shared element

shared

shared state across different PHYs

Description

The Bus class for PHYs. Devices which provide access to PHYs should register using this structure

```
struct mii_bus *mdiobus_alloc(void)
```

Allocate an MDIO bus structure

Parameters**void**

no arguments

Description

The internal state of the MDIO bus will be set of MDIOBUS_ALLOCATED ready for the driver to register the bus.

enum **phy_state**

PHY state machine states:

Constants

PHY_DOWN

PHY device and driver are not ready for anything. probe should be called if and only if the PHY is in this state, given that the PHY device exists. - PHY driver probe function will set the state to **PHY_READY**

PHY_READY

PHY is ready to send and receive packets, but the controller is not. By default, PHYs which do not implement probe will be set to this state by *phy_probe()*. - start will set the state to UP

PHY_HALTED

PHY is up, but no polling or interrupts are done. - phy_start moves to **PHY_UP**

PHY_ERROR

PHY is up, but is in an error state. - phy_stop moves to **PHY_HALTED**

PHY_UP

The PHY and attached device are ready to do work. Interrupts should be started here. - timer moves to **PHY_NOLINK** or **PHY_RUNNING**

PHY_RUNNING

PHY is currently up, running, and possibly sending and/or receiving packets - irq or timer will set **PHY_NOLINK** if link goes down - phy_stop moves to **PHY_HALTED**

PHY_NOLINK

PHY is up, but not currently plugged in. - irq or timer will set **PHY_RUNNING** if link comes back - phy_stop moves to **PHY_HALTED**

PHY_CABLETEST

PHY is performing a cable test. Packet reception/sending is not expected to work, carrier will be indicated as down. PHY will be poll once per second, or on interrupt for its current state. Once complete, move to UP to restart the PHY. - phy_stop aborts the running test and moves to **PHY_HALTED**

struct **phy_c45_device_ids**

802.3-c45 Device Identifiers

Definition:

```
struct phy_c45_device_ids {  
    u32 devices_in_package;  
    u32 mmds_present;  
    u32 device_ids[MDIO_MMD_NUM];  
};
```

Members

devices_in_package

IEEE 802.3 devices in package register value.

mmds_present

bit vector of MMDs present.

device_ids

The device identifier for each present device.

struct phy_device

An instance of a PHY

Definition:

```
struct phy_device {
    struct mdio_device mdio;
    struct phy_driver *drv;
    struct device_link *devlink;
    u32 phy_id;
    struct phy_c45_device_ids c45_ids;
    unsigned is_c45:1;
    unsigned is_internal:1;
    unsigned is_pseudo_fixed_link:1;
    unsigned is_gigabit_capable:1;
    unsigned has_fixups:1;
    unsigned suspended:1;
    unsigned suspended_by_mdio_bus:1;
    unsigned sysfs_links:1;
    unsigned loopback_enabled:1;
    unsigned downshifted_rate:1;
    unsigned is_on_sfp_module:1;
    unsigned mac_managed_pm:1;
    unsigned wol_enabled:1;
    unsigned autoneg:1;
    unsigned link:1;
    unsigned autoneg_complete:1;
    unsigned interrupts:1;
    unsigned irq_suspended:1;
    unsigned irq_rerun:1;
    int rate_matching;
    enum phy_state state;
    u32 dev_flags;
    phy_interface_t interface;
    unsigned long possible_interfaces[BITS_TO_LONGS(PHY_INTERFACE_MODE_MAX)];
    int speed;
    int duplex;
    int port;
    int pause;
    int asym_pause;
    u8 master_slave_get;
    u8 master_slave_set;
    u8 master_slave_state;
    unsigned long supported[BITS_TO_LONGS(__ETHTOOL_LINK_MODE_MASK_NBITS)];
    unsigned long advertising[BITS_TO_LONGS(__ETHTOOL_LINK_MODE_MASK_NBITS)];
    unsigned long lp_advertising[BITS_TO_LONGS(__ETHTOOL_LINK_MODE_MASK_NBITS)];
    unsigned long adv_old[BITS_TO_LONGS(__ETHTOOL_LINK_MODE_MASK_NBITS)];
    unsigned long supported_eee[BITS_TO_LONGS(__ETHTOOL_LINK_MODE_MASK_NBITS)];
```

```

    unsigned long advertising_eee[BITS_TO_LONGS(__ETHTOOL_LINK_MODE_MASK_
→NBITS)];
    bool eee_enabled;
    unsigned long host_interfaces[BITS_TO_LONGS(PHY_INTERFACE_MODE_MAX)];
    u32 eee_broken_modes;
#endif CONFIG_LED_TRIGGER_PHY;
    struct phy_led_trigger *phy_led_triggers;
    unsigned int phy_num_led_triggers;
    struct phy_led_trigger *last_triggered;
    struct phy_led_trigger *led_link_trigger;
#endif;
    struct list_head leds;
    int irq;
    void *priv;
    struct phy_package_shared *shared;
    struct sk_buff *skb;
    void *ehdr;
    struct nla_attr *nest;
    struct delayed_work state_queue;
    struct mutex lock;
    bool sfp_bus_attached;
    struct sfp_bus *sfp_bus;
    struct phylink *phylink;
    struct net_device *attached_dev;
    struct mii_timestamper *mii_ts;
    struct pse_control *psec;
    u8 mdix;
    u8 mdix_ctrl;
    int pma_extable;
    unsigned int link_down_events;
    void (*phy_link_change)(struct phy_device *phydev, bool up);
    void (*adjust_link)(struct net_device *dev);
#if IS_ENABLED(CONFIG_MACSEC);
    const struct macsec_ops *macsec_ops;
#endif;
};

```

Members

mdio

MDIO bus this PHY is on

drv

Pointer to the driver for this PHY instance

devlink

Create a link between phy dev and mac dev, if the external phy used by current mac interface is managed by another mac interface.

phy_id

UID for this device found during discovery

c45_ids

802.3-c45 Device Identifiers if is_c45.

is_c45

Set to true if this PHY uses clause 45 addressing.

is_internal

Set to true if this PHY is internal to a MAC.

is_pseudo_fixed_link

Set to true if this PHY is an Ethernet switch, etc.

is_gigabit_capable

Set to true if PHY supports 1000Mbps

has_fixups

Set to true if this PHY has fixups/quirks.

suspended

Set to true if this PHY has been suspended successfully.

suspended_by_mdio_bus

Set to true if this PHY was suspended by MDIO bus.

sysfs_links

Internal boolean tracking sysfs symbolic links setup/removal.

loopback_enabled

Set true if this PHY has been loopbacked successfully.

downshifted_rate

Set true if link speed has been downshifted.

is_on_sfp_module

Set true if PHY is located on an SFP module.

mac_managed_pm

Set true if MAC driver takes care of suspending/resuming PHY

wol_enabled

Set to true if the PHY or the attached MAC have Wake-on-LAN enabled.

autoneg

Flag autoneg being used

link

Current link state

autoneg_complete

Flag auto negotiation of the link has completed

interrupts

Flag interrupts have been enabled

irq_suspended

Flag indicating PHY is suspended and therefore interrupt handling shall be postponed until PHY has resumed

irq_rerun

Flag indicating interrupts occurred while PHY was suspended, requiring a rerun of the interrupt handler after resume

rate_matching

Current rate matching mode

state

State of the PHY for management purposes

dev_flags

Device-specific flags used by the PHY driver.

- Bits [15:0] are free to use by the PHY driver to communicate driver specific behavior.
- Bits [23:16] are currently reserved for future use.
- Bits [31:24] are reserved for defining generic PHY driver behavior.

interface

enum phy_interface_t value

possible_interfaces

bitmap if interface modes that the attached PHY will switch between depending on media speed.

speed

Current link speed

duplex

Current duplex

port

Current port

pause

Current pause

asym_pause

Current asymmetric pause

master_slave_get

Current master/slave advertisement

master_slave_set

User requested master/slave configuration

master_slave_state

Current master/slave configuration

supported

Combined MAC/PHY supported linkmodes

advertising

Currently advertised linkmodes

lp_advertising

Current link partner advertised linkmodes

adv_old

Saved advertised while power saving for WoL

supported_eee

supported PHY EEE linkmodes

advertising_eee

Currently advertised EEE linkmodes

eee_enabled

Flag indicating whether the EEE feature is enabled

host_interfaces

PHY interface modes supported by host

eee_broken_modes

Energy efficient ethernet modes which should be prohibited

phy_led_triggers

Array of LED triggers

phy_num_led_triggers

Number of triggers in **phy_led_triggers**

last_triggered

last LED trigger for link speed

led_link_trigger

LED trigger for link up/down

leds

list of PHY LED structures

irq

IRQ number of the PHY's interrupt (-1 if none)

priv

Pointer to driver private data

shared

Pointer to private data shared by phys in one package

skb

Netlink message for cable diagnostics

ehdr

nNtlink header for cable diagnostics

nest

Netlink nest used for cable diagnostics

state_queue

Work queue for state machine

lock

Mutex for serialization access to PHY

sfp_bus_attached

Flag indicating whether the SFP bus has been attached

sfp_bus

SFP bus attached to this PHY's fiber port

phylink

Pointer to phylink instance for this PHY

attached_dev

The attached enet driver's device instance ptr

mii_ts

Pointer to time stamper callbacks

psec

Pointer to Power Sourcing Equipment control struct

mdix

Current crossover

mdix_ctrl

User setting of crossover

pma_extable

Cached value of PMA/PMD Extended Abilities Register

link_down_events

Number of times link was lost

phy_link_change

Callback for phylink for notification of link change

adjust_link

Callback for the enet controller to respond to changes: in the link state.

macsec_ops

MACsec offloading ops.

Description

interrupts currently only supports enabled or disabled, but could be changed in the future to support enabling and disabling specific interrupts

Contains some infrastructure for polling and interrupt handling, as well as handling shifts in PHY hardware state

struct phy_tdr_config

Configuration of a TDR raw test

Definition:

```
struct phy_tdr_config {  
    u32 first;  
    u32 last;  
    u32 step;  
    s8 pair;  
};
```

Members**first**

Distance for first data collection point

last

Distance for last data collection point

step

Step between data collection points

pair

Bitmap of cable pairs to collect data for

Description

A structure containing possible configuration parameters for a TDR cable test. The driver does not need to implement all the parameters, but should report what is actually used. All distances are in centimeters.

struct phy_plca_cfg

Configuration of the PLCA (Physical Layer Collision Avoidance) Reconciliation Sublayer.

Definition:

```
struct phy_plca_cfg {
    int version;
    int enabled;
    int node_id;
    int node_cnt;
    int to_tmr;
    int burst_cnt;
    int burst_tmr;
};
```

Members**version**

read-only PLCA register map version. -1 = not available. Ignored when setting the configuration. Format is the same as reported by the PLCA IDVER register (31.CA00). -1 = not available.

enabled

PLCA configured mode (enabled/disabled). -1 = not available / don't set. 0 = disabled, anything else = enabled.

node_id

the PLCA local node identifier. -1 = not available / don't set. Allowed values [0 .. 254]. 255 = node disabled.

node_cnt

the PLCA node count (maximum number of nodes having a TO). Only meaningful for the coordinator (node_id = 0). -1 = not available / don't set. Allowed values [1 .. 255].

to_tmr

The value of the PLCA to_timer in bit-times, which determines the PLCA transmit opportunity window opening. See IEEE802.3 Clause 148 for more details. The to_timer shall be set equal over all nodes. -1 = not available / don't set. Allowed values [0 .. 255].

burst_cnt

controls how many additional frames a node is allowed to send in single transmit opportunity (TO). The default value of 0 means that the node is allowed exactly one frame per TO. A value of 1 allows two frames per TO, and so on. -1 = not available / don't set. Allowed values [0 .. 255].

burst_tmr

controls how many bit times to wait for the MAC to send a new frame before interrupting

the burst. This value should be set to a value greater than the MAC inter-packet gap (which is typically 96 bits). -1 = not available / don't set. Allowed values [0 .. 255].

Description

A structure containing configuration parameters for setting/getting the PLCA RS configuration. The driver does not need to implement all the parameters, but should report what is actually used.

struct **phy_plca_status**

Status of the PLCA (Physical Layer Collision Avoidance) Reconciliation Sublayer.

Definition:

```
struct phy_plca_status {  
    bool pst;  
};
```

Members

pst

The PLCA status as reported by the PST bit in the PLCA STATUS register(31.CA03), indicating BEACON activity.

Description

A structure containing status information of the PLCA RS configuration. The driver does not need to implement all the parameters, but should report what is actually used.

struct **phy_led**

An LED driven by the PHY

Definition:

```
struct phy_led {  
    struct list_head list;  
    struct phy_device *phydev;  
    struct led_classdev led_cdev;  
    u8 index;  
};
```

Members

list

List of LEDs

phydev

PHY this LED is attached to

led_cdev

Standard LED class structure

index

Number of the LED

struct **phy_driver**

Driver structure for a particular PHY type

Definition:

```

struct phy_driver {
    struct mdio_driver_common mdiodrv;
    u32 phy_id;
    char *name;
    u32 phy_id_mask;
    const unsigned long * const features;
    u32 flags;
    const void *driver_data;
    int (*soft_reset)(struct phy_device *phydev);
    int (*config_init)(struct phy_device *phydev);
    int (*probe)(struct phy_device *phydev);
    int (*get_features)(struct phy_device *phydev);
    int (*get_rate_matching)(struct phy_device *phydev, phy_interface_t iface);
    int (*suspend)(struct phy_device *phydev);
    int (*resume)(struct phy_device *phydev);
    int (*config_aneg)(struct phy_device *phydev);
    int (*aneg_done)(struct phy_device *phydev);
    int (*read_status)(struct phy_device *phydev);
    int (*config_intr)(struct phy_device *phydev);
    irqreturn_t (*handle_interrupt)(struct phy_device *phydev);
    void (*remove)(struct phy_device *phydev);
    int (*match_phy_device)(struct phy_device *phydev);
    int (*set_wol)(struct phy_device *dev, struct ethtool_wolinfo *wol);
    void (*get_wol)(struct phy_device *dev, struct ethtool_wolinfo *wol);
    void (*link_change_notify)(struct phy_device *dev);
    int (*read_mmd)(struct phy_device *dev, int devnum, u16 regnum);
    int (*write_mmd)(struct phy_device *dev, int devnum, u16 regnum, u16 val);
    int (*read_page)(struct phy_device *dev);
    int (*write_page)(struct phy_device *dev, int page);
    int (*module_info)(struct phy_device *dev, struct ethtool_modinfo *modinfo);
    int (*module_eeprom)(struct phy_device *dev, struct ethtool_eeprom *ee, u8 *data);
    int (*cable_test_start)(struct phy_device *dev);
    int (*cable_test_tdr_start)(struct phy_device *dev, const struct phy_tdr_config *config);
    int (*cable_test_get_status)(struct phy_device *dev, bool *finished);
    int (*get_sset_count)(struct phy_device *dev);
    void (*get_strings)(struct phy_device *dev, u8 *data);
    void (*get_stats)(struct phy_device *dev, struct ethtool_stats *stats, u64 *data);
    int (*get_tunable)(struct phy_device *dev, struct ethtool_tunable *tuna, void *data);
    int (*set_tunable)(struct phy_device *dev, struct ethtool_tunable *tuna, const void *data);
    int (*set_loopback)(struct phy_device *dev, bool enable);
    int (*get_sqi)(struct phy_device *dev);
    int (*get_sqi_max)(struct phy_device *dev);
    int (*get_plca_cfg)(struct phy_device *dev, struct phy_plca_cfg *plca_cfg);
    int (*set_plca_cfg)(struct phy_device *dev, const struct phy_plca_cfg *plca_cfg);
}

```

```

    int (*get_plca_status)(struct phy_device *dev, struct phy_plca_statusu
    ↵*plca_st);
    int (*led_brightness_set)(struct phy_device *dev, u8 index, enum led_
    ↵brightness value);
    int (*led_blink_set)(struct phy_device *dev, u8 index, unsigned long *delay_
    ↵on, unsigned long *delay_off);
    int (*led_hw_is_supported)(struct phy_device *dev, u8 index, unsigned longu
    ↵rules);
    int (*led_hw_control_set)(struct phy_device *dev, u8 index, unsigned longu
    ↵rules);
    int (*led_hw_control_get)(struct phy_device *dev, u8 index, unsigned longu
    ↵*rules);
};

}

```

Members

mdiodrv

Data common to all MDIO devices

phy_id

The result of reading the UID registers of this PHY type, and ANDing them with the phy_id_mask. This driver only works for PHYs with IDs which match this field

name

The friendly name of this PHY type

phy_id_mask

Defines the important bits of the phy_id

features

A mandatory list of features (speed, duplex, etc) supported by this PHY

flags

A bitfield defining certain other features this PHY supports (like interrupts)

driver_data

Static driver data

soft_reset

Called to issue a PHY software reset

config_init

Called to initialize the PHY, including after a reset

probe

Called during discovery. Used to set up device-specific structures, if any

get_features

Probe the hardware to determine what abilities it has. Should only set phydev->supported.

get_rate_matching

Get the supported type of rate matching for a particular phy interface. This is used by phy consumers to determine whether to advertise lower-speed modes for that interface. It is assumed that if a rate matching mode is supported on an interface, then that interface's rate can be adapted to all slower link speeds supported by the phy. If the interface is not supported, this should return RATE_MATCH_NONE.

suspend

Suspend the hardware, saving state if needed

resume

Resume the hardware, restoring state if needed

config_aneg

Configures the advertisement and resets autonegotiation if phydev->autoneg is on, forces the speed to the current settings in phydev if phydev->autoneg is off

aneg_done

Determines the auto negotiation result

read_status

Determines the negotiated speed and duplex

config_intr

Enables or disables interrupts. It should also clear any pending interrupts prior to enabling the IRQs and after disabling them.

handle_interrupt

Override default interrupt handling

remove

Clears up any memory if needed

match_phy_device

Returns true if this is a suitable driver for the given phydev. If NULL, matching is based on phy_id and phy_id_mask.

set_wol

Some devices (e.g. qnap TS-119P II) require PHY register changes to enable Wake on LAN, so set_wol is provided to be called in the ethernet driver's set_wol function.

get_wol

See set_wol, but for checking whether Wake on LAN is enabled.

link_change_notify

Called to inform a PHY device driver when the core is about to change the link state. This callback is supposed to be used as fixup hook for drivers that need to take action when the link state changes. Drivers are by no means allowed to mess with the PHY device structure in their implementations.

read_mmd

PHY specific driver override for reading a MMD register. This function is optional for PHY specific drivers. When not provided, the default MMD read function will be used by [*phy_read_mmd\(\)*](#), which will use either a direct read for Clause 45 PHYs or an indirect read for Clause 22 PHYs. devnum is the MMD device number within the PHY device, regnum is the register within the selected MMD device.

write_mmd

PHY specific driver override for writing a MMD register. This function is optional for PHY specific drivers. When not provided, the default MMD write function will be used by [*phy_write_mmd\(\)*](#), which will use either a direct write for Clause 45 PHYs, or an indirect write for Clause 22 PHYs. devnum is the MMD device number within the PHY device, regnum is the register within the selected MMD device. val is the value to be written.

read_page

Return the current PHY register page number

write_page

Set the current PHY register page number

module_info

Get the size and type of the eeprom contained within a plug-in module

module_eeprom

Get the eeprom information from the plug-in module

cable_test_start

Start a cable test

cable_test_tdr_start

Start a raw TDR cable test

cable_test_get_status

Once per second, or on interrupt, request the status of the test.

get_sset_count

Number of statistic counters

get_strings

Names of the statistic counters

get_stats

Return the statistic counter values

get_tunable

Return the value of a tunable

set_tunable

Set the value of a tunable

set_loopback

Set the loopback mood of the PHY

get_sqi

Get the signal quality indication

get_sqi_max

Get the maximum signal quality indication

get_plca_cfg

Return the current PLCA configuration

set_plca_cfg

Set the PLCA configuration

get_plca_status

Return the current PLCA status info

led_brightness_set

Set a PHY LED brightness. Index indicates which of the PHYs led should be set. Value follows the standard LED class meaning, e.g. LED_OFF, LED_HALF, LED_FULL.

led_blink_set

Set a PHY LED brightness. Index indicates which of the PHYs led should be configured to blink. Delays are in milliseconds and if both are zero then a sensible default should be

chosen. The call should adjust the timings in that case and if it can't match the values specified exactly.

`led_hw_is_supported`

Can the HW support the given rules. **dev**: PHY device which has the LED **index**: Which LED of the PHY device **rules** The core is interested in these rules

Return 0 if yes, -EOPNOTSUPP if not, or an error code.

`led_hw_control_set`

Set the HW to control the LED **dev**: PHY device which has the LED **index**: Which LED of the PHY device **rules** The rules used to control the LED

Returns 0, or a an error code.

`led_hw_control_get`

Get how the HW is controlling the LED **dev**: PHY device which has the LED **index**: Which LED of the PHY device **rules** Pointer to the rules used to control the LED

Set ***rules** to how the HW is currently blinking. Returns 0 on success, or a error code if the current blinking cannot be represented in rules, or some other error happens.

Description

All functions are optional. If config_aneg or read_status are not implemented, the phy core uses the genphy versions. Note that none of these functions should be called from interrupt time. The goal is for the bus read/write functions to be able to block when the bus transaction is happening, and be freed up by an interrupt (The MPC85xx has this ability, though it is not currently supported in the driver).

`bool phy_id_compare(u32 id1, u32 id2, u32 mask)`
compare **id1** with **id2** taking account of **mask**

Parameters

u32 id1
first PHY ID

u32 id2
second PHY ID

u32 mask
the PHY ID mask, set bits are significant in matching

Description

Return true if the bits from **id1** and **id2** specified by **mask** match. This uses an equivalent test to **(id & mask) == (phy_id & mask)**.

`bool phydev_id_compare(struct phy_device *phydev, u32 id)`
compare **id** with the PHY's Clause 22 ID

Parameters

struct phy_device *phydev
the PHY device

u32 id
the PHY ID to be matched

Description

Compare the **phydev** clause 22 ID with the provided **id** and return true or false depending whether it matches, using the bound driver mask. The **phydev** must be bound to a driver.

bool phy_is_started(struct phy_device *phydev)

Convenience function to check whether PHY is started

Parameters

struct phy_device *phydev

The phy_device struct

int phy_read(struct phy_device *phydev, u32 regnum)

Convenience function for reading a given PHY register

Parameters

struct phy_device *phydev

the phy_device struct

u32 regnum

register number to read

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

int __phy_read(struct phy_device *phydev, u32 regnum)

convenience function for reading a given PHY register

Parameters

struct phy_device *phydev

the phy_device struct

u32 regnum

register number to read

Description

The caller must have taken the MDIO bus lock.

int phy_write(struct phy_device *phydev, u32 regnum, u16 val)

Convenience function for writing a given PHY register

Parameters

struct phy_device *phydev

the phy_device struct

u32 regnum

register number to write

u16 val

value to write to **regnum**

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

```
int __phy_write(struct phy_device *phydev, u32 regnum, u16 val)
```

Convenience function for writing a given PHY register

Parameters

struct phy_device *phydev

the phy_device struct

u32 regnum

register number to write

u16 val

value to write to **regnum**

Description

The caller must have taken the MDIO bus lock.

```
int __phy_modify_changed(struct phy_device *phydev, u32 regnum, u16 mask, u16 set)
```

Convenience function for modifying a PHY register

Parameters

struct phy_device *phydev

a pointer to a *struct phy_device*

u32 regnum

register number

u16 mask

bit mask of bits to clear

u16 set

bit mask of bits to set

Description

Unlocked helper function which allows a PHY register to be modified as new register value = (old register value & ~mask) | set

Returns negative errno, 0 if there was no change, and 1 in case of change

phy_read_mmd_poll_timeout

```
phy_read_mmd_poll_timeout (phydev, devaddr, regnum, val, cond, sleep_us,
                           timeout_us, sleep_before_read)
```

Periodically poll a PHY register until a condition is met or a timeout occurs

Parameters

phydev

The phy_device struct

devaddr

The MMD to read from

regnum

The register on the MMD to read

val

Variable to read the register into

cond

Break condition (usually involving **val**)

sleep_us

Maximum time to sleep between reads in us (0 tight-loops). Should be less than ~20ms since usleep_range is used (see Documentation/timers/timers-howto.rst).

timeout_us

Timeout in us, 0 means never timeout

sleep_before_read

if it is true, sleep **sleep_us** before read. Returns 0 on success and -ETIMEDOUT upon a timeout. In either case, the last read value at **args** is stored in **val**. Must not be called from atomic context if sleep_us or timeout_us are used.

int **__phy_set_bits**(struct *phy_device* *phydev, u32 regnum, u16 val)

Convenience function for setting bits in a PHY register

Parameters**struct phy_device *phydev**

the phy_device struct

u32 regnum

register number to write

u16 val

bits to set

Description

The caller must have taken the MDIO bus lock.

int **__phy_clear_bits**(struct *phy_device* *phydev, u32 regnum, u16 val)

Convenience function for clearing bits in a PHY register

Parameters**struct phy_device *phydev**

the phy_device struct

u32 regnum

register number to write

u16 val

bits to clear

Description

The caller must have taken the MDIO bus lock.

int **phy_set_bits**(struct *phy_device* *phydev, u32 regnum, u16 val)

Convenience function for setting bits in a PHY register

Parameters**struct phy_device *phydev**

the phy_device struct

u32 regnum

register number to write

u16 val

bits to set

int phy_clear_bits(struct *phy_device* *phydev, u32 regnum, u16 val)

Convenience function for clearing bits in a PHY register

Parameters**struct phy_device *phydev**

the phy_device struct

u32 regnum

register number to write

u16 val

bits to clear

int __phy_set_bits_mmd(struct *phy_device* *phydev, int devad, u32 regnum, u16 val)

Convenience function for setting bits in a register on MMD

Parameters**struct phy_device *phydev**

the phy_device struct

int devad

the MMD containing register to modify

u32 regnum

register number to modify

u16 val

bits to set

Description

The caller must have taken the MDIO bus lock.

int __phy_clear_bits_mmd(struct *phy_device* *phydev, int devad, u32 regnum, u16 val)

Convenience function for clearing bits in a register on MMD

Parameters**struct phy_device *phydev**

the phy_device struct

int devad

the MMD containing register to modify

u32 regnum

register number to modify

u16 val

bits to clear

Description

The caller must have taken the MDIO bus lock.

int phy_set_bits_mmd(struct *phy_device* *phydev, int devad, u32 regnum, u16 val)

Convenience function for setting bits in a register on MMD

Parameters

struct phy_device *phydev

the phy_device struct

int devad

the MMD containing register to modify

u32 regnum

register number to modify

u16 val

bits to set

int phy_clear_bits_mmd(struct phy_device *phydev, int devad, u32 regnum, u16 val)

Convenience function for clearing bits in a register on MMD

Parameters

struct phy_device *phydev

the phy_device struct

int devad

the MMD containing register to modify

u32 regnum

register number to modify

u16 val

bits to clear

bool phy_interrupt_is_valid(struct phy_device *phydev)

Convenience function for testing a given PHY irq

Parameters

struct phy_device *phydev

the phy_device struct

NOTE

must be kept in sync with addition/removal of PHY_POLL and PHY_MAC_INTERRUPT

bool phy_polling_mode(struct phy_device *phydev)

Convenience function for testing whether polling is used to detect PHY status changes

Parameters

struct phy_device *phydev

the phy_device struct

bool phy_has_hwstamp(struct phy_device *phydev)

Tests whether a PHY time stamp configuration.

Parameters

struct phy_device *phydev

the phy_device struct

`bool phy_has_rxtstamp(struct phy_device *phydev)`

Tests whether a PHY supports receive time stamping.

Parameters

struct phy_device *phydev

the phy_device struct

`bool phy_has_tsinfo(struct phy_device *phydev)`

Tests whether a PHY reports time stamping and/or PTP hardware clock capabilities.

Parameters

struct phy_device *phydev

the phy_device struct

`bool phy_has_txstamp(struct phy_device *phydev)`

Tests whether a PHY supports transmit time stamping.

Parameters

struct phy_device *phydev

the phy_device struct

`bool phy_is_internal(struct phy_device *phydev)`

Convenience function for testing if a PHY is internal

Parameters

struct phy_device *phydev

the phy_device struct

`bool phy_on_sfp(struct phy_device *phydev)`

Convenience function for testing if a PHY is on an SFP module

Parameters

struct phy_device *phydev

the phy_device struct

`bool phy_interface_mode_is_rgmii(phy_interface_t mode)`

Convenience function for testing if a PHY interface mode is RGMII (all variants)

Parameters

phy_interface_t mode

the `phy_interface_t` enum

`bool phy_interface_mode_is_8023z(phy_interface_t mode)`

does the PHY interface mode use 802.3z negotiation

Parameters

phy_interface_t mode

one of `enum phy_interface_t`

Description

Returns true if the PHY interface mode uses the 16-bit negotiation word as defined in 802.3z. (See 802.3-2015 37.2.1 Config_Reg encoding)

bool phy_interface_is_rgmii(struct phy_device *phydev)

Convenience function for testing if a PHY interface is RGMII (all variants)

Parameters

struct phy_device *phydev

the phy_device struct

bool phy_is_pseudo_fixed_link(struct phy_device *phydev)

Convenience function for testing if this PHY is the CPU port facing side of an Ethernet switch, or similar.

Parameters

struct phy_device *phydev

the phy_device struct

phy_module_driver

phy_module_driver (__phy_drivers, __count)

Helper macro for registering PHY drivers

Parameters

__phy_drivers

array of PHY drivers to register

__count

Numbers of members in array

Description

Helper macro for PHY drivers which do not do anything special in module init/exit. Each module may only use this macro once, and calling it replaces module_init() and module_exit().

int phy_register_fixup(const char *bus_id, u32 phy_uid, u32 phy_uid_mask, int (*run)(struct phy_device*))

creates a new phy_fixup and adds it to the list

Parameters

const char *bus_id

A string which matches phydev->mdio.dev.bus_id (or PHY_ANY_ID)

u32 phy_uid

Used to match against phydev->phy_id (the UID of the PHY) It can also be PHY_ANY_UID

u32 phy_uid_mask

Applied to phydev->phy_id and fixup->phy_uid before comparison

int (*run)(struct phy_device *)

The actual code to be run when a matching PHY is found

int phy_unregister_fixup(const char *bus_id, u32 phy_uid, u32 phy_uid_mask)

remove a phy_fixup from the list

Parameters

const char *bus_id

A string matches fixup->bus_id (or PHY_ANY_ID) in phy_fixup_list

u32 phy_uid

A phy id matches fixup->phy_id (or PHY_ANY_UID) in phy_fixup_list

u32 phy_uid_mask

Applied to phy_uid and fixup->phy_uid before comparison

struct *phy_device* *get_phy_device(struct *mii_bus* *bus, int addr, bool is_c45)

reads the specified PHY device and returns its **phy_device** struct

Parameters**struct mii_bus *bus**

the target MII bus

int addr

PHY address on the MII bus

bool is_c45

If true the PHY uses the 802.3 clause 45 protocol

Description

Probe for a PHY at **addr** on **bus**.

When probing for a clause 22 PHY, then read the ID registers. If we find a valid ID, allocate and return a *struct phy_device*.

When probing for a clause 45 PHY, read the "devices in package" registers. If the "devices in package" appears valid, read the ID registers for each MMD, allocate and return a *struct phy_device*.

Returns an allocated *struct phy_device* on success, -ENODEV if there is no PHY present, or -EIO on bus access error.

int phy_device_register(struct *phy_device* *phydev)

Register the phy device on the MDIO bus

Parameters**struct phy_device *phydev**

phy_device structure to be added to the MDIO bus

void phy_device_remove(struct *phy_device* *phydev)

Remove a previously registered phy device from the MDIO bus

Parameters**struct phy_device *phydev**

phy_device structure to remove

Description

This doesn't free the phy_device itself, it merely reverses the effects of *phy_device_register()*. Use *phy_device_free()* to free the device after calling this function.

int phy_get_c45_ids(struct *phy_device* *phydev)

Read 802.3-c45 IDs for phy device.

Parameters**struct phy_device *phydev**

phy_device structure to read 802.3-c45 IDs

Description

Returns zero on success, -EIO on bus access error, or -ENODEV if the "devices in package" is invalid.

```
struct phy_device *phy_find_first(struct mii_bus *bus)
```

 finds the first PHY device on the bus

Parameters

struct mii_bus *bus

 the target MII bus

```
int phy_connect_direct(struct net_device *dev, struct phy_device *phydev, void  
                          (*handler)(struct net_device*), phy_interface_t interface)
```

 connect an ethernet device to a specific phy_device

Parameters

struct net_device *dev

 the network device to connect

struct phy_device *phydev

 the pointer to the phy device

void (*handler)(struct net_device *)

 callback function for state change notifications

phy_interface_t interface

 PHY device's interface

```
struct phy_device *phy_connect(struct net_device *dev, const char *bus_id, void  
                          (*handler)(struct net_device*), phy_interface_t interface)
```

 connect an ethernet device to a PHY device

Parameters

struct net_device *dev

 the network device to connect

const char *bus_id

 the id string of the PHY device to connect

void (*handler)(struct net_device *)

 callback function for state change notifications

phy_interface_t interface

 PHY device's interface

Description

Convenience function for connecting ethernet

devices to PHY devices. The default behavior is for the PHY infrastructure to handle everything, and only notify the connected driver when the link status changes. If you don't want, or can't use the provided functionality, you may choose to call only the subset of functions which provide the desired functionality.

```
void phy_disconnect(struct phy_device *phydev)
```

 disable interrupts, stop state machine, and detach a PHY device

Parameters

struct phy_device *phydev
target phy_device struct

void phy_sfp_attach(void *upstream, struct sfp_bus *bus)
attach the SFP bus to the PHY upstream network device

Parameters

void *upstream
pointer to the phy device

struct sfp_bus *bus
sfp bus representing cage being attached

Description

This is used to fill in the sfp_upstream_ops .attach member.

void phy_sfp_detach(void *upstream, struct sfp_bus *bus)
detach the SFP bus from the PHY upstream network device

Parameters

void *upstream
pointer to the phy device

struct sfp_bus *bus
sfp bus representing cage being attached

Description

This is used to fill in the sfp_upstream_ops .detach member.

int phy_sfp_probe(struct phy_device *phydev, const struct sfp_upstream_ops *ops)
probe for a SFP cage attached to this PHY device

Parameters

struct phy_device *phydev
Pointer to phy_device

const struct sfp_upstream_ops *ops
SFP's upstream operations

**int phy_attach_direct(struct net_device *dev, struct phy_device *phydev, u32 flags,
 phy_interface_t interface)**
attach a network device to a given PHY device pointer

Parameters

struct net_device *dev
network device to attach

struct phy_device *phydev
Pointer to phy_device to attach

u32 flags
PHY device's dev_flags

phy_interface_t interface

PHY device's interface

Description**Called by drivers to attach to a particular PHY**

device. The phy_device is found, and properly hooked up to the phy_driver. If no driver is attached, then a generic driver is used. The phy_device is given a ptr to the attaching device, and given a callback for link status change. The phy_device is returned to the attaching driver. This function takes a reference on the phy device.

```
struct phy_device *phy_attach(struct net_device *dev, const char *bus_id, phy_interface_t  
                           interface)
```

attach a network device to a particular PHY device

Parameters**struct net_device *dev**

network device to attach

const char *bus_id

Bus ID of PHY device to attach

phy_interface_t interface

PHY device's interface

Description**Same as phy_attach_direct() except that a PHY bus_id**

string is passed instead of a pointer to a *struct phy_device*.

```
int phy_package_join(struct phy_device *phydev, int base_addr, size_t priv_size)
```

join a common PHY group

Parameters**struct phy_device *phydev**

target phy_device struct

int base_addr

cookie and base PHY address of PHY package for offset calculation of global register access

size_t priv_size

if non-zero allocate this amount of bytes for private data

Description

This joins a PHY group and provides a shared storage for all phydevs in this group. This is intended to be used for packages which contain more than one PHY, for example a quad PHY transceiver.

The base_addr parameter serves as cookie which has to have the same values for all members of one group and as the base PHY address of the PHY package for offset calculation to access generic registers of a PHY package. Usually, one of the PHY addresses of the different PHYs in the package provides access to these global registers. The address which is given here, will be used in the phy_package_read() and phy_package_write() convenience functions as base and added to the passed offset in those functions.

This will set the shared pointer of the phydev to the shared storage. If this is the first call for a this cookie the shared storage will be allocated. If priv_size is non-zero, the given amount of bytes are allocated for the priv member.

Returns < 1 on error, 0 on success. Esp. calling [phy_package_join\(\)](#) with the same cookie but a different priv_size is an error.

void phy_package_leave(struct phy_device *phydev)
leave a common PHY group

Parameters

struct phy_device *phydev
target phy_device struct

Description

This leaves a PHY group created by [phy_package_join\(\)](#). If this phydev was the last user of the shared data between the group, this data is freed. Resets the phydev->shared pointer to NULL.

int devm_phy_package_join(struct device *dev, struct phy_device *phydev, int base_addr, size_t priv_size)
resource managed [phy_package_join\(\)](#)

Parameters

struct device *dev
device that is registering this PHY package

struct phy_device *phydev
target phy_device struct

int base_addr
cookie and base PHY address of PHY package for offset calculation of global register access

size_t priv_size
if non-zero allocate this amount of bytes for private data

Description

Managed [phy_package_join\(\)](#). Shared storage fetched by this function, [phy_package_leave\(\)](#) is automatically called on driver detach. See [phy_package_join\(\)](#) for more information.

void phy_detach(struct phy_device *phydev)
detach a PHY device from its network device

Parameters

struct phy_device *phydev
target phy_device struct

Description

This detaches the phy device from its network device and the phy driver, and drops the reference count taken in [phy_attach_direct\(\)](#).

int phy_reset_after_clk_enable(struct phy_device *phydev)
perform a PHY reset if needed

Parameters

struct phy_device *phydev
target phy_device struct

Description**Some PHYs are known to need a reset after their refclk was**

enabled. This function evaluates the flags and perform the reset if it's needed. Returns < 0 on error, 0 if the phy wasn't reset and 1 if the phy was reset.

int **genphy_config_eee_advert**(struct *phy_device* *phydev)
disable unwanted eee mode advertisement

Parameters

struct phy_device *phydev
target phy_device struct

Description**Writes MDIO_AN_EEE_ADV after disabling unsupported energy**

efficient ethernet modes. Returns 0 if the PHY's advertisement hasn't changed, and 1 if it has changed.

int **genphy_setup_forced**(struct *phy_device* *phydev)
configures/forces speed/duplex from **phydev**

Parameters

struct phy_device *phydev
target phy_device struct

Description**Configures MII_BMCR to force speed/duplex**

to the values in phydev. Assumes that the values are valid. Please see [*phy_sanitize_settings\(\)*](#).

int **genphy_restart_aneg**(struct *phy_device* *phydev)
Enable and Restart Autonegotiation

Parameters

struct phy_device *phydev
target phy_device struct

int **genphy_check_and_restart_aneg**(struct *phy_device* *phydev, bool restart)
Enable and restart auto-negotiation

Parameters

struct phy_device *phydev
target phy_device struct

bool restart
whether aneg restart is requested

Description

Check, and restart auto-negotiation if needed.

```
int __genphy_config_aneg(struct phy_device *phydev, bool changed)
    restart auto-negotiation or write BMCR
```

Parameters

struct phy_device *phydev
target phy_device struct

bool changed
whether autoneg is requested

Description**If auto-negotiation is enabled, we configure the**

advertising, and then restart auto-negotiation. If it is not enabled, then we write the BMCR.

```
int genphy_c37_config_aneg(struct phy_device *phydev)
    restart auto-negotiation or write BMCR
```

Parameters

struct phy_device *phydev
target phy_device struct

Description**If auto-negotiation is enabled, we configure the**

advertising, and then restart auto-negotiation. If it is not enabled, then we write the BMCR.
This function is intended for use with Clause 37 1000Base-X mode.

```
int genphy_aneg_done(struct phy_device *phydev)
    return auto-negotiation status
```

Parameters

struct phy_device *phydev
target phy_device struct

Description**Reads the status register and returns 0 either if**

auto-negotiation is incomplete, or if there was an error. Returns BMSR_ANEGCOMPLETE
if auto-negotiation is done.

```
int genphy_update_link(struct phy_device *phydev)
    update link status in phydev
```

Parameters

struct phy_device *phydev
target phy_device struct

Description**Update the value in phydev->link to reflect the**

current link value. In order to do this, we need to read the status register twice, keeping
the second value.

```
int genphy_read_status_fixed(struct phy_device *phydev)
    read the link parameters for !aneg mode
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

Description

Read the current duplex and speed state for a PHY operating with autonegotiation disabled.

```
int genphy_read_status(struct phy_device *phydev)
    check the link status and update current link state
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

Description**Check the link, then figure out the current state**

by comparing what we advertise with what the link partner advertises. Start by checking the gigabit possibilities, then move on to 10/100.

```
int genphy_c37_read_status(struct phy_device *phydev)
    check the link status and update current link state
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

Description**Check the link, then figure out the current state**

by comparing what we advertise with what the link partner advertises. This function is for Clause 37 1000Base-X mode.

```
int genphy_soft_reset(struct phy_device *phydev)
    software reset the PHY via BMCR_RESET bit
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

Description

Perform a software PHY reset using the standard BMCR_RESET bit and poll for the reset bit to be cleared.

Return

0 on success, < 0 on failure

```
int genphy_read_abilities(struct phy_device *phydev)
    read PHY abilities from Clause 22 registers
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

Description

Reads the PHY's abilities and populates phydev->supported accordingly.

Return

0 on success, < 0 on failure

```
void phy_remove_link_mode(struct phy_device *phydev, u32 link_mode)
```

Remove a supported link mode

Parameters

struct phy_device *phydev

phy_device structure to remove link mode from

u32 link_mode

Link mode to be removed

Description

Some MACs don't support all link modes which the PHY does. e.g. a 1G MAC often does not support 1000Half. Add a helper to remove a link mode.

```
void phy_advertise_supported(struct phy_device *phydev)
```

Advertise all supported modes

Parameters

struct phy_device *phydev

target phy_device struct

Description

Called to advertise all supported modes, doesn't touch pause mode advertising.

```
void phy_support_sym_pause(struct phy_device *phydev)
```

Enable support of symmetrical pause

Parameters

struct phy_device *phydev

target phy_device struct

Description

Called by the MAC to indicate it supports symmetrical Pause, but not asym pause.

```
void phy_support_asym_pause(struct phy_device *phydev)
```

Enable support of asym pause

Parameters

struct phy_device *phydev

target phy_device struct

Description

Called by the MAC to indicate it supports Asym Pause.

```
void phy_set_sym_pause(struct phy_device *phydev, bool rx, bool tx, bool autoneg)
```

Configure symmetric Pause

Parameters

```
struct phy_device *phydev
```

target phy_device struct

```
bool rx
```

Receiver Pause is supported

```
bool tx
```

Transmit Pause is supported

```
bool autoneg
```

Auto neg should be used

Description

Configure advertised Pause support depending on if receiver pause and pause auto neg is supported. Generally called from the set_pauseparam .ndo.

```
void phy_set_asym_pause(struct phy_device *phydev, bool rx, bool tx)
```

Configure Pause and Asym Pause

Parameters

```
struct phy_device *phydev
```

target phy_device struct

```
bool rx
```

Receiver Pause is supported

```
bool tx
```

Transmit Pause is supported

Description

Configure advertised Pause support depending on if transmit and receiver pause is supported. If there has been a change in advertising, trigger a new autoneg. Generally called from the set_pauseparam .ndo.

```
bool phy_validate_pause(struct phy_device *phydev, struct ethtool_pauseparam *pp)
```

Test if the PHY/MAC support the pause configuration

Parameters

```
struct phy_device *phydev
```

phy_device struct

```
struct ethtool_pauseparam *pp
```

requested pause configuration

Description

Test if the PHY/MAC combination supports the Pause configuration the user is requesting. Returns True if it is supported, false otherwise.

```
void phy_get_pause(struct phy_device *phydev, bool *tx_pause, bool *rx_pause)
```

resolve negotiated pause modes

Parameters**struct phy_device *phydev**

phy_device struct

bool *tx_pause

pointer to bool to indicate whether transmit pause should be enabled.

bool *rx_pause

pointer to bool to indicate whether receive pause should be enabled.

Description

Resolve and return the flow control modes according to the negotiation result. This includes checking that we are operating in full duplex mode. See linkmode_resolve_pause() for further details.

s32 **phy_get_internal_delay**(struct *phy_device* *phydev, struct device *dev, const int *delay_values, int size, bool is_rx)

returns the index of the internal delay

Parameters**struct phy_device *phydev**

phy_device struct

struct device *dev

pointer to the devices device struct

const int *delay_values

array of delays the PHY supports

int size

the size of the delay array

bool is_rx

boolean to indicate to get the rx internal delay

Description

Returns the index within the array of internal delay passed in. If the device property is not present then the interface type is checked if the interface defines use of internal delay then a 1 is returned otherwise a 0 is returned. The array must be in ascending order. If PHY does not have an ascending order array then size = 0 and the value of the delay property is returned. Return -EINVAL if the delay is invalid or cannot be found.

struct mdio_device ***fwnode_mdio_find_device**(struct fwnode_handle *fwnode)

Given a fwnode, find the mdio_device

Parameters**struct fwnode_handle *fwnode**

pointer to the mdio_device's fwnode

Description

If successful, returns a pointer to the mdio_device with the embedded struct device refcount incremented by one, or NULL on failure. The caller should call put_device() on the mdio_device after its use.

`struct phy_device *fwnode_phy_find_device(struct fwnode_handle *phy_fwnode)`

For provided phy_fwnode, find phy_device.

Parameters

struct fwnode_handle *phy_fwnode

Pointer to the phy's fwnode.

Description

If successful, returns a pointer to the phy_device with the embedded struct device refcount incremented by one, or NULL on failure.

`struct phy_device *device_phy_find_device(struct device *dev)`

For the given device, get the phy_device

Parameters

struct device *dev

Pointer to the given device

Description

Refer return conditions of `fwnode_phy_find_device()`.

`struct fwnode_handle *fwnode_get_phy_node(const struct fwnode_handle *fwnode)`

Get the phy_node using the named reference.

Parameters

const struct fwnode_handle *fwnode

Pointer to fwnode from which phy_node has to be obtained.

Description

Refer return conditions of `fwnode_find_reference()`. For ACPI, only "phy-handle" is supported. Legacy DT properties "phy" and "phy-device" are not supported in ACPI. DT supports all the three named references to the phy node.

`int phy_driver_register(struct phy_driver *new_driver, struct module *owner)`

register a phy_driver with the PHY layer

Parameters

struct phy_driver *new_driver

new phy_driver to register

struct module *owner

module owning this PHY

`int get_phy_c45_ids(struct mii_bus *bus, int addr, struct phy_c45_device_ids *c45_ids)`

reads the specified addr for its 802.3-c45 IDs.

Parameters

struct mii_bus *bus

the target MII bus

int addr

PHY address on the MII bus

struct phy_c45_device_ids *c45_ids
where to store the c45 ID information.

Description

Read the PHY "devices in package". If this appears to be valid, read the PHY identifiers for each device. Return the "devices in package" and identifiers in **c45_ids**.

Returns zero on success, -EIO on bus access error, or -ENODEV if the "devices in package" is invalid.

int get_phy_c22_id(struct mii_bus *bus, int addr, u32 *phy_id)
reads the specified addr for its clause 22 ID.

Parameters

struct mii_bus *bus
the target MII bus

int addr
PHY address on the MII bus

u32 *phy_id
where to store the ID retrieved.

Description

Read the 802.3 clause 22 PHY ID from the PHY at **addr** on the **bus**, placing it in **phy_id**. Return zero on successful read and the ID is valid, -EIO on bus access error, or -ENODEV if no device responds or invalid ID.

void phy_prepare_link(struct phy_device *phydev, void (*handler)(struct net_device*))
prepares the PHY layer to monitor link status

Parameters

struct phy_device *phydev
target phy_device struct

void (*handler)(struct net_device *)
callback function for link status change notifications

Description

Tells the PHY infrastructure to handle the

gory details on monitoring link status (whether through polling or an interrupt), and to call back to the connected device driver when the link status changes. If you want to monitor your own link state, don't call this function.

int phy_poll_reset(struct phy_device *phydev)
Safely wait until a PHY reset has properly completed

Parameters

struct phy_device *phydev
The PHY device to poll

Description

According to IEEE 802.3, Section 2, Subsection 22.2.4.1.1, as

published in 2008, a PHY reset may take up to 0.5 seconds. The MII BMCR register must be polled until the BMCR_RESET bit clears.

Furthermore, any attempts to write to PHY registers may have no effect or even generate MDIO bus errors until this is complete.

Some PHYs (such as the Marvell 88E1111) don't entirely conform to the standard and do not fully reset after the BMCR_RESET bit is set, and may even *REQUIRE* a soft-reset to properly restart autonegotiation. In an effort to support such broken PHYs, this function is separate from the standard phy_init_hw() which will zero all the other bits in the BMCR and reapply all driver-specific and board-specific fixups.

```
int genphy_config_advert(struct phy_device *phydev)
    sanitize and advertise auto-negotiation parameters
```

Parameters

struct phy_device *phydev
target phy_device struct

Description**Writes MII_ADVERTISE with the appropriate values,**

after sanitizing the values to make sure we only advertise what is supported. Returns < 0 on error, 0 if the PHY's advertisement hasn't changed, and > 0 if it has changed.

```
int genphy_c37_config_advert(struct phy_device *phydev)
    sanitize and advertise auto-negotiation parameters
```

Parameters

struct phy_device *phydev
target phy_device struct

Description**Writes MII_ADVERTISE with the appropriate values,**

after sanitizing the values to make sure we only advertise what is supported. Returns < 0 on error, 0 if the PHY's advertisement hasn't changed, and > 0 if it has changed. This function is intended for Clause 37 1000Base-X mode.

```
int phy_probe(struct device *dev)
    probe and init a PHY device
```

Parameters

struct device *dev
device to probe and init

Description

Take care of setting up the phy_device structure, set the state to READY.

```
struct mii_bus *mdiobus_alloc_size(size_t size)
    allocate a mii_bus structure
```

Parameters

size_t size

extra amount of memory to allocate for private storage. If non-zero, then bus->priv is points to that memory.

Description

called by a bus driver to allocate an mii_bus structure to fill in.

```
struct mii_bus *mdio_find_bus(const char *mdio_name)
```

Given the name of a mdiobus, find the mii_bus.

Parameters**const char *mdio_name**

The name of a mdiobus.

Description

Returns a reference to the mii_bus, or NULL if none found. The embedded struct device will have its reference count incremented, and this must be put_device'ded once the bus is finished with.

```
struct mii_bus *of_mdio_find_bus(struct device_node *mdio_bus_np)
```

Given an mii_bus node, find the mii_bus.

Parameters**struct device_node *mdio_bus_np**

Pointer to the mii_bus.

Description

Returns a reference to the mii_bus, or NULL if none found. The embedded struct device will have its reference count incremented, and this must be put once the bus is finished with.

Because the association of a device_node and mii_bus is made via of_mdiobus_register(), the mii_bus cannot be found before it is registered with of_mdiobus_register().

```
struct phy_device *mdiobus_scan_c22(struct mii_bus *bus, int addr)
```

scan one address on a bus for C22 MDIO devices.

Parameters**struct mii_bus *bus**

mii_bus to scan

int addr

address on bus to scan

Description

This function scans one address on the MDIO bus, looking for devices which can be identified using a vendor/product ID in registers 2 and 3. Not all MDIO devices have such registers, but PHY devices typically do. Hence this function assumes anything found is a PHY, or can be treated as a PHY. Other MDIO devices, such as switches, will probably not be found during the scan.

```
int __mdiobus_register(struct mii_bus *bus, struct module *owner)
```

bring up all the PHYs on a given bus and attach them to bus

Parameters

```
struct mii_bus *bus
    target mii_bus
struct module *owner
    module containing bus accessor functions
```

Description

Called by a bus driver to bring up all the PHYs

on a given bus, and attach them to the bus. Drivers should use mdiobus_register() rather than `_mdiobus_register()` unless they need to pass a specific owner module. MDIO devices which are not PHYs will not be brought up by this function. They are expected to be explicitly listed in DT and instantiated by of_mdiobus_register().

Returns 0 on success or < 0 on error.

```
void mdiobus_free(struct mii_bus *bus)
    free a struct mii_bus
```

Parameters

```
struct mii_bus *bus
    mii_bus to free
```

Description

This function releases the reference to the underlying device object in the mii_bus. If this is the last reference, the mii_bus will be freed.

```
int __mdiobus_read(struct mii_bus *bus, int addr, u32 regnum)
    Unlocked version of the mdiobus_read function
```

Parameters

```
struct mii_bus *bus
    the mii_bus struct
```

```
int addr
    the phy address
```

```
u32 regnum
    register number to read
```

Description

Read a MDIO bus register. Caller must hold the mdio bus lock.

NOTE

MUST NOT be called from interrupt context.

```
int __mdiobus_write(struct mii_bus *bus, int addr, u32 regnum, u16 val)
    Unlocked version of the mdiobus_write function
```

Parameters

```
struct mii_bus *bus
    the mii_bus struct
```

```
int addr
    the phy address
```

u32 regnum

register number to write

u16 val

value to write to **regnum**

Description

Write a MDIO bus register. Caller must hold the mdio bus lock.

NOTE

MUST NOT be called from interrupt context.

```
int __mdiobus_modify_changed(struct mii_bus *bus, int addr, u32 regnum, u16 mask, u16 set)
```

Unlocked version of the mdiobus_modify function

Parameters**struct mii_bus *bus**

the mii_bus struct

int addr

the phy address

u32 regnum

register number to modify

u16 mask

bit mask of bits to clear

u16 set

bit mask of bits to set

Description

Read, modify, and if any change, write the register value back to the device. Any error returns a negative number.

NOTE

MUST NOT be called from interrupt context.

```
int __mdiobus_c45_read(struct mii_bus *bus, int addr, int devad, u32 regnum)
```

Unlocked version of the mdiobus_c45_read function

Parameters**struct mii_bus *bus**

the mii_bus struct

int addr

the phy address

int devad

device address to read

u32 regnum

register number to read

Description

Read a MDIO bus register. Caller must hold the mdio bus lock.

NOTE

MUST NOT be called from interrupt context.

`int __mdiobus_c45_write(struct mii_bus *bus, int addr, int devad, u32 regnum, u16 val)`

 Unlocked version of the mdiobus_write function

Parameters

struct mii_bus *bus

 the mii_bus struct

int addr

 the phy address

int devad

 device address to read

u32 regnum

 register number to write

u16 val

 value to write to **regnum**

Description

Write a MDIO bus register. Caller must hold the mdio bus lock.

NOTE

MUST NOT be called from interrupt context.

`int mdiobus_read_nested(struct mii_bus *bus, int addr, u32 regnum)`

 Nested version of the mdiobus_read function

Parameters

struct mii_bus *bus

 the mii_bus struct

int addr

 the phy address

u32 regnum

 register number to read

Description

In case of nested MDIO bus access avoid lockdep false positives by using mutex_lock_nested().

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

`int mdiobus_read(struct mii_bus *bus, int addr, u32 regnum)`

 Convenience function for reading a given MII mgmt register

Parameters

struct mii_bus *bus
the mii_bus struct

int addr
the phy address

u32 regnum
register number to read

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

int mdiobus_c45_read(struct mii_bus *bus, int addr, int devad, u32 regnum)

Convenience function for reading a given MII mgmt register

Parameters

struct mii_bus *bus
the mii_bus struct

int addr
the phy address

int devad
device address to read

u32 regnum
register number to read

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

int mdiobus_c45_read_nested(struct mii_bus *bus, int addr, int devad, u32 regnum)

Nested version of the mdiobus_c45_read function

Parameters

struct mii_bus *bus
the mii_bus struct

int addr
the phy address

int devad
device address to read

u32 regnum
register number to read

Description

In case of nested MDIO bus access avoid lockdep false positives by using mutex_lock_nested().

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

```
int mdiobus_write_nested(struct mii_bus *bus, int addr, u32 regnum, u16 val)
```

Nested version of the mdiobus_write function

Parameters

struct mii_bus *bus
the mii_bus struct

int addr
the phy address

u32 regnum
register number to write

u16 val
value to write to **regnum**

Description

In case of nested MDIO bus access avoid lockdep false positives by using mutex_lock_nested().

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

```
int mdiobus_write(struct mii_bus *bus, int addr, u32 regnum, u16 val)
```

Convenience function for writing a given MII mgmt register

Parameters

struct mii_bus *bus
the mii_bus struct

int addr
the phy address

u32 regnum
register number to write

u16 val
value to write to **regnum**

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

```
int mdiobus_c45_write(struct mii_bus *bus, int addr, int devad, u32 regnum, u16 val)
```

Convenience function for writing a given MII mgmt register

Parameters

struct mii_bus *bus
the mii_bus struct

int addr
the phy address

int devad
device address to read

u32 regnum

register number to write

u16 val

value to write to **regnum**

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

```
int mdiobus_c45_write_nested(struct mii_bus *bus, int addr, int devad, u32 regnum, u16 val)
```

Nested version of the mdiobus_c45_write function

Parameters**struct mii_bus *bus**

the mii_bus struct

int addr

the phy address

int devad

device address to read

u32 regnum

register number to write

u16 val

value to write to **regnum**

Description

In case of nested MDIO bus access avoid lockdep false positives by using mutex_lock_nested().

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

```
int mdiobus_modify(struct mii_bus *bus, int addr, u32 regnum, u16 mask, u16 set)
```

Convenience function for modifying a given mdio device register

Parameters**struct mii_bus *bus**

the mii_bus struct

int addr

the phy address

u32 regnum

register number to write

u16 mask

bit mask of bits to clear

u16 set

bit mask of bits to set

```
int mdiobus_c45_modify(struct mii_bus *bus, int addr, int devad, u32 regnum, u16 mask, u16 set)
```

Convenience function for modifying a given mdio device register

Parameters

struct mii_bus *bus

the mii_bus struct

int addr

the phy address

int devad

device address to read

u32 regnum

register number to write

u16 mask

bit mask of bits to clear

u16 set

bit mask of bits to set

```
int mdiobus_modify_changed(struct mii_bus *bus, int addr, u32 regnum, u16 mask, u16 set)
```

Convenience function for modifying a given mdio device register and returning if it changed

Parameters

struct mii_bus *bus

the mii_bus struct

int addr

the phy address

u32 regnum

register number to write

u16 mask

bit mask of bits to clear

u16 set

bit mask of bits to set

```
int mdiobus_c45_modify_changed(struct mii_bus *bus, int addr, int devad, u32 regnum, u16 mask, u16 set)
```

Convenience function for modifying a given mdio device register and returning if it changed

Parameters

struct mii_bus *bus

the mii_bus struct

int addr

the phy address

int devad

device address to read

u32 regnum
register number to write

u16 mask
bit mask of bits to clear

u16 set
bit mask of bits to set

void mdiobus_release(struct device *d)
mii_bus device release callback

Parameters

struct device *d
the target struct device that contains the mii_bus

Description

called when the last reference to an mii_bus is dropped, to free the underlying memory.

int mdiobus_create_device(struct mii_bus *bus, struct mdio_board_info *bi)
create a full MDIO device given a mdio_board_info structure

Parameters

struct mii_bus *bus
MDIO bus to create the devices on

struct mdio_board_info *bi
mdio_board_info structure describing the devices

Description

Returns 0 on success or < 0 on error.

struct phy_device *mdiobus_scan_c45(struct mii_bus *bus, int addr)
scan one address on a bus for C45 MDIO devices.

Parameters

struct mii_bus *bus
mii_bus to scan

int addr
address on bus to scan

Description

This function scans one address on the MDIO bus, looking for devices which can be identified using a vendor/product ID in registers 2 and 3. Not all MDIO devices have such registers, but PHY devices typically do. Hence this function assumes anything found is a PHY, or can be treated as a PHY. Other MDIO devices, such as switches, will probably not be found during the scan.

int __mdiobus_c45_modify_changed(struct mii_bus *bus, int addr, int devad, u32 regnum, u16 mask, u16 set)

Unlocked version of the mdiobus_modify function

Parameters

```
struct mii_bus *bus
    the mii_bus struct

int addr
    the phy address

int devad
    device address to read

u32 regnum
    register number to modify

u16 mask
    bit mask of bits to clear

u16 set
    bit mask of bits to set
```

Description

Read, modify, and if any change, write the register value back to the device. Any error returns a negative number.

NOTE

MUST NOT be called from interrupt context.

```
int mdio_bus_match(struct device *dev, struct device_driver *drv)
    determine if given MDIO driver supports the given MDIO device
```

Parameters

```
struct device *dev
    target MDIO device

struct device_driver *drv
    given MDIO driver
```

Description

Given a MDIO device, and a MDIO driver, return 1 if

the driver supports the device. Otherwise, return 0. This may require calling the devices own match function, since different classes of MDIO devices have different match criteria.

13.2.3 PHYLINK

PHYLINK interfaces traditional network drivers with PHYLIB, fixed-links, and SFF modules (eg, hot-pluggable SFP) that may contain PHYs. PHYLINK provides management of the link state and link modes.

```
struct phylink_link_state
    link state structure
```

Definition:

```
struct phylink_link_state {
    unsigned long advertising[BITS_TO_LONGS(__ETHTOOL_LINK_MODE_MASK_NBITS)];
    unsigned long lp_advertising[BITS_TO_LONGS(__ETHTOOL_LINK_MODE_MASK_
-NBITS)];
```

```

phy_interface_t interface;
int speed;
int duplex;
int pause;
int rate_matching;
unsigned int link:1;
unsigned int an_complete:1;
};

```

Members**advertising**

ethtool bitmask containing advertised link modes

lp_advertising

ethtool bitmask containing link partner advertised link modes

interface

link *typedef phy_interface_t* mode

speed

link speed, one of the SPEED_* constants.

duplex

link duplex mode, one of DUPLEX_* constants.

pause

link pause state, described by MLO_PAUSE_* constants.

rate_matching

rate matching being performed, one of the RATE_MATCH_* constants. If rate matching is taking place, then the speed/duplex of the medium link mode (**speed** and **duplex**) and the speed/duplex of the phy interface mode (**interface**) are different.

link

true if the link is up.

an_complete

true if autonegotiation has completed.

struct phylink_config

PHYLINK configuration structure

Definition:

```

struct phylink_config {
    struct device *dev;
    enum phylink_op_type type;
    bool poll_fixed_state;
    bool mac_managed_pm;
    bool ovr_an_inband;
    void (*get_fixed_state)(struct phylink_config *config, struct phylink_link_
    -state *state);
    unsigned long supported_interfaces[BITS_TO_LONGS(PHY_INTERFACE_MODE_MAX)];
    unsigned long mac_capabilities;
};

```

Members**dev**

a pointer to a struct device associated with the MAC

type

operation type of PHYLINK instance

poll_fixed_state

if true, starts link_poll, if MAC link is at MLO_AN_FIXED mode.

mac_managed_pm

if true, indicate the MAC driver is responsible for PHY PM.

ovr_an_inband

if true, override PCS to MLO_AN_INBAND

get_fixed_state

callback to execute to determine the fixed link state, if MAC link is at MLO_AN_FIXED mode.

supported_interfaces

bitmap describing which PHY_INTERFACE_MODE_xxx are supported by the MAC/PCS.

mac_capabilities

MAC pause/speed/duplex capabilities.

struct phylink_mac_ops

MAC operations structure.

Definition:

```
struct phylink_mac_ops {
    unsigned long (*mac_get_caps)(struct phylink_config *config, phy_interface_t interface);
    struct phylink_pcs *(*mac_select_pcs)(struct phylink_config *config, phy_interface_t interface);
    int (*mac_prepare)(struct phylink_config *config, unsigned int mode, phy_interface_t iface);
    void (*mac_config)(struct phylink_config *config, unsigned int mode, const struct phylink_link_state *state);
    int (*mac_finish)(struct phylink_config *config, unsigned int mode, phy_interface_t iface);
    void (*mac_link_down)(struct phylink_config *config, unsigned int mode, phy_interface_t interface);
    void (*mac_link_up)(struct phylink_config *config, struct phy_device *phy, unsigned int mode, phy_interface_t interface, int speed, int duplex, bool tx_pause, bool rx_pause);
};
```

Members**mac_get_caps**

Get MAC capabilities for interface mode.

mac_select_pcs

Select a PCS for the interface mode.

mac_prepare

prepare for a major reconfiguration of the interface.

mac_config

configure the MAC for the selected mode and state.

mac_finish

finish a major reconfiguration of the interface.

mac_link_down

take the link down.

mac_link_up

allow the link to come up.

Description

The individual methods are described more fully below.

`unsigned long mac_get_caps(struct phylink_config *config, phy_interface_t interface)`

Get MAC capabilities for interface mode.

Parameters**struct phylink_config *config**

a pointer to a `struct phylink_config`.

phy_interface_t interface

PHY interface mode.

Description

Optional method. When not provided, config->mac_capabilities will be used. When implemented, this returns the MAC capabilities for the specified interface mode where there is some special handling required by the MAC driver (e.g. not supporting half-duplex in certain interface modes.)

`struct phylink_pcs *mac_select_pcs(struct phylink_config *config, phy_interface_t interface)`

Select a PCS for the interface mode.

Parameters**struct phylink_config *config**

a pointer to a `struct phylink_config`.

phy_interface_t interface

PHY interface mode for PCS

Description

Return the `struct phylink_pcs` for the specified interface mode, or NULL if none is required, or an error pointer on error.

This must not modify any state. It is used to query which PCS should be used. Phylink will use this during validation to ensure that the configuration is valid, and when setting a configuration to internally set the PCS that will be used.

`int mac_prepare(struct phylink_config *config, unsigned int mode, phy_interface_t iface)`

prepare to change the PHY interface mode

Parameters

struct phylink_config *config
a pointer to a *struct phylink_config*.
unsigned int mode
one of MLO_AN_FIXED, MLO_AN_PHY, MLO_AN_INBAND.
phy_interface_t iface
interface mode to switch to

Description

phylink will call this method at the beginning of a full initialisation of the link, which includes changing the interface mode or at initial startup time. It may be called for the current mode. The MAC driver should perform whatever actions are required, e.g. disabling the Serdes PHY.

This will be the first call in the sequence: - *mac_prepare()* - *mac_config()* - *pcs_config()* - possible *pcs_an_restart()* - *mac_finish()*

Returns zero on success, or negative errno on failure which will be reported to the kernel log.

void mac_config(struct phylink_config *config, unsigned int mode, const struct phylink_link_state *state)

configure the MAC for the selected mode and state

Parameters

struct phylink_config *config
a pointer to a *struct phylink_config*.
unsigned int mode
one of MLO_AN_FIXED, MLO_AN_PHY, MLO_AN_INBAND.
const struct phylink_link_state *state
a pointer to a *struct phylink_link_state*.

Description

Note - not all members of **state** are valid. In particular, **state->lp_advertising**, **state->link**, **state->an_complete** are never guaranteed to be correct, and so any *mac_config()* implementation must never reference these fields.

This will only be called to reconfigure the MAC for a "major" change in e.g. interface mode. It will not be called for changes in speed, duplex or pause modes or to change the in-band advertisement.

In all negotiation modes, as defined by **mode**, **state->pause** indicates the pause settings which should be applied as follows. If MLO_PAUSE_AN is not set, MLO_PAUSE_TX and MLO_PAUSE_RX indicate whether the MAC should send pause frames and/or act on received pause frames respectively. Otherwise, the results of in-band negotiation/status from the MAC PCS should be used to control the MAC pause mode settings.

The action performed depends on the currently selected mode:

MLO_AN_FIXED, MLO_AN_PHY:

Configure for non-inband negotiation mode, where the link settings are completely communicated via *mac_link_up()*. The physical link protocol from the MAC is specified by **state->interface**.

state->advertising may be used, but is not required.

Older drivers (prior to the `mac_link_up()` change) may use `state->speed`, `state->duplex` and `state->pause` to configure the MAC, but this is deprecated; such drivers should be converted to use `mac_link_up()`.

Other members of `state` must be ignored.

Valid state members: interface, advertising. Deprecated state members: speed, duplex, pause.

MLO_AN_INBAND:

place the link in an inband negotiation mode (such as 802.3z 1000base-X or Cisco SGMII mode depending on the `state->interface` mode). In both cases, link state management (whether the link is up or not) is performed by the MAC, and reported via the `pcs_get_state()` callback. Changes in link state must be made by calling `phylink_mac_change()`.

Interface mode specific details are mentioned below.

If in 802.3z mode, the link speed is fixed, dependent on the `state->interface`. Duplex and pause modes are negotiated via the in-band configuration word. Advertised pause modes are set according to the `state->an_enabled` and `state->advertising` flags. Beware of MACs which only support full duplex at gigabit and higher speeds.

If in Cisco SGMII mode, the link speed and duplex mode are passed in the serial bitstream 16-bit configuration word, and the MAC should be configured to read these bits and acknowledge the configuration word. Nothing is advertised by the MAC. The MAC is responsible for reading the configuration word and configuring itself accordingly.

Valid state members: interface, an_enabled, pause, advertising.

Implementations are expected to update the MAC to reflect the requested settings - i.o.w., if nothing has changed between two calls, no action is expected. If only flow control settings have changed, flow control should be updated *without* taking the link down. This "update" behaviour is critical to avoid bouncing the link up status.

```
int mac_finish(struct phylink_config *config, unsigned int mode, phy_interface_t iface)
    finish a to change the PHY interface mode
```

Parameters

`struct phylink_config *config`
a pointer to a `struct phylink_config`.

`unsigned int mode`
one of MLO_AN_FIXED, MLO_AN_PHY, MLO_AN_INBAND.

`phy_interface_t iface`
interface mode to switch to

Description

phylink will call this if it called `mac_prepare()` to allow the MAC to complete any necessary steps after the MAC and PCS have been configured for the `mode` and `iface`. E.g. a MAC driver may wish to re-enable the Serdes PHY here if it was previously disabled by `mac_prepare()`.

Returns zero on success, or negative errno on failure which will be reported to the kernel log.

```
void mac_link_down(struct phylink_config *config, unsigned int mode, phy_interface_t
    interface)
```

take the link down

Parameters

```
struct phylink_config *config  
    a pointer to a struct phylink_config.  
  
unsigned int mode  
    link autonegotiation mode  
  
phy_interface_t interface  
    link typedef phy_interface_t mode
```

Description

If **mode** is not an in-band negotiation mode (as defined by `phylink_autoneg_inband()`), force the link down and disable any Energy Efficient Ethernet MAC configuration. Interface type selection must be done in `mac_config()`.

```
void mac_link_up(struct phylink_config *config, struct phy_device *phy, unsigned int mode,  
                  phy_interface_t interface, int speed, int duplex, bool tx_pause, bool  
                  rx_pause)
```

allow the link to come up

Parameters

```
struct phylink_config *config  
    a pointer to a struct phylink_config.  
  
struct phy_device *phy  
    any attached phy  
  
unsigned int mode  
    link autonegotiation mode  
  
phy_interface_t interface  
    link typedef phy_interface_t mode  
  
int speed  
    link speed  
  
int duplex  
    link duplex  
  
bool tx_pause  
    link transmit pause enablement status  
  
bool rx_pause  
    link receive pause enablement status
```

Description

Configure the MAC for an established link.

speed, **duplex**, **tx_pause** and **rx_pause** indicate the finalised link settings, and should be used to configure the MAC block appropriately where these settings are not automatically conveyed from the PCS block, or if in-band negotiation (as defined by `phylink_autoneg_inband(mode)`) is disabled.

Note that when 802.3z in-band negotiation is in use, it is possible that the user wishes to override the pause settings, and this should be allowed when considering the implementation of this method.

If in-band negotiation mode is disabled, allow the link to come up. If **phy** is non-NULL, configure Energy Efficient Ethernet by calling [phy_init_eee\(\)](#) and perform appropriate MAC configuration for EEE. Interface type selection must be done in [mac_config\(\)](#).

struct phylink_pcs

PHYLINK PCS instance

Definition:

```
struct phylink_pcs {
    const struct phylink_pcs_ops *ops;
    struct phylink *phylink;
    bool neg_mode;
    bool poll;
};
```

Members

ops

a pointer to the [struct phylink_pcs_ops](#) structure

phylink

pointer to [struct phylink_config](#)

neg_mode

provide PCS neg mode via "mode" argument

poll

poll the PCS for link changes

Description

This structure is designed to be embedded within the PCS private data, and will be passed between phylink and the PCS.

The **phylink** member is private to phylink and must not be touched by the PCS driver.

struct phylink_pcs_ops

MAC PCS operations structure.

Definition:

```
struct phylink_pcs_ops {
    int (*pcs_validate)(struct phylink_pcs *pcs, unsigned long *supported, ↴
    ↵const struct phylink_link_state *state);
    int (*pcs_enable)(struct phylink_pcs *pcs);
    void (*pcs_disable)(struct phylink_pcs *pcs);
    void (*pcs_pre_config)(struct phylink_pcs *pcs, phy_interface_t interface);
    int (*pcs_post_config)(struct phylink_pcs *pcs, phy_interface_t interface);
    void (*pcs_get_state)(struct phylink_pcs *pcs, struct phylink_link_state ↴
    ↵*state);
    int (*pcs_config)(struct phylink_pcs *pcs, unsigned int neg_mode, phy_ ↴
    ↵interface_t interface, const unsigned long *advertising, bool permit_pause_to_ ↴
    ↵mac);
```

```

void (*pcs_an_restart)(struct phylink_pcs *pcs);
void (*pcs_link_up)(struct phylink_pcs *pcs, unsigned int neg_mode, phy_
↳interface_t interface, int speed, int duplex);
};

```

Members

pcs_validate

validate the link configuration.

pcs_enable

enable the PCS.

pcs_disable

disable the PCS.

pcs_pre_config

pre-mac_config method (for errata)

pcs_post_config

post-mac_config method (for arrata)

pcs_get_state

read the current MAC PCS link state from the hardware.

pcs_config

configure the MAC PCS for the selected mode and state.

pcs_an_restart

restart 802.3z BaseX autonegotiation.

pcs_link_up

program the PCS for the resolved link configuration (where necessary).

int pcs_validate(struct phylink_pcs *pcs, unsigned long *supported, const struct phylink_link_state *state)

validate the link configuration.

Parameters

struct phylink_pcs *pcs

a pointer to a *struct phylink_pcs*.

unsigned long *supported

ethtool bitmask for supported link modes.

const struct phylink_link_state *state

a const pointer to a *struct phylink_link_state*.

Description

Validate the interface mode, and advertising's autoneg bit, removing any media ethtool link modes that would not be supportable from the supported mask. Phylink will propagate the changes to the advertising mask. See the *struct phylink_mac_ops validate()* method.

Returns -EINVAL if the interface mode/autoneg mode is not supported. Returns non-zero positive if the link state can be supported.

```
int pcs_enable(struct phylink_pcs *pcs)
    enable the PCS.
```

Parameters

struct phylink_pcs *pcs
 a pointer to a *struct phylink_pcs*.

```
void pcs_disable(struct phylink_pcs *pcs)
    disable the PCS.
```

Parameters

struct phylink_pcs *pcs
 a pointer to a *struct phylink_pcs*.

```
void pcs_get_state(struct phylink_pcs *pcs, struct phylink_link_state *state)
    Read the current inband link state from the hardware
```

Parameters

struct phylink_pcs *pcs
 a pointer to a *struct phylink_pcs*.

struct phylink_link_state *state
 a pointer to a *struct phylink_link_state*.

Description

Read the current inband link state from the MAC PCS, reporting the current speed in **state->speed**, duplex mode in **state->duplex**, pause mode in **state->pause** using the ML0_PAUSE_RX and ML0_PAUSE_TX bits, negotiation completion state in **state->an_complete**, and link up state in **state->link**. If possible, **state->lp_advertising** should also be populated.

When present, this overrides *pcs_get_state()* in *struct phylink_pcs_ops*.

```
int pcs_config(struct phylink_pcs *pcs, unsigned int neg_mode, phy_interface_t interface,
               const unsigned long *advertising, bool permit_pause_to_mac)
```

Configure the PCS mode and advertisement

Parameters

struct phylink_pcs *pcs
 a pointer to a *struct phylink_pcs*.

unsigned int neg_mode
 link negotiation mode (see below)

phy_interface_t interface
 interface mode to be used

const unsigned long *advertising
 advertisement ethtool link mode mask

bool permit_pause_to_mac
 permit forwarding pause resolution to MAC

Description

Configure the PCS for the operating mode, the interface mode, and set the advertisement mask. **permit_pause_to_mac** indicates whether the hardware may forward the pause mode resolution to the MAC.

When operating in **ML0_AN_INBAND**, inband should always be enabled, otherwise inband should be disabled.

For SGMII, there is no advertisement from the MAC side, the PCS should be programmed to acknowledge the inband word from the PHY.

For 1000BASE-X, the advertisement should be programmed into the PCS.

For most 10GBASE-R, there is no advertisement.

The **neg_mode** argument should be tested via the **phylink_mode_***() family of functions, or for PCS that set **pcs->neg_mode** true, should be tested against the **PHYLINK_PCS_NEG_*** definitions.

```
void pcs_an_restart(struct phylink_pcs *pcs)
    restart 802.3z BaseX autonegotiation
```

Parameters

struct phylink_pcs *pcs
a pointer to a *struct phylink_pcs*.

Description

When PCS ops are present, this overrides **mac_an_restart()** in *struct phylink_mac_ops*.

```
void pcs_link_up(struct phylink_pcs *pcs, unsigned int neg_mode, phy_interface_t interface,
                  int speed, int duplex)
    program the PCS for the resolved link configuration
```

Parameters

struct phylink_pcs *pcs
a pointer to a *struct phylink_pcs*.

unsigned int neg_mode
link negotiation mode (see below)

phy_interface_t interface
link *typedef phy_interface_t* mode

int speed
link speed

int duplex
link duplex

Description

This call will be made just before **mac_link_up()** to inform the PCS of the resolved link parameters. For example, a PCS operating in SGMII mode without in-band AN needs to be manually configured for the link and duplex setting. Otherwise, this should be a no-op.

The **mode** argument should be tested via the **phylink_mode_***() family of functions, or for PCS that set **pcs->neg_mode** true, should be tested against the **PHYLINK_PCS_NEG_*** definitions.

```
int phylink_get_link_timer_ns(phy_interface_t interface)
    return the PCS link timer value
```

Parameters

phy_interface_t interface
 link *typedef phy_interface_t* mode

Description

Return the PCS link timer setting in nanoseconds for the PHY **interface** mode, or -EINVAL if not appropriate.

struct phylink
 internal data type for phylink

Definition:

```
struct phylink {
};
```

Members

void phylink_set_port_modes(unsigned long *mask)
 set the port type modes in the ethtool mask

Parameters

unsigned long *mask
 ethtool link mode mask

Description

Sets all the port type modes in the ethtool mask. MAC drivers should use this in their 'validate' callback.

int phylink_interface_max_speed(*phy_interface_t* interface)
 get the maximum speed of a phy interface

Parameters

phy_interface_t interface
 phy interface mode defined by *typedef phy_interface_t*

Description

Determine the maximum speed of a phy interface. This is intended to help determine the correct speed to pass to the MAC when the phy is performing rate matching.

Return

The maximum speed of **interface**

void phylink_caps_to_linkmodes(unsigned long *linkmodes, unsigned long caps)
 Convert capabilities to ethtool link modes

Parameters

unsigned long *linkmodes
 ethtool linkmode mask (must be already initialised)

unsigned long caps
bitmask of MAC capabilities

Description

Set all possible pause, speed and duplex linkmodes in **linkmodes** that are supported by the **caps**. **linkmodes** must have been initialised previously.

void phylink_limit_mac_speed(struct phylink_config *config, u32 max_speed)
limit the phylink_config to a maximum speed

Parameters

struct phylink_config *config
pointer to a *struct phylink_config*

u32 max_speed
maximum speed

Description

Mask off MAC capabilities for speeds higher than the **max_speed** parameter. Any further modifications of config.mac_capabilities will override this.

unsigned long phylink_cap_from_speed_duplex(int speed, unsigned int duplex)
Get mac capability from speed/duplex

Parameters

int speed
the speed to search for

unsigned int duplex
the duplex to search for

Description

Find the mac capability for a given speed and duplex.

Return

A mask with the mac capability patching speed and duplex, or 0 if there were no matches.

unsigned long phylink_get_capabilities(phy_interface_t interface, unsigned long mac_capabilities, int rate_matching)
get capabilities for a given MAC

Parameters

phy_interface_t interface
phy interface mode defined by *typedef phy_interface_t*

unsigned long mac_capabilities
bitmask of MAC capabilities

int rate_matching
type of rate matching being performed

Description

Get the MAC capabilities that are supported by the **interface** mode and **mac_capabilities**.

```
void phylink_validate_mask_caps(unsigned long *supported, struct phylink_link_state
                               *state, unsigned long mac_capabilities)
```

Restrict link modes based on caps

Parameters

unsigned long *supported

ethtool bitmask for supported link modes.

struct phylink_link_state *state

pointer to a *struct phylink_link_state*.

unsigned long mac_capabilities

bitmask of MAC capabilities

Description

Calculate the supported link modes based on **mac_capabilities**, and restrict **supported** and **state** based on that. Use this function if your capabiliees aren't constant, such as if they vary depending on the interface.

```
unsigned int phylink_pcs_neg_mode(unsigned int mode, phy_interface_t interface, const
                                  unsigned long *advertising)
```

helper to determine PCS inband mode

Parameters

unsigned int mode

one of MLO_AN_FIXED, MLO_AN_PHY, MLO_AN_INBAND.

phy_interface_t interface

interface mode to be used

const unsigned long *advertising

advertisment ethtool link mode mask

Description

Determines the negotiation mode to be used by the PCS, and returns one of:

- PHYLINK_PCS_NEG_NONE: interface mode does not support inband
- PHYLINK_PCS_NEG_OUTBAND: an out of band mode (e.g. reading the PHY) will be used.
- PHYLINK_PCS_NEG_INBAND_DISABLED: inband mode selected but autoneg disabled
- PHYLINK_PCS_NEG_INBAND_ENABLED: inband mode selected and autoneg enabled

Note

this is for cases where the PCS itself is involved in negotiation (e.g. Clause 37, SGMII and similar) not Clause 73.

```
struct phylink *phylink_create(struct phylink_config *config, const struct fwnode_handle
                               *fwnode, phy_interface_t iface, const struct phylink_mac_ops
                               *mac_ops)
```

create a phylink instance

Parameters

struct phylink_config *config

a pointer to the target *struct phylink_config*

const struct fwnode_handle *fwnode

a pointer to a *struct fwnode_handle* describing the network interface

phy_interface_t iface

the desired link mode defined by *typedef phy_interface_t*

const struct phylink_mac_ops *mac_ops

a pointer to a *struct phylink_mac_ops* for the MAC.

Description

Create a new phylink instance, and parse the link parameters found in **np**. This will parse in-band modes, fixed-link or SFP configuration.

Returns a pointer to a *struct phylink*, or an error-pointer value. Users must use **IS_ERR()** to check for errors from this function.

Note

the rtnl lock must not be held when calling this function.

void phylink_destroy(struct phylink *pl)

cleanup and destroy the phylink instance

Parameters

struct phylink *pl

a pointer to a *struct phylink* returned from *phylink_create()*

Description

Destroy a phylink instance. Any PHY that has been attached must have been cleaned up via *phylink_disconnect_phy()* prior to calling this function.

Note

the rtnl lock must not be held when calling this function.

bool phylink_expects_phy(struct phylink *pl)

Determine if phylink expects a phy to be attached

Parameters

struct phylink *pl

a pointer to a *struct phylink* returned from *phylink_create()*

Description

When using fixed-link mode, or in-band mode with 1000base-X or 2500base-X, no PHY is needed.

Returns true if phylink will be expecting a PHY.

int phylink_connect_phy(struct phylink *pl, struct phy_device *phy)

connect a PHY to the phylink instance

Parameters

struct phylink *pl

a pointer to a *struct phylink* returned from *phylink_create()*

struct phy_device *phy
 a pointer to a *struct phy_device*.

Description

Connect **phy** to the phylink instance specified by **pl** by calling *phy_attach_direct()*. Configure the **phy** according to the MAC driver's capabilities, start the PHYLIB state machine and enable any interrupts that the PHY supports.

This updates the phylink's ethtool supported and advertising link mode masks.

Returns 0 on success or a negative errno.

int phylink_of_phy_connect(struct phylink *pl, struct device_node *dn, u32 flags)
 connect the PHY specified in the DT mode.

Parameters

struct phylink *pl
 a pointer to a *struct phylink* returned from *phylink_create()*

struct device_node *dn
 a pointer to a *struct device_node*.

u32 flags
 PHY-specific flags to communicate to the PHY device driver

Description

Connect the phy specified in the device node **dn** to the phylink instance specified by **pl**. Actions specified in *phylink_connect_phys()* will be performed.

Returns 0 on success or a negative errno.

int phylink_fwnode_phy_connect(struct phylink *pl, const struct fwnode_handle *fwnode, u32 flags)
 connect the PHY specified in the fwnode.

Parameters

struct phylink *pl
 a pointer to a *struct phylink* returned from *phylink_create()*

const struct fwnode_handle *fwnode
 a pointer to a *struct fwnode_handle*.

u32 flags
 PHY-specific flags to communicate to the PHY device driver

Description

Connect the phy specified **fwnode** to the phylink instance specified by **pl**.

Returns 0 on success or a negative errno.

void phylink_disconnect_phy(struct phylink *pl)
 disconnect any PHY attached to the phylink instance.

Parameters

struct phylink *pl
 a pointer to a *struct phylink* returned from *phylink_create()*

Description

Disconnect any current PHY from the phylink instance described by **pl**.

void phylink_mac_change(struct phylink *pl, bool up)

 notify phylink of a change in MAC state

Parameters

struct phylink *pl

 a pointer to a *struct phylink* returned from *phylink_create()*

bool up

 indicates whether the link is currently up.

Description

The MAC driver should call this driver when the state of its link changes (eg, link failure, new negotiation results, etc.)

void phylink_pcs_change(struct phylink_pcs *pcs, bool up)

 notify phylink of a change to PCS link state

Parameters

struct phylink_pcs *pcs

 pointer to *struct phylink_pcs*

bool up

 indicates whether the link is currently up.

Description

The PCS driver should call this when the state of its link changes (e.g. link failure, new negotiation results, etc.) Note: it should not determine "up" by reading the BMSR. If in doubt about the link state at interrupt time, then pass true if *pcs_get_state()* returns the latched link-down state, otherwise pass false.

void phylink_start(struct phylink *pl)

 start a phylink instance

Parameters

struct phylink *pl

 a pointer to a *struct phylink* returned from *phylink_create()*

Description

Start the phylink instance specified by **pl**, configuring the MAC for the desired link mode(s) and negotiation style. This should be called from the network device driver's *struct net_device_ops ndo_open()* method.

void phylink_stop(struct phylink *pl)

 stop a phylink instance

Parameters

struct phylink *pl

 a pointer to a *struct phylink* returned from *phylink_create()*

Description

Stop the phylink instance specified by **pl**. This should be called from the network device driver's `struct net_device_ops ndo_stop()` method. The network device's carrier state should not be changed prior to calling this function.

This will synchronously bring down the link if the link is not already down (in other words, it will trigger a `mac_link_down()` method call.)

`void phylink_suspend(struct phylink *pl, bool mac_wol)`

handle a network device suspend event

Parameters

`struct phylink *pl`

a pointer to a `struct phylink` returned from `phylink_create()`

`bool mac_wol`

true if the MAC needs to receive packets for Wake-on-Lan

Description

Handle a network device suspend event. There are several cases:

- If Wake-on-Lan is not active, we can bring down the link between the MAC and PHY by calling `phylink_stop()`.
- If Wake-on-Lan is active, and being handled only by the PHY, we can also bring down the link between the MAC and PHY.
- If Wake-on-Lan is active, but being handled by the MAC, the MAC still needs to receive packets, so we can not bring the link down.

`void phylink_resume(struct phylink *pl)`

handle a network device resume event

Parameters

`struct phylink *pl`

a pointer to a `struct phylink` returned from `phylink_create()`

Description

Undo the effects of `phylink_suspend()`, returning the link to an operational state.

`void phylink_ethtool_get_wol(struct phylink *pl, struct ethtool_wolinfo *wol)`

get the wake on lan parameters for the PHY

Parameters

`struct phylink *pl`

a pointer to a `struct phylink` returned from `phylink_create()`

`struct ethtool_wolinfo *wol`

a pointer to `struct ethtool_wolinfo` to hold the read parameters

Description

Read the wake on lan parameters from the PHY attached to the phylink instance specified by **pl**. If no PHY is currently attached, report no support for wake on lan.

```
int phylink_ethtool_set_wol(struct phylink *pl, struct ethtool_wolinfo *wol)
    set wake on lan parameters
```

Parameters

struct phylink *pl
a pointer to a *struct phylink* returned from *phylink_create()*

struct ethtool_wolinfo *wol
a pointer to *struct ethtool_wolinfo* for the desired parameters

Description

Set the wake on lan parameters for the PHY attached to the phylink instance specified by **pl**. If no PHY is attached, returns EOPNOTSUPP error.

Returns zero on success or negative errno code.

```
int phylink_ethtool_ksettings_get(struct phylink *pl, struct ethtool_link_ksettings *kset)
    get the current link settings
```

Parameters

struct phylink *pl
a pointer to a *struct phylink* returned from *phylink_create()*

struct ethtool_link_ksettings *kset
a pointer to a *struct ethtool_link_ksettings* to hold link settings

Description

Read the current link settings for the phylink instance specified by **pl**. This will be the link settings read from the MAC, PHY or fixed link settings depending on the current negotiation mode.

```
int phylink_ethtool_ksettings_set(struct phylink *pl, const struct ethtool_link_ksettings
    *kset)
```

set the link settings

Parameters

struct phylink *pl
a pointer to a *struct phylink* returned from *phylink_create()*

const struct ethtool_link_ksettings *kset
a pointer to a *struct ethtool_link_ksettings* for the desired modes

```
int phylink_ethtool_nway_reset(struct phylink *pl)
    restart negotiation
```

Parameters

struct phylink *pl
a pointer to a *struct phylink* returned from *phylink_create()*

Description

Restart negotiation for the phylink instance specified by **pl**. This will cause any attached phy to restart negotiation with the link partner, and if the MAC is in a BaseX mode, the MAC will also be requested to restart negotiation.

Returns zero on success, or negative error code.

void phylink_ethtool_get_pauseparam(struct *phylink* *pl, struct ethtool_pauseparam *pause)

get the current pause parameters

Parameters

struct phylink *pl

a pointer to a *struct phylink* returned from *phylink_create()*

struct ethtool_pauseparam *pause

a pointer to a *struct ethtool_pauseparam*

int phylink_ethtool_set_pauseparam(struct *phylink* *pl, struct ethtool_pauseparam *pause)

set the current pause parameters

Parameters

struct phylink *pl

a pointer to a *struct phylink* returned from *phylink_create()*

struct ethtool_pauseparam *pause

a pointer to a *struct ethtool_pauseparam*

int phylink_get_eee_err(struct *phylink* *pl)

read the energy efficient ethernet error counter

Parameters

struct phylink *pl

a pointer to a *struct phylink* returned from *phylink_create()*.

Description

Read the Energy Efficient Ethernet error counter from the PHY associated with the phylink instance specified by **pl**.

Returns positive error counter value, or negative error code.

int phylink_init_eee(struct *phylink* *pl, bool clk_stop_enable)

init and check the EEE features

Parameters

struct phylink *pl

a pointer to a *struct phylink* returned from *phylink_create()*

bool clk_stop_enable

allow PHY to stop receive clock

Description

Must be called either with RTNL held or within *mac_link_up()*

int phylink_ethtool_get_eee(struct *phylink* *pl, struct ethtool_eee *eee)

read the energy efficient ethernet parameters

Parameters

struct phylink *pl

a pointer to a *struct phylink* returned from *phylink_create()*

struct ethtool_eee *eee

a pointer to a struct ethtool_eee for the read parameters

int phylink_ethtool_set_eee(struct phylink *pl, struct ethtool_eee *eee)

set the energy efficient ethernet parameters

Parameters

struct phylink *pl

a pointer to a *struct phylink* returned from *phylink_create()*

struct ethtool_eee *eee

a pointer to a struct ethtool_eee for the desired parameters

int phylink_mii_ioctl(struct phylink *pl, struct ifreq *ifr, int cmd)

generic mii ioctl interface

Parameters

struct phylink *pl

a pointer to a *struct phylink* returned from *phylink_create()*

struct ifreq *ifr

a pointer to a *struct ifreq* for socket ioctls

int cmd

ioctl cmd to execute

Description

Perform the specified MII ioctl on the PHY attached to the phylink instance specified by **pl**. If no PHY is attached, emulate the presence of the PHY.

SIOCGMIPHYS:

read register from the current PHY.

SIOCGMIREG:

read register from the specified PHY.

SIOCFSMIREG:

set a register on the specified PHY.

Return

zero on success or negative error code.

int phylink_speed_down(struct phylink *pl, bool sync)

set the non-SFP PHY to lowest speed supported by both link partners

Parameters

struct phylink *pl

a pointer to a *struct phylink* returned from *phylink_create()*

bool sync

perform action synchronously

Description

If we have a PHY that is not part of a SFP module, then set the speed as described in the *phy_speed_down()* function. Please see this function for a description of the **sync** parameter.

Returns zero if there is no PHY, otherwise as per `phy_speed_down()`.

int phylink_speed_up(struct phylink *pl)

restore the advertised speeds prior to the call to `phylink_speed_down()`

Parameters

struct phylink *pl

a pointer to a `struct phylink` returned from `phylink_create()`

Description

If we have a PHY that is not part of a SFP module, then restore the PHY speeds as per `phy_speed_up()`.

Returns zero if there is no PHY, otherwise as per `phy_speed_up()`.

void phylink_decode_usxgmii_word(struct phylink_link_state *state, uint16_t lpa)

decode the USXGMII word from a MAC PCS

Parameters

struct phylink_link_state *state

a pointer to a `struct phylink_link_state`.

uint16_t lpa

a 16 bit value which stores the USXGMII auto-negotiation word

Description

Helper for MAC PCS supporting the USXGMII protocol and the auto-negotiation code word. Decode the USXGMII code word and populate the corresponding fields (speed, duplex) into the `phylink_link_state` structure.

void phylink_decode_usgmii_word(struct phylink_link_state *state, uint16_t lpa)

decode the USGMII word from a MAC PCS

Parameters

struct phylink_link_state *state

a pointer to a `struct phylink_link_state`.

uint16_t lpa

a 16 bit value which stores the USGMII auto-negotiation word

Description

Helper for MAC PCS supporting the USGMII protocol and the auto-negotiation code word. Decode the USGMII code word and populate the corresponding fields (speed, duplex) into the `phylink_link_state` structure. The structure for this word is the same as the USXGMII word, except it only supports speeds up to 1Gbps.

void phylink_mii_c22_pcs_decode_state(struct phylink_link_state *state, u16 bmsr, u16 lpa)

Decode MAC PCS state from MII registers

Parameters

struct phylink_link_state *state

a pointer to a `struct phylink_link_state`.

u16 bmsr

The value of the MII_BMSR register

u16 lpa

The value of the MII_LPA register

Description

Helper for MAC PCS supporting the 802.3 clause 22 register set for clause 37 negotiation and/or SGMII control.

Parse the Clause 37 or Cisco SGMII link partner negotiation word into the phylink **state** structure. This is suitable to be used for implementing the [*pcs_get_state\(\)*](#) member of the [*struct phylink_pcs_ops*](#) structure if accessing **bmsr** and **lpa** cannot be done with MDIO directly.

```
void phylink_mii_c22_pcs_get_state(struct mdio_device *pcs, struct phylink_link_state *state)
```

read the MAC PCS state

Parameters

struct mdio_device *pcs

a pointer to a [*struct mdio_device*](#).

struct phylink_link_state *state

a pointer to a [*struct phylink_link_state*](#).

Description

Helper for MAC PCS supporting the 802.3 clause 22 register set for clause 37 negotiation and/or SGMII control.

Read the MAC PCS state from the MII device configured in **config** and parse the Clause 37 or Cisco SGMII link partner negotiation word into the phylink **state** structure. This is suitable to be directly plugged into the [*pcs_get_state\(\)*](#) member of the [*struct phylink_pcs_ops*](#) structure.

```
int phylink_mii_c22_pcs_encode_advertisement(phy_interface_t interface, const unsigned long *advertising)
```

configure the clause 37 PCS advertisement

Parameters

phy_interface_t interface

the PHY interface mode being configured

const unsigned long *advertising

the ethtool advertisement mask

Description

Helper for MAC PCS supporting the 802.3 clause 22 register set for clause 37 negotiation and/or SGMII control.

Encode the clause 37 PCS advertisement as specified by **interface** and **advertising**.

Return

The new value for **adv**, or -EINVAL if it should not be changed.

```
int phylink_mii_c22_pcs_config(struct mdio_device *pcs, phy_interface_t interface, const
                                unsigned long *advertising, unsigned int neg_mode)
    configure clause 22 PCS
```

Parameters

struct mdio_device *pcs
 a pointer to a **struct mdio_device**.

phy_interface_t interface
 the PHY interface mode being configured

const unsigned long *advertising
 the ethtool advertisement mask

unsigned int neg_mode
 PCS negotiation mode

Description

Configure a Clause 22 PCS PHY with the appropriate negotiation parameters for the **mode**, **interface** and **advertising** parameters. Returns negative error number on failure, zero if the advertisement has not changed, or positive if there is a change.

```
void phylink_mii_c22_pcs_an_restart(struct mdio_device *pcs)
    restart 802.3z autonegotiation
```

Parameters

struct mdio_device *pcs
 a pointer to a **struct mdio_device**.

Description

Helper for MAC PCS supporting the 802.3 clause 22 register set for clause 37 negotiation.

Restart the clause 37 negotiation with the link partner. This is suitable to be directly plugged into the **pcs_get_state()** member of the **struct phylink_pcs_ops** structure.

13.2.4 SFP support

struct sfp_bus
 internal representation of a sfp bus

Definition:

```
struct sfp_bus {  
};
```

Members

struct sfp_eeprom_id
 raw SFP module identification information

Definition:

```
struct sfp_eeprom_id {
    struct sfp_eeprom_base base;
    struct sfp_eeprom_ext ext;
};
```

Members**base**

base SFP module identification structure

ext

extended SFP module identification structure

Description

See the SFF-8472 specification and related documents for the definition of these structure members. This can be obtained from <https://www.snia.org/technology-communities/sff/specifications>

struct sfp_upstream_ops

upstream operations structure

Definition:

```
struct sfp_upstream_ops {
    void (*attach)(void *priv, struct sfp_bus *bus);
    void (*detach)(void *priv, struct sfp_bus *bus);
    int (*module_insert)(void *priv, const struct sfp_eeprom_id *id);
    void (*module_remove)(void *priv);
    int (*module_start)(void *priv);
    void (*module_stop)(void *priv);
    void (*link_down)(void *priv);
    void (*link_up)(void *priv);
    int (*connect_phy)(void *priv, struct phy_device *);
    void (*disconnect_phy)(void *priv);
};
```

Members**attach**

called when the sfp socket driver is bound to the upstream (mandatory).

detach

called when the sfp socket driver is unbound from the upstream (mandatory).

module_insert

called after a module has been detected to determine whether the module is supported for the upstream device.

module_remove

called after the module has been removed.

module_start

called after the PHY probe step

module_stop

called before the PHY is removed

link_down

called when the link is non-operational for whatever reason.

link_up

called when the link is operational.

connect_phy

called when an I2C accessible PHY has been detected on the module.

disconnect_phy

called when a module with an I2C accessible PHY has been removed.

```
int sfp_parse_port(struct sfp_bus *bus, const struct sfp_eeprom_id *id, unsigned long *support)
```

Parse the EEPROM base ID, setting the port type

Parameters**struct sfp_bus *bus**

a pointer to the *struct sfp_bus* structure for the sfp module

const struct sfp_eeprom_id *id

a pointer to the module's *struct sfp_eeprom_id*

unsigned long *support

optional pointer to an array of unsigned long for the ethtool support mask

Description

Parse the EEPROM identification given in **id**, and return one of PORT_TP, PORT_FIBRE or PORT_OTHER. If **support** is non-NULL, also set the ethtool ETHTOOL_LINK_MODE_xxx_BIT corresponding with the connector type.

If the port type is not known, returns PORT_OTHER.

```
bool sfp_may_have_phy(struct sfp_bus *bus, const struct sfp_eeprom_id *id)
```

indicate whether the module may have a PHY

Parameters**struct sfp_bus *bus**

a pointer to the *struct sfp_bus* structure for the sfp module

const struct sfp_eeprom_id *id

a pointer to the module's *struct sfp_eeprom_id*

Description

Parse the EEPROM identification given in **id**, and return whether this module may have a PHY.

```
void sfp_parse_support(struct sfp_bus *bus, const struct sfp_eeprom_id *id, unsigned long *support, unsigned long *interfaces)
```

Parse the eeprom id for supported link modes

Parameters**struct sfp_bus *bus**

a pointer to the *struct sfp_bus* structure for the sfp module

const struct sfp_eeprom_id *id

a pointer to the module's *struct sfp_eeprom_id*

unsigned long *support

pointer to an array of unsigned long for the ethtool support mask

unsigned long *interfaces

pointer to an array of unsigned long for phy interface modes mask

Description

Parse the EEPROM identification information and derive the supported ethtool link modes for the module.

phy_interface_t sfp_select_interface(*struct sfp_bus* *bus, *unsigned long* *link_modes)

Select appropriate phy_interface_t mode

Parameters**struct sfp_bus *bus**

a pointer to the *struct sfp_bus* structure for the sfp module

unsigned long *link_modes

ethtool link modes mask

Description

Derive the phy_interface_t mode for the SFP module from the link modes mask.

void sfp_bus_put(*struct sfp_bus* *bus)

put a reference on the *struct sfp_bus*

Parameters**struct sfp_bus *bus**

the *struct sfp_bus* found via *sfp_bus_find_fwnode()*

Description

Put a reference on the *struct sfp_bus* and free the underlying structure if this was the last reference.

int sfp_get_module_info(*struct sfp_bus* *bus, *struct ethtool_modinfo* *modinfo)

Get the ethtool_modinfo for a SFP module

Parameters**struct sfp_bus *bus**

a pointer to the *struct sfp_bus* structure for the sfp module

struct ethtool_modinfo *modinfo

a *struct ethtool_modinfo*

Description

Fill in the type and eeprom_len parameters in **modinfo** for a module on the sfp bus specified by **bus**.

Returns 0 on success or a negative errno number.

int sfp_get_module_eeprom(*struct sfp_bus* *bus, *struct ethtool_eeprom* *ee, *u8* *data)

Read the SFP module EEPROM

Parameters

struct sfp_bus *bus
 a pointer to the *struct sfp_bus* structure for the sfp module

struct ethtool_eeprom *ee
 a struct *ethtool_eeprom*

u8 *data
 buffer to contain the EEPROM data (must be at least **ee->len** bytes)

Description

Read the EEPROM as specified by the supplied **ee**. See the documentation for *struct ethtool_eeprom* for the region to be read.

Returns 0 on success or a negative errno number.

```
int sfp_get_module_eeprom_by_page(struct sfp_bus *bus, const struct
                                  ethtool_module_eeprom *page, struct netlink_ext_ack
                                  *extack)
```

Read a page from the SFP module EEPROM

Parameters

struct sfp_bus *bus
 a pointer to the *struct sfp_bus* structure for the sfp module

const struct ethtool_module_eeprom *page
 a struct *ethtool_module_eeprom*

struct netlink_ext_ack *extack
 extack for reporting problems

Description

Read an EEPROM page as specified by the supplied **page**. See the documentation for *struct ethtool_module_eeprom* for the page to be read.

Returns 0 on success or a negative errno number. More error information might be provided via extack

```
void sfp_upstream_start(struct sfp_bus *bus)
    Inform the SFP that the network device is up
```

Parameters

struct sfp_bus *bus
 a pointer to the *struct sfp_bus* structure for the sfp module

Description

Inform the SFP socket that the network device is now up, so that the module can be enabled by allowing TX_DISABLE to be deasserted. This should be called from the network device driver's *struct net_device_ops ndo_open()* method.

```
void sfp_upstream_stop(struct sfp_bus *bus)
    Inform the SFP that the network device is down
```

Parameters

struct sfp_bus *bus
 a pointer to the *struct sfp_bus* structure for the sfp module

Description

Inform the SFP socket that the network device is now up, so that the module can be disabled by asserting TX_DISABLE, disabling the laser in optical modules. This should be called from the network device driver's `struct net_device_ops ndo_stop()` method.

```
void sfp_upstream_set_signal_rate(struct sfp_bus *bus, unsigned int rate_kbd)
```

 set data signalling rate

Parameters

struct sfp_bus *bus

 a pointer to the `struct sfp_bus` structure for the sfp module

unsigned int rate_kbd

 signalling rate in units of 1000 baud

Description

Configure the rate select settings on the SFP module for the signalling rate (not the same as the data rate).

Locks that may be held:

Phylink's state_mutex rtnl lock SFP's sm_mutex

```
struct sfp_bus *sfp_bus_find_fwnode(const struct fwnode_handle *fwnode)
```

 parse and locate the SFP bus from fwnode

Parameters

const struct fwnode_handle *fwnode

 firmware node for the parent device (MAC or PHY)

Description

Parse the parent device's firmware node for a SFP bus, and locate the `sfp_bus` structure, incrementing its reference count. This must be put via `sfp_bus_put()` when done.

Return

- on success, a pointer to the `sfp_bus` structure,
- NULL if no SFP is specified,
- on failure, an error pointer value:
- corresponding to the errors detailed for `fwnode_property_get_reference_args()`.
- -ENOMEM if we failed to allocate the bus.
- an error from the upstream's `connect_phy()` method.

```
int sfp_bus_add_upstream(struct sfp_bus *bus, void *upstream, const struct  
                          sfp_upstream_ops *ops)
```

 parse and register the neighbouring device

Parameters

struct sfp_bus *bus

 the `struct sfp_bus` found via `sfp_bus_find_fwnode()`

void *upstream

 the upstream private data

```
const struct sfp_upstream_ops *ops
    the upstream's struct sfp_upstream_ops
```

Description

Add upstream driver for the SFP bus, and if the bus is complete, register the SFP bus using `sfp_register_upstream()`. This takes a reference on the bus, so it is safe to put the bus after this call.

Return

- on success, a pointer to the `sfp_bus` structure,
- `NULL` if no SFP is specified,
- on failure, an error pointer value:
- corresponding to the errors detailed for `fwnode_property_get_reference_args()`.
- `-ENOMEM` if we failed to allocate the bus.
- an error from the upstream's `connect_phy()` method.

```
void sfp_bus_del_upstream(struct sfp_bus *bus)
```

Delete a sfp bus

Parameters

```
struct sfp_bus *bus
```

a pointer to the `struct sfp_bus` structure for the sfp module

Description

Delete a previously registered upstream connection for the SFP module. `bus` should have been added by `sfp_bus_add_upstream()`.

MSG_ZEROCOPY

14.1 Intro

The MSG_ZEROCOPY flag enables copy avoidance for socket send calls. The feature is currently implemented for TCP, UDP and VSOCK (with virtio transport) sockets.

14.1.1 Opportunity and Caveats

Copying large buffers between user process and kernel can be expensive. Linux supports various interfaces that eschew copying, such as sendfile and splice. The MSG_ZEROCOPY flag extends the underlying copy avoidance mechanism to common socket send calls.

Copy avoidance is not a free lunch. As implemented, with page pinning, it replaces per byte copy cost with page accounting and completion notification overhead. As a result, MSG_ZEROCOPY is generally only effective at writes over around 10 KB.

Page pinning also changes system call semantics. It temporarily shares the buffer between process and network stack. Unlike with copying, the process cannot immediately overwrite the buffer after system call return without possibly modifying the data in flight. Kernel integrity is not affected, but a buggy program can possibly corrupt its own data stream.

The kernel returns a notification when it is safe to modify data. Converting an existing application to MSG_ZEROCOPY is not always as trivial as just passing the flag, then.

14.1.2 More Info

Much of this document was derived from a longer paper presented at netdev 2.1. For more in-depth information see that paper and talk, the excellent reporting over at LWN.net or read the original code.

paper, slides, video
<https://netdevconf.org/2.1/session.html?debruijn>

LWN article
<https://lwn.net/Articles/726917/>

patchset
[PATCH net-next v4 0/9] socket sendmsg MSG_ZEROCOPY <https://lore.kernel.org/netdev/20170803202945.70750-1-willemdebruijn.kernel@gmail.com>

14.2 Interface

Passing the MSG_ZEROCOPY flag is the most obvious step to enable copy avoidance, but not the only one.

14.2.1 Socket Setup

The kernel is permissive when applications pass undefined flags to the send system call. By default it simply ignores these. To avoid enabling copy avoidance mode for legacy processes that accidentally already pass this flag, a process must first signal intent by setting a socket option:

```
if (setsockopt(fd, SOL_SOCKET, SO_ZEROCOPY, &one, sizeof(one)))
    error(1, errno, "setsockopt zerocopy");
```

14.2.2 Transmission

The change to send (or sendto, sendmsg, sendmmsg) itself is trivial. Pass the new flag.

```
ret = send(fd, buf, sizeof(buf), MSG_ZEROCOPY);
```

A zero-copy failure will return -1 with errno ENOBUFS. This happens if the socket exceeds its optmem limit or the user exceeds their ulimit on locked pages.

Mixing copy avoidance and copying

Many workloads have a mixture of large and small buffers. Because copy avoidance is more expensive than copying for small packets, the feature is implemented as a flag. It is safe to mix calls with the flag with those without.

14.2.3 Notifications

The kernel has to notify the process when it is safe to reuse a previously passed buffer. It queues completion notifications on the socket error queue, akin to the transmit timestamping interface.

The notification itself is a simple scalar value. Each socket maintains an internal unsigned 32-bit counter. Each send call with MSG_ZEROCOPY that successfully sends data increments the counter. The counter is not incremented on failure or if called with length zero. The counter counts system call invocations, not bytes. It wraps after UINT_MAX calls.

Notification Reception

The below snippet demonstrates the API. In the simplest case, each send syscall is followed by a poll and recvmsg on the error queue.

Reading from the error queue is always a non-blocking operation. The poll call is there to block until an error is outstanding. It will set POLLERR in its output flags. That flag does not have to be set in the events field. Errors are signaled unconditionally.

```
pfds.fd = fd;
pfds.events = 0;
if (poll(&pfds, 1, -1) != 1 || pfd.revents & POLLERR == 0)
    error(1, errno, "poll");

ret = recvmsg(fd, &msg, MSG_ERRQUEUE);
if (ret == -1)
    error(1, errno, "recvmsg");

read_notification(msg);
```

The example is for demonstration purpose only. In practice, it is more efficient to not wait for notifications, but read without blocking every couple of send calls.

Notifications can be processed out of order with other operations on the socket. A socket that has an error queued would normally block other operations until the error is read. Zerocopy notifications have a zero error code, however, to not block send and recv calls.

Notification Batching

Multiple outstanding packets can be read at once using the recvmmmsg call. This is often not needed. In each message the kernel returns not a single value, but a range. It coalesces consecutive notifications while one is outstanding for reception on the error queue.

When a new notification is about to be queued, it checks whether the new value extends the range of the notification at the tail of the queue. If so, it drops the new notification packet and instead increases the range upper value of the outstanding notification.

For protocols that acknowledge data in-order, like TCP, each notification can be squashed into the previous one, so that no more than one notification is outstanding at any one point.

Ordered delivery is the common case, but not guaranteed. Notifications may arrive out of order on retransmission and socket teardown.

Notification Parsing

The below snippet demonstrates how to parse the control message: the read_notification() call in the previous snippet. A notification is encoded in the standard error format, sock_extended_err.

The level and type fields in the control data are protocol family specific, IP_RECVERR or IPV6_RECVERR (for TCP or UDP socket). For VSOCK socket, cmsg_level will be SOL_VSOCK and cmsg_type will be VSOCK_RECVERR.

Error origin is the new type SO_EE_ORIGIN_ZEROCOPY. ee_errno is zero, as explained before, to avoid blocking read and write system calls on the socket.

The 32-bit notification range is encoded as [ee_info, ee_data]. This range is inclusive. Other fields in the struct must be treated as undefined, bar for ee_code, as discussed below.

```
struct sock_extended_err *serr;
struct cmsghdr *cm;

cm = CMSG_FIRSTHDR(msg);
if (cm->cmsg_level != SOL_IP &&
    cm->cmsg_type != IP_RECVERR)
    error(1, 0, "cmsg");

serr = (void *) CMSG_DATA(cm);
if (serr->ee_errno != 0 ||
    serr->ee_origin != SO_EE_ORIGIN_ZEROCOPY)
    error(1, 0, "serr");

printf("completed: %u..%u\n", serr->ee_info, serr->ee_data);
```

Deferred copies

Passing flag MSG_ZEROCOPY is a hint to the kernel to apply copy avoidance, and a contract that the kernel will queue a completion notification. It is not a guarantee that the copy is elided.

Copy avoidance is not always feasible. Devices that do not support scatter-gather I/O cannot send packets made up of kernel generated protocol headers plus zerocopy user data. A packet may need to be converted to a private copy of data deep in the stack, say to compute a checksum.

In all these cases, the kernel returns a completion notification when it releases its hold on the shared pages. That notification may arrive before the (copied) data is fully transmitted. A zerocopy completion notification is not a transmit completion notification, therefore.

Deferred copies can be more expensive than a copy immediately in the system call, if the data is no longer warm in the cache. The process also incurs notification processing cost for no benefit. For this reason, the kernel signals if data was completed with a copy, by setting flag SO_EE_CODE_ZEROCOPY_COPIED in field ee_code on return. A process may use this signal to stop passing flag MSG_ZEROCOPY on subsequent requests on the same socket.

14.3 Implementation

14.3.1 Loopback

For TCP and UDP: Data sent to local sockets can be queued indefinitely if the receive process does not read its socket. Unbound notification latency is not acceptable. For this reason all packets generated with MSG_ZEROCOPY that are looped to a local socket will incur a deferred copy. This includes looping onto packet sockets (e.g., tcpdump) and tun devices.

For VSOCK: Data path sent to local sockets is the same as for non-local sockets.

14.4 Testing

More realistic example code can be found in the kernel source under tools/testing/selftests/net/msg_zerocopy.c.

Be cognizant of the loopback constraint. The test can be run between a pair of hosts. But if run between a local pair of processes, for instance when run with msg_zerocopy.sh between a veth pair across namespaces, the test will not show any improvement. For testing, the loopback restriction can be temporarily relaxed by making skb_orphan_frags_rx identical to skb_orphan_frags.

For VSOCK type of socket example can be found in tools/testing/vsock/vsock_test_zerocopy.c.

FAILOVER

15.1 Overview

The failover module provides a generic interface for paravirtual drivers to register a netdev and a set of ops with a failover instance. The ops are used as event handlers that get called to handle netdev register/ unregister/link change/name change events on slave pci ethernet devices with the same mac address as the failover netdev.

This enables paravirtual drivers to use a VF as an accelerated low latency datapath. It also allows live migration of VMs with direct attached VFs by failing over to the paravirtual datapath when the VF is unplugged.

NET DIM - GENERIC NETWORK DYNAMIC INTERRUPT MODERATION

Author

Tal Gilboa <talgi@mellanox.com>

Contents

- *Net DIM - Generic Network Dynamic Interrupt Moderation*
 - *Assumptions*
 - *Introduction*
 - *Net DIM Algorithm*
 - *Registering a Network Device to DIM*
 - *Example*
 - *Dynamic Interrupt Moderation (DIM) library API*

16.1 Assumptions

This document assumes the reader has basic knowledge in network drivers and in general interrupt moderation.

16.2 Introduction

Dynamic Interrupt Moderation (DIM) (in networking) refers to changing the interrupt moderation configuration of a channel in order to optimize packet processing. The mechanism includes an algorithm which decides if and how to change moderation parameters for a channel, usually by performing an analysis on runtime data sampled from the system. Net DIM is such a mechanism. In each iteration of the algorithm, it analyses a given sample of the data, compares it to the previous sample and if required, it can decide to change some of the interrupt moderation configuration fields. The data sample is composed of data bandwidth, the number of packets and the number of events. The time between samples is also measured. Net DIM compares the current and the previous data and returns an adjusted interrupt moderation configuration object. In some cases, the algorithm might decide not to change anything. The configuration fields are the minimum duration (microseconds) allowed between events and the maximum

number of wanted packets per event. The Net DIM algorithm ascribes importance to increase bandwidth over reducing interrupt rate.

16.3 Net DIM Algorithm

Each iteration of the Net DIM algorithm follows these steps:

1. Calculates new data sample.
2. Compares it to previous sample.
3. Makes a decision - suggests interrupt moderation configuration fields.
4. Applies a schedule work function, which applies suggested configuration.

The first two steps are straightforward, both the new and the previous data are supplied by the driver registered to Net DIM. The previous data is the new data supplied to the previous iteration. The comparison step checks the difference between the new and previous data and decides on the result of the last step. A step would result as "better" if bandwidth increases and as "worse" if bandwidth reduces. If there is no change in bandwidth, the packet rate is compared in a similar fashion - increase == "better" and decrease == "worse". In case there is no change in the packet rate as well, the interrupt rate is compared. Here the algorithm tries to optimize for lower interrupt rate so an increase in the interrupt rate is considered "worse" and a decrease is considered "better". Step #2 has an optimization for avoiding false results: it only considers a difference between samples as valid if it is greater than a certain percentage. Also, since Net DIM does not measure anything by itself, it assumes the data provided by the driver is valid.

Step #3 decides on the suggested configuration based on the result from step #2 and the internal state of the algorithm. The states reflect the "direction" of the algorithm: is it going left (reducing moderation), right (increasing moderation) or standing still. Another optimization is that if a decision to stay still is made multiple times, the interval between iterations of the algorithm would increase in order to reduce calculation overhead. Also, after "parking" on one of the most left or most right decisions, the algorithm may decide to verify this decision by taking a step in the other direction. This is done in order to avoid getting stuck in a "deep sleep" scenario. Once a decision is made, an interrupt moderation configuration is selected from the predefined profiles.

The last step is to notify the registered driver that it should apply the suggested configuration. This is done by scheduling a work function, defined by the Net DIM API and provided by the registered driver.

As you can see, Net DIM itself does not actively interact with the system. It would have trouble making the correct decisions if the wrong data is supplied to it and it would be useless if the work function would not apply the suggested configuration. This does, however, allow the registered driver some room for manoeuvre as it may provide partial data or ignore the algorithm suggestion under some conditions.

16.4 Registering a Network Device to DIM

Net DIM API exposes the main function `net_dim()`. This function is the entry point to the Net DIM algorithm and has to be called every time the driver would like to check if it should change interrupt moderation parameters. The driver should provide two data structures: `struct dim` and `struct dim_sample`. `struct dim` describes the state of DIM for a specific object (RX queue, TX queue, other queues, etc.). This includes the current selected profile, previous data samples, the callback function provided by the driver and more. `struct dim_sample` describes a data sample, which will be compared to the data sample stored in `struct dim` in order to decide on the algorithm's next step. The sample should include bytes, packets and interrupts, measured by the driver.

In order to use Net DIM from a networking driver, the driver needs to call the main `net_dim()` function. The recommended method is to call `net_dim()` on each interrupt. Since Net DIM has a built-in moderation and it might decide to skip iterations under certain conditions, there is no need to moderate the `net_dim()` calls as well. As mentioned above, the driver needs to provide an object of type `struct dim` to the `net_dim()` function call. It is advised for each entity using Net DIM to hold a `struct dim` as part of its data structure and use it as the main Net DIM API object. The `struct dim_sample` should hold the latest bytes, packets and interrupts count. No need to perform any calculations, just include the raw data.

The `net_dim()` call itself does not return anything. Instead Net DIM relies on the driver to provide a callback function, which is called when the algorithm decides to make a change in the interrupt moderation parameters. This callback will be scheduled and run in a separate thread in order not to add overhead to the data flow. After the work is done, Net DIM algorithm needs to be set to the proper state in order to move to the next iteration.

16.5 Example

The following code demonstrates how to register a driver to Net DIM. The actual usage is not complete but it should make the outline of the usage clear.

```
#include <linux/dim.h>

/* Callback for net DIM to schedule on a decision to change moderation */
void my_driver_do_dim_work(struct work_struct *work)
{
    /* Get struct dim from struct work_struct */
    struct dim *dim = container_of(work, struct dim,
                                    work);
    /* Do interrupt moderation related stuff */
    ...

    /* Signal net DIM work is done and it should move to next iteration */
    dim->state = DIM_START_MEASURE;
}

/* My driver's interrupt handler */
int my_driver_handle_interrupt(struct my_driver_entity *my_entity, ...)
{
```

```

...
/* A struct to hold current measured data */
struct dim_sample dim_sample;
...
/* Initiate data sample struct with current data */
dim_update_sample(my_entity->events,
                   my_entity->packets,
                   my_entity->bytes,
                   &dim_sample);
/* Call net DIM */
net_dim(&my_entity->dim, dim_sample);
...
}

/* My entity's initialization function (my_entity was already allocated) */
int my_driver_init_my_entity(struct my_driver_entity *my_entity, ...)
{
...
/* Initiate struct work_struct with my driver's callback function */
INIT_WORK(&my_entity->dim.work, my_driver_do_dim_work);
...
}

```

16.6 Dynamic Interrupt Moderation (DIM) library API

struct dim_cq_moder

Structure for CQ moderation values. Used for communications between DIM and its consumer.

Definition:

```

struct dim_cq_moder {
    u16 usec;
    u16 pkts;
    u16 comps;
    u8 cq_period_mode;
};

```

Members

usec

CQ timer suggestion (by DIM)

pkts

CQ packet counter suggestion (by DIM)

comps

Completion counter

cq_period_mode

CQ period count mode (from CQE/EQE)

struct dim_sample

Structure for DIM sample data. Used for communications between DIM and its consumer.

Definition:

```
struct dim_sample {
    ktime_t time;
    u32 pkt_ctr;
    u32 byte_ctr;
    u16 event_ctr;
    u32 comp_ctr;
};
```

Members**time**

Sample timestamp

pkt_ctr

Number of packets

byte_ctr

Number of bytes

event_ctr

Number of events

comp_ctr

Current completion counter

struct dim_stats

Structure for DIM stats. Used for holding current measured rates.

Definition:

```
struct dim_stats {
    int ppms;
    int bpms;
    int epms;
    int cpms;
    int cpe_ratio;
};
```

Members**ppms**

Packets per msec

bpms

Bytes per msec

epms

Events per msec

cpms

Completions per msec

cpe_ratio

Ratio of completions to events

struct dim

Main structure for dynamic interrupt moderation (DIM). Used for holding all information about a specific DIM instance.

Definition:

```
struct dim {
    u8 state;
    struct dim_stats prev_stats;
    struct dim_sample start_sample;
    struct dim_sample measuring_sample;
    struct work_struct work;
    void *priv;
    u8 profile_ix;
    u8 mode;
    u8 tune_state;
    u8 steps_right;
    u8 steps_left;
    u8 tired;
};
```

Members**state**

Algorithm state (see below)

prev_stats

Measured rates from previous iteration (for comparison)

start_sample

Sampled data at start of current iteration

measuring_sample

A [dim_sample](#) that is used to update the current events

work

Work to perform on action required

priv

A pointer to the struct that points to dim

profile_ix

Current moderation profile

mode

CQ period count mode

tune_state

Algorithm tuning state (see below)

steps_right

Number of steps taken towards higher moderation

steps_left

Number of steps taken towards lower moderation

tired

Parking depth counter

enum dim_cq_period_mode

Modes for CQ period count

Constants**DIM_CQ_PERIOD_MODE_START_FROM_EQE**

Start counting from EQE

DIM_CQ_PERIOD_MODE_START_FROM_CQE

Start counting from CQE (implies timer reset)

DIM_CQ_PERIOD_NUM_MODES

Number of modes

enum dim_state

DIM algorithm states

Constants**DIM_START_MEASURE**

This is the first iteration (also after applying a new profile)

DIM_MEASURE_IN_PROGRESS

Algorithm is already in progress - check if need to perform an action

DIM_APPLY_NEW_PROFILE

DIM consumer is currently applying a profile - no need to measure

Description

These will determine if the algorithm is in a valid state to start an iteration.

enum dim_tune_state

DIM algorithm tune states

Constants**DIM_PARKING_ON_TOP**

Algorithm found a local top point - exit on significant difference

DIM_PARKING_TIRED

Algorithm found a deep top point - don't exit if tired > 0

DIM_GOING_RIGHT

Algorithm is currently trying higher moderation levels

DIM_GOING_LEFT

Algorithm is currently trying lower moderation levels

Description

These will determine which action the algorithm should perform.

enum dim_stats_state

DIM algorithm statistics states

Constants

DIM_STATS_WORSE

Current iteration shows worse performance than before

DIM_STATS_SAME

Current iteration shows same performance than before

DIM_STATS_BETTER

Current iteration shows better performance than before

Description

These will determine the verdict of current iteration.

enum dim_step_result

DIM algorithm step results

Constants

DIM_STEPPED

Performed a regular step

DIM_TOO_TIRED

Same kind of step was done multiple times - should go to tired parking

DIM_ON_EDGE

Stepped to the most left/right profile

Description

These describe the result of a step.

bool dim_on_top(struct dim *dim)

check if current state is a good place to stop (top location)

Parameters

struct dim *dim

DIM context

Description

Check if current profile is a good place to park at. This will result in reducing the DIM checks frequency as we assume we shouldn't probably change profiles, unless traffic pattern wasn't changed.

void dim_turn(struct dim *dim)

change profile altering direction

Parameters

struct dim *dim

DIM context

Description

Go left if we were going right and vice-versa. Do nothing if currently parking.

void dim_park_on_top(struct dim *dim)

enter a parking state on a top location

Parameters

```
struct dim *dim
    DIM context
```

Description

Enter parking state. Clear all movement history.

```
void dim_park_tired(struct dim *dim)
    enter a tired parking state
```

Parameters

```
struct dim *dim
    DIM context
```

Description

Enter parking state. Clear all movement history and cause DIM checks frequency to reduce.

```
bool dim_calc_stats(struct dim_sample *start, struct dim_sample *end, struct dim_stats *curr_stats)
    calculate the difference between two samples
```

Parameters

```
struct dim_sample *start
    start sample
```

```
struct dim_sample *end
    end sample
```

```
struct dim_stats *curr_stats
    delta between samples
```

Description

Calculate the delta between two samples (in data rates). Takes into consideration counter wrap-around. Returned boolean indicates whether curr_stats are reliable.

```
void dim_update_sample(u16 event_ctr, u64 packets, u64 bytes, struct dim_sample *s)
    set a sample's fields with given values
```

Parameters

```
u16 event_ctr
    number of events to set
```

```
u64 packets
    number of packets to set
```

```
u64 bytes
    number of bytes to set
```

```
struct dim_sample *s
    DIM sample
```

```
void dim_update_sample_with_comps(u16 event_ctr, u64 packets, u64 bytes, u64 comps,
                                    struct dim_sample *s)
```

set a sample's fields with given values including the completion parameter

Parameters

u16 event_ctr
number of events to set

u64 packets
number of packets to set

u64 bytes
number of bytes to set

u64 comps
number of completions to set

struct dim_sample *s
DIM sample

struct *dim_cq_moder* **net_dim_get_rx_moderation**(u8 cq_period_mode, int ix)
provide a CQ moderation object for the given RX profile

Parameters

u8 cq_period_mode
CQ period mode

int ix
Profile index

struct *dim_cq_moder* **net_dim_get_def_rx_moderation**(u8 cq_period_mode)
provide the default RX moderation

Parameters

u8 cq_period_mode
CQ period mode

struct *dim_cq_moder* **net_dim_get_tx_moderation**(u8 cq_period_mode, int ix)
provide a CQ moderation object for the given TX profile

Parameters

u8 cq_period_mode
CQ period mode

int ix
Profile index

struct *dim_cq_moder* **net_dim_get_def_tx_moderation**(u8 cq_period_mode)
provide the default TX moderation

Parameters

u8 cq_period_mode
CQ period mode

void **net_dim**(struct *dim* *dim, struct *dim_sample* end_sample)
main DIM algorithm entry point

Parameters

struct dim *dim
DIM instance information

```
struct dim_sample end_sample
    Current data measurement
```

Description

Called by the consumer. This is the main logic of the algorithm, where data is processed in order to decide on next required action.

```
void rdma_dim(struct dim *dim, u64 completions)
```

Runs the adaptive moderation.

Parameters

```
struct dim *dim
```

The moderation struct.

```
u64 completions
```

The number of completions collected in this round.

Description

Each call to rdma_dim takes the latest amount of completions that have been collected and counts them as a new event. Once enough events have been collected the algorithm decides a new moderation level.

NET_FAILOVER

17.1 Overview

The net_failover driver provides an automated failover mechanism via APIs to create and destroy a failover master netdev and manages a primary and standby slave netdevs that get registered via the generic failover infrastructure.

The failover netdev acts a master device and controls 2 slave devices. The original paravirtual interface is registered as 'standby' slave netdev and a passthru/vf device with the same MAC gets registered as 'primary' slave netdev. Both 'standby' and 'failover' netdevs are associated with the same 'pci' device. The user accesses the network interface via 'failover' netdev. The 'failover' netdev chooses 'primary' netdev as default for transmits when it is available with link up and running.

This can be used by paravirtual drivers to enable an alternate low latency datapath. It also enables hypervisor controlled live migration of a VM with direct attached VF by failing over to the paravirtual datapath when the VF is unplugged.

17.2 virtio-net accelerated datapath: STANDBY mode

net_failover enables hypervisor controlled accelerated datapath to virtio-net enabled VMs in a transparent manner with no/minimal guest userspace changes.

To support this, the hypervisor needs to enable VIRTIO_NET_F_STANDBY feature on the virtio-net interface and assign the same MAC address to both virtio-net and VF interfaces.

Here is an example libvirt XML snippet that shows such configuration:

```
<interface type='network'>
  <mac address='52:54:00:00:12:53' />
  <source network='enp66s0f0_br' />
  <target dev='tap01' />
  <model type='virtio' />
  <driver name='vhost' queues='4' />
  <link state='down' />
  <teaming type='persistent' />
  <alias name='ua-backup0' />
</interface>
<interface type='hostdev' managed='yes'>
  <mac address='52:54:00:00:12:53' />
```

```

<source>
  <address type='pci' domain='0x0000' bus='0x42' slot='0x02' function='0x5' />
</source>
<teaming type='transient' persistent='ua-backup0' />
</interface>

```

In this configuration, the first device definition is for the virtio-net interface and this acts as the 'persistent' device indicating that this interface will always be plugged in. This is specified by the 'teaming' tag with required attribute type having value 'persistent'. The link state for the virtio-net device is set to 'down' to ensure that the 'failover' netdev prefers the VF passthrough device for normal communication. The virtio-net device will be brought UP during live migration to allow uninterrupted communication.

The second device definition is for the VF passthrough interface. Here the 'teaming' tag is provided with type 'transient' indicating that this device may periodically be unplugged. A second attribute - 'persistent' is provided and points to the alias name declared for the virtio-net device.

Booting a VM with the above configuration will result in the following 3 interfaces created in the VM:

```

4: ens10: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
  ↳ group default qlen 1000
    link/ether 52:54:00:00:12:53 brd ff:ff:ff:ff:ff:ff
    inet 192.168.12.53/24 brd 192.168.12.255 scope global dynamic ens10
      valid_lft 42482sec preferred_lft 42482sec
    inet6 fe80::97d8:db2:8c10:b6d6/64 scope link
      valid_lft forever preferred_lft forever
5: ens10nsby: <BROADCAST,MULTICAST> mtu 1500 qdisc fq_codel master ens10 state DOWN
  ↳ group default qlen 1000
    link/ether 52:54:00:00:12:53 brd ff:ff:ff:ff:ff:ff
7: ens11: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq master ens10
  ↳ state UP group default qlen 1000
    link/ether 52:54:00:00:12:53 brd ff:ff:ff:ff:ff:ff

```

Here, ens10 is the 'failover' master interface, ens10nsby is the slave 'standby' virtio-net interface, and ens11 is the slave 'primary' VF passthrough interface.

One point to note here is that some user space network configuration daemons like systemd-networkd, ifupdown, etc, do not understand the 'net_failover' device; and on the first boot, the VM might end up with both 'failover' device and VF acquiring IP addresses (either same or different) from the DHCP server. This will result in lack of connectivity to the VM. So some tweaks might be needed to these network configuration daemons to make sure that an IP is received only on the 'failover' device.

Below is the patch snippet used with 'cloud-ifupdown-helper' script found on Debian cloud images:

```

@@ -27,6 +27,8 @@ do_setup() {
  local working="$cfgdir/.${INTERFACE}" local final="$cfgdir/${INTERFACE}"
  • if [ -d "/sys/class/net/${INTERFACE}/master" ]; then exit 0; fi

```

- **if ifup --no-act "\$INTERFACE" > /dev/null 2>&1; then**
 # interface is already known to ifupdown, no need to generate cfg log "Skipping configuration generation for \$INTERFACE"

17.3 Live Migration of a VM with SR-IOV VF & virtio-net in STANDBY mode

net_failover also enables hypervisor controlled live migration to be supported with VMs that have direct attached SR-IOV VF devices by automatic failover to the paravirtual datapath when the VF is unplugged.

Here is a sample script that shows the steps to initiate live migration from the source hypervisor. Note: It is assumed that the VM is connected to a software bridge 'br0' which has a single VF attached to it along with the vnet device to the VM. This is not the VF that was passthrough'd to the VM (seen in the vf.xml file).

```
# cat vf.xml
<interface type='hostdev' managed='yes'>
  <mac address='52:54:00:00:12:53' />
  <source>
    <address type='pci' domain='0x0000' bus='0x42' slot='0x02' function='0x5' />
  </source>
  <teaming type='transient' persistent='ua-backup0' />
</interface>

# Source Hypervisor migrate.sh
#!/bin/bash

DOMAIN=vm-01
PF=ens6np0
VF=ens6v1          # VF attached to the bridge.
VF_NUM=1
TAP_IF=vmtap01      # virtio-net interface in the VM.
VF_XML=vf.xml

MAC=52:54:00:00:12:53
ZERO_MAC=00:00:00:00:00:00

# Set the virtio-net interface up.
virsh domif-setlink $DOMAIN $TAP_IF up

# Remove the VF that was passthrough'd to the VM.
virsh detach-device --live --config $DOMAIN $VF_XML

ip link set $PF vf $VF_NUM mac $ZERO_MAC

# Add FDB entry for traffic to continue going to the VM via
# the VF -> br0 -> vnet interface path.
bridge fdb add $MAC dev $VF
bridge fdb add $MAC dev $TAP_IF master
```

```
# Migrate the VM
virsh migrate --live --persistent $DOMAIN qemu+ssh://$REMOTE_HOST/system

# Clean up FDB entries after migration completes.
bridge fdb del $MAC dev $VF
bridge fdb del $MAC dev $TAP_IF master
```

On the destination hypervisor, a shared bridge 'br0' is created before migration starts, and a VF from the destination PF is added to the bridge. Similarly an appropriate FDB entry is added.

The following script is executed on the destination hypervisor once migration completes, and it reattaches the VF to the VM and brings down the virtio-net interface.

```
::
# reattach-vf.sh #!/bin/bash

bridge fdb del 52:54:00:00:12:53 dev ens36v0 bridge fdb del 52:54:00:00:12:53 dev vm-
tap01 master virsh attach-device --config --live vm01 vf.xml virsh domif-setlink vm01 vm-
tap01 down
```

PAGE POOL API

The page_pool allocator is optimized for recycling page or page fragment used by skb packet and xdp frame.

Basic use involves replacing any alloc_pages() calls with page_pool_alloc(), which allocate memory with or without page splitting depending on the requested memory size.

If the driver knows that it always requires full pages or its allocations are always smaller than half a page, it can use one of the more specific API calls:

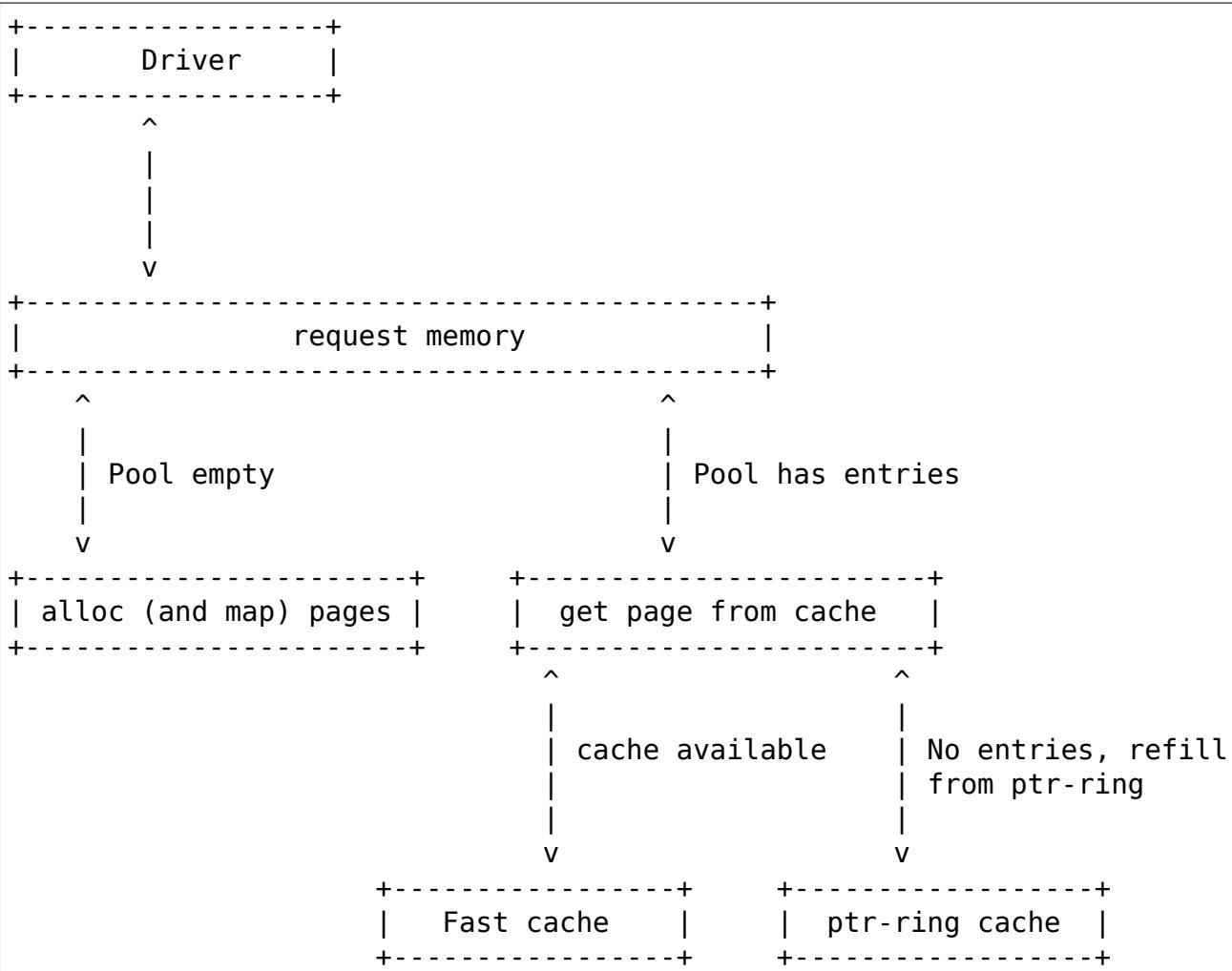
1. page_pool_alloc_pages(): allocate memory without page splitting when driver knows that the memory it need is always bigger than half of the page allocated from page pool. There is no cache line dirtying for 'struct page' when a page is recycled back to the page pool.
2. page_pool_alloc_frag(): allocate memory with page splitting when driver knows that the memory it need is always smaller than or equal to half of the page allocated from page pool. Page splitting enables memory saving and thus avoids TLB/cache miss for data access, but there also is some cost to implement page splitting, mainly some cache line dirtying/bouncing for 'struct page' and atomic operation for page->pp_ref_count.

The API keeps track of in-flight pages, in order to let API users know when it is safe to free a page_pool object, the API users must call `page_pool_put_page()` or `page_pool_free_va()` to free the page_pool object, or attach the page_pool object to a page_pool-aware object like skbs marked with `skb_mark_for_recycle()`.

`page_pool_put_page()` may be called multiple times on the same page if a page is split into multiple fragments. For the last fragment, it will either recycle the page, or in case of `page->refcount > 1`, it will release the DMA mapping and in-flight state accounting.

`dma_sync_single_range_for_device()` is only called for the last fragment when page_pool is created with PP_FLAG_DMA_SYNC_DEV flag, so it depends on the last freed fragment to do the sync_for_device operation for all fragments in the same page when a page is split. The API user must setup pool->p.max_len and pool->p.offset correctly and ensure that `page_pool_put_page()` is called with `dma_sync_size` being -1 for fragment API.

18.1 Architecture overview



18.2 Monitoring

Information about page pools on the system can be accessed via the netdev genetlink family (see Documentation/netlink/specs/netdev.yaml).

18.3 API interface

The number of pools created **must** match the number of hardware queues unless hardware restrictions make that impossible. This would otherwise beat the purpose of page pool, which is allocate pages fast from cache without locking. This lockless guarantee naturally comes from running under a NAPI softirq. The protection doesn't strictly have to be NAPI, any guarantee that allocating a page will cause no race conditions is enough.

```
struct page_pool *page_pool_create(const struct page_pool_params *params)
create a page pool.
```

Parameters

```
const struct page_pool_params *params
    parameters, see struct page_pool_params

struct page_pool_params
    page pool parameters
```

Definition:

```
struct page_pool_params {
    unsigned int      flags;
    unsigned int      order;
    unsigned int      pool_size;
    int              nid;
    struct device    *dev;
    struct napi_struct *napi;
    enum dma_data_direction dma_dir;
    unsigned int      max_len;
    unsigned int      offset;
    STRUCT_GROUP( struct net_device *netdev;
};
```

Members

flags
PP_FLAG_DMA_MAP, PP_FLAG_DMA_SYNC_DEV

order
 2^{order} pages on allocation

pool_size
size of the ptr_ring

nid
NUMA node id to allocate from pages from

dev
device, for DMA pre-mapping purposes

napi
NAPI which is the sole consumer of pages, otherwise NULL

dma_dir
DMA mapping direction

max_len
max DMA sync memory size for PP_FLAG_DMA_SYNC_DEV

offset
DMA sync address offset for PP_FLAG_DMA_SYNC_DEV

netdev
netdev this pool will serve (leave as NULL if none or multiple)

struct page ***page_pool_dev_alloc_pages**(struct page_pool *pool)
allocate a page.

Parameters

struct page_pool *pool
pool from which to allocate

Description

Get a page from the page allocator or page_pool caches.

struct page *page_pool_dev_alloc_frag(**struct page_pool *pool**, **unsigned int *offset**,
unsigned int size)
allocate a page fragment.

Parameters

struct page_pool *pool
pool from which to allocate

unsigned int *offset
offset to the allocated page

unsigned int size
requested size

Description

Get a page fragment from the page allocator or page_pool caches.

Return

Return allocated page fragment, otherwise return NULL.

struct page *page_pool_dev_alloc(**struct page_pool *pool**, **unsigned int *offset**, **unsigned int *size**)
allocate a page or a page fragment.

Parameters

struct page_pool *pool
pool from which to allocate

unsigned int *offset
offset to the allocated page

unsigned int *size
in as the requested size, out as the allocated size

Description

Get a page or a page fragment from the page allocator or page_pool caches depending on the requested size in order to allocate memory with least memory utilization and performance penalty.

Return

Return allocated page or page fragment, otherwise return NULL.

void *page_pool_dev_alloc_va(**struct page_pool *pool**, **unsigned int *size**)
allocate a page or a page fragment and return its va.

Parameters

struct page_pool *pool
pool from which to allocate

unsigned int *size
in as the requested size, out as the allocated size

Description

This is just a thin wrapper around the page_pool_alloc() API, and it returns va of the allocated page or page fragment.

Return

Return the va for the allocated page or page fragment, otherwise return NULL.

enum dma_data_direction page_pool_get_dma_dir(struct page_pool *pool)
Retrieve the stored DMA direction.

Parameters

struct page_pool *pool
pool from which page was allocated

Description

Get the stored dma direction. A driver might decide to store this locally and avoid the extra cache line from page_pool to determine the direction.

void page_pool_put_page(struct page_pool *pool, struct page *page, unsigned int dma_sync_size, bool allow_direct)
release a reference to a page pool page

Parameters

struct page_pool *pool
pool from which page was allocated

struct page *page
page to release a reference on

unsigned int dma_sync_size
how much of the page may have been touched by the device

bool allow_direct
released by the consumer, allow lockless caching

Description

The outcome of this depends on the page refcnt. If the driver bumps the refcnt > 1 this will unmap the page. If the page refcnt is 1 the allocator owns the page and will try to recycle it in one of the pool caches. If PP_FLAG_DMA_SYNC_DEV is set, the page will be synced for_device using dma_sync_single_range_for_device().

void page_pool_put_full_page(struct page_pool *pool, struct page *page, bool allow_direct)
release a reference on a page pool page

Parameters

struct page_pool *pool
pool from which page was allocated

struct page *page
page to release a reference on

bool allow_direct

released by the consumer, allow lockless caching

Description

Similar to `page_pool_put_page()`, but will DMA sync the entire memory area as configured in `page_pool_params.max_len`.

void page_pool_recycle_direct(struct page_pool *pool, struct page *page)

release a reference on a page pool page

Parameters

struct page_pool *pool

pool from which page was allocated

struct page *page

page to release a reference on

Description

Similar to `page_pool_put_full_page()` but caller must guarantee safe context (e.g NAPI), since it will recycle the page directly into the pool fast cache.

void page_pool_free_va(struct page_pool *pool, void *va, bool allow_direct)

free a va into the page_pool

Parameters

struct page_pool *pool

pool from which va was allocated

void *va

va to be freed

bool allow_direct

freed by the consumer, allow lockless caching

Description

Free a va allocated from `page_pool_allo_va()`.

dma_addr_t page_pool_get_dma_addr(struct page *page)

Retrieve the stored DMA address.

Parameters

struct page *page

page allocated from a page pool

Description

Fetch the DMA address of the page. The page pool to which the page belongs must have been created with `PP_FLAG_DMA_MAP`.

bool page_pool_get_stats(const struct page_pool *pool, struct page_pool_stats *stats)

fetch page pool stats

Parameters

const struct page_pool *pool

pool from which page was allocated

```
struct page_pool_stats *stats
    struct page_pool_stats to fill in
```

Description

Retrieve statistics about the page_pool. This API is only available if the kernel has been configured with CONFIG_PAGE_POOL_STATS=y. A pointer to a caller allocated *struct page_pool_stats* structure is passed to this API which is filled in. The caller can then report those stats to the user (perhaps via ethtool, debugfs, etc.).

```
void page_pool_put_page_bulk(struct page_pool *pool, void **data, int count)
    release references on multiple pages
```

Parameters

```
struct page_pool *pool
    pool from which pages were allocated
```

```
void **data
    array holding page pointers
```

```
int count
    number of pages in data
```

Description

Tries to refill a number of pages into the ptr_ring cache holding ptr_ring producer lock. If the ptr_ring is full, *page_pool_put_page_bulk()* will release leftover pages to the page allocator. *page_pool_put_page_bulk()* is suitable to be run inside the driver NAPI tx completion loop for the XDP_REDIRECT use case.

Please note the caller must not use data area after running *page_pool_put_page_bulk()*, as this function overwrites it.

18.3.1 DMA sync

Driver is always responsible for syncing the pages for the CPU. Drivers may choose to take care of syncing for the device as well or set the PP_FLAG_DMA_SYNC_DEV flag to request that pages allocated from the page pool are already synced for the device.

If PP_FLAG_DMA_SYNC_DEV is set, the driver must inform the core what portion of the buffer has to be synced. This allows the core to avoid syncing the entire page when the drivers knows that the device only accessed a portion of the page.

Most drivers will reserve headroom in front of the frame. This part of the buffer is not touched by the device, so to avoid syncing it drivers can set the offset field in *struct page_pool_params* appropriately.

For pages recycled on the XDP xmit and skb paths the page pool will use the max_len member of *struct page_pool_params* to decide how much of the page needs to be synced (starting at offset). When directly freeing pages in the driver (*page_pool_put_page()*) the dma_sync_size argument specifies how much of the buffer needs to be synced.

If in doubt set offset to 0, max_len to PAGE_SIZE and pass -1 as dma_sync_size. That combination of arguments is always correct.

Note that the syncing parameters are for the entire page. This is important to remember when using fragments (PP_FLAG_PAGE_FRAG), where allocated buffers may be smaller than a full page.

Unless the driver author really understands page pool internals it's recommended to always use `offset = 0, max_len = PAGE_SIZE` with fragmented page pools.

18.3.2 Stats API and structures

If the kernel is configured with `CONFIG_PAGE_POOL_STATS=y`, the API `page_pool_get_stats()` and structures described below are available. It takes a pointer to a `struct page_pool` and a pointer to a `struct page_pool_stats` allocated by the caller.

Older drivers expose page pool statistics via ethtool or debugfs. The same statistics are accessible via the netlink netdev family in a driver-independent fashion.

```
struct page_pool_alloc_stats
    allocation statistics
```

Definition:

```
struct page_pool_alloc_stats {
    u64 fast;
    u64 slow;
    u64 slow_high_order;
    u64 empty;
    u64 refill;
    u64 waive;
};
```

Members

fast

successful fast path allocations

slow

slow path order-0 allocations

slow_high_order

slow path high order allocations

empty

ptr ring is empty, so a slow path allocation was forced

refill

an allocation which triggered a refill of the cache

waive

pages obtained from the ptr ring that cannot be added to the cache due to a NUMA mismatch

```
struct page_pool_recycle_stats
```

recycling (freeing) statistics

Definition:

```
struct page_pool_recycle_stats {
    u64 cached;
    u64 cache_full;
    u64 ring;
```

```
    u64 ring_full;
    u64 released_refcnt;
};
```

Members**cached**

recycling placed page in the page pool cache

cache_full

page pool cache was full

ring

page placed into the ptr ring

ring_full

page released from page pool because the ptr ring was full

released_refcnt

page released (and not recycled) because refcnt > 1

struct page_pool_stats

combined page pool use statistics

Definition:

```
struct page_pool_stats {
    struct page_pool_alloc_stats alloc_stats;
    struct page_pool_recycle_stats recycle_stats;
};
```

Members**alloc_stats**

see [struct page_pool_alloc_stats](#)

recycle_stats

see [struct page_pool_recycle_stats](#)

Description

Wrapper struct for combining page pool stats with different storage requirements.

18.4 Coding examples

18.4.1 Registration

```
/* Page pool registration */
struct page_pool_params pp_params = { 0 };
struct xdp_rxq_info xdp_rxq;
int err;

pp_params.order = 0;
/* internal DMA mapping in page_pool */
```

```

pp_params.flags = PP_FLAG_DMA_MAP;
pp_params.pool_size = DESC_NUM;
pp_params.nid = NUMA_NO_NODE;
pp_params.dev = priv->dev;
pp_params.napi = napi; /* only if locking is tied to NAPI */
pp_params.dma_dir = xdp_prog ? DMA_BIDIRECTIONAL : DMA_FROM_DEVICE;
page_pool = page_pool_create(&pp_params);

err = xdp_rxq_info_reg(&xdp_rxq, ndev, 0);
if (err)
    goto err_out;

err = xdp_rxq_info_reg_mem_model(&xdp_rxq, MEM_TYPE_PAGE_POOL, page_pool);
if (err)
    goto err_out;

```

18.4.2 NAPI poller

```

/* NAPI Rx poller */
enum dma_data_direction dma_dir;

dma_dir = page_pool_get_dma_dir(dring->page_pool);
while (done < budget) {
    if (some error)
        page_pool_recycle_direct(page_pool, page);
    if (packet_is_xdp) {
        if XDP_DROP:
            page_pool_recycle_direct(page_pool, page);
    } else (packet_is_skb) {
        skb_mark_for_recycle(skb);
        new_page = page_pool_dev_alloc_pages(page_pool);
    }
}

```

18.4.3 Stats

```

#ifdef CONFIG_PAGE_POOL_STATS
/* retrieve stats */
struct page_pool_stats stats = { 0 };
if (page_pool_get_stats(page_pool, &stats)) {
    /* perhaps the driver reports statistics with ethtool */
    ethtool_print_allocation_stats(&stats.alloc_stats);
    ethtool_print_recycle_stats(&stats.recycle_stats);
}
#endif

```

18.4.4 Driver unload

```
/* Driver unload */
page_pool_put_full_page(page_pool, page, false);
xdp_rxq_info_unreg(&xdp_rxq);
```


PHY ABSTRACTION LAYER

19.1 Purpose

Most network devices consist of set of registers which provide an interface to a MAC layer, which communicates with the physical connection through a PHY. The PHY concerns itself with negotiating link parameters with the link partner on the other side of the network connection (typically, an ethernet cable), and provides a register interface to allow drivers to determine what settings were chosen, and to configure what settings are allowed.

While these devices are distinct from the network devices, and conform to a standard layout for the registers, it has been common practice to integrate the PHY management code with the network driver. This has resulted in large amounts of redundant code. Also, on embedded systems with multiple (and sometimes quite different) ethernet controllers connected to the same management bus, it is difficult to ensure safe use of the bus.

Since the PHYs are devices, and the management busses through which they are accessed are, in fact, busses, the PHY Abstraction Layer treats them as such. In doing so, it has these goals:

1. Increase code-reuse
2. Increase overall code-maintainability
3. Speed development time for new network drivers, and for new systems

Basically, this layer is meant to provide an interface to PHY devices which allows network driver writers to write as little code as possible, while still providing a full feature set.

19.2 The MDIO bus

Most network devices are connected to a PHY by means of a management bus. Different devices use different busses (though some share common interfaces). In order to take advantage of the PAL, each bus interface needs to be registered as a distinct device.

1. read and write functions must be implemented. Their prototypes are:

```
int write(struct mii_bus *bus, int mii_id, int regnum, u16 value);
int read(struct mii_bus *bus, int mii_id, int regnum);
```

mii_id is the address on the bus for the PHY, and regnum is the register number. These functions are guaranteed not to be called from interrupt time, so it is safe for them to block, waiting for an interrupt to signal the operation is complete

2. A reset function is optional. This is used to return the bus to an initialized state.
3. A probe function is needed. This function should set up anything the bus driver needs, setup the mii_bus structure, and register with the PAL using mdiobus_register. Similarly, there's a remove function to undo all of that (use mdiobus_unregister).
4. Like any driver, the device_driver structure must be configured, and init exit functions are used to register the driver.
5. The bus must also be declared somewhere as a device, and registered.

As an example for how one driver implemented an mdio bus driver, see drivers/net/ethernet/freescale/fsl_pq_mdio.c and an associated DTS file for one of the users. (e.g. "git grep fsl.*-mdio arch/powerpc/boot/dts/")

19.3 (RG)MII/electrical interface considerations

The Reduced Gigabit Medium Independent Interface (RGMII) is a 12-pin electrical signal interface using a synchronous 125Mhz clock signal and several data lines. Due to this design decision, a 1.5ns to 2ns delay must be added between the clock line (RXC or TXC) and the data lines to let the PHY (clock sink) have a large enough setup and hold time to sample the data lines correctly. The PHY library offers different types of PHY_INTERFACE_MODE_RGMII* values to let the PHY driver and optionally the MAC driver, implement the required delay. The values of phy_interface_t must be understood from the perspective of the PHY device itself, leading to the following:

- **PHY_INTERFACE_MODE_RGMII**: the PHY is not responsible for inserting any internal delay by itself, it assumes that either the Ethernet MAC (if capable) or the PCB traces insert the correct 1.5-2ns delay
- **PHY_INTERFACE_MODE_RGMII_TXID**: the PHY should insert an internal delay for the transmit data lines (TXD[3:0]) processed by the PHY device
- **PHY_INTERFACE_MODE_RGMII_RXID**: the PHY should insert an internal delay for the receive data lines (RXD[3:0]) processed by the PHY device
- **PHY_INTERFACE_MODE_RGMII_ID**: the PHY should insert internal delays for both transmit AND receive data lines from/to the PHY device

Whenever possible, use the PHY side RGMII delay for these reasons:

- PHY devices may offer sub-nanosecond granularity in how they allow a receiver/transmitter side delay (e.g: 0.5, 1.0, 1.5ns) to be specified. Such precision may be required to account for differences in PCB trace lengths
- PHY devices are typically qualified for a large range of applications (industrial, medical, automotive...), and they provide a constant and reliable delay across temperature/pressure/voltage ranges
- PHY device drivers in PHYLIB being reusable by nature, being able to configure correctly a specified delay enables more designs with similar delay requirements to be operated correctly

For cases where the PHY is not capable of providing this delay, but the Ethernet MAC driver is capable of doing so, the correct phy_interface_t value should be PHY_INTERFACE_MODE_RGMII, and the Ethernet MAC driver should be configured correctly

in order to provide the required transmit and/or receive side delay from the perspective of the PHY device. Conversely, if the Ethernet MAC driver looks at the `phy_interface_t` value, for any other mode but `PHY_INTERFACE_MODE_RGMII`, it should make sure that the MAC-level delays are disabled.

In case neither the Ethernet MAC, nor the PHY are capable of providing the required delays, as defined per the RGMII standard, several options may be available:

- Some SoCs may offer a pin pad/mux/controller capable of configuring a given set of pins' strength, delays, and voltage; and it may be a suitable option to insert the expected 2ns RGMII delay.
- Modifying the PCB design to include a fixed delay (e.g: using a specifically designed serpentine), which may not require software configuration at all.

19.3.1 Common problems with RGMII delay mismatch

When there is a RGMII delay mismatch between the Ethernet MAC and the PHY, this will most likely result in the clock and data line signals to be unstable when the PHY or MAC take a snapshot of these signals to translate them into logical 1 or 0 states and reconstruct the data being transmitted/received. Typical symptoms include:

- Transmission/reception partially works, and there is frequent or occasional packet loss observed
- Ethernet MAC may report some or all packets ingressing with a FCS/CRC error, or just discard them all
- Switching to lower speeds such as 10/100Mbits/sec makes the problem go away (since there is enough setup/hold time in that case)

19.4 Connecting to a PHY

Sometime during startup, the network driver needs to establish a connection between the PHY device, and the network device. At this time, the PHY's bus and drivers need to all have been loaded, so it is ready for the connection. At this point, there are several ways to connect to the PHY:

1. The PAL handles everything, and only calls the network driver when the link state changes, so it can react.
2. The PAL handles everything except interrupts (usually because the controller has the interrupt registers).
3. The PAL handles everything, but checks in with the driver every second, allowing the network driver to react first to any changes before the PAL does.
4. The PAL serves only as a library of functions, with the network device manually calling functions to update status, and configure the PHY

19.5 Letting the PHY Abstraction Layer do Everything

If you choose option 1 (The hope is that every driver can, but to still be useful to drivers that can't), connecting to the PHY is simple:

First, you need a function to react to changes in the link state. This function follows this protocol:

```
static void adjust_link(struct net_device *dev);
```

Next, you need to know the device name of the PHY connected to this device. The name will look something like, "0:00", where the first number is the bus id, and the second is the PHY's address on that bus. Typically, the bus is responsible for making its ID unique.

Now, to connect, just call this function:

```
phydev = phy_connect(dev, phy_name, &adjust_link, interface);
```

phydev is a pointer to the *phy_device* structure which represents the PHY. If *phy_connect* is successful, it will return the pointer. *dev*, here, is the pointer to your *net_device*. Once done, this function will have started the PHY's software state machine, and registered for the PHY's interrupt, if it has one. The *phydev* structure will be populated with information about the current state, though the PHY will not yet be truly operational at this point.

PHY-specific flags should be set in *phydev->dev_flags* prior to the call to *phy_connect()* such that the underlying PHY driver can check for flags and perform specific operations based on them. This is useful if the system has put hardware restrictions on the PHY/controller, of which the PHY needs to be aware.

interface is a u32 which specifies the connection type used between the controller and the PHY. Examples are GMII, MII, RGMII, and SGMII. See "PHY interface mode" below. For a full list, see *include/linux/phy.h*

Now just make sure that *phydev->supported* and *phydev->advertising* have any values pruned from them which don't make sense for your controller (a 10/100 controller may be connected to a gigabit capable PHY, so you would need to mask off SUPPORTED_1000baseT*). See *include/linux/ethtool.h* for definitions for these bitfields. Note that you should not SET any bits, except the SUPPORTED_Pause and SUPPORTED_AsymPause bits (see below), or the PHY may get put into an unsupported state.

Lastly, once the controller is ready to handle network traffic, you call *phy_start(phydev)*. This tells the PAL that you are ready, and configures the PHY to connect to the network. If the MAC interrupt of your network driver also handles PHY status changes, just set *phydev->irq* to *PHY_MAC_INTERRUPT* before you call *phy_start* and use *phy_mac_interrupt()* from the network driver. If you don't want to use interrupts, set *phydev->irq* to *PHY_POLL*. *phy_start()* enables the PHY interrupts (if applicable) and starts the phylib state machine.

When you want to disconnect from the network (even if just briefly), you call *phy_stop(phydev)*. This function also stops the phylib state machine and disables PHY interrupts.

19.6 PHY interface modes

The PHY interface mode supplied in the `phy_connect()` family of functions defines the initial operating mode of the PHY interface. This is not guaranteed to remain constant; there are PHYs which dynamically change their interface mode without software interaction depending on the negotiation results.

Some of the interface modes are described below:

PHY_INTERFACE_MODE_SMII

This is serial MII, clocked at 125MHz, supporting 100M and 10M speeds. Some details can be found in <https://opencores.org/ocsvn/smii/smii/trunk/doc/SMII.pdf>

PHY_INTERFACE_MODE_1000BASEX

This defines the 1000BASE-X single-lane serdes link as defined by the 802.3 standard section 36. The link operates at a fixed bit rate of 1.25Gbaud using a 10B/8B encoding scheme, resulting in an underlying data rate of 1Gbps. Embedded in the data stream is a 16-bit control word which is used to negotiate the duplex and pause modes with the remote end. This does not include "up-clocked" variants such as 2.5Gbps speeds (see below.)

PHY_INTERFACE_MODE_2500BASEX

This defines a variant of 1000BASE-X which is clocked 2.5 times as fast as the 802.3 standard, giving a fixed bit rate of 3.125Gbaud.

PHY_INTERFACE_MODE_SGMII

This is used for Cisco SGMII, which is a modification of 1000BASE-X as defined by the 802.3 standard. The SGMII link consists of a single serdes lane running at a fixed bit rate of 1.25Gbaud with 10B/8B encoding. The underlying data rate is 1Gbps, with the slower speeds of 100Mbps and 10Mbps being achieved through replication of each data symbol. The 802.3 control word is re-purposed to send the negotiated speed and duplex information from to the MAC, and for the MAC to acknowledge receipt. This does not include "up-clocked" variants such as 2.5Gbps speeds.

Note: mismatched SGMII vs 1000BASE-X configuration on a link can successfully pass data in some circumstances, but the 16-bit control word will not be correctly interpreted, which may cause mismatches in duplex, pause or other settings. This is dependent on the MAC and/or PHY behaviour.

PHY_INTERFACE_MODE_5GBASER

This is the IEEE 802.3 Clause 129 defined 5GBASE-R protocol. It is identical to the 10GBASE-R protocol defined in Clause 49, with the exception that it operates at half the frequency. Please refer to the IEEE standard for the definition.

PHY_INTERFACE_MODE_10GBASER

This is the IEEE 802.3 Clause 49 defined 10GBASE-R protocol used with various different mediums. Please refer to the IEEE standard for a definition of this.

Note: 10GBASE-R is just one protocol that can be used with XFI and SFI. XFI and SFI permit multiple protocols over a single SERDES lane, and also defines the electrical characteristics of the signals with a host compliance board plugged into the host XFP/SFP connector. Therefore, XFI and SFI are not PHY interface types in their own right.

PHY_INTERFACE_MODE_10GKR

This is the IEEE 802.3 Clause 49 defined 10GBASE-R with Clause 73 autonegotiation. Please refer to the IEEE standard for further information.

Note: due to legacy usage, some 10GBASE-R usage incorrectly makes use of this definition.

PHY_INTERFACE_MODE_25GBASER

This is the IEEE 802.3 PCS Clause 107 defined 25GBASE-R protocol. The PCS is identical to 10GBASE-R, i.e. 64B/66B encoded running 2.5 as fast, giving a fixed bit rate of 25.78125 Gbaud. Please refer to the IEEE standard for further information.

PHY_INTERFACE_MODE_100BASEX

This defines IEEE 802.3 Clause 24. The link operates at a fixed data rate of 125Mpbs using a 4B/5B encoding scheme, resulting in an underlying data rate of 100Mpbs.

PHY_INTERFACE_MODE_QUSGMII

This defines the Cisco the Quad USGMII mode, which is the Quad variant of the USGMII (Universal SGMII) link. It's very similar to QSGMII, but uses a Packet Control Header (PCH) instead of the 7 bytes preamble to carry not only the port id, but also so-called "extensions". The only documented extension so-far in the specification is the inclusion of timestamps, for PTP-enabled PHYs. This mode isn't compatible with QSGMII, but offers the same capabilities in terms of link speed and negotiation.

PHY_INTERFACE_MODE_1000BASEKX

This is 1000BASE-X as defined by IEEE 802.3 Clause 36 with Clause 73 autonegotiation. Generally, it will be used with a Clause 70 PMD. To contrast with the 1000BASE-X phy mode used for Clause 38 and 39 PMDs, this interface mode has different autonegotiation and only supports full duplex.

PHY_INTERFACE_MODE_PSGMII

This is the Penta SGMII mode, it is similar to QSGMII but it combines 5 SGMII lines into a single link compared to 4 on QSGMII.

19.7 Pause frames / flow control

The PHY does not participate directly in flow control/pause frames except by making sure that the SUPPORTED_Pause and SUPPORTED_AsymPause bits are set in MII_ADVVERTISE to indicate towards the link partner that the Ethernet MAC controller supports such a thing. Since flow control/pause frames generation involves the Ethernet MAC driver, it is recommended that this driver takes care of properly indicating advertisement and support for such features by setting the SUPPORTED_Pause and SUPPORTED_AsymPause bits accordingly. This can be done either before or after `phy_connect()` and/or as a result of implementing the ethtool::set_pauseparam feature.

19.8 Keeping Close Tabs on the PAL

It is possible that the PAL's built-in state machine needs a little help to keep your network device and the PHY properly in sync. If so, you can register a helper function when connecting to the PHY, which will be called every second before the state machine reacts to any changes. To do this, you need to manually call `phy_attach()` and `phy_prepare_link()`, and then call `phy_start_machine()` with the second argument set to point to your special handler.

Currently there are no examples of how to use this functionality, and testing on it has been limited because the author does not have any drivers which use it (they all use option 1). So Caveat Emptor.

19.9 Doing it all yourself

There's a remote chance that the PAL's built-in state machine cannot track the complex interactions between the PHY and your network device. If this is so, you can simply call `phy_attach()`, and not call `phy_start_machine` or `phy_prepare_link()`. This will mean that `phydev->state` is entirely yours to handle (`phy_start` and `phy_stop` toggle between some of the states, so you might need to avoid them).

An effort has been made to make sure that useful functionality can be accessed without the state-machine running, and most of these functions are descended from functions which did not interact with a complex state-machine. However, again, no effort has been made so far to test running without the state machine, so tryer beware.

Here is a brief rundown of the functions:

```
int phy_read(struct phy_device *phydev, u16 regnum);
int phy_write(struct phy_device *phydev, u16 regnum, u16 val);
```

Simple read/write primitives. They invoke the bus's read/write function pointers.

```
void phy_print_status(struct phy_device *phydev);
```

A convenience function to print out the PHY status neatly.

```
void phy_request_interrupt(struct phy_device *phydev);
```

Requests the IRQ for the PHY interrupts.

```
struct phy_device * phy_attach(struct net_device *dev, const char *phy_id,
                               phy_interface_t interface);
```

Attaches a network device to a particular PHY, binding the PHY to a generic driver if none was found during bus initialization.

```
int phy_start_aneg(struct phy_device *phydev);
```

Using variables inside the `phydev` structure, either configures advertising and resets autonegotiation, or disables autonegotiation, and configures forced settings.

```
static inline int phy_read_status(struct phy_device *phydev);
```

Fills the `phydev` structure with up-to-date information about the current settings in the PHY.

```
int phy_ethtool_ksettings_set(struct phy_device *phydev,
                               const struct ethtool_link_ksettings *cmd);
```

Ethtool convenience functions.

```
int phy_mii_ioctl(struct phy_device *phydev,
                  struct mii_ioctl_data *mii_data, int cmd);
```

The MII ioctl. Note that this function will completely screw up the state machine if you write registers like BMCR, BMSR, ADVERTISE, etc. Best to use this only to write registers which are not standard, and don't set off a renegotiation.

19.10 PHY Device Drivers

With the PHY Abstraction Layer, adding support for new PHYs is quite easy. In some cases, no work is required at all! However, many PHYs require a little hand-holding to get up-and-running.

19.10.1 Generic PHY driver

If the desired PHY doesn't have any errata, quirks, or special features you want to support, then it may be best to not add support, and let the PHY Abstraction Layer's Generic PHY Driver do all of the work.

19.10.2 Writing a PHY driver

If you do need to write a PHY driver, the first thing to do is make sure it can be matched with an appropriate PHY device. This is done during bus initialization by reading the device's UID (stored in registers 2 and 3), then comparing it to each driver's phy_id field by ANDing it with each driver's phy_id_mask field. Also, it needs a name. Here's an example:

```
static struct phy_driver dm9161_driver = {
    .phy_id          = 0x0181b880,
    .name            = "Davicom DM9161E",
    .phy_id_mask     = 0xffffffff0,
    ...
}
```

Next, you need to specify what features (speed, duplex, autoneg, etc) your PHY device and driver support. Most PHYs support PHY_BASIC_FEATURES, but you can look in include/mii.h for other features.

Each driver consists of a number of function pointers, documented in include/linux/phy.h under the phy_driver structure.

Of these, only config_aneg and read_status are required to be assigned by the driver code. The rest are optional. Also, it is preferred to use the generic phy driver's versions of these two functions if at all possible: genphy_read_status and genphy_config_aneg. If this is not possible, it is likely that you only need to perform some actions before and after invoking these functions, and so your functions will wrap the generic ones.

Feel free to look at the Marvell, Cicada, and Davicom drivers in drivers/net/phy/ for examples (the lxt and qsemi drivers have not been tested as of this writing).

The PHY's MMD register accesses are handled by the PAL framework by default, but can be overridden by a specific PHY driver if required. This could be the case if a PHY was released for manufacturing before the MMD PHY register definitions were standardized by the IEEE. Most modern PHYs will be able to use the generic PAL framework for accessing the PHY's MMD registers. An example of such usage is for Energy Efficient Ethernet support, implemented in the PAL. This support uses the PAL to access MMD registers for EEE query and configuration if the PHY supports the IEEE standard access mechanisms, or can use the PHY's specific access interfaces if overridden by the specific PHY driver. See the Micrel driver in drivers/net/phy/ for an example of how this can be implemented.

19.11 Board Fixups

Sometimes the specific interaction between the platform and the PHY requires special handling. For instance, to change where the PHY's clock input is, or to add a delay to account for latency issues in the data path. In order to support such contingencies, the PHY Layer allows platform code to register fixups to be run when the PHY is brought up (or subsequently reset).

When the PHY Layer brings up a PHY it checks to see if there are any fixups registered for it, matching based on UID (contained in the PHY device's `phy_id` field) and the bus identifier (contained in `phydev->dev.bus_id`). Both must match, however two constants, `PHY_ANY_ID` and `PHY_ANY_UID`, are provided as wildcards for the bus ID and UID, respectively.

When a match is found, the PHY layer will invoke the run function associated with the fixup. This function is passed a pointer to the `phy_device` of interest. It should therefore only operate on that PHY.

The platform code can either register the fixup using `phy_register_fixup()`:

```
int phy_register_fixup(const char *phy_id,
                      u32 phy_uid, u32 phy_uid_mask,
                      int (*run)(struct phy_device *));
```

Or using one of the two stubs, `phy_register_fixup_for_uid()` and `phy_register_fixup_for_id()`:

```
int phy_register_fixup_for_uid(u32 phy_uid, u32 phy_uid_mask,
                               int (*run)(struct phy_device *));
int phy_register_fixup_for_id(const char *phy_id,
                             int (*run)(struct phy_device *));
```

The stubs set one of the two matching criteria, and set the other one to match anything.

When `phy_register_fixup()` or *_`for_uid()`/*_`for_id()` is called at module load time, the module needs to unregister the fixup and free allocated memory when it's unloaded.

Call one of following function before unloading module:

```
int phy_unregister_fixup(const char *phy_id, u32 phy_uid, u32 phy_uid_mask);
int phy_unregister_fixup_for_uid(u32 phy_uid, u32 phy_uid_mask);
int phy_register_fixup_for_id(const char *phy_id);
```

19.12 Standards

IEEE Standard 802.3: CSMA/CD Access Method and Physical Layer Specifications, Section Two: http://standards.ieee.org/getieee802/download/802.3-2008_section2.pdf

RGMII v1.3: http://web.archive.org/web/20160303212629/http://www.hp.com/rnd/pdfs/RGMIIv1_3.pdf

RGMII v2.0: http://web.archive.org/web/20160303171328/http://www.hp.com/rnd/pdfs/RGMIIv2_0_final_hp.pdf

PHYLINK

20.1 Overview

phylink is a mechanism to support hot-pluggable networking modules directly connected to a MAC without needing to re-initialise the adapter on hot-plug events.

phylink supports conventional phylib-based setups, fixed link setups and SFP (Small Formfactor Pluggable) modules at present.

20.2 Modes of operation

phylink has several modes of operation, which depend on the firmware settings.

1. PHY mode

In PHY mode, we use phylib to read the current link settings from the PHY, and pass them to the MAC driver. We expect the MAC driver to configure exactly the modes that are specified without any negotiation being enabled on the link.

2. Fixed mode

Fixed mode is the same as PHY mode as far as the MAC driver is concerned.

3. In-band mode

In-band mode is used with 802.3z, SGMII and similar interface modes, and we are expecting to use and honor the in-band negotiation or control word sent across the serdes channel.

By example, what this means is that:

```
&eth {  
    phy = <&phy>;  
    phy-mode = "sgmii";  
};
```

does not use in-band SGMII signalling. The PHY is expected to follow exactly the settings given to it in its *mac_config()* function. The link should be forced up or down appropriately in the *mac_link_up()* and *mac_link_down()* functions.

```
&eth {  
    managed = "in-band-status";
```

```
phy = <&phy>;
phy-mode = "sgmii";
};
```

uses in-band mode, where results from the PHY's negotiation are passed to the MAC through the SGMII control word, and the MAC is expected to acknowledge the control word. The `mac_link_up()` and `mac_link_down()` functions must not force the MAC side link up and down.

20.3 Rough guide to converting a network driver to sfp/phylink

This guide briefly describes how to convert a network driver from phylib to the sfp/phylink support. Please send patches to improve this documentation.

1. Optionally split the network driver's phylib update function into two parts dealing with link-down and link-up. This can be done as a separate preparation commit.

An older example of this preparation can be found in git [commit fc548b991fb0](#), although this was splitting into three parts; the link-up part now includes configuring the MAC for the link settings. Please see `mac_link_up()` for more information on this.

2. Replace:

```
select FIXED_PHY
select PHYLIB
```

with:

```
select PHYLINK
```

in the driver's Kconfig stanza.

3. Add:

```
#include <linux/phylink.h>
```

to the driver's list of header files.

4. Add:

```
struct phylink *phylink;
struct phylink_config phylink_config;
```

to the driver's private data structure. We shall refer to the driver's private data pointer as `priv` below, and the driver's private data structure as `struct foo_priv`.

5. Replace the following functions:

Original function	Replacement function
phy_start(phydev)	phylink_start(priv->phylink)
phy_stop(phydev)	phylink_stop(priv->phylink)
phy_mii_ioctl(phydev, ifr, cmd)	phylink_mii_ioctl(priv->phylink, ifr, cmd)
phy_ethtool_get_wol(phydev, wol)	phylink_ethtool_get_wol(priv->phylink, wol)
phy_ethtool_set_wol(phydev, wol)	phylink_ethtool_set_wol(priv->phylink, wol)
phy_disconnect(phydev)	phylink_disconnect_phy(priv->phylink)

Please note that some of these functions must be called under the rtnl lock, and will warn if not. This will normally be the case, except if these are called from the driver suspend/resume paths.

6. Add/replace ksettings get/set methods with:

```
static int foo_ethtool_set_link_ksettings(struct net_device *dev,
                                         const struct ethtool_link_
                                         ↵ksettings *cmd)
{
    struct foo_priv *priv = netdev_priv(dev);

    return phylink_ethtool_ksettings_set(priv->phylink, cmd);
}

static int foo_ethtool_get_link_ksettings(struct net_device *dev,
                                         struct ethtool_link_ksettings_
                                         ↵*cmd)
{
    struct foo_priv *priv = netdev_priv(dev);

    return phylink_ethtool_ksettings_get(priv->phylink, cmd);
}
```

7. Replace the call to:

```
phy_dev = of_phy_connect(dev, node, link_func, flags, phy_interface);
```

and associated code with a call to:

```
err = phylink_of_phy_connect(priv->phylink, node, flags);
```

For the most part, flags can be zero; these flags are passed to the `phy_attach_direct()` inside this function call if a PHY is specified in the DT node node.

node should be the DT node which contains the network phy property, fixed link properties, and will also contain the sfp property.

The setup of fixed links should also be removed; these are handled internally by phylink.

`of_phy_connect()` was also passed a function pointer for link updates. This function is replaced by a different form of MAC updates described below in (8).

Manipulation of the PHY's supported/advertised happens within phylink based on the validate callback, see below in (8).

Note that the driver no longer needs to store the `phy_interface`, and also note that `phy_interface` becomes a dynamic property, just like the speed, duplex etc. settings.

Finally, note that the MAC driver has no direct access to the PHY anymore; that is because in the phylink model, the PHY can be dynamic.

8. Add a `struct phylink_mac_ops` instance to the driver, which is a table of function pointers, and implement these functions. The old link update function for `of_phy_connect()` becomes three methods: `mac_link_up()`, `mac_link_down()`, and `mac_config()`. If step 1 was performed, then the functionality will have been split there.

It is important that if in-band negotiation is used, `mac_link_up()` and `mac_link_down()` do not prevent the in-band negotiation from completing, since these functions are called when the in-band link state changes - otherwise the link will never come up.

The `mac_get_caps()` method is optional, and if provided should return the phylink MAC capabilities that are supported for the passed interface mode. In general, there is no need to implement this method. Phylink will use these capabilities in combination with permissible capabilities for interface to determine the allowable ethtool link modes.

The `mac_link_state()` method is used to read the link state from the MAC, and report back the settings that the MAC is currently using. This is particularly important for in-band negotiation methods such as 1000base-X and SGMII.

The `mac_link_up()` method is used to inform the MAC that the link has come up. The call includes the negotiation mode and interface for reference only. The finalised link parameters are also supplied (speed, duplex and flow control/pause enablement settings) which should be used to configure the MAC when the MAC and PCS are not tightly integrated, or when the settings are not coming from in-band negotiation.

The `mac_config()` method is used to update the MAC with the requested state, and must avoid unnecessarily taking the link down when making changes to the MAC configuration. This means the function should modify the state and only take the link down when absolutely necessary to change the MAC configuration. An example of how to do this can be found in `mvneta_mac_config()` in `drivers/net/ethernet/marvell/mvneta.c`.

For further information on these methods, please see the inline documentation in `struct phylink_mac_ops`.

9. Remove calls to `of_parse_phandle()` for the PHY, `of_phy_register_fixed_link()` for fixed links etc. from the probe function, and replace with:

```
struct phylink *phylink;
priv->phylink_config.dev = &dev.dev;
priv->phylink_config.type = PHYLINK_NETDEV;

phylink = phylink_create(&priv->phylink_config, node, phy_mode, &phylink_
    .ops);
if (IS_ERR(phylink)) {
    err = PTR_ERR(phylink);
    fail_probe;
}

priv->phylink = phylink;
```

and arrange to destroy the phylink in the probe failure path as appropriate and the removal path too by calling:

```
phylink_destroy(priv->phylink);
```

10. Arrange for MAC link state interrupts to be forwarded into phylink, via:

```
phylink_mac_change(priv->phylink, link_is_up);
```

where `link_is_up` is true if the link is currently up or false otherwise. If a MAC is unable to provide these interrupts, then it should set `priv->phylink_config.pcs_poll = true;` in step 9.

11. Verify that the driver does not call:

```
netif_carrier_on()  
netif_carrier_off()
```

as these will interfere with phylink's tracking of the link state, and cause phylink to omit calls via the `mac_link_up()` and `mac_link_down()` methods.

Network drivers should call `phylink_stop()` and `phylink_start()` via their suspend/resume paths, which ensures that the appropriate `struct phylink_mac_ops` methods are called as necessary.

For information describing the SFP cage in DT, please see the binding documentation in the kernel source tree `Documentation/devicetree/bindings/net/sff,sfp.yaml`.

IP-ALIASING

IP-aliases are an obsolete way to manage multiple IP-addresses/masks per interface. Newer tools such as iproute2 support multiple address/prefixes per interface, but aliases are still supported for backwards compatibility.

An alias is formed by adding a colon and a string when running ifconfig. This string is usually numeric, but this is not a must.

21.1 Alias creation

Alias creation is done by 'magic' interface naming: eg. to create a 200.1.1.1 alias for eth0 ...

```
# ifconfig eth0:0 200.1.1.1 etc,etc....  
~~ -> request alias #0 creation (if not yet exists) for eth0
```

The corresponding route is also set up by this command. Please note: The route always points to the base interface.

21.2 Alias deletion

The alias is removed by shutting the alias down:

```
# ifconfig eth0:0 down  
~~~~~ -> will delete alias
```

21.3 Alias (re-)configuring

Aliases are not real devices, but programs should be able to configure and refer to them as usual (ifconfig, route, etc).

21.4 Relationship with main device

If the base device is shut down the added aliases will be deleted too.

ETHERNET BRIDGING

22.1 Introduction

The IEEE 802.1Q-2022 (Bridges and Bridged Networks) standard defines the operation of bridges in computer networks. A bridge, in the context of this standard, is a device that connects two or more network segments and operates at the data link layer (Layer 2) of the OSI (Open Systems Interconnection) model. The purpose of a bridge is to filter and forward frames between different segments based on the destination MAC (Media Access Control) address.

22.2 Bridge kAPI

Here are some core structures of bridge code. Note that the kAPI is *unstable*, and can be changed at any time.

```
struct net_bridge_vlan
    per-vlan entry
```

Definition:

```
struct net_bridge_vlan {
    struct rhash_head           vnode;
    struct rhash_head           tnode;
    u16 vid;
    u16 flags;
    u16 priv_flags;
    u8 state;
    struct pcpu_sw_netstats __percpu *stats;
    union {
        struct net_bridge      *br;
        struct net_bridge_port *port;
    };
    union {
        refcount_t refcnt;
        struct net_bridge_vlan *brvlan;
    };
    struct br_tunnel_info        tinfo;
    union {
        struct net_bridge_mcast   br_mcast_ctx;
        struct net_bridge_mcast_port port_mcast_ctx;
    };
}
```

```

};

u16 msti;
struct list_head vlist;
struct rcu_head rcu;
};

```

Members

vnode
rhashtable member

tnode
rhashtable member

vid
VLAN id

flags
bridge vlan flags

priv_flags
private (in-kernel) bridge vlan flags

state
STP state (e.g. blocking, learning, forwarding)

stats
per-cpu VLAN statistics

{unnamed_union}
anonymous

br
if MASTER flag set, this points to a bridge struct

port
if MASTER flag unset, this points to a port struct

{unnamed_union}
anonymous

refcnt
if MASTER flag set, this is bumped for each port referencing it

brvlan
if MASTER flag unset, this points to the global per-VLAN context for this VLAN entry

tinfo
bridge tunnel info

{unnamed_union}
anonymous

br_mcast_ctx
if MASTER flag set, this is the global vlan multicast context

port_mcast_ctx
if MASTER flag unset, this is the per-port/vlan multicast context

msti

if MASTER flag set, this holds the VLANs MST instance

vlist

sorted list of VLAN entries

rcu

used for entry destruction

Description

This structure is shared between the global per-VLAN entries contained in the bridge rhashtable and the local per-port per-VLAN entries contained in the port's rhashtable. The union entries should be interpreted depending on the entry flags that are set.

22.3 Bridge uAPI

Modern Linux bridge uAPI is accessed via Netlink interface. You can find below files where the bridge and bridge port netlink attributes are defined.

22.3.1 Bridge netlink attributes

Please note that the timer values in the following section are expected in `clock_t` format, which is seconds multiplied by `USER_HZ` (generally defined as 100).

IFLA_BR_FORWARD_DELAY

The bridge forwarding delay is the time spent in LISTENING state (before moving to LEARNING) and in LEARNING state (before moving to FORWARDING). Only relevant if STP is enabled.

The valid values are between $(2 * \text{USER_HZ})$ and $(30 * \text{USER_HZ})$. The default value is $(15 * \text{USER_HZ})$.

IFLA_BR_HELLO_TIME

The time between hello packets sent by the bridge, when it is a root bridge or a designated bridge. Only relevant if STP is enabled.

The valid values are between $(1 * \text{USER_HZ})$ and $(10 * \text{USER_HZ})$. The default value is $(2 * \text{USER_HZ})$.

IFLA_BR_MAX_AGE

The hello packet timeout is the time until another bridge in the spanning tree is assumed to be dead, after reception of its last hello message. Only relevant if STP is enabled.

The valid values are between $(6 * \text{USER_HZ})$ and $(40 * \text{USER_HZ})$. The default value is $(20 * \text{USER_HZ})$.

IFLA_BR_AGEING_TIME

Configure the bridge's FDB entries aging time. It is the time a MAC address will be kept in the FDB after a packet has been received from that address. After this time has passed, entries are cleaned up. Allow values outside the 802.1 standard specification for special cases:

- 0 - entry never ages (all permanent)
- 1 - entry disappears (no persistence)

The default value is (300 * USER_HZ).

IFLA_BR_STP_STATE

Turn spanning tree protocol on (*IFLA_BR_STP_STATE* > 0) or off (*IFLA_BR_STP_STATE* == 0) for this bridge.

The default value is 0 (disabled).

IFLA_BR_PRIORITY

Set this bridge's spanning tree priority, used during STP root bridge election.

The valid values are between 0 and 65535.

IFLA_BR_VLAN_FILTERING

Turn VLAN filtering on (*IFLA_BR_VLAN_FILTERING* > 0) or off (*IFLA_BR_VLAN_FILTERING* == 0). When disabled, the bridge will not consider the VLAN tag when handling packets.

The default value is 0 (disabled).

IFLA_BR_VLAN_PROTOCOL

Set the protocol used for VLAN filtering.

The valid values are 0x8100(802.1Q) or 0x88A8(802.1AD). The default value is 0x8100(802.1Q).

IFLA_BR_GROUP_FWD_MASK

The group forwarding mask. This is the bitmask that is applied to decide whether to forward incoming frames destined to link-local addresses (of the form 01:80:C2:00:00:0X).

The default value is 0, which means the bridge does not forward any link-local frames coming on this port.

IFLA_BR_ROOT_ID

The bridge root id, read only.

IFLA_BR_BRIDGE_ID

The bridge id, read only.

IFLA_BR_ROOT_PORT

The bridge root port, read only.

IFLA_BR_ROOT_PATH_COST

The bridge root path cost, read only.

IFLA_BR_TOPOLOGY_CHANGE

The bridge topology change, read only.

IFLA_BR_TOPOLOGY_CHANGE_DETECTED

The bridge topology change detected, read only.

IFLA_BR_HELLO_TIMER

The bridge hello timer, read only.

IFLA_BR_TCN_TIMER

The bridge tcn timer, read only.

IFLA_BR_TOPOLOGY_CHANGE_TIMER

The bridge topology change timer, read only.

IFLA_BR_GC_TIMER

The bridge gc timer, read only.

IFLA_BR_GROUP_ADDR

Set the MAC address of the multicast group this bridge uses for STP. The address must be a link-local address in standard Ethernet MAC address format. It is an address of the form 01:80:C2:00:00:0X, with X in [0, 4..f].

The default value is 0.

IFLA_BR_FDB_FLUSH

Flush bridge's fdb dynamic entries.

IFLA_BR_MCAST_ROUTER

Set bridge's multicast router if IGMP snooping is enabled. The valid values are:

- 0 - disabled.
- 1 - automatic (queried).
- 2 - permanently enabled.

The default value is 1.

IFLA_BR_MCAST_SNOOPING

Turn multicast snooping on (*IFLA_BR_MCAST_SNOOPING > 0*) or off (*IFLA_BR_MCAST_SNOOPING == 0*).

The default value is 1.

IFLA_BR_MCAST_QUERY_USE_IFADDR

If enabled use the bridge's own IP address as source address for IGMP queries (*IFLA_BR_MCAST_QUERY_USE_IFADDR > 0*) or the default of 0.0.0.0 (*IFLA_BR_MCAST_QUERY_USE_IFADDR == 0*).

The default value is 0 (disabled).

IFLA_BR_MCAST_QUERIER

Enable (*IFLA_BR_MULTICAST_QUERIER > 0*) or disable (*IFLA_BR_MULTICAST_QUERIER == 0*) IGMP querier, ie sending of multicast queries by the bridge.

The default value is 0 (disabled).

IFLA_BR_MCAST_HASH_ELASTICITY

Set multicast database hash elasticity, It is the maximum chain length in the multicast hash table. This attribute is *deprecated* and the value is always 16.

IFLA_BR_MCAST_HASH_MAX

Set maximum size of the multicast hash table

The default value is 4096, the value must be a power of 2.

IFLA_BR_MCAST_LAST_MEMBER_CNT

The Last Member Query Count is the number of Group-Specific Queries sent before the router assumes there are no local members. The Last Member Query Count is also the number of Group-and-Source-Specific Queries sent before the router assumes there are no listeners for a particular source.

The default value is 2.

IFLA_BR_MCAST_STARTUP_QUERY_CNT

The Startup Query Count is the number of Queries sent out on startup, separated by the Startup Query Interval.

The default value is 2.

IFLA_BR_MCAST_LAST_MEMBER_INTVL

The Last Member Query Interval is the Max Response Time inserted into Group-Specific Queries sent in response to Leave Group messages, and is also the amount of time between Group-Specific Query messages.

The default value is (1 * USER_HZ).

IFLA_BR_MCAST_MEMBERSHIP_INTVL

The interval after which the bridge will leave a group, if no membership reports for this group are received.

The default value is (260 * USER_HZ).

IFLA_BR_MCAST_QUERIER_INTVL

The interval between queries sent by other routers. If no queries are seen after this delay has passed, the bridge will start to send its own queries (as if *IFLA_BR_MCAST_QUERIER_INTVL* was enabled).

The default value is (255 * USER_HZ).

IFLA_BR_MCAST_QUERY_INTVL

The Query Interval is the interval between General Queries sent by the Querier.

The default value is (125 * USER_HZ). The minimum value is (1 * USER_HZ).

IFLA_BR_MCAST_QUERY_RESPONSE_INTVL

The Max Response Time used to calculate the Max Resp Code inserted into the periodic General Queries.

The default value is (10 * USER_HZ).

IFLA_BR_MCAST_STARTUP_QUERY_INTVL

The interval between queries in the startup phase.

The default value is (125 * USER_HZ) / 4. The minimum value is (1 * USER_HZ).

IFLA_BR_NF_CALL_IPTABLES

Enable (*NF_CALL_IPTABLES* > 0) or disable (*NF_CALL_IPTABLES* == 0) iptables hooks on the bridge.

The default value is 0 (disabled).

IFLA_BR_NF_CALL_IP6TABLES

Enable (*NF_CALL_IP6TABLES* > 0) or disable (*NF_CALL_IP6TABLES* == 0) ip6tables hooks on the bridge.

The default value is 0 (disabled).

IFLA_BR_NF_CALL_ARPTABLES

Enable (*NF_CALL_ARPTABLES* > 0) or disable (*NF_CALL_ARPTABLES* == 0) arptables hooks on the bridge.

The default value is 0 (disabled).

IFLA_BR_VLAN_DEFAULT_PVID

VLAN ID applied to untagged and priority-tagged incoming packets.

The default value is 1. Setting to the special value 0 makes all ports of this bridge not have a PVID by default, which means that they will not accept VLAN-untagged traffic.

IFLA_BR_PAD

Bridge attribute padding type for netlink message.

IFLA_BR_VLAN_STATS_ENABLED

Enable (*IFLA_BR_VLAN_STATS_ENABLED* == 1) or disable (*IFLA_BR_VLAN_STATS_ENABLED* == 0) per-VLAN stats accounting.

The default value is 0 (disabled).

IFLA_BR_MCAST_STATS_ENABLED

Enable (*IFLA_BR_MCAST_STATS_ENABLED* > 0) or disable (*IFLA_BR_MCAST_STATS_ENABLED* == 0) multicast (IGMP/MLD) stats accounting.

The default value is 0 (disabled).

IFLA_BR_MCAST_IGMP_VERSION

Set the IGMP version.

The valid values are 2 and 3. The default value is 2.

IFLA_BR_MCAST_MLD_VERSION

Set the MLD version.

The valid values are 1 and 2. The default value is 1.

IFLA_BR_VLAN_STATS_PER_PORT

Enable (*IFLA_BR_VLAN_STATS_PER_PORT* == 1) or disable (*IFLA_BR_VLAN_STATS_PER_PORT* == 0) per-VLAN per-port stats accounting. Can be changed only when there are no port VLANs configured.

The default value is 0 (disabled).

IFLA_BR_MULTI_BOOLOPT

The multi_booopt is used to control new boolean options to avoid adding new netlink attributes. You can look at enum `br_booopt_id` for those options.

IFLA_BR_MCAST_QUERIER_STATE

Bridge mcast querier states, read only.

IFLA_BR_FDB_N_LEARNED

The number of dynamically learned FDB entries for the current bridge, read only.

IFLA_BR_FDB_MAX_LEARNED

Set the number of max dynamically learned FDB entries for the current bridge.

22.3.2 Bridge port netlink attributes

IFLA_BRPORT_STATE

The operation state of the port. Here are the valid values.

- 0 - port is in STP *DISABLED* state. Make this port completely inactive for STP. This is also called BPDU filter and could be used to disable STP on an untrusted port, like a leaf virtual device. The traffic forwarding is also stopped on this port.
- 1 - port is in STP *LISTENING* state. Only valid if STP is enabled on the bridge. In this state the port listens for STP BPDUs and drops all other traffic frames.
- 2 - port is in STP *LEARNING* state. Only valid if STP is enabled on the bridge. In this state the port will accept traffic only for the purpose of updating MAC address tables.
- 3 - port is in STP *FORWARDING* state. Port is fully active.
- 4 - port is in STP *BLOCKING* state. Only valid if STP is enabled on the bridge. This state is used during the STP election process. In this state, port will only process STP BPDUs.

IFLA_BRPORT_PRIORITY

The STP port priority. The valid values are between 0 and 255.

IFLA_BRPORT_COST

The STP path cost of the port. The valid values are between 1 and 65535.

IFLA_BRPORT_MODE

Set the bridge port mode. See *BRIDGE_MODE_HAIRPIN* for more details.

IFLA_BRPORT_GUARD

Controls whether STP BPDUs will be processed by the bridge port. By default, the flag is turned off to allow BPDU processing. Turning this flag on will disable the bridge port if a STP BPDU packet is received.

If the bridge has Spanning Tree enabled, hostile devices on the network may send BPDU on a port and cause network failure. Setting *guard on* will detect and stop this by disabling the port. The port will be restarted if the link is brought down, or removed and reattached.

IFLA_BRPORT_PROTECT

Controls whether a given port is allowed to become a root port or not. Only used when STP is enabled on the bridge. By default the flag is off.

This feature is also called root port guard. If BPDU is received from a leaf (edge) port, it should not be elected as root port. This could be used if using STP on a bridge and the downstream bridges are not fully trusted; this prevents a hostile guest from rerouting traffic.

IFLA_BRPORT_FAST_LEAVE

This flag allows the bridge to immediately stop multicast traffic forwarding on a port that receives an IGMP Leave message. It is only used when IGMP snooping is enabled on the bridge. By default the flag is off.

IFLA_BRPORT_LEARNING

Controls whether a given port will learn *source* MAC addresses from received traffic or not. Also controls whether dynamic FDB entries (which can also be added by software) will be refreshed by incoming traffic. By default this flag is on.

IFLA_BRPORT_UNICAST_FLOOD

Controls whether unicast traffic for which there is no FDB entry will be flooded towards this port. By default this flag is on.

IFLA_BRPORT_PROXYARP

Enable proxy ARP on this port.

IFLA_BRPORT_LEARNING_SYNC

Controls whether a given port will sync MAC addresses learned on device port to bridge FDB.

IFLA_BRPORT_PROXYARP_WIFI

Enable proxy ARP on this port which meets extended requirements by IEEE 802.11 and Hotspot 2.0 specifications.

IFLA_BRPORT_ROOT_ID**IFLA_BRPORT_BRIDGE_ID****IFLA_BRPORT_DESIGNATED_PORT****IFLA_BRPORT_DESIGNATED_COST****IFLA_BRPORT_ID****IFLA_BRPORT_NO****IFLA_BRPORT_TOPOLOGY_CHANGE_ACK****IFLA_BRPORT_CONFIG_PENDING****IFLA_BRPORT_MESSAGE_AGE_TIMER****IFLA_BRPORT_FORWARD_DELAY_TIMER****IFLA_BRPORT_HOLD_TIMER****IFLA_BRPORT_FLUSH**

Flush bridge ports' fdb dynamic entries.

IFLA_BRPORT_MULTICAST_ROUTER

Configure the port's multicast router presence. A port with a multicast router will receive all multicast traffic. The valid values are:

- 0 disable multicast routers on this port
- 1 let the system detect the presence of routers (default)
- 2 permanently enable multicast traffic forwarding on this port
- **3 enable multicast routers temporarily on this port, not depending on incoming queries.**

IFLA_BRPORT_PAD**IFLA_BRPORT_MCAST_FLOOD**

Controls whether a given port will flood multicast traffic for which there is no MDB entry. By default this flag is on.

IFLA_BRPORT_MCAST_TO_UCAST

Controls whether a given port will replicate packets using unicast instead of multicast. By default this flag is off.

This is done by copying the packet per host and changing the multicast destination MAC to a unicast one accordingly.

mcast_to_unicast works on top of the multicast snooping feature of the bridge. Which means unicast copies are only delivered to hosts which are interested in unicast and signaled this via IGMP/MLD reports previously.

This feature is intended for interface types which have a more reliable and/or efficient way to deliver unicast packets than broadcast ones (e.g. WiFi).

However, it should only be enabled on interfaces where no IGMPv2/MLDv1 report suppression takes place. IGMP/MLD report suppression issue is usually overcome by the network daemon (supplicant) enabling AP isolation and by that separating all STAs.

Delivery of STA-to-STA IP multicast is made possible again by enabling and utilizing the bridge hairpin mode, which considers the incoming port as a potential outgoing port, too (see *BRIDGE_MODE_HAIRPIN* option). Hairpin mode is performed after multicast snooping, therefore leading to only deliver reports to STAs running a multicast router.

IFLA_BRPORT_VLAN_TUNNEL

Controls whether vlan to tunnel mapping is enabled on the port. By default this flag is off.

IFLA_BRPORT_BCAST_FLOOD

Controls flooding of broadcast traffic on the given port. By default this flag is on.

IFLA_BRPORT_GROUP_FWD_MASK

Set the group forward mask. This is a bitmask that is applied to decide whether to forward incoming frames destined to link-local addresses. The addresses of the form are 01:80:C2:00:00:0X (defaults to 0, which means the bridge does not forward any link-local frames coming on this port).

IFLA_BRPORT_NEIGH_SUPPRESS

Controls whether neighbor discovery (arp and nd) proxy and suppression is enabled on the port. By default this flag is off.

IFLA_BRPORT_ISOLATED

Controls whether a given port will be isolated, which means it will be able to communicate with non-isolated ports only. By default this flag is off.

IFLA_BRPORT_BACKUP_PORT

Set a backup port. If the port loses carrier all traffic will be redirected to the configured backup port. Set the value to 0 to disable it.

IFLA_BRPORT_MRP_RING_OPEN**IFLA_BRPORT_MRP_IN_OPEN****IFLA_BRPORT_MCAST_EHT_HOSTS_LIMIT**

The number of per-port EHT hosts limit. The default value is 512. Setting to 0 is not allowed.

IFLA_BRPORT_MCAST_EHT_HOSTS_CNT

The current number of tracked hosts, read only.

IFLA_BRPORT_LOCKED

Controls whether a port will be locked, meaning that hosts behind the port will not be able to communicate through the port unless an FDB entry with the unit's MAC address is in the FDB. The common use case is that hosts are allowed access through authentication with the IEEE 802.1X protocol or based on whitelists. By default this flag is off.

Please note that secure 802.1X deployments should always use the `BR_BOLOPT_NO_LL_LEARN` flag, to not permit the bridge to populate its FDB based on link-local (EAPOL) traffic received on the port.

IFLA_BRPORT_MAB

Controls whether a port will use MAC Authentication Bypass (MAB), a technique through which select MAC addresses may be allowed on a locked port, without using 802.1X authentication. Packets with an unknown source MAC address generates a "locked" FDB entry on the incoming bridge port. The common use case is for user space to react to these bridge FDB notifications and optionally replace the locked FDB entry with a normal one, allowing traffic to pass for whitelisted MAC addresses.

Setting this flag also requires `IFLA_BRPORT_LOCKED` and `IFLA_BRPORT_LEARNING`. `IFLA_BRPORT_LOCKED` ensures that unauthorized data packets are dropped, and `IFLA_BRPORT_LEARNING` allows the dynamic FDB entries installed by user space (as replacements for the locked FDB entries) to be refreshed and/or aged out.

IFLA_BRPORT_MCAST_N_GROUPS

IFLA_BRPORT_MCAST_MAX_GROUPS

Sets the maximum number of MDB entries that can be registered for a given port. Attempts to register more MDB entries at the port than this limit allows will be rejected, whether they are done through netlink (e.g. the bridge tool), or IGMP or MLD membership reports. Setting a limit of 0 disables the limit. The default value is 0.

IFLA_BRPORT_NEIGH_VLAN_SUPPRESS

Controls whether neighbor discovery (arp and nd) proxy and suppression is enabled for a given port. By default this flag is off.

Note that this option only takes effect when `IFLA_BRPORT_NEIGH_SUPPRESS` is enabled for a given port.

IFLA_BRPORT_BACKUP_NHID

The FDB nexthop object ID to attach to packets being redirected to a backup port that has VLAN tunnel mapping enabled (via the `IFLA_BRPORT_VLAN_TUNNEL` option). Setting a value of 0 (default) has the effect of not attaching any ID.

22.3.3 Bridge sysfs

The sysfs interface is deprecated and should not be extended if new options are added.

22.4 STP

The STP (Spanning Tree Protocol) implementation in the Linux bridge driver is a critical feature that helps prevent loops and broadcast storms in Ethernet networks by identifying and disabling redundant links. In a Linux bridge context, STP is crucial for network stability and availability.

STP is a Layer 2 protocol that operates at the Data Link Layer of the OSI model. It was originally developed as IEEE 802.1D and has since evolved into multiple versions, including Rapid Spanning Tree Protocol (RSTP) and [Multiple Spanning Tree Protocol \(MSTP\)](#).

The 802.1D-2004 removed the original Spanning Tree Protocol, instead incorporating the Rapid Spanning Tree Protocol (RSTP). By 2014, all the functionality defined by IEEE 802.1D has been

incorporated into either IEEE 802.1Q (Bridges and Bridged Networks) or IEEE 802.1AC (MAC Service Definition). 802.1D has been officially withdrawn in 2022.

22.4.1 Bridge Ports and STP States

In the context of STP, bridge ports can be in one of the following states:

- Blocking: The port is disabled for data traffic and only listens for BPDUs (Bridge Protocol Data Units) from other devices to determine the network topology.
- Listening: The port begins to participate in the STP process and listens for BPDUs.
- Learning: The port continues to listen for BPDUs and begins to learn MAC addresses from incoming frames but does not forward data frames.
- Forwarding: The port is fully operational and forwards both BPDUs and data frames.
- Disabled: The port is administratively disabled and does not participate in the STP process. The data frames forwarding are also disabled.

22.4.2 Root Bridge and Convergence

In the context of networking and Ethernet bridging in Linux, the root bridge is a designated switch in a bridged network that serves as a reference point for the spanning tree algorithm to create a loop-free topology.

Here's how the STP works and root bridge is chosen:

1. Bridge Priority: Each bridge running a spanning tree protocol, has a configurable Bridge Priority value. The lower the value, the higher the priority. By default, the Bridge Priority is set to a standard value (e.g., 32768).
2. Bridge ID: The Bridge ID is composed of two components: Bridge Priority and the MAC address of the bridge. It uniquely identifies each bridge in the network. The Bridge ID is used to compare the priorities of different bridges.
3. Bridge Election: When the network starts, all bridges initially assume that they are the root bridge. They start advertising Bridge Protocol Data Units (BPDU) to their neighbors, containing their Bridge ID and other information.
4. BPDU Comparison: Bridges exchange BPDUs to determine the root bridge. Each bridge examines the received BPDUs, including the Bridge Priority and Bridge ID, to determine if it should adjust its own priorities. The bridge with the lowest Bridge ID will become the root bridge.
5. Root Bridge Announcement: Once the root bridge is determined, it sends BPDUs with information about the root bridge to all other bridges in the network. This information is used by other bridges to calculate the shortest path to the root bridge and, in doing so, create a loop-free topology.
6. Forwarding Ports: After the root bridge is selected and the spanning tree topology is established, each bridge determines which of its ports should be in the forwarding state (used for data traffic) and which should be in the blocking state (used to prevent loops). The root bridge's ports are all in the forwarding state, while other bridges have some ports in the blocking state to avoid loops.

7. Root Ports: After the root bridge is selected and the spanning tree topology is established, each non-root bridge processes incoming BPDUs and determines which of its ports provides the shortest path to the root bridge based on the information in the received BPDUs. This port is designated as the root port. And it is in the Forwarding state, allowing it to actively forward network traffic.
8. Designated ports: A designated port is the port through which the non-root bridge will forward traffic towards the designated segment. Designated ports are placed in the Forwarding state. All other ports on the non-root bridge that are not designated for specific segments are placed in the Blocking state to prevent network loops.

STP ensures network convergence by calculating the shortest path and disabling redundant links. When network topology changes occur (e.g., a link failure), STP recalculates the network topology to restore connectivity while avoiding loops.

Proper configuration of STP parameters, such as the bridge priority, can influence network performance, path selection and which bridge becomes the Root Bridge.

22.4.3 User space STP helper

The user space STP helper *bridge-stp* is a program to control whether to use user mode spanning tree. The `/sbin/bridge-stp <bridge> <start|stop>` is called by the kernel when STP is enabled/disabled on a bridge (via `brctl stp <bridge> <on|off>` or `ip link set <bridge> type bridge stp_state <0|1>`). The kernel enables `user_stp` mode if that command returns 0, or enables `kernel_stp` mode if that command returns any other value.

22.5 VLAN

A LAN (Local Area Network) is a network that covers a small geographic area, typically within a single building or a campus. LANs are used to connect computers, servers, printers, and other networked devices within a localized area. LANs can be wired (using Ethernet cables) or wireless (using Wi-Fi).

A VLAN (Virtual Local Area Network) is a logical segmentation of a physical network into multiple isolated broadcast domains. VLANs are used to divide a single physical LAN into multiple virtual LANs, allowing different groups of devices to communicate as if they were on separate physical networks.

Typically there are two VLAN implementations, IEEE 802.1Q and IEEE 802.1ad (also known as QinQ). IEEE 802.1Q is a standard for VLAN tagging in Ethernet networks. It allows network administrators to create logical VLANs on a physical network and tag Ethernet frames with VLAN information, which is called *VLAN-tagged frames*. IEEE 802.1ad, commonly known as QinQ or Double VLAN, is an extension of the IEEE 802.1Q standard. QinQ allows for the stacking of multiple VLAN tags within a single Ethernet frame. The Linux bridge supports both the IEEE 802.1Q and [802.1AD](#) protocol for VLAN tagging.

[VLAN filtering](#) on a bridge is disabled by default. After enabling VLAN filtering on a bridge, it will start forwarding frames to appropriate destinations based on their destination MAC address and VLAN tag (both must match).

22.6 Multicast

The Linux bridge driver has multicast support allowing it to process Internet Group Management Protocol (IGMP) or Multicast Listener Discovery (MLD) messages, and to efficiently forward multicast data packets. The bridge driver supports IGMPv2/IGMPv3 and MLDv1/MLDv2.

22.6.1 Multicast snooping

Multicast snooping is a networking technology that allows network switches to intelligently manage multicast traffic within a local area network (LAN).

The switch maintains a multicast group table, which records the association between multicast group addresses and the ports where hosts have joined these groups. The group table is dynamically updated based on the IGMP/MLD messages received. With the multicast group information gathered through snooping, the switch optimizes the forwarding of multicast traffic. Instead of blindly broadcasting the multicast traffic to all ports, it sends the multicast traffic based on the destination MAC address only to ports which have subscribed the respective destination multicast group.

When created, the Linux bridge devices have multicast snooping enabled by default. It maintains a Multicast forwarding database (MDB) which keeps track of port and group relationships.

22.6.2 IGMPv3/MLDv2 EHT support

The Linux bridge supports IGMPv3/MLDv2 EHT (Explicit Host Tracking), which was added by [474ddb37fa3a](#) ("net: bridge: multicast: add EHT allow/block handling")

The explicit host tracking enables the device to keep track of each individual host that is joined to a particular group or channel. The main benefit of the explicit host tracking in IGMP is to allow minimal leave latencies when a host leaves a multicast group or channel.

The length of time between a host wanting to leave and a device stopping traffic forwarding is called the IGMP leave latency. A device configured with IGMPv3 or MLDv2 and explicit tracking can immediately stop forwarding traffic if the last host to request to receive traffic from the device indicates that it no longer wants to receive traffic. The leave latency is thus bound only by the packet transmission latencies in the multiaccess network and the processing time in the device.

22.6.3 Other multicast features

The Linux bridge also supports per-VLAN multicast snooping, which is disabled by default but can be enabled. And [Multicast Router Discovery](#), which help identify the location of multicast routers.

22.7 Switchdev

Linux Bridge Switchdev is a feature in the Linux kernel that extends the capabilities of the traditional Linux bridge to work more efficiently with hardware switches that support switchdev. With Linux Bridge Switchdev, certain networking functions like forwarding, filtering, and learning of Ethernet frames can be offloaded to a hardware switch. This offloading reduces the burden on the Linux kernel and CPU, leading to improved network performance and lower latency.

To use Linux Bridge Switchdev, you need hardware switches that support the switchdev interface. This means that the switch hardware needs to have the necessary drivers and functionality to work in conjunction with the Linux kernel.

Please see the [Ethernet switch device driver model \(switchdev\)](#) document for more details.

22.8 Netfilter

The bridge netfilter module is a legacy feature that allows to filter bridged packets with iptables and ip6tables. Its use is discouraged. Users should consider using nftables for packet filtering.

The older ebtables tool is more feature-limited compared to nftables, but just like nftables it doesn't need this module either to function.

The br_nf module intercepts packets entering the bridge, performs minimal sanity tests on ipv4 and ipv6 packets and then pretends that these packets are being routed, not bridged. br_nf then calls the ip and ipv6 netfilter hooks from the bridge layer, i.e. ip(6)tables rulesets will also see these packets.

br_nf is also the reason for the iptables *physdev* match: This match is the only way to reliably tell routed and bridged packets apart in an iptables ruleset.

Note that ebtables and nftables will work fine without the br_nf module. iptables/ip6tables/arptables do not work for bridged traffic because they plug in the routing stack. nftables rules in ip/ip6/inet/arp families won't see traffic that is forwarded by a bridge either, but that's very much how it should be.

Historically the feature set of ebtables was very limited (it still is), this module was added to pretend packets are routed and invoke the ipv4/ipv6 netfilter hooks from the bridge so users had access to the more feature-rich iptables matching capabilities (including conntrack). nftables doesn't have this limitation, pretty much all features work regardless of the protocol family.

So, br_nf is only needed if users, for some reason, need to use ip(6)tables to filter packets forwarded by the bridge, or NAT bridged traffic. For pure link layer filtering, this module isn't needed.

22.9 Other Features

The Linux bridge also supports IEEE 802.11 Proxy ARP, Media Redundancy Protocol (MRP), Media Redundancy Protocol (MRP) LC mode, IEEE 802.1X port authentication, and MAC Authentication Bypass (MAB).

22.10 FAQ

22.10.1 What does a bridge do?

A bridge transparently forwards traffic between multiple network interfaces. In plain English this means that a bridge connects two or more physical Ethernet networks, to form one larger (logical) Ethernet network.

22.10.2 Is it L3 protocol independent?

Yes. The bridge sees all frames, but it *uses* only L2 headers/information. As such, the bridging functionality is protocol independent, and there should be no trouble forwarding IPX, NetBEUI, IP, IPv6, etc.

22.11 Contact Info

The code is currently maintained by Roopa Prabhu <roopa@nvidia.com> and Nikolay Aleksandrov <razor@blackwall.org>. Bridge bugs and enhancements are discussed on the `linux-netdev` mailing list netdev@vger.kernel.org and bridge@lists.linux-foundation.org.

The list is open to anyone interested: <http://vger.kernel.org/vger-lists.html#netdev>

22.12 External Links

The old Documentation for Linux bridging is on: <https://wiki.linuxfoundation.org/networking/bridge>

SNMP COUNTER

This document explains the meaning of SNMP counters.

23.1 General IPv4 counters

All layer 4 packets and ICMP packets will change these counters, but these counters won't be changed by layer 2 packets (such as STP) or ARP packets.

- IpInReceives

Defined in [RFC1213 ipInReceives](#)

The number of packets received by the IP layer. It gets increasing at the beginning of `ip_rcv` function, always be updated together with `IpExtInOctets`. It will be increased even if the packet is dropped later (e.g. due to the IP header is invalid or the checksum is wrong and so on). It indicates the number of aggregated segments after GRO/LRO.

- IpInDelivers

Defined in [RFC1213 ipInDelivers](#)

The number of packets delivers to the upper layer protocols. E.g. TCP, UDP, ICMP and so on. If no one listens on a raw socket, only kernel supported protocols will be delivered, if someone listens on the raw socket, all valid IP packets will be delivered.

- IpOutRequests

Defined in [RFC1213 ipOutRequests](#)

The number of packets sent via IP layer, for both single cast and multicast packets, and would always be updated together with `IpExtOutOctets`.

- IpExtInOctets and IpExtOutOctets

They are Linux kernel extensions, no RFC definitions. Please note, RFC1213 indeed defines `ifInOctets` and `ifOutOctets`, but they are different things. The `ifInOctets` and `ifOutOctets` include the MAC layer header size but `IpExtInOctets` and `IpExtOutOctets` don't, they only include the IP layer header and the IP layer data.

- `IpExtInNoECTPkts`, `IpExtInECT1Pkts`, `IpExtInECT0Pkts`, `IpExtInCEPkts`

They indicate the number of four kinds of ECN IP packets, please refer [Explicit Congestion Notification](#) for more details.

These 4 counters calculate how many packets received per ECN status. They count the real frame number regardless the LRO/GRO. So for the same packet, you might find that IpInReceives count 1, but IpExtInNoECTPkts counts 2 or more.

- `IpInHdrErrors`

Defined in [RFC1213 ipInHdrErrors](#). It indicates the packet is dropped due to the IP header error. It might happen in both IP input and IP forward paths.

- `IpInAddrErrors`

Defined in [RFC1213 ipInAddrErrors](#). It will be increased in two scenarios: (1) The IP address is invalid. (2) The destination IP address is not a local address and IP forwarding is not enabled

- `IpExtInNoRoutes`

This counter means the packet is dropped when the IP stack receives a packet and can't find a route for it from the route table. It might happen when IP forwarding is enabled and the destination IP address is not a local address and there is no route for the destination IP address.

- `IpInUnknownProtos`

Defined in [RFC1213 ipInUnknownProtos](#). It will be increased if the layer 4 protocol is unsupported by kernel. If an application is using raw socket, kernel will always deliver the packet to the raw socket and this counter won't be increased.

- `IpExtInTruncatedPkts`

For IPv4 packet, it means the actual data size is smaller than the "Total Length" field in the IPv4 header.

- `IpInDiscards`

Defined in [RFC1213 ipInDiscards](#). It indicates the packet is dropped in the IP receiving path and due to kernel internal reasons (e.g. no enough memory).

- `IpOutDiscards`

Defined in [RFC1213 ipOutDiscards](#). It indicates the packet is dropped in the IP sending path and due to kernel internal reasons.

- `IpOutNoRoutes`

Defined in [RFC1213 ipOutNoRoutes](#). It indicates the packet is dropped in the IP sending path and no route is found for it.

23.2 ICMP counters

- `IcmpInMsgs` and `IcmpOutMsgs`

Defined by [RFC1213 icmpInMsgs](#) and [RFC1213 icmpOutMsgs](#)

As mentioned in the RFC1213, these two counters include errors, they would be increased even if the ICMP packet has an invalid type. The ICMP output path will check the header of a raw socket, so the `IcmpOutMsgs` would still be updated if the IP header is constructed by a userspace program.

- ICMP named types

These counters include most of common ICMP types, they are:

IcmpInDestUnreachs: [RFC1213 icmpInDestUnreachs](#)
 IcmpInTimeExcds: [RFC1213 icmpInTimeExcds](#)
 IcmpInParmProbs: [RFC1213 icmpInParmProbs](#)
 IcmpInSrcQuenches: [RFC1213 icmpInSrcQuenches](#)
 IcmpInRedirects: [RFC1213 icmpInRedirects](#)
 IcmpInEchos: [RFC1213 icmpInEchos](#)
 IcmpInEchoReps: [RFC1213 icmpInEchoReps](#)
 IcmpInTimestamps: [RFC1213 icmpInTimestamps](#)
 IcmpInTimestampReps: [RFC1213 icmpInTimestampReps](#)
 IcmpInAddrMasks: [RFC1213 icmpInAddrMasks](#)
 IcmpInAddrMaskReps: [RFC1213 icmpInAddrMaskReps](#)
 IcmpOutDestUnreachs: [RFC1213 icmpOutDestUnreachs](#)
 IcmpOutTimeExcds: [RFC1213 icmpOutTimeExcds](#)
 IcmpOutParmProbs: [RFC1213 icmpOutParmProbs](#)
 IcmpOutSrcQuenches: [RFC1213 icmpOutSrcQuenches](#)
 IcmpOutRedirects: [RFC1213 icmpOutRedirects](#)
 IcmpOutEchos: [RFC1213 icmpOutEchos](#)
 IcmpOutEchoReps: [RFC1213 icmpOutEchoReps](#)
 IcmpOutTimestamps: [RFC1213 icmpOutTimestamps](#)
 IcmpOutTimestampReps: [RFC1213 icmpOutTimestampReps](#)
 IcmpOutAddrMasks: [RFC1213 icmpOutAddrMasks](#)
 IcmpOutAddrMaskReps: [RFC1213 icmpOutAddrMaskReps](#)

Every ICMP type has two counters: 'In' and 'Out'. E.g., for the ICMP Echo packet, they are IcmpInEchos and IcmpOutEchos. Their meanings are straightforward. The 'In' counter means kernel receives such a packet and the 'Out' counter means kernel sends such a packet.

- ICMP numeric types

They are IcmpMsgInType[N] and IcmpMsgOutType[N], the [N] indicates the ICMP type number. These counters track all kinds of ICMP packets. The ICMP type number definition could be found in the [ICMP parameters](#) document.

For example, if the Linux kernel sends an ICMP Echo packet, the IcmpMsgOutType8 would increase 1. And if kernel gets an ICMP Echo Reply packet, IcmpMsgInType0 would increase 1.

- IcmpInCsumErrors

This counter indicates the checksum of the ICMP packet is wrong. Kernel verifies the checksum after updating the IcmpInMsgs and before updating IcmpMsgInType[N]. If a packet has bad checksum, the IcmpInMsgs would be updated but none of IcmpMsgInType[N] would be updated.

- IcmpInErrors and IcmpOutErrors

Defined by [RFC1213 icmpInErrors](#) and [RFC1213 icmpOutErrors](#)

When an error occurs in the ICMP packet handler path, these two counters would be updated. The receiving packet path use IcmpInErrors and the sending packet path use IcmpOutErrors. When IcmpInCsumErrors is increased, IcmpInErrors would always be increased too.

23.2.1 relationship of the ICMP counters

The sum of IcmpMsgOutType[N] is always equal to IcmpOutMsgs, as they are updated at the same time. The sum of IcmpMsgInType[N] plus IcmpInErrors should be equal or larger than IcmpInMsgs. When kernel receives an ICMP packet, kernel follows below logic:

1. increase IcmpInMsgs
2. if has any error, update IcmpInErrors and finish the process
3. update IcmpMsgOutType[N]
4. handle the packet depending on the type, if has any error, update IcmpInErrors and finish the process

So if all errors occur in step (2), IcmpInMsgs should be equal to the sum of IcmpMsgOutType[N] plus IcmpInErrors. If all errors occur in step (4), IcmpInMsgs should be equal to the sum of IcmpMsgOutType[N]. If the errors occur in both step (2) and step (4), IcmpInMsgs should be less than the sum of IcmpMsgOutType[N] plus IcmpInErrors.

23.3 General TCP counters

- TcpInSegs

Defined in [RFC1213](#) `tcpInSegs`

The number of packets received by the TCP layer. As mentioned in RFC1213, it includes the packets received in error, such as checksum error, invalid TCP header and so on. Only one error won't be included: if the layer 2 destination address is not the NIC's layer 2 address. It might happen if the packet is a multicast or broadcast packet, or the NIC is in promiscuous mode. In these situations, the packets would be delivered to the TCP layer, but the TCP layer will discard these packets before increasing TcpInSegs. The TcpInSegs counter isn't aware of GRO. So if two packets are merged by GRO, the TcpInSegs counter would only increase 1.

- TcpOutSegs

Defined in [RFC1213](#) `tcpOutSegs`

The number of packets sent by the TCP layer. As mentioned in RFC1213, it excludes the retransmitted packets. But it includes the SYN, ACK and RST packets. Doesn't like TcpInSegs, the TcpOutSegs is aware of GSO, so if a packet would be split to 2 by GSO, TcpOutSegs will increase 2.

- TcpActiveOpens

Defined in [RFC1213](#) `tcpActiveOpens`

It means the TCP layer sends a SYN, and come into the SYN-SENT state. Every time TcpActiveOpens increases 1, TcpOutSegs should always increase 1.

- TcpPassiveOpens

Defined in [RFC1213](#) `tcpPassiveOpens`

It means the TCP layer receives a SYN, replies a SYN+ACK, come into the SYN-RCVD state.

- TcpExtTCPRecvCoalesce

When packets are received by the TCP layer and are not be read by the application, the TCP layer will try to merge them. This counter indicate how many packets are merged in such situation. If GRO is enabled, lots of packets would be merged by GRO, these packets wouldn't be counted to TcpExtTCPRecvCoalesce.

- TcpExtTCPAutoCorking

When sending packets, the TCP layer will try to merge small packets to a bigger one. This counter increase 1 for every packet merged in such situation. Please refer to the LWN article for more details: <https://lwn.net/Articles/576263/>

- TcpExtTCPOrigDataSent

This counter is explained by kernel commit [f19c29e3e391](#), I pasted the explanation below:

TCPOrigDataSent: number of outgoing packets with original data (excluding retransmission but including data-in-SYN). This counter is different from TcpOutSegs because TcpOutSegs also tracks pure ACKs. TCPOrigDataSent is more useful to track the TCP retransmission rate.

- TCPSynRetrans

This counter is explained by kernel commit [f19c29e3e391](#), I pasted the explanation below:

TCPSynRetrans: number of SYN and SYN/ACK retransmits to break down retransmissions into SYN, fast-retransmits, timeout retransmits, etc.

- TCPFastOpenActiveFail

This counter is explained by kernel commit [f19c29e3e391](#), I pasted the explanation below:

TCPFastOpenActiveFail: Fast Open attempts (SYN/data) failed because the remote does not accept it or the attempts timed out.

- TcpExtListenOverflows and TcpExtListenDrops

When kernel receives a SYN from a client, and if the TCP accept queue is full, kernel will drop the SYN and add 1 to TcpExtListenOverflows. At the same time kernel will also add 1 to TcpExtListenDrops. When a TCP socket is in LISTEN state, and kernel need to drop a packet, kernel would always add 1 to TcpExtListenDrops. So increase TcpExtListenOverflows would let TcpExtListenDrops increasing at the same time, but TcpExtListenDrops would also increase without TcpExtListenOverflows increasing, e.g. a memory allocation fail would also let TcpExtListenDrops increase.

Note: The above explanation is based on kernel 4.10 or above version, on an old kernel, the TCP stack has different behavior when TCP accept queue is full. On the old kernel, TCP stack won't drop the SYN, it would complete the 3-way handshake. As the accept queue is full, TCP stack will keep the socket in the TCP half-open queue. As it is in the half open queue, TCP stack will send SYN+ACK on an exponential backoff timer, after client replies ACK, TCP stack checks whether the accept queue is still full, if it is not full, moves the socket to the accept queue, if it is full, keeps the socket in the half-open queue, at next time client replies ACK, this socket will get another chance to move to the accept queue.

23.4 TCP Fast Open

- TcpEstabResets

Defined in [RFC1213](#) `tcpEstabResets`.

- TcpAttemptFails

Defined in [RFC1213](#) `tcpAttemptFails`.

- TcpOutRsts

Defined in [RFC1213](#) `tcpOutRsts`. The RFC says this counter indicates the 'segments sent containing the RST flag', but in linux kernel, this counter indicates the segments kernel tried to send. The sending process might be failed due to some errors (e.g. memory alloc failed).

- TcpExtTCPSpuriousRtxHostQueues

When the TCP stack wants to retransmit a packet, and finds that packet is not lost in the network, but the packet is not sent yet, the TCP stack would give up the retransmission and update this counter. It might happen if a packet stays too long time in a qdisc or driver queue.

- TcpEstabResets

The socket receives a RST packet in Establish or CloseWait state.

- TcpExtTCPKeepAlive

This counter indicates many keepalive packets were sent. The keepalive won't be enabled by default. A userspace program could enable it by setting the `SO_KEEPALIVE` socket option.

- TcpExtTCPSpuriousRTOs

The spurious retransmission timeout detected by the F-RTO algorithm.

23.5 TCP Fast Path

When kernel receives a TCP packet, it has two paths to handle the packet, one is fast path, another is slow path. The comment in kernel code provides a good explanation of them, I pasted them below:

It is split into a fast path and a slow path. The fast path is disabled when:

- A zero window was announced from us
- zero window probing
is only handled properly on the slow path.
- Out of order segments arrived.
- Urgent data is expected.
- There is no buffer space left
- Unexpected TCP flags/window values/header lengths are received
(detected by checking the TCP header against `pred_flags`)
- Data is sent in both directions. The fast path only supports pure senders or pure receivers (this means either the sequence number or the ack value must stay constant)
- Unexpected TCP option.

Kernel will try to use fast path unless any of the above conditions are satisfied. If the packets are out of order, kernel will handle them in slow path, which means the performance might be not very good. Kernel would also come into slow path if the "Delayed ack" is used, because when using "Delayed ack", the data is sent in both directions. When the TCP window scale option is not used, kernel will try to enable fast path immediately when the connection comes into the established state, but if the TCP window scale option is used, kernel will disable the fast path at first, and try to enable it after kernel receives packets.

- TcpExtTCPPureAcks and TcpExtTCPHPAcks

If a packet set ACK flag and has no data, it is a pure ACK packet, if kernel handles it in the fast path, TcpExtTCPHPAcks will increase 1, if kernel handles it in the slow path, TcpExtTCP-PureAcks will increase 1.

- TcpExtTCPHPHits

If a TCP packet has data (which means it is not a pure ACK packet), and this packet is handled in the fast path, TcpExtTCPHPHits will increase 1.

23.6 TCP abort

- TcpExtTCPAbortOnData

It means TCP layer has data in flight, but need to close the connection. So TCP layer sends a RST to the other side, indicate the connection is not closed very graceful. An easy way to increase this counter is using the SO_LINGER option. Please refer to the SO_LINGER section of the [socket man page](#):

By default, when an application closes a connection, the close function will return immediately and kernel will try to send the in-flight data async. If you use the SO_LINGER option, set l_onoff to 1, and l_linger to a positive number, the close function won't return immediately, but wait for the in-flight data are acked by the other side, the max wait time is l_linger seconds. If set l_onoff to 1 and set l_linger to 0, when the application closes a connection, kernel will send a RST immediately and increase the TcpExtTCPAbortOnData counter.

- TcpExtTCPAbortOnClose

This counter means the application has unread data in the TCP layer when the application wants to close the TCP connection. In such a situation, kernel will send a RST to the other side of the TCP connection.

- TcpExtTCPAbortOnMemory

When an application closes a TCP connection, kernel still need to track the connection, let it complete the TCP disconnect process. E.g. an app calls the close method of a socket, kernel sends fin to the other side of the connection, then the app has no relationship with the socket any more, but kernel need to keep the socket, this socket becomes an orphan socket, kernel waits for the reply of the other side, and would come to the TIME_WAIT state finally. When kernel has no enough memory to keep the orphan socket, kernel would send an RST to the other side, and delete the socket, in such situation, kernel will increase 1 to the TcpExtTCPAbortOnMemory. Two conditions would trigger TcpExtTCPAbortOnMemory:

1. the memory used by the TCP protocol is higher than the third value of the tcp_mem. Please refer the [tcp_mem](#) section in the [TCP man page](#):
2. the orphan socket count is higher than net.ipv4.tcp_max_orphans

- `TcpExtTCPAbortOnTimeout`

This counter will increase when any of the TCP timers expire. In such situation, kernel won't send RST, just give up the connection.

- `TcpExtTCPAbortOnLinger`

When a TCP connection comes into `FIN_WAIT_2` state, instead of waiting for the fin packet from the other side, kernel could send a RST and delete the socket immediately. This is not the default behavior of Linux kernel TCP stack. By configuring the `TCP_LINGER2` socket option, you could let kernel follow this behavior.

- `TcpExtTCPAbortFailed`

The kernel TCP layer will send RST if the [RFC2525 2.17 section](#) is satisfied. If an internal error occurs during this process, `TcpExtTCPAbortFailed` will be increased.

23.7 TCP Hybrid Slow Start

The Hybrid Slow Start algorithm is an enhancement of the traditional TCP congestion window Slow Start algorithm. It uses two pieces of information to detect whether the max bandwidth of the TCP path is approached. The two pieces of information are ACK train length and increase in packet delay. For detail information, please refer the [Hybrid Slow Start paper](#). Either ACK train length or packet delay hits a specific threshold, the congestion control algorithm will come into the Congestion Avoidance state. Until v4.20, two congestion control algorithms are using Hybrid Slow Start, they are cubic (the default congestion control algorithm) and cdg. Four snmp counters relate with the Hybrid Slow Start algorithm.

- `TcpExtTCPHystartTrainDetect`

How many times the ACK train length threshold is detected

- `TcpExtTCPHystartTrainCwnd`

The sum of CWND detected by ACK train length. Dividing this value by `TcpExtTCPHystart-TrainDetect` is the average CWND which detected by the ACK train length.

- `TcpExtTCPHystartDelayDetect`

How many times the packet delay threshold is detected.

- `TcpExtTCPHystartDelayCwnd`

The sum of CWND detected by packet delay. Dividing this value by `TcpExtTCPHystartDelay-Detect` is the average CWND which detected by the packet delay.

23.8 TCP retransmission and congestion control

The TCP protocol has two retransmission mechanisms: SACK and fast recovery. They are exclusive with each other. When SACK is enabled, the kernel TCP stack would use SACK, or kernel would use fast recovery. The SACK is a TCP option, which is defined in [RFC2018](#), the fast recovery is defined in [RFC6582](#), which is also called 'Reno'.

The TCP congestion control is a big and complex topic. To understand the related snmp counter, we need to know the states of the congestion control state machine. There

are 5 states: Open, Disorder, CWR, Recovery and Loss. For details about these states, please refer page 5 and page 6 of this document: <https://pdfs.semanticscholar.org/0e9c/968d09ab2e53e24c4dca5b2d67c7f7140f8e.pdf>

- TcpExtTCPRenoRecovery and TcpExtTCPSackRecovery

When the congestion control comes into Recovery state, if sack is used, TcpExtTCPSackRecovery increases 1, if sack is not used, TcpExtTCPRenoRecovery increases 1. These two counters mean the TCP stack begins to retransmit the lost packets.

- TcpExtTCPSACKReneging

A packet was acknowledged by SACK, but the receiver has dropped this packet, so the sender needs to retransmit this packet. In this situation, the sender adds 1 to TcpExtTCPSACKReneging. A receiver could drop a packet which has been acknowledged by SACK, although it is unusual, it is allowed by the TCP protocol. The sender doesn't really know what happened on the receiver side. The sender just waits until the RTO expires for this packet, then the sender assumes this packet has been dropped by the receiver.

- TcpExtTCPRenoReorder

The reorder packet is detected by fast recovery. It would only be used if SACK is disabled. The fast recovery algorithm detects recorder by the duplicate ACK number. E.g., if retransmission is triggered, and the original retransmitted packet is not lost, it is just out of order, the receiver would acknowledge multiple times, one for the retransmitted packet, another for the arriving of the original out of order packet. Thus the sender would find more ACKs than its expectation, and the sender knows out of order occurs.

- TcpExtTCPTSReorder

The reorder packet is detected when a hole is filled. E.g., assume the sender sends packet 1,2,3,4,5, and the receiving order is 1,2,4,5,3. When the sender receives the ACK of packet 3 (which will fill the hole), two conditions will let TcpExtTCPTSReorder increase 1: (1) if the packet 3 is not re-transmitted yet. (2) if the packet 3 is retransmitted but the timestamp of the packet 3's ACK is earlier than the retransmission timestamp.

- TcpExtTCPSACKReorder

The reorder packet detected by SACK. The SACK has two methods to detect reorder: (1) DSACK is received by the sender. It means the sender sends the same packet more than one times. And the only reason is the sender believes an out of order packet is lost so it sends the packet again. (2) Assume packet 1,2,3,4,5 are sent by the sender, and the sender has received SACKs for packet 2 and 5, now the sender receives SACK for packet 4 and the sender doesn't retransmit the packet yet, the sender would know packet 4 is out of order. The TCP stack of kernel will increase TcpExtTCPSACKReorder for both of the above scenarios.

- TcpExtTCPSlowStartRetrans

The TCP stack wants to retransmit a packet and the congestion control state is 'Loss'.

- TcpExtTCPFastRetrans

The TCP stack wants to retransmit a packet and the congestion control state is not 'Loss'.

- TcpExtTCPLostRetransmit

A SACK points out that a retransmission packet is lost again.

- TcpExtTCPRetransFail

The TCP stack tries to deliver a retransmission packet to lower layers but the lower layers return an error.

- `TcpExtTCPSSynRetrans`

The TCP stack retransmits a SYN packet.

23.9 DSACK

The DSACK is defined in [RFC2883](#). The receiver uses DSACK to report duplicate packets to the sender. There are two kinds of duplications: (1) a packet which has been acknowledged is duplicate. (2) an out of order packet is duplicate. The TCP stack counts these two kinds of duplications on both receiver side and sender side.

- `TcpExtTCPDSACKOldSent`

The TCP stack receives a duplicate packet which has been acked, so it sends a DSACK to the sender.

- `TcpExtTCPDSACKOfoSent`

The TCP stack receives an out of order duplicate packet, so it sends a DSACK to the sender.

- `TcpExtTCPDSACKRecv`

The TCP stack receives a DSACK, which indicates an acknowledged duplicate packet is received.

- `TcpExtTCPDSACKOfoRecv`

The TCP stack receives a DSACK, which indicate an out of order duplicate packet is received.

23.10 invalid SACK and DSACK

When a SACK (or DSACK) block is invalid, a corresponding counter would be updated. The validation method is base on the start/end sequence number of the SACK block. For more details, please refer the comment of the function `tcp_is_sackblock_valid` in the kernel source code. A SACK option could have up to 4 blocks, they are checked individually. E.g., if 3 blocks of a SACK is invalid, the corresponding counter would be updated 3 times. The comment of [commit 18f02545a9a1 \("\[TCP\] MIB: Add counters for discarded SACK blocks"\)](#) has additional explanation:

- `TcpExtTCPSSACKDiscard`

This counter indicates how many SACK blocks are invalid. If the invalid SACK block is caused by ACK recording, the TCP stack will only ignore it and won't update this counter.

- `TcpExtTCPDSACKIgnoredOld` and `TcpExtTCPDSACKIgnoredNoUndo`

When a DSACK block is invalid, one of these two counters would be updated. Which counter will be updated depends on the `undo_marker` flag of the TCP socket. If the `undo_marker` is not set, the TCP stack isn't likely to re-transmit any packets, and we still receive an invalid DSACK block, the reason might be that the packet is duplicated in the middle of the network. In such scenario, `TcpExtTCPDSACKIgnoredNoUndo` will be updated. If the `undo_marker` is set, `TcpExtTCPDSACKIgnoredOld` will be updated. As implied in its name, it might be an old packet.

23.11 SACK shift

The linux networking stack stores data in sk_buff struct (skb for short). If a SACK block acrosses multiple skb, the TCP stack will try to re-arrange data in these skb. E.g. if a SACK block acknowledges seq 10 to 15, skb1 has seq 10 to 13, skb2 has seq 14 to 20. The seq 14 and 15 in skb2 would be moved to skb1. This operation is 'shift'. If a SACK block acknowledges seq 10 to 20, skb1 has seq 10 to 13, skb2 has seq 14 to 20. All data in skb2 will be moved to skb1, and skb2 will be discard, this operation is 'merge'.

- TcpExtTCPSSackShifted

A skb is shifted

- TcpExtTCPSSackMerged

A skb is merged

- TcpExtTCPSSackShiftFallback

A skb should be shifted or merged, but the TCP stack doesn't do it for some reasons.

23.12 TCP out of order

- TcpExtTCPOFOQueue

The TCP layer receives an out of order packet and has enough memory to queue it.

- TcpExtTCPOFODrop

The TCP layer receives an out of order packet but doesn't have enough memory, so drops it. Such packets won't be counted into TcpExtTCPOFOQueue.

- TcpExtTCPOFOMerge

The received out of order packet has an overlay with the previous packet. the overlay part will be dropped. All of TcpExtTCPOFOMerge packets will also be counted into TcpExtTCPOFOQueue.

23.13 TCP PAWS

PAWS (Protection Against Wrapped Sequence numbers) is an algorithm which is used to drop old packets. It depends on the TCP timestamps. For detail information, please refer the [timestamp](#) wiki and the [RFC of PAWS](#).

- TcpExtPAWSActive

Packets are dropped by PAWS in Syn-Sent status.

- TcpExtPAWSEstab

Packets are dropped by PAWS in any status other than Syn-Sent.

23.14 TCP ACK skip

In some scenarios, kernel would avoid sending duplicate ACKs too frequently. Please find more details in the `tcp_invalid_ratelimit` section of the [sysctl document](#). When kernel decides to skip an ACK due to `tcp_invalid_ratelimit`, kernel would update one of below counters to indicate the ACK is skipped in which scenario. The ACK would only be skipped if the received packet is either a SYN packet or it has no data.

- `TcpExtTCPACKSkippedSynRecv`

The ACK is skipped in Syn-Recv status. The Syn-Recv status means the TCP stack receives a SYN and replies SYN+ACK. Now the TCP stack is waiting for an ACK. Generally, the TCP stack doesn't need to send ACK in the Syn-Recv status. But in several scenarios, the TCP stack need to send an ACK. E.g., the TCP stack receives the same SYN packet repeatedly, the received packet does not pass the PAWS check, or the received packet sequence number is out of window. In these scenarios, the TCP stack needs to send ACK. If the ACK sending frequency is higher than `tcp_invalid_ratelimit` allows, the TCP stack will skip sending ACK and increase `TcpExtTCPACKSkippedSynRecv`.

- `TcpExtTCPACKSkippedPAWS`

The ACK is skipped due to PAWS (Protect Against Wrapped Sequence numbers) check fails. If the PAWS check fails in Syn-Recv, Fin-Wait-2 or Time-Wait statuses, the skipped ACK would be counted to `TcpExtTCPACKSkippedSynRecv`, `TcpExtTCPACKSkippedFinWait2` or `TcpExtTCPACKSkippedTimeWait`. In all other statuses, the skipped ACK would be counted to `TcpExtTCPACKSkippedPAWS`.

- `TcpExtTCPACKSkippedSeq`

The sequence number is out of window and the timestamp passes the PAWS check and the TCP status is not Syn-Recv, Fin-Wait-2, and Time-Wait.

- `TcpExtTCPACKSkippedFinWait2`

The ACK is skipped in Fin-Wait-2 status, the reason would be either PAWS check fails or the received sequence number is out of window.

- `TcpExtTCPACKSkippedTimeWait`

The ACK is skipped in Time-Wait status, the reason would be either PAWS check failed or the received sequence number is out of window.

- `TcpExtTCPACKSkippedChallenge`

The ACK is skipped if the ACK is a challenge ACK. The RFC 5961 defines 3 kind of challenge ACK, please refer [RFC 5961 section 3.2](#), [RFC 5961 section 4.2](#) and [RFC 5961 section 5.2](#). Besides these three scenarios, In some TCP status, the linux TCP stack would also send challenge ACKs if the ACK number is before the first unacknowledged number (more strict than [RFC 5961 section 5.2](#)).

23.15 TCP receive window

- `TcpExtTCPWantZeroWindowAdv`

Depending on current memory usage, the TCP stack tries to set receive window to zero. But the receive window might still be a no-zero value. For example, if the previous window size is 10, and the TCP stack receives 3 bytes, the current window size would be 7 even if the window size calculated by the memory usage is zero.

- `TcpExtTCPToZeroWindowAdv`

The TCP receive window is set to zero from a no-zero value.

- `TcpExtTCPFromZeroWindowAdv`

The TCP receive window is set to no-zero value from zero.

23.16 Delayed ACK

The TCP Delayed ACK is a technique which is used for reducing the packet count in the network. For more details, please refer the [Delayed ACK](#) wiki

- `TcpExtDelayedACKs`

A delayed ACK timer expires. The TCP stack will send a pure ACK packet and exit the delayed ACK mode.

- `TcpExtDelayedACKLocked`

A delayed ACK timer expires, but the TCP stack can't send an ACK immediately due to the socket is locked by a userspace program. The TCP stack will send a pure ACK later (after the userspace program unlock the socket). When the TCP stack sends the pure ACK later, the TCP stack will also update `TcpExtDelayedACKs` and exit the delayed ACK mode.

- `TcpExtDelayedACKLost`

It will be updated when the TCP stack receives a packet which has been ACKed. A Delayed ACK loss might cause this issue, but it would also be triggered by other reasons, such as a packet is duplicated in the network.

23.17 Tail Loss Probe (TLP)

TLP is an algorithm which is used to detect TCP packet loss. For more details, please refer the [TLP paper](#).

- `TcpExtTCPLossProbes`

A TLP probe packet is sent.

- `TcpExtTCPLossProbeRecovery`

A packet loss is detected and recovered by TLP.

23.18 TCP Fast Open description

TCP Fast Open is a technology which allows data transfer before the 3-way handshake complete. Please refer the [TCP Fast Open wiki](#) for a general description.

- `TcpExtTCPFastOpenActive`

When the TCP stack receives an ACK packet in the SYN-SENT status, and the ACK packet acknowledges the data in the SYN packet, the TCP stack understand the TFO cookie is accepted by the other side, then it updates this counter.

- `TcpExtTCPFastOpenActiveFail`

This counter indicates that the TCP stack initiated a TCP Fast Open, but it failed. This counter would be updated in three scenarios: (1) the other side doesn't acknowledge the data in the SYN packet. (2) The SYN packet which has the TFO cookie is timeout at least once. (3) after the 3-way handshake, the retransmission timeout happens `net.ipv4.tcp_retries1` times, because some middle-boxes may black-hole fast open after the handshake.

- `TcpExtTCPFastOpenPassive`

This counter indicates how many times the TCP stack accepts the fast open request.

- `TcpExtTCPFastOpenPassiveFail`

This counter indicates how many times the TCP stack rejects the fast open request. It is caused by either the TFO cookie is invalid or the TCP stack finds an error during the socket creating process.

- `TcpExtTCPFastOpenListenOverflow`

When the pending fast open request number is larger than `fastopenq->max_qlen`, the TCP stack will reject the fast open request and update this counter. When this counter is updated, the TCP stack won't update `TcpExtTCPFastOpenPassive` or `TcpExtTCPFastOpenPassiveFail`. The `fastopenq->max_qlen` is set by the `TCP_FASTOPEN` socket operation and it could not be larger than `net.core.somaxconn`. For example:

```
setsockopt(sfd, SOL_TCP, TCP_FASTOPEN, &qlen, sizeof(qlen));
```

- `TcpExtTCPFastOpenCookieReqd`

This counter indicates how many times a client wants to request a TFO cookie.

23.19 SYN cookies

SYN cookies are used to mitigate SYN flood, for details, please refer the [SYN cookies wiki](#).

- `TcpExtSyncookiesSent`

It indicates how many SYN cookies are sent.

- `TcpExtSyncookiesRecv`

How many reply packets of the SYN cookies the TCP stack receives.

- `TcpExtSyncookiesFailed`

The MSS decoded from the SYN cookie is invalid. When this counter is updated, the received packet won't be treated as a SYN cookie and the TcpExtSyncookiesRecv counter won't be updated.

23.20 Challenge ACK

For details of challenge ACK, please refer the explanation of TcpExtTCPACKSkippedChallenge.

- TcpExtTCPChallengeACK

The number of challenge acks sent.

- TcpExtTCPSYNChallenge

The number of challenge acks sent in response to SYN packets. After updates this counter, the TCP stack might send a challenge ACK and update the TcpExtTCPChallengeACK counter, or it might also skip to send the challenge and update the TcpExtTCPACKSkippedChallenge.

23.21 prune

When a socket is under memory pressure, the TCP stack will try to reclaim memory from the receiving queue and out of order queue. One of the reclaiming method is 'collapse', which means allocate a big skb, copy the contiguous skbs to the single big skb, and free these contiguous skbs.

- TcpExtPruneCalled

The TCP stack tries to reclaim memory for a socket. After updates this counter, the TCP stack will try to collapse the out of order queue and the receiving queue. If the memory is still not enough, the TCP stack will try to discard packets from the out of order queue (and update the TcpExtOfoPruned counter)

- TcpExtOfoPruned

The TCP stack tries to discard packet on the out of order queue.

- TcpExtRcvPruned

After 'collapse' and discard packets from the out of order queue, if the actually used memory is still larger than the max allowed memory, this counter will be updated. It means the 'prune' fails.

- TcpExtTCPRcvCollapsed

This counter indicates how many skbs are freed during 'collapse'.

23.22 examples

23.22.1 ping test

Run the ping command against the public dns server 8.8.8.8:

```
nstatuser@nstat-a:~$ ping 8.8.8.8 -c 1
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=119 time=17.8 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 17.875/17.875/17.875/0.000 ms
```

The nstat result:

```
nstatuser@nstat-a:~$ nstat
#kernel
IpInReceives          1          0.0
IpInDelivers           1          0.0
IpOutRequests          1          0.0
IcmpInMsgs              1          0.0
IcmpInEchoReps         1          0.0
IcmpOutMsgs             1          0.0
IcmpOutEchos            1          0.0
IcmpMsgInType0          1          0.0
IcmpMsgOutType8         1          0.0
IpExtInOctets           84          0.0
IpExtOutOctets          84          0.0
IpExtInNoECTPkts        1          0.0
```

The Linux server sent an ICMP Echo packet, so IpOutRequests, IcmpOutMsgs, IcmpOutEchos and IcmpMsgOutType8 were increased 1. The server got ICMP Echo Reply from 8.8.8.8, so IpInReceives, IcmpInMsgs, IcmpInEchoReps and IcmpMsgInType0 were increased 1. The ICMP Echo Reply was passed to the ICMP layer via IP layer, so IpInDelivers was increased 1. The default ping data size is 48, so an ICMP Echo packet and its corresponding Echo Reply packet are constructed by:

- 14 bytes MAC header
- 20 bytes IP header
- 16 bytes ICMP header
- 48 bytes data (default value of the ping command)

So the IpExtInOctets and IpExtOutOctets are $20+16+48=84$.

23.22.2 tcp 3-way handshake

On server side, we run:

```
nstatuser@nstat-b:~$ nc -lknv 0.0.0.0 9000
Listening on [0.0.0.0] (family 0, port 9000)
```

On client side, we run:

```
nstatuser@nstat-a:~$ nc -nv 192.168.122.251 9000
Connection to 192.168.122.251 9000 port [tcp/*] succeeded!
```

The server listened on tcp 9000 port, the client connected to it, they completed the 3-way handshake.

On server side, we can find below nstat output:

```
nstatuser@nstat-b:~$ nstat | grep -i tcp
TcpPassiveOpens          1              0.0
TcpInSegs                 2              0.0
TcpOutSegs                1              0.0
TcpExtTCPPureAcks         1              0.0
```

On client side, we can find below nstat output:

```
nstatuser@nstat-a:~$ nstat | grep -i tcp
TcpActiveOpens            1              0.0
TcpInSegs                 1              0.0
TcpOutSegs                2              0.0
```

When the server received the first SYN, it replied a SYN+ACK, and came into SYN-RCVD state, so TcpPassiveOpens increased 1. The server received SYN, sent SYN+ACK, received ACK, so server sent 1 packet, received 2 packets, TcpInSegs increased 2, TcpOutSegs increased 1. The last ACK of the 3-way handshake is a pure ACK without data, so TcpExtTCPPureAcks increased 1.

When the client sent SYN, the client came into the SYN-SENT state, so TcpActiveOpens increased 1, the client sent SYN, received SYN+ACK, sent ACK, so client sent 2 packets, received 1 packet, TcpInSegs increased 1, TcpOutSegs increased 2.

23.22.3 TCP normal traffic

Run nc on server:

```
nstatuser@nstat-b:~$ nc -lkv 0.0.0.0 9000
Listening on [0.0.0.0] (family 0, port 9000)
```

Run nc on client:

```
nstatuser@nstat-a:~$ nc -v nstat-b 9000
Connection to nstat-b 9000 port [tcp/*] succeeded!
```

Input a string in the nc client ('hello' in our example):

```
nstatuser@nstat-a:~$ nc -v nstat-b 9000
Connection to nstat-b 9000 port [tcp/*] succeeded!
hello
```

The client side nstat output:

```
nstatuser@nstat-a:~$ nstat
#kernel
IpInReceives          1          0.0
IpInDelivers          1          0.0
IpOutRequests          1          0.0
TcpInSegs              1          0.0
TcpOutSegs             1          0.0
TcpExtTCPPureAcks      1          0.0
TcpExtTCPOrigDataSent  1          0.0
IpExtInOctets          52         0.0
IpExtOutOctets         58         0.0
IpExtInNoECTPkts       1          0.0
```

The server side nstat output:

```
nstatuser@nstat-b:~$ nstat
#kernel
IpInReceives          1          0.0
IpInDelivers          1          0.0
IpOutRequests          1          0.0
TcpInSegs              1          0.0
TcpOutSegs             1          0.0
IpExtInOctets          58         0.0
IpExtOutOctets         52         0.0
IpExtInNoECTPkts       1          0.0
```

Input a string in nc client side again ('world' in our example):

```
nstatuser@nstat-a:~$ nc -v nstat-b 9000
Connection to nstat-b 9000 port [tcp/*] succeeded!
hello
world
```

Client side nstat output:

```
nstatuser@nstat-a:~$ nstat
#kernel
IpInReceives          1          0.0
IpInDelivers          1          0.0
IpOutRequests          1          0.0
TcpInSegs              1          0.0
TcpOutSegs             1          0.0
TcpExtTCPHPAcks        1          0.0
TcpExtTCPOrigDataSent  1          0.0
IpExtInOctets          52         0.0
```

IpExtOutOctets	58	0.0
IpExtInNoECTPkts	1	0.0

Server side nstat output:

nstatuser@nstat-b:~\$ nstat		
#kernel		
IpInReceives	1	0.0
IpInDelivers	1	0.0
IpOutRequests	1	0.0
TcpInSegs	1	0.0
TcpOutSegs	1	0.0
TcpExtTCPHPHits	1	0.0
IpExtInOctets	58	0.0
IpExtOutOctets	52	0.0
IpExtInNoECTPkts	1	0.0

Compare the first client-side nstat and the second client-side nstat, we could find one difference: the first one had a 'TcpExtTCPPureAcks', but the second one had a 'TcpExtTCPHPAcks'. The first server-side nstat and the second server-side nstat had a difference too: the second server-side nstat had a TcpExtTCPHPHits, but the first server-side nstat didn't have it. The network traffic patterns were exactly the same: the client sent a packet to the server, the server replied an ACK. But kernel handled them in different ways. When the TCP window scale option is not used, kernel will try to enable fast path immediately when the connection comes into the established state, but if the TCP window scale option is used, kernel will disable the fast path at first, and try to enable it after kernel receives packets. We could use the 'ss' command to verify whether the window scale option is used. e.g. run below command on either server or client:

nstatuser@nstat-a:~\$ ss -o state established -i '(dport = :9000 or sport = :9000)'					
Netid	Recv-Q	Send-Q	Local Address:Port	Peer	Address:Port
tcp	0	0	192.168.122.250:40654	192.168.122.	:251:9000
			ts sack cubic wscale:7,7 rto:204 rtt:0.98/0.49 mss:1448 pmtu:1500		
			rcvmss:536 advmss:1448 cwnd:10 bytes_acked:1 segs_out:2 segs_in:1 send 118.		
			2Mbps lastsnd:46572 lastrcv:46572 lastack:46572 pacing_rate 236.4Mbps rcv_		
			space:29200 rcv_ssthresh:29200 minrtt:0.98		

The 'wscale:7,7' means both server and client set the window scale option to 7. Now we could explain the nstat output in our test:

In the first nstat output of client side, the client sent a packet, server reply an ACK, when kernel handled this ACK, the fast path was not enabled, so the ACK was counted into 'TcpExtTCP-PureAcks'.

In the second nstat output of client side, the client sent a packet again, and received another ACK from the server, in this time, the fast path is enabled, and the ACK was qualified for fast path, so it was handled by the fast path, so this ACK was counted into TcpExtTCPHPAcks.

In the first nstat output of server side, fast path was not enabled, so there was no 'TcpExtTCPHPHits'.

In the second nstat output of server side, the fast path was enabled, and the packet received from client qualified for fast path, so it was counted into 'TcpExtTCPHPHits'.

23.22.4 TcpExtTCPAbortOnClose

On the server side, we run below python script:

```
import socket
import time

port = 9000

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('0.0.0.0', port))
s.listen(1)
sock, addr = s.accept()
while True:
    time.sleep(9999999)
```

This python script listen on 9000 port, but doesn't read anything from the connection.

On the client side, we send the string "hello" by nc:

```
nstatuser@nstat-a:~$ echo "hello" | nc nstat-b 9000
```

Then, we come back to the server side, the server has received the "hello" packet, and the TCP layer has acked this packet, but the application didn't read it yet. We type Ctrl-C to terminate the server script. Then we could find TcpExtTCPAbortOnClose increased 1 on the server side:

```
nstatuser@nstat-b:~$ nstat | grep -i abort
TcpExtTCPAbortOnClose          1                  0.0
```

If we run tcpdump on the server side, we could find the server sent a RST after we type Ctrl-C.

23.22.5 TcpExtTCPAbortOnMemory and TcpExtTCPAbortOnTimeout

Below is an example which let the orphan socket count be higher than net.ipv4.tcp_max_orphans. Change tcp_max_orphans to a smaller value on client:

```
sudo bash -c "echo 10 > /proc/sys/net/ipv4/tcp_max_orphans"
```

Client code (create 64 connection to server):

```
nstatuser@nstat-a:~$ cat client_orphan.py
import socket
import time

server = 'nstat-b' # server address
port = 9000

count = 64
```

```

connection_list = []

for i in range(64):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((server, port))
    connection_list.append(s)
    print("connection_count: %d" % len(connection_list))

while True:
    time.sleep(99999)

```

Server code (accept 64 connection from client):

```

nstatuser@nstat-b:~$ cat server_orphan.py
import socket
import time

port = 9000
count = 64

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('0.0.0.0', port))
s.listen(count)
connection_list = []
while True:
    sock, addr = s.accept()
    connection_list.append((sock, addr))
    print("connection_count: %d" % len(connection_list))

```

Run the python scripts on server and client.

On server:

```
python3 server_orphan.py
```

On client:

```
python3 client_orphan.py
```

Run iptables on server:

```
sudo iptables -A INPUT -i ens3 -p tcp --destination-port 9000 -j DROP
```

Type Ctrl-C on client, stop client_orphan.py.

Check TcpExtTCPAbortOnMemory on client:

```

nstatuser@nstat-a:~$ nstat | grep -i abort
TcpExtTCPAbortOnMemory          54                  0.0

```

Check orphaned socket count on client:

```
nstatuser@nstat-a:~$ ss -s
Total: 131 (kernel 0)
TCP: 14 (estab 1, closed 0, orphaned 10, synrecv 0, timewait 0/0), ports 0

Transport Total IP IPv6
* 0 - -
RAW 1 0 1
UDP 1 1 0
TCP 14 13 1
INET 16 14 2
FRAG 0 0 0
```

The explanation of the test: after run `server_orphan.py` and `client_orphan.py`, we set up 64 connections between server and client. Run the `iptables` command, the server will drop all packets from the client, type Ctrl-C on `client_orphan.py`, the system of the client would try to close these connections, and before they are closed gracefully, these connections became orphan sockets. As the `iptables` of the server blocked packets from the client, the server won't receive fin from the client, so all connection on clients would be stuck on `FIN_WAIT_1` stage, so they will keep as orphan sockets until timeout. We have echo 10 to `/proc/sys/net/ipv4/tcp_max_orphans`, so the client system would only keep 10 orphan sockets, for all other orphan sockets, the client system sent RST for them and delete them. We have 64 connections, so the '`ss -s`' command shows the system has 10 orphan sockets, and the value of `TcpExtTCPAbortOnMemory` was 54.

An additional explanation about orphan socket count: You could find the exactly orphan socket count by the '`ss -s`' command, but when kernel decide whither increases `TcpExtTCPAbortOnMemory` and sends RST, kernel doesn't always check the exactly orphan socket count. For increasing performance, kernel checks an approximate count firstly, if the approximate count is more than `tcp_max_orphans`, kernel checks the exact count again. So if the approximate count is less than `tcp_max_orphans`, but exactly count is more than `tcp_max_orphans`, you would find `TcpExtTCPAbortOnMemory` is not increased at all. If `tcp_max_orphans` is large enough, it won't occur, but if you decrease `tcp_max_orphans` to a small value like our test, you might find this issue. So in our test, the client set up 64 connections although the `tcp_max_orphans` is 10. If the client only set up 11 connections, we can't find the change of `TcpExtTCPAbortOnMemory`.

Continue the previous test, we wait for several minutes. Because of the `iptables` on the server blocked the traffic, the server wouldn't receive fin, and all the client's orphan sockets would timeout on the `FIN_WAIT_1` state finally. So we wait for a few minutes, we could find 10 timeout on the client:

```
nstatuser@nstat-a:~$ nstat | grep -i abort
TcpExtTCPAbortOnTimeout 10 0.0
```

23.22.6 TcpExtTCPAbortOnLinger

The server side code:

```
nstatuser@nstat-b:~$ cat server_linger.py
import socket
import time

port = 9000

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('0.0.0.0', port))
s.listen(1)
sock, addr = s.accept()
while True:
    time.sleep(9999999)
```

The client side code:

```
nstatuser@nstat-a:~$ cat client_linger.py
import socket
import struct

server = 'nstat-b' # server address
port = 9000

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_LINGER, struct.pack('ii', 1, 10))
s.setsockopt(socket.SOL_TCP, socket.TCP_LINGER2, struct.pack('i', -1))
s.connect((server, port))
s.close()
```

Run server_linger.py on server:

```
nstatuser@nstat-b:~$ python3 server_linger.py
```

Run client_linger.py on client:

```
nstatuser@nstat-a:~$ python3 client_linger.py
```

After run client_linger.py, check the output of nstat:

```
nstatuser@nstat-a:~$ nstat | grep -i abort
TcpExtTCPAbortOnLinger      1          0.0
```

23.22.7 TcpExtTCPRecvCoalesce

On the server, we run a program which listen on TCP port 9000, but doesn't read any data:

```
import socket
import time
port = 9000
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('0.0.0.0', port))
s.listen(1)
sock, addr = s.accept()
while True:
    time.sleep(9999999)
```

Save the above code as server_coalesce.py, and run:

```
python3 server_coalesce.py
```

On the client, save below code as client_coalesce.py:

```
import socket
server = 'nstat-b'
port = 9000
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((server, port))
```

Run:

```
nstatuser@nstat-a:~$ python3 -i client_coalesce.py
```

We use '-i' to come into the interactive mode, then a packet:

```
>>> s.send(b'foo')
3
```

Send a packet again:

```
>>> s.send(b'bar')
3
```

On the server, run nstat:

```
ubuntu@nstat-b:~$ nstat
#kernel
IpInReceives          2          0.0
IpInDelivers           2          0.0
IpOutRequests          2          0.0
TcpInSegs               2          0.0
TcpOutSegs              2          0.0
TcpExtTCPRecvCoalesce   1          0.0
IpExtInOctets           110        0.0
IpExtOutOctets          104        0.0
IpExtInNoECTPkts        2          0.0
```

The client sent two packets, server didn't read any data. When the second packet arrived at server, the first packet was still in the receiving queue. So the TCP layer merged the two packets, and we could find the `TcpExtTCPRecvCoalesce` increased 1.

23.22.8 TcpExtListenOverflows and TcpExtListenDrops

On server, run the nc command, listen on port 9000:

```
nstatuser@nstat-b:~$ nc -lkv 0.0.0.0 9000
Listening on [0.0.0.0] (family 0, port 9000)
```

On client, run 3 nc commands in different terminals:

```
nstatuser@nstat-a:~$ nc -v nstat-b 9000
Connection to nstat-b 9000 port [tcp/*] succeeded!
```

The nc command only accepts 1 connection, and the accept queue length is 1. On current linux implementation, set queue length to n means the actual queue length is n+1. Now we create 3 connections, 1 is accepted by nc, 2 in accepted queue, so the accept queue is full.

Before running the 4th nc, we clean the nstat history on the server:

```
nstatuser@nstat-b:~$ nstat -n
```

Run the 4th nc on the client:

```
nstatuser@nstat-a:~$ nc -v nstat-b 9000
```

If the nc server is running on kernel 4.10 or higher version, you won't see the "Connection to ... succeeded!" string, because kernel will drop the SYN if the accept queue is full. If the nc client is running on an old kernel, you would see that the connection is succeeded, because kernel would complete the 3 way handshake and keep the socket on half open queue. I did the test on kernel 4.15. Below is the nstat on the server:

```
nstatuser@nstat-b:~$ nstat
#kernel
IpInReceives          4          0.0
IpInDelivers           4          0.0
TcpInSegs               4          0.0
TcpExtListenOverflows   4          0.0
TcpExtListenDrops        4          0.0
IpExtInOctets            240         0.0
IpExtInNoECTPkts         4          0.0
```

Both `TcpExtListenOverflows` and `TcpExtListenDrops` were 4. If the time between the 4th nc and the nstat was longer, the value of `TcpExtListenOverflows` and `TcpExtListenDrops` would be larger, because the SYN of the 4th nc was dropped, the client was retrying.

23.22.9 IpInAddrErrors, IpExtInNoRoutes and IpOutNoRoutes

server A IP address: 192.168.122.250 server B IP address: 192.168.122.251 Prepare on server A, add a route to server B:

```
$ sudo ip route add 8.8.8.8/32 via 192.168.122.251
```

Prepare on server B, disable send_redirects for all interfaces:

```
$ sudo sysctl -w net.ipv4.conf.all.send_redirects=0
$ sudo sysctl -w net.ipv4.conf.ens3.send_redirects=0
$ sudo sysctl -w net.ipv4.conf.lo.send_redirects=0
$ sudo sysctl -w net.ipv4.conf.default.send_redirects=0
```

We want to let sever A send a packet to 8.8.8.8, and route the packet to server B. When server B receives such packet, it might send a ICMP Redirect message to server A, set send_redirects to 0 will disable this behavior.

First, generate InAddrErrors. On server B, we disable IP forwarding:

```
$ sudo sysctl -w net.ipv4.conf.all.forwarding=0
```

On server A, we send packets to 8.8.8.8:

```
$ nc -v 8.8.8.8 53
```

On server B, we check the output of nstat:

```
$ nstat
#kernel
IpInReceives          3          0.0
IpInAddrErrors         3          0.0
IpExtInOctets          180        0.0
IpExtInNoECTPkts       3          0.0
```

As we have let server A route 8.8.8.8 to server B, and we disabled IP forwarding on server B, Server A sent packets to server B, then server B dropped packets and increased IpInAddrErrors. As the nc command would re-send the SYN packet if it didn't receive a SYN+ACK, we could find multiple IpInAddrErrors.

Second, generate IpExtInNoRoutes. On server B, we enable IP forwarding:

```
$ sudo sysctl -w net.ipv4.conf.all.forwarding=1
```

Check the route table of server B and remove the default route:

```
$ ip route show
default via 192.168.122.1 dev ens3 proto static
192.168.122.0/24 dev ens3 proto kernel scope link src 192.168.122.251
$ sudo ip route delete default via 192.168.122.1 dev ens3 proto static
```

On server A, we contact 8.8.8.8 again:

```
$ nc -v 8.8.8.8 53
nc: connect to 8.8.8.8 port 53 (tcp) failed: Network is unreachable
```

On server B, run nstat:

```
$ nstat
#kernel
IpInReceives          1          0.0
IpOutRequests          1          0.0
IcmpOutMsgs            1          0.0
IcmpOutDestUnreachs   1          0.0
IcmpMsgOutType3        1          0.0
IpExtInNoRoutes        1          0.0
IpExtInOctets          60         0.0
IpExtOutOctets         88         0.0
IpExtInNoECTPkts       1          0.0
```

We enabled IP forwarding on server B, when server B received a packet which destination IP address is 8.8.8.8, server B will try to forward this packet. We have deleted the default route, there was no route for 8.8.8.8, so server B increase IpExtInNoRoutes and sent the "ICMP Destination Unreachable" message to server A.

Third, generate IpOutNoRoutes. Run ping command on server B:

```
$ ping -c 1 8.8.8.8
connect: Network is unreachable
```

Run nstat on server B:

```
$ nstat
#kernel
IpOutNoRoutes          1          0.0
```

We have deleted the default route on server B. Server B couldn't find a route for the 8.8.8.8 IP address, so server B increased IpOutNoRoutes.

23.22.10 TcpExtTCPACKSkippedSynRecv

In this test, we send 3 same SYN packets from client to server. The first SYN will let server create a socket, set it to Syn-Recv status, and reply a SYN/ACK. The second SYN will let server reply the SYN/ACK again, and record the reply time (the duplicate ACK reply time). The third SYN will let server check the previous duplicate ACK reply time, and decide to skip the duplicate ACK, then increase the TcpExtTCPACKSkippedSynRecv counter.

Run tcpdump to capture a SYN packet:

```
nstatuser@nstat-a:~$ sudo tcpdump -c 1 -w /tmp/syn.pcap port 9000
tcpdump: listening on ens3, link-type EN10MB (Ethernet), capture size 262144 bytes
```

Open another terminal, run nc command:

```
nstatuser@nstat-a:~$ nc nstat-b 9000
```

As the nstat-b didn't listen on port 9000, it should reply a RST, and the nc command exited immediately. It was enough for the tcpdump command to capture a SYN packet. A linux server might use hardware offload for the TCP checksum, so the checksum in the /tmp/syn.pcap might be not correct. We call tcprewrite to fix it:

```
nstatuser@nstat-a:~$ tcprewrite --infile=/tmp/syn.pcap --outfile=/tmp/syn_fixcsum.pcap --fixcsum
```

On nstat-b, we run nc to listen on port 9000:

```
nstatuser@nstat-b:~$ nc -lrv 9000  
Listening on [0.0.0.0] (family 0, port 9000)
```

On nstat-a, we blocked the packet from port 9000, or nstat-a would send RST to nstat-b:

```
nstatuser@nstat-a:~$ sudo iptables -A INPUT -p tcp --sport 9000 -j DROP
```

Send 3 SYN repeatedly to nstat-b:

```
nstatuser@nstat-a:~$ for i in {1..3}; do sudo tcpreplay -i ens3 /tmp/syn_fixcsum.pcap; done
```

Check snmp counter on nstat-b:

```
nstatuser@nstat-b:~$ nstat | grep -i skip  
TcpExtTCPACKSkippedSynRecv 1 0.0
```

As we expected, TcpExtTCPACKSkippedSynRecv is 1.

23.22.11 TcpExtTCPACKSkippedPAWS

To trigger PAWS, we could send an old SYN.

On nstat-b, let nc listen on port 9000:

```
nstatuser@nstat-b:~$ nc -lrv 9000  
Listening on [0.0.0.0] (family 0, port 9000)
```

On nstat-a, run tcpdump to capture a SYN:

```
nstatuser@nstat-a:~$ sudo tcpdump -w /tmp/paws_pre.pcap -c 1 port 9000  
tcpdump: listening on ens3, link-type EN10MB (Ethernet), capture size 262144 bytes
```

On nstat-a, run nc as a client to connect nstat-b:

```
nstatuser@nstat-a:~$ nc -v nstat-b 9000  
Connection to nstat-b 9000 port [tcp/*] succeeded!
```

Now the tcpdump has captured the SYN and exit. We should fix the checksum:

```
nstatuser@nstat-a:~$ tcprewrite --infile /tmp/paws_pre.pcap --outfile /tmp/
→paws.pcap --fixcsum
```

Send the SYN packet twice:

```
nstatuser@nstat-a:~$ for i in {1..2}; do sudo tcpreplay -i ens3 /tmp/paws.pcap;
→ done
```

On nstat-b, check the snmp counter:

```
nstatuser@nstat-b:~$ nstat | grep -i skip
TcpExtTCPACKSkippedPAWS           1                   0.0
```

We sent two SYN via tcpreplay, both of them would let PAWS check failed, the nstat-b replied an ACK for the first SYN, skipped the ACK for the second SYN, and updated TcpExtTCPACK-SkippedPAWS.

23.22.12 TcpExtTCPACKSkippedSeq

To trigger TcpExtTCPACKSkippedSeq, we send packets which have valid timestamp (to pass PAWS check) but the sequence number is out of window. The linux TCP stack would avoid to skip if the packet has data, so we need a pure ACK packet. To generate such a packet, we could create two sockets: one on port 9000, another on port 9001. Then we capture an ACK on port 9001, change the source/destination port numbers to match the port 9000 socket. Then we could trigger TcpExtTCPACKSkippedSeq via this packet.

On nstat-b, open two terminals, run two nc commands to listen on both port 9000 and port 9001:

```
nstatuser@nstat-b:~$ nc -lrv 9000
Listening on [0.0.0.0] (family 0, port 9000)

nstatuser@nstat-b:~$ nc -lrv 9001
Listening on [0.0.0.0] (family 0, port 9001)
```

On nstat-a, run two nc clients:

```
nstatuser@nstat-a:~$ nc -v nstat-b 9000
Connection to nstat-b 9000 port [tcp/*] succeeded!

nstatuser@nstat-a:~$ nc -v nstat-b 9001
Connection to nstat-b 9001 port [tcp/*] succeeded!
```

On nstat-a, run tcpdump to capture an ACK:

```
nstatuser@nstat-a:~$ sudo tcpdump -w /tmp/seq_pre.pcap -c 1 dst port 9001
tcpdump: listening on ens3, link-type EN10MB (Ethernet), capture size 262144
→ bytes
```

On nstat-b, send a packet via the port 9001 socket. E.g. we sent a string 'foo' in our example:

```
nstatuser@nstat-b:~$ nc -lkv 9001
Listening on [0.0.0.0] (family 0, port 9001)
Connection from nstat-a 42132 received!
foo
```

On nstat-a, the tcpdump should have captured the ACK. We should check the source port numbers of the two nc clients:

```
nstatuser@nstat-a:~$ ss -ta '( dport = :9000 || dport = :9001 )' | tee
State  Recv-Q  Send-Q          Local Address:Port          Peer Address:Port
ESTAB   0        0              192.168.122.250:50208    192.168.122.251:9000
ESTAB   0        0              192.168.122.250:42132    192.168.122.251:9001
```

Run tcprewrite, change port 9001 to port 9000, change port 42132 to port 50208:

```
nstatuser@nstat-a:~$ tcprewrite --infile /tmp/seq_pre.pcap --outfile /tmp/seq.
-pcap -r 9001:9000 -r 42132:50208 --fixcsum
```

Now the /tmp/seq.pcap is the packet we need. Send it to nstat-b:

```
nstatuser@nstat-a:~$ for i in {1..2}; do sudo tcpreplay -i ens3 /tmp/seq.pcap; done
```

Check TcpExtTCPACKSkippedSeq on nstat-b:

```
nstatuser@nstat-b:~$ nstat | grep -i skip
TcpExtTCPACKSkippedSeq           1                  0.0
```

CHECKSUM OFFLOADS

24.1 Introduction

This document describes a set of techniques in the Linux networking stack to take advantage of checksum offload capabilities of various NICs.

The following technologies are described:

- TX Checksum Offload
- LCO: Local Checksum Offload
- RCO: Remote Checksum Offload

Things that should be documented here but aren't yet:

- RX Checksum Offload
- CHECKSUM_UNNECESSARY conversion

24.2 TX Checksum Offload

The interface for offloading a transmit checksum to a device is explained in detail in comments near the top of include/linux/skbuff.h.

In brief, it allows to request the device fill in a single ones-complement checksum defined by the sk_buff fields skb->csum_start and skb->csum_offset. The device should compute the 16-bit ones-complement checksum (i.e. the 'IP-style' checksum) from csum_start to the end of the packet, and fill in the result at (csum_start + csum_offset).

Because csum_offset cannot be negative, this ensures that the previous value of the checksum field is included in the checksum computation, thus it can be used to supply any needed corrections to the checksum (such as the sum of the pseudo-header for UDP or TCP).

This interface only allows a single checksum to be offloaded. Where encapsulation is used, the packet may have multiple checksum fields in different header layers, and the rest will have to be handled by another mechanism such as LCO or RCO.

CRC32c can also be offloaded using this interface, by means of filling skb->csum_start and skb->csum_offset as described above, and setting skb->csum_not_inet: see skbuff.h comment (section 'D') for more details.

No offloading of the IP header checksum is performed; it is always done in software. This is OK because when we build the IP header, we obviously have it in cache, so summing it isn't expensive. It's also rather short.

The requirements for GSO are more complicated, because when segmenting an encapsulated packet both the inner and outer checksums may need to be edited or recomputed for each resulting segment. See the `skbuff.h` comment (section 'E') for more details.

A driver declares its offload capabilities in `netdev->hw_features`; see [Netdev features mess and how to get out from it alive](#) for more. Note that a device which only advertises `NETIF_F_IP[V6]_CSUM` must still obey the `csum_start` and `csum_offset` given in the SKB; if it tries to deduce these itself in hardware (as some NICs do) the driver should check that the values in the SKB match those which the hardware will deduce, and if not, fall back to checksumming in software instead (with `skb_csum_hwoffload_help()` or one of the `skb_checksum_help()` / `skb_crc32c_csum_help` functions, as mentioned in `include/linux/skbuff.h`).

The stack should, for the most part, assume that checksum offload is supported by the underlying device. The only place that should check is `validate_xmit_skb()`, and the functions it calls directly or indirectly. That function compares the offload features requested by the SKB (which may include other offloads besides TX Checksum Offload) and, if they are not supported or enabled on the device (determined by `netdev->features`), performs the corresponding offload in software. In the case of TX Checksum Offload, that means calling `skb_csum_hwoffload_help(skb, features)`.

24.3 LCO: Local Checksum Offload

LCO is a technique for efficiently computing the outer checksum of an encapsulated datagram when the inner checksum is due to be offloaded.

The ones-complement sum of a correctly checksummed TCP or UDP packet is equal to the complement of the sum of the pseudo header, because everything else gets 'cancelled out' by the checksum field. This is because the sum was complemented before being written to the checksum field.

More generally, this holds in any case where the 'IP-style' ones complement checksum is used, and thus any checksum that TX Checksum Offload supports.

That is, if we have set up TX Checksum Offload with a start/offset pair, we know that after the device has filled in that checksum, the ones complement sum from `csum_start` to the end of the packet will be equal to the complement of whatever value we put in the checksum field beforehand. This allows us to compute the outer checksum without looking at the payload: we simply stop summing when we get to `csum_start`, then add the complement of the 16-bit word at (`csum_start + csum_offset`).

Then, when the true inner checksum is filled in (either by hardware or by `skb_checksum_help()`), the outer checksum will become correct by virtue of the arithmetic.

LCO is performed by the stack when constructing an outer UDP header for an encapsulation such as VXLAN or GENEVE, in `udp_set_csum()`. Similarly for the IPv6 equivalents, in `udp6_set_csum()`.

It is also performed when constructing an IPv4 GRE header, in `net/ipv4/ip_gre.c:build_header()`. It is *not* currently performed when constructing an IPv6 GRE header; the GRE checksum is

computed over the whole packet in net/ipv6/ip6_gre.c:ip6gre_xmit2(), but it should be possible to use LCO here as IPv6 GRE still uses an IP-style checksum.

All of the LCO implementations use a helper function lco_csum(), in include/linux/skbuff.h.

LCO can safely be used for nested encapsulations; in this case, the outer encapsulation layer will sum over both its own header and the 'middle' header. This does mean that the 'middle' header will get summed multiple times, but there doesn't seem to be a way to avoid that without incurring bigger costs (e.g. in SKB bloat).

24.4 RCO: Remote Checksum Offload

RCO is a technique for eliding the inner checksum of an encapsulated datagram, allowing the outer checksum to be offloaded. It does, however, involve a change to the encapsulation protocols, which the receiver must also support. For this reason, it is disabled by default.

RCO is detailed in the following Internet-Drafts:

- <https://tools.ietf.org/html/draft-herbert-remotecsumoffload-00>
- <https://tools.ietf.org/html/draft-herbert-vxlan-rco-00>

In Linux, RCO is implemented individually in each encapsulation protocol, and most tunnel types have flags controlling its use. For instance, VXLAN has the flag VXLAN_F_REMCSUM_TX (per struct vxlan_rdst) to indicate that RCO should be used when transmitting to a given remote destination.

SEGMENTATION OFFLOADS

25.1 Introduction

This document describes a set of techniques in the Linux networking stack to take advantage of segmentation offload capabilities of various NICs.

The following technologies are described:

- TCP Segmentation Offload - TSO
- UDP Fragmentation Offload - UFO
- IPIP, SIT, GRE, and UDP Tunnel Offloads
- Generic Segmentation Offload - GSO
- Generic Receive Offload - GRO
- Partial Generic Segmentation Offload - GSO_PARTIAL
- SCTP acceleration with GSO - GSO_BY_FRAGS

25.2 TCP Segmentation Offload

TCP segmentation allows a device to segment a single frame into multiple frames with a data payload size specified in `skb_shinfo()->gso_size`. When TCP segmentation requested the bit for either `SKB_GSO_TCPV4` or `SKB_GSO_TCPV6` should be set in `skb_shinfo()->gso_type` and `skb_shinfo()->gso_size` should be set to a non-zero value.

TCP segmentation is dependent on support for the use of partial checksum offload. For this reason TSO is normally disabled if the Tx checksum offload for a given device is disabled.

In order to support TCP segmentation offload it is necessary to populate the network and transport header offsets of the skbuff so that the device drivers will be able determine the offsets of the IP or IPv6 header and the TCP header. In addition as `CHECKSUM_PARTIAL` is required `csum_start` should also point to the TCP header of the packet.

For IPv4 segmentation we support one of two types in terms of the IP ID. The default behavior is to increment the IP ID with every segment. If the GSO type `SKB_GSO_TCP_FIXEDID` is specified then we will not increment the IP ID and all segments will use the same IP ID. If a device has `NETIF_F_TSO_MANGLEID` set then the IP ID can be ignored when performing TSO and we will either increment the IP ID for all frames, or leave it at a static value based on driver preference.

25.3 UDP Fragmentation Offload

UDP fragmentation offload allows a device to fragment an oversized UDP datagram into multiple IPv4 fragments. Many of the requirements for UDP fragmentation offload are the same as TSO. However the IPv4 ID for fragments should not increment as a single IPv4 datagram is fragmented.

UFO is deprecated: modern kernels will no longer generate UFO skbs, but can still receive them from tuntap and similar devices. Offload of UDP-based tunnel protocols is still supported.

25.4 IPIP, SIT, GRE, UDP Tunnel, and Remote Checksum Offloads

In addition to the offloads described above it is possible for a frame to contain additional headers such as an outer tunnel. In order to account for such instances an additional set of segmentation offload types were introduced including SKB_GSO_IPXIP4, SKB_GSO_IPXIP6, SKB_GSO_GRE, and SKB_GSO_UDP_TUNNEL. These extra segmentation types are used to identify cases where there are more than just 1 set of headers. For example in the case of IPIP and SIT we should have the network and transport headers moved from the standard list of headers to "inner" header offsets.

Currently only two levels of headers are supported. The convention is to refer to the tunnel headers as the outer headers, while the encapsulated data is normally referred to as the inner headers. Below is the list of calls to access the given headers:

IPIP/SIT Tunnel:

	Outer	Inner
MAC	skb_mac_header	
Network	skb_network_header	skb_inner_network_header
Transport	skb_transport_header	

UDP/GRE Tunnel:

	Outer	Inner
MAC	skb_mac_header	skb_inner_mac_header
Network	skb_network_header	skb_inner_network_header
Transport	skb_transport_header	skb_inner_transport_header

In addition to the above tunnel types there are also SKB_GSO_GRE_CSUM and SKB_GSO_UDP_TUNNEL_CSUM. These two additional tunnel types reflect the fact that the outer header also requests to have a non-zero checksum included in the outer header.

Finally there is SKB_GSO_TUNNEL_REMCSUM which indicates that a given tunnel header has requested a remote checksum offload. In this case the inner headers will be left with a partial checksum and only the outer header checksum will be computed.

25.5 Generic Segmentation Offload

Generic segmentation offload is a pure software offload that is meant to deal with cases where device drivers cannot perform the offloads described above. What occurs in GSO is that a given skbuff will have its data broken out over multiple skbuffs that have been resized to match the MSS provided via `skb_shinfo()->gso_size`.

Before enabling any hardware segmentation offload a corresponding software offload is required in GSO. Otherwise it becomes possible for a frame to be re-routed between devices and end up being unable to be transmitted.

25.6 Generic Receive Offload

Generic receive offload is the complement to GSO. Ideally any frame assembled by GRO should be segmented to create an identical sequence of frames using GSO, and any sequence of frames segmented by GSO should be able to be reassembled back to the original by GRO. The only exception to this is IPv4 ID in the case that the DF bit is set for a given IP header. If the value of the IPv4 ID is not sequentially incrementing it will be altered so that it is when a frame assembled via GRO is segmented via GSO.

25.7 Partial Generic Segmentation Offload

Partial generic segmentation offload is a hybrid between TSO and GSO. What it effectively does is take advantage of certain traits of TCP and tunnels so that instead of having to rewrite the packet headers for each segment only the inner-most transport header and possibly the outer-most network header need to be updated. This allows devices that do not support tunnel offloads or tunnel offloads with checksum to still make use of segmentation.

With the partial offload what occurs is that all headers excluding the inner transport header are updated such that they will contain the correct values for if the header was simply duplicated. The one exception to this is the outer IPv4 ID field. It is up to the device drivers to guarantee that the IPv4 ID field is incremented in the case that a given header does not have the DF bit set.

25.8 SCTP acceleration with GSO

SCTP - despite the lack of hardware support - can still take advantage of GSO to pass one large packet through the network stack, rather than multiple small packets.

This requires a different approach to other offloads, as SCTP packets cannot be just segmented to (P)MTU. Rather, the chunks must be contained in IP segments, padding respected. So unlike regular GSO, SCTP can't just generate a big skb, set `gso_size` to the fragmentation point and deliver it to IP layer.

Instead, the SCTP protocol layer builds an skb with the segments correctly padded and stored as chained skbs, and `skb_segment()` splits based on those. To signal this, `gso_size` is set to the special value `GSO_BY_FRAGS`.

Therefore, any code in the core networking stack must be aware of the possibility that gso_size will be GSO_BY_FRAGS and handle that case appropriately.

There are some helpers to make this easier:

- `skb_is_gso(skb) && skb_is_gso_sctp(skb)` is the best way to see if an skb is an SCTP GSO skb.
- For size checks, the `skb_gso_validate_*_len` family of helpers correctly considers GSO_BY_FRAGS.
- For manipulating packets, `skb_increase_gso_size` and `skb_decrease_gso_size` will check for GSO_BY_FRAGS and WARN if asked to manipulate these skbs.

This also affects drivers with the `NETIF_F_FRAGLIST` & `NETIF_F_GSO_SCTP` bits set. Note also that `NETIF_F_GSO_SCTP` is included in `NETIF_F_GSO_SOFTWARE`.

SCALING IN THE LINUX NETWORKING STACK

26.1 Introduction

This document describes a set of complementary techniques in the Linux networking stack to increase parallelism and improve performance for multi-processor systems.

The following technologies are described:

- RSS: Receive Side Scaling
- RPS: Receive Packet Steering
- RFS: Receive Flow Steering
- Accelerated Receive Flow Steering
- XPS: Transmit Packet Steering

26.2 RSS: Receive Side Scaling

Contemporary NICs support multiple receive and transmit descriptor queues (multi-queue). On reception, a NIC can send different packets to different queues to distribute processing among CPUs. The NIC distributes packets by applying a filter to each packet that assigns it to one of a small number of logical flows. Packets for each flow are steered to a separate receive queue, which in turn can be processed by separate CPUs. This mechanism is generally known as "Receive-side Scaling" (RSS). The goal of RSS and the other scaling techniques is to increase performance uniformly. Multi-queue distribution can also be used for traffic prioritization, but that is not the focus of these techniques.

The filter used in RSS is typically a hash function over the network and/or transport layer headers-- for example, a 4-tuple hash over IP addresses and TCP ports of a packet. The most common hardware implementation of RSS uses a 128-entry indirection table where each entry stores a queue number. The receive queue for a packet is determined by masking out the low order seven bits of the computed hash for the packet (usually a Toeplitz hash), taking this number as a key into the indirection table and reading the corresponding value.

Some NICs support symmetric RSS hashing where, if the IP (source address, destination address) and TCP/UDP (source port, destination port) tuples are swapped, the computed hash is the same. This is beneficial in some applications that monitor TCP/IP flows (IDS, firewalls, ...etc) and need both directions of the flow to land on the same Rx queue (and CPU). The "Symmetric-XOR" is a type of RSS algorithms that achieves this hash symmetry by XORing the input source

and destination fields of the IP and/or L4 protocols. This, however, results in reduced input entropy and could potentially be exploited. Specifically, the algorithm XORs the input as follows:

```
# (SRC_IP ^ DST_IP, SRC_IP ^ DST_IP, SRC_PORT ^ DST_PORT, SRC_PORT ^ DST_PORT)
```

The result is then fed to the underlying RSS algorithm.

Some advanced NICs allow steering packets to queues based on programmable filters. For example, webserver bound TCP port 80 packets can be directed to their own receive queue. Such “n-tuple” filters can be configured from ethtool (--config-ntuple).

26.2.1 RSS Configuration

The driver for a multi-queue capable NIC typically provides a kernel module parameter for specifying the number of hardware queues to configure. In the bnx2x driver, for instance, this parameter is called num_queues. A typical RSS configuration would be to have one receive queue for each CPU if the device supports enough queues, or otherwise at least one for each memory domain, where a memory domain is a set of CPUs that share a particular memory level (L1, L2, NUMA node, etc.).

The indirection table of an RSS device, which resolves a queue by masked hash, is usually programmed by the driver at initialization. The default mapping is to distribute the queues evenly in the table, but the indirection table can be retrieved and modified at runtime using ethtool commands (--show-rxfh-indir and --set-rxfh-indir). Modifying the indirection table could be done to give different queues different relative weights.

RSS IRQ Configuration

Each receive queue has a separate IRQ associated with it. The NIC triggers this to notify a CPU when new packets arrive on the given queue. The signaling path for PCIe devices uses message signaled interrupts (MSI-X), that can route each interrupt to a particular CPU. The active mapping of queues to IRQs can be determined from /proc/interrupts. By default, an IRQ may be handled on any CPU. Because a non-negligible part of packet processing takes place in receive interrupt handling, it is advantageous to spread receive interrupts between CPUs. To manually adjust the IRQ affinity of each interrupt see Documentation/core-api/irq/irq-affinity.rst. Some systems will be running irqbalance, a daemon that dynamically optimizes IRQ assignments and as a result may override any manual settings.

Suggested Configuration

RSS should be enabled when latency is a concern or whenever receive interrupt processing forms a bottleneck. Spreading load between CPUs decreases queue length. For low latency networking, the optimal setting is to allocate as many queues as there are CPUs in the system (or the NIC maximum, if lower). The most efficient high-rate configuration is likely the one with the smallest number of receive queues where no receive queue overflows due to a saturated CPU, because in default mode with interrupt coalescing enabled, the aggregate number of interrupts (and thus work) grows with each additional queue.

Per-cpu load can be observed using the mpstat utility, but note that on processors with hyper-threading (HT), each hyperthread is represented as a separate CPU. For interrupt handling, HT

has shown no benefit in initial tests, so limit the number of queues to the number of CPU cores in the system.

Dedicated RSS contexts

Modern NICs support creating multiple co-existing RSS configurations which are selected based on explicit matching rules. This can be very useful when application wants to constrain the set of queues receiving traffic for e.g. a particular destination port or IP address. The example below shows how to direct all traffic to TCP port 22 to queues 0 and 1.

To create an additional RSS context use:

```
# ethtool -X eth0 hfunc toeplitz context new
New RSS context is 1
```

Kernel reports back the ID of the allocated context (the default, always present RSS context has ID of 0). The new context can be queried and modified using the same APIs as the default context:

```
# ethtool -x eth0 context 1
RX flow hash indirection table for eth0 with 13 RX ring(s):
 0:    0    1    2    3    4    5    6    7
 8:    8    9   10   11   12    0    1    2
[...]
# ethtool -X eth0 equal 2 context 1
# ethtool -x eth0 context 1
RX flow hash indirection table for eth0 with 13 RX ring(s):
 0:    0    1    0    1    0    1    0    1
 8:    0    1    0    1    0    1    0    1
[...]
```

To make use of the new context direct traffic to it using an n-tuple filter:

```
# ethtool -N eth0 flow-type tcp6 dst-port 22 context 1
Added rule with ID 1023
```

When done, remove the context and the rule:

```
# ethtool -N eth0 delete 1023
# ethtool -X eth0 context 1 delete
```

26.3 RPS: Receive Packet Steering

Receive Packet Steering (RPS) is logically a software implementation of RSS. Being in software, it is necessarily called later in the datapath. Whereas RSS selects the queue and hence CPU that will run the hardware interrupt handler, RPS selects the CPU to perform protocol processing above the interrupt handler. This is accomplished by placing the packet on the desired CPU's backlog queue and waking up the CPU for processing. RPS has some advantages over RSS:

- 1) it can be used with any NIC

- 2) software filters can easily be added to hash over new protocols
- 3) it does not increase hardware device interrupt rate (although it does introduce inter-processor interrupts (IPIs))

RPS is called during bottom half of the receive interrupt handler, when a driver sends a packet up the network stack with `netif_rx()` or `netif_receive_skb()`. These call the `get_rps_cpu()` function, which selects the queue that should process a packet.

The first step in determining the target CPU for RPS is to calculate a flow hash over the packet's addresses or ports (2-tuple or 4-tuple hash depending on the protocol). This serves as a consistent hash of the associated flow of the packet. The hash is either provided by hardware or will be computed in the stack. Capable hardware can pass the hash in the receive descriptor for the packet; this would usually be the same hash used for RSS (e.g. computed Toeplitz hash). The hash is saved in `skb->hash` and can be used elsewhere in the stack as a hash of the packet's flow.

Each receive hardware queue has an associated list of CPUs to which RPS may enqueue packets for processing. For each received packet, an index into the list is computed from the flow hash modulo the size of the list. The indexed CPU is the target for processing the packet, and the packet is queued to the tail of that CPU's backlog queue. At the end of the bottom half routine, IPIs are sent to any CPUs for which packets have been queued to their backlog queue. The IPI wakes backlog processing on the remote CPU, and any queued packets are then processed up the networking stack.

26.3.1 RPS Configuration

RPS requires a kernel compiled with the `CONFIG_RPS` kconfig symbol (on by default for SMP). Even when compiled in, RPS remains disabled until explicitly configured. The list of CPUs to which RPS may forward traffic can be configured for each receive queue using a sysfs file entry:

```
/sys/class/net/<dev>/queues/rx-<n>/rps_cpus
```

This file implements a bitmap of CPUs. RPS is disabled when it is zero (the default), in which case packets are processed on the interrupting CPU. Documentation/core-api/irq/irq-affinity.rst explains how CPUs are assigned to the bitmap.

Suggested Configuration

For a single queue device, a typical RPS configuration would be to set the `rps_cpus` to the CPUs in the same memory domain of the interrupting CPU. If NUMA locality is not an issue, this could also be all CPUs in the system. At high interrupt rate, it might be wise to exclude the interrupting CPU from the map since that already performs much work.

For a multi-queue system, if RSS is configured so that a hardware receive queue is mapped to each CPU, then RPS is probably redundant and unnecessary. If there are fewer hardware queues than CPUs, then RPS might be beneficial if the `rps_cpus` for each queue are the ones that share the same memory domain as the interrupting CPU for that queue.

26.3.2 RPS Flow Limit

RPS scales kernel receive processing across CPUs without introducing reordering. The trade-off to sending all packets from the same flow to the same CPU is CPU load imbalance if flows vary in packet rate. In the extreme case a single flow dominates traffic. Especially on common server workloads with many concurrent connections, such behavior indicates a problem such as a misconfiguration or spoofed source Denial of Service attack.

Flow Limit is an optional RPS feature that prioritizes small flows during CPU contention by dropping packets from large flows slightly ahead of those from small flows. It is active only when an RPS or RFS destination CPU approaches saturation. Once a CPU's input packet queue exceeds half the maximum queue length (as set by sysctl net.core.netdev_max_backlog), the kernel starts a per-flow packet count over the last 256 packets. If a flow exceeds a set ratio (by default, half) of these packets when a new packet arrives, then the new packet is dropped. Packets from other flows are still only dropped once the input packet queue reaches netdev_max_backlog. No packets are dropped when the input packet queue length is below the threshold, so flow limit does not sever connections outright: even large flows maintain connectivity.

Interface

Flow limit is compiled in by default (CONFIG_NET_FLOW_LIMIT), but not turned on. It is implemented for each CPU independently (to avoid lock and cache contention) and toggled per CPU by setting the relevant bit in sysctl net.core.flow_limit_cpu_bitmap. It exposes the same CPU bitmap interface as rps_cpus (see above) when called from procfs:

```
/proc/sys/net/core/flow_limit_cpu_bitmap
```

Per-flow rate is calculated by hashing each packet into a hashtable bucket and incrementing a per-bucket counter. The hash function is the same that selects a CPU in RPS, but as the number of buckets can be much larger than the number of CPUs, flow limit has finer-grained identification of large flows and fewer false positives. The default table has 4096 buckets. This value can be modified through sysctl:

```
net.core.flow_limit_table_len
```

The value is only consulted when a new table is allocated. Modifying it does not update active tables.

Suggested Configuration

Flow limit is useful on systems with many concurrent connections, where a single connection taking up 50% of a CPU indicates a problem. In such environments, enable the feature on all CPUs that handle network rx interrupts (as set in /proc/irq/N/smp_affinity).

The feature depends on the input packet queue length to exceed the flow limit threshold (50% + the flow history length (256). Setting net.core.netdev_max_backlog to either 1000 or 10000 performed well in experiments.

26.4 RFS: Receive Flow Steering

While RPS steers packets solely based on hash, and thus generally provides good load distribution, it does not take into account application locality. This is accomplished by Receive Flow Steering (RFS). The goal of RFS is to increase datacache hitrate by steering kernel processing of packets to the CPU where the application thread consuming the packet is running. RFS relies on the same RPS mechanisms to enqueue packets onto the backlog of another CPU and to wake up that CPU.

In RFS, packets are not forwarded directly by the value of their hash, but the hash is used as index into a flow lookup table. This table maps flows to the CPUs where those flows are being processed. The flow hash (see RPS section above) is used to calculate the index into this table. The CPU recorded in each entry is the one which last processed the flow. If an entry does not hold a valid CPU, then packets mapped to that entry are steered using plain RPS. Multiple table entries may point to the same CPU. Indeed, with many flows and few CPUs, it is very likely that a single application thread handles flows with many different flow hashes.

`rps_sock_flow_table` is a global flow table that contains the *desired* CPU for flows: the CPU that is currently processing the flow in userspace. Each table value is a CPU index that is updated during calls to `recvmsg` and `sendmsg` (specifically, `inet_recvmsg()`, `inet_sendmsg()` and `tcp_splice_read()`).

When the scheduler moves a thread to a new CPU while it has outstanding receive packets on the old CPU, packets may arrive out of order. To avoid this, RFS uses a second flow table to track outstanding packets for each flow: `rps_dev_flow_table` is a table specific to each hardware receive queue of each device. Each table value stores a CPU index and a counter. The CPU index represents the *current* CPU onto which packets for this flow are enqueued for further kernel processing. Ideally, kernel and userspace processing occur on the same CPU, and hence the CPU index in both tables is identical. This is likely false if the scheduler has recently migrated a userspace thread while the kernel still has packets enqueued for kernel processing on the old CPU.

The counter in `rps_dev_flow_table` values records the length of the current CPU's backlog when a packet in this flow was last enqueued. Each backlog queue has a head counter that is incremented on dequeue. A tail counter is computed as head counter + queue length. In other words, the counter in `rps_dev_flow[i]` records the last element in flow *i* that has been enqueued onto the currently designated CPU for flow *i* (of course, entry *i* is actually selected by hash and multiple flows may hash to the same entry *i*).

And now the trick for avoiding out of order packets: when selecting the CPU for packet processing (from `get_rps_cpu()`) the `rps_sock_flow` table and the `rps_dev_flow` table of the queue that the packet was received on are compared. If the desired CPU for the flow (found in the `rps_sock_flow` table) matches the current CPU (found in the `rps_dev_flow` table), the packet is enqueued onto that CPU's backlog. If they differ, the current CPU is updated to match the desired CPU if one of the following is true:

- The current CPU's queue head counter \geq the recorded tail counter value in `rps_dev_flow[i]`
- The current CPU is unset ($\geq \text{nr_cpu_ids}$)
- The current CPU is offline

After this check, the packet is sent to the (possibly updated) current CPU. These rules aim to ensure that a flow only moves to a new CPU when there are no packets outstanding on the old

CPU, as the outstanding packets could arrive later than those about to be processed on the new CPU.

26.4.1 RFS Configuration

RFS is only available if the kconfig symbol CONFIG_RPS is enabled (on by default for SMP). The functionality remains disabled until explicitly configured. The number of entries in the global flow table is set through:

```
/proc/sys/net/core/rps_sock_flow_entries
```

The number of entries in the per-queue flow table are set through:

```
/sys/class/net/<dev>/queues/rx-<n>/rps_flow_cnt
```

Suggested Configuration

Both of these need to be set before RFS is enabled for a receive queue. Values for both are rounded up to the nearest power of two. The suggested flow count depends on the expected number of active connections at any given time, which may be significantly less than the number of open connections. We have found that a value of 32768 for rps_sock_flow_entries works fairly well on a moderately loaded server.

For a single queue device, the rps_flow_cnt value for the single queue would normally be configured to the same value as rps_sock_flow_entries. For a multi-queue device, the rps_flow_cnt for each queue might be configured as rps_sock_flow_entries / N, where N is the number of queues. So for instance, if rps_sock_flow_entries is set to 32768 and there are 16 configured receive queues, rps_flow_cnt for each queue might be configured as 2048.

26.5 Accelerated RFS

Accelerated RFS is to RFS what RSS is to RPS: a hardware-accelerated load balancing mechanism that uses soft state to steer flows based on where the application thread consuming the packets of each flow is running. Accelerated RFS should perform better than RFS since packets are sent directly to a CPU local to the thread consuming the data. The target CPU will either be the same CPU where the application runs, or at least a CPU which is local to the application thread's CPU in the cache hierarchy.

To enable accelerated RFS, the networking stack calls the ndo_rx_flow_steer driver function to communicate the desired hardware queue for packets matching a particular flow. The network stack automatically calls this function every time a flow entry in rps_dev_flow_table is updated. The driver in turn uses a device specific method to program the NIC to steer the packets.

The hardware queue for a flow is derived from the CPU recorded in rps_dev_flow_table. The stack consults a CPU to hardware queue map which is maintained by the NIC driver. This is an auto-generated reverse map of the IRQ affinity table shown by /proc/interrupts. Drivers can use functions in the cpu_rmap ("CPU affinity reverse map") kernel library to populate the map. For each CPU, the corresponding queue in the map is set to be one whose processing CPU is closest in cache locality.

26.5.1 Accelerated RFS Configuration

Accelerated RFS is only available if the kernel is compiled with CONFIG_RFS_ACCEL and support is provided by the NIC device and driver. It also requires that ntuple filtering is enabled via ethtool. The map of CPU to queues is automatically deduced from the IRQ affinities configured for each receive queue by the driver, so no additional configuration should be necessary.

Suggested Configuration

This technique should be enabled whenever one wants to use RFS and the NIC supports hardware acceleration.

26.6 XPS: Transmit Packet Steering

Transmit Packet Steering is a mechanism for intelligently selecting which transmit queue to use when transmitting a packet on a multi-queue device. This can be accomplished by recording two kinds of maps, either a mapping of CPU to hardware queue(s) or a mapping of receive queue(s) to hardware transmit queue(s).

1. XPS using CPUs map

The goal of this mapping is usually to assign queues exclusively to a subset of CPUs, where the transmit completions for these queues are processed on a CPU within this set. This choice provides two benefits. First, contention on the device queue lock is significantly reduced since fewer CPUs contend for the same queue (contention can be eliminated completely if each CPU has its own transmit queue). Secondly, cache miss rate on transmit completion is reduced, in particular for data cache lines that hold the `sk_buff` structures.

2. XPS using receive queues map

This mapping is used to pick transmit queue based on the receive queue(s) map configuration set by the administrator. A set of receive queues can be mapped to a set of transmit queues (many:many), although the common use case is a 1:1 mapping. This will enable sending packets on the same queue associations for transmit and receive. This is useful for busy polling multi-threaded workloads where there are challenges in associating a given CPU to a given application thread. The application threads are not pinned to CPUs and each thread handles packets received on a single queue. The receive queue number is cached in the socket for the connection. In this model, sending the packets on the same transmit queue corresponding to the associated receive queue has benefits in keeping the CPU overhead low. Transmit completion work is locked into the same queue-association that a given application is polling on. This avoids the overhead of triggering an interrupt on another CPU. When the application cleans up the packets during the busy poll, transmit completion may be processed along with it in the same thread context and so result in reduced latency.

XPS is configured per transmit queue by setting a bitmap of CPUs/receive-queues that may use that queue to transmit. The reverse mapping, from CPUs to transmit queues or from receive-queues to transmit queues, is computed and maintained for each network device. When transmitting the first packet in a flow, the function `get_xps_queue()` is called to select a queue. This function uses the ID of the receive queue for the socket connection for a match in the receive queue-to-transmit queue lookup table. Alternatively, this function can also use the ID of the running CPU as a key into the CPU-to-queue lookup table. If the ID matches a single queue, that is used for transmission. If multiple queues match, one is selected by using the flow hash

to compute an index into the set. When selecting the transmit queue based on receive queue(s) map, the transmit device is not validated against the receive device as it requires expensive lookup operation in the datapath.

The queue chosen for transmitting a particular flow is saved in the corresponding socket structure for the flow (e.g. a TCP connection). This transmit queue is used for subsequent packets sent on the flow to prevent out of order (ooo) packets. The choice also amortizes the cost of calling `get_xps_queues()` over all packets in the flow. To avoid ooo packets, the queue for a flow can subsequently only be changed if `skb->ooo_okay` is set for a packet in the flow. This flag indicates that there are no outstanding packets in the flow, so the transmit queue can change without the risk of generating out of order packets. The transport layer is responsible for setting `ooo_okay` appropriately. TCP, for instance, sets the flag when all data for a connection has been acknowledged.

26.6.1 XPS Configuration

XPS is only available if the kconfig symbol `CONFIG_XPS` is enabled (on by default for SMP). If compiled in, it is driver dependent whether, and how, XPS is configured at device init. The mapping of CPUs/receive-queues to transmit queue can be inspected and configured using sysfs:

For selection based on CPUs map:

```
/sys/class/net/<dev>/queues/tx-<n>/xps_cpus
```

For selection based on receive-queues map:

```
/sys/class/net/<dev>/queues/tx-<n>/xps_rxqs
```

Suggested Configuration

For a network device with a single transmission queue, XPS configuration has no effect, since there is no choice in this case. In a multi-queue system, XPS is preferably configured so that each CPU maps onto one queue. If there are as many queues as there are CPUs in the system, then each queue can also map onto one CPU, resulting in exclusive pairings that experience no contention. If there are fewer queues than CPUs, then the best CPUs to share a given queue are probably those that share the cache with the CPU that processes transmit completions for that queue (transmit interrupts).

For transmit queue selection based on receive queue(s), XPS has to be explicitly configured mapping receive-queue(s) to transmit queue(s). If the user configuration for receive-queue map does not apply, then the transmit queue is selected based on the CPUs map.

26.7 Per TX Queue rate limitation

These are rate-limitation mechanisms implemented by HW, where currently a max-rate attribute is supported, by setting a Mbps value to:

```
/sys/class/net/<dev>/queues/tx-<n>/tx_maxrate
```

A value of zero means disabled, and this is the default.

26.8 Further Information

RPS and RFS were introduced in kernel 2.6.35. XPS was incorporated into 2.6.38. Original patches were submitted by Tom Herbert (therbert@google.com)

Accelerated RFS was introduced in 2.6.35. Original patches were submitted by Ben Hutchings (bwh@kernel.org)

Authors:

- Tom Herbert (therbert@google.com)
- Willem de Bruijn (willem@willemb.org)

KERNEL TLS

27.1 Overview

Transport Layer Security (TLS) is a Upper Layer Protocol (ULP) that runs over TCP. TLS provides end-to-end data integrity and confidentiality.

27.2 User interface

27.2.1 Creating a TLS connection

First create a new TCP socket and set the TLS ULP.

```
sock = socket(AF_INET, SOCK_STREAM, 0);
setsockopt(sock, SOL_TCP, TCP_ULP, "tls", sizeof("tls"));
```

Setting the TLS ULP allows us to set/get TLS socket options. Currently only the symmetric encryption is handled in the kernel. After the TLS handshake is complete, we have all the parameters required to move the data-path to the kernel. There is a separate socket option for moving the transmit and the receive into the kernel.

```
/* From linux/tls.h */
struct tls_crypto_info {
    unsigned short version;
    unsigned short cipher_type;
};

struct tls12_crypto_info_aes_gcm_128 {
    struct tls_crypto_info info;
    unsigned char iv[TLS_CIPHER_AES_GCM_128_IV_SIZE];
    unsigned char key[TLS_CIPHER_AES_GCM_128_KEY_SIZE];
    unsigned char salt[TLS_CIPHER_AES_GCM_128_SALT_SIZE];
    unsigned char rec_seq[TLS_CIPHER_AES_GCM_128_REC_SEQ_SIZE];
};

struct tls12_crypto_info_aes_gcm_128 crypto_info;
crypto_info.info.version = TLS_1_2_VERSION;
```

```

crypto_info.info.cipher_type = TLS_CIPHER_AES_GCM_128;
memcpy(crypto_info.iv, iv_write, TLS_CIPHER_AES_GCM_128_IV_SIZE);
memcpy(crypto_info.rec_seq, seq_number_write,
       TLS_CIPHER_AES_GCM_128_REC_SEQ_SIZE);
memcpy(crypto_info.key, cipher_key_write, TLS_CIPHER_AES_GCM_128_KEY_SIZE);
memcpy(crypto_info.salt, implicit_iv_write, TLS_CIPHER_AES_GCM_128_SALT_SIZE);

setsockopt(sock, SOL_TLS, TLS_TX, &crypto_info, sizeof(crypto_info));

```

Transmit and receive are set separately, but the setup is the same, using either TLS_TX or TLS_RX.

27.2.2 Sending TLS application data

After setting the TLS_TX socket option all application data sent over this socket is encrypted using TLS and the parameters provided in the socket option. For example, we can send an encrypted hello world record as follows:

```

const char *msg = "hello world\n";
send(sock, msg, strlen(msg));

```

send() data is directly encrypted from the userspace buffer provided to the encrypted kernel send buffer if possible.

The sendfile system call will send the file's data over TLS records of maximum length (2^{14}).

```

file = open(filename, O_RDONLY);
fstat(file, &stat);
sendfile(sock, file, &offset, stat.st_size);

```

TLS records are created and sent after each send() call, unless MSG_MORE is passed. MSG_MORE will delay creation of a record until MSG_MORE is not passed, or the maximum record size is reached.

The kernel will need to allocate a buffer for the encrypted data. This buffer is allocated at the time send() is called, such that either the entire send() call will return -ENOMEM (or block waiting for memory), or the encryption will always succeed. If send() returns -ENOMEM and some data was left on the socket buffer from a previous call using MSG_MORE, the MSG_MORE data is left on the socket buffer.

27.2.3 Receiving TLS application data

After setting the TLS_RX socket option, all recv family socket calls are decrypted using TLS parameters provided. A full TLS record must be received before decryption can happen.

```

char buffer[16384];
recv(sock, buffer, 16384);

```

Received data is decrypted directly in to the user buffer if it is large enough, and no additional allocations occur. If the userspace buffer is too small, data is decrypted in the kernel and copied to userspace.

`EINVAL` is returned if the TLS version in the received message does not match the version passed in `setsockopt`.

`EMSGSIZE` is returned if the received message is too big.

`EBADMSG` is returned if decryption failed for any other reason.

27.2.4 Send TLS control messages

Other than application data, TLS has control messages such as alert messages (record type 21) and handshake messages (record type 22), etc. These messages can be sent over the socket by providing the TLS record type via a CMSG. For example the following function sends @data of @length bytes using a record of type @record_type.

```
/* send TLS control message using record_type */
static int klts_send_ctrl_message(int sock, unsigned char record_type,
                                  void *data, size_t length)
{
    struct msghdr msg = {0};
    int cmsg_len = sizeof(record_type);
    struct cmsghdr *cmsg;
    char buf[CMSG_SPACE(cmsg_len)];
    struct iovec msg iov; /* Vector of data to send/receive into. */

    msg.msg_control = buf;
    msg.msg_controllen = sizeof(buf);
    cmsg = CMSG_FIRSTHDR(&msg);
    cmsg->cmsg_level = SOL_TLS;
    cmsg->cmsg_type = TLS_SET_RECORD_TYPE;
    cmsg->cmsg_len = CMSG_LEN(cmsg_len);
    *CMSG_DATA(cmsg) = record_type;
    msg.msg_controllen = cmsg->cmsg_len;

    msg iov.iov_base = data;
    msg iov.iov_len = length;
    msg .msg iov = &msg iov;
    msg .msg iovlen = 1;

    return sendmsg(sock, &msg, 0);
}
```

Control message data should be provided unencrypted, and will be encrypted by the kernel.

27.2.5 Receiving TLS control messages

TLS control messages are passed in the userspace buffer, with message type passed via cmsg. If no cmsg buffer is provided, an error is returned if a control message is received. Data messages may be received without a cmsg buffer set.

```

char buffer[16384];
char cmsg[CMSG_SPACE(sizeof(unsigned char))];
struct msghdr msg = {0};
msg.msg_control = cmsg;
msg.msg_controllen = sizeof(cmsg);

struct iovec msg iov;
msg iov .iov_base = buffer;
msg iov .iov_len = 16384;

msg.msg iov = &msg iov;
msg.msg iovlen = 1;

int ret = recvmsg(sock, &msg, 0 /* flags */);

struct cmsghdr *cmsg = CMSG_FIRSTHDR(&msg);
if (cmsg->cmsg_level == SOL_TLS &&
    cmsg->cmsg_type == TLS_GET_RECORD_TYPE) {
    int record_type = *((unsigned char *)CMSG_DATA(cmsg));
    // Do something with record_type, and control message data in
    // buffer.
    //
    // Note that record_type may be == to application data (23).
} else {
    // Buffer contains application data.
}

```

recv will never return data from mixed types of TLS records.

27.2.6 Integrating in to userspace TLS library

At a high level, the kernel TLS ULP is a replacement for the record layer of a userspace TLS library.

A patchset to OpenSSL to use ktls as the record layer is [here](#).

An example of calling send directly after a handshake using gnutls. Since it doesn't implement a full record layer, control messages are not supported.

27.2.7 Optional optimizations

There are certain condition-specific optimizations the TLS ULP can make, if requested. Those optimizations are either not universally beneficial or may impact correctness, hence they require an opt-in. All options are set per-socket using `setsockopt()`, and their state can be checked using `getsockopt()` and via socket diag (`ss`).

TLS_TX_ZEROCOPY_RO

For device offload only. Allow `sendfile()` data to be transmitted directly to the NIC without making an in-kernel copy. This allows true zero-copy behavior when device offload is enabled.

The application must make sure that the data is not modified between being submitted and transmission completing. In other words this is mostly applicable if the data sent on a socket via `sendfile()` is read-only.

Modifying the data may result in different versions of the data being used for the original TCP transmission and TCP retransmissions. To the receiver this will look like TLS records had been tampered with and will result in record authentication failures.

TLS_RX_EXPECT_NO_PAD

TLS 1.3 only. Expect the sender to not pad records. This allows the data to be decrypted directly into user space buffers with TLS 1.3.

This optimization is safe to enable only if the remote end is trusted, otherwise it is an attack vector to doubling the TLS processing cost.

If the record decrypted turns out to had been padded or is not a data record it will be decrypted again into a kernel buffer without zero copy. Such events are counted in the `TlsDecryptRetry` statistic.

27.3 Statistics

TLS implementation exposes the following per-namespace statistics (`/proc/net/tls_stat`):

- `TlsCurrTxSw`, `TlsCurrRxSw` - number of TX and RX sessions currently installed where host handles cryptography
- `TlsCurrTxDevice`, `TlsCurrRxDevice` - number of TX and RX sessions currently installed where NIC handles cryptography
- `TlsTxSw`, `TlsRxSw` - number of TX and RX sessions opened with host cryptography
- `TlsTxDevice`, `TlsRxDevice` - number of TX and RX sessions opened with NIC cryptography
- `TlsDecryptError` - record decryption failed (e.g. due to incorrect authentication tag)
- `TlsDeviceRxResync` - number of RX resyncs sent to NICs handling cryptography
- `TlsDecryptRetry` - number of RX records which had to be re-decrypted due to `TLS_RX_EXPECT_NO_PAD` mis-prediction. Note that this counter will also increment for non-data records.

- `TlsRxNoPadViolation` - number of data RX records which had to be re-decrypted due to `TLS_RX_EXPECT_NO_PAD` mis-prediction.

KERNEL TLS OFFLOAD

28.1 Kernel TLS operation

Linux kernel provides TLS connection offload infrastructure. Once a TCP connection is in ESTABLISHED state user space can enable the TLS Upper Layer Protocol (ULP) and install the cryptographic connection state. For details regarding the user-facing interface refer to the TLS documentation in [Documentation/networking/tls.rst](#).

`ktls` can operate in three modes:

- Software crypto mode (TLS_SW) - CPU handles the cryptography. In most basic cases only crypto operations synchronous with the CPU can be used, but depending on calling context CPU may utilize asynchronous crypto accelerators. The use of accelerators introduces extra latency on socket reads (decryption only starts when a read syscall is made) and additional I/O load on the system.
- Packet-based NIC offload mode (TLS_HW) - the NIC handles crypto on a packet by packet basis, provided the packets arrive in order. This mode integrates best with the kernel stack and is described in detail in the remaining part of this document (`ethtool` flags `tls-hw-tx-offload` and `tls-hw-rx-offload`).
- Full TCP NIC offload mode (TLS_HW_RECORD) - mode of operation where NIC driver and firmware replace the kernel networking stack with its own TCP handling, it is not usable in production environments making use of the Linux networking stack for example any firewalling abilities or QoS and packet scheduling (`ethtool` flag `tls-hw-record`).

The operation mode is selected automatically based on device configuration, offload opt-in or opt-out on per-connection basis is not currently supported.

28.1.1 TX

At a high level user write requests are turned into a scatter list, the TLS ULP intercepts them, inserts record framing, performs encryption (in TLS_SW mode) and then hands the modified scatter list to the TCP layer. From this point on the TCP stack proceeds as normal.

In TLS_HW mode the encryption is not performed in the TLS ULP. Instead packets reach a device driver, the driver will mark the packets for crypto offload based on the socket the packet is attached to, and send them to the device for encryption and transmission.

28.1.2 RX

On the receive side if the device handled decryption and authentication successfully, the driver will set the decrypted bit in the associated *struct sk_buff*. The packets reach the TCP stack and are handled normally. ktls is informed when data is queued to the socket and the strparser mechanism is used to delineate the records. Upon read request, records are retrieved from the socket and passed to decryption routine. If device decrypted all the segments of the record the decryption is skipped, otherwise software path handles decryption.

Fig. 1: Layers of Kernel TLS stack

28.2 Device configuration

During driver initialization device sets the `NETIF_F_HW_TLS_RX` and `NETIF_F_HW_TLS_TX` features and installs its `struct tlsdev_ops` pointer in the `tlsdev_ops` member of the `struct net_device`.

When TLS cryptographic connection state is installed on a `ktls` socket (note that it is done twice, once for RX and once for TX direction, and the two are completely independent), the kernel checks if the underlying network device is offload-capable and attempts the offload. In case offload fails the connection is handled entirely in software using the same mechanism as if the offload was never tried.

Offload request is performed via the `tls_dev_add` callback of `struct tlsdev_ops`:

```
int (*tls_dev_add)(struct net_device *netdev, struct sock *sk,
                   enum tls_offload_ctx_dir direction,
                   struct tls_crypto_info *crypto_info,
                   u32 start_offload_tcp_sn);
```

`direction` indicates whether the cryptographic information is for the received or transmitted packets. Driver uses the `sk` parameter to retrieve the connection 5-tuple and socket family (IPv4 vs IPv6). Cryptographic information in `crypto_info` includes the key, iv, salt as well as TLS record sequence number. `start_offload_tcp_sn` indicates which TCP sequence number corresponds to the beginning of the record with sequence number from `crypto_info`. The driver can add its state at the end of kernel structures (see `driver_state` members in `include/net/tls.h`) to avoid additional allocations and pointer dereferences.

28.2.1 TX

After TX state is installed, the stack guarantees that the first segment of the stream will start exactly at the `start_offload_tcp_sn` sequence number, simplifying TCP sequence number matching.

TX offload being fully initialized does not imply that all segments passing through the driver and which belong to the offloaded socket will be after the expected sequence number and will have kernel record information. In particular, already encrypted data may have been queued to the socket before installing the connection state in the kernel.

28.2.2 RX

In RX direction local networking stack has little control over the segmentation, so the initial records' TCP sequence number may be anywhere inside the segment.

28.3 Normal operation

At the minimum the device maintains the following state for each connection, in each direction:

- crypto secrets (key, iv, salt)
- crypto processing state (partial blocks, partial authentication tag, etc.)
- record metadata (sequence number, processing offset and length)
- expected TCP sequence number

There are no guarantees on record length or record segmentation. In particular segments may start at any point of a record and contain any number of records. Assuming segments are received in order, the device should be able to perform crypto operations and authentication regardless of segmentation. For this to be possible device has to keep small amount of segment-to-segment state. This includes at least:

- partial headers (if a segment carried only a part of the TLS header)
- partial data block
- partial authentication tag (all data had been seen but part of the authentication tag has to be written or read from the subsequent segment)

Record reassembly is not necessary for TLS offload. If the packets arrive in order the device should be able to handle them separately and make forward progress.

28.3.1 TX

The kernel stack performs record framing reserving space for the authentication tag and populating all other TLS header and tailer fields.

Both the device and the driver maintain expected TCP sequence numbers due to the possibility of retransmissions and the lack of software fallback once the packet reaches the device. For segments passed in order, the driver marks the packets with a connection identifier (note that a 5-tuple lookup is insufficient to identify packets requiring HW offload, see the [5-tuple matching limitations](#) section) and hands them to the device. The device identifies the packet as requiring TLS handling and confirms the sequence number matches its expectation. The device performs encryption and authentication of the record data. It replaces the authentication tag and TCP checksum with correct values.

28.3.2 RX

Before a packet is DMAed to the host (but after NIC's embedded switching and packet transformation functions) the device validates the Layer 4 checksum and performs a 5-tuple lookup to find any TLS connection the packet may belong to (technically a 4-tuple lookup is sufficient - IP addresses and TCP port numbers, as the protocol is always TCP). If connection is matched device confirms if the TCP sequence number is the expected one and proceeds to TLS handling (record delineation, decryption, authentication for each record in the packet). The device leaves the record framing unmodified, the stack takes care of record decapsulation. Device indicates successful handling of TLS offload in the per-packet context (descriptor) passed to the host.

Upon reception of a TLS offloaded packet, the driver sets the decrypted mark in `struct sk_buff` corresponding to the segment. Networking stack makes sure decrypted and non-decrypted segments do not get coalesced (e.g. by GRO or socket layer) and takes care of partial decryption.

28.4 Resync handling

In presence of packet drops or network packet reordering, the device may lose synchronization with the TLS stream, and require a resync with the kernel's TCP stack.

Note that resync is only attempted for connections which were successfully added to the device table and are in TLS_HW mode. For example, if the table was full when cryptographic state was installed in the kernel, such connection will never get offloaded. Therefore the resync request does not carry any cryptographic connection state.

28.4.1 TX

Segments transmitted from an offloaded socket can get out of sync in similar ways to the receive side-retransmissions - local drops are possible, though network reorders are not. There are currently two mechanisms for dealing with out of order segments.

Crypto state rebuilding

Whenever an out of order segment is transmitted the driver provides the device with enough information to perform cryptographic operations. This means most likely that the part of the record preceding the current segment has to be passed to the device as part of the packet context, together with its TCP sequence number and TLS record number. The device can then initialize its crypto state, process and discard the preceding data (to be able to insert the authentication tag) and move onto handling the actual packet.

In this mode depending on the implementation the driver can either ask for a continuation with the crypto state and the new sequence number (next expected segment is the one after the out of order one), or continue with the previous stream state - assuming that the out of order segment was just a retransmission. The former is simpler, and does not require retransmission detection therefore it is the recommended method until such time it is proven inefficient.

Next record sync

Whenever an out of order segment is detected the driver requests that the `ktls` software fall-back code encrypt it. If the segment's sequence number is lower than expected the driver assumes retransmission and doesn't change device state. If the segment is in the future, it may imply a local drop, the driver asks the stack to sync the device to the next record state and falls back to software.

Resync request is indicated with:

```
void tls_offload_tx_resync_request(struct sock *sk, u32 got_seq, u32 exp_seq)
```

Until resync is complete driver should not access its expected TCP sequence number (as it will be updated from a different context). Following helper should be used to test if resync is complete:

```
bool tls_offload_tx_resync_pending(struct sock *sk)
```

Next time `ktls` pushes a record it will first send its TCP sequence number and TLS record number to the driver. Stack will also make sure that the new record will start on a segment boundary (like it does when the connection is initially added).

28.4.2 RX

A small amount of RX reorder events may not require a full resynchronization. In particular the device should not lose synchronization when record boundary can be recovered:

Fig. 2: Reorder of non-header segment

Green segments are successfully decrypted, blue ones are passed as received on wire, red stripes mark start of new records.

In above case segment 1 is received and decrypted successfully. Segment 2 was dropped so 3 arrives out of order. The device knows the next record starts inside 3, based on record length in segment 1. Segment 3 is passed untouched, because due to lack of data from segment 2 the remainder of the previous record inside segment 3 cannot be handled. The device can, however, collect the authentication algorithm's state and partial block from the new record in segment 3 and when 4 and 5 arrive continue decryption. Finally when 2 arrives it's completely outside of expected window of the device so it's passed as is without special handling. `ktls` software fallback handles the decryption of record spanning segments 1, 2 and 3. The device did not get out of sync, even though two segments did not get decrypted.

Kernel synchronization may be necessary if the lost segment contained a record header and arrived after the next record header has already passed:

Fig. 3: Reorder of segment with a TLS header

In this example segment 2 gets dropped, and it contains a record header. Device can only detect that segment 4 also contains a TLS header if it knows the length of the previous record from segment 2. In this case the device will lose synchronization with the stream.

Stream scan resynchronization

When the device gets out of sync and the stream reaches TCP sequence numbers more than a max size record past the expected TCP sequence number, the device starts scanning for a known header pattern. For example for TLS 1.2 and TLS 1.3 subsequent bytes of value `0x03` `0x03` occur in the SSL/TLS version field of the header. Once pattern is matched the device continues attempting parsing headers at expected locations (based on the length fields at guessed locations). Whenever the expected location does not contain a valid header the scan is restarted.

When the header is matched the device sends a confirmation request to the kernel, asking if the guessed location is correct (if a TLS record really starts there), and which record sequence number the given header had. The kernel confirms the guessed location was correct and tells the device the record sequence number. Meanwhile, the device had been parsing and counting all records since the just-confirmed one, it adds the number of records it had seen to the record number provided by the kernel. At this point the device is in sync and can resume decryption at next segment boundary.

In a pathological case the device may latch onto a sequence of matching headers and never hear back from the kernel (there is no negative confirmation from the kernel). The implementation may choose to periodically restart scan. Given how unlikely falsely-matching stream is, however, periodic restart is not deemed necessary.

Special care has to be taken if the confirmation request is passed asynchronously to the packet stream and record may get processed by the kernel before the confirmation request.

Stack-driven resynchronization

The driver may also request the stack to perform resynchronization whenever it sees the records are no longer getting decrypted. If the connection is configured in this mode the stack automatically schedules resynchronization after it has received two completely encrypted records.

The stack waits for the socket to drain and informs the device about the next expected record number and its TCP sequence number. If the records continue to be received fully encrypted stack retries the synchronization with an exponential back off (first after 2 encrypted records, then after 4 records, after 8, after 16... up until every 128 records).

28.5 Error handling

28.5.1 TX

Packets may be redirected or rerouted by the stack to a different device than the selected TLS offload device. The stack will handle such condition using the `sk_validate_xmit_skb()` helper (TLS offload code installs `tls_validate_xmit_skb()` at this hook). Offload maintains information about all records until the data is fully acknowledged, so if skbs reach the wrong device they can be handled by software fallback.

Any device TLS offload handling error on the transmission side must result in the packet being dropped. For example if a packet got out of order due to a bug in the stack or the device, reached the device and can't be encrypted such packet must be dropped.

28.5.2 RX

If the device encounters any problems with TLS offload on the receive side it should pass the packet to the host's networking stack as it was received on the wire.

For example authentication failure for any record in the segment should result in passing the unmodified packet to the software fallback. This means packets should not be modified "in place". Splitting segments to handle partial decryption is not advised. In other words either all records in the packet had been handled successfully and authenticated or the packet has to be passed to the host's stack as it was on the wire (recovering original packet in the driver if device provides precise error is sufficient).

The Linux networking stack does not provide a way of reporting per-packet decryption and authentication errors, packets with errors must simply not have the decrypted mark set.

A packet should also not be handled by the TLS offload if it contains incorrect checksums.

28.6 Performance metrics

TLS offload can be characterized by the following basic metrics:

- max connection count
- connection installation rate
- connection installation latency
- total cryptographic performance

Note that each TCP connection requires a TLS session in both directions, the performance may be reported treating each direction separately.

28.6.1 Max connection count

The number of connections device can support can be exposed via devlink resource API.

28.6.2 Total cryptographic performance

Offload performance may depend on segment and record size.

Overload of the cryptographic subsystem of the device should not have significant performance impact on non-offloaded streams.

28.7 Statistics

Following minimum set of TLS-related statistics should be reported by the driver:

- `rx_tls_decrypted_packets` - number of successfully decrypted RX packets which were part of a TLS stream.
- `rx_tls_decrypted_bytes` - number of TLS payload bytes in RX packets which were successfully decrypted.

- `rx_tls_ctx` - number of TLS RX HW offload contexts added to device for decryption.
- `rx_tls_del` - number of TLS RX HW offload contexts deleted from device (connection has finished).
- **`rx_tls_resync_req_pkt` - number of received TLS packets with a resync request.**
- **`rx_tls_resync_req_start` - number of times the TLS async resync request was started.**
- **`rx_tls_resync_req_end` - number of times the TLS async resync request properly ended with providing the HW tracked tcp-seq.**
- **`rx_tls_resync_req_skip` - number of times the TLS async resync request procedure was started by not properly ended.**
- **`rx_tls_resync_res_ok` - number of times the TLS resync response call to the driver was successfully handled.**
- **`rx_tls_resync_res_skip` - number of times the TLS resync response call to the driver was terminated unsuccessfully.**
- `rx_tls_err` - number of RX packets which were part of a TLS stream but were not decrypted due to unexpected error in the state machine.
- `tx_tls_encrypted_packets` - number of TX packets passed to the device for encryption of their TLS payload.
- `tx_tls_encrypted_bytes` - number of TLS payload bytes in TX packets passed to the device for encryption.
- `tx_tls_ctx` - number of TLS TX HW offload contexts added to device for encryption.
- `tx_tls_ooo` - number of TX packets which were part of a TLS stream but did not arrive in the expected order.
- `tx_tls_skip_no_sync_data` - number of TX packets which were part of a TLS stream and arrived out-of-order, but skipped the HW offload routine and went to the regular transmit flow as they were retransmissions of the connection handshake.
- `tx_tls_drop_no_sync_data` - number of TX packets which were part of a TLS stream dropped, because they arrived out of order and associated record could not be found.
- `tx_tls_drop_bypass_req` - number of TX packets which were part of a TLS stream dropped, because they contain both data that has been encrypted by software and data that expects hardware crypto offload.

28.8 Notable corner cases, exceptions and additional requirements

28.8.1 5-tuple matching limitations

The device can only recognize received packets based on the 5-tuple of the socket. Current `ktls` implementation will not offload sockets routed through software interfaces such as those used for tunneling or virtual networking. However, many packet transformations performed by the networking stack (most notably any BPF logic) do not require any intermediate software

device, therefore a 5-tuple match may consistently miss at the device level. In such cases the device should still be able to perform TX offload (encryption) and should fallback cleanly to software decryption (RX).

28.8.2 Out of order

Introducing extra processing in NICs should not cause packets to be transmitted or received out of order, for example pure ACK packets should not be reordered with respect to data segments.

28.8.3 Ingress reorder

A device is permitted to perform packet reordering for consecutive TCP segments (i.e. placing packets in the correct order) but any form of additional buffering is disallowed.

28.8.4 Coexistence with standard networking offload features

Offloaded `ktls` sockets should support standard TCP stack features transparently. Enabling device TLS offload should not cause any difference in packets as seen on the wire.

28.8.5 Transport layer transparency

The device should not modify any packet headers for the purpose of simplifying TLS offload. The device should not depend on any packet headers beyond what is strictly necessary for TLS offload.

28.8.6 Segment drops

Dropping packets is acceptable only in the event of catastrophic system errors and should never be used as an error handling mechanism in cases arising from normal operation. In other words, reliance on TCP retransmissions to handle corner cases is not acceptable.

28.8.7 TLS device features

Drivers should ignore the changes to the TLS device feature flags. These flags will be acted upon accordingly by the core `ktls` code. TLS device feature flags only control adding of new TLS connection offloads, old connections will remain active after flags are cleared.

TLS encryption cannot be offloaded to devices without checksum calculation offload. Hence, TLS TX device feature flag requires TX csum offload being set. Disabling the latter implies clearing the former. Disabling TX checksum offload should not affect old connections, and drivers should make sure checksum calculation does not break for them. Similarly, device-offloaded TLS decryption implies doing RXCSUM. If the user does not want to enable RX csum offload, TLS RX device feature is disabled as well.

IN-KERNEL TLS HANDSHAKE

29.1 Overview

Transport Layer Security (TLS) is a Upper Layer Protocol (ULP) that runs over TCP. TLS provides end-to-end data integrity and confidentiality in addition to peer authentication.

The kernel's kTLS implementation handles the TLS record subprotocol, but does not handle the TLS handshake subprotocol which is used to establish a TLS session. Kernel consumers can use the API described here to request TLS session establishment.

There are several possible ways to provide a handshake service in the kernel. The API described here is designed to hide the details of those implementations so that in-kernel TLS consumers do not need to be aware of how the handshake gets done.

29.2 User handshake agent

As of this writing, there is no TLS handshake implementation in the Linux kernel. To provide a handshake service, a handshake agent (typically in user space) is started in each network namespace where a kernel consumer might require a TLS handshake. Handshake agents listen for events sent from the kernel that indicate a handshake request is waiting.

An open socket is passed to a handshake agent via a netlink operation, which creates a socket descriptor in the agent's file descriptor table. If the handshake completes successfully, the handshake agent promotes the socket to use the TLS ULP and sets the session information using the SOL_TLS socket options. The handshake agent returns the socket to the kernel via a second netlink operation.

29.3 Kernel Handshake API

A kernel TLS consumer initiates a client-side TLS handshake on an open socket by invoking one of the `tls_client_hello()` functions. First, it fills in a structure that contains the parameters of the request:

```
struct tls_handshake_args {
    struct socket *ta_sock;
    tls_done_func_t ta_done;
    void *ta_data;
    const char *ta_peername;
```

```

unsigned int ta_timeout_ms;
key_serial_t ta_keyring;
key_serial_t ta_my_cert;
key_serial_t ta_my_privkey;
unsigned int ta_num_peerids;
key_serial_t ta_my_peerids[5];
};

```

The @ta_sock field references an open and connected socket. The consumer must hold a reference on the socket to prevent it from being destroyed while the handshake is in progress. The consumer must also have instantiated a struct file in sock->file.

@ta_done contains a callback function that is invoked when the handshake has completed. Further explanation of this function is in the "Handshake Completion" section below.

The consumer can provide a NUL-terminated hostname in the @ta_peername field that is sent as part of ClientHello. If no peername is provided, the DNS hostname associated with the server's IP address is used instead.

The consumer can fill in the @ta_timeout_ms field to force the servicing handshake agent to exit after a number of milliseconds. This enables the socket to be fully closed once both the kernel and the handshake agent have closed their endpoints.

Authentication material such as x.509 certificates, private certificate keys, and pre-shared keys are provided to the handshake agent in keys that are instantiated by the consumer before making the handshake request. The consumer can provide a private keyring that is linked into the handshake agent's process keyring in the @ta_keyring field to prevent access of those keys by other subsystems.

To request an x.509-authenticated TLS session, the consumer fills in the @ta_my_cert and @ta_my_privkey fields with the serial numbers of keys containing an x.509 certificate and the private key for that certificate. Then, it invokes this function:

```
ret = tls_client_hello_x509(args, gfp_flags);
```

The function returns zero when the handshake request is under way. A zero return guarantees the callback function @ta_done will be invoked for this socket. The function returns a negative errno if the handshake could not be started. A negative errno guarantees the callback function @ta_done will not be invoked on this socket.

To initiate a client-side TLS handshake with a pre-shared key, use:

```
ret = tls_client_hello_psk(args, gfp_flags);
```

However, in this case, the consumer fills in the @ta_my_peerids array with serial numbers of keys containing the peer identities it wishes to offer, and the @ta_num_peerids field with the number of array entries it has filled in. The other fields are filled in as above.

To initiate an anonymous client-side TLS handshake use:

```
ret = tls_client_hello_anon(args, gfp_flags);
```

The handshake agent presents no peer identity information to the remote during this type of handshake. Only server authentication (ie the client verifies the server's identity) is performed during the handshake. Thus the established session uses encryption only.

Consumers that are in-kernel servers use:

```
ret = tls_server_hello_x509(args, gfp_flags);
```

or

```
ret = tls_server_hello_psk(args, gfp_flags);
```

The argument structure is filled in as above.

If the consumer needs to cancel the handshake request, say, due to a ^C or other exigent event, the consumer can invoke:

```
bool tls_handshake_cancel(sock);
```

This function returns true if the handshake request associated with @sock has been canceled. The consumer's handshake completion callback will not be invoked. If this function returns false, then the consumer's completion callback has already been invoked.

29.4 Handshake Completion

When the handshake agent has completed processing, it notifies the kernel that the socket may be used by the consumer again. At this point, the consumer's handshake completion callback, provided in the @ta_done field in the `tls_handshake_args` structure, is invoked.

The synopsis of this function is:

```
typedef void (*tls_done_func_t)(void *data, int status,
                               key_serial_t peerid);
```

The consumer provides a cookie in the @ta_data field of the `tls_handshake_args` structure that is returned in the @data parameter of this callback. The consumer uses the cookie to match the callback to the thread waiting for the handshake to complete.

The success status of the handshake is returned via the @status parameter:

status	meaning
0	TLS session established successfully
-EACCESS	Remote peer rejected the handshake or authentication failed
-ENOMEM	Temporary resource allocation failure
-EINVAL	Consumer provided an invalid argument
-ENOKEY	Missing authentication material
-EIO	An unexpected fault occurred

The @peerid parameter contains the serial number of a key containing the remote peer's identity or the value `TLS_NO_PEERID` if the session is not authenticated.

A best practice is to close and destroy the socket immediately if the handshake failed.

29.4.1 Other considerations

While a handshake is under way, the kernel consumer must alter the socket's `sk_data_ready` callback function to ignore all incoming data. Once the handshake completion callback function has been invoked, normal receive operation can be resumed.

Once a TLS session is established, the consumer must provide a buffer for and then examine the control message (CMSG) that is part of every subsequent `sock_recvmsg()`. Each control message indicates whether the received message data is TLS record data or session metadata.

See [Kernel TLS](#) for details on how a kTLS consumer recognizes incoming (decrypted) application data, alerts, and handshake packets once the socket has been promoted to use the TLS ULP.

LINUX NFC SUBSYSTEM

The Near Field Communication (NFC) subsystem is required to standardize the NFC device drivers development and to create an unified userspace interface.

This document covers the architecture overview, the device driver interface description and the userspace interface description.

30.1 Architecture overview

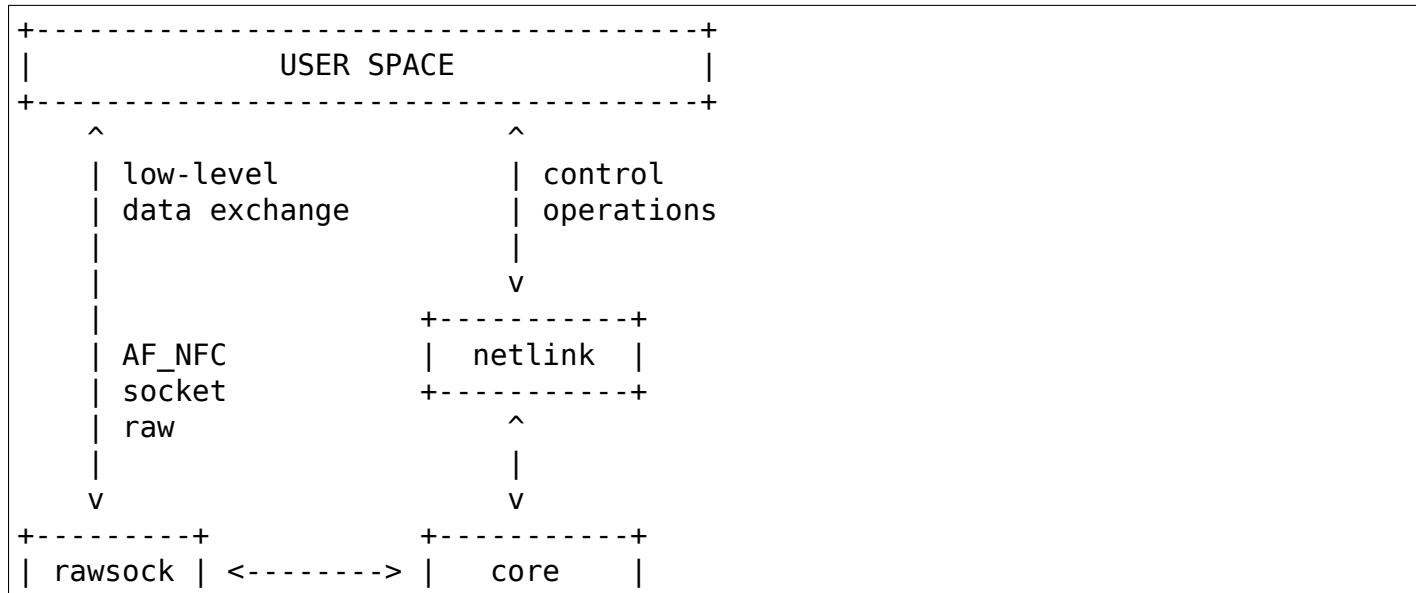
The NFC subsystem is responsible for:

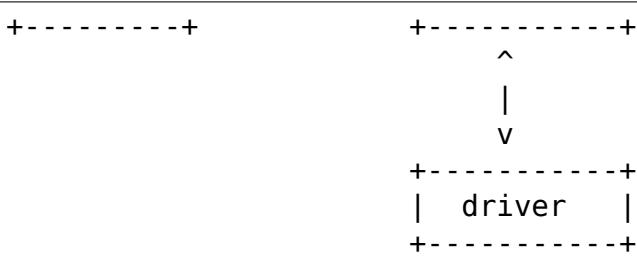
- NFC adapters management;
- Polling for targets;
- Low-level data exchange;

The subsystem is divided in some parts. The 'core' is responsible for providing the device driver interface. On the other side, it is also responsible for providing an interface to control operations and low-level data exchange.

The control operations are available to userspace via generic netlink.

The low-level data exchange interface is provided by the new socket family PF_NFC. The NFC_SOCKPROTO_RAW performs raw communication with NFC targets.





30.2 Device Driver Interface

When registering on the NFC subsystem, the device driver must inform the core of the set of supported NFC protocols and the set of ops callbacks. The ops callbacks that must be implemented are the following:

- `start_poll` - setup the device to poll for targets
- `stop_poll` - stop on progress polling operation
- `activate_target` - select and initialize one of the targets found
- `deactivate_target` - deselect and deinitialize the selected target
- `data_exchange` - send data and receive the response (transceive operation)

30.3 Userspace interface

The userspace interface is divided in control operations and low-level data exchange operation.

CONTROL OPERATIONS:

Generic netlink is used to implement the interface to the control operations. The operations are composed by commands and events, all listed below:

- `NFC_CMD_GET_DEVICE` - get specific device info or dump the device list
- `NFC_CMD_START_POLL` - setup a specific device to polling for targets
- `NFC_CMD_STOP_POLL` - stop the polling operation in a specific device
- `NFC_CMD_GET_TARGET` - dump the list of targets found by a specific device
- `NFC_EVENT_DEVICE_ADDED` - reports an NFC device addition
- `NFC_EVENT_DEVICE_REMOVED` - reports an NFC device removal
- `NFC_EVENT_TARGETS_FOUND` - reports START_POLL results when 1 or more targets are found

The user must call `START_POLL` to poll for NFC targets, passing the desired NFC protocols through `NFC_ATTR_PROTOCOLS` attribute. The device remains in polling state until it finds any target. However, the user can stop the polling operation by calling `STOP_POLL` command. In this case, it will be checked if the requester of `STOP_POLL` is the same of `START_POLL`.

If the polling operation finds one or more targets, the event `TARGETS_FOUND` is sent (including the device id). The user must call `GET_TARGET` to get the list of all targets found by such device.

Each reply message has target attributes with relevant information such as the supported NFC protocols.

All polling operations requested through one netlink socket are stopped when it's closed.

LOW-LEVEL DATA EXCHANGE:

The userspace must use PF_NFC sockets to perform any data communication with targets. All NFC sockets use AF_NFC:

```
struct sockaddr_nfc {
    sa_family_t sa_family;
    __u32 dev_idx;
    __u32 target_idx;
    __u32 nfc_protocol;
};
```

To establish a connection with one target, the user must create an NFC_SOCKPROTO_RAW socket and call the 'connect' syscall with the sockaddr_nfc struct correctly filled. All information comes from NFC_EVENT_TARGETS_FOUND netlink event. As a target can support more than one NFC protocol, the user must inform which protocol it wants to use.

Internally, 'connect' will result in an activate_target call to the driver. When the socket is closed, the target is deactivated.

The data format exchanged through the sockets is NFC protocol dependent. For instance, when communicating with MIFARE tags, the data exchanged are MIFARE commands and their responses.

The first received package is the response to the first sent package and so on. In order to allow valid "empty" responses, every data received has a NULL header of 1 byte.

NETDEV PRIVATE DATAROOM FOR 6LOWPAN INTERFACES

All 6lowpan able net devices, means all interfaces with ARPHRD_6LOWPAN, must have "struct lowpan_priv" placed at beginning of netdev_priv.

The priv_size of each interface should be calculate by:

```
dev->priv_size = LOWPAN_PRIV_SIZE(LL_6LOWPAN_PRIV_DATA);
```

Where LL_PRIV_6LOWPAN_DATA is sizeof linklayer 6lowpan private data struct. To access the LL_PRIV_6LOWPAN_DATA structure you can cast:

```
lowpan_priv(dev)->priv;
```

to your LL_6LOWPAN_PRIV_DATA structure.

Before registering the lowpan netdev interface you must run:

```
lowpan_netdev_setup(dev, LOWPAN_LLTYPE_FOOBAR);
```

wheres LOWPAN_LLTYPE_FOOBAR is a define for your 6LoWPAN linklayer type of enum lowpan_lltypes.

Example to evaluate the private usually you can do:

```
static inline struct lowpan_priv_foobar *
lowpan_foobar_priv(struct net_device *dev)
{
    return (struct lowpan_priv_foobar *)lowpan_priv(dev)->priv;
}

switch (dev->type) {
case ARPHRD_6LOWPAN:
    lowpan_priv = lowpan_priv(dev);
    /* do great stuff which is ARPHRD_6LOWPAN related */
    switch (lowpan_priv->lltype) {
        case LOWPAN_LLTYPE_FOOBAR:
            /* do 802.15.4 6LoWPAN handling here */
            lowpan_foobar_priv(dev)->bar = foo;
            break;
        ...
    }
    break;
}
```

```
...  
}
```

In case of generic 6lowpan branch ("net/6lowpan") you can remove the check on ARPHRD_6LOWPAN, because you can be sure that these function are called by ARPHRD_6LOWPAN interfaces.

CHAPTER
THIRTYTWO

6PACK PROTOCOL

This is the 6pack-mini-HOWTO, written by
Andreas König DG3KQ

Internet

ajk@comnets.uni-bremen.de

AMPR-net

dg3kq@db0pra.ampr.org

AX.25

dg3kq@db0ach.#nrw.deu.eu

Last update: April 7, 1998

32.1 1. What is 6pack, and what are the advantages to KISS?

6pack is a transmission protocol for data exchange between the PC and the TNC over a serial line. It can be used as an alternative to KISS.

6pack has two major advantages:

- The PC is given full control over the radio channel. Special control data is exchanged between the PC and the TNC so that the PC knows at any time if the TNC is receiving data, if a TNC buffer underrun or overrun has occurred, if the PTT is set and so on. This control data is processed at a higher priority than normal data, so a data stream can be interrupted at any time to issue an important event. This helps to improve the channel access and timing algorithms as everything is computed in the PC. It would even be possible to experiment with something completely different from the known CSMA and DAMA channel access methods. This kind of real-time control is especially important to supply several TNCs that are connected between each other and the PC by a daisy chain (however, this feature is not supported yet by the Linux 6pack driver).
- Each packet transferred over the serial line is supplied with a checksum, so it is easy to detect errors due to problems on the serial line. Received packets that are corrupt are not passed on to the AX.25 layer. Damaged packets that the TNC has received from the PC are not transmitted.

More details about 6pack are described in the file 6pack.ps that is located in the doc directory of the AX.25 utilities package.

32.2 2. Who has developed the 6pack protocol?

The 6pack protocol has been developed by Ekki Plicht DF4OR, Henning Rech DF9IC and Gunter Jost DK7WJ. A driver for 6pack, written by Gunter Jost and Matthias Welwarsky DG2FEF, comes along with the PC version of FlexNet. They have also written a firmware for TNCs to perform the 6pack protocol (see section 4 below).

32.3 3. Where can I get the latest version of 6pack for LinuX?

At the moment, the 6pack stuff can obtained via anonymous ftp from db0bm.automation.fh-aachen.de. In the directory /incoming/dg3kq, there is a file named 6pack.tgz.

32.4 4. Preparing the TNC for 6pack operation

To be able to use 6pack, a special firmware for the TNC is needed. The EPROM of a newly bought TNC does not contain 6pack, so you will have to program an EPROM yourself. The image file for 6pack EPROMs should be available on any packet radio box where PC/FlexNet can be found. The name of the file is 6pack.bin. This file is copyrighted and maintained by the FlexNet team. It can be used under the terms of the license that comes along with PC/FlexNet. Please do not ask me about the internals of this file as I don't know anything about it. I used a textual description of the 6pack protocol to program the Linux driver.

TNCs contain a 64kByte EPROM, the lower half of which is used for the firmware/KISS. The upper half is either empty or is sometimes programmed with software called TAPR. In the latter case, the TNC is supplied with a DIP switch so you can easily change between the two systems. When programming a new EPROM, one of the systems is replaced by 6pack. It is useful to replace TAPR, as this software is rarely used nowadays. If your TNC is not equipped with the switch mentioned above, you can build in one yourself that switches over the highest address pin of the EPROM between HIGH and LOW level. After having inserted the new EPROM and switched to 6pack, apply power to the TNC for a first test. The connect and the status LED are lit for about a second if the firmware initialises the TNC correctly.

32.5 5. Building and installing the 6pack driver

The driver has been tested with kernel version 2.1.90. Use with older kernels may lead to a compilation error because the interface to a kernel function has been changed in the 2.1.8x kernels.

32.6 How to turn on 6pack support:

- In the linux kernel configuration program, select the code maturity level options menu and turn on the prompting for development drivers.
- Select the amateur radio support menu and turn on the serial port 6pack driver.
- Compile and install the kernel and the modules.

To use the driver, the kissattach program delivered with the AX.25 utilities has to be modified.

- Do a cd to the directory that holds the kissattach sources. Edit the kissattach.c file. At the top, insert the following lines:

```
#ifndef N_6PACK
#define N_6PACK (N_AX25+1)
#endif
```

Then find the line:

```
int disc = N_AX25;
```

and replace N_AX25 by N_6PACK.

- Recompile kissattach. Rename it to spattach to avoid confusions.

32.6.1 Installing the driver:

- Do an insmod 6pack. Look at your /var/log/messages file to check if the module has printed its initialization message.
- Do a spattach as you would launch kissattach when starting a KISS port. Check if the kernel prints the message '6pack: TNC found'.
- From here, everything should work as if you were setting up a KISS port. The only difference is that the network device that represents the 6pack port is called sp instead of sl or ax. So, sp0 would be the first 6pack port.

Although the driver has been tested on various platforms, I still declare it ALPHA. BE CAREFUL! Sync your disks before insmoding the 6pack module and spattaching. Watch out if your computer behaves strangely. Read section 6 of this file about known problems.

Note that the connect and status LEDs of the TNC are controlled in a different way than they are when the TNC is used with PC/FlexNet. When using FlexNet, the connect LED is on if there is a connection; the status LED is on if there is data in the buffer of the PC's AX.25 engine that has to be transmitted. Under Linux, the 6pack layer is beyond the AX.25 layer, so the 6pack driver doesn't know anything about connects or data that has not yet been transmitted. Therefore the LEDs are controlled as they are in KISS mode: The connect LED is turned on if data is transferred from the PC to the TNC over the serial line, the status LED if data is sent to the PC.

32.7 6. Known problems

When testing the driver with 2.0.3x kernels and operating with data rates on the radio channel of 9600 Baud or higher, the driver may, on certain systems, sometimes print the message '6pack: bad checksum', which is due to data loss if the other station sends two or more subsequent packets. I have been told that this is due to a problem with the serial driver of 2.0.3x kernels. I don't know yet if the problem still exists with 2.1.x kernels, as I have heard that the serial driver code has been changed with 2.1.x.

When shutting down the sp interface with ifconfig, the kernel crashes if there is still an AX.25 connection left over which an IP connection was running, even if that IP connection is already closed. The problem does not occur when there is a bare AX.25 connection still running. I don't know if this is a problem of the 6pack driver or something else in the kernel.

The driver has been tested as a module, not yet as a kernel-builtin driver.

The 6pack protocol supports daisy-chaining of TNCs in a token ring, which is connected to one serial port of the PC. This feature is not implemented and at least at the moment I won't be able to do it because I do not have the opportunity to build a TNC daisy-chain and test it.

Some of the comments in the source code are inaccurate. They are left from the SLIP/KISS driver, from which the 6pack driver has been derived. I haven't modified or removed them yet -- sorry! The code itself needs some cleaning and optimizing. This will be done in a later release.

If you encounter a bug or if you have a question or suggestion concerning the driver, feel free to mail me, using the addresses given at the beginning of this file.

Have fun!

Andreas

ARCNET HARDWARE

Note:

- 1) This file is a supplement to *ARCnet*. Please read that for general driver configuration help.
 - 2) This file is no longer Linux-specific. It should probably be moved out of the kernel sources.
Ideas?
-

Because so many people (myself included) seem to have obtained ARCnet cards without manuals, this file contains a quick introduction to ARCnet hardware, some cabling tips, and a listing of all jumper settings I can find. Please e-mail apenwarr@worldvisions.ca with any settings for your particular card, or any other information you have!

33.1 Introduction to ARCnet

ARCnet is a network type which works in a way similar to popular Ethernet networks but which is also different in some very important ways.

First of all, you can get ARCnet cards in at least two speeds: 2.5 Mbps (slower than Ethernet) and 100 Mbps (faster than normal Ethernet). In fact, there are others as well, but these are less common. The different hardware types, as far as I'm aware, are not compatible and so you cannot wire a 100 Mbps card to a 2.5 Mbps card, and so on. From what I hear, my driver does work with 100 Mbps cards, but I haven't been able to verify this myself, since I only have the 2.5 Mbps variety. It is probably not going to saturate your 100 Mbps card. Stop complaining. :)

You also cannot connect an ARCnet card to any kind of Ethernet card and expect it to work.

There are two "types" of ARCnet - STAR topology and BUS topology. This refers to how the cards are meant to be wired together. According to most available documentation, you can only connect STAR cards to STAR cards and BUS cards to BUS cards. That makes sense, right? Well, it's not quite true; see below under "Cabling."

Once you get past these little stumbling blocks, ARCnet is actually quite a well-designed standard. It uses something called "modified token passing" which makes it completely incompatible with so-called "Token Ring" cards, but which makes transfers much more reliable than Ethernet does. In fact, ARCnet will guarantee that a packet arrives safely at the destination, and even if it can't possibly be delivered properly (ie. because of a cable break, or because the destination computer does not exist) it will at least tell the sender about it.

Because of the carefully defined action of the "token", it will always make a pass around the "ring" within a maximum length of time. This makes it useful for realtime networks.

In addition, all known ARCnet cards have an (almost) identical programming interface. This means that with one ARCnet driver you can support any card, whereas with Ethernet each manufacturer uses what is sometimes a completely different programming interface, leading to a lot of different, sometimes very similar, Ethernet drivers. Of course, always using the same programming interface also means that when high-performance hardware facilities like PCI bus mastering DMA appear, it's hard to take advantage of them. Let's not go into that.

One thing that makes ARCnet cards difficult to program for, however, is the limit on their packet sizes; standard ARCnet can only send packets that are up to 508 bytes in length. This is smaller than the Internet "bare minimum" of 576 bytes, let alone the Ethernet MTU of 1500. To compensate, an extra level of encapsulation is defined by RFC1201, which I call "packet splitting," that allows "virtual packets" to grow as large as 64K each, although they are generally kept down to the Ethernet-style 1500 bytes.

For more information on the advantages and disadvantages (mostly the advantages) of ARCnet networks, you might try the "ARCnet Trade Association" WWW page:

<http://www.arcnet.com>

33.2 Cabling ARCnet Networks

This section was rewritten by

Vojtech Pavlik <vojtech@suse.cz>

using information from several people, including:

- Avery Pennraun <apenwarr@worldvisions.ca>
- Stephen A. Wood <saw@hallc1.cebaf.gov>
- John Paul Morrison <jmorriso@bogomips.ee.ubc.ca>
- Joachim Koenig <jojo@repas.de>

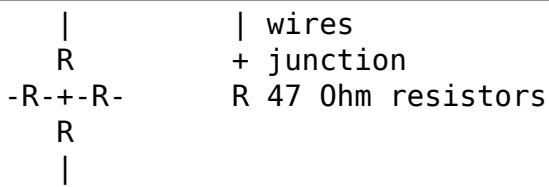
and Avery touched it up a bit, at Vojtech's request.

ARCnet (the classic 2.5 Mbps version) can be connected by two different types of cabling: coax and twisted pair. The other ARCnet-type networks (100 Mbps TCNS and 320 kbps - 32 Mbps ARCnet Plus) use different types of cabling (Type1, Fiber, C1, C4, C5).

For a coax network, you "should" use 93 Ohm RG-62 cable. But other cables also work fine, because ARCnet is a very stable network. I personally use 75 Ohm TV antenna cable.

Cards for coax cabling are shipped in two different variants: for BUS and STAR network topologies. They are mostly the same. The only difference lies in the hybrid chip installed. BUS cards use high impedance output, while STAR use low impedance. Low impedance card (STAR) is electrically equal to a high impedance one with a terminator installed.

Usually, the ARCnet networks are built up from STAR cards and hubs. There are two types of hubs - active and passive. Passive hubs are small boxes with four BNC connectors containing four 47 Ohm resistors:



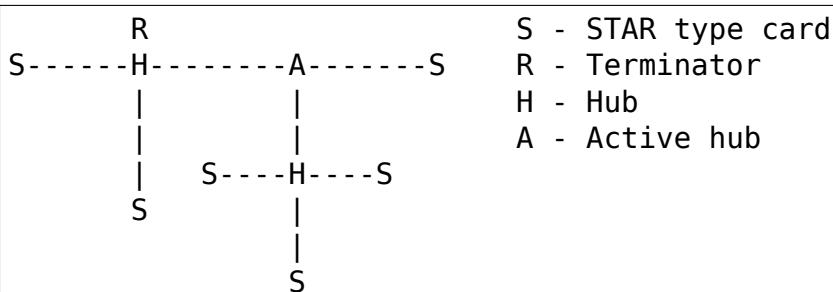
The shielding is connected together. Active hubs are much more complicated; they are powered and contain electronics to amplify the signal and send it to other segments of the net. They usually have eight connectors. Active hubs come in two variants - dumb and smart. The dumb variant just amplifies, but the smart one decodes to digital and encodes back all packets coming through. This is much better if you have several hubs in the net, since many dumb active hubs may worsen the signal quality.

And now to the cabling. What you can connect together:

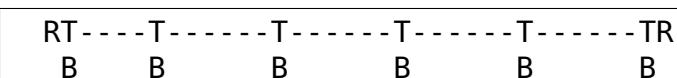
1. A card to a card. This is the simplest way of creating a 2-computer network.
2. A card to a passive hub. Remember that all unused connectors on the hub must be properly terminated with 93 Ohm (or something else if you don't have the right ones) terminators.
(Avery's note: oops, I didn't know that. Mine (TV cable) works anyway, though.)
3. A card to an active hub. Here is no need to terminate the unused connectors except some kind of aesthetic feeling. But, there may not be more than eleven active hubs between any two computers. That of course doesn't limit the number of active hubs on the network.
4. An active hub to another.
5. An active hub to passive hub.

Remember that you cannot connect two passive hubs together. The power loss implied by such a connection is too high for the net to operate reliably.

An example of a typical ARCnet network:



The BUS topology is very similar to the one used by Ethernet. The only difference is in cable and terminators: they should be 93 Ohm. Ethernet uses 50 Ohm impedance. You use T connectors to put the computers on a single line of cable, the bus. You have to put terminators at both ends of the cable. A typical BUS ARCnet network looks like:

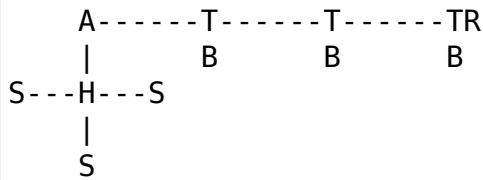


B - BUS type card

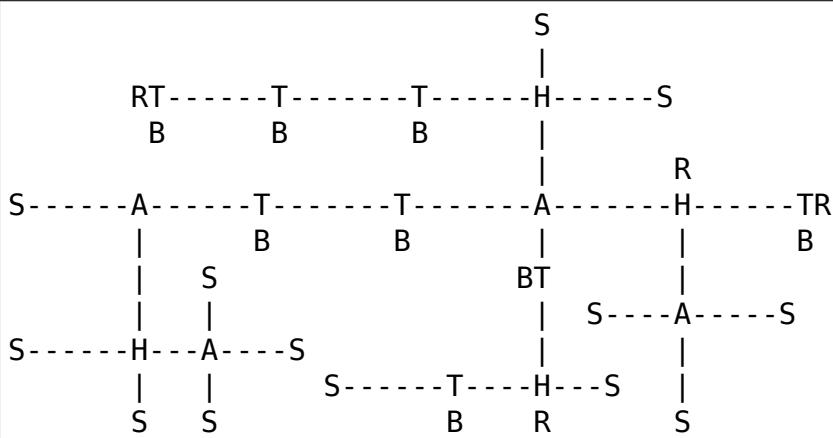
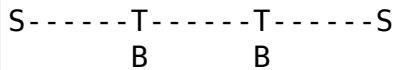
R - Terminator

T - T connector

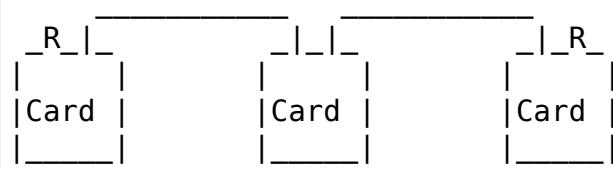
But that is not all! The two types can be connected together. According to the official documentation the only way of connecting them is using an active hub:



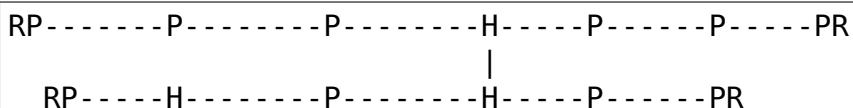
The official docs also state that you can use STAR cards at the ends of BUS network in place of a BUS card and a terminator:



A basically different cabling scheme is used with Twisted Pair cabling. Each of the TP cards has two RJ (phone-cord style) connectors. The cards are then daisy-chained together using a cable connecting every two neighboring cards. The ends are terminated with RJ 93 Ohm terminators which plug into the empty connectors of cards on the ends of the chain. An example:



There are also hubs for the TP topology. There is nothing difficult involved in using them; you just connect a TP chain to a hub on any end or even at both. This way you can create almost any network configuration. The maximum of 11 hubs between any two computers on the net applies here as well. An example:



 PR	 PR
--------	--------

R - RJ Terminator

P - TP Card

H - TP Hub

Like any network, ARCnet has a limited cable length. These are the maximum cable lengths between two active ends (an active end being an active hub or a STAR card).

RG-62	93 Ohm	up to 650 m
RG-59/U	75 Ohm	up to 457 m
RG-11/U	75 Ohm	up to 533 m
IBM Type 1	150 Ohm	up to 200 m
IBM Type 3	100 Ohm	up to 100 m

The maximum length of all cables connected to a passive hub is limited to 65 meters for RG-62 cabling; less for others. You can see that using passive hubs in a large network is a bad idea. The maximum length of a single "BUS Trunk" is about 300 meters for RG-62. The maximum distance between the two most distant points of the net is limited to 3000 meters. The maximum length of a TP cable between two cards/hubs is 650 meters.

33.3 Setting the Jumpers

All ARCnet cards should have a total of four or five different settings:

- the I/O address: this is the "port" your ARCnet card is on. Probed values in the Linux ARCnet driver are only from 0x200 through 0x3F0. (If your card has additional ones, which is possible, please tell me.) This should not be the same as any other device on your system. According to a doc I got from Novell, MS Windows prefers values of 0x300 or more, eating net connections on my system (at least) otherwise. My guess is this may be because, if your card is at 0x2E0, probing for a serial port at 0x2E8 will reset the card and probably mess things up royally.
 - Avery's favourite: 0x300.
- **the IRQ: on 8-bit cards, it might be 2 (9), 3, 4, 5, or 7.**
 on 16-bit cards, it might be 2 (9), 3, 4, 5, 7, or 10-15.

Make sure this is different from any other card on your system. Note that IRQ2 is the same as IRQ9, as far as Linux is concerned. You can "cat /proc/interrupts" for a somewhat complete list of which ones are in use at any given time. Here is a list of common usages from Vojtech Pavlik <vojtech@suse.cz>:

("Not on bus" means there is no way for a card to generate this interrupt)

IRQ 0	Timer 0 (Not on bus)
IRQ 1	Keyboard (Not on bus)
IRQ 2	IRQ Controller 2 (Not on bus, nor does interrupt the CPU)
IRQ 3	COM2
IRQ 4	COM1
IRQ 5	FREE (LPT2 if you have it; sometimes COM3; maybe PLIP)
IRQ 6	Floppy disk controller
IRQ 7	FREE (LPT1 if you don't use the polling driver; PLIP)
IRQ 8	Realtime Clock Interrupt (Not on bus)
IRQ 9	FREE (VGA vertical sync interrupt if enabled)
IRQ 10	FREE
IRQ 11	FREE
IRQ 12	FREE
IRQ 13	Numeric Coprocessor (Not on bus)
IRQ 14	Fixed Disk Controller
IRQ 15	FREE (Fixed Disk Controller 2 if you have it)

Note: IRQ 9 is used on some video cards for the "vertical retrace" interrupt. This interrupt would have been handy for things like video games, as it occurs exactly once per screen refresh, but unfortunately IBM cancelled this feature starting with the original VGA and thus many VGA/SVGA cards do not support it. For this reason, no modern software uses this interrupt and it can almost always be safely disabled, if your video card supports it at all.

If your card for some reason CANNOT disable this IRQ (usually there is a jumper), one solution would be to clip the printed circuit contact on the board: it's the fourth contact from the left on the back side. I take no responsibility if you try this.

- Avery's favourite: IRQ2 (actually IRQ9). Watch that VGA, though.

- the memory address: Unlike most cards, ARCnets use "shared memory" for copying buffers around. Make SURE it doesn't conflict with any other used memory in your system!

A0000	- VGA graphics memory (ok if you don't have VGA)
B0000	- Monochrome text mode
C0000	\ One of these is your VGA BIOS - usually C0000.
E0000	/
F0000	- System BIOS

Anything less than 0xA0000 is, well, a BAD idea since it isn't above 640k.

- Avery's favourite: 0xD0000

- the station address: Every ARCnet card has its own "unique" network address from 0 to 255. Unlike Ethernet, you can set this address yourself with a jumper or switch (or on some cards, with special software). Since it's only 8 bits, you can only have 254 ARCnet cards on a network. DON'T use 0 or 255, since these are reserved (although neat stuff will probably happen if you DO use them). By the way, if you haven't already guessed, don't set this the same as any other ARCnet on your network!

- Avery's favourite: 3 and 4. Not that it matters.
- There may be ETS1 and ETS2 settings. These may or may not make a difference on your card (many manuals call them "reserved"), but are used to change the delays used when powering up a computer on the network. This is only necessary when wiring VERY long range ARCnet networks, on the order of 4km or so; in any case, the only real requirement here is that all cards on the network with ETS1 and ETS2 jumpers have them in the same position. Chris Hindy <chrish@io.org> sent in a chart with actual values for this:

ET1	ET2	Response Time	Reconfiguration Time
open	open	74.7us	840us
open	closed	283.4us	1680us
closed	open	561.8us	1680us
closed	closed	1118.6us	1680us

Make sure you set ETS1 and ETS2 to the SAME VALUE for all cards on your network.

Also, on many cards (not mine, though) there are red and green LED's. Vojtech Pavlik <vojtech@suse.cz> tells me this is what they mean:

GREEN	RED	Status
OFF	OFF	Power off
OFF	Short flashes	Cabling problems (broken cable or not terminated)
OFF (short)	ON	Card init
ON	ON	Normal state - everything OK, nothing happens
ON	Long flashes	Data transfer
ON	OFF	Never happens (maybe when wrong ID)

The following is all the specific information people have sent me about their own particular ARCnet cards. It is officially a mess, and contains huge amounts of duplicated information. I have no time to fix it. If you want to, PLEASE DO! Just send me a 'diff -u' of all your changes.

The model # is listed right above specifics for that card, so you should be able to use your text viewer's "search" function to find the entry you want. If you don't KNOW what kind of card you have, try looking through the various diagrams to see if you can tell.

If your model isn't listed and/or has different settings, PLEASE PLEASE tell me. I had to figure mine out without the manual, and it WASN'T FUN!

Even if your ARCnet model isn't listed, but has the same jumpers as another model that is, please e-mail me to say so.

Cards Listed in this file (in this order, mostly):

Manufacturer	Model #	Bits
SMC	PC100	8
SMC	PC110	8
SMC	PC120	8
SMC	PC130	8
SMC	PC270E	8
SMC	PC500	16
SMC	PC500Longboard	16
SMC	PC550Longboard	16
SMC	PC600	16
SMC	PC710	8
SMC?	LCS-8830(-T)	8/16
Puredata	PDI507	8
CNet Tech	CN120-Series	8
CNet Tech	CN160-Series	16
Lantech?	UM9065L chipset	8
Acer	5210-003	8
Datapoint?	LAN-ARC-8	8
Topware	TA-ARC/10	8
Thomas-Conrad	500-6242-0097 REV A	8
Waterloo?	(C)1985 Waterloo Micro.	8
No Name	--	8/16
No Name	Taiwan R.O.C?	8
No Name	Model 9058	8
Tiara	Tiara Lancard?	8

- SMC = Standard Microsystems Corp.
- CNet Tech = CNet Technology, Inc.

33.4 Unclassified Stuff

- Please send any other information you can find.
- And some other stuff (more info is welcome!):

```
From: root@ultraworld.xs4all.nl (Timo Hilbrink)
To: apenwarr@foxnet.net (Avery Pennarun)
Date: Wed, 26 Oct 1994 02:10:32 +0000 (GMT)
Reply-To: timoh@xs4all.nl
```

[...parts deleted...]

About the jumpers: On my PC130 there is one more jumper, located near the cable-connector and it's for changing to star or bus topology;
 closed: star - open: bus

On the PC500 are some more jumper-pins, one block labeled with RX,PDN,TXI and another with ALE,LA17,LA18,LA19 these are undocumented..

[...more parts deleted...]

--- CUT ---

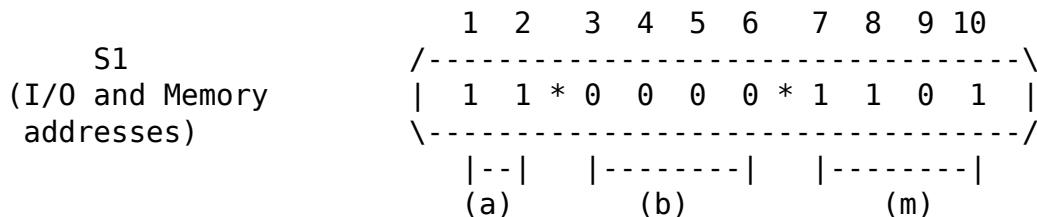
33.5 Standard Microsystems Corp (SMC)

33.5.1 PC100, PC110, PC120, PC130 (8-bit cards) and PC500, PC600 (16-bit cards)

- mainly from Avery Pennarun <apenwarr@worldvisions.ca>. Values depicted are from Avery's setup.
- special thanks to Timo Hilbrink <timoh@xs4all.nl> for noting that PC120, 130, 500, and 600 all have the same switches as Avery's PC100. PC500/600 have several extra, undocumented pins though. (?)
- PC110 settings were verified by Stephen A. Wood <saw@cebaf.gov>
- Also, the JP- and S-numbers probably don't match your card exactly. Try to find jumpers/switches with the same number of settings - it's probably more reliable.

JP5 (IRQ Setting)	$\begin{matrix} [] & : & : & : & : \\ \hline \end{matrix}$ IRQ2 IRQ3 IRQ4 IRQ5 IRQ7
----------------------	--

Put exactly one jumper on exactly one set of pins.



WARNING. It's very important when setting these which way you're holding the card, and which way you think is '1'!

If you suspect that your settings are not being made correctly, try reversing the direction or inverting the switch positions.

a: The first digit of the I/O address.

Setting	Value
-----	-----
00	0
01	1
10	2
11	3

b: The second digit of the I/O address.

Setting	Value
-----	-----

0000	0
0001	1
0010	2
...	...
1110	E
1111	F

The I/O address is in the form ab0. For example, if a is 0x2 and b is 0xE, the address will be 0x2E0.

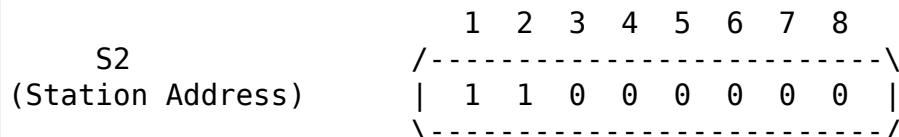
DO NOT SET THIS LESS THAN 0x200!!!!!

m: The first digit of the memory address.

Setting	Value
-----	-----
0000	0
0001	1
0010	2
...	...
1110	E
1111	F

The memory address is in the form m0000. For example, if m is D, the address will be 0xD0000.

DO NOT SET THIS TO C0000, F0000, OR LESS THAN A0000!



Setting	Value
-----	-----
00000000	00
10000000	01
01000000	02
...	...
01111111	FE
11111111	FF

Note that this is binary with the digits reversed!

DO NOT SET THIS TO 0 OR 255 (0xFF)!

33.5.2 PC130E/PC270E (8-bit cards)

- from Juergen Seifert <seifert@htwm.de>

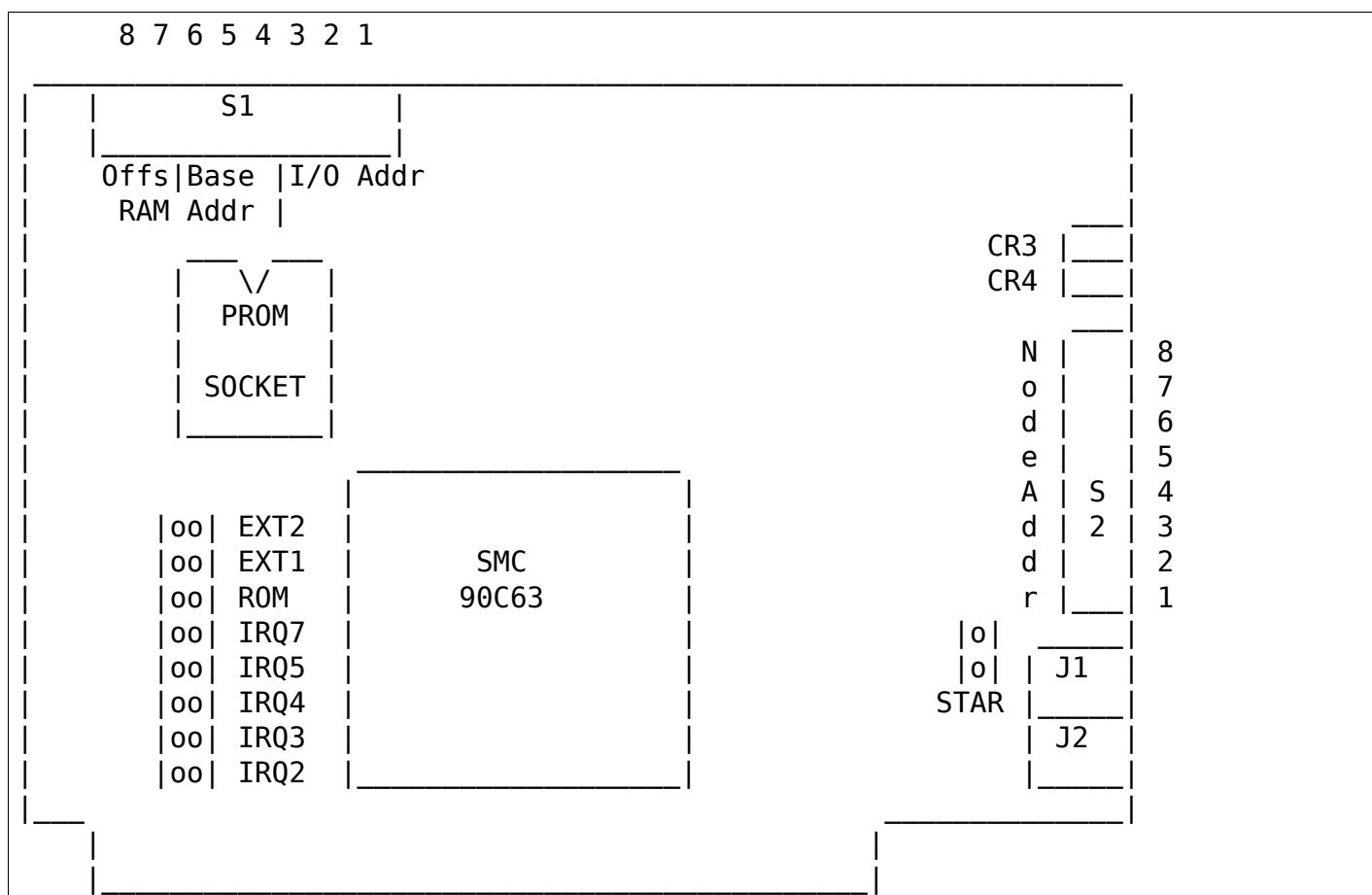
This description has been written by Juergen Seifert <seifert@htwm.de> using information from the following Original SMC Manual

"Configuration Guide for ARCNET(R)-PC130E/PC270 Network Controller Boards
Pub. # 900.044A June, 1989"

ARCNET is a registered trademark of the Datapoint Corporation SMC is a registered trademark of the Standard Microsystems Corporation

The PC130E is an enhanced version of the PC130 board, is equipped with a standard BNC female connector for connection to RG-62/U coax cable. Since this board is designed both for point-to-point connection in star networks and for connection to bus networks, it is downwardly compatible with all the other standard boards designed for coax networks (that is, the PC120, PC110 and PC100 star topology boards and the PC220, PC210 and PC200 bus topology boards).

The PC270E is an enhanced version of the PC260 board, is equipped with two modular RJ11-type jacks for connection to twisted pair wiring. It can be used in a star or a daisy-chained network.



Legend:

SMC 90C63	ARCNET Controller / Transceiver / Logic
S1 1-3:	I/O Base Address Select
4-6:	Memory Base Address Select

	7-8:	RAM Offset Select	
S2	1-8:	Node ID Select	
EXT		Extended Timeout Select	
ROM		ROM Enable Select	
STAR		Selected - Star Topology	(PC130E only)
		Deselected - Bus Topology	(PC130E only)
CR3/CR4		Diagnostic LEDs	
J1		BNC RG62/U Connector	(PC130E only)
J1		6-position Telephone Jack	(PC270E only)
J2		6-position Telephone Jack	(PC270E only)

Setting one of the switches to Off/Open means "1", On/Closed means "0".

Setting the Node ID

The eight switches in group S2 are used to set the node ID. These switches work in a way similar to the PC100-series cards; see that entry for more information.

Setting the I/O Base Address

The first three switches in switch group S1 are used to select one of eight possible I/O Base addresses using the following table:

Switch	Hex I/O Address
1 2 3	Address
-----	-----
0 0 0	260
0 0 1	290
0 1 0	2E0 (Manufacturer's default)
0 1 1	2F0
1 0 0	300
1 0 1	350
1 1 0	380
1 1 1	3E0

Setting the Base Memory (RAM) buffer Address

The memory buffer requires 2K of a 16K block of RAM. The base of this 16K block can be located in any of eight positions. Switches 4-6 of switch group S1 select the Base of the 16K block. Within that 16K address space, the buffer may be assigned any one of four positions, determined by the offset, switches 7 and 8 of group S1.

Switch	Hex RAM Address	Hex ROM Address *)
4 5 6 7 8	Address	Address *)
-----	-----	-----
0 0 0 0 0	C0000	C2000
0 0 0 0 1	C0800	C2000
0 0 0 1 0	C1000	C2000
0 0 0 1 1	C1800	C2000

0 0 1 0 0	C4000	C6000
0 0 1 0 1	C4800	C6000
0 0 1 1 0	C5000	C6000
0 0 1 1 1	C5800	C6000
0 1 0 0 0	CC000	CE000
0 1 0 0 1	CC800	CE000
0 1 0 1 0	CD000	CE000
0 1 0 1 1	CD800	CE000
0 1 1 0 0	D0000	D2000 (Manufacturer's default)
0 1 1 0 1	D0800	D2000
0 1 1 1 0	D1000	D2000
0 1 1 1 1	D1800	D2000
1 0 0 0 0	D4000	D6000
1 0 0 0 1	D4800	D6000
1 0 0 1 0	D5000	D6000
1 0 0 1 1	D5800	D6000
1 0 1 0 0	D8000	DA000
1 0 1 0 1	D8800	DA000
1 0 1 1 0	D9000	DA000
1 0 1 1 1	D9800	DA000
1 1 0 0 0	DC000	DE000
1 1 0 0 1	DC800	DE000
1 1 0 1 0	DD000	DE000
1 1 0 1 1	DD800	DE000
1 1 1 0 0	E0000	E2000
1 1 1 0 1	E0800	E2000
1 1 1 1 0	E1000	E2000
1 1 1 1 1	E1800	E2000

*) To enable the 8K Boot PROM install the jumper ROM.
The default is jumper ROM not installed.

Setting the Timeouts and Interrupt

The jumpers labeled EXT1 and EXT2 are used to determine the timeout parameters. These two jumpers are normally left open.

To select a hardware interrupt level set one (only one!) of the jumpers IRQ2, IRQ3, IRQ4, IRQ5, IRQ7. The Manufacturer's default is IRQ2.

Configuring the PC130E for Star or Bus Topology

The single jumper labeled STAR is used to configure the PC130E board for star or bus topology. When the jumper is installed, the board may be used in a star network, when it is removed, the board can be used in a bus topology.

Diagnostic LEDs

Two diagnostic LEDs are visible on the rear bracket of the board. The green LED monitors the network activity: the red one shows the board activity:

Green	Status	Red	Status
on	normal activity	flash/on	data transfer
blink	reconfiguration	off	no data transfer;
off	defective board or node ID is zero		incorrect memory or I/O address

33.5.3 PC500/PC550 Longboard (16-bit cards)

- from Juergen Seifert <seifert@htwm.de>

Note: There is another Version of the PC500 called Short Version, which is different in hard- and software! The most important differences are:

- The long board has no Shared memory.
 - On the long board the selection of the interrupt is done by binary coded switch, on the short board directly by jumper.
-

[Avery's note: pay special attention to that: the long board HAS NO SHARED MEMORY. This means the current Linux-ARCnet driver can't use these cards. I have obtained a PC500Longboard and will be doing some experiments on it in the future, but don't hold your breath. Thanks again to Juergen Seifert for his advice about this!]

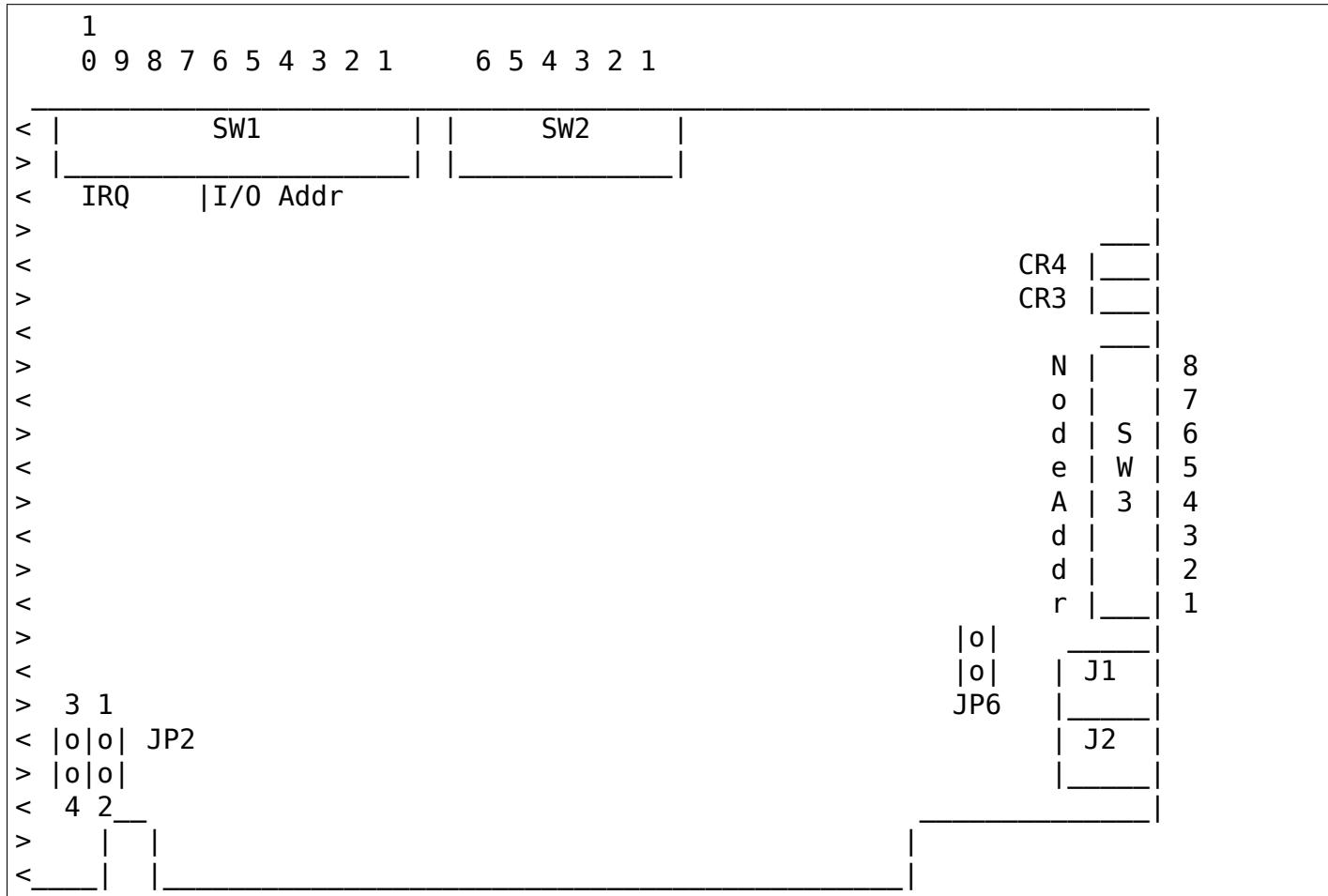
This description has been written by Juergen Seifert <seifert@htwm.de> using information from the following Original SMC Manual

"Configuration Guide for SMC ARCNET-PC500/PC550 Series Network Controller Boards Pub. # 900.033 Rev. A November, 1989"

ARCNET is a registered trademark of the Datapoint Corporation SMC is a registered trademark of the Standard Microsystems Corporation

The PC500 is equipped with a standard BNC female connector for connection to RG-62/U coax cable. The board is designed both for point-to-point connection in star networks and for connection to bus networks.

The PC550 is equipped with two modular RJ11-type jacks for connection to twisted pair wiring. It can be used in a star or a daisy-chained (BUS) network.



Legend:

SW1	1-6:	I/O Base Address Select	
	7-10:	Interrupt Select	
SW2	1-6:	Reserved for Future Use	
SW3	1-8:	Node ID Select	
JP2	1-4:	Extended Timeout Select	
JP6		Selected - Star Topology	(PC500 only)
		Deselected - Bus Topology	(PC500 only)
CR3	Green	Monitors Network Activity	
CR4	Red	Monitors Board Activity	
J1		BNC RG62/U Connector	(PC500 only)
J1		6-position Telephone Jack	(PC550 only)
J2		6-position Telephone Jack	(PC550 only)

Setting one of the switches to Off/Open means "1", On/Closed means "0".

Setting the Node ID

The eight switches in group SW3 are used to set the node ID. Each node attached to the network must have an unique node ID which must be different from 0. Switch 1 serves as the least significant bit (LSB).

The node ID is the sum of the values of all switches set to "1" These values are:

Switch	Value
1	1
2	2
3	4
4	8
5	16
6	32
7	64
8	128

Some Examples:

Switch 8 7 6 5 4 3 2 1	Hex Node ID	Decimal Node ID
0 0 0 0 0 0 0 0		not allowed
0 0 0 0 0 0 0 1	1	1
0 0 0 0 0 0 1 0	2	2
0 0 0 0 0 0 1 1	3	3
.	.	.
0 1 0 1 0 1 0 1	55	85
.	.	.
1 0 1 0 1 0 1 0	AA	170
.	.	.
1 1 1 1 1 1 0 1	FD	253
1 1 1 1 1 1 1 0	FE	254
1 1 1 1 1 1 1 1	FF	255

Setting the I/O Base Address

The first six switches in switch group SW1 are used to select one of 32 possible I/O Base addresses using the following table:

Switch 6 5 4 3 2 1	Hex I/O Address
0 1 0 0 0 0	200
0 1 0 0 0 1	210
0 1 0 0 1 0	220
0 1 0 0 1 1	230
0 1 0 1 0 0	240
0 1 0 1 0 1	250

0	1	0	1	1	0		260
0	1	0	1	1	1		270
0	1	1	0	0	0		280
0	1	1	0	0	1		290
0	1	1	0	1	0		2A0
0	1	1	0	1	1		2B0
0	1	1	1	0	0		2C0
0	1	1	1	0	1		2D0
0	1	1	1	1	0		2E0 (Manufacturer's default)
0	1	1	1	1	1		2F0
1	1	0	0	0	0		300
1	1	0	0	0	1		310
1	1	0	0	1	0		320
1	1	0	0	1	1		330
1	1	0	1	0	0		340
1	1	0	1	0	1		350
1	1	0	1	1	0		360
1	1	0	1	1	1		370
1	1	1	0	0	0		380
1	1	1	0	0	1		390
1	1	1	0	1	0		3A0
1	1	1	0	1	1		3B0
1	1	1	1	0	0		3C0
1	1	1	1	0	1		3D0
1	1	1	1	1	0		3E0
1	1	1	1	1	1		3F0

Setting the Interrupt

Switches seven through ten of switch group SW1 are used to select the interrupt level. The interrupt level is binary coded, so selections from 0 to 15 would be possible, but only the following eight values will be supported: 3, 4, 5, 7, 9, 10, 11, 12.

Switch	IRQ
10 9 8 7	
-----	-----
0 0 1 1	3
0 1 0 0	4
0 1 0 1	5
0 1 1 1	7
1 0 0 1	9 (=2) (default)
1 0 1 0	10
1 0 1 1	11
1 1 0 0	12

Setting the Timeouts

The two jumpers JP2 (1-4) are used to determine the timeout parameters. These two jumpers are normally left open. Refer to the COM9026 Data Sheet for alternate configurations.

Configuring the PC500 for Star or Bus Topology

The single jumper labeled JP6 is used to configure the PC500 board for star or bus topology. When the jumper is installed, the board may be used in a star network, when it is removed, the board can be used in a bus topology.

Diagnostic LEDs

Two diagnostic LEDs are visible on the rear bracket of the board. The green LED monitors the network activity: the red one shows the board activity:

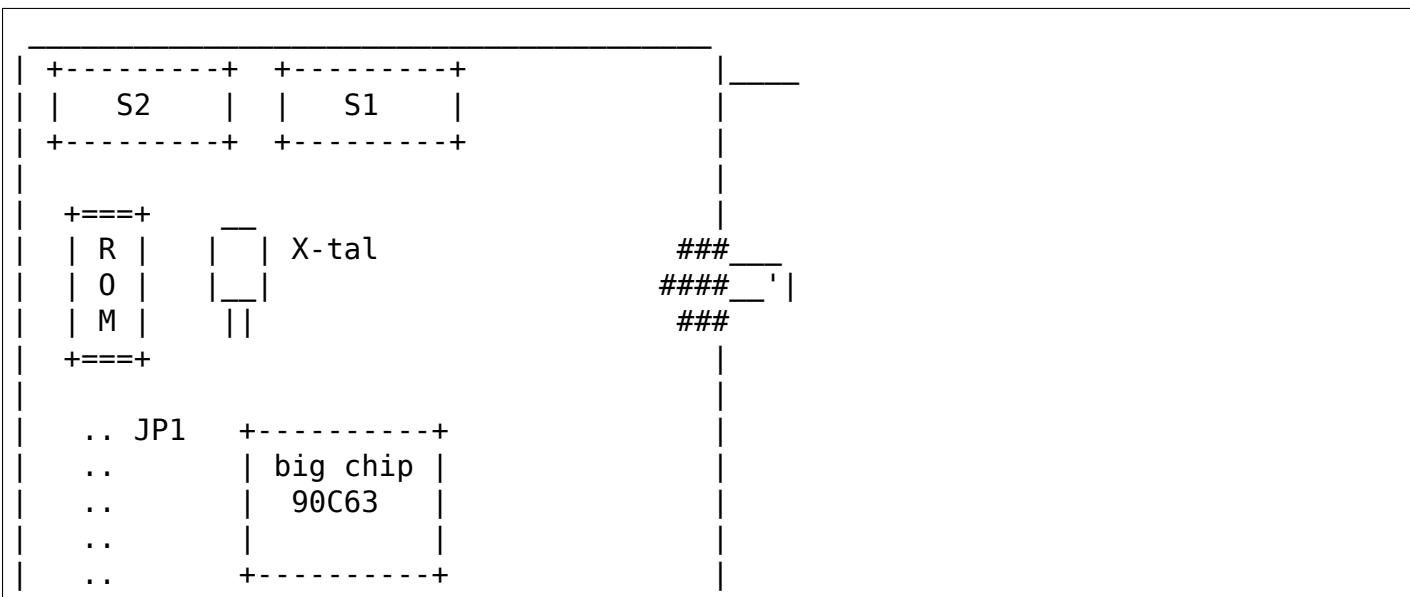
Green	Status	Red	Status
on	normal activity	flash/on	data transfer
blink	reconfiguration	off	no data transfer;
off	defective board or node ID is zero		incorrect memory or I/O address

33.5.4 PC710 (8-bit card)

- from J.S. van Oosten <jvoosten@compiler.tdcnet.nl>

Note: this data is gathered by experimenting and looking at info of other cards. However, I'm sure I got 99% of the settings right.

The SMC710 card resembles the PC270 card, but is much more basic (i.e. no LEDs, RJ11 jacks, etc.) and 8 bit. Here's a little drawing:





The row of jumpers at JP1 actually consists of 8 jumpers, (sometimes labelled) the same as on the PC270, from top to bottom: EXT2, EXT1, ROM, IRQ7, IRQ5, IRQ4, IRQ3, IRQ2 (gee, wonder what they would do? :-))

S1 and S2 perform the same function as on the PC270, only their numbers are swapped (S1 is the nodeaddress, S2 sets IO- and RAM-address).

I know it works when connected to a PC110 type ARCnet board.

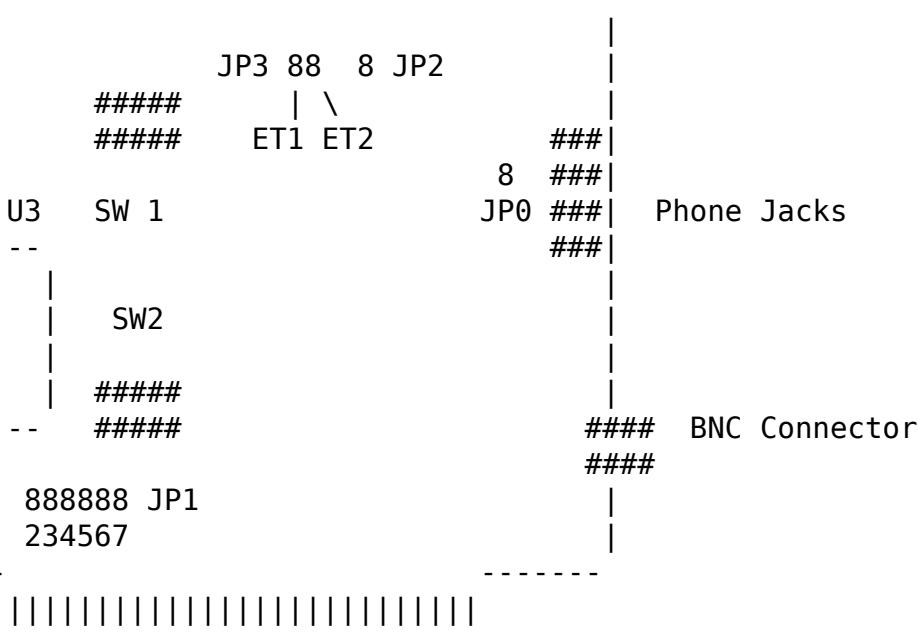
33.6 Possibly SMC

33.6.1 LCS-8830(-T) (8 and 16-bit cards)

- from Mathias Katzer <mkatzer@HRZ.Uni-Bielefeld.DE>
- Marek Michalkiewicz <marekm@i17linuxb.ists.pwr.wroc.pl> says the LCS-8830 is slightly different from LCS-8830-T. These are 8 bit, BUS only (the JP0 jumper is hardwired), and BNC only.

This is a LCS-8830-T made by SMC, I think ('SMC' only appears on one PLCC, nowhere else, not even on the few Xeroxed sheets from the manual).

SMC ARCnet Board Type LCS-8830-T:



SW1: DIP-Switches for Station Address

SW2: DIP-Switches for Memory Base and I/O Base addresses

JP0: If closed, internal termination on (default open)

JP1: IRQ Jumpers

JP2: Boot-ROM enabled if closed

JP3: Jumpers for response timeout

U3: Boot-ROM Socket

ET1	ET2	Response Time	Idle Time	Reconfiguration Time
		78	86	840
X		285	316	1680
	X	563	624	1680
X	X	1130	1237	1680

(X means closed jumper)

(DIP-Switch downwards means "0")

The station address is binary-coded with SW1.

The I/O base address is coded with DIP-Switches 6,7 and 8 of SW2:

Switches	Base
678	Address
000	260-26f
100	290-29f
010	2e0-2ef
110	2f0-2ff
001	300-30f
101	350-35f
011	380-38f
111	3e0-3ef

DIP Switches 1-5 of SW2 encode the RAM and ROM Address Range:

Switches	RAM	ROM
12345	Address Range	Address Range
00000	C:0000-C:07ff	C:2000-C:3fff
10000	C:0800-C:0fff	
01000	C:1000-C:17ff	
11000	C:1800-C:1fff	
00100	C:4000-C:47ff	C:6000-C:7fff
10100	C:4800-C:4fff	
01100	C:5000-C:57ff	
11100	C:5800-C:5fff	
00010	C:C000-C:C7ff	C:E000-C:ffff
10010	C:C800-C:Cfff	

continues on next page

Table 1 – continued from previous page

Switches	RAM	ROM
12345	Address Range	Address Range
01010	C:D000-C:D7ff	
11010	C:D800-C:Dfff	
00110	D:0000-D:07ff	D:2000-D:3fff
10110	D:0800-D:0fff	
01110	D:1000-D:17ff	
11110	D:1800-D:1fff	
00001	D:4000-D:47ff	D:6000-D:7fff
10001	D:4800-D:4fff	
01001	D:5000-D:57ff	
11001	D:5800-D:5fff	
00101	D:8000-D:87ff	D:A000-D:bfff
10101	D:8800-D:8fff	
01101	D:9000-D:97ff	
11101	D:9800-D:9fff	
00011	D:C000-D:c7ff	D:E000-D:ffff
10011	D:C800-D:cfff	
01011	D:D000-D:d7ff	
11011	D:D800-D:ffff	
00111	E:0000-E:07ff	E:2000-E:3fff
10111	E:0800-E:0fff	
01111	E:1000-E:17ff	
11111	E:1800-E:1fff	

33.7 PureData Corp

33.7.1 PDI507 (8-bit card)

- from Mark Rejhon <mdrejhon@magi.com> (slight modifications by Avery)
- Avery's note: I think PDI508 cards (but definitely NOT PDI508Plus cards) are mostly the same as this. PDI508Plus cards appear to be mainly software-configured.

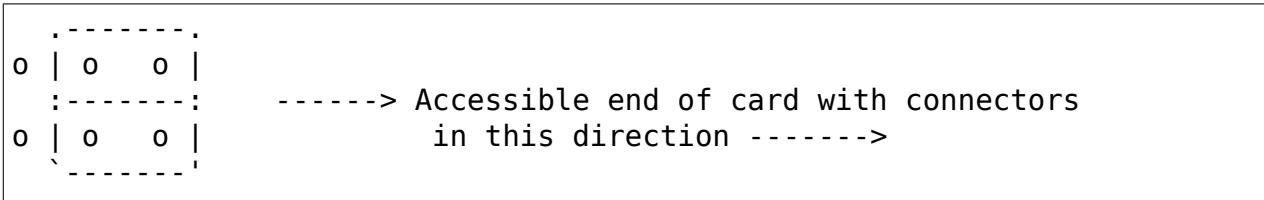
Jumpers:

There is a jumper array at the bottom of the card, near the edge connector. This array is labelled J1. They control the IRQs and something else. Put only one jumper on the IRQ pins.

ETS1, ETS2 are for timing on very long distance networks. See the more general information near the top of this file.

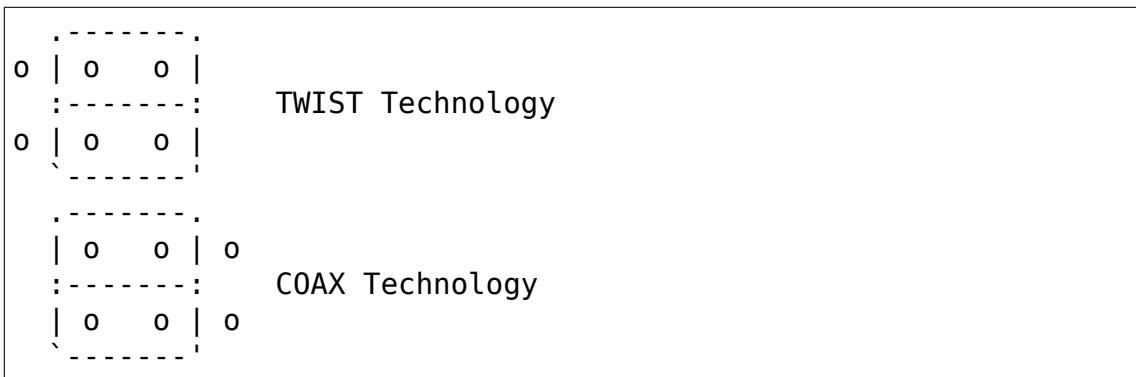
There is a J2 jumper on two pins. A jumper should be put on them, since it was already there when I got the card. I don't know what this jumper is for though.

There is a two-jumper array for J3. I don't know what it is for, but there were already two jumpers on it when I got the card. It's a six pin grid in a two-by-three fashion. The jumpers were configured as follows:



Carl de Billy <CARL@carainfo.com> explains J3 and J4:

J3 Diagram:



- If using coax cable in a bus topology the J4 jumper must be removed; place it on one pin.
- If using bus topology with twisted pair wiring move the J3 jumpers so they connect the middle pin and the pins closest to the RJ11 Connectors. Also the J4 jumper must be removed; place it on one pin of J4 jumper for storage.
- If using star topology with twisted pair wiring move the J3 jumpers so they connect the middle pin and the pins closest to the RJ11 connectors.

DIP Switches:

The DIP switches accessible on the accessible end of the card while it is installed, is used to set the ARCnet address. There are 8 switches. Use an address from 1 to 254

Switch No.	ARCnet address
12345678	
00000000	FF (Don't use this!)
00000001	FE
00000010	FD
...	
11111101	2
11111110	1
11111111	0 (Don't use this!)

There is another array of eight DIP switches at the top of the card. There are five labelled MS0-MS4 which seem to control the memory address, and another three labelled IO0-IO2 which seem to control the base I/O address of the card.

This was difficult to test by trial and error, and the I/O addresses are in a weird order. This was tested by setting the DIP switches, rebooting the computer, and attempting

to load ARCEther at various addresses (mostly between 0x200 and 0x400). The address that caused the red transmit LED to blink, is the one that I thought works.

Also, the address 0x3D0 seem to have a special meaning, since the ARCEther packet driver loaded fine, but without the red LED blinking. I don't know what 0x3D0 is for though. I recommend using an address of 0x300 since Windows may not like addresses below 0x300.

IO Switch No.	I/O address
210	
111	0x260
110	0x290
101	0x2E0
100	0x2F0
011	0x300
010	0x350
001	0x380
000	0x3E0

The memory switches set a reserved address space of 0x1000 bytes (0x100 segment units, or 4k). For example if I set an address of 0xD000, it will use up addresses 0xD000 to 0xD100.

The memory switches were tested by booting using QEMM386 stealth, and using LOADHI to see what address automatically became excluded from the upper memory regions, and then attempting to load ARCEther using these addresses.

I recommend using an ARCnet memory address of 0xD000, and putting the EMS page frame at 0xC000 while using QEMM stealth mode. That way, you get contiguous high memory from 0xD100 almost all the way the end of the megabyte.

Memory Switch 0 (MS0) didn't seem to work properly when set to OFF on my card. It could be malfunctioning on my card. Experiment with it ON first, and if it doesn't work, set it to OFF. (It may be a modifier for the 0x200 bit?)

MS Switch No.	
43210	Memory address
00001	0xE100 (guessed - was not detected by QEMM)
00011	0xE000 (guessed - was not detected by QEMM)
00101	0xDD00
00111	0xDC00
01001	0xD900
01011	0xD800
01101	0xD500
01111	0xD400
10001	0xD100
10011	0xD000
10101	0xCD00
10111	0xCC00
11001	0xC900 (guessed - crashes tested system)
11011	0xC800 (guessed - crashes tested system)
11101	0xC500 (guessed - crashes tested system)
11111	0xC400 (guessed - crashes tested system)

33.8 CNet Technology Inc. (8-bit cards)

33.8.1 120 Series (8-bit cards)

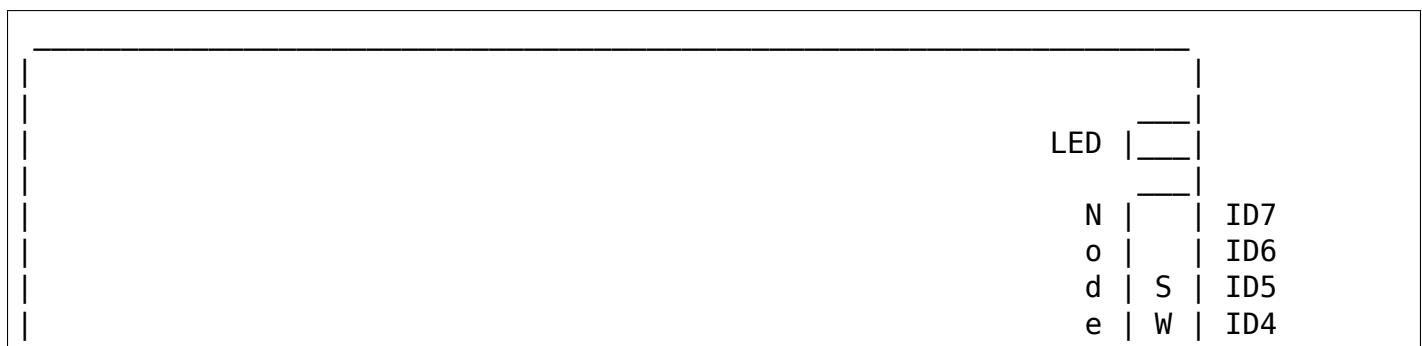
- from Juergen Seifert <seifert@htwm.de>

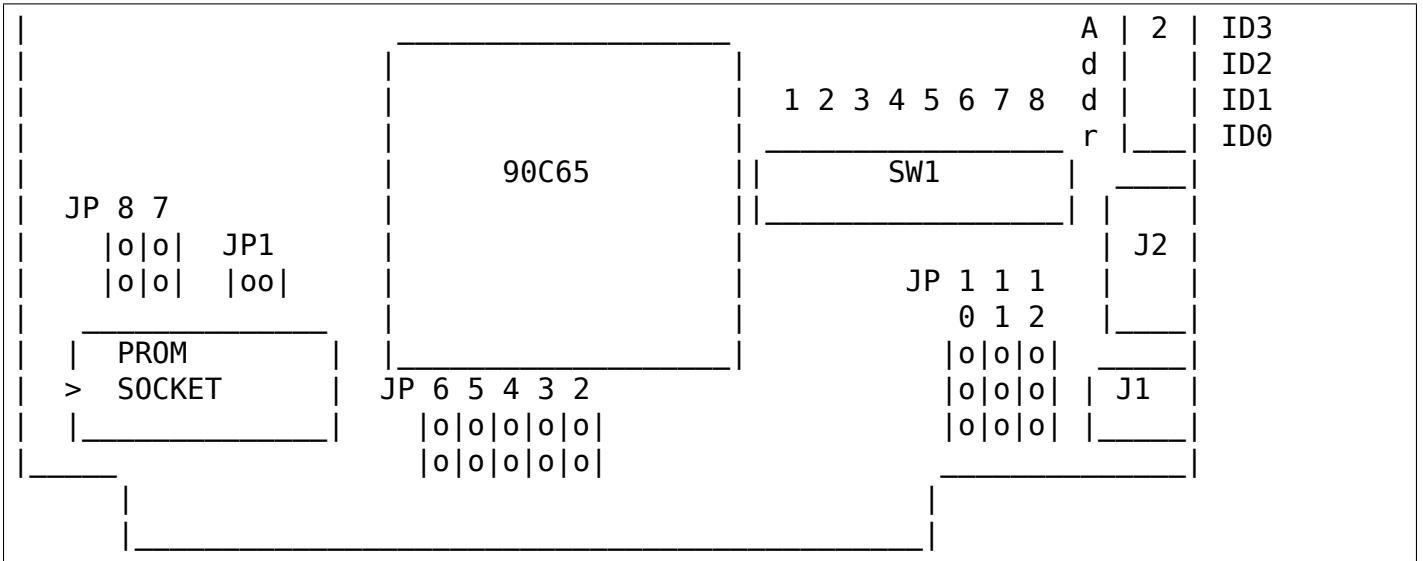
This description has been written by Juergen Seifert <seifert@htwm.de> using information from the following Original CNet Manual

"ARCNET USER'S MANUAL for CN120A CN120AB CN120TP CN120ST CN120SBT
P/N:12-01-0007 Revision 3.00"

ARCNET is a registered trademark of the Datapoint Corporation

- P/N 120A ARCNET 8 bit XT/AT Star
- P/N 120AB ARCNET 8 bit XT/AT Bus
- P/N 120TP ARCNET 8 bit XT/AT Twisted Pair
- P/N 120ST ARCNET 8 bit XT/AT Star, Twisted Pair
- P/N 120SBT ARCNET 8 bit XT/AT Star, Bus, Twisted Pair





Legend:

90C65	ARCNET Probe
S1 1-5:	Base Memory Address Select
6-8:	Base I/O Address Select
S2 1-8:	Node ID Select (ID0-ID7)
JP1	ROM Enable Select
JP2	IRQ2
JP3	IRQ3
JP4	IRQ4
JP5	IRQ5
JP6	IRQ7
JP7/JP8	ET1, ET2 Timeout Parameters
JP10/JP11	Coax / Twisted Pair Select (CN120ST/SBT only)
JP12	Terminator Select (CN120AB/ST/SBT only)
J1	BNC RG62/U Connector (all except CN120TP)
J2	Two 6-position Telephone Jack (CN120TP/ST/SBT only)

Setting one of the switches to Off means "1", On means "0".

Setting the Node ID

The eight switches in SW2 are used to set the node ID. Each node attached to the network must have an unique node ID which must be different from 0. Switch 1 (ID0) serves as the least significant bit (LSB).

The node ID is the sum of the values of all switches set to "1" These values are:

Switch	Label	Value
1	ID0	1
2	ID1	2
3	ID2	4
4	ID3	8
5	ID4	16
6	ID5	32
7	ID6	64
8	ID7	128

Some Examples:

Switch 8 7 6 5 4 3 2 1	Hex Node ID	Decimal Node ID
0 0 0 0 0 0 0 0		not allowed
0 0 0 0 0 0 0 1	1	1
0 0 0 0 0 0 1 0	2	2
0 0 0 0 0 0 1 1	3	3
.	.	.
0 1 0 1 0 1 0 1	55	85
.	.	.
1 0 1 0 1 0 1 0	AA	170
.	.	.
1 1 1 1 1 1 0 1	FD	253
1 1 1 1 1 1 1 0	FE	254
1 1 1 1 1 1 1 1	FF	255

Setting the I/O Base Address

The last three switches in switch block SW1 are used to select one of eight possible I/O Base addresses using the following table:

Switch 6 7 8	Hex I/O Address
ON ON ON	260
OFF ON ON	290
ON OFF ON	2E0 (Manufacturer's default)
OFF OFF ON	2F0
ON ON OFF	300
OFF ON OFF	350
ON OFF OFF	380
OFF OFF OFF	3E0

Setting the Base Memory (RAM) buffer Address

The memory buffer (RAM) requires 2K. The base of this buffer can be located in any of eight positions. The address of the Boot Prom is memory base + 8K or memory base + 0x2000. Switches 1-5 of switch block SW1 select the Memory Base address.

Switch					Hex RAM	Hex ROM
1	2	3	4	5	Address	Address *)
ON	ON	ON	ON	ON	C0000	C2000
ON	ON	OFF	ON	ON	C4000	C6000
ON	ON	ON	OFF	ON	CC000	CE000
ON	ON	OFF	OFF	ON	D0000	D2000 (Manufacturer's default)
ON	ON	ON	ON	OFF	D4000	D6000
ON	ON	OFF	ON	OFF	D8000	DA000
ON	ON	ON	OFF	OFF	DC000	DE000
ON	ON	OFF	OFF	OFF	E0000	E2000

*) To enable the Boot ROM install the jumper JP1

Note: Since the switches 1 and 2 are always set to ON it may be possible that they can be used to add an offset of 2K, 4K or 6K to the base address, but this feature is not documented in the manual and I haven't tested it yet.

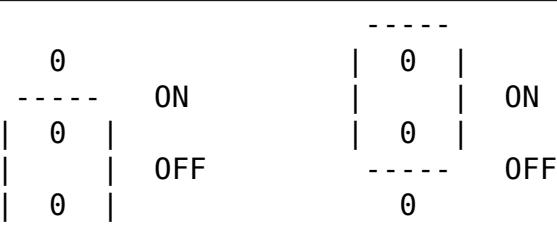
Setting the Interrupt Line

To select a hardware interrupt level install one (only one!) of the jumpers JP2, JP3, JP4, JP5, JP6. JP2 is the default:

Jumper	IRQ
2	2
3	3
4	4
5	5
6	7

Setting the Internal Terminator on CN120AB/TP/SBT

The jumper JP12 is used to enable the internal terminator:



Terminator disabled	Terminator enabled
------------------------	-----------------------

Selecting the Connector Type on CN120ST/SBT

JP10	JP11	JP10	JP11
0	0	0	0
0	0	0	0
0	0	0	0
Coaxial Cable (Default)		Twisted Pair Cable	

Setting the Timeout Parameters

The jumpers labeled EXT1 and EXT2 are used to determine the timeout parameters. These two jumpers are normally left open.

33.9 CNet Technology Inc. (16-bit cards)

33.9.1 160 Series (16-bit cards)

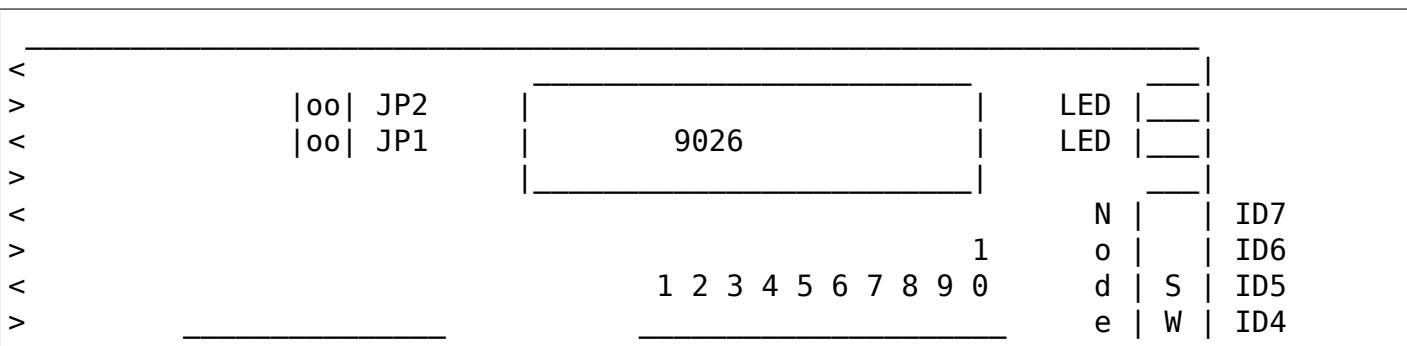
- from Juergen Seifert <seifert@htwm.de>

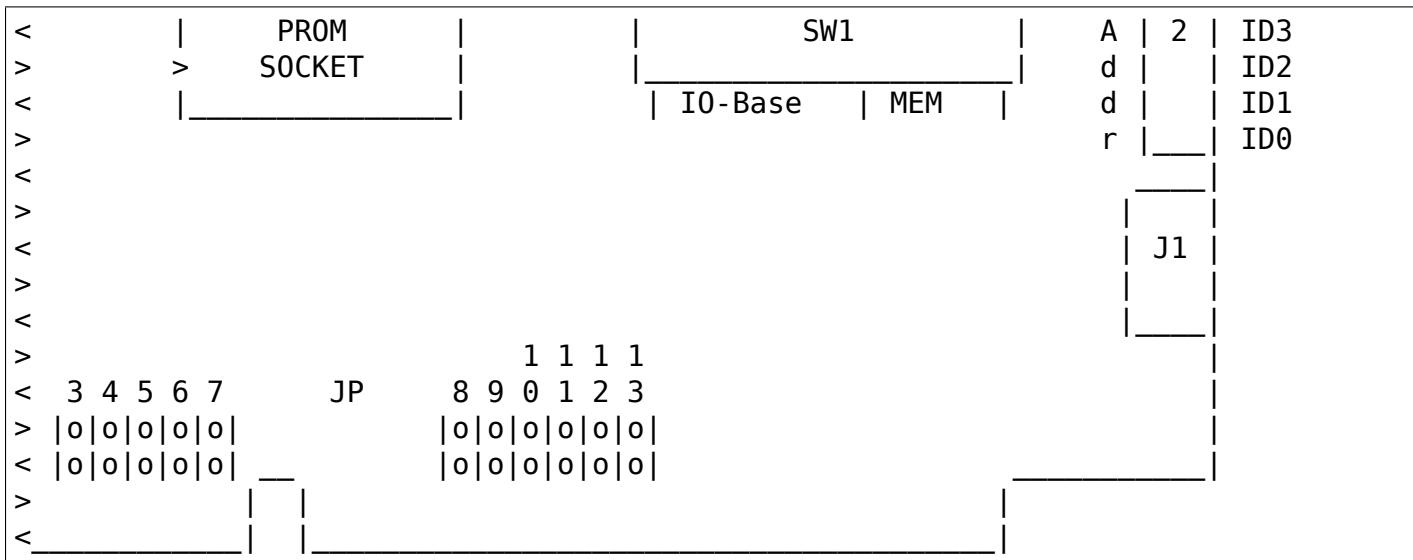
This description has been written by Juergen Seifert <seifert@htwm.de> using information from the following Original CNet Manual

"ARCNET USER'S MANUAL for CN160A CN160AB CN160TP P/N:12-01-0006 Revision 3.00"

ARCNET is a registered trademark of the Datapoint Corporation

- P/N 160A ARCNET 16 bit XT/AT Star
- P/N 160AB ARCNET 16 bit XT/AT Bus
- P/N 160TP ARCNET 16 bit XT/AT Twisted Pair





Legend:

9026	ARCNET Probe	
SW1 1-6:	Base I/O Address Select	
7-10:	Base Memory Address Select	
SW2 1-8:	Node ID Select (ID0-ID7)	
JP1/JP2	ET1, ET2 Timeout Parameters	
JP3-JP13	Interrupt Select	
J1	BNC RG62/U Connector	(CN160A/AB only)
J1	Two 6-position Telephone Jack	(CN160TP only)
LED		

Setting one of the switches to Off means "1", On means "0".

Setting the Node ID

The eight switches in SW2 are used to set the node ID. Each node attached to the network must have an unique node ID which must be different from 0. Switch 1 (ID0) serves as the least significant bit (LSB).

The node ID is the sum of the values of all switches set to "1" These values are:

Switch	Label	Value
1	ID0	1
2	ID1	2
3	ID2	4
4	ID3	8
5	ID4	16
6	ID5	32
7	ID6	64
8	ID7	128

Some Examples:

Switch 8 7 6 5 4 3 2 1	Hex Node ID	Decimal Node ID
0 0 0 0 0 0 0 0	not allowed	
0 0 0 0 0 0 0 1	1	1
0 0 0 0 0 0 1 0	2	2
0 0 0 0 0 0 1 1	3	3
.	.	.
0 1 0 1 0 1 0 1	55	85
.	.	.
1 0 1 0 1 0 1 0	AA	170
.	.	.
1 1 1 1 1 1 0 1	FD	253
1 1 1 1 1 1 1 0	FE	254
1 1 1 1 1 1 1 1	FF	255

Setting the I/O Base Address

The first six switches in switch block SW1 are used to select the I/O Base address using the following table:

Switch 1 2 3 4 5 6	Hex I/O Address
OFF ON ON OFF OFF ON	260
OFF ON OFF ON ON OFF	290
OFF ON OFF OFF OFF ON	2E0 (Manufacturer's default)
OFF ON OFF OFF OFF OFF	2F0
OFF OFF ON ON ON ON	300
OFF OFF ON OFF ON OFF	350
OFF OFF OFF ON ON ON	380
OFF OFF OFF OFF OFF ON	3E0

Note: Other IO-Base addresses seem to be selectable, but only the above combinations are documented.

Setting the Base Memory (RAM) buffer Address

The switches 7-10 of switch block SW1 are used to select the Memory Base address of the RAM (2K) and the PROM:

Switch 7 8 9 10	Hex RAM Address	Hex ROM Address
OFF OFF ON ON	C0000	C8000
OFF OFF ON OFF	D0000	D8000 (Default)
OFF OFF OFF ON	E0000	E8000

Note: Other MEM-Base addresses seem to be selectable, but only the above combinations are documented.

Setting the Interrupt Line

To select a hardware interrupt level install one (only one!) of the jumpers JP3 through JP13 using the following table:

Jumper	IRQ
3	14
4	15
5	12
6	11
7	10
8	3
9	4
10	5
11	6
12	7
13	2 (=9) Default!

Note:

- Do not use JP11=IRQ6, it may conflict with your Floppy Disk Controller
 - Use JP3=IRQ14 only, if you don't have an IDE-, MFM-, or RLL- Hard Disk, it may conflict with their controllers
-

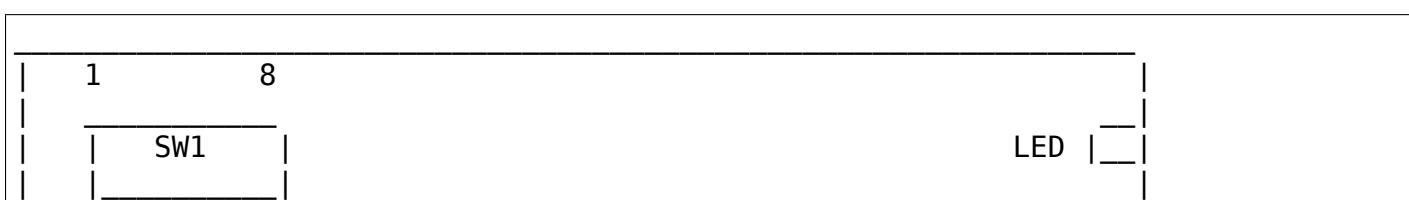
33.9.2 Setting the Timeout Parameters

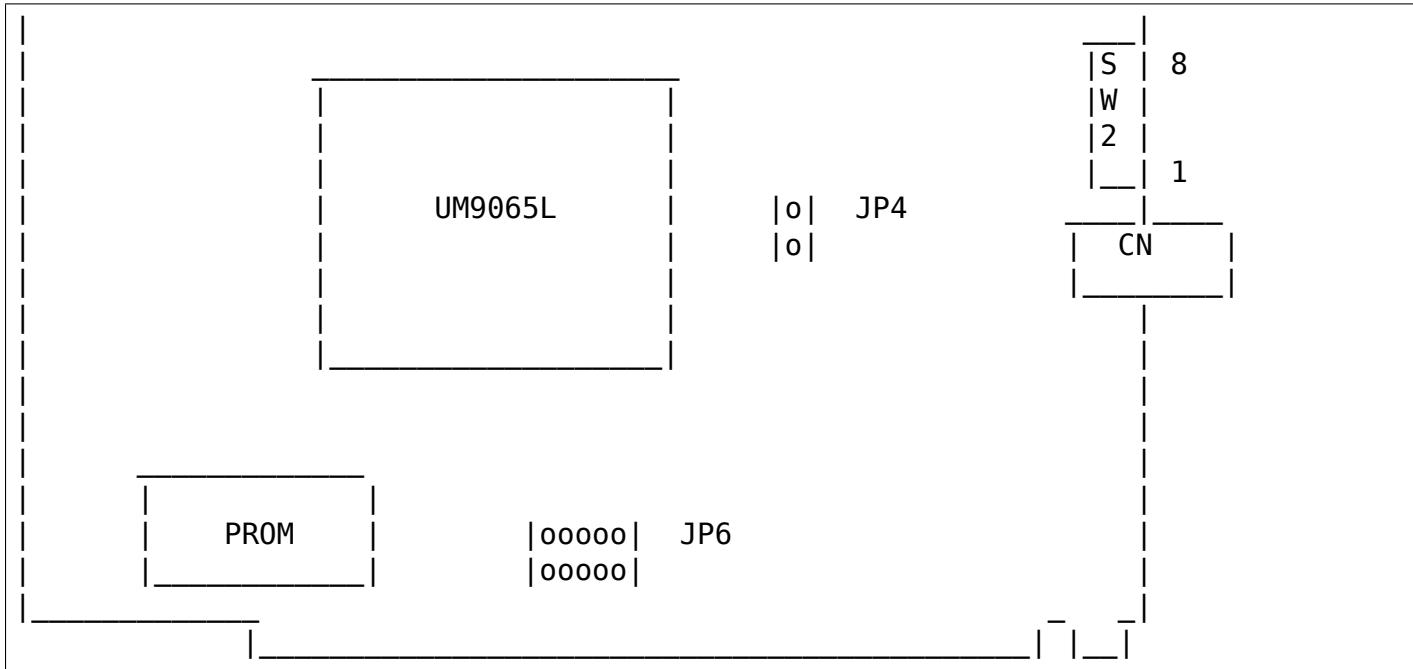
The jumpers labeled JP1 and JP2 are used to determine the timeout parameters. These two jumpers are normally left open.

33.10 Lantech

33.10.1 8-bit card, unknown model

- from Vlad Lungu <vlungu@ugal.ro> - his e-mail address seemed broken at the time I tried to reach him. Sorry Vlad, if you didn't get my reply.





UM9065L : ARCnet Controller

SW 1 : Shared Memory Address and I/O Base

ON=0

12345	Memory Address

00001	D4000
00010	CC000
00110	D0000
01110	D1000
01101	D9000
10010	CC800
10011	DC800
11110	D1800

It seems that the bits are considered in reverse order. Also, you must observe that some of those addresses are unusual and I didn't probe them; I used a memory dump in DOS to identify them. For the 00000 configuration and some others that I didn't write here the card seems to conflict with the video card (an S3 GENDAC). I leave the full decoding of those addresses to you.

678	I/O Address

000	260
001	failed probe
010	2E0
011	380
100	290
101	350
110	failed probe

```

111| 3E0

SW 2 : Node ID (binary coded)

JP 4 : Boot PROM enable  CLOSE - enabled
          OPEN - disabled

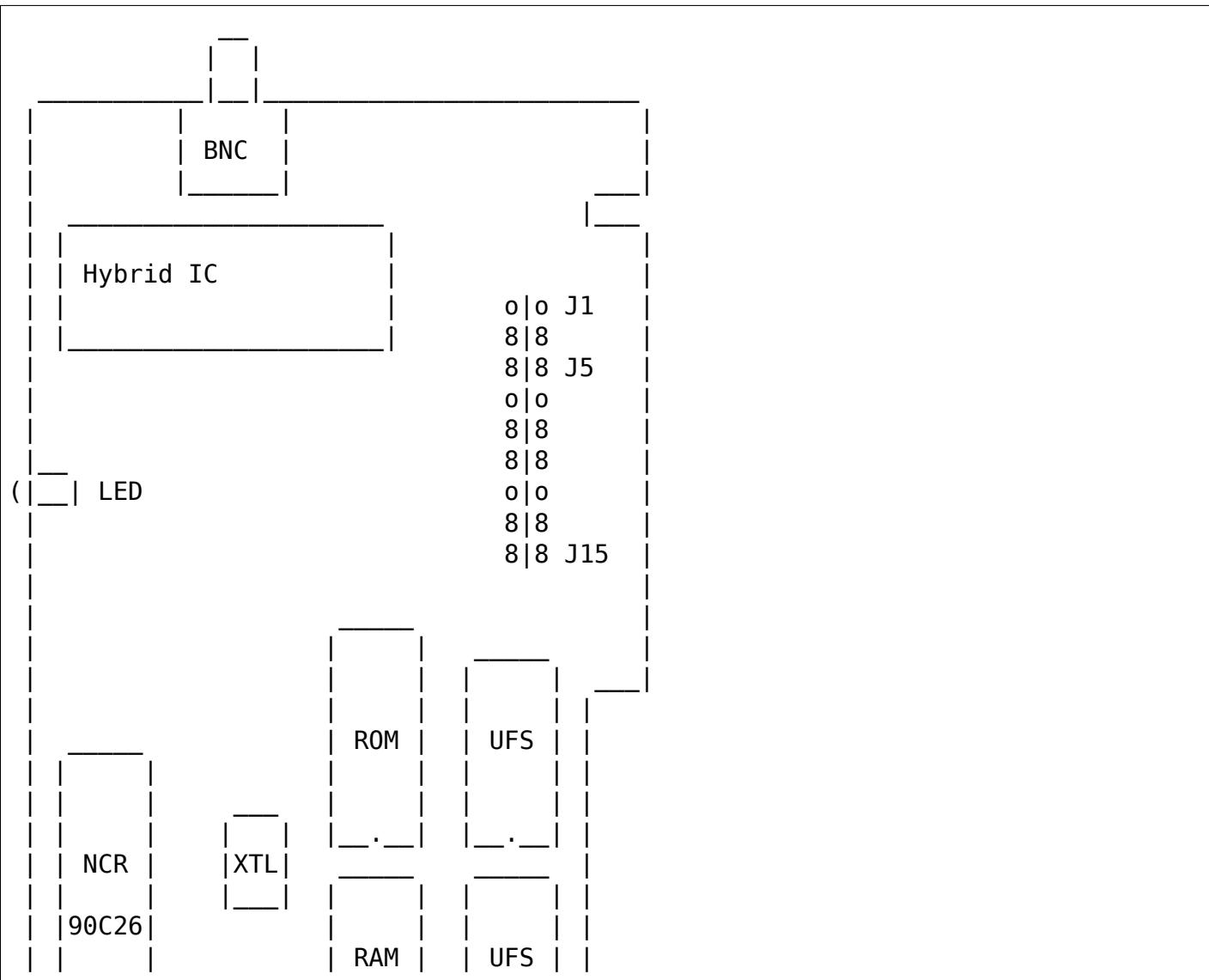
JP 6 : IRQ set (ONLY ONE jumper on 1-5 for IRQ 2-6)

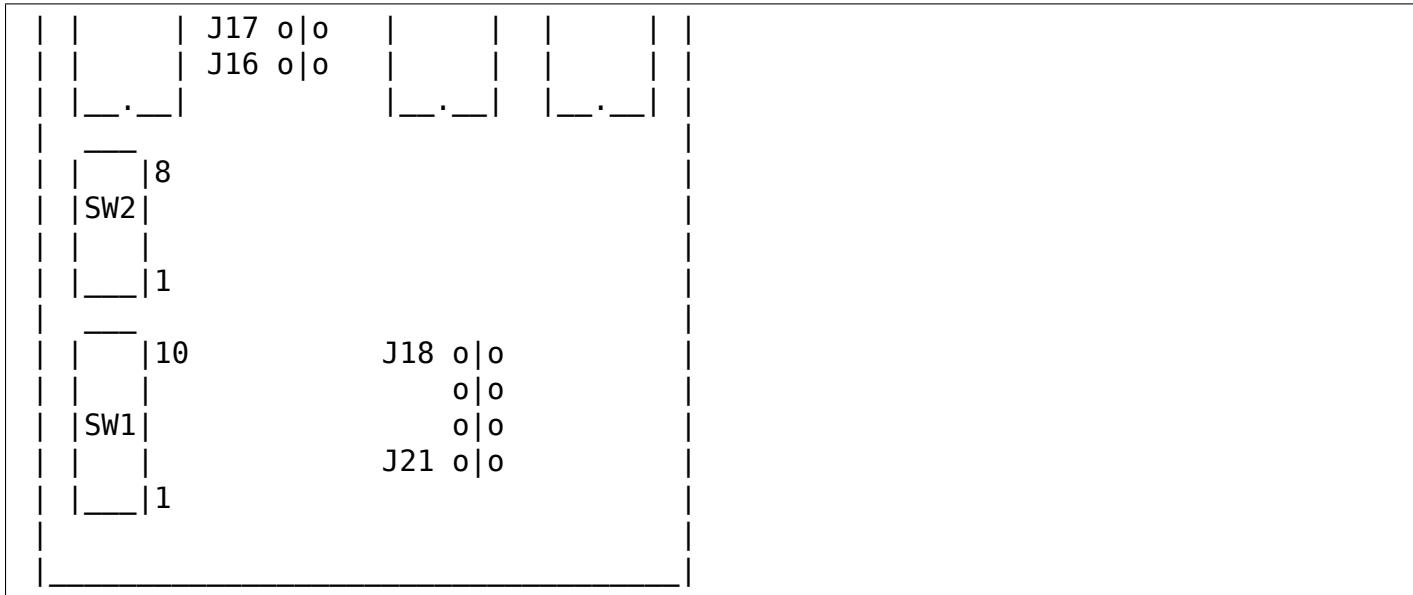
```

33.11 Acer

33.11.1 8-bit card, Model 5210-003

- from Vojtech Pavlik <vojtech@suse.cz> using portions of the existing arcnet-hardware file.
- This is a 90C26 based card. Its configuration seems similar to the SMC PC100, but has some additional jumpers I don't know the meaning of.





Legend:

90C26	ARCNET Chip
XTL	20 MHz Crystal
SW1 1-6	Base I/O Address Select
7-10	Memory Address Select
SW2 1-8	Node ID Select (ID0-ID7)
J1-J5	IRQ Select
J6-J21	Unknown (Probably extra timeouts & ROM enable ...)
LED1	Activity LED
BNC	Coax connector (STAR ARCnet)
RAM	2k of SRAM
ROM	Boot ROM socket
UFS	Unidentified Flying Sockets

Setting the Node ID

The eight switches in SW2 are used to set the node ID. Each node attached to the network must have an unique node ID which must not be 0. Switch 1 (ID0) serves as the least significant bit (LSB).

Setting one of the switches to OFF means "1", ON means "0".

The node ID is the sum of the values of all switches set to "1" These values are:

Switch	Value
1	1
2	2
3	4
4	8
5	16
6	32

7		64
8		128

Don't set this to 0 or 255; these values are reserved.

Setting the I/O Base Address

The switches 1 to 6 of switch block SW1 are used to select one of 32 possible I/O Base addresses using the following tables:

Switch	Hex Value
1	200
2	100
3	80
4	40
5	20
6	10

The I/O address is sum of all switches set to "1". Remember that the I/O address space below 0x200 is RESERVED for mainboard, so switch 1 should be ALWAYS SET TO OFF.

Setting the Base Memory (RAM) buffer Address

The memory buffer (RAM) requires 2K. The base of this buffer can be located in any of sixteen positions. However, the addresses below A0000 are likely to cause system hang because there's main RAM.

Jumpers 7-10 of switch block SW1 select the Memory Base address:

Switch 7 8 9 10	Hex RAM Address
OFF OFF OFF OFF	F0000 (conflicts with main BIOS)
OFF OFF OFF ON	E0000
OFF OFF ON OFF	D0000
OFF OFF ON ON	C0000 (conflicts with video BIOS)
OFF ON OFF OFF	B0000 (conflicts with mono video)
OFF ON OFF ON	A0000 (conflicts with graphics)

Setting the Interrupt Line

Jumpers 1-5 of the jumper block J1 control the IRQ level. ON means shorted, OFF means open:

Jumper					IRQ
1	2	3	4	5	
ON	OFF	OFF	OFF	OFF	7
OFF	ON	OFF	OFF	OFF	5
OFF	OFF	ON	OFF	OFF	4
OFF	OFF	OFF	ON	OFF	3
OFF	OFF	OFF	OFF	ON	2

Unknown jumpers & sockets

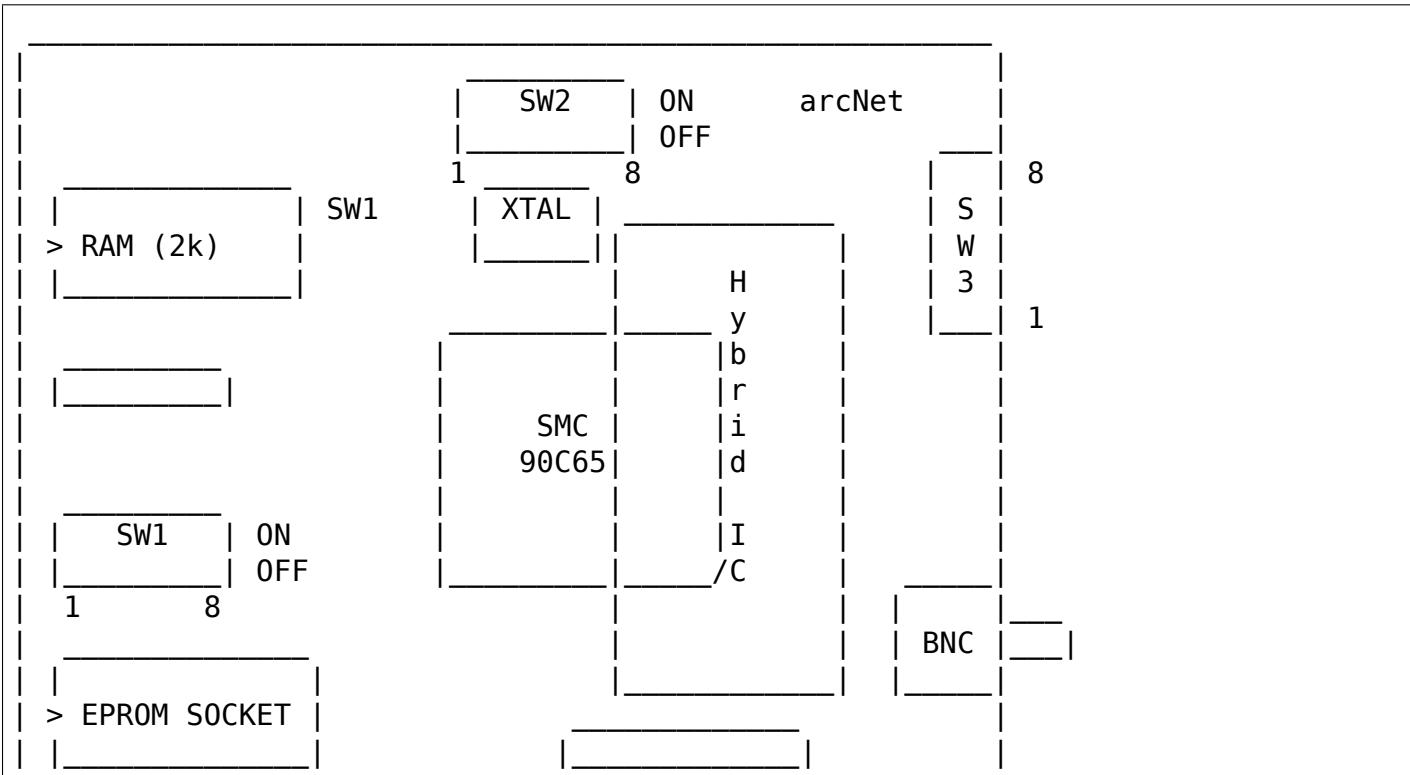
I know nothing about these. I just guess that J16&J17 are timeout jumpers and maybe one of J18-J21 selects ROM. Also J6-J10 and J11-J15 are connecting IRQ2-7 to some pins on the UFSs. I can't guess the purpose.

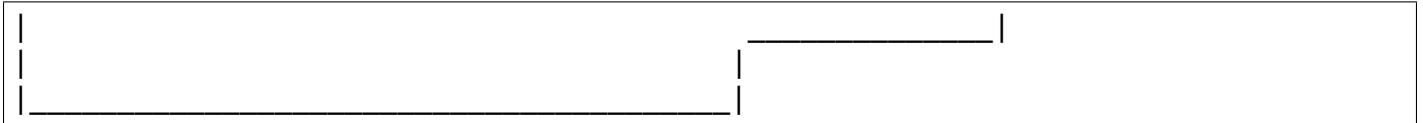
33.12 Datapoint?

33.12.1 LAN-ARC-8, an 8-bit card

- from Vojtech Pavlik <vojtech@suse.cz>

This is another SMC 90C65-based ARCnet card. I couldn't identify the manufacturer, but it might be DataPoint, because the card has the original arcNet logo in its upper right corner.





Legend:

90C65	ARCNET Chip
SW1 1-5:	Base Memory Address Select
6-8:	Base I/O Address Select
SW2 1-8:	Node ID Select
SW3 1-5:	IRQ Select
6-7:	Extra Timeout
8 :	ROM Enable
BNC	Coax connector
XTAL	20 MHz Crystal

Setting the Node ID

The eight switches in SW3 are used to set the node ID. Each node attached to the network must have an unique node ID which must not be 0. Switch 1 serves as the least significant bit (LSB).

Setting one of the switches to Off means "1", On means "0".

The node ID is the sum of the values of all switches set to "1" These values are:

Switch	Value
1	1
2	2
3	4
4	8
5	16
6	32
7	64
8	128

Setting the I/O Base Address

The last three switches in switch block SW1 are used to select one of eight possible I/O Base addresses using the following table:

Switch 6 7 8	Hex I/O Address
ON ON ON	260
OFF ON ON	290
ON OFF ON	2E0 (Manufacturer's default)
OFF OFF ON	2F0
ON ON OFF	300
OFF ON OFF	350

ON	OFF	OFF	OFF		380
OFF	OFF	OFF	OFF		3E0

Setting the Base Memory (RAM) buffer Address

The memory buffer (RAM) requires 2K. The base of this buffer can be located in any of eight positions. The address of the Boot Prom is memory base + 0x2000.

Jumpers 3-5 of switch block SW1 select the Memory Base address.

Switch					Hex RAM	Hex ROM
1	2	3	4	5	Address	Address *)
ON	ON	ON	ON	ON	C0000	C2000
ON	ON	OFF	ON	ON	C4000	C6000
ON	ON	ON	OFF	ON	CC000	CE000
ON	ON	OFF	OFF	ON	D0000	D2000 (Manufacturer's default)
ON	ON	ON	ON	OFF	D4000	D6000
ON	ON	OFF	ON	OFF	D8000	DA000
ON	ON	ON	OFF	OFF	DC000	DE000
ON	ON	OFF	OFF	OFF	E0000	E2000

*) To enable the Boot ROM set the switch 8 of switch block SW3 to position ON.

The switches 1 and 2 probably add 0x0800 and 0x1000 to RAM base address.

Setting the Interrupt Line

Switches 1-5 of the switch block SW3 control the IRQ level:

Jumper					IRQ
1	2	3	4	5	
ON	OFF	OFF	OFF	OFF	3
OFF	ON	OFF	OFF	OFF	4
OFF	OFF	ON	OFF	OFF	5
OFF	OFF	OFF	ON	OFF	7
OFF	OFF	OFF	OFF	ON	2

Setting the Timeout Parameters

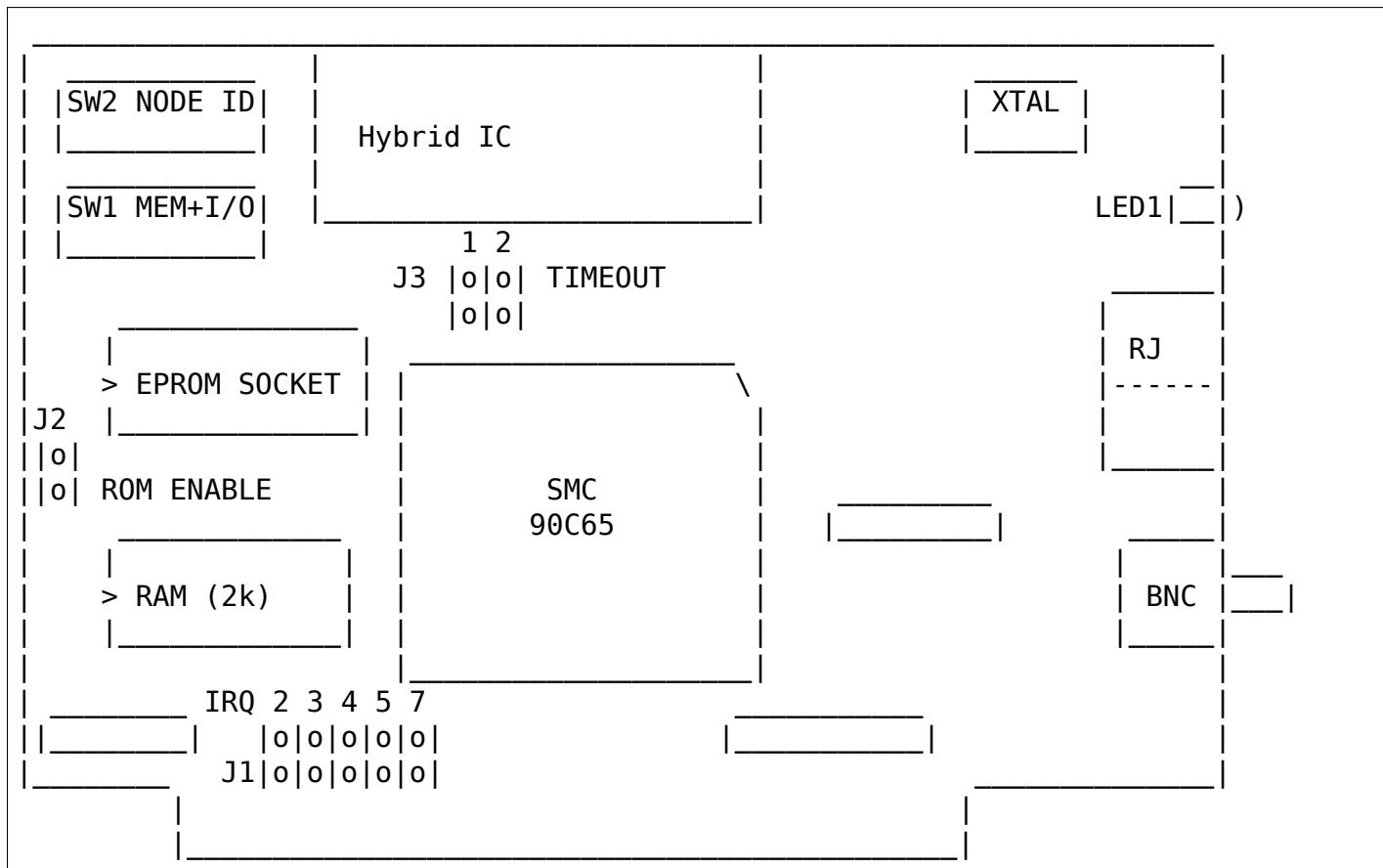
The switches 6-7 of the switch block SW3 are used to determine the timeout parameters. These two switches are normally left in the OFF position.

33.13 Topware

33.13.1 8-bit card, TA-ARC/10

- from Vojtech Pavlik <vojtech@suse.cz>

This is another very similar 90C65 card. Most of the switches and jumpers are the same as on other clones.



Legend:

90C65	ARCNET Chip
XTAL	20 MHz Crystal
SW1 1-5	Base Memory Address Select
6-8	Base I/O Address Select
SW2 1-8	Node ID Select (ID0-ID7)
J1	IRQ Select
J2	ROM Enable
J3	Extra Timeout
LED1	Activity LED
BNC	Coax connector (BUS ARCnet)
RJ	Twisted Pair Connector (daisy chain)

Setting the Node ID

The eight switches in SW2 are used to set the node ID. Each node attached to the network must have an unique node ID which must not be 0. Switch 1 (ID0) serves as the least significant bit (LSB).

Setting one of the switches to Off means "1", On means "0".

The node ID is the sum of the values of all switches set to "1" These values are:

Switch	Label	Value
1	ID0	1
2	ID1	2
3	ID2	4
4	ID3	8
5	ID4	16
6	ID5	32
7	ID6	64
8	ID7	128

Setting the I/O Base Address

The last three switches in switch block SW1 are used to select one of eight possible I/O Base addresses using the following table:

Switch 6 7 8	Hex I/O Address
ON ON ON	260 (Manufacturer's default)
OFF ON ON	290
ON OFF ON	2E0
OFF OFF ON	2F0
ON ON OFF	300
OFF ON OFF	350
ON OFF OFF	380
OFF OFF OFF	3E0

Setting the Base Memory (RAM) buffer Address

The memory buffer (RAM) requires 2K. The base of this buffer can be located in any of eight positions. The address of the Boot Prom is memory base + 0x2000.

Jumpers 3-5 of switch block SW1 select the Memory Base address.

Switch 1 2 3 4 5	Hex RAM Address	Hex ROM Address *)
ON ON ON ON ON	C0000	C2000
ON ON OFF ON ON	C4000	C6000 (Manufacturer's default)
ON ON ON OFF ON	CC000	CE000

ON	ON	OFF	OFF	ON		D0000		D2000
ON	ON	ON	ON	OFF		D4000		D6000
ON	ON	OFF	ON	OFF		D8000		DA000
ON	ON	ON	OFF	OFF		DC000		DE000
ON	ON	OFF	OFF	OFF		E0000		E2000

*) To enable the Boot ROM short the jumper J2.

The jumpers 1 and 2 probably add 0x0800 and 0x1000 to RAM address.

Setting the Interrupt Line

Jumpers 1-5 of the jumper block J1 control the IRQ level. ON means shorted, OFF means open:

Jumper					IRQ
1	2	3	4	5	
<hr/>					
ON	OFF	OFF	OFF	OFF	2
OFF	ON	OFF	OFF	OFF	3
OFF	OFF	ON	OFF	OFF	4
OFF	OFF	OFF	ON	OFF	5
OFF	OFF	OFF	OFF	ON	7

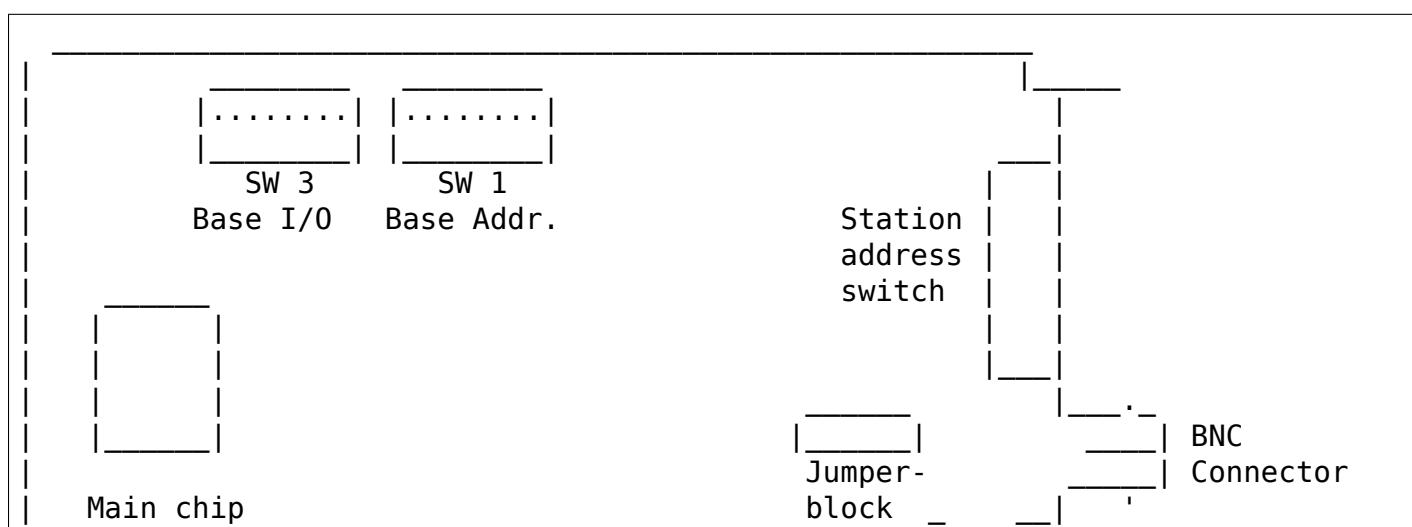
Setting the Timeout Parameters

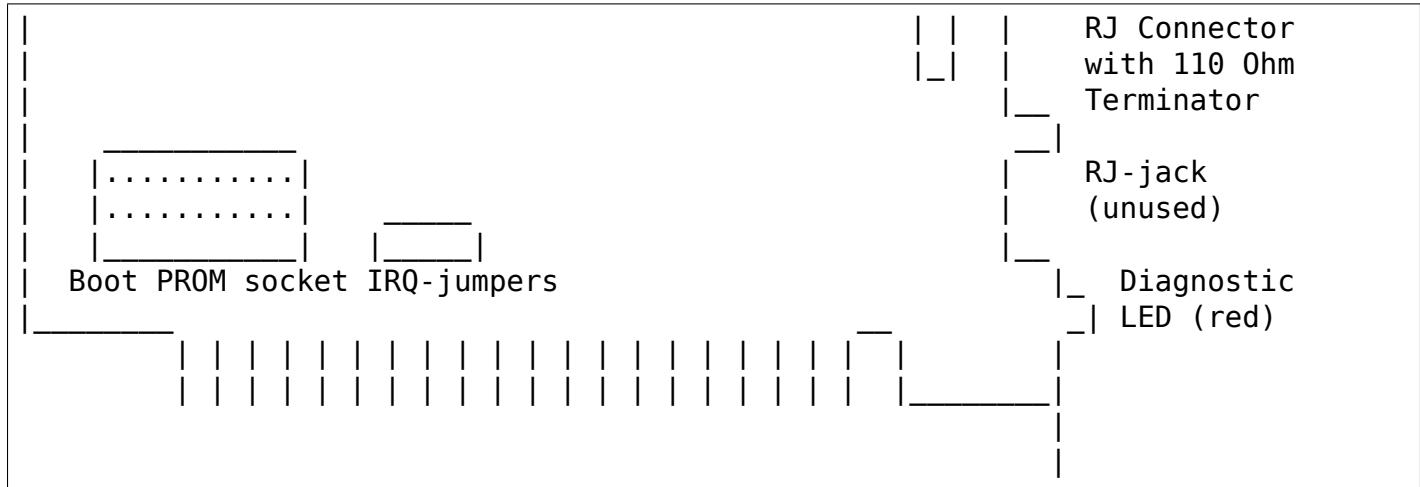
The jumpers J3 are used to set the timeout parameters. These two jumpers are normally left open.

33.14 Thomas-Conrad

33.14.1 Model #500-6242-0097 REV A (8-bit card)

- from Lars Karlsson <100617.3473@compuserve.com>





And here are the settings for some of the switches and jumpers on the cards.

I/O

1 2 3 4 5 6 7 8

2E0-----	0	0	0	1	0	0	0	1
2F0-----	0	0	0	1	0	0	0	0
300-----	0	0	0	0	1	1	1	1
350-----	0	0	0	0	1	1	1	0

"0" in the above example means switch is off "1" means that it is on.

ShMem address .

1 2 3 4 5 6 7 8

CX00--	0	0	1	1				
DX00--	0	0	1	0				
X000-----			1	1				
X400-----			1	0				
X800-----			0	1				
XC00-----			0	0				
ENHANCED-----					1			
COMPATIBLE-----					0			

IRQ

3 4 5 7 2

There is a DIP-switch with 8 switches, used to set the shared memory address to be used. The first 6 switches set the address, the 7th doesn't have any function, and the 8th switch is used to select "compatible" or "enhanced". When I got my two cards, one of them had this switch set to "enhanced". That card didn't work at all, it wasn't even recognized by the driver. The other

card had this switch set to "compatible" and it behaved absolutely normally. I guess that the switch on one of the cards, must have been changed accidentally when the card was taken out of its former host. The question remains unanswered, what is the purpose of the "enhanced" position?

[Avery's note: "enhanced" probably either disables shared memory (use IO ports instead) or disables IO ports (use memory addresses instead). This varies by the type of card involved. I fail to see how either of these enhance anything. Send me more detailed information about this mode, or just use "compatible" mode instead.]

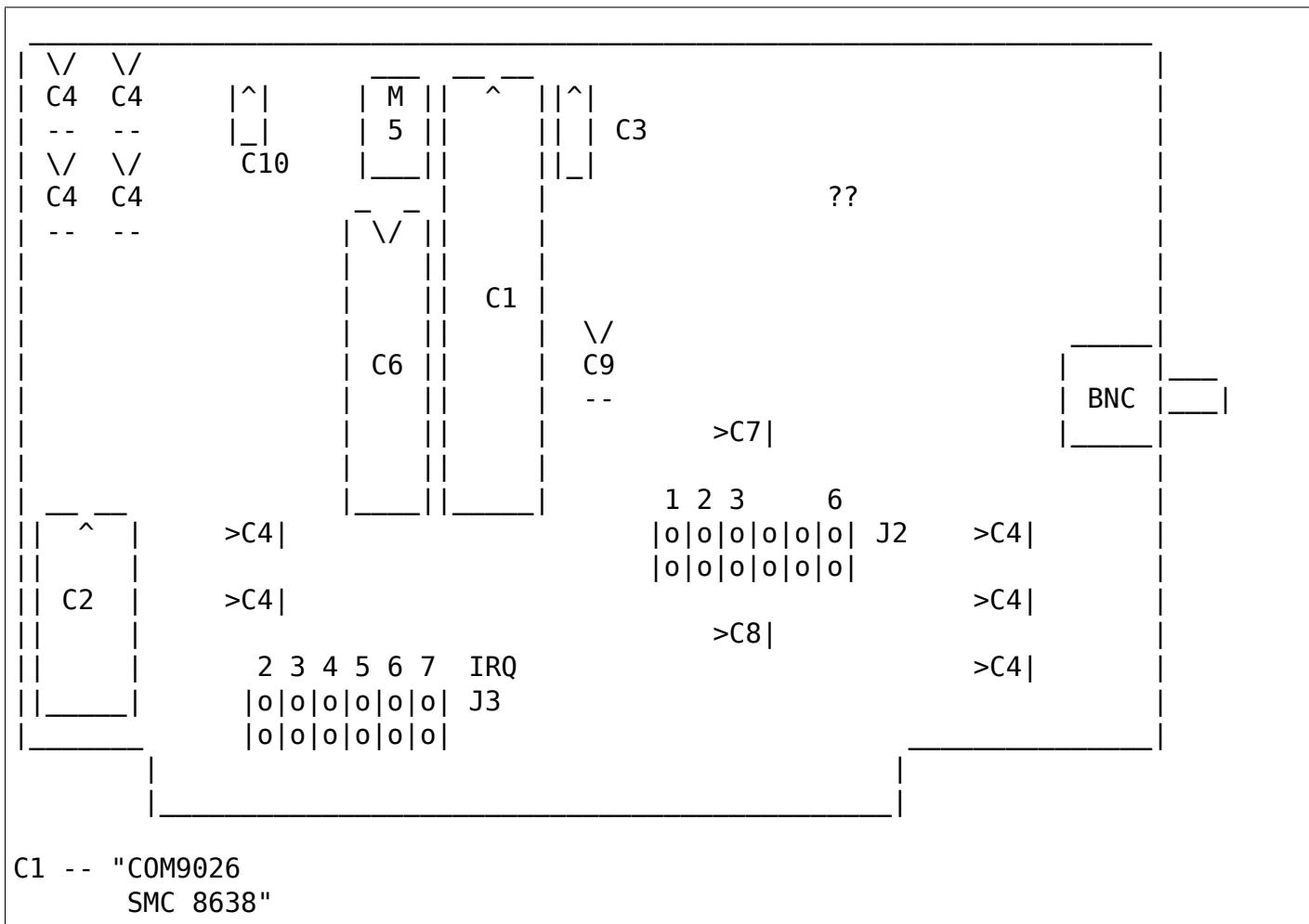
33.15 Waterloo Microsystems Inc. ??

33.15.1 8-bit card (C) 1985

- from Robert Michael Best <rmb117@cs.usask.ca>

[Avery's note: these don't work with my driver for some reason. These cards SEEM to have settings similar to the PDI508Plus, which is software-configured and doesn't work with my driver either. The "Waterloo chip" is a boot PROM, probably designed specifically for the University of Waterloo. If you have any further information about this card, please e-mail me.]

The probe has not been able to detect the card on any of the J2 settings, and I tried them again with the "Waterloo" chip removed.



In a chip socket.

C2 -- "@Copyright
Waterloo Microsystems Inc.
1985"
In a chip Socket with info printed on a label covering a round window showing the circuit inside. (The window indicates it is an EPROM chip.)

C3 -- "COM9032
SMC 8643"
In a chip socket.

C4 -- "74LS"
9 total no sockets.

M5 -- "50006-136
20.000000 MHZ
MTQ-T1-S3
0 M-TRON 86-40"
Metallic case with 4 pins, no socket.

C6 -- "MOSTEK@TC8643
MK6116N-20
MALAYSIA"
No socket.

C7 -- No stamp or label but in a 20 pin chip socket.

C8 -- "PAL10L8CN
8623"
In a 20 pin socket.

C9 -- "PAL16R4A-2CN
8641"
In a 20 pin socket.

C10 -- "M8640
NMC
9306N"
In an 8 pin socket.

?? -- Some components on a smaller board and attached with 20 pins all along the side closest to the BNC connector. The are coated in a dark resin.

On the board there are two jumper banks labeled J2 and J3. The manufacturer didn't put a J1 on the board. The two boards I have both came with a jumper box for each bank.

J2 -- Numbered 1 2 3 4 5 6.
4 and 5 are not stamped due to solder points.

J3	--	IRQ	2	3	4	5	6	7
----	----	-----	---	---	---	---	---	---

The board itself has a maple leaf stamped just above the irq jumpers and "-2 46-86" beside C2. Between C1 and C6 "ASS 'Y 300163" and "@1986 CORMAN CUSTOM ELECTRONICS CORP." stamped just below the BNC connector. Below that "MADE IN CANADA"

33.16 No Name

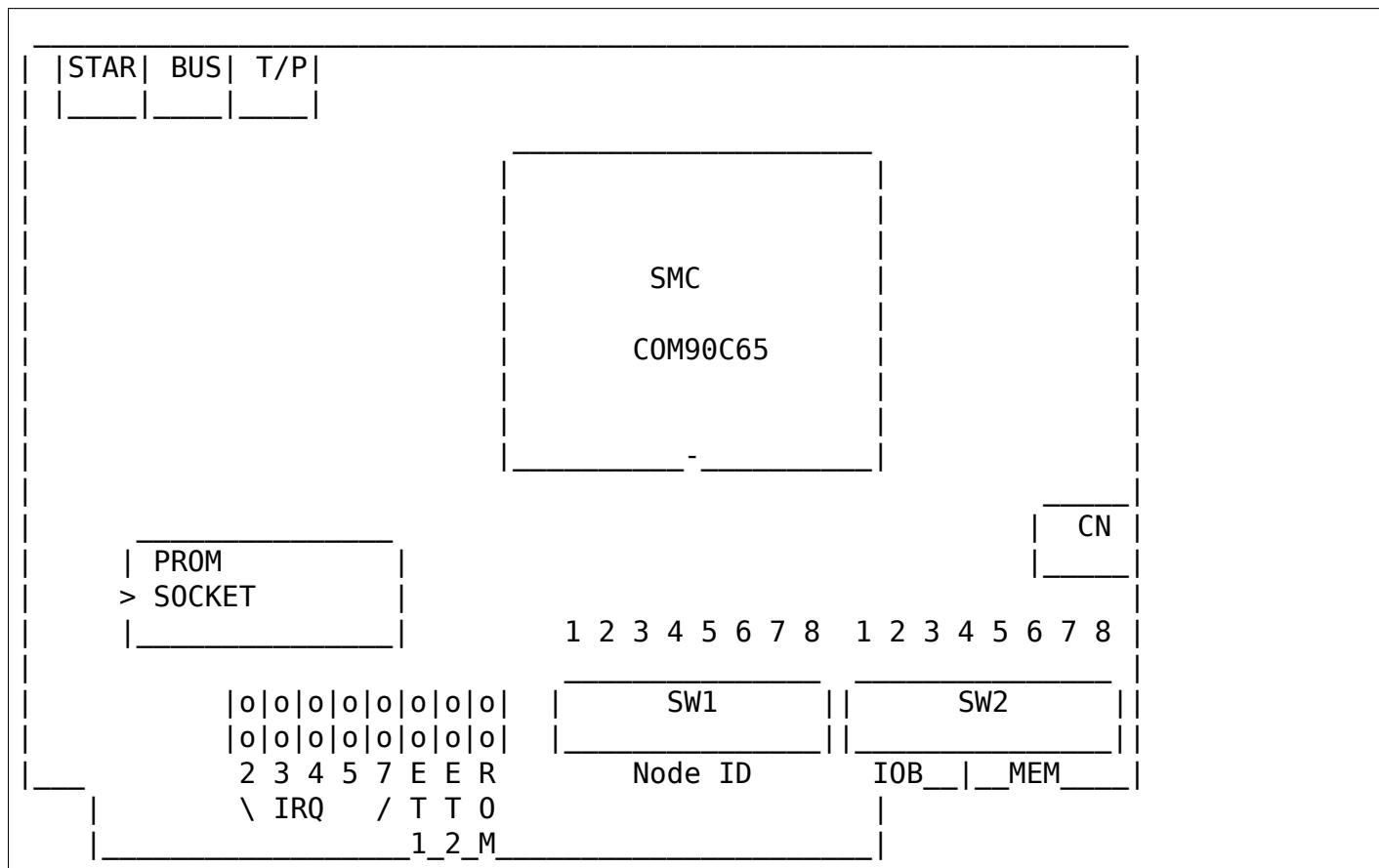
33.16.1 8-bit cards, 16-bit cards

- from Juergen Seifert <seifert@htwm.de>

I have named this ARCnet card "NONAME", since there is no name of any manufacturer on the Installation manual nor on the shipping box. The only hint to the existence of a manufacturer at all is written in copper, it is "Made in Taiwan"

This description has been written by Juergen Seifert <seifert@htwm.de> using information from the Original

"ARCnet Installation Manual"



Legend:

COM90C65:	ARCnet Probe
S1 1-8:	Node ID Select
S2 1-3:	I/O Base Address Select

4-6:	Memory Base Address Select
7-8:	RAM Offset Select
ET1, ET2	Extended Timeout Select
ROM	ROM Enable Select
CN	RG62 Coax Connector
STAR BUS T/P	Three fields for placing a sign (colored circle) indicating the topology of the card

Setting one of the switches to Off means "1", On means "0".

Setting the Node ID

The eight switches in group SW1 are used to set the node ID. Each node attached to the network must have an unique node ID which must be different from 0. Switch 8 serves as the least significant bit (LSB).

The node ID is the sum of the values of all switches set to "1" These values are:

Switch	Value
8	1
7	2
6	4
5	8
4	16
3	32
2	64
1	128

Some Examples:

Switch 1 2 3 4 5 6 7 8	Hex Node ID	Decimal Node ID
0 0 0 0 0 0 0 0	not allowed	
0 0 0 0 0 0 0 1	1	1
0 0 0 0 0 0 1 0	2	2
0 0 0 0 0 0 1 1	3	3
. . .		
0 1 0 1 0 1 0 1	55	85
. . .		
1 0 1 0 1 0 1 0	AA	170
. . .		
1 1 1 1 1 1 0 1	FD	253
1 1 1 1 1 1 1 0	FE	254
1 1 1 1 1 1 1 1	FF	255

Setting the I/O Base Address

The first three switches in switch group SW2 are used to select one of eight possible I/O Base addresses using the following table:

Switch 1 2 3	Hex I/O Address
ON ON ON	260
ON ON OFF	290
ON OFF ON	2E0 (Manufacturer's default)
ON OFF OFF	2F0
OFF ON ON	300
OFF ON OFF	350
OFF OFF ON	380
OFF OFF OFF	3E0

Setting the Base Memory (RAM) buffer Address

The memory buffer requires 2K of a 16K block of RAM. The base of this 16K block can be located in any of eight positions. Switches 4-6 of switch group SW2 select the Base of the 16K block. Within that 16K address space, the buffer may be assigned any one of four positions, determined by the offset, switches 7 and 8 of group SW2.

Switch 4 5 6 7 8	Hex RAM Address	Hex ROM Address *)
0 0 0 0 0	C0000	C2000
0 0 0 0 1	C0800	C2000
0 0 0 1 0	C1000	C2000
0 0 0 1 1	C1800	C2000
0 0 1 0 0	C4000	C6000
0 0 1 0 1	C4800	C6000
0 0 1 1 0	C5000	C6000
0 0 1 1 1	C5800	C6000
0 1 0 0 0	CC000	CE000
0 1 0 0 1	CC800	CE000
0 1 0 1 0	CD000	CE000
0 1 0 1 1	CD800	CE000
0 1 1 0 0	D0000	D2000 (Manufacturer's default)
0 1 1 0 1	D0800	D2000
0 1 1 1 0	D1000	D2000
0 1 1 1 1	D1800	D2000
1 0 0 0 0	D4000	D6000
1 0 0 0 1	D4800	D6000
1 0 0 1 0	D5000	D6000

1	0	0	1	1	D5800	D6000
1	0	1	0	0	D8000	DA000
1	0	1	0	1	D8800	DA000
1	0	1	1	0	D9000	DA000
1	0	1	1	1	D9800	DA000
1	1	0	0	0	DC000	DE000
1	1	0	0	1	DC800	DE000
1	1	0	1	0	DD000	DE000
1	1	0	1	1	DD800	DE000
1	1	1	0	0	E0000	E2000
1	1	1	0	1	E0800	E2000
1	1	1	1	0	E1000	E2000
1	1	1	1	1	E1800	E2000

*) To enable the 8K Boot PROM install the jumper ROM.
The default is jumper ROM not installed.

Setting Interrupt Request Lines (IRQ)

To select a hardware interrupt level set one (only one!) of the jumpers IRQ2, IRQ3, IRQ4, IRQ5 or IRQ7. The manufacturer's default is IRQ2.

Setting the Timeouts

The two jumpers labeled ET1 and ET2 are used to determine the timeout parameters (response and reconfiguration time). Every node in a network must be set to the same timeout values.

ET1	ET2	Response Time (us)	Reconfiguration Time (ms)
Off	Off	78	840 (Default)
Off	On	285	1680
On	Off	563	1680
On	On	1130	1680

On means jumper installed, Off means jumper not installed

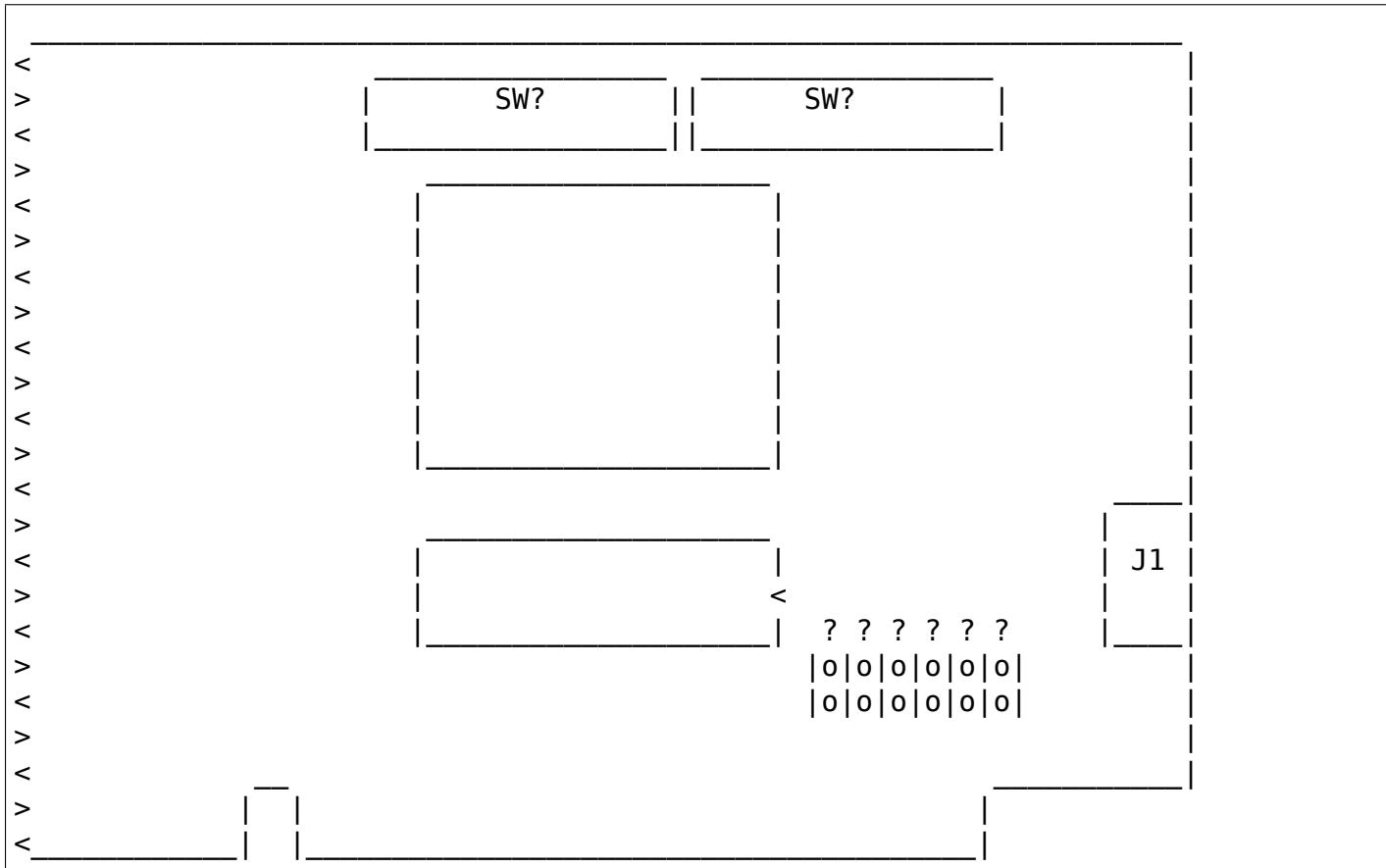
33.16.2 16-BIT ARCNET

The manual of my 8-Bit NONAME ARCnet Card contains another description of a 16-Bit Coax / Twisted Pair Card. This description is incomplete, because there are missing two pages in the manual booklet. (The table of contents reports pages ... 2-9, 2-11, 2-12, 3-1, ... but inside the booklet there is a different way of counting ... 2-9, 2-10, A-1, (empty page), 3-1, ..., 3-18, A-1 (again), A-2) Also the picture of the board layout is not as good as the picture of 8-Bit card, because there isn't any letter like "SW1" written to the picture.

Should somebody have such a board, please feel free to complete this description or to send a mail to me!

This description has been written by Juergen Seifert <seifert@htwm.de> using information from the Original

"ARCnet Installation Manual"



Setting one of the switches to Off means "1", On means "0".

Setting the Node ID

The eight switches in group SW2 are used to set the node ID. Each node attached to the network must have an unique node ID which must be different from 0. Switch 8 serves as the least significant bit (LSB).

The node ID is the sum of the values of all switches set to "1". These values are:

Switch	Value
8	1
7	2
6	4
5	8
4	16
3	32
2	64

1 | 128

Some Examples:

Switch 1 2 3 4 5 6 7 8	Hex Node ID	Decimal Node ID
0 0 0 0 0 0 0 0		not allowed
0 0 0 0 0 0 0 1	1	1
0 0 0 0 0 0 1 0	2	2
0 0 0 0 0 0 1 1	3	3
.	.	.
0 1 0 1 0 1 0 1	55	85
.	.	.
1 0 1 0 1 0 1 0	AA	170
.	.	.
1 1 1 1 1 1 0 1	FD	253
1 1 1 1 1 1 1 0	FE	254
1 1 1 1 1 1 1 1	FF	255

Setting the I/O Base Address

The first three switches in switch group SW1 are used to select one of eight possible I/O Base addresses using the following table:

Switch 3 2 1	Hex I/O Address
ON ON ON	260
ON ON OFF	290
ON OFF ON	2E0 (Manufacturer's default)
ON OFF OFF	2F0
OFF ON ON	300
OFF ON OFF	350
OFF OFF ON	380
OFF OFF OFF	3E0

Setting the Base Memory (RAM) buffer Address

The memory buffer requires 2K of a 16K block of RAM. The base of this 16K block can be located in any of eight positions. Switches 6-8 of switch group SW1 select the Base of the 16K block. Within that 16K address space, the buffer may be assigned any one of four positions, determined by the offset, switches 4 and 5 of group SW1:

Switch 8 7 6 5 4	Hex RAM Address	Hex ROM Address
0 0 0 0 0	C0000	C2000
0 0 0 0 1	C0800	C2000

0	0	0	1	0	C1000	C2000
0	0	0	1	1	C1800	C2000
0	0	1	0	0	C4000	C6000
0	0	1	0	1	C4800	C6000
0	0	1	1	0	C5000	C6000
0	0	1	1	1	C5800	C6000
0	1	0	0	0	CC000	CE000
0	1	0	0	1	CC800	CE000
0	1	0	1	0	CD000	CE000
0	1	0	1	1	CD800	CE000
0	1	1	0	0	D0000	D2000 (Manufacturer's default)
0	1	1	0	1	D0800	D2000
0	1	1	1	0	D1000	D2000
0	1	1	1	1	D1800	D2000
1	0	0	0	0	D4000	D6000
1	0	0	0	1	D4800	D6000
1	0	0	1	0	D5000	D6000
1	0	0	1	1	D5800	D6000
1	0	1	0	0	D8000	DA000
1	0	1	0	1	D8800	DA000
1	0	1	1	0	D9000	DA000
1	0	1	1	1	D9800	DA000
1	1	0	0	0	DC000	DE000
1	1	0	0	1	DC800	DE000
1	1	0	1	0	DD000	DE000
1	1	0	1	1	DD800	DE000
1	1	1	0	0	E0000	E2000
1	1	1	0	1	E0800	E2000
1	1	1	1	0	E1000	E2000
1	1	1	1	1	E1800	E2000

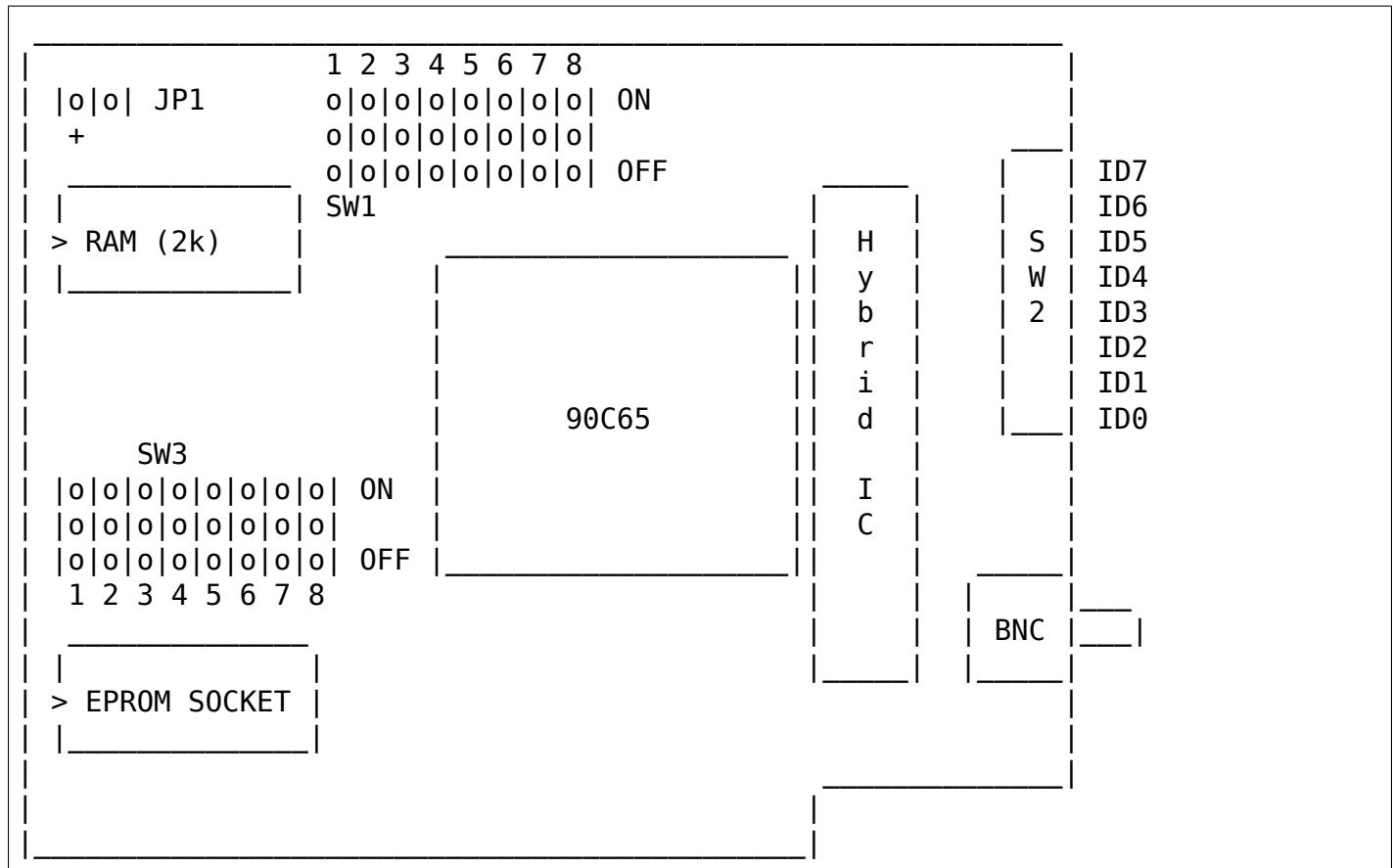
Setting Interrupt Request Lines (IRQ)

Setting the Timeouts

33.16.3 8-bit cards ("Made in Taiwan R.O.C.")

- from Vojtech Pavlik <vojtech@suse.cz>

I have named this ARCnet card "NONAME", since I got only the card with no manual at all and the only text identifying the manufacturer is "MADE IN TAIWAN R.O.C" printed on the card.



Legend:

90C65	ARCNET Chip
SW1 1-5:	Base Memory Address Select
6-8:	Base I/O Address Select
SW2 1-8:	Node ID Select (ID0-ID7)
SW3 1-5:	IRQ Select
6-7:	Extra Timeout
8 :	ROM Enable
JP1	Led connector
BNC	Coax connector

Although the jumpers SW1 and SW3 are marked SW, not JP, they are jumpers, not switches.

Setting the jumpers to ON means connecting the upper two pins, off the bottom two - or - in case of IRQ setting, connecting none of them at all.

Setting the Node ID

The eight switches in SW2 are used to set the node ID. Each node attached to the network must have an unique node ID which must not be 0. Switch 1 (ID0) serves as the least significant bit (LSB).

Setting one of the switches to Off means "1", On means "0".

The node ID is the sum of the values of all switches set to "1" These values are:

Switch	Label	Value
1	ID0	1
2	ID1	2
3	ID2	4
4	ID3	8
5	ID4	16
6	ID5	32
7	ID6	64
8	ID7	128

Some Examples:

Switch 8 7 6 5 4 3 2 1	Hex Node ID	Decimal Node ID
0 0 0 0 0 0 0 0		not allowed
0 0 0 0 0 0 0 1	1	1
0 0 0 0 0 0 1 0	2	2
0 0 0 0 0 0 1 1	3	3
.	.	.
0 1 0 1 0 1 0 1	55	85
.	.	.
1 0 1 0 1 0 1 0	AA	170
.	.	.
1 1 1 1 1 1 0 1	FD	253
1 1 1 1 1 1 1 0	FE	254
1 1 1 1 1 1 1 1	FF	255

Setting the I/O Base Address

The last three switches in switch block SW1 are used to select one of eight possible I/O Base addresses using the following table:

Switch 6 7 8	Hex I/O Address
ON ON ON	260
OFF ON ON	290
ON OFF ON	2E0 (Manufacturer's default)
OFF OFF ON	2F0
ON ON OFF	300

OFF	ON	OFF		350
ON	OFF	OFF		380
OFF	OFF	OFF		3E0

Setting the Base Memory (RAM) buffer Address

The memory buffer (RAM) requires 2K. The base of this buffer can be located in any of eight positions. The address of the Boot Prom is memory base + 0x2000.

Jumpers 3-5 of jumper block SW1 select the Memory Base address.

Switch					Hex RAM	Hex ROM
1	2	3	4	5	Address	Address *)
ON	ON	ON	ON	ON	C0000	C2000
ON	ON	OFF	ON	ON	C4000	C6000
ON	ON	ON	OFF	ON	CC000	CE000
ON	ON	OFF	OFF	ON	D0000	D2000 (Manufacturer's default)
ON	ON	ON	ON	OFF	D4000	D6000
ON	ON	OFF	ON	OFF	D8000	DA000
ON	ON	ON	OFF	OFF	DC000	DE000
ON	ON	OFF	OFF	OFF	E0000	E2000

*) To enable the Boot ROM set the jumper 8 of jumper block SW3 to position ON.

The jumpers 1 and 2 probably add 0x0800, 0x1000 and 0x1800 to RAM adders.

Setting the Interrupt Line

Jumpers 1-5 of the jumper block SW3 control the IRQ level:

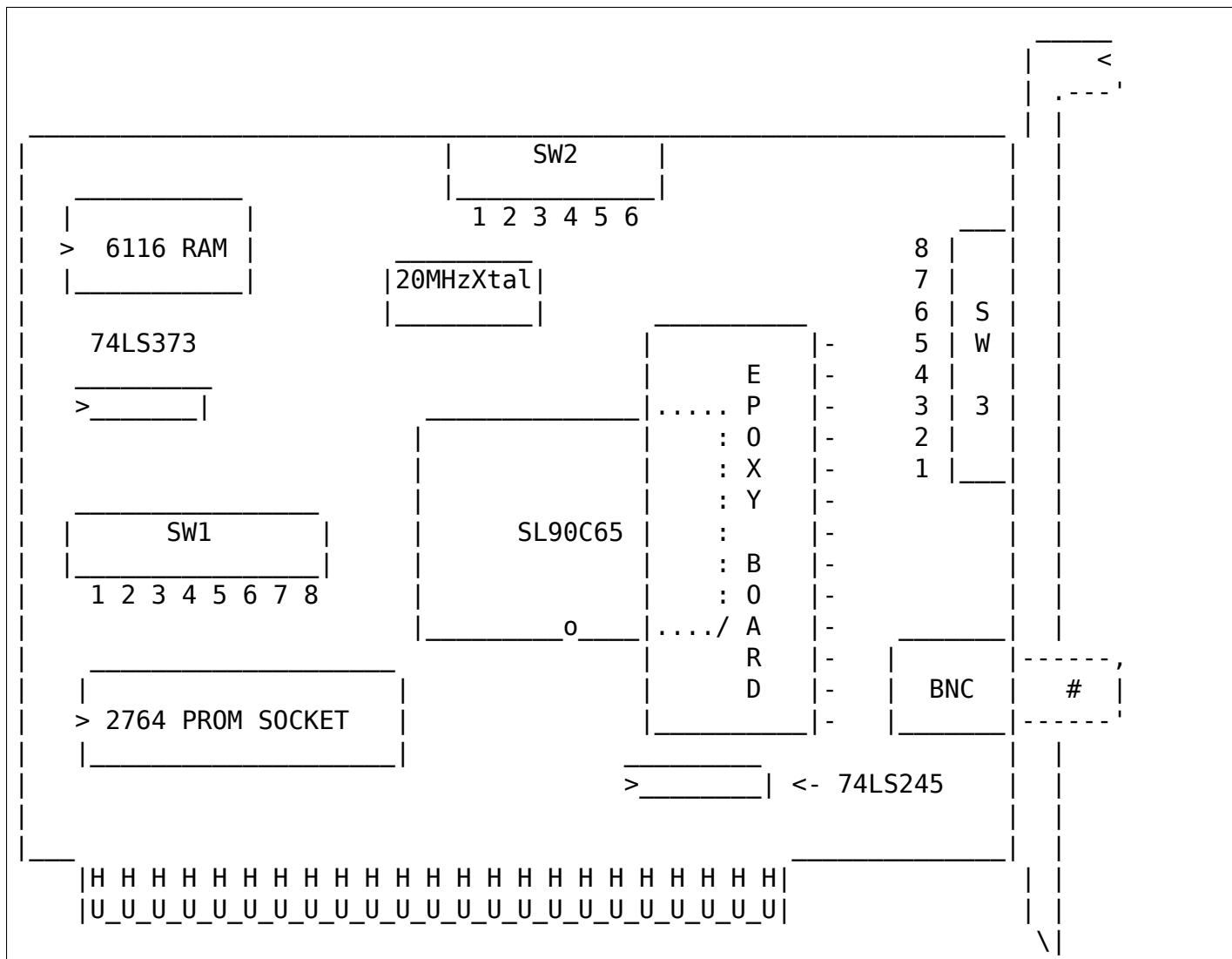
Jumper					IRQ
1	2	3	4	5	
ON	OFF	OFF	OFF	OFF	2
OFF	ON	OFF	OFF	OFF	3
OFF	OFF	ON	OFF	OFF	4
OFF	OFF	OFF	ON	OFF	5
OFF	OFF	OFF	OFF	ON	7

Setting the Timeout Parameters

The jumpers 6-7 of the jumper block SW3 are used to determine the timeout parameters. These two jumpers are normally left in the OFF position.

33.16.4 (Generic Model 9058)

- from Andrew J. Kroll <ag784@freenet.buffalo.edu>
- Sorry this sat in my to-do box for so long, Andrew! (yikes - over a year!)



Legend:

SL90C65	ARCNET Controller / Transceiver / Logic
SW1 1-5:	IRQ Select
6:	ET1
7:	ET2
8:	ROM ENABLE
SW2 1-3:	Memory Buffer/PROM Address
3-6:	I/O Address Map

SW3 1-8: Node ID Select
 BNC BNC RG62/U Connection
 I have had success using RG59B/U with *NO* terminators!
 What gives?!

SW1: Timeouts, Interrupt and ROM

To select a hardware interrupt level set one (only one!) of the dip switches up (on) SW1... (switches 1-5) IRQ3, IRQ4, IRQ5, IRQ7, IRQ2. The Manufacturer's default is IRQ2.

The switches on SW1 labeled EXT1 (switch 6) and EXT2 (switch 7) are used to determine the timeout parameters. These two dip switches are normally left off (down).

To enable the 8K Boot PROM position SW1 switch 8 on (UP) labeled ROM. The default is jumper ROM not installed.

Setting the I/O Base Address

The last three switches in switch group SW2 are used to select one of eight possible I/O Base addresses using the following table:

Switch	Hex I/O Address
4 5 6	
0 0 0	260
0 0 1	290
0 1 0	2E0 (Manufacturer's default)
0 1 1	2F0
1 0 0	300
1 0 1	350
1 1 0	380
1 1 1	3E0

Setting the Base Memory Address (RAM & ROM)

The memory buffer requires 2K of a 16K block of RAM. The base of this 16K block can be located in any of eight positions. Switches 1-3 of switch group SW2 select the Base of the 16K block. (0 = DOWN, 1 = UP) I could, however, only verify two settings...

Switch	Hex RAM Address	Hex ROM Address
1 2 3		
0 0 0	E0000	E2000
0 0 1	D0000	D2000 (Manufacturer's default)
0 1 0	?????	?????
0 1 1	?????	?????
1 0 0	?????	?????
1 0 1	?????	?????

1	1	0	?????	?????	?????
1	1	1	?????	?????	?????

Setting the Node ID

The eight switches in group SW3 are used to set the node ID. Each node attached to the network must have an unique node ID which must be different from 0. Switch 1 serves as the least significant bit (LSB). switches in the DOWN position are OFF (0) and in the UP position are ON (1)

The node ID is the sum of the values of all switches set to "1" These values are:

Switch	Value
1	1
2	2
3	4
4	8
5	16
6	32
7	64
8	128

Some Examples:

Switch#	Hex	Decimal
8 7 6 5 4 3 2 1	Node ID	Node ID
0 0 0 0 0 0 0 0	not allowed	<- .
0 0 0 0 0 0 0 1	1	1
0 0 0 0 0 0 1 0	2	2
0 0 0 0 0 0 1 1	3	3
.		
0 1 0 1 0 1 0 1	55	85
.		+ Don't use 0 or 255!
1 0 1 0 1 0 1 0	AA	170
.		
1 1 1 1 1 1 0 1	FD	253
1 1 1 1 1 1 1 0	FE	254
1 1 1 1 1 1 1 1	FF	255 <-

33.17 Tiara

33.17.1 (model unknown)

- from Christoph Lameter <christoph@lameter.com>

Here is information about my card as far as I could figure it out:

```
----- tiara
Tiara LanCard of Tiara Computer Systems.

+-----+
!      ! Transmitter Unit !
!      +-----+
!      MEM                      Coax Connector
! ROM   7654321 <- I/O          -----
! : : +-----+                  !
! : : ! 90C66LJ!                ++++
! : : !           !             !D  Switch to set
! : : !           !             !I  the Nodenumber
! : : +-----+                  !P
!                               !++
!      234567 <- IR0          !
+-----!!!!!!!!!!!!!!-----+
                           !!!!!!!
```

- 0 = Jumper Installed
- 1 = Open

Top Jumper line Bit 7 = ROM Enable 654=Memory location 321=I/O

Settings for Memory Location (Top Jumper Line)

456	Address selected
000	C0000
001	C4000
010	CC000
011	D0000
100	D4000
101	D8000
110	DC000
111	E0000

Settings for I/O Address (Top Jumper Line)

	Port
123	260
000	290
001	2E0
010	2F0
100	300
101	350
110	380
111	3E0

Settings for IRQ Selection (Lower Jumper Line)

234567	
011111	IRQ 2
101111	IRQ 3
110111	IRQ 4
111011	IRQ 5
111110	IRQ 7

33.18 Other Cards

I have no information on other models of ARCnet cards at the moment. Please send any and all info to:

apenwarr@worldvisions.ca

Thanks.

**CHAPTER
THIRTYFOUR**

ARCNET

Note: See also *ARCnet Hardware* in this directory for jumper-setting and cabling information if you're like many of us and didn't happen to get a manual with your ARCnet card.

Since no one seems to listen to me otherwise, perhaps a poem will get your attention:

This driver's getting fat and beefy,
But my cat is still named Fifi.

Hmm, I think I'm allowed to call that a poem, even though it's only two lines. Hey, I'm in Computer Science, not English. Give me a break.

The point is: I REALLY REALLY REALLY REALLY want to hear from you if you test this and get it working. Or if you don't. Or anything.

ARCnet 0.32 ALPHA first made it into the Linux kernel 1.1.80 - this was nice, but after that even FEWER people started writing to me because they didn't even have to install the patch.
<sigh>

Come on, be a sport! Send me a success report!

(hey, that was even better than my original poem... this is getting bad!)

Warning: If you don't e-mail me about your success/failure soon, I may be forced to start SINGING. And we don't want that, do we?

(You know, it might be argued that I'm pushing this point a little too much. If you think so, why not flame me in a quick little e-mail? Please also include the type of card(s) you're using, software, size of network, and whether it's working or not.)

My e-mail address is: aopenwarr@worldvisions.ca

These are the ARCnet drivers for Linux.

This new release (2.91) has been put together by David Woodhouse <dwmw2@infradead.org>, in an attempt to tidy up the driver after adding support for yet another chipset. Now the generic support has been separated from the individual chipset drivers, and the source files aren't quite so packed with #ifdefs! I've changed this file a bit, but kept it in the first person from Avery, because I didn't want to completely rewrite it.

The previous release resulted from many months of on-and-off effort from me (Avery Pennarun), many bug reports/fixes and suggestions from others, and in particular a lot of input and coding

from Tomasz Motylewski. Starting with ARCnet 2.10 ALPHA, Tomasz's all-new-and-improved RFC1051 support has been included and seems to be working fine!

34.1 Where do I discuss these drivers?

Tomasz has been so kind as to set up a new and improved mailing list. Subscribe by sending a message with the BODY "subscribe linux-arcnet YOUR REAL NAME" to listserv@tichy.ch.uj.edu.pl. Then, to submit messages to the list, mail to linux-arcnet@tichy.ch.uj.edu.pl.

There are archives of the mailing list at:

<http://epistolary.org/mailman/listinfo.cgi/arcnet>

The people on linux-net@vger.kernel.org (now defunct, replaced by netdev@vger.kernel.org) have also been known to be very helpful, especially when we're talking about ALPHA Linux kernels that may or may not work right in the first place.

34.2 Other Drivers and Info

You can try my ARCNET page on the World Wide Web at:

<http://www.qis.net/~jschmitz/arcnet/>

Also, SMC (one of the companies that makes ARCnet cards) has a WWW site you might be interested in, which includes several drivers for various cards including ARCnet. Try:

<http://www.smc.com/>

Performance Technologies makes various network software that supports ARCnet:

<http://www.perftech.com/> or [ftp to ftp.perftech.com](ftp://ftp.perftech.com).

Novell makes a networking stack for DOS which includes ARCnet drivers. Try FTPing to [ftp.novell.com](ftp://ftp.novell.com).

You can get the Crynwr packet driver collection (including arcether.com, the one you'll want to use with ARCnet cards) from oak.oakland.edu:/simtel/msdos/pktdrv. It won't work perfectly on a 386+ without patches, though, and also doesn't like several cards. Fixed versions are available on my WWW page, or via e-mail if you don't have WWW access.

34.3 Installing the Driver

All you will need to do in order to install the driver is:

```
make config
      (be sure to choose ARCnet in the network devices
       and at least one chipset driver.)
make clean
make zImage
```

If you obtained this ARCnet package as an upgrade to the ARCnet driver in your current kernel, you will need to first copy arcnet.c over the one in the linux/drivers/net directory.

You will know the driver is installed properly if you get some ARCnet messages when you reboot into the new Linux kernel.

There are four chipset options:

1. Standard ARCnet COM90xx chipset.

This is the normal ARCnet card, which you've probably got. This is the only chipset driver which will autoprobe if not told where the card is. It follows options on the command line:

```
com90xx=[<io>[,<irq>[,<shmem>]]][,<name>] | <name>
```

If you load the chipset support as a module, the options are:

```
io=<io> irq=<irq> shmem=<shmem> device=<name>
```

To disable the autoprobe, just specify "com90xx=" on the kernel command line. To specify the name alone, but allow autoprobe, just put "com90xx=<name>"

2. ARCnet COM20020 chipset.

This is the new chipset from SMC with support for promiscuous mode (packet sniffing), extra diagnostic information, etc. Unfortunately, there is no sensible method of autoprobing for these cards. You must specify the I/O address on the kernel command line.

The command line options are:

```
com20020=<io>[,<irq>[,<node_ID>[,backplane[,CKP[,timeout]]]]][,name]
```

If you load the chipset support as a module, the options are:

```
io=<io> irq=<irq> node=<node_ID> backplane=<backplane> clock=<CKP>
timeout=<timeout> device=<name>
```

The COM20020 chipset allows you to set the node ID in software, overriding the default which is still set in DIP switches on the card. If you don't have the COM20020 data sheets, and you don't know what the other three options refer to, then they won't interest you - forget them.

3. ARCnet COM90xx chipset in IO-mapped mode.

This will also work with the normal ARCnet cards, but doesn't use the shared memory. It performs less well than the above driver, but is provided in case you have a card which doesn't support shared memory, or (strangely) in case you have so many ARCnet cards in your machine that you run out of shmem slots. If you don't give the IO address on the kernel command line, then the driver will not find the card.

The command line options are:

```
com90io=<io>[,<irq>][,<name>]
```

If you load the chipset support as a module, the options are:

```
io=<io> irq=<irq> device=<name>
```

4. ARCnet RIM I cards.

These are COM90xx chips which are completely memory mapped. The support for these is not tested. If you have one, please mail the author with a success report. All options must be specified, except the device name. Command line options:

```
arcrimi=<shmem>,<irq>,<node_ID>[ ,<name>]
```

If you load the chipset support as a module, the options are:

```
shmem=<shmem> irq=<irq> node=<node_ID> device=<name>
```

34.4 Loadable Module Support

Configure and rebuild Linux. When asked, answer 'm' to "Generic ARCnet support" and to support for your ARCnet chipset if you want to use the loadable module. You can also say 'y' to "Generic ARCnet support" and 'm' to the chipset support if you wish.

```
make config  
make clean  
make zImage  
make modules
```

If you're using a loadable module, you need to use insmod to load it, and you can specify various characteristics of your card on the command line. (In recent versions of the driver, autoprobing is much more reliable and works as a module, so most of this is now unnecessary.)

For example:

```
cd /usr/src/linux/modules  
insmod arcnet.o  
insmod com90xx.o  
insmod com20020.o io=0x2e0 device=eth1
```

34.5 Using the Driver

If you build your kernel with ARCnet COM90xx support included, it should probe for your card automatically when you boot. If you use a different chipset driver complied into the kernel, you must give the necessary options on the kernel command line, as detailed above.

Go read the NET-2-HOWTO and ETHERNET-HOWTO for Linux; they should be available where you picked up this driver. Think of your ARCnet as a souped-up (or down, as the case may be) Ethernet card.

By the way, be sure to change all references from "eth0" to "arc0" in the HOWTOs. Remember that ARCnet isn't a "true" Ethernet, and the device name is DIFFERENT.

34.6 Multiple Cards in One Computer

Linux has pretty good support for this now, but since I've been busy, the ARCnet driver has somewhat suffered in this respect. COM90xx support, if compiled into the kernel, will (try to) autodetect all the installed cards.

If you have other cards, with support compiled into the kernel, then you can just repeat the options on the kernel command line, e.g.:

```
LIL0: linux com20020=0x2e0 com20020=0x380 com90io=0x260
```

If you have the chipset support built as a loadable module, then you need to do something like this:

```
insmod -o arc0 com90xx
insmod -o arc1 com20020 io=0x2e0
insmod -o arc2 com90xx
```

The ARCnet drivers will now sort out their names automatically.

34.7 How do I get it to work with...?

NFS:

Should be fine linux->linux, just pretend you're using Ethernet cards. oak.oakland.edu:/simtel/msdos/nfs has some nice DOS clients. There is also a DOS-based NFS server called SOSS. It doesn't multitask quite the way Linux does (actually, it doesn't multitask AT ALL) but you never know what you might need.

With AmiTCP (and possibly others), you may need to set the following options in your Amiga nfstab: MD 1024 MR 1024 MW 1024 (Thanks to Christian Gottschling <ferksy@indigo.tng.oche.de> for this.)

Probably these refer to maximum NFS data/read/write block sizes. I don't know why the defaults on the Amiga didn't work; write to me if you know more.

DOS:

If you're using the freeware arcether.com, you might want to install the driver patch from my web page. It helps with PC/TCP, and also can get arcether to load if it timed out too quickly during initialization. In fact, if you use it on a 386+ you REALLY need the patch, really.

Windows:

See DOS :) Trumpet Winsock works fine with either the Novell or Arcether client, assuming you remember to load winpkt of course.

LAN Manager and Windows for Workgroups:

These programs use protocols that are incompatible with the Internet standard. They try to pretend the cards are Ethernet, and confuse everyone else on the network.

However, v2.00 and higher of the Linux ARCnet driver supports this protocol via the 'arc0e' device. See the section on "Multiprotocol Support" for more information.

Using the freeware Samba server and clients for Linux, you can now interface quite nicely with TCP/IP-based WfWg or Lan Manager networks.

Windows 95:

Tools are included with Win95 that let you use either the LANMAN style network drivers (NDIS) or Novell drivers (ODI) to handle your ARCnet packets. If you use ODI, you'll need to use the 'arc0' device with Linux. If you use NDIS, then try the 'arc0e' device. See the "Multiprotocol Support" section below if you need arc0e, you're completely insane, and/or you need to build some kind of hybrid network that uses both encapsulation types.

OS/2:

I've been told it works under Warp Connect with an ARCnet driver from SMC. You need to use the 'arc0e' interface for this. If you get the SMC driver to work with the TCP/IP stuff included in the "normal" Warp Bonus Pack, let me know.

ftp.microsoft.com also has a freeware "Lan Manager for OS/2" client which should use the same protocol as WfWg does. I had no luck installing it under Warp, however. Please mail me with any results.

NetBSD/AmiTCP:

These use an old version of the Internet standard ARCnet protocol (RFC1051) which is compatible with the Linux driver v2.10 ALPHA and above using the arc0s device. (See "Multiprotocol ARCnet" below.) ** Newer versions of NetBSD apparently support RFC1201.

34.8 Using Multiprotocol ARCnet

The ARCnet driver v2.10 ALPHA supports three protocols, each on its own "virtual network device":

arc0	RFC1201 protocol, the official Internet standard which just happens to be 100% compatible with Novell's TRXNET driver. Version 1.00 of the ARCnet driver supported <i>only</i> this protocol. arc0 is the fastest of the three protocols (for whatever reason), and allows larger packets to be used because it supports RFC1201 "packet splitting" operations. Unless you have a specific need to use a different protocol, I strongly suggest that you stick with this one.
arc0e	"Ethernet-Encapsulation" which sends packets over ARCnet that are actually a lot like Ethernet packets, including the 6-byte hardware addresses. This protocol is compatible with Microsoft's NDIS ARCnet driver, like the one in WfWg and LANMAN. Because the MTU of 493 is actually smaller than the one "required" by TCP/IP (576), there is a chance that some network operations will not function properly. The Linux TCP/IP layer can compensate in most cases, however, by automatically fragmenting the TCP/IP packets to make them fit. arc0e also works slightly more slowly than arc0, for reasons yet to be determined. (Probably it's the smaller MTU that does it.)
arc0s	The "[s]imple" RFC1051 protocol is the "previous" Internet standard that is completely incompatible with the new standard. Some software today, however, continues to support the old standard (and only the old standard) including NetBSD and AmiTCP. RFC1051 also does not support RFC1201's packet splitting, and the MTU of 507 is still smaller than the Internet "requirement," so it's quite possible that you may run into problems. It's also slower than RFC1201 by about 25%, for the same reason as arc0e. The arc0s support was contributed by Tomasz Motylewski and modified somewhat by me. Bugs are probably my fault.

You can choose not to compile arc0e and arc0s into the driver if you want - this will save you a bit of memory and avoid confusion when eg. trying to use the "NFS-root" stuff in recent Linux kernels.

The arc0e and arc0s devices are created automatically when you first ifconfig the arc0 device. To actually use them, though, you need to also ifconfig the other virtual devices you need. There are a number of ways you can set up your network then:

1. Single Protocol.

This is the simplest way to configure your network: use just one of the two available protocols. As mentioned above, it's a good idea to use only arc0 unless you have a good reason (like some other software, ie. WfWg, that only works with arc0e).

If you need only arc0, then the following commands should get you going:

```
ifconfig arc0 MY.IP.ADD.RESS
route add MY.IP.ADD.RESS arc0
route add -net SUB.NET.ADD.RESS arc0
[add other local routes here]
```

If you need arc0e (and only arc0e), it's a little different:

```
ifconfig arc0 MY.IP.ADD.RESS
ifconfig arc0e MY.IP.ADD.RESS
```

```
route add MY.IP.ADD.RESS arc0e
route add -net SUB.NET.ADD.RESS arc0e
```

arc0s works much the same way as arc0e.

2. More than one protocol on the same wire.

Now things start getting confusing. To even try it, you may need to be partly crazy. Here's what I did. :) Note that I don't include arc0s in my home network; I don't have any NetBSD or AmiTCP computers, so I only use arc0s during limited testing.

I have three computers on my home network; two Linux boxes (which prefer RFC1201 protocol, for reasons listed above), and one XT that can't run Linux but runs the free Microsoft LANMAN Client instead.

Worse, one of the Linux computers (freedom) also has a modem and acts as a router to my Internet provider. The other Linux box (insight) also has its own IP address and needs to use freedom as its default gateway. The XT (patience), however, does not have its own Internet IP address and so I assigned it one on a "private subnet" (as defined by RFC1597).

To start with, take a simple network with just insight and freedom. Insight needs to:

- talk to freedom via RFC1201 (arc0) protocol, because I like it more and it's faster.
- use freedom as its Internet gateway.

That's pretty easy to do. Set up insight like this:

```
ifconfig arc0 insight
route add insight arc0
route add freedom arc0 /* I would use the subnet here (like I said
                         to in "single protocol" above),
                         but the rest of the subnet
                         unfortunately lies across the PPP
                         link on freedom, which confuses
                         things. */
route add default gw freedom
```

And freedom gets configured like so:

```
ifconfig arc0 freedom
route add freedom arc0
route add insight arc0
/* and default gateway is configured by pppd */
```

Great, now insight talks to freedom directly on arc0, and sends packets to the Internet through freedom. If you didn't know how to do the above, you should probably stop reading this section now because it only gets worse.

Now, how do I add patience into the network? It will be using LANMAN Client, which means I need the arc0e device. It needs to be able to talk to both insight and freedom, and also use freedom as a gateway to the Internet. (Recall that patience has a "private IP address" which won't work on the Internet; that's okay, I configured Linux IP masquerading on freedom for this subnet).

So patience (necessarily; I don't have another IP number from my provider) has an IP address on a different subnet than freedom and insight, but needs to use freedom as an Internet gateway. Worse, most DOS networking programs, including LANMAN, have braindead networking schemes that rely completely on the netmask and a 'default gateway' to determine how to route packets. This means that to get to freedom or insight, patience WILL send through its default gateway, regardless of the fact that both freedom and insight (courtesy of the arc0e device) could understand a direct transmission.

I compensate by giving freedom an extra IP address - aliased 'gatekeeper' - that is on my private subnet, the same subnet that patience is on. I then define gatekeeper to be the default gateway for patience.

To configure freedom (in addition to the commands above):

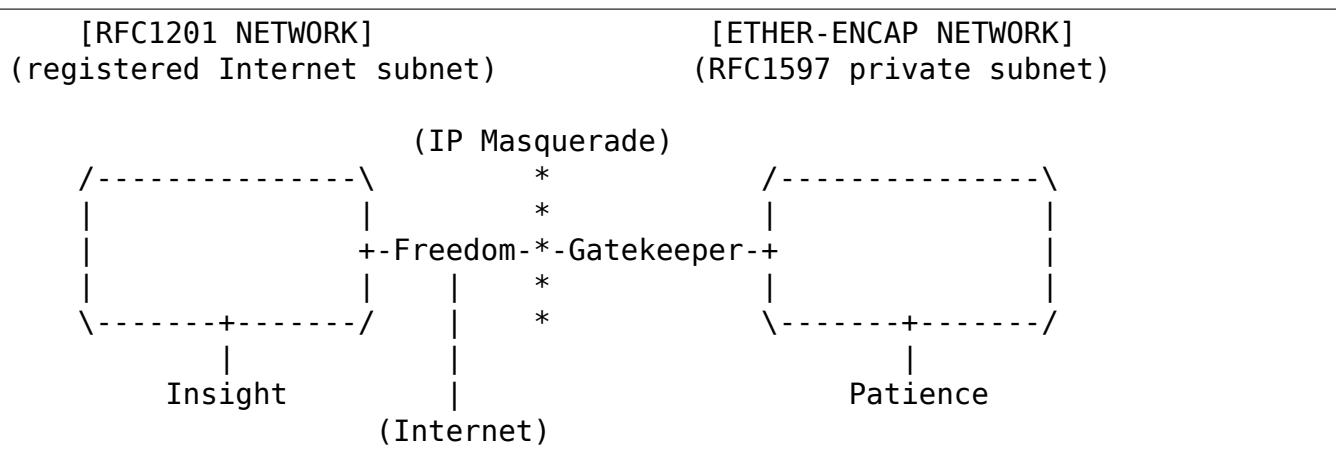
```
ifconfig arc0e gatekeeper
route add gatekeeper arc0e
route add patience arc0e
```

This way, freedom will send all packets for patience through arc0e, giving its IP address as gatekeeper (on the private subnet). When it talks to insight or the Internet, it will use its "freedom" Internet IP address.

You will notice that we haven't configured the arc0e device on insight. This would work, but is not really necessary, and would require me to assign insight another special IP number from my private subnet. Since both insight and patience are using freedom as their default gateway, the two can already talk to each other.

It's quite fortunate that I set things up like this the first time (cough cough) because it's really handy when I boot insight into DOS. There, it runs the Novell ODI protocol stack, which only works with RFC1201 ARCnet. In this mode it would be impossible for insight to communicate directly with patience, since the Novell stack is incompatible with Microsoft's Ethernet-Encap. Without changing any settings on freedom or patience, I simply set freedom as the default gateway for insight (now in DOS, remember) and all the forwarding happens "automagically" between the two hosts that would normally not be able to communicate at all.

For those who like diagrams, I have created two "virtual subnets" on the same physical ARCnet wire. You can picture it like this:



34.9 It works: what now?

Send mail describing your setup, preferably including driver version, kernel version, ARCnet card model, CPU type, number of systems on your network, and list of software in use to me at the following address:

apenwarr@worldvisions.ca

I do send (sometimes automated) replies to all messages I receive. My email can be weird (and also usually gets forwarded all over the place along the way to me), so if you don't get a reply within a reasonable time, please resend.

34.10 It doesn't work: what now?

Do the same as above, but also include the output of the ifconfig and route commands, as well as any pertinent log entries (ie. anything that starts with "arcnet:" and has shown up since the last reboot) in your mail.

If you want to try fixing it yourself (I strongly recommend that you mail me about the problem first, since it might already have been solved) you may want to try some of the debug levels available. For heavy testing on D_DURING or more, it would be a REALLY good idea to kill your klogd daemon first! D_DURING displays 4-5 lines for each packet sent or received. D_TX, D_RX, and D_SKB actually DISPLAY each packet as it is sent or received, which is obviously quite big.

Starting with v2.40 ALPHA, the autoprobe routines have changed significantly. In particular, they won't tell you why the card was not found unless you turn on the D_INIT_REASONS debugging flag.

Once the driver is running, you can run the arcdump shell script (available from me or in the full ARCnet package, if you have it) as root to list the contents of the arcnet buffers at any time. To make any sense at all out of this, you should grab the pertinent RFCs. (some are listed near the top of arcnet.c). arcdump assumes your card is at 0xD0000. If it isn't, edit the script.

Buffers 0 and 1 are used for receiving, and Buffers 2 and 3 are for sending. Ping-pong buffers are implemented both ways.

If your debug level includes D_DURING and you did NOT define SLOW_XMIT_COPY, the buffers are cleared to a constant value of 0x42 every time the card is reset (which should only happen when you do an ifconfig up, or when Linux decides that the driver is broken). During a transmit, unused parts of the buffer will be cleared to 0x42 as well. This is to make it easier to figure out which bytes are being used by a packet.

You can change the debug level without recompiling the kernel by typing:

```
ifconfig arc0 down metric 1xxx  
/etc/rc.d/rc.inet1
```

where "xxx" is the debug level you want. For example, "metric 1015" would put you at debug level 15. Debug level 7 is currently the default.

Note that the debug level is (starting with v1.90 ALPHA) a binary combination of different debug flags; so debug level 7 is really 1+2+4 or D_NORMAL+D_EXTRA+D_INIT. To include D_DURING, you would add 16 to this, resulting in debug level 23.

If you don't understand that, you probably don't want to know anyway. E-mail me about your problem.

34.11 I want to send money: what now?

Go take a nap or something. You'll feel better in the morning.

**CHAPTER
THIRTYFIVE**

ATM

In order to use anything but the most primitive functions of ATM, several user-mode programs are required to assist the kernel. These programs and related material can be found via the ATM on Linux Web page at <http://linux-atm.sourceforge.net/>

If you encounter problems with ATM, please report them on the ATM on Linux mailing list. Subscription information, archives, etc., can be found on <http://linux-atm.sourceforge.net/>

CHAPTER
THIRTYSIX

AX.25

To use the amateur radio protocols within Linux you will need to get a suitable copy of the AX.25 Utilities. More detailed information about AX.25, NET/ROM and ROSE, associated programs and utilities can be found on <https://linux-ax25.in-berlin.de>.

There is a mailing list for discussing Linux amateur radio matters called linux-hams@vger.kernel.org. To subscribe to it, send a message to majordomo@vger.kernel.org with the words "subscribe linux-hams" in the body of the message, the subject field is ignored. You don't need to be subscribed to post but of course that means you might miss an answer.

LINUX ETHERNET BONDING DRIVER HOWTO

Latest update: 27 April 2011

Initial release: Thomas Davis <tadavis at lbl.gov>

Corrections, HA extensions: 2000/10/03-15:

- Willy Tarreau <willy at meta-x.org>
- Constantine Gavrilov <const-g at xpert.com>
- Chad N. Tindel <ctindel at ieee dot org>
- Janice Girouard <girouard at us dot ibm dot com>
- Jay Vosburgh <fubar at us dot ibm dot com>

Reorganized and updated Feb 2005 by Jay Vosburgh Added Sysfs information: 2006/04/24

- Mitch Williams <mitch.a.williams at intel.com>

37.1 Introduction

The Linux bonding driver provides a method for aggregating multiple network interfaces into a single logical "bonded" interface. The behavior of the bonded interfaces depends upon the mode; generally speaking, modes provide either hot standby or load balancing services. Additionally, link integrity monitoring may be performed.

The bonding driver originally came from Donald Becker's beowulf patches for kernel 2.0. It has changed quite a bit since, and the original tools from extreme-linux and beowulf sites will not work with this version of the driver.

For new versions of the driver, updated userspace tools, and who to ask for help, please follow the links at the end of this file.

37.2 1. Bonding Driver Installation

Most popular distro kernels ship with the bonding driver already available as a module. If your distro does not, or you have need to compile bonding from source (e.g., configuring and installing a mainline kernel from kernel.org), you'll need to perform the following steps:

37.2.1 1.1 Configure and build the kernel with bonding

The current version of the bonding driver is available in the drivers/net/bonding subdirectory of the most recent kernel source (which is available on <http://kernel.org>). Most users "rolling their own" will want to use the most recent kernel from kernel.org.

Configure kernel with "make menuconfig" (or "make xconfig" or "make config"), then select "Bonding driver support" in the "Network device support" section. It is recommended that you configure the driver as module since it is currently the only way to pass parameters to the driver or configure more than one bonding device.

Build and install the new kernel and modules.

37.2.2 1.2 Bonding Control Utility

It is recommended to configure bonding via iproute2 (netlink) or sysfs, the old ifenslave control utility is obsolete.

37.3 2. Bonding Driver Options

Options for the bonding driver are supplied as parameters to the bonding module at load time, or are specified via sysfs.

Module options may be given as command line arguments to the insmod or modprobe command, but are usually specified in either the /etc/modprobe.d/*.conf configuration files, or in a distro-specific configuration file (some of which are detailed in the next section).

Details on bonding support for sysfs is provided in the "Configuring Bonding Manually via Sysfs" section, below.

The available bonding driver parameters are listed below. If a parameter is not specified the default value is used. When initially configuring a bond, it is recommended "tail -f /var/log/messages" be run in a separate window to watch for bonding driver error messages.

It is critical that either the miimon or arp_interval and arp_ip_target parameters be specified, otherwise serious network degradation will occur during link failures. Very few devices do not support at least miimon, so there is really no reason not to use it.

Options with textual values will accept either the text name or, for backwards compatibility, the option value. E.g., "mode=802.3ad" and "mode=4" set the same mode.

The parameters are as follows:

active_slave

Specifies the new active slave for modes that support it (active-backup, balance-alb and balance-tlb). Possible values are the name of any currently enslaved interface,

or an empty string. If a name is given, the slave and its link must be up in order to be selected as the new active slave. If an empty string is specified, the current active slave is cleared, and a new active slave is selected automatically.

Note that this is only available through the sysfs interface. No module parameter by this name exists.

The normal value of this option is the name of the currently active slave, or the empty string if there is no active slave or the current mode does not use an active slave.

ad_actor_sys_prio

In an AD system, this specifies the system priority. The allowed range is 1 - 65535. If the value is not specified, it takes 65535 as the default value.

This parameter has effect only in 802.3ad mode and is available through SysFs interface.

ad_actor_system

In an AD system, this specifies the mac-address for the actor in protocol packet exchanges (LACPDU). The value cannot be a multicast address. If the all-zeroes MAC is specified, bonding will internally use the MAC of the bond itself. It is preferred to have the local-admin bit set for this mac but driver does not enforce it. If the value is not given then system defaults to using the masters' mac address as actors' system address.

This parameter has effect only in 802.3ad mode and is available through SysFs interface.

ad_select

Specifies the 802.3ad aggregation selection logic to use. The possible values and their effects are:

stable or 0

The active aggregator is chosen by largest aggregate bandwidth.

Reselection of the active aggregator occurs only when all slaves of the active aggregator are down or the active aggregator has no slaves.

This is the default value.

bandwidth or 1

The active aggregator is chosen by largest aggregate bandwidth. Reselection occurs if:

- A slave is added to or removed from the bond
- Any slave's link state changes
- Any slave's 802.3ad association state changes
- The bond's administrative state changes to up

count or 2

The active aggregator is chosen by the largest number of ports (slaves). Reselection occurs as described under the "bandwidth" setting, above.

The bandwidth and count selection policies permit failover of 802.3ad aggregations when partial failure of the active aggregator occurs. This keeps the aggregator with the highest availability (either in bandwidth or in number of ports) active at all times.

This option was added in bonding version 3.4.0.

ad_user_port_key

In an AD system, the port-key has three parts as shown below -

Bits	Use
00	Duplex
01-05	Speed
06-15	User-defined

This defines the upper 10 bits of the port key. The values can be from 0 - 1023. If not given, the system defaults to 0.

This parameter has effect only in 802.3ad mode and is available through SysFs interface.

all_slaves_active

Specifies that duplicate frames (received on inactive ports) should be dropped (0) or delivered (1).

Normally, bonding will drop duplicate frames (received on inactive ports), which is desirable for most users. But there are some times it is nice to allow duplicate frames to be delivered.

The default value is 0 (drop duplicate frames received on inactive ports).

arp_interval

Specifies the ARP link monitoring frequency in milliseconds.

The ARP monitor works by periodically checking the slave devices to determine whether they have sent or received traffic recently (the precise criteria depends upon the bonding mode, and the state of the slave). Regular traffic is generated via ARP probes issued for the addresses specified by the arp_ip_target option.

This behavior can be modified by the arp_validate option, below.

If ARP monitoring is used in an etherchannel compatible mode (modes 0 and 2), the switch should be configured in a mode that evenly distributes packets across all links. If the switch is configured to distribute the packets in an XOR fashion, all replies from the ARP targets will be received on the same link which could cause the other team members to fail. ARP monitoring should not be used in conjunction with miimon. A value of 0 disables ARP monitoring. The default value is 0.

arp_ip_target

Specifies the IP addresses to use as ARP monitoring peers when arp_interval is > 0. These are the targets of the ARP request sent to determine the health of the link to the targets. Specify these values in ddd.ddd.ddd.ddd format. Multiple IP addresses must be separated by a comma. At least one IP address must be given for ARP monitoring to function. The maximum number of targets that can be specified is 16. The default value is no IP addresses.

ns_ip6_target

Specifies the IPv6 addresses to use as IPv6 monitoring peers when arp_interval is > 0. These are the targets of the NS request sent to determine the health of the link to the targets. Specify these values in ffff:ffff::ffff:ffff format. Multiple IPv6 addresses must be separated by a comma. At least one IPv6 address must be given for NS/NA monitoring to function. The maximum number of targets that can be specified is 16. The default value is no IPv6 addresses.

arp_validate

Specifies whether or not ARP probes and replies should be validated in any mode that supports arp monitoring, or whether non-ARP traffic should be filtered (disregarded) for link monitoring purposes.

Possible values are:

none or 0

No validation or filtering is performed.

active or 1

Validation is performed only for the active slave.

backup or 2

Validation is performed only for backup slaves.

all or 3

Validation is performed for all slaves.

filter or 4

Filtering is applied to all slaves. No validation is performed.

filter_active or 5

Filtering is applied to all slaves, validation is performed only for the active slave.

filter_backup or 6

Filtering is applied to all slaves, validation is performed only for backup slaves.

Validation:

Enabling validation causes the ARP monitor to examine the incoming ARP requests and replies, and only consider a slave to be up if it is receiving the appropriate ARP traffic.

For an active slave, the validation checks ARP replies to confirm that they were generated by an arp_ip_target. Since backup slaves do not typically receive these replies, the validation performed for backup slaves is on the broadcast ARP request sent out via the active slave. It is possible that some switch or network configurations may result in situations wherein the backup slaves do not receive the ARP requests; in such a situation, validation of backup slaves must be disabled.

The validation of ARP requests on backup slaves is mainly helping bonding to decide which slaves are more likely to work in case of the active slave failure, it doesn't really guarantee that the backup slave will work if it's selected as the next active slave.

Validation is useful in network configurations in which multiple bonding hosts are concurrently issuing ARPs to one or more targets beyond a common switch. Should the link between the switch and target fail (but not the switch itself), the probe traffic generated by the multiple bonding instances will fool the standard ARP monitor into considering the links as still up. Use of validation can resolve this, as the ARP monitor will only consider ARP requests and replies associated with its own instance of bonding.

Filtering:

Enabling filtering causes the ARP monitor to only use incoming ARP packets for link availability purposes. Arriving packets that are not ARPs are delivered normally, but do not count when determining if a slave is available.

Filtering operates by only considering the reception of ARP packets (any ARP packet, regardless of source or destination) when determining if a slave has received traffic for link availability purposes.

Filtering is useful in network configurations in which significant levels of third party broadcast traffic would fool the standard ARP monitor into considering the links as still up. Use of filtering can resolve this, as only ARP traffic is considered for link availability purposes.

This option was added in bonding version 3.1.0.

`arp_all_targets`

Specifies the quantity of `arp_ip_targets` that must be reachable in order for the ARP monitor to consider a slave as being up. This option affects only active-backup mode for slaves with `arp_validation` enabled.

Possible values are:

`any` or `0`

 consider the slave up only when any of the `arp_ip_targets` is reachable

`all` or `1`

 consider the slave up only when all of the `arp_ip_targets` are reachable

`arp_missed_max`

Specifies the number of `arp_interval` monitor checks that must fail in order for an interface to be marked down by the ARP monitor.

In order to provide orderly failover semantics, backup interfaces are permitted an extra monitor check (i.e., they must fail `arp_missed_max + 1` times before being marked down).

The default value is 2, and the allowable range is 1 - 255.

`downdelay`

Specifies the time, in milliseconds, to wait before disabling a slave after a link failure has been detected. This option is only valid for the `miimon` link monitor. The down-

delay value should be a multiple of the miimon value; if not, it will be rounded down to the nearest multiple. The default value is 0.

fail_over_mac

Specifies whether active-backup mode should set all slaves to the same MAC address at enslavement (the traditional behavior), or, when enabled, perform special handling of the bond's MAC address in accordance with the selected policy.

Possible values are:

none or 0

This setting disables fail_over_mac, and causes bonding to set all slaves of an active-backup bond to the same MAC address at enslavement time. This is the default.

active or 1

The "active" fail_over_mac policy indicates that the MAC address of the bond should always be the MAC address of the currently active slave. The MAC address of the slaves is not changed; instead, the MAC address of the bond changes during a failover.

This policy is useful for devices that cannot ever alter their MAC address, or for devices that refuse incoming broadcasts with their own source MAC (which interferes with the ARP monitor).

The down side of this policy is that every device on the network must be updated via gratuitous ARP, vs. just updating a switch or set of switches (which often takes place for any traffic, not just ARP traffic, if the switch snoops incoming traffic to update its tables) for the traditional method. If the gratuitous ARP is lost, communication may be disrupted.

When this policy is used in conjunction with the mii monitor, devices which assert link up prior to being able to actually transmit and receive are particularly susceptible to loss of the gratuitous ARP, and an appropriate updelay setting may be required.

follow or 2

The "follow" fail_over_mac policy causes the MAC address of the bond to be selected normally (normally the MAC address of the first slave added to the bond). However, the second and subsequent slaves are not set to this MAC address while they are in a backup role; a slave is programmed with the bond's MAC address at failover time (and the formerly active slave receives the newly active slave's MAC address).

This policy is useful for multiport devices that either become confused or incur a performance penalty when multiple ports are programmed with the same MAC address.

The default policy is none, unless the first slave cannot change its MAC address, in which case the active policy is selected by default.

This option may be modified via sysfs only when no slaves are present in the bond.

This option was added in bonding version 3.2.0. The "follow" policy was added in bonding version 3.3.0.

lacp_active

Option specifying whether to send LACPDU frames periodically.

off or 0

LACPDU frames acts as "speak when spoken to".

on or 1

LACPDU frames are sent along the configured links periodically. See `lacp_rate` for more details.

The default is on.

lacp_rate

Option specifying the rate in which we'll ask our link partner to transmit LACPDU packets in 802.3ad mode. Possible values are:

slow or 0

Request partner to transmit LACPDUs every 30 seconds

fast or 1

Request partner to transmit LACPDUs every 1 second

The default is slow.

max_bonds

Specifies the number of bonding devices to create for this instance of the bonding driver. E.g., if `max_bonds` is 3, and the bonding driver is not already loaded, then `bond0`, `bond1` and `bond2` will be created. The default value is 1. Specifying a value of 0 will load bonding, but will not create any devices.

miimon

Specifies the MII link monitoring frequency in milliseconds. This determines how often the link state of each slave is inspected for link failures. A value of zero disables MII link monitoring. A value of 100 is a good starting point. The `use_carrier` option, below, affects how the link state is determined. See the High Availability section for additional information. The default value is 100 if `arp_interval` is not set.

min_links

Specifies the minimum number of links that must be active before asserting carrier. It is similar to the Cisco EtherChannel min-links feature. This allows setting the minimum number of member ports that must be up (link-up state) before marking the bond device as up (carrier on). This is useful for situations where higher level services such as clustering want to ensure a minimum number of low bandwidth links are active before switchover. This option only affect 802.3ad mode.

The default value is 0. This will cause carrier to be asserted (for 802.3ad mode) whenever there is an active aggregator, regardless of the number of available links in that aggregator. Note that, because an aggregator cannot be active without at least one available link, setting this option to 0 or to 1 has the exact same effect.

mode

Specifies one of the bonding policies. The default is `balance-rr` (round robin). Possible values are:

`balance-rr` or `0`

Round-robin policy: Transmit packets in sequential order from the first available slave through the last. This mode provides load balancing and fault tolerance.

active-backup or 1

Active-backup policy: Only one slave in the bond is active. A different slave becomes active if, and only if, the active slave fails. The bond's MAC address is externally visible on only one port (network adapter) to avoid confusing the switch.

In bonding version 2.6.2 or later, when a failover occurs in active-backup mode, bonding will issue one or more gratuitous ARPs on the newly active slave. One gratuitous ARP is issued for the bonding master interface and each VLAN interfaces configured above it, provided that the interface has at least one IP address configured. Gratuitous ARPs issued for VLAN interfaces are tagged with the appropriate VLAN id.

This mode provides fault tolerance. The primary option, documented below, affects the behavior of this mode.

balance-xor or 2

XOR policy: Transmit based on the selected transmit hash policy. The default policy is a simple [(source MAC address XOR'd with destination MAC address XOR packet type ID) modulo slave count]. Alternate transmit policies may be selected via the xmit_hash_policy option, described below.

This mode provides load balancing and fault tolerance.

broadcast or 3

Broadcast policy: transmits everything on all slave interfaces. This mode provides fault tolerance.

802.3ad or 4

IEEE 802.3ad Dynamic link aggregation. Creates aggregation groups that share the same speed and duplex settings. Utilizes all slaves in the active aggregator according to the 802.3ad specification.

Slave selection for outgoing traffic is done according to the transmit hash policy, which may be changed from the default simple XOR policy via the xmit_hash_policy option, documented below. Note that not all transmit policies may be 802.3ad compliant, particularly in regards to the packet misordering requirements of section 43.2.4 of the 802.3ad standard. Differing peer implementations will have varying tolerances for noncompliance.

Prerequisites:

1. Ethtool support in the base drivers for retrieving the speed and duplex of each slave.
2. A switch that supports IEEE 802.3ad Dynamic link aggregation.

Most switches will require some type of configuration to enable 802.3ad mode.

balance-tlb or 5

Adaptive transmit load balancing: channel bonding that does not require any special switch support.

In `tlb_dynamic_lb=1` mode; the outgoing traffic is distributed according to the current load (computed relative to the speed) on each slave.

In `tlb_dynamic_lb=0` mode; the load balancing based on current load is disabled and the load is distributed only using the hash distribution.

Incoming traffic is received by the current slave. If the receiving slave fails, another slave takes over the MAC address of the failed receiving slave.

Prerequisite:

Ethtool support in the base drivers for retrieving the speed of each slave.

`balance-alb` or 6

Adaptive load balancing: includes balance-tlb plus receive load balancing (rlb) for IPV4 traffic, and does not require any special switch support. The receive load balancing is achieved by ARP negotiation. The bonding driver intercepts the ARP Replies sent by the local system on their way out and overwrites the source hardware address with the unique hardware address of one of the slaves in the bond such that different peers use different hardware addresses for the server.

Receive traffic from connections created by the server is also balanced. When the local system sends an ARP Request the bonding driver copies and saves the peer's IP information from the ARP packet. When the ARP Reply arrives from the peer, its hardware address is retrieved and the bonding driver initiates an ARP reply to this peer assigning it to one of the slaves in the bond. A problematic outcome of using ARP negotiation for balancing is that each time that an ARP request is broadcast it uses the hardware address of the bond. Hence, peers learn the hardware address of the bond and the balancing of receive traffic collapses to the current slave. This is handled by sending updates (ARP Replies) to all the peers with their individually assigned hardware address such that the traffic is redistributed. Receive traffic is also redistributed when a new slave is added to the bond and when an inactive slave is re-activated. The receive load is distributed sequentially (round robin) among the group of highest speed slaves in the bond.

When a link is reconnected or a new slave joins the bond the receive traffic is redistributed among all active slaves in the bond by initiating ARP Replies with the selected MAC address to each of the clients. The updelay parameter (detailed below) must be set to a value equal or greater than the switch's forwarding delay so that the ARP Replies sent to the peers will not be blocked by the switch.

Prerequisites:

1. Ethtool support in the base drivers for retrieving the speed of each slave.
2. Base driver support for setting the hardware address of a device while it is open. This is required so that there will always be one slave in the team using the bond hardware address (the `curr_active_slave`) while having a unique hardware address for each slave in the bond. If the `curr_active_slave` fails

its hardware address is swapped with the new curr_active_slave that was chosen.

`num_grat_arp, num_unsol_na`

Specify the number of peer notifications (gratuitous ARPs and unsolicited IPv6 Neighbor Advertisements) to be issued after a failover event. As soon as the link is up on the new slave (possibly immediately) a peer notification is sent on the bonding device and each VLAN sub-device. This is repeated at the rate specified by peer_notif_delay if the number is greater than 1.

The valid range is 0 - 255; the default value is 1. These options affect only the active-backup mode. These options were added for bonding versions 3.3.0 and 3.4.0 respectively.

From Linux 3.0 and bonding version 3.7.1, these notifications are generated by the ipv4 and ipv6 code and the numbers of repetitions cannot be set independently.

`packets_per_slave`

Specify the number of packets to transmit through a slave before moving to the next one. When set to 0 then a slave is chosen at random.

The valid range is 0 - 65535; the default value is 1. This option has effect only in balance-rr mode.

`peer_notif_delay`

Specify the delay, in milliseconds, between each peer notification (gratuitous ARP and unsolicited IPv6 Neighbor Advertisement) when they are issued after a failover event. This delay should be a multiple of the MII link monitor interval (miimon).

The valid range is 0 - 300000. The default value is 0, which means to match the value of the MII link monitor interval.

prio

Slave priority. A higher number means higher priority. The primary slave has the highest priority. This option also follows the primary_reselect rules.

This option could only be configured via netlink, and is only valid for active-backup(1), balance-tlb (5) and balance-alb (6) mode. The valid value range is a signed 32 bit integer.

The default value is 0.

`primary`

A string (eth0, eth2, etc) specifying which slave is the primary device. The specified device will always be the active slave while it is available. Only when the primary is off-line will alternate devices be used. This is useful when one slave is preferred over another, e.g., when one slave has higher throughput than another.

The primary option is only valid for active-backup(1), balance-tlb (5) and balance-alb (6) mode.

`primary_reselect`

Specifies the reselection policy for the primary slave. This affects how the primary slave is chosen to become the active slave when failure of the active slave or recovery of the primary slave occurs. This option is designed to prevent flip-flopping between the primary slave and other slaves. Possible values are:

always or 0 (default)

The primary slave becomes the active slave whenever it comes back up.

better or 1

The primary slave becomes the active slave when it comes back up, if the speed and duplex of the primary slave is better than the speed and duplex of the current active slave.

failure or 2

The primary slave becomes the active slave only if the current active slave fails and the primary slave is up.

The `primary_reselect` setting is ignored in two cases:

If no slaves are active, the first slave to recover is made the active slave.

When initially enslaved, the primary slave is always made the active slave.

Changing the `primary_reselect` policy via sysfs will cause an immediate selection of the best active slave according to the new policy. This may or may not result in a change of the active slave, depending upon the circumstances.

This option was added for bonding version 3.6.0.

`tlb_dynamic_lb`

Specifies if dynamic shuffling of flows is enabled in tlb or alb mode. The value has no effect on any other modes.

The default behavior of tlb mode is to shuffle active flows across slaves based on the load in that interval. This gives nice lb characteristics but can cause packet reordering. If re-ordering is a concern use this variable to disable flow shuffling and rely on load balancing provided solely by the hash distribution. `xmit-hash-policy` can be used to select the appropriate hashing for the setup.

The sysfs entry can be used to change the setting per bond device and the initial value is derived from the module parameter. The sysfs entry is allowed to be changed only if the bond device is down.

The default value is "1" that enables flow shuffling while value "0" disables it. This option was added in bonding driver 3.7.1

`updelay`

Specifies the time, in milliseconds, to wait before enabling a slave after a link recovery has been detected. This option is only valid for the miimon link monitor. The updelay value should be a multiple of the miimon value; if not, it will be rounded down to the nearest multiple. The default value is 0.

`use_carrier`

Specifies whether or not miimon should use MII or ETHTOOL ioctls vs. `netif_carrier_ok()` to determine the link status. The MII or ETHTOOL ioctls are less efficient and utilize a deprecated calling sequence within the kernel. The `netif_carrier_ok()` relies on the device driver to maintain its state with `netif_carrier_on/off`; at this writing, most, but not all, device drivers support this facility.

If bonding insists that the link is up when it should not be, it may be that your network device driver does not support netif_carrier_on/off. The default state for netif_carrier is "carrier on," so if a driver does not support netif_carrier, it will appear as if the link is always up. In this case, setting use_carrier to 0 will cause bonding to revert to the MII / ETHTOOL ioctl method to determine the link state.

A value of 1 enables the use of `netif_carrier_ok()`, a value of 0 will use the deprecated MII / ETHTOOL ioctls. The default value is 1.

xmit_hash_policy

Selects the transmit hash policy to use for slave selection in balance-xor, 802.3ad, and tlb modes. Possible values are:

layer2

Uses XOR of hardware MAC addresses and packet type ID field to generate the hash. The formula is

$$\text{hash} = \text{source MAC[5]} \text{ XOR } \text{destination MAC[5]} \text{ XOR } \text{packet type ID}$$

$$\text{slave number} = \text{hash modulo slave count}$$

This algorithm will place all traffic to a particular network peer on the same slave.

This algorithm is 802.3ad compliant.

layer2+3

This policy uses a combination of layer2 and layer3 protocol information to generate the hash.

Uses XOR of hardware MAC addresses and IP addresses to generate the hash. The formula is

$$\text{hash} = \text{source MAC[5]} \text{ XOR } \text{destination MAC[5]} \text{ XOR } \text{packet type ID}$$

$$\text{hash} = \text{hash XOR source IP} \text{ XOR } \text{destination IP}$$

$$\text{hash} = \text{hash XOR (hash RSHIFT 16)}$$

$$\text{hash} = \text{hash XOR (hash RSHIFT 8)}$$

And then hash is reduced modulo slave count.

If the protocol is IPv6 then the source and destination addresses are first hashed using `ipv6_addr_hash`.

This algorithm will place all traffic to a particular network peer on the same slave. For non-IP traffic, the formula is the same as for the layer2 transmit hash policy.

This policy is intended to provide a more balanced distribution of traffic than layer2 alone, especially in environments where a layer3 gateway device is required to reach most destinations.

This algorithm is 802.3ad compliant.

layer3+4

This policy uses upper layer protocol information, when available, to generate the hash. This allows for traffic to a particular network peer to span multiple slaves, although a single connection will not span multiple slaves.

The formula for unfragmented TCP and UDP packets is

hash = source port, destination port (as in the header) hash = hash XOR source IP XOR destination IP hash = hash XOR (hash RSHIFT 16) hash = hash XOR (hash RSHIFT 8) hash = hash RSHIFT 1 And then hash is reduced modulo slave count.

If the protocol is IPv6 then the source and destination addresses are first hashed using ipv6_addr_hash.

For fragmented TCP or UDP packets and all other IPv4 and IPv6 protocol traffic, the source and destination port information is omitted. For non-IP traffic, the formula is the same as for the layer2 transmit hash policy.

This algorithm is not fully 802.3ad compliant. A single TCP or UDP conversation containing both fragmented and unfragmented packets will see packets striped across two interfaces. This may result in out of order delivery. Most traffic types will not meet this criteria, as TCP rarely fragments traffic, and most UDP traffic is not involved in extended conversations. Other implementations of 802.3ad may or may not tolerate this noncompliance.

encap2+3

This policy uses the same formula as layer2+3 but it relies on skb_flow_dissect to obtain the header fields which might result in the use of inner headers if an encapsulation protocol is used. For example this will improve the performance for tunnel users because the packets will be distributed according to the encapsulated flows.

encap3+4

This policy uses the same formula as layer3+4 but it relies on skb_flow_dissect to obtain the header fields which might result in the use of inner headers if an encapsulation protocol is used. For example this will improve the performance for tunnel users because the packets will be distributed according to the encapsulated flows.

vlan+srcmac

This policy uses a very rudimentary vlan ID and source mac hash to load-balance traffic per-vlan, with failover should one leg fail. The intended use case is for a bond shared by multiple virtual machines, all configured to use their own vlan, to give lacp-like functionality without requiring lacp-capable switching hardware.

The formula for the hash is simply

hash = (vlan ID) XOR (source MAC vendor) XOR (source MAC dev)

The default value is layer2. This option was added in bonding version 2.6.3. In earlier versions of bonding, this parameter does not exist, and the layer2 policy is the only policy. The layer2+3 value was added for bonding version 3.2.2.

resend_igmp

Specifies the number of IGMP membership reports to be issued after a failover event. One membership report is issued immediately after the failover, subsequent packets are sent in each 200ms interval.

The valid range is 0 - 255; the default value is 1. A value of 0 prevents the IGMP membership report from being issued in response to the failover event.

This option is useful for bonding modes balance-rr (0), active-backup (1), balance-tlb (5) and balance-alb (6), in which a failover can switch the IGMP traffic from one slave to another. Therefore a fresh IGMP report must be issued to cause the switch to forward the incoming IGMP traffic over the newly selected slave.

This option was added for bonding version 3.7.0.

lp_interval

Specifies the number of seconds between instances where the bonding driver sends learning packets to each slaves peer switch.

The valid range is 1 - 0x7fffffff; the default value is 1. This Option has effect only in balance-tlb and balance-alb modes.

37.4 3. Configuring Bonding Devices

You can configure bonding using either your distro's network initialization scripts, or manually using either iproute2 or the sysfs interface. Distros generally use one of three packages for the network initialization scripts: initscripts, sysconfig or interfaces. Recent versions of these packages have support for bonding, while older versions do not.

We will first describe the options for configuring bonding for distros using versions of initscripts, sysconfig and interfaces with full or partial support for bonding, then provide information on enabling bonding without support from the network initialization scripts (i.e., older versions of initscripts or sysconfig).

If you're unsure whether your distro uses sysconfig, initscripts or interfaces, or don't know if it's new enough, have no fear. Determining this is fairly straightforward.

First, look for a file called interfaces in /etc/network directory. If this file is present in your system, then your system use interfaces. See Configuration with Interfaces Support.

Else, issue the command:

```
$ rpm -qf /sbin/ifup
```

It will respond with a line of text starting with either "initscripts" or "sysconfig," followed by some numbers. This is the package that provides your network initialization scripts.

Next, to determine if your installation supports bonding, issue the command:

```
$ grep ifenslave /sbin/ifup
```

If this returns any matches, then your initscripts or sysconfig has support for bonding.

37.4.1 3.1 Configuration with Sysconfig Support

This section applies to distros using a version of sysconfig with bonding support, for example, SuSE Linux Enterprise Server 9.

SuSE SLES 9's networking configuration system does support bonding, however, at this writing, the YaST system configuration front end does not provide any means to work with bonding devices. Bonding devices can be managed by hand, however, as follows.

First, if they have not already been configured, configure the slave devices. On SLES 9, this is most easily done by running the `yast2 sysconfig` configuration utility. The goal is for to create an `ifcfg-id` file for each slave device. The simplest way to accomplish this is to configure the devices for DHCP (this is only to get the file `ifcfg-id` file created; see below for some issues with DHCP). The name of the configuration file for each device will be of the form:

```
ifcfg-id-xx:xx:xx:xx:xx:xx
```

Where the "xx" portion will be replaced with the digits from the device's permanent MAC address.

Once the set of `ifcfg-id-xx:xx:xx:xx:xx:xx` files has been created, it is necessary to edit the configuration files for the slave devices (the MAC addresses correspond to those of the slave devices). Before editing, the file will contain multiple lines, and will look something like this:

```
BOOTPROTO='dhcp'
STARTMODE='on'
USERCTL='no'
UNIQUE='XNzu.WeZG0GF+4wE'
_nm_name='bus-pci-0001:61:01.0'
```

Change the `BOOTPROTO` and `STARTMODE` lines to the following:

```
BOOTPROTO='none'
STARTMODE='off'
```

Do not alter the `UNIQUE` or `_nm_name` lines. Remove any other lines (`USERCTL`, etc).

Once the `ifcfg-id-xx:xx:xx:xx:xx:xx` files have been modified, it's time to create the configuration file for the bonding device itself. This file is named `ifcfg-bondX`, where X is the number of the bonding device to create, starting at 0. The first such file is `ifcfg-bond0`, the second is `ifcfg-bond1`, and so on. The sysconfig network configuration system will correctly start multiple instances of bonding.

The contents of the `ifcfg-bondX` file is as follows:

```
BOOTPROTO="static"
BROADCAST="10.0.2.255"
IPADDR="10.0.2.10"
NETMASK="255.255.0.0"
NETWORK="10.0.2.0"
REMOTE_IPADDR=""
STARTMODE="onboot"
BONDING_MASTER="yes"
BONDING_MODULE_OPTS="mode=active-backup miimon=100"
```

```
BONDING_SLAVE0="eth0"
BONDING_SLAVE1="bus-pci-0000:06:08.1"
```

Replace the sample BROADCAST, IPADDR, NETMASK and NETWORK values with the appropriate values for your network.

The STARTMODE specifies when the device is brought online. The possible values are:

onboot	The device is started at boot time. If you're not sure, this is probably what you want.
manual	The device is started only when ifup is called manually. Bonding devices may be configured this way if you do not wish them to start automatically at boot for some reason.
hotplug	The device is started by a hotplug event. This is not a valid choice for a bonding device.
off or	The device configuration is ignored.
ignore	

The line BONDING_MASTER='yes' indicates that the device is a bonding master device. The only useful value is "yes."

The contents of BONDING_MODULE_OPTS are supplied to the instance of the bonding module for this device. Specify the options for the bonding mode, link monitoring, and so on here. Do not include the max_bonds bonding parameter; this will confuse the configuration system if you have multiple bonding devices.

Finally, supply one BONDING_SLAVE n =“slave device” for each slave, where “ n ” is an increasing value, one for each slave. The “slave device” is either an interface name, e.g., “eth0”, or a device specifier for the network device. The interface name is easier to find, but the ethN names are subject to change at boot time if, e.g., a device early in the sequence has failed. The device specifiers (bus-pci-0000:06:08.1 in the example above) specify the physical network device, and will not change unless the device's bus location changes (for example, it is moved from one PCI slot to another). The example above uses one of each type for demonstration purposes; most configurations will choose one or the other for all slave devices.

When all configuration files have been modified or created, networking must be restarted for the configuration changes to take effect. This can be accomplished via the following:

```
# /etc/init.d/network restart
```

Note that the network control script (/sbin/ifdown) will remove the bonding module as part of the network shutdown processing, so it is not necessary to remove the module by hand if, e.g., the module parameters have changed.

Also, at this writing, YaST/YaST2 will not manage bonding devices (they do not show bonding interfaces on its list of network devices). It is necessary to edit the configuration file by hand to change the bonding configuration.

Additional general options and details of the ifcfg file format can be found in an example ifcfg template file:

```
/etc/sysconfig/network/ifcfg.template
```

Note that the template does not document the various `BONDING_*` settings described above, but does describe many of the other options.

37.4.2 3.1.1 Using DHCP with Sysconfig

Under sysconfig, configuring a device with `BOOTPROTO='dhcp'` will cause it to query DHCP for its IP address information. At this writing, this does not function for bonding devices; the scripts attempt to obtain the device address from DHCP prior to adding any of the slave devices. Without active slaves, the DHCP requests are not sent to the network.

37.4.3 3.1.2 Configuring Multiple Bonds with Sysconfig

The sysconfig network initialization system is capable of handling multiple bonding devices. All that is necessary is for each bonding instance to have an appropriately configured `ifcfg-bondX` file (as described above). Do not specify the "max_bonds" parameter to any instance of bonding, as this will confuse sysconfig. If you require multiple bonding devices with identical parameters, create multiple `ifcfg-bondX` files.

Because the sysconfig scripts supply the bonding module options in the `ifcfg-bondX` file, it is not necessary to add them to the system `/etc/modules.d/* .conf` configuration files.

37.4.4 3.2 Configuration with Initscripts Support

This section applies to distros using a recent version of initscripts with bonding support, for example, Red Hat Enterprise Linux version 3 or later, Fedora, etc. On these systems, the network initialization scripts have knowledge of bonding, and can be configured to control bonding devices. Note that older versions of the initscripts package have lower levels of support for bonding; this will be noted where applicable.

These distros will not automatically load the network adapter driver unless the `ethX` device is configured with an IP address. Because of this constraint, users must manually configure a network-script file for all physical adapters that will be members of a `bondX` link. Network script files are located in the directory:

`/etc/sysconfig/network-scripts`

The file name must be prefixed with "ifcfg-eth" and suffixed with the adapter's physical adapter number. For example, the script for `eth0` would be named `/etc/sysconfig/network-scripts/ifcfg-eth0`. Place the following text in the file:

```
DEVICE=eth0
USERCTL=no
ONBOOT=yes
MASTER=bond0
SLAVE=yes
BOOTPROTO=none
```

The `DEVICE=` line will be different for every `ethX` device and must correspond with the name of the file, i.e., `ifcfg-eth1` must have a device line of `DEVICE=eth1`. The setting of the `MASTER=` line will also depend on the final bonding interface name chosen for your bond. As with other network devices, these typically start at 0, and go up one for each device, i.e., the first bonding instance is `bond0`, the second is `bond1`, and so on.

Next, create a bond network script. The file name for this script will be /etc/sysconfig/network-scripts/ifcfg-bondX where X is the number of the bond. For bond0 the file is named "ifcfg-bond0", for bond1 it is named "ifcfg-bond1", and so on. Within that file, place the following text:

```
DEVICE=bond0
IPADDR=192.168.1.1
NETMASK=255.255.255.0
NETWORK=192.168.1.0
BROADCAST=192.168.1.255
ONBOOT=yes
BOOTPROTO=none
USERCTL=no
```

Be sure to change the networking specific lines (IPADDR, NETMASK, NETWORK and BROADCAST) to match your network configuration.

For later versions of initscripts, such as that found with Fedora 7 (or later) and Red Hat Enterprise Linux version 5 (or later), it is possible, and, indeed, preferable, to specify the bonding options in the ifcfg-bond0 file, e.g. a line of the format:

```
BONDING_OPTS="mode=active-backup arp_interval=60 arp_ip_target=192.168.1.254"
```

will configure the bond with the specified options. The options specified in BONDING_OPTS are identical to the bonding module parameters except for the arp_ip_target field when using versions of initscripts older than 8.57 (Fedora 8) and 8.45.19 (Red Hat Enterprise Linux 5.2). When using older versions each target should be included as a separate option and should be preceded by a '+' to indicate it should be added to the list of queried targets, e.g.:

```
arp_ip_target=+192.168.1.1 arp_ip_target=+192.168.1.2
```

is the proper syntax to specify multiple targets. When specifying options via BONDING_OPTS, it is not necessary to edit /etc/modprobe.d/*.conf.

For even older versions of initscripts that do not support BONDING_OPTS, it is necessary to edit /etc/modprobe.d/.conf, depending upon your distro) to load the bonding module with your desired options when the bond0 interface is brought up. The following lines in /etc/modprobe.d/.conf will load the bonding module, and select its options:

```
alias bond0 bonding options bond0 mode=balance-alb miimon=100
```

Replace the sample parameters with the appropriate set of options for your configuration.

Finally run "/etc/rc.d/init.d/network restart" as root. This will restart the networking subsystem and your bond link should be now up and running.

37.4.5 3.2.1 Using DHCP with Initscripts

Recent versions of initscripts (the versions supplied with Fedora Core 3 and Red Hat Enterprise Linux 4, or later versions, are reported to work) have support for assigning IP information to bonding devices via DHCP.

To configure bonding for DHCP, configure it as described above, except replace the line "BOOTPROTO=none" with "BOOTPROTO=dhcp" and add a line consisting of "TYPE=Bonding". Note that the TYPE value is case sensitive.

37.4.6 3.2.2 Configuring Multiple Bonds with Initscripts

Initscripts packages that are included with Fedora 7 and Red Hat Enterprise Linux 5 support multiple bonding interfaces by simply specifying the appropriate BONDING_OPTS= in ifcfg-bondX where X is the number of the bond. This support requires sysfs support in the kernel, and a bonding driver of version 3.0.0 or later. Other configurations may not support this method for specifying multiple bonding interfaces; for those instances, see the "Configuring Multiple Bonds Manually" section, below.

37.4.7 3.3 Configuring Bonding Manually with iproute2

This section applies to distros whose network initialization scripts (the sysconfig or initscripts package) do not have specific knowledge of bonding. One such distro is SuSE Linux Enterprise Server version 8.

The general method for these systems is to place the bonding module parameters into a config file in /etc/modprobe.d/ (as appropriate for the installed distro), then add modprobe and/or *ip link* commands to the system's global init script. The name of the global init script differs; for sysconfig, it is /etc/init.d/boot.local and for initscripts it is /etc/rc.d/rc.local.

For example, if you wanted to make a simple bond of two e100 devices (presumed to be eth0 and eth1), and have it persist across reboots, edit the appropriate file (/etc/init.d/boot.local or /etc/rc.d/rc.local), and add the following:

```
modprobe bonding mode=balance-alb miimon=100
modprobe e100
ifconfig bond0 192.168.1.1 netmask 255.255.255.0 up
ip link set eth0 master bond0
ip link set eth1 master bond0
```

Replace the example bonding module parameters and bond0 network configuration (IP address, netmask, etc) with the appropriate values for your configuration.

Unfortunately, this method will not provide support for the ifup and ifdown scripts on the bond devices. To reload the bonding configuration, it is necessary to run the initialization script, e.g.:

```
# /etc/init.d/boot.local
```

or:

```
# /etc/rc.d/rc.local
```

It may be desirable in such a case to create a separate script which only initializes the bonding configuration, then call that separate script from within boot.local. This allows for bonding to be enabled without re-running the entire global init script.

To shut down the bonding devices, it is necessary to first mark the bonding device itself as being down, then remove the appropriate device driver modules. For our example above, you can do the following:

```
# ifconfig bond0 down
# rmmod bonding
# rmmod e100
```

Again, for convenience, it may be desirable to create a script with these commands.

37.4.8 3.3.1 Configuring Multiple Bonds Manually

This section contains information on configuring multiple bonding devices with differing options for those systems whose network initialization scripts lack support for configuring multiple bonds.

If you require multiple bonding devices, but all with the same options, you may wish to use the "max_bonds" module parameter, documented above.

To create multiple bonding devices with differing options, it is preferable to use bonding parameters exported by sysfs, documented in the section below.

For versions of bonding without sysfs support, the only means to provide multiple instances of bonding with differing options is to load the bonding driver multiple times. Note that current versions of the sysconfig network initialization scripts handle this automatically; if your distro uses these scripts, no special action is needed. See the section Configuring Bonding Devices, above, if you're not sure about your network initialization scripts.

To load multiple instances of the module, it is necessary to specify a different name for each instance (the module loading system requires that every loaded module, even multiple instances of the same module, have a unique name). This is accomplished by supplying multiple sets of bonding options in `/etc/modprobe.d/*.conf`, for example:

```
alias bond0 bonding
options bond0 -o bond0 mode=balance-rr miimon=100

alias bond1 bonding
options bond1 -o bond1 mode=balance-alb miimon=50
```

will load the bonding module two times. The first instance is named "bond0" and creates the bond0 device in balance-rr mode with an miimon of 100. The second instance is named "bond1" and creates the bond1 device in balance-alb mode with an miimon of 50.

In some circumstances (typically with older distributions), the above does not work, and the second bonding instance never sees its options. In that case, the second options line can be substituted as follows:

```
install bond1 /sbin/modprobe --ignore-install bonding -o bond1 \
mode=balance-alb miimon=50
```

This may be repeated any number of times, specifying a new and unique name in place of bond1 for each subsequent instance.

It has been observed that some Red Hat supplied kernels are unable to rename modules at load time (the “-o bond1” part). Attempts to pass that option to modprobe will produce an “Operation not permitted” error. This has been reported on some Fedora Core kernels, and has been seen on RHEL 4 as well. On kernels exhibiting this problem, it will be impossible to configure multiple bonds with differing parameters (as they are older kernels, and also lack sysfs support).

37.4.9 3.4 Configuring Bonding Manually via Sysfs

Starting with version 3.0.0, Channel Bonding may be configured via the sysfs interface. This interface allows dynamic configuration of all bonds in the system without unloading the module. It also allows for adding and removing bonds at runtime. Ifenslave is no longer required, though it is still supported.

Use of the sysfs interface allows you to use multiple bonds with different configurations without having to reload the module. It also allows you to use multiple, differently configured bonds when bonding is compiled into the kernel.

You must have the sysfs filesystem mounted to configure bonding this way. The examples in this document assume that you are using the standard mount point for sysfs, e.g. /sys. If your sysfs filesystem is mounted elsewhere, you will need to adjust the example paths accordingly.

37.4.10 Creating and Destroying Bonds

To add a new bond foo:

```
# echo +foo > /sys/class/net/bonding_masters
```

To remove an existing bond bar:

```
# echo -bar > /sys/class/net/bonding_masters
```

To show all existing bonds:

```
# cat /sys/class/net/bonding_masters
```

Note: due to 4K size limitation of sysfs files, this list may be truncated if you have more than a few hundred bonds. This is unlikely to occur under normal operating conditions.

37.4.11 Adding and Removing Slaves

Interfaces may be enslaved to a bond using the file /sys/class/net/<bond>/bonding/slaves. The semantics for this file are the same as for the bonding_masters file.

To enslave interface eth0 to bond bond0:

```
# ifconfig bond0 up
# echo +eth0 > /sys/class/net/bond0/bonding/slaves
```

To free slave eth0 from bond bond0:

```
# echo -eth0 > /sys/class/net/bond0/bonding/slaves
```

When an interface is enslaved to a bond, symlinks between the two are created in the sysfs filesystem. In this case, you would get /sys/class/net/bond0/slave_eth0 pointing to /sys/class/net/eth0, and /sys/class/net/eth0/master pointing to /sys/class/net/bond0.

This means that you can tell quickly whether or not an interface is enslaved by looking for the master symlink. Thus: # echo -eth0 > /sys/class/net/eth0/master/bonding/slaves will free eth0 from whatever bond it is enslaved to, regardless of the name of the bond interface.

37.4.12 Changing a Bond's Configuration

Each bond may be configured individually by manipulating the files located in /sys/class/net/<bond name>/bonding

The names of these files correspond directly with the command-line parameters described elsewhere in this file, and, with the exception of arp_ip_target, they accept the same values. To see the current setting, simply cat the appropriate file.

A few examples will be given here; for specific usage guidelines for each parameter, see the appropriate section in this document.

To configure bond0 for balance-alb mode:

```
# ifconfig bond0 down
# echo 6 > /sys/class/net/bond0/bonding	mode
- or -
# echo balance-alb > /sys/class/net/bond0/bonding	mode
```

Note: The bond interface must be down before the mode can be changed.

To enable MII monitoring on bond0 with a 1 second interval:

```
# echo 1000 > /sys/class/net/bond0/bonding/miimon
```

Note: If ARP monitoring is enabled, it will be disabled when MII monitoring is enabled, and vice-versa.

To add ARP targets:

```
# echo +192.168.0.100 > /sys/class/net/bond0/bonding/arp_ip_target
# echo +192.168.0.101 > /sys/class/net/bond0/bonding/arp_ip_target
```

Note: up to 16 target addresses may be specified.

To remove an ARP target:

```
# echo -192.168.0.100 > /sys/class/net/bond0/bonding/arp_ip_target
```

To configure the interval between learning packet transmits:

```
# echo 12 > /sys/class/net/bond0/bonding/lp_interval
```

Note: the lp_interval is the number of seconds between instances where the bonding driver sends learning packets to each slaves peer switch. The default interval is 1 second.

37.4.13 Example Configuration

We begin with the same example that is shown in section 3.3, executed with sysfs, and without using ifenslave.

To make a simple bond of two e100 devices (presumed to be eth0 and eth1), and have it persist across reboots, edit the appropriate file (/etc/init.d/boot.local or /etc/rc.d/rc.local), and add the following:

```
modprobe bonding
modprobe e100
echo balance-alb > /sys/class/net/bond0/bonding	mode
ifconfig bond0 192.168.1.1 netmask 255.255.255.0 up
echo 100 > /sys/class/net/bond0/bonding/miimon
echo +eth0 > /sys/class/net/bond0/bonding/slaves
echo +eth1 > /sys/class/net/bond0/bonding/slaves
```

To add a second bond, with two e1000 interfaces in active-backup mode, using ARP monitoring, add the following lines to your init script:

```
modprobe e1000
echo +bond1 > /sys/class/net/bonding_masters
echo active-backup > /sys/class/net/bond1/bonding	mode
ifconfig bond1 192.168.2.1 netmask 255.255.255.0 up
echo +192.168.2.100 /sys/class/net/bond1/bonding/arp_ip_target
echo 2000 > /sys/class/net/bond1/bonding/arp_interval
echo +eth2 > /sys/class/net/bond1/bonding/slaves
echo +eth3 > /sys/class/net/bond1/bonding/slaves
```

37.4.14 3.5 Configuration with Interfaces Support

This section applies to distros which use /etc/network/interfaces file to describe network interface configuration, most notably Debian and its derivatives.

The ifup and ifdown commands on Debian don't support bonding out of the box. The ifenslave-2.6 package should be installed to provide bonding support. Once installed, this package will provide bond-* options to be used into /etc/network/interfaces.

Note that ifenslave-2.6 package will load the bonding module and use the ifenslave command when appropriate.

37.4.15 Example Configurations

In /etc/network/interfaces, the following stanza will configure bond0, in active-backup mode, with eth0 and eth1 as slaves:

```
auto bond0
iface bond0 inet dhcp
    bond-slaves eth0 eth1
    bond-mode active-backup
    bond-miimon 100
    bond-primary eth0 eth1
```

If the above configuration doesn't work, you might have a system using upstart for system startup. This is most notably true for recent Ubuntu versions. The following stanza in /etc/network/interfaces will produce the same result on those systems:

```
auto bond0
iface bond0 inet dhcp
    bond-slaves none
    bond-mode active-backup
    bond-miimon 100

auto eth0
iface eth0 inet manual
    bond-master bond0
    bond-primary eth0 eth1

auto eth1
iface eth1 inet manual
    bond-master bond0
    bond-primary eth0 eth1
```

For a full list of bond-* supported options in /etc/network/interfaces and some more advanced examples tailored to your particular distros, see the files in /usr/share/doc/ifenslave-2.6.

37.4.16 3.6 Overriding Configuration for Special Cases

When using the bonding driver, the physical port which transmits a frame is typically selected by the bonding driver, and is not relevant to the user or system administrator. The output port is simply selected using the policies of the selected bonding mode. On occasion however, it is helpful to direct certain classes of traffic to certain physical interfaces on output to implement slightly more complex policies. For example, to reach a web server over a bonded interface in which eth0 connects to a private network, while eth1 connects via a public network, it may be desirous to bias the bond to send said traffic over eth0 first, using eth1 only as a fall back, while all other traffic can safely be sent over either interface. Such configurations may be achieved using the traffic control utilities inherent in linux.

By default the bonding driver is multiqueue aware and 16 queues are created when the driver initializes (see [HOWTO for multiqueue network device support](#) for details). If more or less queues are desired the module parameter tx_queues can be used to change this value. There is no sysfs parameter available as the allocation is done at module init time.

The output of the file /proc/net/bonding/bondX has changed so the output Queue ID is now printed for each slave:

```
Bonding Mode: fault-tolerance (active-backup)
Primary Slave: None
Currently Active Slave: eth0
MII Status: up
MII Polling Interval (ms): 0
Up Delay (ms): 0
Down Delay (ms): 0

Slave Interface: eth0
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:1a:a0:12:8f:cb
Slave queue ID: 0

Slave Interface: eth1
MII Status: up
Link Failure Count: 0
Permanent HW addr: 00:1a:a0:12:8f:cc
Slave queue ID: 2
```

The queue_id for a slave can be set using the command:

```
# echo "eth1:2" > /sys/class/net/bond0/bonding/queue_id
```

Any interface that needs a queue_id set should set it with multiple calls like the one above until proper priorities are set for all interfaces. On distributions that allow configuration via initscripts, multiple 'queue_id' arguments can be added to BONDING_OPTS to set all needed slave queues.

These queue id's can be used in conjunction with the tc utility to configure a multiqueue qdisc and filters to bias certain traffic to transmit on certain slave devices. For instance, say we wanted, in the above configuration to force all traffic bound to 192.168.1.100 to use eth1 in the bond as its output device. The following commands would accomplish this:

```
# tc qdisc add dev bond0 handle 1 root multiq
# tc filter add dev bond0 protocol ip parent 1: prio 1 u32 match ip \
    dst 192.168.1.100 action skbredit queue_mapping 2
```

These commands tell the kernel to attach a multiqueue queue discipline to the bond0 interface and filter traffic enqueued to it, such that packets with a dst ip of 192.168.1.100 have their output queue mapping value overwritten to 2. This value is then passed into the driver, causing the normal output path selection policy to be overridden, selecting instead qid 2, which maps to eth1.

Note that qid values begin at 1. Qid 0 is reserved to initiate to the driver that normal output policy selection should take place. One benefit to simply leaving the qid for a slave to 0 is the multiqueue awareness in the bonding driver that is now present. This awareness allows tc filters to be placed on slave devices as well as bond devices and the bonding driver will simply act as a pass-through for selecting output queues on the slave device rather than output port selection.

This feature first appeared in bonding driver version 3.7.0 and support for output slave selection was limited to round-robin and active-backup modes.

37.4.17 3.7 Configuring LACP for 802.3ad mode in a more secure way

When using 802.3ad bonding mode, the Actor (host) and Partner (switch) exchange LACPDUs. These LACPDUs cannot be sniffed, because they are destined to link local mac addresses (which switches/bridges are not supposed to forward). However, most of the values are easily predictable or are simply the machine's MAC address (which is trivially known to all other hosts in the same L2). This implies that other machines in the L2 domain can spoof LACPDU packets from other hosts to the switch and potentially cause mayhem by joining (from the point of view of the switch) another machine's aggregate, thus receiving a portion of that host's incoming traffic and / or spoofing traffic from that machine themselves (potentially even successfully terminating some portion of flows). Though this is not a likely scenario, one could avoid this possibility by simply configuring few bonding parameters:

- (a) `ad_actor_system` : You can set a random mac-address that can be used for these LACPDU exchanges. The value can not be either NULL or Multicast. Also it's preferable to set the local-admin bit. Following shell code generates a random mac-address as described above:

```
# sys_mac_addr=$(printf '%02x:%02x:%02x:%02x:%02x:%02x' \
    $(( (RANDOM & 0xFE) | 0x02 )) \
    $(( RANDOM & 0xFF )))
# echo $sys_mac_addr > /sys/class/net/bond0/bonding/ad_actor_system
```

- (b) `ad_actor_sys_prio` : Randomize the system priority. The default value is 65535, but system can take the value from 1 - 65535. Following shell code generates random priority and sets it:

```
# sys_prio=$(( 1 + RANDOM + RANDOM ))
# echo $sys_prio > /sys/class/net/bond0/bonding/ad_actor_sys_prio
```

- (c) ad_user_port_key : Use the user portion of the port-key. The default keeps this empty. These are the upper 10 bits of the port-key and value ranges from 0 - 1023. Following shell code generates these 10 bits and sets it:

```
# usr_port_key=$(( RANDOM & 0x3FF ))
# echo $usr_port_key > /sys/class/net/bond0/bonding/ad_user_port_key
```

37.5 4 Querying Bonding Configuration

37.5.1 4.1 Bonding Configuration

Each bonding device has a read-only file residing in the /proc/net/bonding directory. The file contents include information about the bonding configuration, options and state of each slave.

For example, the contents of /proc/net/bonding/bond0 after the driver is loaded with parameters of mode=0 and miimon=1000 is generally as follows:

```
Ethernet Channel Bonding Driver: 2.6.1 (October 29, 2004)
Bonding Mode: load balancing (round-robin)
Currently Active Slave: eth0
MII Status: up
MII Polling Interval (ms): 1000
Up Delay (ms): 0
Down Delay (ms): 0

Slave Interface: eth1
MII Status: up
Link Failure Count: 1

Slave Interface: eth0
MII Status: up
Link Failure Count: 1
```

The precise format and contents will change depending upon the bonding configuration, state, and version of the bonding driver.

37.5.2 4.2 Network configuration

The network configuration can be inspected using the ifconfig command. Bonding devices will have the MASTER flag set; Bonding slave devices will have the SLAVE flag set. The ifconfig output does not contain information on which slaves are associated with which masters.

In the example below, the bond0 interface is the master (MASTER) while eth0 and eth1 are slaves (SLAVE). Notice all slaves of bond0 have the same MAC address (HWaddr) as bond0 for all modes except TLB and ALB that require a unique MAC address for each slave:

```
# /sbin/ifconfig
bond0      Link encap:Ethernet HWaddr 00:C0:F0:1F:37:B4
           inet addr:XXX.XXX.XXX.YYY Bcast:XXX.XXX.XXX.255 Mask:255.255.252.0
             UP BROADCAST RUNNING MASTER MULTICAST MTU:1500 Metric:1
             RX packets:7224794 errors:0 dropped:0 overruns:0 frame:0
             TX packets:3286647 errors:1 dropped:0 overruns:1 carrier:0
             collisions:0 txqueuelen:0

eth0       Link encap:Ethernet HWaddr 00:C0:F0:1F:37:B4
           UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500 Metric:1
           RX packets:3573025 errors:0 dropped:0 overruns:0 frame:0
           TX packets:1643167 errors:1 dropped:0 overruns:1 carrier:0
           collisions:0 txqueuelen:100
           Interrupt:10 Base address:0x1080

eth1       Link encap:Ethernet HWaddr 00:C0:F0:1F:37:B4
           UP BROADCAST RUNNING SLAVE MULTICAST MTU:1500 Metric:1
           RX packets:3651769 errors:0 dropped:0 overruns:0 frame:0
           TX packets:1643480 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:100
           Interrupt:9 Base address:0x1400
```

37.6 5. Switch Configuration

For this section, "switch" refers to whatever system the bonded devices are directly connected to (i.e., where the other end of the cable plugs into). This may be an actual dedicated switch device, or it may be another regular system (e.g., another computer running Linux),

The active-backup, balance-tlb and balance-alb modes do not require any specific configuration of the switch.

The 802.3ad mode requires that the switch have the appropriate ports configured as an 802.3ad aggregation. The precise method used to configure this varies from switch to switch, but, for example, a Cisco 3550 series switch requires that the appropriate ports first be grouped together in a single etherchannel instance, then that etherchannel is set to mode "lacp" to enable 802.3ad (instead of standard EtherChannel).

The balance-rr, balance-xor and broadcast modes generally require that the switch have the appropriate ports grouped together. The nomenclature for such a group differs between switches, it may be called an "etherchannel" (as in the Cisco example, above), a "trunk group" or some other similar variation. For these modes, each switch will also have its own configuration options for the switch's transmit policy to the bond. Typical choices include XOR of either the MAC or IP addresses. The transmit policy of the two peers does not need to match. For these three modes, the bonding mode really selects a transmit policy for an EtherChannel group; all three will interoperate with another EtherChannel group.

37.7 6. 802.1q VLAN Support

It is possible to configure VLAN devices over a bond interface using the 8021q driver. However, only packets coming from the 8021q driver and passing through bonding will be tagged by default. Self generated packets, for example, bonding's learning packets or ARP packets generated by either ALB mode or the ARP monitor mechanism, are tagged internally by bonding itself. As a result, bonding must "learn" the VLAN IDs configured above it, and use those IDs to tag self generated packets.

For reasons of simplicity, and to support the use of adapters that can do VLAN hardware acceleration offloading, the bonding interface declares itself as fully hardware offloading capable, it gets the add_vid/kill_vid notifications to gather the necessary information, and it propagates those actions to the slaves. In case of mixed adapter types, hardware accelerated tagged packets that should go through an adapter that is not offloading capable are "un-accelerated" by the bonding driver so the VLAN tag sits in the regular location.

VLAN interfaces *must* be added on top of a bonding interface only after enslaving at least one slave. The bonding interface has a hardware address of 00:00:00:00:00:00 until the first slave is added. If the VLAN interface is created prior to the first enslavement, it would pick up the all-zeroes hardware address. Once the first slave is attached to the bond, the bond device itself will pick up the slave's hardware address, which is then available for the VLAN device.

Also, be aware that a similar problem can occur if all slaves are released from a bond that still has one or more VLAN interfaces on top of it. When a new slave is added, the bonding interface will obtain its hardware address from the first slave, which might not match the hardware address of the VLAN interfaces (which was ultimately copied from an earlier slave).

There are two methods to insure that the VLAN device operates with the correct hardware address if all slaves are removed from a bond interface:

1. Remove all VLAN interfaces then recreate them
2. Set the bonding interface's hardware address so that it matches the hardware address of the VLAN interfaces.

Note that changing a VLAN interface's HW address would set the underlying device -- i.e. the bonding interface -- to promiscuous mode, which might not be what you want.

37.8 7. Link Monitoring

The bonding driver at present supports two schemes for monitoring a slave device's link state: the ARP monitor and the MII monitor.

At the present time, due to implementation restrictions in the bonding driver itself, it is not possible to enable both ARP and MII monitoring simultaneously.

37.8.1 7.1 ARP Monitor Operation

The ARP monitor operates as its name suggests: it sends ARP queries to one or more designated peer systems on the network, and uses the response as an indication that the link is operating. This gives some assurance that traffic is actually flowing to and from one or more peers on the local network.

37.8.2 7.2 Configuring Multiple ARP Targets

While ARP monitoring can be done with just one target, it can be useful in a High Availability setup to have several targets to monitor. In the case of just one target, the target itself may go down or have a problem making it unresponsive to ARP requests. Having an additional target (or several) increases the reliability of the ARP monitoring.

Multiple ARP targets must be separated by commas as follows:

```
# example options for ARP monitoring with three targets
alias bond0 bonding
options bond0 arp_interval=60 arp_ip_target=192.168.0.1,192.168.0.3,192.168.0.9
```

For just a single target the options would resemble:

```
# example options for ARP monitoring with one target
alias bond0 bonding
options bond0 arp_interval=60 arp_ip_target=192.168.0.100
```

37.8.3 7.3 MII Monitor Operation

The MII monitor monitors only the carrier state of the local network interface. It accomplishes this in one of three ways: by depending upon the device driver to maintain its carrier state, by querying the device's MII registers, or by making an ethtool query to the device.

If the `use_carrier` module parameter is 1 (the default value), then the MII monitor will rely on the driver for carrier state information (via the `netif_carrier` subsystem). As explained in the `use_carrier` parameter information, above, if the MII monitor fails to detect carrier loss on the device (e.g., when the cable is physically disconnected), it may be that the driver does not support `netif_carrier`.

If `use_carrier` is 0, then the MII monitor will first query the device's (via `ioctl`) MII registers and check the link state. If that request fails (not just that it returns carrier down), then the MII monitor will make an ethtool `ETHTOOL_GLINK` request to attempt to obtain the same information. If both methods fail (i.e., the driver either does not support or had some error in processing both the MII register and ethtool requests), then the MII monitor will assume the link is up.

37.9 8. Potential Sources of Trouble

37.9.1 8.1 Adventures in Routing

When bonding is configured, it is important that the slave devices not have routes that supersede routes of the master (or, generally, not have routes at all). For example, suppose the bonding device bond0 has two slaves, eth0 and eth1, and the routing table is as follows:

Kernel IP routing table							
Destination	Gateway	Genmask	Flags	MSS	Window	irtt	Iface
10.0.0.0	0.0.0.0	255.255.0.0	U	40	0		0 eth0
10.0.0.0	0.0.0.0	255.255.0.0	U	40	0		0 eth1
10.0.0.0	0.0.0.0	255.255.0.0	U	40	0		0 bond0
127.0.0.0	0.0.0.0	255.0.0.0	U	40	0		0 lo

This routing configuration will likely still update the receive/transmit times in the driver (needed by the ARP monitor), but may bypass the bonding driver (because outgoing traffic to, in this case, another host on network 10 would use eth0 or eth1 before bond0).

The ARP monitor (and ARP itself) may become confused by this configuration, because ARP requests (generated by the ARP monitor) will be sent on one interface (bond0), but the corresponding reply will arrive on a different interface (eth0). This reply looks to ARP as an unsolicited ARP reply (because ARP matches replies on an interface basis), and is discarded. The MII monitor is not affected by the state of the routing table.

The solution here is simply to insure that slaves do not have routes of their own, and if for some reason they must, those routes do not supersede routes of their master. This should generally be the case, but unusual configurations or errant manual or automatic static route additions may cause trouble.

37.9.2 8.2 Ethernet Device Renaming

On systems with network configuration scripts that do not associate physical devices directly with network interface names (so that the same physical device always has the same "ethX" name), it may be necessary to add some special logic to config files in /etc/modprobe.d/.

For example, given a modules.conf containing the following:

```
alias bond0 bonding
options bond0 mode=some-mode miimon=50
alias eth0 tg3
alias eth1 tg3
alias eth2 e1000
alias eth3 e1000
```

If neither eth0 and eth1 are slaves to bond0, then when the bond0 interface comes up, the devices may end up reordered. This happens because bonding is loaded first, then its slave device's drivers are loaded next. Since no other drivers have been loaded, when the e1000 driver loads, it will receive eth0 and eth1 for its devices, but the bonding configuration tries to enslave eth2 and eth3 (which may later be assigned to the tg3 devices).

Adding the following:

```
add above bonding e1000 tg3
```

causes modprobe to load e1000 then tg3, in that order, when bonding is loaded. This command is fully documented in the modules.conf manual page.

On systems utilizing modprobe an equivalent problem can occur. In this case, the following can be added to config files in /etc/modprobe.d/ as:

```
softdep bonding pre: tg3 e1000
```

This will load tg3 and e1000 modules before loading the bonding one. Full documentation on this can be found in the modprobe.d and modprobe manual pages.

37.9.3 8.3. Painfully Slow Or No Failed Link Detection By Miimon

By default, bonding enables the use_carrier option, which instructs bonding to trust the driver to maintain carrier state.

As discussed in the options section, above, some drivers do not support the netif_carrier_on/_off link state tracking system. With use_carrier enabled, bonding will always see these links as up, regardless of their actual state.

Additionally, other drivers do support netif_carrier, but do not maintain it in real time, e.g., only polling the link state at some fixed interval. In this case, miimon will detect failures, but only after some long period of time has expired. If it appears that miimon is very slow in detecting link failures, try specifying use_carrier=0 to see if that improves the failure detection time. If it does, then it may be that the driver checks the carrier state at a fixed interval, but does not cache the MII register values (so the use_carrier=0 method of querying the registers directly works). If use_carrier=0 does not improve the failover, then the driver may cache the registers, or the problem may be elsewhere.

Also, remember that miimon only checks for the device's carrier state. It has no way to determine the state of devices on or beyond other ports of a switch, or if a switch is refusing to pass traffic while still maintaining carrier on.

37.10 9. SNMP agents

If running SNMP agents, the bonding driver should be loaded before any network drivers participating in a bond. This requirement is due to the interface index (ipAdEntIfIndex) being associated to the first interface found with a given IP address. That is, there is only one ipAdEntIfIndex for each IP address. For example, if eth0 and eth1 are slaves of bond0 and the driver for eth0 is loaded before the bonding driver, the interface for the IP address will be associated with the eth0 interface. This configuration is shown below, the IP address 192.168.1.1 has an interface index of 2 which indexes to eth0 in the ifDescr table (ifDescr.2).

```
interfaces.ifTable.ifEntry.ifDescr.1 = lo
interfaces.ifTable.ifEntry.ifDescr.2 = eth0
interfaces.ifTable.ifEntry.ifDescr.3 = eth1
interfaces.ifTable.ifEntry.ifDescr.4 = eth2
interfaces.ifTable.ifEntry.ifDescr.5 = eth3
interfaces.ifTable.ifEntry.ifDescr.6 = bond0
```

```
ip.ipAddrTable.ipAddrEntry.ipAdEntIfIndex.10.10.10.10 = 5
ip.ipAddrTable.ipAddrEntry.ipAdEntIfIndex.192.168.1.1 = 2
ip.ipAddrTable.ipAddrEntry.ipAdEntIfIndex.10.74.20.94 = 4
ip.ipAddrTable.ipAddrEntry.ipAdEntIfIndex.127.0.0.1 = 1
```

This problem is avoided by loading the bonding driver before any network drivers participating in a bond. Below is an example of loading the bonding driver first, the IP address 192.168.1.1 is correctly associated with ifDescr.2.

```
interfaces.ifTable.ifEntry.ifDescr.1 = lo interfaces.ifTable.ifEntry.ifDescr.2 = bond0
interfaces.ifTable.ifEntry.ifDescr.3 = eth0 interfaces.ifTable.ifEntry.ifDescr.4 = eth1
interfaces.ifTable.ifEntry.ifDescr.5 = eth2 interfaces.ifTable.ifEntry.ifDescr.6 = eth3
ip.ipAddrTable.ipAddrEntry.ipAdEntIfIndex.10.10.10.10 = 6
ip.ipAddrTable.ipAddrEntry.ipAdEntIfIndex.192.168.1.1 = 2
ip.ipAddrTable.ipAddrEntry.ipAdEntIfIndex.10.74.20.94 = 5
ip.ipAddrTable.ipAddrEntry.ipAdEntIfIndex.127.0.0.1 = 1
```

While some distributions may not report the interface name in ifDescr, the association between the IP address and IfIndex remains and SNMP functions such as `Interface_Scan_Next` will report that association.

37.11 10. Promiscuous mode

When running network monitoring tools, e.g., `tcpdump`, it is common to enable promiscuous mode on the device, so that all traffic is seen (instead of seeing only traffic destined for the local host). The bonding driver handles promiscuous mode changes to the bonding master device (e.g., `bond0`), and propagates the setting to the slave devices.

For the `balance-rr`, `balance-xor`, `broadcast`, and `802.3ad` modes, the promiscuous mode setting is propagated to all slaves.

For the `active-backup`, `balance-tlb` and `balance-alb` modes, the promiscuous mode setting is propagated only to the active slave.

For `balance-tlb` mode, the active slave is the slave currently receiving inbound traffic.

For `balance-alb` mode, the active slave is the slave used as a "primary." This slave is used for mode-specific control traffic, for sending to peers that are unassigned or if the load is unbalanced.

For the `active-backup`, `balance-tlb` and `balance-alb` modes, when the active slave changes (e.g., due to a link failure), the promiscuous setting will be propagated to the new active slave.

37.12 11. Configuring Bonding for High Availability

High Availability refers to configurations that provide maximum network availability by having redundant or backup devices, links or switches between the host and the rest of the world. The goal is to provide the maximum availability of network connectivity (i.e., the network always works), even though other configurations could provide higher throughput.

37.12.1 11.1 High Availability in a Single Switch Topology

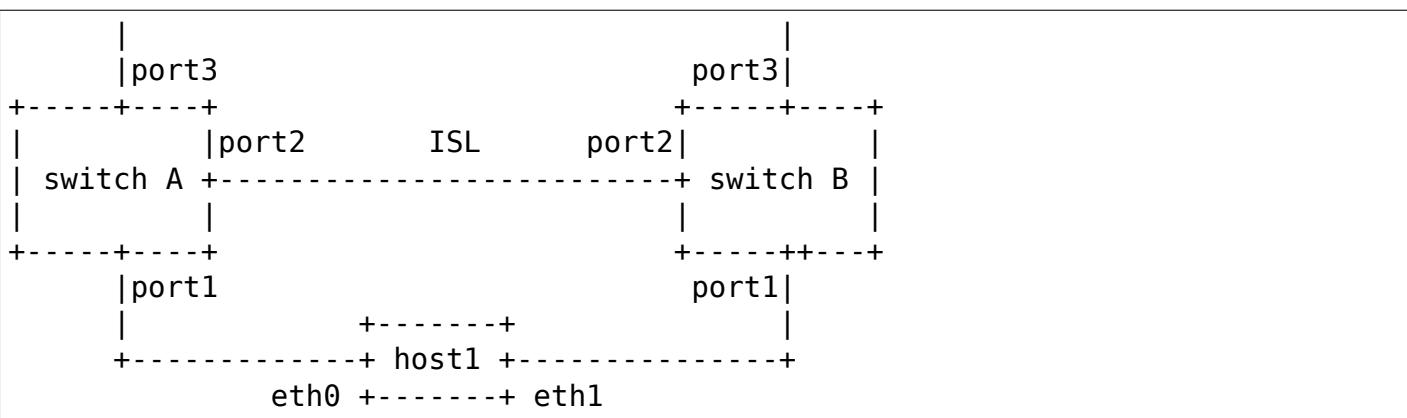
If two hosts (or a host and a single switch) are directly connected via multiple physical links, then there is no availability penalty to optimizing for maximum bandwidth. In this case, there is only one switch (or peer), so if it fails, there is no alternative access to fail over to. Additionally, the bonding load balance modes support link monitoring of their members, so if individual links fail, the load will be rebalanced across the remaining devices.

See Section 12, "Configuring Bonding for Maximum Throughput" for information on configuring bonding with one peer device.

37.12.2 11.2 High Availability in a Multiple Switch Topology

With multiple switches, the configuration of bonding and the network changes dramatically. In multiple switch topologies, there is a trade off between network availability and usable bandwidth.

Below is a sample network, configured to maximize the availability of the network:



In this configuration, there is a link between the two switches (ISL, or inter switch link), and multiple ports connecting to the outside world ("port3" on each switch). There is no technical reason that this could not be extended to a third switch.

37.12.3 11.2.1 HA Bonding Mode Selection for Multiple Switch Topology

In a topology such as the example above, the active-backup and broadcast modes are the only useful bonding modes when optimizing for availability; the other modes require all links to terminate on the same peer for them to behave rationally.

active-backup:

This is generally the preferred mode, particularly if the switches have an ISL and play together well. If the network configuration is such that one switch is specifically a backup switch (e.g., has lower capacity, higher cost, etc), then the primary option can be used to insure that the preferred link is always used when it is available.

broadcast:

This mode is really a special purpose mode, and is suitable only for very specific needs. For example, if the two switches are not connected (no ISL), and the networks beyond them are totally independent. In this case, if it is necessary for some specific one-way traffic to reach both independent networks, then the broadcast mode may be suitable.

37.12.4 11.2.2 HA Link Monitoring Selection for Multiple Switch Topology

The choice of link monitoring ultimately depends upon your switch. If the switch can reliably fail ports in response to other failures, then either the MII or ARP monitors should work. For example, in the above example, if the "port3" link fails at the remote end, the MII monitor has no direct means to detect this. The ARP monitor could be configured with a target at the remote end of port3, thus detecting that failure without switch support.

In general, however, in a multiple switch topology, the ARP monitor can provide a higher level of reliability in detecting end to end connectivity failures (which may be caused by the failure of any individual component to pass traffic for any reason). Additionally, the ARP monitor should be configured with multiple targets (at least one for each switch in the network). This will insure that, regardless of which switch is active, the ARP monitor has a suitable target to query.

Note, also, that of late many switches now support a functionality generally referred to as "trunk failover." This is a feature of the switch that causes the link state of a particular switch port to be set down (or up) when the state of another switch port goes down (or up). Its purpose is to propagate link failures from logically "exterior" ports to the logically "interior" ports that bonding is able to monitor via miimon. Availability and configuration for trunk failover varies by switch, but this can be a viable alternative to the ARP monitor when using suitable switches.

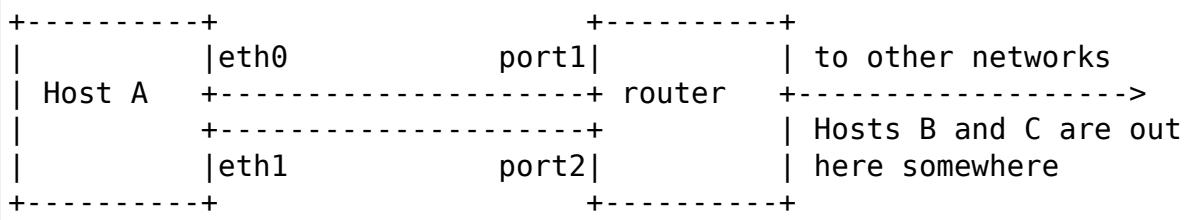
37.13 12. Configuring Bonding for Maximum Throughput

37.13.1 12.1 Maximizing Throughput in a Single Switch Topology

In a single switch configuration, the best method to maximize throughput depends upon the application and network environment. The various load balancing modes each have strengths and weaknesses in different environments, as detailed below.

For this discussion, we will break down the topologies into two categories. Depending upon the destination of most traffic, we categorize them into either "gateways" or "local" configurations.

In a gatewayed configuration, the "switch" is acting primarily as a router, and the majority of traffic passes through this router to other networks. An example would be the following:

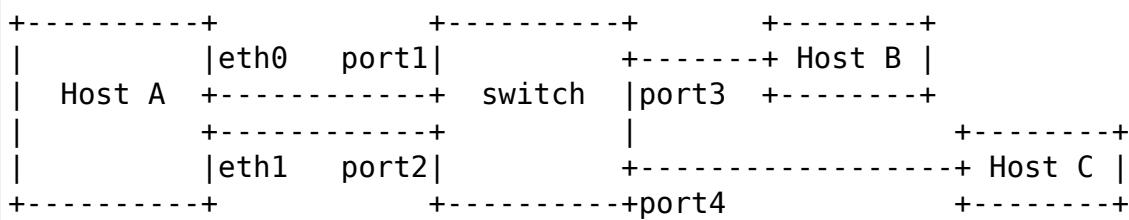


The router may be a dedicated router device, or another host acting as a gateway. For our discussion, the important point is that the majority of traffic from Host A will pass through the router to some other network before reaching its final destination.

In a gatewayed network configuration, although Host A may communicate with many other systems, all of its traffic will be sent and received via one other peer on the local network, the router.

Note that the case of two systems connected directly via multiple physical links is, for purposes of configuring bonding, the same as a gatewayed configuration. In that case, it happens that all traffic is destined for the "gateway" itself, not some other network beyond the gateway.

In a local configuration, the "switch" is acting primarily as a switch, and the majority of traffic passes through this switch to reach other stations on the same network. An example would be the following:



Again, the switch may be a dedicated switch device, or another host acting as a gateway. For our discussion, the important point is that the majority of traffic from Host A is destined for other hosts on the same local network (Hosts B and C in the above example).

In summary, in a gatewayed configuration, traffic to and from the bonded device will be to the same MAC level peer on the network (the gateway itself, i.e., the router), regardless of its final destination. In a local configuration, traffic flows directly to and from the final destinations, thus, each destination (Host B, Host C) will be addressed directly by their individual MAC addresses.

This distinction between a gatewayed and a local network configuration is important because many of the load balancing modes available use the MAC addresses of the local network source and destination to make load balancing decisions. The behavior of each mode is described below.

37.13.2 12.1.1 MT Bonding Mode Selection for Single Switch Topology

This configuration is the easiest to set up and to understand, although you will have to decide which bonding mode best suits your needs. The trade offs for each mode are detailed below:

balance-rr:

This mode is the only mode that will permit a single TCP/IP connection to stripe traffic across multiple interfaces. It is therefore the only mode that will allow a single TCP/IP stream to utilize more than one interface's worth of throughput. This comes at a cost, however: the striping generally results in peer systems receiving packets out of order, causing TCP/IP's congestion control system to kick in, often by retransmitting segments.

It is possible to adjust TCP/IP's congestion limits by altering the `net.ipv4.tcp_reordering` sysctl parameter. The usual default value is 3. But keep in mind TCP stack is able to automatically increase this when it detects reorders.

Note that the fraction of packets that will be delivered out of order is highly variable, and is unlikely to be zero. The level of reordering depends upon a variety of factors, including the networking interfaces, the switch, and the topology of the configuration. Speaking in general terms, higher speed network cards produce more reordering (due to factors such as packet coalescing), and a "many to many" topology will reorder at a higher rate than a "many slow to one fast" configuration.

Many switches do not support any modes that stripe traffic (instead choosing a port based upon IP or MAC level addresses); for those devices, traffic for a particular connection flowing through the switch to a balance-rr bond will not utilize greater than one interface's worth of bandwidth.

If you are utilizing protocols other than TCP/IP, UDP for example, and your application can tolerate out of order delivery, then this mode can allow for single stream datagram performance that scales near linearly as interfaces are added to the bond.

This mode requires the switch to have the appropriate ports configured for "etherchannel" or "trunking."

active-backup:

There is not much advantage in this network topology to the active-backup mode, as the inactive backup devices are all connected to the same peer as the primary. In this case, a load balancing mode (with link monitoring) will provide the same level of network availability, but with increased available bandwidth. On the plus side, active-backup mode does not require any configuration of the switch, so it may have value if the hardware available does not support any of the load balance modes.

balance-xor:

This mode will limit traffic such that packets destined for specific peers will always be sent over the same interface. Since the destination is determined by the MAC addresses involved, this mode works best in a "local" network configuration (as described above), with destinations all on the same local network. This mode is likely to be suboptimal if all your traffic is passed through a single router (i.e., a "gateawayed" network configuration, as described above).

As with balance-rr, the switch ports need to be configured for "etherchannel" or "trunking."

broadcast:

Like active-backup, there is not much advantage to this mode in this type of network topology.

802.3ad:

This mode can be a good choice for this type of network topology. The 802.3ad mode is an IEEE standard, so all peers that implement 802.3ad should interoperate well. The 802.3ad protocol includes automatic configuration of the aggregates, so minimal manual configuration of the switch is needed (typically only to designate that some set of devices is available for 802.3ad). The 802.3ad standard also mandates that frames be delivered in order (within certain limits), so in general single connections will not see misordering of packets. The 802.3ad mode does have some drawbacks: the standard mandates that all devices in the aggregate operate at the same speed and duplex. Also, as with all bonding load balance modes other than balance-rr, no single connection will be able to utilize more than a single interface's worth of bandwidth.

Additionally, the linux bonding 802.3ad implementation distributes traffic by peer (using an XOR of MAC addresses and packet type ID), so in a "gatewayed" configuration, all outgoing traffic will generally use the same device. Incoming traffic may also end up on a single device, but that is dependent upon the balancing policy of the peer's 802.3ad implementation. In a "local" configuration, traffic will be distributed across the devices in the bond.

Finally, the 802.3ad mode mandates the use of the MII monitor, therefore, the ARP monitor is not available in this mode.

balance-tlb:

The balance-tlb mode balances outgoing traffic by peer. Since the balancing is done according to MAC address, in a "gatewayed" configuration (as described above), this mode will send all traffic across a single device. However, in a "local" network configuration, this mode balances multiple local network peers across devices in a vaguely intelligent manner (not a simple XOR as in balance-xor or 802.3ad mode), so that mathematically unlucky MAC addresses (i.e., ones that XOR to the same value) will not all "bunch up" on a single interface.

Unlike 802.3ad, interfaces may be of differing speeds, and no special switch configuration is required. On the down side, in this mode all incoming traffic arrives over a single interface, this mode requires certain ethtool support in the network device driver of the slave interfaces, and the ARP monitor is not available.

balance-alb:

This mode is everything that balance-tlb is, and more. It has all of the features (and restrictions) of balance-tlb, and will also balance incoming traffic from local network peers (as described in the Bonding Module Options section, above).

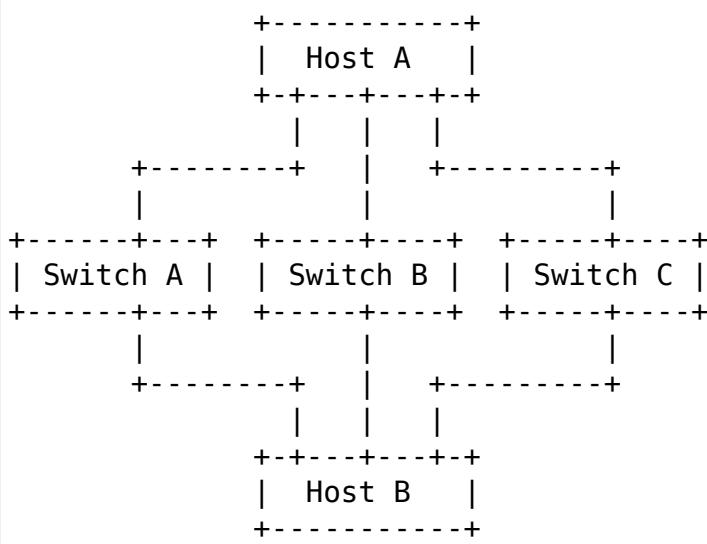
The only additional down side to this mode is that the network device driver must support changing the hardware address while the device is open.

37.13.3 12.1.2 MT Link Monitoring for Single Switch Topology

The choice of link monitoring may largely depend upon which mode you choose to use. The more advanced load balancing modes do not support the use of the ARP monitor, and are thus restricted to using the MII monitor (which does not provide as high a level of end to end assurance as the ARP monitor).

37.13.4 12.2 Maximum Throughput in a Multiple Switch Topology

Multiple switches may be utilized to optimize for throughput when they are configured in parallel as part of an isolated network between two or more systems, for example:



In this configuration, the switches are isolated from one another. One reason to employ a topology such as this is for an isolated network with many hosts (a cluster configured for high performance, for example), using multiple smaller switches can be more cost effective than a single larger switch, e.g., on a network with 24 hosts, three 24 port switches can be significantly less expensive than a single 72 port switch.

If access beyond the network is required, an individual host can be equipped with an additional network device connected to an external network; this host then additionally acts as a gateway.

37.13.5 12.2.1 MT Bonding Mode Selection for Multiple Switch Topology

In actual practice, the bonding mode typically employed in configurations of this type is balance-rr. Historically, in this network configuration, the usual caveats about out of order packet delivery are mitigated by the use of network adapters that do not do any kind of packet coalescing (via the use of NAPI, or because the device itself does not generate interrupts until some number of packets has arrived). When employed in this fashion, the balance-rr mode allows individual connections between two hosts to effectively utilize greater than one interface's bandwidth.

37.13.6 12.2.2 MT Link Monitoring for Multiple Switch Topology

Again, in actual practice, the MII monitor is most often used in this configuration, as performance is given preference over availability. The ARP monitor will function in this topology, but its advantages over the MII monitor are mitigated by the volume of probes needed as the number of systems involved grows (remember that each host in the network is configured with bonding).

37.14 13. Switch Behavior Issues

37.14.1 13.1 Link Establishment and Failover Delays

Some switches exhibit undesirable behavior with regard to the timing of link up and down reporting by the switch.

First, when a link comes up, some switches may indicate that the link is up (carrier available), but not pass traffic over the interface for some period of time. This delay is typically due to some type of autonegotiation or routing protocol, but may also occur during switch initialization (e.g., during recovery after a switch failure). If you find this to be a problem, specify an appropriate value to the `updelay` bonding module option to delay the use of the relevant interface(s).

Second, some switches may "bounce" the link state one or more times while a link is changing state. This occurs most commonly while the switch is initializing. Again, an appropriate `updelay` value may help.

Note that when a bonding interface has no active links, the driver will immediately reuse the first link that goes up, even if the `updelay` parameter has been specified (the `updelay` is ignored in this case). If there are slave interfaces waiting for the `updelay` timeout to expire, the interface that first went into that state will be immediately reused. This reduces down time of the network if the value of `updelay` has been overestimated, and since this occurs only in cases with no connectivity, there is no additional penalty for ignoring the `updelay`.

In addition to the concerns about switch timings, if your switches take a long time to go into backup mode, it may be desirable to not activate a backup interface immediately after a link goes down. Failover may be delayed via the `downdelay` bonding module option.

37.14.2 13.2 Duplicated Incoming Packets

NOTE: Starting with version 3.0.2, the bonding driver has logic to suppress duplicate packets, which should largely eliminate this problem. The following description is kept for reference.

It is not uncommon to observe a short burst of duplicated traffic when the bonding device is first used, or after it has been idle for some period of time. This is most easily observed by issuing a "ping" to some other host on the network, and noticing that the output from ping flags duplicates (typically one per slave).

For example, on a bond in active-backup mode with five slaves all connected to one switch, the output may appear as follows:

```
# ping -n 10.0.4.2
PING 10.0.4.2 (10.0.4.2) from 10.0.3.10 : 56(84) bytes of data.
64 bytes from 10.0.4.2: icmp_seq=1 ttl=64 time=13.7 ms
64 bytes from 10.0.4.2: icmp_seq=1 ttl=64 time=13.8 ms (DUP!)
64 bytes from 10.0.4.2: icmp_seq=1 ttl=64 time=13.8 ms (DUP!)
64 bytes from 10.0.4.2: icmp_seq=1 ttl=64 time=13.8 ms (DUP!)
64 bytes from 10.0.4.2: icmp_seq=1 ttl=64 time=13.8 ms (DUP!)
64 bytes from 10.0.4.2: icmp_seq=2 ttl=64 time=0.216 ms
64 bytes from 10.0.4.2: icmp_seq=3 ttl=64 time=0.267 ms
64 bytes from 10.0.4.2: icmp_seq=4 ttl=64 time=0.222 ms
```

This is not due to an error in the bonding driver, rather, it is a side effect of how many switches update their MAC forwarding tables. Initially, the switch does not associate the MAC address in the packet with a particular switch port, and so it may send the traffic to all ports until its MAC forwarding table is updated. Since the interfaces attached to the bond may occupy multiple ports on a single switch, when the switch (temporarily) floods the traffic to all ports, the bond device receives multiple copies of the same packet (one per slave device).

The duplicated packet behavior is switch dependent, some switches exhibit this, and some do not. On switches that display this behavior, it can be induced by clearing the MAC forwarding table (on most Cisco switches, the privileged command "clear mac address-table dynamic" will accomplish this).

37.15 14. Hardware Specific Considerations

This section contains additional information for configuring bonding on specific hardware platforms, or for interfacing bonding with particular switches or other devices.

37.15.1 14.1 IBM BladeCenter

This applies to the JS20 and similar systems.

On the JS20 blades, the bonding driver supports only balance-rr, active-backup, balance-tlb and balance-alb modes. This is largely due to the network topology inside the BladeCenter, detailed below.

37.15.2 JS20 network adapter information

All JS20s come with two Broadcom Gigabit Ethernet ports integrated on the planar (that's "motherboard" in IBM-speak). In the BladeCenter chassis, the eth0 port of all JS20 blades is hard wired to I/O Module #1; similarly, all eth1 ports are wired to I/O Module #2. An add-on Broadcom daughter card can be installed on a JS20 to provide two more Gigabit Ethernet ports. These ports, eth2 and eth3, are wired to I/O Modules 3 and 4, respectively.

Each I/O Module may contain either a switch or a passthrough module (which allows ports to be directly connected to an external switch). Some bonding modes require a specific BladeCenter internal network topology in order to function; these are detailed below.

Additional BladeCenter-specific networking information can be found in two IBM Redbooks (www.ibm.com/redbooks):

- "IBM eServer BladeCenter Networking Options"
- "IBM eServer BladeCenter Layer 2-7 Network Switching"

37.15.3 BladeCenter networking configuration

Because a BladeCenter can be configured in a very large number of ways, this discussion will be confined to describing basic configurations.

Normally, Ethernet Switch Modules (ESMs) are used in I/O modules 1 and 2. In this configuration, the eth0 and eth1 ports of a JS20 will be connected to different internal switches (in the respective I/O modules).

A passthrough module (OPM or CPM, optical or copper, passthrough module) connects the I/O module directly to an external switch. By using PMs in I/O module #1 and #2, the eth0 and eth1 interfaces of a JS20 can be redirected to the outside world and connected to a common external switch.

Depending upon the mix of ESMs and PMs, the network will appear to bonding as either a single switch topology (all PMs) or as a multiple switch topology (one or more ESMs, zero or more PMs). It is also possible to connect ESMs together, resulting in a configuration much like the example in "High Availability in a Multiple Switch Topology," above.

37.15.4 Requirements for specific modes

The balance-rr mode requires the use of passthrough modules for devices in the bond, all connected to an common external switch. That switch must be configured for "etherchannel" or "trunking" on the appropriate ports, as is usual for balance-rr.

The balance-alb and balance-tlb modes will function with either switch modules or passthrough modules (or a mix). The only specific requirement for these modes is that all network interfaces must be able to reach all destinations for traffic sent over the bonding device (i.e., the network must converge at some point outside the BladeCenter).

The active-backup mode has no additional requirements.

37.15.5 Link monitoring issues

When an Ethernet Switch Module is in place, only the ARP monitor will reliably detect link loss to an external switch. This is nothing unusual, but examination of the BladeCenter cabinet would suggest that the "external" network ports are the ethernet ports for the system, when in fact there is a switch between these "external" ports and the devices on the JS20 system itself. The MII monitor is only able to detect link failures between the ESM and the JS20 system.

When a passthrough module is in place, the MII monitor does detect failures to the "external" port, which is then directly connected to the JS20 system.

37.15.6 Other concerns

The Serial Over LAN (SoL) link is established over the primary ethernet (eth0) only, therefore, any loss of link to eth0 will result in losing your SoL connection. It will not fail over with other network traffic, as the SoL system is beyond the control of the bonding driver.

It may be desirable to disable spanning tree on the switch (either the internal Ethernet Switch Module, or an external switch) to avoid fail-over delay issues when using bonding.

37.16 15. Frequently Asked Questions

37.16.1 1. Is it SMP safe?

Yes. The old 2.0.xx channel bonding patch was not SMP safe. The new driver was designed to be SMP safe from the start.

37.16.2 2. What type of cards will work with it?

Any Ethernet type cards (you can even mix cards - a Intel EtherExpress PRO/100 and a 3com 3c905b, for example). For most modes, devices need not be of the same speed.

Starting with version 3.2.1, bonding also supports Infiniband slaves in active-backup mode.

37.16.3 3. How many bonding devices can I have?

There is no limit.

37.16.4 4. How many slaves can a bonding device have?

This is limited only by the number of network interfaces Linux supports and/or the number of network cards you can place in your system.

37.16.5 5. What happens when a slave link dies?

If link monitoring is enabled, then the failing device will be disabled. The active-backup mode will fail over to a backup link, and other modes will ignore the failed link. The link will continue to be monitored, and should it recover, it will rejoin the bond (in whatever manner is appropriate for the mode). See the sections on High Availability and the documentation for each mode for additional information.

Link monitoring can be enabled via either the miimon or arp_interval parameters (described in the module parameters section, above). In general, miimon monitors the carrier state as sensed by the underlying network device, and the arp monitor (arp_interval) monitors connectivity to another host on the local network.

If no link monitoring is configured, the bonding driver will be unable to detect link failures, and will assume that all links are always available. This will likely result in lost packets, and a resulting degradation of performance. The precise performance loss depends upon the bonding mode and network configuration.

37.16.6 6. Can bonding be used for High Availability?

Yes. See the section on High Availability for details.

37.16.7 7. Which switches/systems does it work with?

The full answer to this depends upon the desired mode.

In the basic balance modes (balance-rr and balance-xor), it works with any system that supports etherchannel (also called trunking). Most managed switches currently available have such support, and many unmanaged switches as well.

The advanced balance modes (balance-tlb and balance-alb) do not have special switch requirements, but do need device drivers that support specific features (described in the appropriate section under module parameters, above).

In 802.3ad mode, it works with systems that support IEEE 802.3ad Dynamic Link Aggregation. Most managed and many unmanaged switches currently available support 802.3ad.

The active-backup mode should work with any Layer-II switch.

37.16.8 8. Where does a bonding device get its MAC address from?

When using slave devices that have fixed MAC addresses, or when the fail_over_mac option is enabled, the bonding device's MAC address is the MAC address of the active slave.

For other configurations, if not explicitly configured (with ifconfig or ip link), the MAC address of the bonding device is taken from its first slave device. This MAC address is then passed to all following slaves and remains persistent (even if the first slave is removed) until the bonding device is brought down or reconfigured.

If you wish to change the MAC address, you can set it with ifconfig or ip link:

```
# ifconfig bond0 hw ether 00:11:22:33:44:55
# ip link set bond0 address 66:77:88:99:aa:bb
```

The MAC address can be also changed by bringing down/up the device and then changing its slaves (or their order):

```
# ifconfig bond0 down ; modprobe -r bonding
# ifconfig bond0 .... up
# ifenslave bond0 eth...
```

This method will automatically take the address from the next slave that is added.

To restore your slaves' MAC addresses, you need to detach them from the bond (`ifenslave -d bond0 eth0`). The bonding driver will then restore the MAC addresses that the slaves had before they were enslaved.

37.17 16. Resources and Links

The latest version of the bonding driver can be found in the latest version of the linux kernel, found on <http://kernel.org>

The latest version of this document can be found in the latest kernel source (named *Linux Ethernet Bonding Driver HOWTO*).

Discussions regarding the development of the bonding driver take place on the main Linux network mailing list, hosted at vger.kernel.org. The list address is:

netdev@vger.kernel.org

The administrative interface (to subscribe or unsubscribe) can be found at:

<http://vger.kernel.org/vger-lists.html#netdev>

CDC_MBIM - DRIVER FOR CDC MBIM MOBILE BROADBAND MODEMS

The cdc_mbim driver supports USB devices conforming to the "Universal Serial Bus Communications Class Subclass Specification for Mobile Broadband Interface Model" [1], which is a further development of "Universal Serial Bus Communications Class Subclass Specifications for Network Control Model Devices" [2] optimized for Mobile Broadband devices, aka "3G/LTE modems".

38.1 Command Line Parameters

The cdc_mbim driver has no parameters of its own. But the probing behaviour for NCM 1.0 backwards compatible MBIM functions (an "NCM/MBIM function" as defined in section 3.2 of [1]) is affected by a cdc_ncm driver parameter:

38.1.1 prefer_mbim

Type

Boolean

Valid Range

N/Y (0-1)

Default Value

Y (MBIM is preferred)

This parameter sets the system policy for NCM/MBIM functions. Such functions will be handled by either the cdc_ncm driver or the cdc_mbim driver depending on the prefer_mbim setting. Setting prefer_mbim=N makes the cdc_mbim driver ignore these functions and lets the cdc_ncm driver handle them instead.

The parameter is writable, and can be changed at any time. A manual unbind/bind is required to make the change effective for NCM/MBIM functions bound to the "wrong" driver

38.2 Basic usage

MBIM functions are inactive when unmanaged. The `cdc_mbim` driver only provides a userspace interface to the MBIM control channel, and will not participate in the management of the function. This implies that a userspace MBIM management application always is required to enable a MBIM function.

Such userspace applications includes, but are not limited to:

- `mbimcli` (included with the `libmbim` [3] library), and
- `ModemManager` [4]

Establishing a MBIM IP session rerequires at least these actions by the management application:

- open the control channel
- configure network connection settings
- connect to network
- configure IP interface

38.2.1 Management application development

The driver <-> userspace interfaces are described below. The MBIM control channel protocol is described in [1].

38.3 MBIM control channel userspace ABI

38.3.1 /dev/cdc-wdmX character device

The driver creates a two-way pipe to the MBIM function control channel using the `cdc-wdm` driver as a subdriver. The userspace end of the control channel pipe is a `/dev/cdc-wdmX` character device.

The `cdc_mbim` driver does not process or police messages on the control channel. The channel is fully delegated to the userspace management application. It is therefore up to this application to ensure that it complies with all the control channel requirements in [1].

The `cdc-wdmX` device is created as a child of the MBIM control interface USB device. The character device associated with a specific MBIM function can be looked up using sysfs. For example:

```
bjorn@nemi:~$ ls /sys/bus/usb/drivers/cdc_mbim/2-4:2.12/usbmisc
cdc-wdm0

bjorn@nemi:~$ grep . /sys/bus/usb/drivers/cdc_mbim/2-4:2.12/usbmisc/cdc-wdm0/
  ↳ dev
180:0
```

38.3.2 USB configuration descriptors

The wMaxControlMessage field of the CDC MBIM functional descriptor limits the maximum control message size. The management application is responsible for negotiating a control message size complying with the requirements in section 9.3.1 of [1], taking this descriptor field into consideration.

The userspace application can access the CDC MBIM functional descriptor of a MBIM function using either of the two USB configuration descriptor kernel interfaces described in [6] or [7].

See also the ioctl documentation below.

38.3.3 Fragmentation

The userspace application is responsible for all control message fragmentation and defragmentation, as described in section 9.5 of [1].

38.3.4 /dev/cdc-wdmX write()

The MBIM control messages from the management application *must not* exceed the negotiated control message size.

38.3.5 /dev/cdc-wdmX read()

The management application *must* accept control messages of up the negotiated control message size.

38.3.6 /dev/cdc-wdmX ioctl()

IOCTL_WDM_MAX_COMMAND: Get Maximum Command Size This ioctl returns the wMaxControlMessage field of the CDC MBIM functional descriptor for MBIM devices. This is intended as a convenience, eliminating the need to parse the USB descriptors from userspace.

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/types.h>
#include <linux/usb/cdc-wdm.h>
int main()
{
    __u16 max;
    int fd = open("/dev/cdc-wdm0", O_RDWR);
    if (!ioctl(fd, IOCTL_WDM_MAX_COMMAND, &max))
        printf("wMaxControlMessage is %d\n", max);
}
```

38.3.7 Custom device services

The MBIM specification allows vendors to freely define additional services. This is fully supported by the cdc_mbim driver.

Support for new MBIM services, including vendor specified services, is implemented entirely in userspace, like the rest of the MBIM control protocol

New services should be registered in the MBIM Registry [5].

38.4 MBIM data channel userspace ABI

38.4.1 wwanY network device

The cdc_mbim driver represents the MBIM data channel as a single network device of the "wwan" type. This network device is initially mapped to MBIM IP session 0.

38.4.2 Multiplexed IP sessions (IPS)

MBIM allows multiplexing up to 256 IP sessions over a single USB data channel. The cdc_mbim driver models such IP sessions as 802.1q VLAN subdevices of the master wwanY device, mapping MBIM IP session Z to VLAN ID Z for all values of Z greater than 0.

The device maximum Z is given in the MBIM_DEVICE_CAPS_INFO structure described in section 10.5.1 of [1].

The userspace management application is responsible for adding new VLAN links prior to establishing MBIM IP sessions where the SessionId is greater than 0. These links can be added by using the normal VLAN kernel interfaces, either ioctl or netlink.

For example, adding a link for a MBIM IP session with SessionId 3:

```
ip link add link wwan0 name wwan0.3 type vlan id 3
```

The driver will automatically map the "wwan0.3" network device to MBIM IP session 3.

38.4.3 Device Service Streams (DSS)

MBIM also allows up to 256 non-IP data streams to be multiplexed over the same shared USB data channel. The cdc_mbim driver models these sessions as another set of 802.1q VLAN subdevices of the master wwanY device, mapping MBIM DSS session A to VLAN ID (256 + A) for all values of A.

The device maximum A is given in the MBIM_DEVICE_SERVICES_INFO structure described in section 10.5.29 of [1].

The DSS VLAN subdevices are used as a practical interface between the shared MBIM data channel and a MBIM DSS aware userspace application. It is not intended to be presented as-is to an end user. The assumption is that a userspace application initiating a DSS session also takes care of the necessary framing of the DSS data, presenting the stream to the end user in an appropriate way for the stream type.

The network device ABI requires a dummy ethernet header for every DSS data frame being transported. The contents of this header is arbitrary, with the following exceptions:

- TX frames using an IP protocol (0x0800 or 0x86dd) will be dropped
- RX frames will have the protocol field set to ETH_P_802_3 (but will not be properly formatted 802.3 frames)
- RX frames will have the destination address set to the hardware address of the master device

The DSS supporting userspace management application is responsible for adding the dummy ethernet header on TX and stripping it on RX.

This is a simple example using tools commonly available, exporting DssSessionId 5 as a pty character device pointed to by a /dev/nmea symlink:

```
ip link add link wwan0 name wwan0.dss5 type vlan id 261
ip link set dev wwan0.dss5 up
socat INTERFACE:wwan0.dss5,type=2 PTY:,echo=0,link=/dev/nmea
```

This is only an example, most suitable for testing out a DSS service. Userspace applications supporting specific MBIM DSS services are expected to use the tools and programming interfaces required by that service.

Note that adding VLAN links for DSS sessions is entirely optional. A management application may instead choose to bind a packet socket directly to the master network device, using the received VLAN tags to map frames to the correct DSS session and adding 18 byte VLAN ethernet headers with the appropriate tag on TX. In this case using a socket filter is recommended, matching only the DSS VLAN subset. This avoid unnecessary copying of unrelated IP session data to userspace. For example:

```
static struct sock_filter dssfilter[] = {
    /* use special negative offsets to get VLAN tag */
    BPF_STMT(BPF_LD|BPF_B|BPF_ABS, SKF_AD_OFF + SKF_AD_VLAN_TAG_PRESENT),
    BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 1, 0, 6), /* true */

    /* verify DSS VLAN range */
    BPF_STMT(BPF_LD|BPF_H|BPF_ABS, SKF_AD_OFF + SKF_AD_VLAN_TAG),
    BPF_JUMP(BPF_JMP|BPF_JGE|BPF_K, 256, 0, 4), /* 256 is first DSS VLAN */
    BPF_JUMP(BPF_JMP|BPF_JGE|BPF_K, 512, 3, 0), /* 511 is last DSS VLAN */

    /* verify ethertype */
    BPF_STMT(BPF_LD|BPF_H|BPF_ABS, 2 * ETH_ALEN),
    BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, ETH_P_802_3, 0, 1),

    BPF_STMT(BPF_RET|BPF_K, (u_int)-1), /* accept */
    BPF_STMT(BPF_RET|BPF_K, 0), /* ignore */
};
```

38.4.4 Tagged IP session 0 VLAN

As described above, MBIM IP session 0 is treated as special by the driver. It is initially mapped to untagged frames on the wwanY network device.

This mapping implies a few restrictions on multiplexed IPS and DSS sessions, which may not always be practical:

- no IPS or DSS session can use a frame size greater than the MTU on IP session 0
- no IPS or DSS session can be in the up state unless the network device representing IP session 0 also is up

These problems can be avoided by optionally making the driver map IP session 0 to a VLAN subdevice, similar to all other IP sessions. This behaviour is triggered by adding a VLAN link for the magic VLAN ID 4094. The driver will then immediately start mapping MBIM IP session 0 to this VLAN, and will drop untagged frames on the master wwanY device.

Tip: It might be less confusing to the end user to name this VLAN subdevice after the MBIM SessionID instead of the VLAN ID. For example:

```
ip link add link wwan0 name wwan0.0 type vlan id 4094
```

38.4.5 VLAN mapping

Summarizing the cdc_mbim driver mapping described above, we have this relationship between VLAN tags on the wwanY network device and MBIM sessions on the shared USB data channel:

VLAN ID	MBIM type	MBIM SessionID	Notes
untagged	IPS	0	a)
1 - 255	IPS	1 - 255 <VLANID>	
256 - 511	DSS	0 - 255 <VLANID - 256>	b)
512 - 4093			c)
4094	IPS	0	

- a) if no VLAN ID 4094 link exists, else dropped
- b) unsupported VLAN range, unconditionally dropped
- c) if a VLAN ID 4094 link exists, else dropped

38.5 References

- 1) USB Implementers Forum, Inc. - "Universal Serial Bus Communications Class Subclass Specification for Mobile Broadband Interface Model", Revision 1.0 (Errata 1), May 1, 2013
 - http://www.usb.org/developers/docs/devclass_docs/
- 2) USB Implementers Forum, Inc. - "Universal Serial Bus Communications Class Subclass Specifications for Network Control Model Devices", Revision 1.0 (Errata 1), November 24, 2010
 - http://www.usb.org/developers/docs/devclass_docs/

- 3) libmbim - "a glib-based library for talking to WWAN modems and devices which speak the Mobile Interface Broadband Model (MBIM) protocol"
 - <http://www.freedesktop.org/wiki/Software/libmbim/>
- 4) ModemManager - "a DBus-activated daemon which controls mobile broadband (2G/3G/4G) devices and connections"
 - <http://www.freedesktop.org/wiki/Software/ModemManager/>
- 5) "MBIM (Mobile Broadband Interface Model) Registry"
 - <http://compliance.usb.org/mbim/>
- 6) "/sys/kernel/debug/usb/devices output format"
 - Documentation/driver-api/usb/usb.rst
- 7) "/sys/bus/usb/devices/.../descriptors"
 - Documentation/ABI/stable/sysfs-bus-usb

DCCP PROTOCOL

39.1 Introduction

Datagram Congestion Control Protocol (DCCP) is an unreliable, connection oriented protocol designed to solve issues present in UDP and TCP, particularly for real-time and multimedia (streaming) traffic. It divides into a base protocol (RFC 4340) and pluggable congestion control modules called CCIDs. Like pluggable TCP congestion control, at least one CCID needs to be enabled in order for the protocol to function properly. In the Linux implementation, this is the TCP-like CCID2 (RFC 4341). Additional CCIDs, such as the TCP-friendly CCID3 (RFC 4342), are optional. For a brief introduction to CCIDs and suggestions for choosing a CCID to match given applications, see section 10 of RFC 4340.

It has a base protocol and pluggable congestion control IDs (CCIDs).

DCCP is a Proposed Standard (RFC 2026), and the homepage for DCCP as a protocol is at <http://www.ietf.org/html.charters/dccp-charter.html>

39.2 Missing features

The Linux DCCP implementation does not currently support all the features that are specified in RFCs 4340...42.

The known bugs are at:

<http://www.linuxfoundation.org/collaborate/workgroups/networking/todo#DCCP>

For more up-to-date versions of the DCCP implementation, please consider using the experimental DCCP test tree; instructions for checking this out are on: http://www.linuxfoundation.org/collaborate/workgroups/networking/dccp_testing#Experimental_DCCP_source_tree

39.3 Socket options

DCCP_SOCKOPT_QPOLICY_ID sets the dequeuing policy for outgoing packets. It takes a policy ID as argument and can only be set before the connection (i.e. changes during an established connection are not supported). Currently, two policies are defined: the "simple" policy (DCCPQ_POLICY_SIMPLE), which does nothing special, and a priority-based variant (DCCPQ_POLICY_PRIO). The latter allows to pass an u32 priority value as ancillary data to sendmsg(), where higher numbers indicate a higher packet priority (similar to SO_PRIORITY). This ancillary data needs to be formatted using a cmsg(3) message header filled in as follows:

```
cmsg->cmsg_level = SOL_DCCP;
cmsg->cmsg_type = DCCP_SCM_PRIORITY;
cmsg->cmsg_len = CMSG_LEN(sizeof(uint32_t)); /* or CMSG_LEN(4) */
```

DCCP_SOCKOPT_QPOLICY_TXQLEN sets the maximum length of the output queue. A zero value is always interpreted as unbounded queue length. If different from zero, the interpretation of this parameter depends on the current dequeuing policy (see above): the "simple" policy will enforce a fixed queue size by returning EAGAIN, whereas the "prio" policy enforces a fixed queue length by dropping the lowest-priority packet first. The default value for this parameter is initialised from /proc/sys/net/dccp/default/tx_qlen.

DCCP_SOCKOPT_SERVICE sets the service. The specification mandates use of service codes (RFC 4340, sec. 8.1.2); if this socket option is not set, the socket will fall back to 0 (which means that no meaningful service code is present). On active sockets this is set before connect(); specifying more than one code has no effect (all subsequent service codes are ignored). The case is different for passive sockets, where multiple service codes (up to 32) can be set before calling bind().

DCCP_SOCKOPT_GET_CUR_MPS is read-only and retrieves the current maximum packet size (application payload size) in bytes, see RFC 4340, section 14.

DCCP_SOCKOPT_AVAILABLE_CCIDS is also read-only and returns the list of CCIDs supported by the endpoint. The option value is an array of type uint8_t whose size is passed as option length. The minimum array size is 4 elements, the value returned in the optlen argument always reflects the true number of built-in CCIDs.

DCCP_SOCKOPT_CCID is write-only and sets both the TX and RX CCIDs at the same time, combining the operation of the next two socket options. This option is preferable over the latter two, since often applications will use the same type of CCID for both directions; and mixed use of CCIDs is not currently well understood. This socket option takes as argument at least one uint8_t value, or an array of uint8_t values, which must match available CCIDS (see above). CCIDs must be registered on the socket before calling connect() or listen().

DCCP_SOCKOPT_TX_CCID is read/write. It returns the current CCID (if set) or sets the preference list for the TX CCID, using the same format as DCCP_SOCKOPT_CCID. Please note that the getsockopt argument type here is int, not uint8_t.

DCCP_SOCKOPT_RX_CCID is analogous to DCCP_SOCKOPT_TX_CCID, but for the RX CCID.

DCCP_SOCKOPT_SERVER_TIMEWAIT enables the server (listening socket) to hold timewait state when closing the connection (RFC 4340, 8.3). The usual case is that the closing server sends a CloseReq, whereupon the client holds timewait state. When this boolean socket option is on, the server sends a Close instead and will enter TIMEWAIT. This option must be set after accept() returns.

DCCP_SOCKOPT_SEND_CSCOV and DCCP_SOCKOPT_RECV_CSCOV are used for setting the partial checksum coverage (RFC 4340, sec. 9.2). The default is that checksums always cover the entire packet and that only fully covered application data is accepted by the receiver. Hence, when using this feature on the sender, it must be enabled at the receiver, too with suitable choice of CsCov.

DCCP_SOCKOPT_SEND_CSCOV sets the sender checksum coverage. Values in the range 0..15 are acceptable. The default setting is 0 (full coverage), values between 1..15 indicate partial coverage.

DCCP_SOCKOPT_RECV_CSCOV is for the receiver and has a different meaning: it sets a threshold, where again values 0..15 are acceptable. The default of 0 means that all packets with a partial coverage will be discarded. Values in the range 1..15 indicate that packets with minimally such a coverage value are also acceptable. The higher the number, the more restrictive this setting (see [RFC 4340, sec. 9.2.1]). Partial coverage settings are inherited to the child socket after accept().

The following two options apply to CCID 3 exclusively and are getsockopt()-only. In either case, a TFRC info struct (defined in <linux/tfrc.h>) is returned.

DCCP_SOCKOPT_CCID_RX_INFO

Returns a struct `tfrc_rx_info` in optval; the buffer for optval and optlen must be set to at least sizeof(struct `tfrc_rx_info`).

DCCP_SOCKOPT_CCID_TX_INFO

Returns a struct `tfrc_tx_info` in optval; the buffer for optval and optlen must be set to at least sizeof(struct `tfrc_tx_info`).

On unidirectional connections it is useful to close the unused half-connection via shutdown (SHUT_WR or SHUT_RD): this will reduce per-packet processing costs.

39.4 Sysctl variables

Several DCCP default parameters can be managed by the following sysctls (sysctl net.dccp.default or /proc/sys/net/dccp/default):

request_retries

The number of active connection initiation retries (the number of Requests minus one) before timing out. In addition, it also governs the behaviour of the other, passive side: this variable also sets the number of times DCCP repeats sending a Response when the initial handshake does not progress from RESPOND to OPEN (i.e. when no Ack is received after the initial Request). This value should be greater than 0, suggested is less than 10. Analogue of `tcp_syn_retries`.

retries1

How often a DCCP Response is retransmitted until the listening DCCP side considers its connecting peer dead. Analogue of `tcp_retries1`.

retries2

The number of times a general DCCP packet is retransmitted. This has importance for retransmitted acknowledgments and feature negotiation, data packets are never retransmitted. Analogue of `tcp_retries2`.

tx_ccid = 2

Default CCID for the sender-receiver half-connection. Depending on the choice of CCID, the Send Ack Vector feature is enabled automatically.

rx_ccid = 2

Default CCID for the receiver-sender half-connection; see `tx_ccid`.

seq_window = 100

The initial sequence window (sec. 7.5.2) of the sender. This influences the local ackno validity and the remote seqno validity windows (7.5.1). Values in the range Wmin = 32 (RFC 4340, 7.5.2) up to $2^{32}-1$ can be set.

tx_qlen = 5

The size of the transmit buffer in packets. A value of 0 corresponds to an unbounded transmit buffer.

sync_ratelimit = 125 ms

The timeout between subsequent DCCP-Sync packets sent in response to sequence-invalid packets on the same socket (RFC 4340, 7.5.4). The unit of this parameter is milliseconds; a value of 0 disables rate-limiting.

39.5 IOCTLs

FIONREAD

Works as in `udp(7)`: returns in the `int` argument pointer the size of the next pending datagram in bytes, or 0 when no datagram is pending.

SIOCOUTQ

Returns the number of unsent data bytes in the socket send queue as `int` into the buffer specified by the argument pointer.

39.6 Other tunables

Per-route rto_min support

CCID-2 supports the `RTAX_RTO_MIN` per-route setting for the minimum value of the RTO timer. This setting can be modified via the '`rto_min`' option of `iproute2`; for example:

```
> ip route change 10.0.0.0/24    rto_min 250j dev wlan0
> ip route add      10.0.0.254/32 rto_min 800j dev wlan0
> ip route show dev wlan0
```

CCID-3 also supports the `rto_min` setting: it is used to define the lower bound for the expiry of the nofeedback timer. This can be useful on LANs with very low RTTs (e.g., loopback, Gbit ethernet).

39.7 Notes

DCCP does not travel through NAT successfully at present on many boxes. This is because the checksum covers the pseudo-header as per TCP and UDP. Linux NAT support for DCCP has been added.

DCTCP (DATACENTER TCP)

DCTCP is an enhancement to the TCP congestion control algorithm for data center networks and leverages Explicit Congestion Notification (ECN) in the data center network to provide multi-bit feedback to the end hosts.

To enable it on end hosts:

```
sysctl -w net.ipv4.tcp_congestion_control=dctcp
sysctl -w net.ipv4.tcp_ecn_fallback=0 (optional)
```

All switches in the data center network running DCTCP must support ECN marking and be configured for marking when reaching defined switch buffer thresholds. The default ECN marking threshold heuristic for DCTCP on switches is 20 packets (30KB) at 1Gbps, and 65 packets (~100KB) at 10Gbps, but might need further careful tweaking.

For more details, see below documents:

Paper:

The algorithm is further described in detail in the following two SIGCOMM/SIGMETRICS papers:

- i) Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan:

“Data Center TCP (DCTCP)”, Data Center Networks session

Proc. ACM SIGCOMM, New Delhi, 2010.

http://simula.stanford.edu/~alizade/Site/DCTCP_files/dctcp-final.pdf

<http://www.sigcomm.org/CCR/papers/2010/October/1851275.1851192>

- ii) Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar:

“Analysis of DCTCP: Stability, Convergence, and Fairness” Proc. ACM SIGMETRICS, San Jose, 2011.

http://simula.stanford.edu/~alizade/Site/DCTCP_files/dctcp_analysis-full.pdf

IETF informational draft:

<http://tools.ietf.org/html/draft-bensley-tcpm-dctcp-00>

DCTCP site:

<http://simula.stanford.edu/~alizade/Site/DCTCP.html>

DNS RESOLVER MODULE

41.1 Overview

The DNS resolver module provides a way for kernel services to make DNS queries by way of requesting a key of key type dns_resolver. These queries are upcalled to userspace through /sbin/request-key.

These routines must be supported by userspace tools dns.upcall, cifs.upcall and request-key. It is under development and does not yet provide the full feature set. The features it does support include:

(*) Implements the dns_resolver key_type to contact userspace.

It does not yet support the following AFS features:

(*) Dns query support for AFSDB resource record.

This code is extracted from the CIFS filesystem.

41.2 Compilation

The module should be enabled by turning on the kernel configuration options:

```
CONFIG_DNS_RESOLVER      - tristate "DNS Resolver support"
```

41.3 Setting up

To set up this facility, the /etc/request-key.conf file must be altered so that /sbin/request-key can appropriately direct the upcalls. For example, to handle basic dname to IPv4/IPv6 address resolution, the following line should be added:

```
#OP      TYPE          DESC    CO-INFO PROGRAM ARG1 ARG2 ARG3 ...
===== ====== ====== ====== ====== ====== ====== =====
create  dns_resolver   *       *       /usr/sbin/cifs.upcall %k
```

To direct a query for query type 'foo', a line of the following should be added before the more general line given above as the first match is the one taken:

```
create dns_resolver    foo:*   *      /usr/sbin/dns.foo %k
```

41.4 Usage

To make use of this facility, one of the following functions that are implemented in the module can be called after doing:

```
#include <linux/dns_resolver.h>
::
int dns_query(const char *type, const char *name, size_t namelen,
              const char *options, char **_result, time_t *_expiry);
```

This is the basic access function. It looks for a cached DNS query and if it doesn't find it, it upcalls to userspace to make a new DNS query, which may then be cached. The key description is constructed as a string of the form::

```
[<type>:<name>]
```

where `<type>` optionally specifies the particular upcall program to invoke, and thus the type of query to do, and `<name>` specifies the string to be looked up. The default query type is a straight hostname to IP address set lookup.

The name parameter is not required to be a NUL-terminated string, and its length should be given by the namelen argument.

The options parameter may be NULL or it may be a set of options appropriate to the query type.

The return value is a string appropriate to the query type. For instance, for the default query type it is just a list of comma-separated IPv4 and IPv6 addresses. The caller must free the result.

The length of the result string is returned on success, and a negative error code is returned otherwise. -EKEYREJECTED will be returned if the DNS lookup failed.

If `_expiry` is non-NULL, the expiry time (TTL) of the result will be returned also.

The kernel maintains an internal keyring in which it caches looked up keys. This can be cleared by any process that has the CAP_SYS_ADMIN capability by the use of KEYCTL_KEYRING_CLEAR on the keyring ID.

41.5 Reading DNS Keys from Userspace

Keys of dns_resolver type can be read from userspace using keyctl_read() or "keyctl read/print/pipe".

41.6 Mechanism

The dnsresolver module registers a key type called "dns_resolver". Keys of this type are used to transport and cache DNS lookup results from userspace.

When dns_query() is invoked, it calls request_key() to search the local keyrings for a cached DNS result. If that fails to find one, it upcalls to userspace to get a new result.

Upcalls to userspace are made through the request_key() upcall vector, and are directed by means of configuration lines in /etc/request-key.conf that tell /sbin/request-key what program to run to instantiate the key.

The upcall handler program is responsible for querying the DNS, processing the result into a form suitable for passing to the keyctl_instantiate_key() routine. This then passes the data to dns_resolver_instantiate() which strips off and processes any options included in the data, and then attaches the remainder of the string to the key as its payload.

The upcall handler program should set the expiry time on the key to that of the lowest TTL of all the records it has extracted a result from. This means that the key will be discarded and recreated when the data it holds has expired.

dns_query() returns a copy of the value attached to the key, or an error if that is indicated instead.

See <[file:Documentation/security/keys/request-key.rst](#)> for further information about request-key function.

41.7 Debugging

Debugging messages can be turned on dynamically by writing a 1 into the following file:

```
/sys/module/dns_resolver/parameters/debug
```


SOFTNET DRIVER ISSUES

42.1 Probing guidelines

42.1.1 Address validation

Any hardware layer address you obtain for your device should be verified. For example, for ethernet check it with linux/etherdevice.h:`is_valid_ether_addr()`

42.2 Close/stop guidelines

42.2.1 Quiescence

After the `ndo_stop` routine has been called, the hardware must not receive or transmit any data. All in flight packets must be aborted. If necessary, poll or wait for completion of any reset commands.

42.2.2 Auto-close

The `ndo_stop` routine will be called by `unregister_netdevice` if device is still UP.

42.3 Transmit path guidelines

42.3.1 Stop queues in advance

The `ndo_start_xmit` method must not return `NETDEV_TX_BUSY` under any normal circumstances. It is considered a hard error unless there is no way your device can tell ahead of time when its transmit function will become busy.

Instead it must maintain the queue properly. For example, for a driver implementing scatter-gather this means:

```
static u32 drv_tx_avail(struct drv_ring *dr)
{
    u32 used = READ_ONCE(dr->prod) - READ_ONCE(dr->cons);

    return dr->tx_ring_size - (used & bp->tx_ring_mask);
```

```

}

static netdev_tx_t drv_hard_start_xmit(struct sk_buff *skb,
                                       struct net_device *dev)
{
    struct drv *dp = netdev_priv(dev);
    struct netdev_queue *txq;
    struct drv_ring *dr;
    int idx;

    idx = skb_get_queue_mapping(skb);
    dr = dp->tx_rings[idx];
    txq = netdev_get_tx_queue(dev, idx);

    //...
    /* This should be a very rare race - log it. */
    if (drv_tx_avail(dr) <= skb_shinfo(skb)->nr_frags + 1) {
        netif_stop_queue(dev);
        netdev_warn(dev, "Tx Ring full when queue awake!\n");
        return NETDEV_TX_BUSY;
    }

    //... queue packet to card ...

    netdev_tx_sent_queue(txq, skb->len);

    //... update tx producer index using WRITE_ONCE() ...

    if (!netif_txq_maybe_stop(txq, drv_tx_avail(dr),
                             MAX_SKB_FRAGS + 1, 2 * MAX_SKB_FRAGS))
        dr->stats.stopped++;

    //...
    return NETDEV_TX_OK;
}

```

And then at the end of your TX reclamation event handling:

```

//... update tx consumer index using WRITE_ONCE() ...

netif_txq_completed_wake(txq, cmpl_pkts, cmpl_bytes,
                        drv_tx_avail(dr), 2 * MAX_SKB_FRAGS);

```

Lockless queue stop / wake helper macros

The `netif_txq_maybe_stop()` and `_netif_txq_completed_wake()` macros are designed to safely implement stopping and waking netdev queues without full lock protection.

We assume that there can be no concurrent stop attempts and no concurrent wake attempts. The try-stop should happen from the xmit handler, while wake up should be triggered from NAPI poll context. The two may run concurrently (single producer, single consumer).

The try-stop side is expected to run from the xmit handler and therefore it does not reschedule Tx (`netif_tx_start_queue()` instead of `netif_tx_wake_queue()`). Uses of the `stop` macros outside of the xmit handler may lead to xmit queue being enabled but not run. The waking side does not have similar context restrictions.

The macros guarantee that rings will not remain stopped if there's space available, but they do *not* prevent false wake ups when the ring is full! Drivers should check for ring full at the start for the xmit handler.

All descriptor ring indexes (and other relevant shared state) must be updated before invoking the macros.

42.3.2 No exclusive ownership

An `ndo_start_xmit` method must not modify the shared parts of a cloned SKB.

42.3.3 Timely completions

Do not forget that once you return `NETDEV_TX_OK` from your `ndo_start_xmit` method, it is your driver's responsibility to free up the SKB and in some finite amount of time.

For example, this means that it is not allowed for your TX mitigation scheme to let TX packets "hang out" in the TX ring unreclaimed forever if no new TX packets are sent. This error can deadlock sockets waiting for send buffer room to be freed up.

If you return `NETDEV_TX_BUSY` from the `ndo_start_xmit` method, you must not keep any reference to that SKB and you must not attempt to free it up.

EQL DRIVER: SERIAL IP LOAD BALANCING HOWTO

Simon "Guru Aleph-Null" Janes, simon@ncm.com

v1.1, February 27, 1995

This is the manual for the EQL device driver. EQL is a software device that lets you load-balance IP serial links (SLIP or uncompressed PPP) to increase your bandwidth. It will not reduce your latency (i.e. ping times) except in the case where you already have lots of traffic on your link, in which it will help them out. This driver has been tested with the 1.1.75 kernel, and is known to have patched cleanly with 1.1.86. Some testing with 1.1.92 has been done with the v1.1 patch which was only created to patch cleanly in the very latest kernel source trees. (Yes, it worked fine.)

43.1 1. Introduction

Which is worse? A huge fee for a 56K leased line or two phone lines? It's probably the former. If you find yourself craving more bandwidth, and have a ISP that is flexible, it is now possible to bind modems together to work as one point-to-point link to increase your bandwidth. All without having to have a special black box on either side.

The eql driver has only been tested with the Livingston PortMaster-2e terminal server. I do not know if other terminal servers support load-balancing, but I do know that the PortMaster does it, and does it almost as well as the eql driver seems to do it (--) Unfortunately, in my testing so far, the Livingston PortMaster 2e's load-balancing is a good 1 to 2 KB/s slower than the test machine working with a 28.8 Kbps and 14.4 Kbps connection. However, I am not sure that it really is the PortMaster, or if it's Linux's TCP drivers. I'm told that Linux's TCP implementation is pretty fast though.--)

I suggest to ISPs out there that it would probably be fair to charge a load-balancing client 75% of the cost of the second line and 50% of the cost of the third line etc...

Hey, we can all dream you know...

43.2 2. Kernel Configuration

Here I describe the general steps of getting a kernel up and working with the eql driver. From patching, building, to installing.

43.2.1 2.1. Patching The Kernel

If you do not have or cannot get a copy of the kernel with the eql driver folded into it, get your copy of the driver from ftp://slaughter.ncm.com/pub/Linux/LOAD_BALANCING/eql-1.1.tar.gz. Unpack this archive someplace obvious like /usr/local/src/. It will create the following files:

```
-rw-r--r-- guru/ncm      198 Jan 19 18:53 eql-1.1/NO-WARRANTY  
-rw-r--r-- guru/ncm     30620 Feb 27 21:40 1995 eql-1.1/eql-1.1.patch  
-rwxr-xr-x guru/ncm    16111 Jan 12 22:29 1995 eql-1.1/eql_enslave  
-rw-r--r-- guru/ncm     2195 Jan 10 21:48 1995 eql-1.1/eql_enslave.c
```

Unpack a recent kernel (something after 1.1.92) someplace convenient like say /usr/src/linux-1.1.92.eql. Use symbolic links to point /usr/src/linux to this development directory.

Apply the patch by running the commands:

```
cd /usr/src  
patch </usr/local/src/eql-1.1/eql-1.1.patch
```

43.2.2 2.2. Building The Kernel

After patching the kernel, run make config and configure the kernel for your hardware.

After configuration, make and install according to your habit.

43.3 3. Network Configuration

So far, I have only used the eql device with the DSLIP SLIP connection manager by Matt Dillon (-- "The man who sold his soul to code so much so quickly."--) . How you configure it for other "connection" managers is up to you. Most other connection managers that I've seen don't do a very good job when it comes to handling more than one connection.

43.3.1 3.1. /etc/rc.d/rc.inet1

In rc.inet1, ifconfig the eql device to the IP address you usually use for your machine, and the MTU you prefer for your SLIP lines. One could argue that MTU should be roughly half the usual size for two modems, one-third for three, one-fourth for four, etc... But going too far below 296 is probably overkill. Here is an example ifconfig command that sets up the eql device:

```
ifconfig eql 198.67.33.239 mtu 1006
```

Once the eql device is up and running, add a static default route to it in the routing table using the cool new route syntax that makes life so much easier:

```
route add default eql
```

43.3.2 3.2. Enslaving Devices By Hand

Enslaving devices by hand requires two utility programs: `eql_enslave` and `eql_emancipate` (-- `eql_emancipate` hasn't been written because when an enslaved device "dies", it is automatically taken out of the queue. I haven't found a good reason to write it yet... other than for completeness, but that isn't a good motivator is it?--)

The syntax for enslaving a device is "`eql_enslave <master-name> <slave-name> <estimated-bps>`". Here are some example enslavings:

```
eql_enslave eql sl0 28800
eql_enslave eql ppp0 14400
eql_enslave eql sl1 57600
```

When you want to free a device from its life of slavery, you can either down the device with ifconfig (eql will automatically bury the dead slave and remove it from its queue) or use `eql_emancipate` to free it. (-- Or just ifconfig it down, and the eql driver will take it out for you.--):

```
eql_emancipate eql sl0
eql_emancipate eql ppp0
eql_emancipate eql sl1
```

43.3.3 3.3. DSLIP Configuration for the eql Device

The general idea is to bring up and keep up as many SLIP connections as you need, automatically.

3.3.1. /etc/slip/runslip.conf

Here is an example runslip.conf:

```
name          sl-line-1
enabled
baud         38400
mtu          576
ducmd        -e /etc/slip/dialout/cua2-288.xp -t 9
command      eql_enslave eql $interface 28800
address      198.67.33.239
line         /dev/cua2

name          sl-line-2
enabled
baud         38400
mtu          576
ducmd        -e /etc/slip/dialout/cua3-288.xp -t 9
command      eql_enslave eql $interface 28800
address      198.67.33.239
line         /dev/cua3
```

43.3.4 3.4. Using PPP and the eql Device

I have not yet done any load-balancing testing for PPP devices, mainly because I don't have a PPP-connection manager like SLIP has with DSLIP. I did find a good tip from LinuxNET:Billy for PPP performance: make sure you have asyncmap set to something so that control characters are not escaped.

I tried to fix up a PPP script/system for redialing lost PPP connections for use with the eql driver the weekend of Feb 25-26 '95 (Hereafter known as the 8-hour PPP Hate Festival). Perhaps later this year.

43.4 4. About the Slave Scheduler Algorithm

The slave scheduler probably could be replaced with a dozen other things and push traffic much faster. The formula in the current set up of the driver was tuned to handle slaves with wildly different bits-per-second "priorities".

All testing I have done was with two 28.8 V.FC modems, one connecting at 28800 bps or slower, and the other connecting at 14400 bps all the time.

One version of the scheduler was able to push 5.3 K/s through the 28800 and 14400 connections, but when the priorities on the links were very wide apart (57600 vs. 14400) the "faster" modem received all traffic and the "slower" modem starved.

43.5 5. Testers' Reports

Some people have experimented with the eql device with newer kernels (than 1.1.75). I have since updated the driver to patch cleanly in newer kernels because of the removal of the old "slave-balancing" driver config option.

- icee from LinuxNET patched 1.1.86 without any rejects and was able to boot the kernel and enslave a couple of ISDN PPP links.

43.5.1 5.1. Randolph Bentson's Test Report

```
From bentson@grieg.seaslug.org Wed Feb  8 19:08:09 1995
Date: Tue, 7 Feb 95 22:57 PST
From: Randolph Bentson <bentson@grieg.seaslug.org>
To: guru@ncm.com
Subject: EQL driver tests
```

I have been checking out your eql driver. (Nice work, that!) Although you may already done this performance testing, here are some data I've discovered.

Randolph Bentson
bentson@grieg.seaslug.org

A pseudo-device driver, EQL, written by Simon Janes, can be used to bundle multiple SLIP connections into what appears to be a single connection. This allows one to improve dial-up network connectivity gradually, without having to buy expensive DSU/CSU hardware and services.

I have done some testing of this software, with two goals in mind: first, to ensure it actually works as described and second, as a method of exercising my device driver.

The following performance measurements were derived from a set of SLIP connections run between two Linux systems (1.1.84) using a 486DX2/66 with a Cyclom-8Ys and a 486SLC/40 with a Cyclom-16Y. (Ports 0,1,2,3 were used. A later configuration will distribute port selection across the different Cirrus chips on the boards.) Once a link was established, I timed a binary ftp transfer of 289284 bytes of data. If there were no overhead (packet headers, inter-character and inter-packet delays, etc.) the transfers would take the following times:

bits/sec	seconds
345600	8.3
234600	12.3
172800	16.7
153600	18.8
76800	37.6
57600	50.2
38400	75.3
28800	100.4

19200	150.6
9600	301.3

A single line running at the lower speeds and with large packets comes to within 2% of this. Performance is limited for the higher speeds (as predicted by the Cirrus databook) to an aggregate of about 160 kbytes/sec. The next round of testing will distribute the load across two or more Cirrus chips.

The good news is that one gets nearly the full advantage of the second, third, and fourth line's bandwidth. (The bad news is that the connection establishment seemed fragile for the higher speeds. Once established, the connection seemed robust enough.)

#lines	speed kbit/sec	mtu	seconds duration	theory speed	actual speed	%of max
3	115200	900	—	345600		
3	115200	400	18.1	345600	159825	46
2	115200	900	—	230400		
2	115200	600	18.1	230400	159825	69
2	115200	400	19.3	230400	149888	65
4	57600	900	—	234600		
4	57600	600	—	234600		
4	57600	400	—	234600		
3	57600	600	20.9	172800	138413	80
3	57600	900	21.2	172800	136455	78
3	115200	600	21.7	345600	133311	38
3	57600	400	22.5	172800	128571	74
4	38400	900	25.2	153600	114795	74
4	38400	600	26.4	153600	109577	71
4	38400	400	27.3	153600	105965	68
2	57600	900	29.1	115200	99410.3	86
1	115200	900	30.7	115200	94229.3	81
2	57600	600	30.2	115200	95789.4	83
3	38400	900	30.3	115200	95473.3	82
3	38400	600	31.2	115200	92719.2	80
1	115200	600	31.3	115200	92423	80
2	57600	400	32.3	115200	89561.6	77
1	115200	400	32.8	115200	88196.3	76
3	38400	400	33.5	115200	86353.4	74
2	38400	900	43.7	76800	66197.7	86
2	38400	600	44	76800	65746.4	85
2	38400	400	47.2	76800	61289	79
4	19200	900	50.8	76800	56945.7	74
4	19200	400	53.2	76800	54376.7	70
4	19200	600	53.7	76800	53870.4	70
1	57600	900	54.6	57600	52982.4	91
1	57600	600	56.2	57600	51474	89
3	19200	900	60.5	57600	47815.5	83
1	57600	400	60.2	57600	48053.8	83
3	19200	600	62	57600	46658.7	81

continues on next page

Table 1 – continued from previous page

#lines	speed kbit/sec	mtu	seconds	duration	theory speed	actual speed	%of max
3	19200	400	64.7		57600	44711.6	77
1	38400	900	79.4		38400	36433.8	94
1	38400	600	82.4		38400	35107.3	91
2	19200	900	84.4		38400	34275.4	89
1	38400	400	86.8		38400	33327.6	86
2	19200	600	87.6		38400	33023.3	85
2	19200	400	91.2		38400	31719.7	82
4	9600	900	94.7		38400	30547.4	79
4	9600	400	106		38400	27290.9	71
4	9600	600	110		38400	26298.5	68
3	9600	900	118		28800	24515.6	85
3	9600	600	120		28800	24107	83
3	9600	400	131		28800	22082.7	76
1	19200	900	155		19200	18663.5	97
1	19200	600	161		19200	17968	93
1	19200	400	170		19200	17016.7	88
2	9600	600	176		19200	16436.6	85
2	9600	900	180		19200	16071.3	83
2	9600	400	181		19200	15982.5	83
1	9600	900	305		9600	9484.72	98
1	9600	600	314		9600	9212.87	95
1	9600	400	332		9600	8713.37	90

43.5.2 5.2. Anthony Healy's Report

Date: Mon, 13 Feb 1995 16:17:29 +1100 (EST)
 From: Antony Healey <ahealey@st.nepean.uws.edu.au>
 To: Simon Janes <guru@ncm.com>
 Subject: Re: Load Balancing

Hi Simon,

I've installed your patch and it works great. I have trialed it over twin SL/IP lines, just over null modems, but I was able to data at over 48Kb/s [ISDN link -Simon]. I managed a transfer of up to 7.5 Kbyte/s on one go, but averaged around 6.4 Kbyte/s, which I think is pretty cool. :)

LC-TRIE IMPLEMENTATION NOTES

44.1 Node types

leaf

An end node with data. This has a copy of the relevant key, along with 'hlist' with routing table entries sorted by prefix length. See struct leaf and struct leaf_info.

trie node or tnode

An internal node, holding an array of child (leaf or tnode) pointers, indexed through a subset of the key. See Level Compression.

44.2 A few concepts explained

Bits (tnode)

The number of bits in the key segment used for indexing into the child array - the "child index". See Level Compression.

Pos (tnode)

The position (in the key) of the key segment used for indexing into the child array. See Path Compression.

Path Compression / skipped bits

Any given tnode is linked to from the child array of its parent, using a segment of the key specified by the parent's "pos" and "bits". In certain cases, this tnode's own "pos" will not be immediately adjacent to the parent (pos+bits), but there will be some bits in the key skipped over because they represent a single path with no deviations. These "skipped bits" constitute Path Compression. Note that the search algorithm will simply skip over these bits when searching, making it necessary to save the keys in the leaves to verify that they actually do match the key we are searching for.

Level Compression / child arrays

the trie is kept level balanced moving, under certain conditions, the children of a full child (see "full_children") up one level, so that instead of a pure binary tree, each internal node ("tnode") may contain an arbitrarily large array of links to several children. Conversely, a tnode with a mostly empty child array (see empty_children) may be "halved", having some of its children moved downwards one level, in order to avoid ever-increasing child arrays.

empty_children

the number of positions in the child array of a given tnode that are NULL.

full_children

the number of children of a given tnode that aren't path compressed. (in other words, they aren't NULL or leaves and their "pos" is equal to this tnode's "pos"+"bits").

(The word "full" here is used more in the sense of "complete" than as the opposite of "empty", which might be a tad confusing.)

44.3 Comments

We have tried to keep the structure of the code as close to fib_hash as possible to allow verification and help up reviewing.

fib_find_node()

A good start for understanding this code. This function implements a straightforward trie lookup.

fib_insert_node()

Inserts a new leaf node in the trie. This is bit more complicated than fib_find_node(). Inserting a new node means we might have to run the level compression algorithm on part of the trie.

trie_leaf_remove()

Looks up a key, deletes it and runs the level compression algorithm.

trie_rebalance()

The key function for the dynamic trie after any change in the trie it is run to optimize and reorganize. It will walk the trie upwards towards the root from a given tnode, doing a resize() at each step to implement level compression.

resize()

Analyzes a tnode and optimizes the child array size by either inflating or shrinking it repeatedly until it fulfills the criteria for optimal level compression. This part follows the original paper pretty closely and there may be some room for experimentation here.

inflate()

Doubles the size of the child array within a tnode. Used by resize().

halve()

Halves the size of the child array within a tnode - the inverse of inflate(). Used by resize();

fn_trie_insert(), fn_trie_delete(), fn_trie_select_default()

The route manipulation functions. Should conform pretty closely to the corresponding functions in fib_hash.

fn_trie_flush()

This walks the full trie (using nextleaf()) and searches for empty leaves which have to be removed.

fn_trie_dump()

Dumps the routing table ordered by prefix length. This is somewhat slower than the corresponding fib_hash function, as we have to walk the entire trie for each prefix length. In comparison, fib_hash is organized as one "zone"/hash per prefix length.

44.4 Locking

`fib_lock` is used for an RW-lock in the same way that this is done in `fib_hash`. However, the functions are somewhat separated for other possible locking scenarios. It might conceivably be possible to run `trie_rebalance` via RCU to avoid `read_lock` in the `fn_trie_lookup()` function.

44.5 Main lookup mechanism

`fn_trie_lookup()` is the main lookup function.

The lookup is in its simplest form just like `fib_find_node()`. We descend the trie, key segment by key segment, until we find a leaf. `check_leaf()` does the `fib_semantic_match` in the leaf's sorted prefix hlist.

If we find a match, we are done.

If we don't find a match, we enter prefix matching mode. The prefix length, starting out at the same as the key length, is reduced one step at a time, and we backtrack upwards through the trie trying to find a longest matching prefix. The goal is always to reach a leaf and get a positive result from the `fib_semantic_match` mechanism.

Inside each tnode, the search for longest matching prefix consists of searching through the child array, chopping off (zeroing) the least significant "1" of the child index until we find a match or the child index consists of nothing but zeros.

At this point we backtrack (`t->stats.backtrack++`) up the trie, continuing to chop off part of the key in order to find the longest matching prefix.

At this point we will repeatedly descend subtrees to look for a match, and there are some optimizations available that can provide us with "shortcuts" to avoid descending into dead ends. Look for "HL_OPTIMIZE" sections in the code.

To alleviate any doubts about the correctness of the route selection process, a new netlink operation has been added. Look for `NETLINK_FIB_LOOKUP`, which gives userland access to `fib_lookup()`.

LINUX SOCKET FILTERING AKA BERKELEY PACKET FILTER (BPF)

45.1 Notice

This file used to document the eBPF format and mechanisms even when not related to socket filtering. The `../bpf/index.rst` has more details on eBPF.

45.2 Introduction

Linux Socket Filtering (LSF) is derived from the Berkeley Packet Filter. Though there are some distinct differences between the BSD and Linux Kernel filtering, but when we speak of BPF or LSF in Linux context, we mean the very same mechanism of filtering in the Linux kernel.

BPF allows a user-space program to attach a filter onto any socket and allow or disallow certain types of data to come through the socket. LSF follows exactly the same filter code structure as BSD's BPF, so referring to the BSD `bpf.4` manpage is very helpful in creating filters.

On Linux, BPF is much simpler than on BSD. One does not have to worry about devices or anything like that. You simply create your filter code, send it to the kernel via the `SO_ATTACH_FILTER` option and if your filter code passes the kernel check on it, you then immediately begin filtering data on that socket.

You can also detach filters from your socket via the `SO_DETACH_FILTER` option. This will probably not be used much since when you close a socket that has a filter on it the filter is automatically removed. The other less common case may be adding a different filter on the same socket where you had another filter that is still running: the kernel takes care of removing the old one and placing your new one in its place, assuming your filter has passed the checks, otherwise if it fails the old filter will remain on that socket.

`SO_LOCK_FILTER` option allows to lock the filter attached to a socket. Once set, a filter cannot be removed or changed. This allows one process to setup a socket, attach a filter, lock it then drop privileges and be assured that the filter will be kept until the socket is closed.

The biggest user of this construct might be libpcap. Issuing a high-level filter command like `tcpdump -i em1 port 22` passes through the libpcap internal compiler that generates a structure that can eventually be loaded via `SO_ATTACH_FILTER` to the kernel. `tcpdump -i em1 port 22 -ddd` displays what is being placed into this structure.

Although we were only speaking about sockets here, BPF in Linux is used in many more places. There's `xt_bpf` for netfilter, `cls_bpf` in the kernel qdisc layer, SECCOMP-BPF (SECure COMPUTing¹), and lots of other places such as team driver, PTP code, etc where BPF is being used.

¹ Documentation/userspace-api/seccomp_filter.rst

Original BPF paper:

Steven McCanne and Van Jacobson. 1993. The BSD packet filter: a new architecture for user-level packet capture. In Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings (USENIX'93). USENIX Association, Berkeley, CA, USA, 2-2. [<http://www.tcpdump.org/papers/bpf-usenix93.pdf>]

45.3 Structure

User space applications include <linux/filter.h> which contains the following relevant structures:

```
struct sock_filter {      /* Filter block */
    __u16  code;        /* Actual filter code */
    __u8   jt;          /* Jump true */
    __u8   jf;          /* Jump false */
    __u32  k;           /* Generic multiuse field */
};
```

Such a structure is assembled as an array of 4-tuples, that contains a code, jt, jf and k value. jt and jf are jump offsets and k a generic value to be used for a provided code:

```
struct sock_fprog {                      /* Required for S0_ATTACH_FILTER. */
    unsigned short          len; /* Number of filter blocks */
    struct sock_filter __user *filter;
};
```

For socket filtering, a pointer to this structure (as shown in follow-up example) is being passed to the kernel through setsockopt(2).

45.4 Example

```
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <linux/if_ether.h>
/* ... */

/* From the example above: tcpdump -i em1 port 22 -dd */
struct sock_filter code[] = {
    { 0x28,  0,  0, 0x0000000c },
    { 0x15,  0,  8, 0x000086dd },
    { 0x30,  0,  0, 0x00000014 },
    { 0x15,  2,  0, 0x00000084 },
    { 0x15,  1,  0, 0x00000006 },
    { 0x15,  0,  17, 0x00000011 },
    { 0x28,  0,  0, 0x00000036 },
    { 0x15,  14, 0, 0x00000016 },
    { 0x28,  0,  0, 0x00000038 },
```

```

{ 0x15, 12, 13, 0x00000016 },
{ 0x15, 0, 12, 0x00000800 },
{ 0x30, 0, 0, 0x00000017 },
{ 0x15, 2, 0, 0x00000084 },
{ 0x15, 1, 0, 0x00000006 },
{ 0x15, 0, 8, 0x00000011 },
{ 0x28, 0, 0, 0x00000014 },
{ 0x45, 6, 0, 0x00001fff },
{ 0xb1, 0, 0, 0x0000000e },
{ 0x48, 0, 0, 0x0000000e },
{ 0x15, 2, 0, 0x00000016 },
{ 0x48, 0, 0, 0x00000010 },
{ 0x15, 0, 1, 0x00000016 },
{ 0x06, 0, 0, 0x0000ffff },
{ 0x06, 0, 0, 0x00000000 },
};

struct sock_fprog bpf = {
    .len = ARRAY_SIZE(code),
    .filter = code,
};

sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
if (sock < 0)
    /* ... bail out ... */

ret = setsockopt(sock, SOL_SOCKET, SO_ATTACH_FILTER, &bpf, sizeof(bpf));
if (ret < 0)
    /* ... bail out ... */

/* ... */
close(sock);

```

The above example code attaches a socket filter for a PF_PACKET socket in order to let all IPv4/IPv6 packets with port 22 pass. The rest will be dropped for this socket.

The `setsockopt(2)` call to `SO_DETACH_FILTER` doesn't need any arguments and `SO_LOCK_FILTER` for preventing the filter to be detached, takes an integer value with 0 or 1.

Note that socket filters are not restricted to PF_PACKET sockets only, but can also be used on other socket families.

Summary of system calls:

- `setsockopt(sockfd, SOL_SOCKET, SO_ATTACH_FILTER, &val, sizeof(val));`
- `setsockopt(sockfd, SOL_SOCKET, SO_DETACH_FILTER, &val, sizeof(val));`
- `setsockopt(sockfd, SOL_SOCKET, SO_LOCK_FILTER, &val, sizeof(val));`

Normally, most use cases for socket filtering on packet sockets will be covered by libpcap in high-level syntax, so as an application developer you should stick to that. libpcap wraps its own layer around all that.

Unless i) using/linking to libpcap is not an option, ii) the required BPF filters use Linux extensions that are not supported by libpcap's compiler, iii) a filter might be more complex and not cleanly implementable with libpcap's compiler, or iv) particular filter codes should be optimized differently than libpcap's internal compiler does; then in such cases writing such a filter "by hand" can be of an alternative. For example, xt_bpf and cls_bpf users might have requirements that could result in more complex filter code, or one that cannot be expressed with libpcap (e.g. different return codes for various code paths). Moreover, BPF JIT implementors may wish to manually write test cases and thus need low-level access to BPF code as well.

45.5 BPF engine and instruction set

Under tools/bpf/ there's a small helper tool called bpf_asm which can be used to write low-level filters for example scenarios mentioned in the previous section. Asm-like syntax mentioned here has been implemented in bpf_asm and will be used for further explanations (instead of dealing with less readable opcodes directly, principles are the same). The syntax is closely modelled after Steven McCanne's and Van Jacobson's BPF paper.

The BPF architecture consists of the following basic elements:

Element	Description
A	32 bit wide accumulator
X	32 bit wide X register
M[]	16 x 32 bit wide misc registers aka "scratch memory store", addressable from 0 to 15

A program, that is translated by bpf_asm into "opcodes" is an array that consists of the following elements (as already mentioned):

op:16, jt:8, jf:8, k:32

The element op is a 16 bit wide opcode that has a particular instruction encoded. jt and jf are two 8 bit wide jump targets, one for condition "jump if true", the other one "jump if false". Eventually, element k contains a miscellaneous argument that can be interpreted in different ways depending on the given instruction in op.

The instruction set consists of load, store, branch, alu, miscellaneous and return instructions that are also represented in bpf_asm syntax. This table lists all bpf_asm instructions available resp. what their underlying opcodes as defined in linux/filter.h stand for:

Instruction	Addressing mode	Description
ld	1, 2, 3, 4, 12	Load word into A
ldi	4	Load word into A
ldh	1, 2	Load half-word into A
ldb	1, 2	Load byte into A
ldx	3, 4, 5, 12	Load word into X
ldxi	4	Load word into X
ldxb	5	Load byte into X
st	3	Store A into M[]

continues on next page

Table 1 – continued from previous page

Instruction	Addressing mode	Description
stx	3	Store X into M[]
jmp	6	Jump to label
ja	6	Jump to label
jeq	7, 8, 9, 10	Jump on A == <x>
jneq	9, 10	Jump on A != <x>
jne	9, 10	Jump on A != <x>
jlt	9, 10	Jump on A < <x>
jle	9, 10	Jump on A <= <x>
jgt	7, 8, 9, 10	Jump on A > <x>
jge	7, 8, 9, 10	Jump on A >= <x>
jset	7, 8, 9, 10	Jump on A & <x>
add	0, 4	A + <x>
sub	0, 4	A - <x>
mul	0, 4	A * <x>
div	0, 4	A / <x>
mod	0, 4	A % <x>
neg		!A
and	0, 4	A & <x>
or	0, 4	A <x>
xor	0, 4	A ^ <x>
lsh	0, 4	A << <x>
rsh	0, 4	A >> <x>
tax		Copy A into X
txa		Copy X into A
ret	4, 11	Return

The next table shows addressing formats from the 2nd column:

Addressing mode	Syntax	Description
0	x/%x	Register X
1	[k]	BHW at byte offset k in the packet
2	[x + k]	BHW at the offset X + k in the packet
3	M[k]	Word at offset k in M[]
4	#k	Literal value stored in k
5	4*([k]&0xf)	Lower nibble * 4 at byte offset k in the packet
6	L	Jump label L
7	#k,Lt,Lf	Jump to Lt if true, otherwise jump to Lf
8	x/%x,Lt,Lf	Jump to Lt if true, otherwise jump to Lf
9	#k,Lt	Jump to Lt if predicate is true
10	x/%x,Lt	Jump to Lt if predicate is true
11	a/%a	Accumulator A
12	extension	BPF extension

The Linux kernel also has a couple of BPF extensions that are used along with the class of load instructions by “overloading” the k argument with a negative offset + a particular extension offset. The result of such BPF extensions are loaded into A.

Possible BPF extensions are shown in the following table:

Extension	Description
len	skb->len
proto	skb->protocol
type	skb->pkt_type
poff	Payload start offset
ifidx	skb->dev->ifindex
nla	Netlink attribute of type X with offset A
nlan	Nested Netlink attribute of type X with offset A
mark	skb->mark
queue	skb->queue_mapping
hatype	skb->dev->type
rxhash	skb->hash
cpu	raw_smp_processor_id()
vlan_tci	skb_vlan_tag_get(skb)
vlan_avail	skb_vlan_tag_present(skb)
vlan_tpid	skb->vlan_proto
rand	get_random_u32()

These extensions can also be prefixed with '#'. Examples for low-level BPF:

ARP packets:

```
ldh [12]
jne #0x806, drop
ret #-1
drop: ret #0
```

IPv4 TCP packets:

```
ldh [12]
jne #0x800, drop
ldb [23]
jneq #6, drop
ret #-1
drop: ret #0
```

icmp random packet sampling, 1 in 4:

```
ldh [12]
jne #0x800, drop
ldb [23]
jneq #1, drop
# get a random uint32 number
ld rand
mod #4
jneq #1, drop
ret #-1
drop: ret #0
```

SECCOMP filter example:

```

ld [4]          /* offsetof(struct seccomp_data, arch) */
jne #0xc000003e, bad   /* AUDIT_ARCH_X86_64 */
ld [0]          /* offsetof(struct seccomp_data, nr) */
jeq #15, good    /* __NR_rt_sigreturn */
jeq #231, good   /* __NR_exit_group */
jeq #60, good    /* __NR_exit */
jeq #0, good     /* __NR_read */
jeq #1, good     /* __NR_write */
jeq #5, good     /* __NR_fstat */
jeq #9, good     /* __NR_mmap */
jeq #14, good    /* __NR_rt_sigprocmask */
jeq #13, good    /* __NR_rt_sigaction */
jeq #35, good    /* __NR_nanosleep */
bad: ret #0      /* SECCOMP_RET_KILL_THREAD */
good: ret #0x7fff0000 /* SECCOMP_RET_ALLOW */

```

Examples for low-level BPF extension:

Packet for interface index 13:

```

ld ifidx
jneq #13, drop
ret #-1
drop: ret #0

```

(Accelerated) VLAN w/ id 10:

```

ld vlan_tci
jneq #10, drop
ret #-1
drop: ret #0

```

The above example code can be placed into a file (here called "foo"), and then be passed to the `bpf_asm` tool for generating opcodes, output that `xt_bpf` and `cls_bpf` understands and can directly be loaded with. Example with above ARP code:

```
$ ./bpf_asm foo
4,40 0 0 12,21 0 1 2054,6 0 0 4294967295,6 0 0 0,
```

In copy and paste C-like output:

```

$ ./bpf_asm -c foo
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 1, 0x00000806 },
{ 0x06, 0, 0, 0xffffffff },
{ 0x06, 0, 0, 0000000000 },

```

In particular, as usage with `xt_bpf` or `cls_bpf` can result in more complex BPF filters that might not be obvious at first, it's good to test filters before attaching to a live system. For that purpose, there's a small tool called `bpf_dbg` under `tools/bpf/` in the kernel source directory. This debugger allows for testing BPF filters against given pcap files, single stepping through the BPF code on the pcap's packets and to do BPF machine register dumps.

Starting bpf_dbg is trivial and just requires issuing:

```
# ./bpf_dbg
```

In case input and output do not equal stdin/stdout, bpf_dbg takes an alternative stdin source as a first argument, and an alternative stdout sink as a second one, e.g. `./bpf_dbg test_in.txt test_out.txt`.

Other than that, a particular libreadline configuration can be set via file `~/.bpf_dbg_init` and the command history is stored in the file `~/.bpf_dbg_history`.

Interaction in bpf_dbg happens through a shell that also has auto-completion support (follow-up example commands starting with '`>`' denote bpf_dbg shell). The usual workflow would be to ...

- load bpf 6,40 0 0 12,21 0 3 2048,48 0 0 23,21 0 1 1,6 0 0 65535,6 0 0 0 Loads a BPF filter from standard output of bpf_asm, or transformed via e.g. `tcpdump -i em1 -ddd port 22 | tr '\n' ','`. Note that for JIT debugging (next section), this command creates a temporary socket and loads the BPF code into the kernel. Thus, this will also be useful for JIT developers.
- load pcap foo.pcap
Loads standard tcpdump pcap file.
- run [<n>]

bpf passes:1 fails:9

Runs through all packets from a pcap to account how many passes and fails the filter will generate. A limit of packets to traverse can be given.

- disassemble:

```
l0:    ldh [12]
l1:    jeq #0x800, l2, l5
l2:    ldb [23]
l3:    jeq #0x1, l4, l5
l4:    ret #0xffff
l5:    ret #0
```

Prints out BPF code disassembly.

- dump:

```
/* { op, jt, jf, k }, */
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 3, 0x00000800 },
{ 0x30, 0, 0, 0x00000017 },
{ 0x15, 0, 1, 0x00000001 },
{ 0x06, 0, 0, 0x0000ffff },
{ 0x06, 0, 0, 0000000000 },
```

Prints out C-style BPF code dump.

- breakpoint 0:

```
breakpoint at: l0:    ldh [12]
```

- breakpoint 1:

```
breakpoint at: l1:      jeq #0x800, l2, l5
```

...

Sets breakpoints at particular BPF instructions. Issuing a *run* command will walk through the pcap file continuing from the current packet and break when a breakpoint is being hit (another *run* will continue from the currently active breakpoint executing next instructions):

- run:

```
-- register dump --
pc:          [0]                                <-- program counter
code:        [40] jt[0] jf[0] k[12]             <-- plain BPF code of current instruction
curr:        l0:    ldh [12]                  <-- disassembly of current instruction
A:           [00000000][0]                <-- content of A (hex, decimal)
X:           [00000000][0]                <-- content of X (hex, decimal)
M[0,15]:     [00000000][0]              <-- folded content of M (hex, decimal)
-- packet dump --                         <-- Current packet from pcap (hex)
len: 42
  0: 00 19 cb 55 55 a4 00 14 a4 43 78 69 08 06 00 01
16: 08 00 06 04 00 01 00 14 a4 43 78 69 0a 3b 01 26
32: 00 00 00 00 00 00 0a 3b 01 01
(breakpoint)
>
```

- breakpoint:

```
breakpoints: 0 1
```

Prints currently set breakpoints.

- step [-<n>, +<n>]

Performs single stepping through the BPF program from the current pc offset. Thus, on each step invocation, above register dump is issued. This can go forwards and backwards in time, a plain *step* will break on the next BPF instruction, thus +1. (No *run* needs to be issued here.)

- select <n>

Selects a given packet from the pcap file to continue from. Thus, on the next *run* or *step*, the BPF program is being evaluated against the user pre-selected packet. Numbering starts just as in Wireshark with index 1.

- quit

Exits bpf_dbg.

45.6 JIT compiler

The Linux kernel has a built-in BPF JIT compiler for x86_64, SPARC, PowerPC, ARM, ARM64, MIPS, RISC-V and s390 and can be enabled through CONFIG_BPF_JIT. The JIT compiler is transparently invoked for each attached filter from user space or for internal kernel users if it has been previously enabled by root:

```
echo 1 > /proc/sys/net/core/bpf_jit_enable
```

For JIT developers, doing audits etc, each compile run can output the generated opcode image into the kernel log via:

```
echo 2 > /proc/sys/net/core/bpf_jit_enable
```

Example output from dmesg:

```
[ 3389.935842]flen=6 proglen=70 pass=3 image=fffffffffffa0069c8f
[ 3389.935847]JIT code: 00000000: 55 48 89 e5 48 83 ec 60 48 89 5d f8 44 8b
 ↳4f 68
[ 3389.935849]JIT code: 00000010: 44 2b 4f 6c 4c 8b 87 d8 00 00 00 be 0c 00
 ↳00 00
[ 3389.935850]JIT code: 00000020: e8 1d 94 ff e0 3d 00 08 00 00 75 16 be 17
 ↳00 00
[ 3389.935851]JIT code: 00000030: 00 e8 28 94 ff e0 83 f8 01 75 07 b8 ff ff
 ↳00 00
[ 3389.935852]JIT code: 00000040: eb 02 31 c0 c9 c3
```

When CONFIG_BPF_JIT_ALWAYS_ON is enabled, bpf_jit_enable is permanently set to 1 and setting any other value than that will return in failure. This is even the case for setting bpf_jit_enable to 2, since dumping the final JIT image into the kernel log is discouraged and introspection through bpftool (under tools/bpf/bpftool/) is the generally recommended approach instead.

In the kernel source tree under tools/bpf/, there's bpf_jit_disasm for generating disassembly out of the kernel log's hexdump:

```
# ./bpf_jit_disasm
70 bytes emitted from JIT compiler (pass:3,flen:6)
fffffffffffa0069c8f + <x>:
0:    push    %rbp
1:    mov     %rsp,%rbp
4:    sub     $0x60,%rsp
8:    mov     %rbx,-0x8(%rbp)
c:    mov     0x68(%rdi),%r9d
10:   sub    0x6c(%rdi),%r9d
14:   mov     0xd8(%rdi),%r8
1b:   mov     $0xc,%esi
20:   callq   0xfffffffffe0ff9442
25:   cmp     $0x800,%eax
2a:   jne    0x0000000000000042
2c:   mov     $0x17,%esi
31:   callq   0xfffffffffe0ff945e
```

```

36:    cmp    $0x1,%eax
39:    jne    0x0000000000000042
3b:    mov    $0xffff,%eax
40:    jmp    0x0000000000000044
42:    xor    %eax,%eax
44:    leaveq
45:    retq

```

Issuing option ` -o` will "annotate" opcodes to resulting assembler instructions, which can be very useful for JIT developers:

```

# ./bpf_jit_disasm -o
70 bytes emitted from JIT compiler (pass:3, flen:6)
fffffffffa0069c8f + <x>:
0:    push   %rbp
      55
1:    mov    %rsp,%rbp
      48 89 e5
4:    sub    $0x60,%rsp
      48 83 ec 60
8:    mov    %rbx,-0x8(%rbp)
      48 89 5d f8
c:    mov    0x68(%rdi),%r9d
      44 8b 4f 68
10:   sub   0x6c(%rdi),%r9d
      44 2b 4f 6c
14:   mov    0xd8(%rdi),%r8
      4c 8b 87 d8 00 00 00
1b:   mov    $0xc,%esi
      be 0c 00 00 00
20:   callq 0xfffffffffe0ff9442
      e8 1d 94 ff e0
25:   cmp    $0x800,%eax
      3d 00 08 00 00
2a:   jne    0x0000000000000042
      75 16
2c:   mov    $0x17,%esi
      be 17 00 00 00
31:   callq 0xfffffffffe0ff945e
      e8 28 94 ff e0
36:   cmp    $0x1,%eax
      83 f8 01
39:   jne    0x0000000000000042
      75 07
3b:   mov    $0xffff,%eax
      b8 ff ff 00 00
40:   jmp    0x0000000000000044
      eb 02
42:   xor    %eax,%eax
      31 c0

```

```

44:    leaveq
      c9
45:    retq
      c3

```

For BPF JIT developers, `bpf_jit_disasm`, `bpf_asm` and `bpf_dbg` provides a useful toolchain for developing and testing the kernel's JIT compiler.

45.7 BPF kernel internals

Internally, for the kernel interpreter, a different instruction set format with similar underlying principles from BPF described in previous paragraphs is being used. However, the instruction set format is modelled closer to the underlying architecture to mimic native instruction sets, so that a better performance can be achieved (more details later). This new ISA is called eBPF. See the `../bpf/index.rst` for details. (Note: eBPF which originates from [e]xtended BPF is not the same as BPF extensions! While eBPF is an ISA, BPF extensions date back to classic BPF's 'overloading' of `BPF_LD` | `BPF_{B,H,W}` | `BPF_ABS` instruction.)

The new instruction set was originally designed with the possible goal in mind to write programs in "restricted C" and compile into eBPF with a optional GCC/LLVM backend, so that it can just-in-time map to modern 64-bit CPUs with minimal performance overhead over two steps, that is, C -> eBPF -> native code.

Currently, the new format is being used for running user BPF programs, which includes sec-comp BPF, classic socket filters, `cls_bpf` traffic classifier, team driver's classifier for its load-balancing mode, netfilter's `xt_bpf` extension, PTP dissector/classifier, and much more. They are all internally converted by the kernel into the new instruction set representation and run in the eBPF interpreter. For in-kernel handlers, this all works transparently by using `bpf_prog_create()` for setting up the filter, resp. `bpf_prog_destroy()` for destroying it. The function `bpf_prog_run(filter, ctx)` transparently invokes eBPF interpreter or JITed code to run the filter. 'filter' is a pointer to struct `bpf_prog` that we got from `bpf_prog_create()`, and 'ctx' the given context (e.g. `skb` pointer). All constraints and restrictions from `bpf_check_classic()` apply before a conversion to the new layout is being done behind the scenes!

Currently, the classic BPF format is being used for JITing on most 32-bit architectures, whereas x86-64, aarch64, s390x, powerpc64, sparc64, arm32, riscv64, riscv32, loongarch64 perform JIT compilation from eBPF instruction set.

45.8 Testing

Next to the BPF toolchain, the kernel also ships a test module that contains various test cases for classic and eBPF that can be executed against the BPF interpreter and JIT compiler. It can be found in `lib/test_bpf.c` and enabled via Kconfig:

```
CONFIG_TEST_BPF=m
```

After the module has been built and installed, the test suite can be executed via `insmod` or `modprobe` against '`test_bpf`' module. Results of the test cases including timings in nsec can be found in the kernel log (`dmesg`).

45.9 Misc

Also trinity, the Linux syscall fuzzer, has built-in support for BPF and SECCOMP-BPF kernel fuzzing.

45.10 Written by

The document was written in the hope that it is found useful and in order to give potential BPF hackers or security auditors a better overview of the underlying architecture.

- Jay Schlist <jschlst@samba.org>
- Daniel Borkmann <daniel@iogearbox.net>
- Alexei Starovoitov <ast@kernel.org>

GENERIC HDLC LAYER

Krzysztof Halasa <khc@pm.waw.pl>

Generic HDLC layer currently supports:

1. Frame Relay (ANSI, CCITT, Cisco and no LMI)
 - Normal (routed) and Ethernet-bridged (Ethernet device emulation) interfaces can share a single PVC.
 - ARP support (no InARP support in the kernel - there is an experimental InARP user-space daemon available on: <http://www.kernel.org/pub/linux/utils/net/hdlc/>).
2. raw HDLC - either IP (IPv4) interface or Ethernet device emulation
3. Cisco HDLC
4. PPP
5. X.25 (uses X.25 routines).

Generic HDLC is a protocol driver only - it needs a low-level driver for your particular hardware. Ethernet device emulation (using HDLC or Frame-Relay PVC) is compatible with IEEE 802.1Q (VLANs) and 802.1D (Ethernet bridging).

Make sure the hdlc.o and the hardware driver are loaded. It should create a number of "hdlc" (hdlc0 etc) network devices, one for each WAN port. You'll need the "seth DLC" utility, get it from:

<http://www.kernel.org/pub/linux/utils/net/hdlc/>

Compile seth DLC.c utility:

```
gcc -O2 -Wall -o seth DLC seth DLC.c
```

Make sure you're using a correct version of seth DLC for your kernel.

Use seth DLC to set physical interface, clock rate, HDLC mode used, and add any required PVCs if using Frame Relay. Usually you want something like:

```
seth DLC hdlc0 clock int rate 128000
seth DLC hdlc0 cisco interval 10 timeout 25
```

or:

```
seth DLC hdlc0 rs232 clock ext
seth DLC hdlc0 fr lmi ansi
seth DLC hdlc0 create 99
```

```
ifconfig hdlc0 up
ifconfig pvc0 localIP pointopoint remoteIP
```

In Frame Relay mode, ifconfig master hdlc device up (without assigning any IP address to it) before using pvc devices.

Setting interface:

- **v35 | rs232 | x21 | t1 | e1**

- sets physical interface for a given port if the card has software-selectable interfaces

- **loopback**

- activate hardware loopback (for testing only)

- **clock ext**

- both RX clock and TX clock external

- **clock int**

- both RX clock and TX clock internal

- **clock txint**

- RX clock external, TX clock internal

- **clock txfromrx**

- RX clock external, TX clock derived from RX clock

- **rate**

- sets clock rate in bps (for "int" or "txint" clock only)

Setting protocol:

- **hdlc** - sets raw HDLC (IP-only) mode

nrz / nrzi / fm-mark / fm-space / manchester - sets transmission code

no-parity / crc16 / crc16-pr0 (CRC16 with preset zeros) / crc32-itu

crc16-itu (CRC16 with ITU-T polynomial) / crc16-itu-pr0 - sets parity

- **hdlc-eth** - Ethernet device emulation using HDLC. Parity and encoding as above.

- **cisco** - sets Cisco HDLC mode (IP, IPv6 and IPX supported)

interval - time in seconds between keepalive packets

timeout - time in seconds after last received keepalive packet before

we assume the link is down

- **ppp** - sets synchronous PPP mode

- **x25** - sets X.25 mode

- **fr** - Frame Relay mode

lmi ansi / ccitt / cisco / none - LMI (link management) type

dce - Frame Relay DCE (network) side LMI instead of default DTE (user).

It has nothing to do with clocks!

- t391 - link integrity verification polling timer (in seconds) - user
- t392 - polling verification timer (in seconds) - network
- n391 - full status polling counter - user
- n392 - error threshold - both user and network
- n393 - monitored events count - both user and network

Frame-Relay only:

- create n | delete n - adds / deletes PVC interface with DLCI #n. Newly created interface will be named pvc0, pvc1 etc.
- create ether n | delete ether n - adds a device for Ethernet-bridged frames. The device will be named pvceth0, pvceth1 etc.

46.1 Board-specific issues

n2.o and c101.o need parameters to work:

```
insmod n2 hw=io,irq,ram,ports[:io,irq,...]
```

example:

```
insmod n2 hw=0x300,10,0xD0000,01
```

or:

```
insmod c101 hw=irq,ram[:irq,...]
```

example:

```
insmod c101 hw=9,0xdc000
```

If built into the kernel, these drivers need kernel (command line) parameters:

```
n2.hw=io,irq,ram,ports:...
```

or:

```
c101.hw=irq,ram:...
```

If you have a problem with N2, C101 or PLX200SYN card, you can issue the "private" command to see port's packet descriptor rings (in kernel logs):

```
sethdlc hdlc0 private
```

The hardware driver has to be build with `#define DEBUG_RINGS`. Attaching this info to bug reports would be helpful. Anyway, let me know if you have problems using this.

For patches and other info look at: <<http://www.kernel.org/pub/linux/utils/net/hdcl/>>.

CHAPTER
FORTYSEVEN

GENERIC NETLINK

A wiki document on how to use Generic Netlink can be found here:

- https://wiki.linuxfoundation.org/networking/generic_netlink_howto

GENERIC NETWORKING STATISTICS FOR NETLINK USERS

Statistic counters are grouped into structs:

Struct	TLV type	Description
gnet_stats_basic	TCA_STATS_BASIC	Basic statistics
gnet_stats_rate_est	TCA_STATS_RATE_EST	Rate estimator
gnet_stats_queue	TCA_STATS_QUEUE	Queue statistics
none	TCA_STATS_APP	Application specific

48.1 Collecting:

Declare the statistic structs you need:

```
struct mystruct {
    struct gnet_stats_basic bstats;
    struct gnet_stats_queue qstats;
    ...
};
```

Update statistics, in dequeue() methods only, (while owning qdisc->running):

```
mystruct->tstats.packet++;
mystruct->qstats.backlog += skb->pkt_len;
```

48.2 Export to userspace (Dump):

```
my_dumping_routine(struct sk_buff *skb, ...)
{
    struct gnet_dump dump;

    if (gnet_stats_start_copy(skb, TCA_STATS2, &mystruct->lock, &dump,
                             TCA_PAD) < 0)
        goto rtattr_failure;

    if (gnet_stats_copy_basic(&dump, &mystruct->bstats) < 0 ||
        gnet_stats_copy_queue(&dump, &mystruct->qstats) < 0 ||
```

```

        gnet_stats_copy_app(&dump, &xstats, sizeof(xstats)) < 0)
        goto rtattr_failure;

    if (gnet_stats_finish_copy(&dump) < 0)
        goto rtattr_failure;
    ...
}

```

48.3 TCA_STATS/TCA_XSTATS backward compatibility:

Prior users of struct tc_stats and xstats can maintain backward compatibility by calling the compat wrappers to keep providing the existing TLV types:

```

my_dumping_routine(struct sk_buff *skb, ...)
{
    if (gnet_stats_start_copy_compat(skb, TCA_STATS2, TCA_STATS,
                                     TCA_XSTATS, &mystruct->lock, &dump,
                                     TCA_PAD) < 0)
        goto rtattr_failure;
    ...
}

```

A struct tc_stats will be filled out during gnet_stats_copy_* calls and appended to the skb. TCA_XSTATS is provided if gnet_stats_copy_app was called.

48.4 Locking:

Locks are taken before writing and released once all statistics have been written. Locks are always released in case of an error. You are responsible for making sure that the lock is initialized.

48.5 Rate Estimator:

- 0) Prepare an estimator attribute. Most likely this would be in user space. The value of this TLV should contain a tc_estimator structure. As usual, such a TLV needs to be 32 bit aligned and therefore the length needs to be appropriately set, etc. The estimator interval and ewma log need to be converted to the appropriate values. tc_estimator.c::tc_setup_estimator() is advisable to be used as the conversion routine. It does a few clever things. It takes a time interval in microsecs, a time constant also in microsecs and a struct tc_estimator to be populated. The returned tc_estimator can be transported to the kernel. Transfer such a structure in a TLV of type TCA_RATE to your code in the kernel.

In the kernel when setting up:

- 1) make sure you have basic stats and rate stats setup first.
- 2) make sure you have initialized stats lock that is used to setup such stats.

3) Now initialize a new estimator:

```
int ret = gen_new_estimator(my_basicstats,my_rate_est_stats,  
    mystats_lock, attr_with_tcestimator_struct);  
  
if ret == 0  
    success  
else  
    failed
```

From now on, every time you dump my_rate_est_stats it will contain up-to-date info.

Once you are done, call gen_kill_estimator(my_basicstats, my_rate_est_stats) Make sure that my_basicstats and my_rate_est_stats are still valid (i.e still exist) at the time of making this call.

48.6 Authors:

- Thomas Graf <tgraf@suug.ch>
- Jamal Hadi Salim <hadi@cyberus.ca>

THE LINUX KERNEL GTP TUNNELING MODULE

Documentation by

Harald Welte <laforge@gnumonks.org> and Andreas Schultz <aschultz@tpip.net>

In 'drivers/net/gtp.c' you are finding a kernel-level implementation of a GTP tunnel endpoint.

49.1 What is GTP

GTP is the Generic Tunnel Protocol, which is a 3GPP protocol used for tunneling User-IP payload between a mobile station (phone, modem) and the interconnection between an external packet data network (such as the internet).

So when you start a 'data connection' from your mobile phone, the phone will use the control plane to signal for the establishment of such a tunnel between that external data network and the phone. The tunnel endpoints thus reside on the phone and in the gateway. All intermediate nodes just transport the encapsulated packet.

The phone itself does not implement GTP but uses some other technology-dependent protocol stack for transmitting the user IP payload, such as LLC/SNDCP/RLC/MAC.

At some network element inside the cellular operator infrastructure (SGSN in case of GPRS/EGPRS or classic UMTS, hNodeB in case of a 3G femtocell, eNodeB in case of 4G/LTE), the cellular protocol stacking is translated into GTP *without breaking the end-to-end tunnel*. So intermediate nodes just perform some specific relay function.

At some point the GTP packet ends up on the so-called GGSN (GSM/UMTS) or P-GW (LTE), which terminates the tunnel, decapsulates the packet and forwards it onto an external packet data network. This can be public internet, but can also be any private IP network (or even theoretically some non-IP network like X.25).

You can find the protocol specification in 3GPP TS 29.060, available publicly via the 3GPP website at <http://www.3gpp.org/DynaReport/29060.htm>

A direct PDF link to v13.6.0 is provided for convenience below: http://www.etsi.org/deliver/etsi_ts/129000_129099/129060/13.06.00_60/ts_129060v130600p.pdf

49.2 The Linux GTP tunnelling module

The module implements the function of a tunnel endpoint, i.e. it is able to decapsulate tunneled IP packets in the uplink originated by the phone, and encapsulate raw IP packets received from the external packet network in downlink towards the phone.

It *only* implements the so-called 'user plane', carrying the User-IP payload, called GTP-U. It does not implement the 'control plane', which is a signaling protocol used for establishment and teardown of GTP tunnels (GTP-C).

So in order to have a working GGSN/P-GW setup, you will need a userspace program that implements the GTP-C protocol and which then uses the netlink interface provided by the GTP-U module in the kernel to configure the kernel module.

This split architecture follows the tunneling modules of other protocols, e.g. PPPoE or L2TP, where you also run a userspace daemon to handle the tunnel establishment, authentication etc. and only the data plane is accelerated inside the kernel.

Don't be confused by terminology: The GTP User Plane goes through kernel accelerated path, while the GTP Control Plane goes to Userspace :)

The official homepage of the module is at <https://osmocom.org/projects/linux-kernel-gtp-u/wiki>

49.3 Userspace Programs with Linux Kernel GTP-U support

At the time of this writing, there are at least two Free Software implementations that implement GTP-C and can use the netlink interface to make use of the Linux kernel GTP-U support:

- OpenGGSN (classic 2G/3G GGSN in C): <https://osmocom.org/projects/openggsn/wiki/OpenGGSN>
- ergw (GGSN + P-GW in Erlang): <https://github.com/travelping/ergw>

49.4 Userspace Library / Command Line Utilities

There is a userspace library called 'libgtpnl' which is based on libmnl and which implements a C-language API towards the netlink interface provided by the Kernel GTP module:

<http://git.osmocom.org/libgtpnl/>

49.5 Protocol Versions

There are two different versions of GTP-U: v0 [GSM TS 09.60] and v1 [3GPP TS 29.281]. Both are implemented in the Kernel GTP module. Version 0 is a legacy version, and deprecated from recent 3GPP specifications.

GTP-U uses UDP for transporting PDUs. The receiving UDP port is 2151 for GTPv1-U and 3386 for GTPv0-U.

There are three versions of GTP-C: v0, v1, and v2. As the kernel doesn't implement GTP-C, we don't have to worry about this. It's the responsibility of the control plane implementation in userspace to implement that.

49.6 IPv6

The 3GPP specifications indicate either IPv4 or IPv6 can be used both on the inner (user) IP layer, or on the outer (transport) layer.

Unfortunately, the Kernel module currently supports IPv6 neither for the User IP payload, nor for the outer IP layer. Patches or other Contributions to fix this are most welcome!

49.7 Mailing List

If you have questions regarding how to use the Kernel GTP module from your own software, or want to contribute to the code, please use the [osmocom-net-grps mailing list](mailto:osmocom-net-grps@lists.osmocom.org) for related discussion. The list can be reached at osmocom-net-gprs@lists.osmocom.org and the mailman interface for managing your subscription is at <https://lists.osmocom.org/mailman/listinfo/osmocom-net-gprs>

49.8 Issue Tracker

The Osmocom project maintains an issue tracker for the Kernel GTP-U module at <https://osmocom.org/projects/linux-kernel-gtp-u/issues>

49.9 History / Acknowledgements

The Module was originally created in 2012 by Harald Welte, but never completed. Pablo came in to finish the mess Harald left behind. But due to a lack of user interest, it never got merged.

In 2015, Andreas Schultz came to the rescue and fixed lots more bugs, extended it with new features and finally pushed all of us to get it mainline, where it was merged in 4.7.0.

49.10 Architectural Details

49.10.1 Local GTP-U entity and tunnel identification

GTP-U uses UDP for transporting PDU's. The receiving UDP port is 2152 for GTPv1-U and 3386 for GTPv0-U.

There is only one GTP-U entity (and therefore SGSN/GGSN/S-GW/PDN-GW instance) per IP address. Tunnel Endpoint Identifier (TEID) are unique per GTP-U entity.

A specific tunnel is only defined by the destination entity. Since the destination port is constant, only the destination IP and TEID define a tunnel. The source IP and Port have no meaning for the tunnel.

Therefore:

- when sending, the remote entity is defined by the remote IP and the tunnel endpoint id. The source IP and port have no meaning and can be changed at any time.

- when receiving the local entity is defined by the local destination IP and the tunnel endpoint id. The source IP and port have no meaning and can change at any time.

[3GPP TS 29.281] Section 4.3.0 defines this so:

The TEID in the GTP-U header is used to de-multiplex traffic incoming from remote tunnel endpoints so that it is delivered to the User plane entities in a way that allows multiplexing of different users, different packet protocols and different QoS levels. Therefore no two remote GTP-U endpoints shall send traffic to a GTP-U protocol entity using the same TEID value except for data forwarding as part of mobility procedures.

The definition above only defines that two remote GTP-U endpoints *should not* send to the same TEID, it *does not* forbid or exclude such a scenario. In fact, the mentioned mobility procedures make it necessary that the GTP-U entity accepts traffic for TEIDs from multiple or unknown peers.

Therefore, the receiving side identifies tunnels exclusively based on TEIDs, not based on the source IP!

49.11 APN vs. Network Device

The GTP-U driver creates a Linux network device for each Gi/SGi interface.

[3GPP TS 29.281] calls the Gi/SGi reference point an interface. This may lead to the impression that the GGSN/P-GW can have only one such interface.

Correct is that the Gi/SGi reference point defines the interworking between +the 3GPP packet domain (PDN) based on GTP-U tunnel and IP based networks.

There is no provision in any of the 3GPP documents that limits the number of Gi/SGi interfaces implemented by a GGSN/P-GW.

[3GPP TS 29.061] Section 11.3 makes it clear that the selection of a specific Gi/SGi interfaces is made through the Access Point Name (APN):

2. each private network manages its own addressing. In general this will result in different private networks having overlapping address ranges. A logically separate connection (e.g. an IP in IP tunnel or layer 2 virtual circuit) is used between the GGSN/P-GW and each private network.

In this case the IP address alone is not necessarily unique. The pair of values, Access Point Name (APN) and IPv4 address and/or IPv6 prefixes, is unique.

In order to support the overlapping address range use case, each APN is mapped to a separate Gi/SGi interface (network device).

Note: The Access Point Name is purely a control plane (GTP-C) concept. At the GTP-U level, only Tunnel Endpoint Identifiers are present in GTP-U packets and network devices are known

Therefore for a given UE the mapping in IP to PDN network is:

- network device + MS IP -> Peer IP + Peer TEID,

and from PDN to IP network:

- local GTP-U IP + TEID -> network device

Furthermore, before a received T-PDU is injected into the network device the MS IP is checked against the IP recorded in PDP context.

IDENTIFIER LOCATOR ADDRESSING (ILA)

50.1 Introduction

Identifier-locator addressing (ILA) is a technique used with IPv6 that differentiates between location and identity of a network node. Part of an address expresses the immutable identity of the node, and another part indicates the location of the node which can be dynamic. Identifier-locator addressing can be used to efficiently implement overlay networks for network virtualization as well as solutions for use cases in mobility.

ILA can be thought of as means to implement an overlay network without encapsulation. This is accomplished by performing network address translation on destination addresses as a packet traverses a network. To the network, an ILA translated packet appears to be no different than any other IPv6 packet. For instance, if the transport protocol is TCP then an ILA translated packet looks like just another TCP/IPv6 packet. The advantage of this is that ILA is transparent to the network so that optimizations in the network, such as ECMP, RSS, GRO, GSO, etc., just work.

The ILA protocol is described in Internet-Draft [draft-herbert-intarea-ila](#).

50.2 ILA terminology

- **Identifier**

A number that identifies an addressable node in the network independent of its location. ILA identifiers are sixty-four bit values.

- **Locator**

A network prefix that routes to a physical host. Locators provide the topological location of an addressed node. ILA locators are sixty-four bit prefixes.

- **ILA mapping**

A mapping of an ILA identifier to a locator (or to a locator and meta data). An ILA domain maintains a database that contains mappings for all destinations in the domain.

- **SIR address**

An IPv6 address composed of a SIR prefix (upper sixty-four bits) and an identifier (lower sixty-four bits). SIR addresses are visible to applications and provide a means for them to address nodes independent of their location.

- **ILA address**

An IPv6 address composed of a locator (upper sixty-four bits) and an identifier (low order sixty-four bits). ILA addresses are never visible to an application.

- **ILA host**
An end host that is capable of performing ILA translations on transmit or receive.
- **ILA router**
A network node that performs ILA translation and forwarding of translated packets.
- **ILA forwarding cache**
A type of ILA router that only maintains a working set cache of mappings.
- **ILA node**
A network node capable of performing ILA translations. This can be an ILA router, ILA forwarding cache, or ILA host.

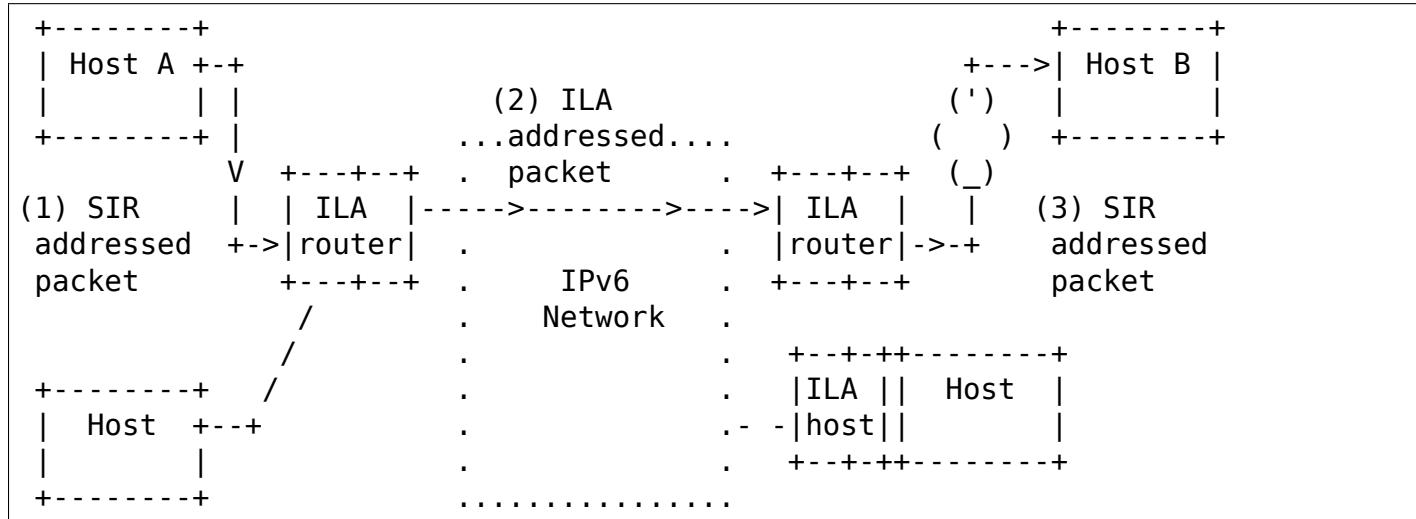
50.3 Operation

There are two fundamental operations with ILA:

- Translate a SIR address to an ILA address. This is performed on ingress to an ILA overlay.
- Translate an ILA address to a SIR address. This is performed on egress from the ILA overlay.

ILA can be deployed either on end hosts or intermediate devices in the network; these are provided by "ILA hosts" and "ILA routers" respectively. Configuration and datapath for these two points of deployment is somewhat different.

The diagram below illustrates the flow of packets through ILA as well as showing ILA hosts and routers:



50.4 Transport checksum handling

When an address is translated by ILA, an encapsulated transport checksum that includes the translated address in a pseudo header may be rendered incorrect on the wire. This is a problem for intermediate devices, including checksum offload in NICs, that process the checksum. There are three options to deal with this:

- **no action Allow the checksum to be incorrect on the wire.** Before a receiver verifies a checksum the ILA to SIR address translation must be done.
- **adjust transport checksum**
When ILA translation is performed the packet is parsed and if a transport layer checksum is found then it is adjusted to reflect the correct checksum per the translated address.
- **checksum neutral mapping**
When an address is translated the difference can be offset elsewhere in a part of the packet that is covered by the checksum. The low order sixteen bits of the identifier are used. This method is preferred since it doesn't require parsing a packet beyond the IP header and in most cases the adjustment can be precomputed and saved with the mapping.

Note that the checksum neutral adjustment affects the low order sixteen bits of the identifier. When ILA to SIR address translation is done on egress the low order bits are restored to the original value which restores the identifier as it was originally sent.

50.5 Identifier types

ILA defines different types of identifiers for different use cases.

The defined types are:

- 0: interface identifier
- 1: locally unique identifier
- 2: virtual networking identifier for IPv4 address
- 3: virtual networking identifier for IPv6 unicast address
- 4: virtual networking identifier for IPv6 multicast address
- 5: non-local address identifier

In the current implementation of kernel ILA only locally unique identifiers (LUID) are supported. LUID allows for a generic, unformatted 64 bit identifier.

50.6 Identifier formats

Kernel ILA supports two optional fields in an identifier for formatting: "C-bit" and "identifier type". The presence of these fields is determined by configuration as demonstrated below.

If the identifier type is present it occupies the three highest order bits of an identifier. The possible values are given in the above list.

If the C-bit is present, this is used as an indication that checksum neutral mapping has been done. The C-bit can only be set in an ILA address, never a SIR address.

In the simplest format the identifier types, C-bit, and checksum adjustment value are not present so an identifier is considered an unstructured sixty-four bit value:

```
+-----+
|           Identifier           |
+-----+
|-----+
|-----+
|-----+
```

The checksum neutral adjustment may be configured to always be present using neutral-map-auto. In this case there is no C-bit, but the checksum adjustment is in the low order 16 bits. The identifier is still sixty-four bits:

```
+-----+
|           Identifier           |
+-----+
|-----+
|           | Checksum-neutral adjustment |
+-----+
```

The C-bit may be used to explicitly indicate that checksum neutral mapping has been applied to an ILA address. The format is:

```
+-----+
|   |C|           Identifier           |
|   +-+-----+
|           | Checksum-neutral adjustment |
+-----+
```

The identifier type field may be present to indicate the identifier type. If it is not present then the type is inferred based on mapping configuration. The checksum neutral adjustment may automatically be used with the identifier type as illustrated below:

```
+-----+
| Type|           Identifier           |
+-----+
|           | Checksum-neutral adjustment |
+-----+
```

If the identifier type and the C-bit can be present simultaneously so the identifier format would be:

Type C	Identifier
	+-----+ Checksum-neutral adjustment +-----+

50.7 Configuration

There are two methods to configure ILA mappings. One is by using LWT routes and the other is `ila_xlat` (called from NFHOOK PREROUTING hook). `ila_xlat` is intended to be used in the receive path for ILA hosts .

An ILA router has also been implemented in XDP. Description of that is outside the scope of this document.

The usage of for ILA LWT routes is:

```
ip route add DEST/128 encapsulate ila LOC csum-mode MODE ident-type TYPE via ADDR
```

Destination (DEST) can either be a SIR address (for an ILA host or ingress ILA router) or an ILA address (egress ILA router). LOC is the sixty-four bit locator (with format W:X:Y:Z) that overwrites the upper sixty-four bits of the destination address. Checksum MODE is one of "no-action", "adj-transport", "neutral-map", and "neutral-map-auto". If neutral-map is set then the C-bit will be present. Identifier TYPE one of "luid" or "use-format." In the case of use-format, the identifier type field is present and the effective type is taken from that.

The usage of `ila_xlat` is:

```
ip ila add loc_match MATCH loc LOC csum-mode MODE ident-type TYPE
```

MATCH indicates the incoming locator that must be matched to apply a the translaiton. LOC is the locator that overwrites the upper sixty-four bits of the destination address. MODE and TYPE have the same meanings as described above.

50.8 Some examples

```
# Configure an ILA route that uses checksum neutral mapping as well
# as type field. Note that the type field is set in the SIR address
# (the 2000 implies type is 1 which is LUID).
ip route add 3333:0:0:1:2000:0:1:87/128 encapsulate ila 2001:0:87:0 \
    csum-mode neutral-map ident-type use-format

# Configure an ILA LWT route that uses auto checksum neutral mapping
# (no C-bit) and configure identifier type to be LUID so that the
# identifier type field will not be present.
ip route add 3333:0:0:1:2000:0:2:87/128 encapsulate ila 2001:0:87:1 \
    csum-mode neutral-map-auto ident-type luid

ila_xlat configuration
```

```
# Configure an ILA to SIR mapping that matches a locator and overwrites
# it with a SIR address (3333:0:0:1 in this example). The C-bit and
# identifier field are used.
ip ila add loc_match 2001:0:119:0 loc 3333:0:0:1 \
    csum-mode neutral-map-auto ident-type use-format

# Configure an ILA to SIR mapping where checksum neutral is automatically
# set without the C-bit and the identifier type is configured to be LUID
# so that the identifier type field is not present.
ip ila add loc_match 2001:0:119:0 loc 3333:0:0:1 \
    csum-mode neutral-map-auto ident-type use-format
```

IOAM6 SYSFS VARIABLES

51.1 /proc/sys/net/conf/<iface>/ioam6_* variables:

ioam6_enabled - BOOL

Accept (= enabled) or ignore (= disabled) IPv6 IOAM options on ingress for this interface.

- 0 - disabled (default)
- 1 - enabled

ioam6_id - SHORT INTEGER

Define the IOAM id of this interface.

Default is ~0.

ioam6_id_wide - INTEGER

Define the wide IOAM id of this interface.

Default is ~0.

IP DYNAMIC ADDRESS HACK-PORT V0.03

This stuff allows diald ONESHOT connections to get established by dynamically changing packet source address (and socket's if local procs). It is implemented for TCP diald-box connections(1) and IP_MASQuerading(2).

If enabled¹ and forwarding interface has changed:

- 1) Socket (and packet) source address is rewritten ON RETRANSMISSIONS while in SYN_SENT state (diald-box processes).
- 2) Out-bounded MASQueraded source address changes ON OUTPUT (when internal host does retransmission) until a packet from outside is received by the tunnel.

This is specially helpful for auto dialup links (diald), where the actual outgoing address is unknown at the moment the link is going up. So, the *same* (local AND masqueraded) connections requests that bring the link up will be able to get established.

Enjoy!

Juanjo <jjciarla@raiz.uncu.edu.ar>

¹ At boot, by default no address rewriting is attempted.
To enable:

```
# echo 1 > /proc/sys/net/ipv4/ip_dynaddr
```

To enable verbose mode:

```
# echo 2 > /proc/sys/net/ipv4/ip_dynaddr
```

To disable (default):

```
# echo 0 > /proc/sys/net/ipv4/ip_dynaddr
```

**CHAPTER
FIFTYTHREE**

IPSEC

Here documents known IPsec corner cases which need to be keep in mind when deploy various IPsec configuration in real world production environment.

1. IPcomp:

Small IP packet won't get compressed at sender, and failed on policy check on receiver.

Quote from RFC3173:

2.2. Non-Expansion Policy

If the total size of a compressed payload and the IPComp header, as defined in section 3, is not smaller than the size of the original payload, the IP datagram MUST be sent in the original non-compressed form. To clarify: If an IP datagram is sent non-compressed, no

IPComp header is added to the datagram. This policy ensures saving the decompression processing cycles and avoiding incurring IP datagram fragmentation when the expanded datagram is larger than the MTU.

Small IP datagrams are likely to expand as a result of compression. Therefore, a numeric threshold should be applied before compression, where IP datagrams of size smaller than the threshold are sent in the original form without attempting compression. The numeric threshold is implementation dependent.

Current IPComp implementation is indeed by the book, while as in practice when sending non-compressed packet to the peer (whether or not packet len is smaller than the threshold or the compressed len is larger than original packet len), the packet is dropped when checking the policy as this packet matches the selector but not coming from any XFRM layer, i.e., with no security path. Such naked packet will not eventually make it to upper layer. The result is much more wired to the user when ping peer with different payload length.

One workaround is try to set "level use" for each policy if user observed above scenario. The consequence of doing so is small packet(uncompressed) will skip policy checking on receiver side.

IP SYSCTL

54.1 /proc/sys/net/ipv4/* Variables

ip_forward - BOOLEAN

- 0 - disabled (default)
- not 0 - enabled

Forward Packets between interfaces.

This variable is special, its change resets all configuration parameters to their default state (RFC1122 for hosts, RFC1812 for routers)

ip_default_ttl - INTEGER

Default value of TTL field (Time To Live) for outgoing (but not forwarded) IP packets. Should be between 1 and 255 inclusive. Default: 64 (as recommended by RFC1700)

ip_no_pmtu_disc - INTEGER

Disable Path MTU Discovery. If enabled in mode 1 and a fragmentation-required ICMP is received, the PMTU to this destination will be set to the smallest of the old MTU to this destination and min_pmtu (see below). You will need to raise min_pmtu to the smallest interface MTU on your system manually if you want to avoid locally generated fragments.

In mode 2 incoming Path MTU Discovery messages will be discarded. Outgoing frames are handled the same as in mode 1, implicitly setting IP_PMTUDISC_DONT on every created socket.

Mode 3 is a hardened pmtu discover mode. The kernel will only accept fragmentation-needed errors if the underlying protocol can verify them besides a plain socket lookup. Current protocols for which pmtu events will be honored are TCP, SCTP and DCCP as they verify e.g. the sequence number or the association. This mode should not be enabled globally but is only intended to secure e.g. name servers in namespaces where TCP path mtu must still work but path MTU information of other protocols should be discarded. If enabled globally this mode could break other protocols.

Possible values: 0-3

Default: FALSE

min_pmtu - INTEGER

default 552 - minimum Path MTU. Unless this is changed manually, each cached pmtu will never be lower than this setting.

ip_forward_use_pmtu - BOOLEAN

By default we don't trust protocol path MTUs while forwarding because they could be easily forged and can lead to unwanted fragmentation by the router. You only need to enable this if you have user-space software which tries to discover path mtus by itself and depends on the kernel honoring this information. This is normally not the case.

Default: 0 (disabled)

Possible values:

- 0 - disabled
- 1 - enabled

fwmark_reflect - BOOLEAN

Controls the fwmark of kernel-generated IPv4 reply packets that are not associated with a socket for example, TCP RSTs or ICMP echo replies). If unset, these packets have a fwmark of zero. If set, they have the fwmark of the packet they are replying to.

Default: 0

fib_multipath_use_neigh - BOOLEAN

Use status of existing neighbor entry when determining nexthop for multipath routes. If disabled, neighbor information is not used and packets could be directed to a failed nexthop. Only valid for kernels built with CONFIG_IP_ROUTE_MULTIPATH enabled.

Default: 0 (disabled)

Possible values:

- 0 - disabled
- 1 - enabled

fib_multipath_hash_policy - INTEGER

Controls which hash policy to use for multipath routes. Only valid for kernels built with CONFIG_IP_ROUTE_MULTIPATH enabled.

Default: 0 (Layer 3)

Possible values:

- 0 - Layer 3
- 1 - Layer 4
- 2 - Layer 3 or inner Layer 3 if present
- 3 - Custom multipath hash. Fields used for multipath hash calculation are determined by fib_multipath_hash_fields sysctl

fib_multipath_hash_fields - UNSIGNED INTEGER

When fib_multipath_hash_policy is set to 3 (custom multipath hash), the fields used for multipath hash calculation are determined by this sysctl.

This value is a bitmask which enables various fields for multipath hash calculation.

Possible fields are:

0x0001	Source IP address
0x0002	Destination IP address
0x0004	IP protocol
0x0008	Unused (Flow Label)
0x0010	Source port
0x0020	Destination port
0x0040	Inner source IP address
0x0080	Inner destination IP address
0x0100	Inner IP protocol
0x0200	Inner Flow Label
0x0400	Inner source port
0x0800	Inner destination port

Default: 0x0007 (source IP, destination IP and IP protocol)

fib_sync_mem - UNSIGNED INTEGER

Amount of dirty memory from fib entries that can be backlogged before synchronize_rcu is forced.

Default: 512kB Minimum: 64kB Maximum: 64MB

ip_forward_update_priority - INTEGER

Whether to update SKB priority from "TOS" field in IPv4 header after it is forwarded. The new SKB priority is mapped from TOS field value according to an rt_tos2priority table (see e.g. man tc-prio).

Default: 1 (Update priority.)

Possible values:

- 0 - Do not update priority.
- 1 - Update priority.

route/max_size - INTEGER

Maximum number of routes allowed in the kernel. Increase this when using large numbers of interfaces and/or routes.

From linux kernel 3.6 onwards, this is deprecated for ipv4 as route cache is no longer used.

From linux kernel 6.3 onwards, this is deprecated for ipv6 as garbage collection manages cached route entries.

neigh/default/gc_thresh1 - INTEGER

Minimum number of entries to keep. Garbage collector will not purge entries if there are fewer than this number.

Default: 128

neigh/default/gc_thresh2 - INTEGER

Threshold when garbage collector becomes more aggressive about purging entries. Entries older than 5 seconds will be cleared when over this number.

Default: 512

neigh/default/gc_thresh3 - INTEGER

Maximum number of non-PERMANENT neighbor entries allowed. Increase this when us-

ing large numbers of interfaces and when communicating with large numbers of directly-connected peers.

Default: 1024

neigh/default/unres_qlen_bytes - INTEGER

The maximum number of bytes which may be used by packets queued for each unresolved address by other network layers. (added in linux 3.3)

Setting negative value is meaningless and will return error.

Default: SK_WMEM_MAX, (same as net.core.wmem_default).

Exact value depends on architecture and kernel options, but should be enough to allow queuing 256 packets of medium size.

neigh/default/unres_qlen - INTEGER

The maximum number of packets which may be queued for each unresolved address by other network layers.

(deprecated in linux 3.3) : use unres_qlen_bytes instead.

Prior to linux 3.3, the default value is 3 which may cause unexpected packet loss. The current default value is calculated according to default value of unres_qlen_bytes and true size of packet.

Default: 101

neigh/default/interval_probe_time_ms - INTEGER

The probe interval for neighbor entries with NTF_MANAGED flag, the min value is 1.

Default: 5000

mtu_expires - INTEGER

Time, in seconds, that cached PMTU information is kept.

min_adv_mss - INTEGER

The advertised MSS depends on the first hop route MTU, but will never be lower than this setting.

fib_notify_on_flag_change - INTEGER

Whether to emit RTM_NEWRROUTE notifications whenever RTM_F_OFFLOAD/RTM_F_TRAP/RTM_F_OFFLOAD_FAILED flags are changed.

After installing a route to the kernel, user space receives an acknowledgment, which means the route was installed in the kernel, but not necessarily in hardware. It is also possible for a route already installed in hardware to change its action and therefore its flags. For example, a host route that is trapping packets can be "promoted" to perform decapsulation following the installation of an IPinIP/VXLAN tunnel. The notifications will indicate to user-space the state of the route.

Default: 0 (Do not emit notifications.)

Possible values:

- 0 - Do not emit notifications.
- 1 - Emit notifications.
- 2 - Emit notifications only for RTM_F_OFFLOAD_FAILED flag change.

IP Fragmentation:

ipfrag_high_thresh - LONG INTEGER

Maximum memory used to reassemble IP fragments.

ipfrag_low_thresh - LONG INTEGER

(Obsolete since linux-4.17) Maximum memory used to reassemble IP fragments before the kernel begins to remove incomplete fragment queues to free up resources. The kernel still accepts new fragments for defragmentation.

ipfrag_time - INTEGER

Time in seconds to keep an IP fragment in memory.

ipfrag_max_dist - INTEGER

ipfrag_max_dist is a non-negative integer value which defines the maximum "disorder" which is allowed among fragments which share a common IP source address. Note that reordering of packets is not unusual, but if a large number of fragments arrive from a source IP address while a particular fragment queue remains incomplete, it probably indicates that one or more fragments belonging to that queue have been lost. When ipfrag_max_dist is positive, an additional check is done on fragments before they are added to a reassembly queue - if ipfrag_max_dist (or more) fragments have arrived from a particular IP address between additions to any IP fragment queue using that source address, it's presumed that one or more fragments in the queue are lost. The existing fragment queue will be dropped, and a new one started. An ipfrag_max_dist value of zero disables this check.

Using a very small value, e.g. 1 or 2, for ipfrag_max_dist can result in unnecessarily dropping fragment queues when normal reordering of packets occurs, which could lead to poor application performance. Using a very large value, e.g. 50000, increases the likelihood of incorrectly reassembling IP fragments that originate from different IP datagrams, which could result in data corruption. Default: 64

bc_forwarding - INTEGER

bc_forwarding enables the feature described in rfc1812#section-5.3.5.2 and rfc2644. It allows the router to forward directed broadcast. To enable this feature, the 'all' entry and the input interface entry should be set to 1. Default: 0

54.2 INET peer storage

inet_peer_threshold - INTEGER

The approximate size of the storage. Starting from this threshold entries will be thrown aggressively. This threshold also determines entries' time-to-live and time intervals between garbage collection passes. More entries, less time-to-live, less GC interval.

inet_peer_minttl - INTEGER

Minimum time-to-live of entries. Should be enough to cover fragment time-to-live on the reassembling side. This minimum time-to-live is guaranteed if the pool size is less than inet_peer_threshold. Measured in seconds.

inet_peer_maxttl - INTEGER

Maximum time-to-live of entries. Unused entries will expire after this period of time if there is no memory pressure on the pool (i.e. when the number of entries in the pool is very small). Measured in seconds.

54.3 TCP variables

somaxconn - INTEGER

Limit of socket listen() backlog, known in userspace as SOMAXCONN. Defaults to 4096. (Was 128 before linux-5.4) See also `tcp_max_syn_backlog` for additional tuning for TCP sockets.

tcp_abort_on_overflow - BOOLEAN

If listening service is too slow to accept new connections, reset them. Default state is FALSE. It means that if overflow occurred due to a burst, connection will recover. Enable this option *_only_* if you are really sure that listening daemon cannot be tuned to accept connections faster. Enabling this option can harm clients of your server.

tcp_adv_win_scale - INTEGER

Obsolete since linux-6.6 Count buffering overhead as bytes/2^{tcp_adv_win_scale} (if `tcp_adv_win_scale > 0`) or bytes-bytes/2^(-tcp_adv_win_scale), if it is ≤ 0 .

Possible values are [-31, 31], inclusive.

Default: 1

tcp_allowed_congestion_control - STRING

Show/set the congestion control choices available to non-privileged processes. The list is a subset of those listed in `tcp_available_congestion_control`.

Default is "reno" and the default setting (`tcp_congestion_control`).

tcp_app_win - INTEGER

Reserve max(window/2^{tcp_app_win}, mss) of window for application buffer. Value 0 is special, it means that nothing is reserved.

Possible values are [0, 31], inclusive.

Default: 31

tcp_autocorking - BOOLEAN

Enable TCP auto corking : When applications do consecutive small write()/sendmsg() system calls, we try to coalesce these small writes as much as possible, to lower total amount of sent packets. This is done if at least one prior packet for the flow is waiting in Qdisc queues or device transmit queue. Applications can still use TCP_CORK for optimal behavior when they know how/when to uncork their sockets.

Default : 1

tcp_available_congestion_control - STRING

Shows the available congestion control choices that are registered. More congestion control algorithms may be available as modules, but not loaded.

tcp_base_mss - INTEGER

The initial value of `search_low` to be used by the packetization layer Path MTU discovery (MTU probing). If MTU probing is enabled, this is the initial MSS used by the connection.

tcp_mtu_probe_floor - INTEGER

If MTU probing is enabled this caps the minimum MSS used for `search_low` for the connection.

Default : 48

tcp_min_snd_mss - INTEGER

TCP SYN and SYNACK messages usually advertise an ADVMSS option, as described in RFC 1122 and RFC 6691.

If this ADVMSS option is smaller than `tcp_min_snd_mss`, it is silently capped to `tcp_min_snd_mss`.

Default : 48 (at least 8 bytes of payload per segment)

tcp_congestion_control - STRING

Set the congestion control algorithm to be used for new connections. The algorithm "reno" is always available, but additional choices may be available based on kernel configuration. Default is set as part of kernel configuration. For passive connections, the listener congestion control choice is inherited.

[see `setsockopt(listenfd, SOL_TCP, TCP_CONGESTION, "name" ...)`]

tcp_dsack - BOOLEAN

Allows TCP to send "duplicate" SACKs.

tcp_early_retrans - INTEGER

Tail loss probe (TLP) converts RTOs occurring due to tail losses into fast recovery (draft-ietf-tcpm-rack). Note that TLP requires RACK to function properly (see `tcp_recovery` below)

Possible values:

- 0 disables TLP
- 3 or 4 enables TLP

Default: 3

tcp_ecn - INTEGER

Control use of Explicit Congestion Notification (ECN) by TCP. ECN is used only when both ends of the TCP connection indicate support for it. This feature is useful in avoiding losses due to congestion by allowing supporting routers to signal congestion before having to drop packets.

Possible values are:

0	Disable ECN. Neither initiate nor accept ECN.
1	Enable ECN when requested by incoming connections and also request ECN on outgoing connection attempts.
2	Enable ECN when requested by incoming connections but do not request ECN on outgoing connections.

Default: 2

tcp_ecn_fallback - BOOLEAN

If the kernel detects that ECN connection misbehaves, enable fall back to non-ECN. Currently, this knob implements the fallback from RFC3168, section 6.1.1.1., but we reserve that in future, additional detection mechanisms could be implemented under this knob. The value is not used, if `tcp_ecn` or per route (or congestion control) ECN settings are disabled.

Default: 1 (fallback enabled)

tcp_fack - BOOLEAN

This is a legacy option, it has no effect anymore.

tcp_fin_timeout - INTEGER

The length of time an orphaned (no longer referenced by any application) connection will remain in the FIN_WAIT_2 state before it is aborted at the local end. While a perfectly valid "receive only" state for an un-orphaned connection, an orphaned connection in FIN_WAIT_2 state could otherwise wait forever for the remote to close its end of the connection.

Cf. `tcp_max_orphans`

Default: 60 seconds

tcp_frto - INTEGER

Enables Forward RTO-Recovery (F-RTO) defined in RFC5682. F-RTO is an enhanced recovery algorithm for TCP retransmission timeouts. It is particularly beneficial in networks where the RTT fluctuates (e.g., wireless). F-RTO is sender-side only modification. It does not require any support from the peer.

By default it's enabled with a non-zero value. 0 disables F-RTO.

tcp_fwmark_accept - BOOLEAN

If set, incoming connections to listening sockets that do not have a socket mark will set the mark of the accepting socket to the fwmark of the incoming SYN packet. This will cause all packets on that connection (starting from the first SYNACK) to be sent with that fwmark. The listening socket's mark is unchanged. Listening sockets that already have a fwmark set via `setsockopt(SOL_SOCKET, SO_MARK, ...)` are unaffected.

Default: 0

tcp_invalid_ratelimit - INTEGER

Limit the maximal rate for sending duplicate acknowledgments in response to incoming TCP packets that are for an existing connection but that are invalid due to any of these reasons:

- (a) out-of-window sequence number,
- (b) out-of-window acknowledgment number, or
- (c) PAWS (Protection Against Wrapped Sequence numbers) check failure

This can help mitigate simple "ack loop" DoS attacks, wherein a buggy or malicious middlebox or man-in-the-middle can rewrite TCP header fields in manner that causes each endpoint to think that the other is sending invalid TCP segments, thus causing each side to send an unterminating stream of duplicate acknowledgments for invalid segments.

Using 0 disables rate-limiting of dupacks in response to invalid segments; otherwise this value specifies the minimal space between sending such dupacks, in milliseconds.

Default: 500 (milliseconds).

tcp_keepalive_time - INTEGER

How often TCP sends out keepalive messages when keepalive is enabled. Default: 2hours.

tcp_keepalive_probes - INTEGER

How many keepalive probes TCP sends out, until it decides that the connection is broken. Default value: 9.

tcp_keepalive_intvl - INTEGER

How frequently the probes are send out. Multiplied by `tcp_keepalive_probes` it is time to kill not responding connection, after probes started. Default value: 75sec i.e. connection will be aborted after ~11 minutes of retries.

tcp_l3mdev_accept - BOOLEAN

Enables child sockets to inherit the L3 master device index. Enabling this option allows a "global" listen socket to work across L3 master domains (e.g., VRFs) with connected sockets derived from the listen socket to be bound to the L3 domain in which the packets originated. Only valid when the kernel was compiled with `CONFIG_NET_L3_MASTER_DEV`.

Default: 0 (disabled)

tcp_low_latency - BOOLEAN

This is a legacy option, it has no effect anymore.

tcp_max_orphans - INTEGER

Maximal number of TCP sockets not attached to any user file handle, held by system. If this number is exceeded orphaned connections are reset immediately and warning is printed. This limit exists only to prevent simple DoS attacks, you `_must_` not rely on this or lower the limit artificially, but rather increase it (probably, after increasing installed memory), if network conditions require more than default value, and tune network services to linger and kill such states more aggressively. Let me to remind again: each orphan eats up to ~64K of unswappable memory.

tcp_max_syn_backlog - INTEGER

Maximal number of remembered connection requests (SYN_RECV), which have not received an acknowledgment from connecting client.

This is a per-listener limit.

The minimal value is 128 for low memory machines, and it will increase in proportion to the memory of machine.

If server suffers from overload, try increasing this number.

Remember to also check `/proc/sys/net/core/somaxconn` A SYN_RECV request socket consumes about 304 bytes of memory.

tcp_max_tw_buckets - INTEGER

Maximal number of timewait sockets held by system simultaneously. If this number is exceeded time-wait socket is immediately destroyed and warning is printed. This limit exists only to prevent simple DoS attacks, you `_must_` not lower the limit artificially, but rather increase it (probably, after increasing installed memory), if network conditions require more than default value.

tcp_mem - vector of 3 INTEGERs: min, pressure, max

min: below this number of pages TCP is not bothered about its memory appetite.

pressure: when amount of memory allocated by TCP exceeds this number of pages, TCP moderates its memory consumption and enters memory pressure mode, which is exited when memory consumption falls under "min".

max: number of pages allowed for queueing by all TCP sockets.

Defaults are calculated at boot time from amount of available memory.

tcp_min_rtt_wlen - INTEGER

The window length of the windowed min filter to track the minimum RTT. A shorter window

lets a flow more quickly pick up new (higher) minimum RTT when it is moved to a longer path (e.g., due to traffic engineering). A longer window makes the filter more resistant to RTT inflations such as transient congestion. The unit is seconds.

Possible values: 0 - 86400 (1 day)

Default: 300

tcp_moderate_rcvbuf - BOOLEAN

If set, TCP performs receive buffer auto-tuning, attempting to automatically size the buffer (no greater than `tcp_rmem[2]`) to match the size required by the path for full throughput. Enabled by default.

tcp_mtu_probing - INTEGER

Controls TCP Packetization-Layer Path MTU Discovery. Takes three values:

- 0 - Disabled
- 1 - Disabled by default, enabled when an ICMP black hole detected
- 2 - Always enabled, use initial MSS of `tcp_base_mss`.

tcp_probe_interval - UNSIGNED INTEGER

Controls how often to start TCP Packetization-Layer Path MTU Discovery reprobe. The default is reprobining every 10 minutes as per RFC4821.

tcp_probe_threshold - INTEGER

Controls when TCP Packetization-Layer Path MTU Discovery probing will stop in respect to the width of search range in bytes. Default is 8 bytes.

tcp_no_metrics_save - BOOLEAN

By default, TCP saves various connection metrics in the route cache when the connection closes, so that connections established in the near future can use these to set initial conditions. Usually, this increases overall performance, but may sometimes cause performance degradation. If set, TCP will not cache metrics on closing connections.

tcp_no_ssthresh_metrics_save - BOOLEAN

Controls whether TCP saves ssthresh metrics in the route cache.

Default is 1, which disables ssthresh metrics.

tcp_orphan_retries - INTEGER

This value influences the timeout of a locally closed TCP connection, when RTO retransmissions remain unacknowledged. See `tcp_retries2` for more details.

The default value is 8.

If your machine is a loaded WEB server, you should think about lowering this value, such sockets may consume significant resources. Cf. `tcp_max_orphans`.

tcp_recovery - INTEGER

This value is a bitmap to enable various experimental loss recovery features.

RACK: 0x1	enables the RACK loss detection for fast detection of lost retransmissions and tail drops. It also subsumes and disables RFC6675 recovery for SACK connections.
RACK: 0x2	makes RACK's reordering window static (<code>min_rtt/4</code>).
RACK: 0x4	disables RACK's DUPACK threshold heuristic

Default: 0x1

tcp_reflect_tos - BOOLEAN

For listening sockets, reuse the DSCP value of the initial SYN message for outgoing packets. This allows to have both directions of a TCP stream to use the same DSCP value, assuming DSCP remains unchanged for the lifetime of the connection.

This option affects both IPv4 and IPv6.

Default: 0 (disabled)

tcp_reordering - INTEGER

Initial reordering level of packets in a TCP stream. TCP stack can then dynamically adjust flow reordering level between this initial value and `tcp_max_reordering`

Default: 3

tcp_max_reordering - INTEGER

Maximal reordering level of packets in a TCP stream. 300 is a fairly conservative value, but you might increase it if paths are using per packet load balancing (like bonding rr mode)

Default: 300

tcp_retrans_collapse - BOOLEAN

Bug-to-bug compatibility with some broken printers. On retransmit try to send bigger packets to work around bugs in certain TCP stacks.

tcp_retries1 - INTEGER

This value influences the time, after which TCP decides, that something is wrong due to unacknowledged RTO retransmissions, and reports this suspicion to the network layer. See `tcp_retries2` for more details.

RFC 1122 recommends at least 3 retransmissions, which is the default.

tcp_retries2 - INTEGER

This value influences the timeout of an alive TCP connection, when RTO retransmissions remain unacknowledged. Given a value of N, a hypothetical TCP connection following exponential backoff with an initial RTO of `TCP_RTO_MIN` would retransmit N times before killing the connection at the (N+1)th RTO.

The default value of 15 yields a hypothetical timeout of 924.6 seconds and is a lower bound for the effective timeout. TCP will effectively time out at the first RTO which exceeds the hypothetical timeout.

RFC 1122 recommends at least 100 seconds for the timeout, which corresponds to a value of at least 8.

tcp_rfc1337 - BOOLEAN

If set, the TCP stack behaves conforming to RFC1337. If unset, we are not conforming to RFC, but prevent TCP TIME_WAIT assassination.

Default: 0

tcp_rmem - vector of 3 INTEGERs: min, default, max

`min`: Minimal size of receive buffer used by TCP sockets. It is guaranteed to each TCP socket, even under moderate memory pressure.

Default: 4K

default: initial size of receive buffer used by TCP sockets. This value overrides net.core.rmem_default used by other protocols. Default: 131072 bytes. This value results in initial window of 65535.

max: maximal size of receive buffer allowed for automatically selected receiver buffers for TCP socket. This value does not override net.core.rmem_max. Calling setsockopt() with SO_RCVBUF disables automatic tuning of that socket's receive buffer size, in which case this value is ignored. Default: between 131072 and 6MB, depending on RAM size.

tcp_sack - BOOLEAN

Enable select acknowledgments (SACKS).

tcp_comp_sack_delay_ns - LONG INTEGER

TCP tries to reduce number of SACK sent, using a timer based on 5% of SRTT, capped by this sysctl, in nano seconds. The default is 1ms, based on TSO autosizing period.

Default : 1,000,000 ns (1 ms)

tcp_comp_sack_slack_ns - LONG INTEGER

This sysctl control the slack used when arming the timer used by SACK compression. This gives extra time for small RTT flows, and reduces system overhead by allowing opportunistic reduction of timer interrupts.

Default : 100,000 ns (100 us)

tcp_comp_sack_nr - INTEGER

Max number of SACK that can be compressed. Using 0 disables SACK compression.

Default : 44

tcp_backlog_ack_defer - BOOLEAN

If set, user thread processing socket backlog tries sending one ACK for the whole queue. This helps to avoid potential long latencies at end of a TCP socket syscall.

Default : true

tcp_slow_start_after_idle - BOOLEAN

If set, provide RFC2861 behavior and time out the congestion window after an idle period. An idle period is defined at the current RTO. If unset, the congestion window will not be timed out after an idle period.

Default: 1

tcp_stdurg - BOOLEAN

Use the Host requirements interpretation of the TCP urgent pointer field. Most hosts use the older BSD interpretation, so if you turn this on Linux might not communicate correctly with them.

Default: FALSE

tcp_synack_retries - INTEGER

Number of times SYNACKs for a passive TCP connection attempt will be retransmitted. Should not be higher than 255. Default value is 5, which corresponds to 31seconds till the last retransmission with the current initial RTO of 1second. With this the final timeout for a passive TCP connection will happen after 63seconds.

tcp_syncookies - INTEGER

Only valid when the kernel was compiled with CONFIG_SYN_COOKIES Send out syncooki-

ies when the syn backlog queue of a socket overflows. This is to prevent against the common 'SYN flood attack' Default: 1

Note, that syncookies is fallback facility. It MUST NOT be used to help highly loaded servers to stand against legal connection rate. If you see SYN flood warnings in your logs, but investigation shows that they occur because of overload with legal connections, you should tune another parameters until this warning disappear. See: `tcp_max_syn_backlog`, `tcp_synack_retries`, `tcp_abort_on_overflow`.

syncookies seriously violate TCP protocol, do not allow to use TCP extensions, can result in serious degradation of some services (f.e. SMTP relaying), visible not by you, but your clients and relays, contacting you. While you see SYN flood warnings in logs not being really flooded, your server is seriously misconfigured.

If you want to test which effects syncookies have to your network connections you can set this knob to 2 to enable unconditionally generation of syncookies.

tcp_migrate_req - BOOLEAN

The incoming connection is tied to a specific listening socket when the initial SYN packet is received during the three-way handshake. When a listener is closed, in-flight request sockets during the handshake and established sockets in the accept queue are aborted.

If the listener has `SO_REUSEPORT` enabled, other listeners on the same port should have been able to accept such connections. This option makes it possible to migrate such child sockets to another listener after `close()` or `shutdown()`.

The `BPF_SK_REUSEPORT_SELECT_OR_MIGRATE` type of eBPF program should usually be used to define the policy to pick an alive listener. Otherwise, the kernel will randomly pick an alive listener only if this option is enabled.

Note that migration between listeners with different settings may crash applications. Let's say migration happens from listener A to B, and only B has `TCP_SAVE_SYN` enabled. B cannot read SYN data from the requests migrated from A. To avoid such a situation, cancel migration by returning `SK_DROP` in the type of eBPF program, or disable this option.

Default: 0

tcp_fastopen - INTEGER

Enable TCP Fast Open (RFC7413) to send and accept data in the opening SYN packet.

The client support is enabled by flag 0x1 (on by default). The client then must use `sendmsg()` or `sendto()` with the `MSG_FASTOPEN` flag, rather than `connect()` to send data in SYN.

The server support is enabled by flag 0x2 (off by default). Then either enable for all listeners with another flag (0x400) or enable individual listeners via `TCP_FASTOPEN` socket option with the option value being the length of the syn-data backlog.

The values (bitmap) are

0x1	(client)	enables sending data in the opening SYN on the client.
0x2	(server)	enables the server support, i.e., allowing data in a SYN packet to be accepted and passed to the application before 3-way handshake finishes.
0x4	(client)	send data in the opening SYN regardless of cookie availability and without a cookie option.
0x200	(server)	accept data-in-SYN w/o any cookie option present.
0x400	(server)	enable all listeners to support Fast Open by default without explicit TCP_FASTOPEN socket option.

Default: 0x1

Note that additional client or server features are only effective if the basic support (0x1 and 0x2) are enabled respectively.

tcp_fastopen_blackhole_timeout_sec - INTEGER

Initial time period in second to disable Fastopen on active TCP sockets when a TFO firewall blackhole issue happens. This time period will grow exponentially when more blackhole issues get detected right after Fastopen is re-enabled and will reset to initial value when the blackhole issue goes away. 0 to disable the blackhole detection.

By default, it is set to 0 (feature is disabled).

tcp_fastopen_key - list of comma separated 32-digit hexadecimal INTEGERS

The list consists of a primary key and an optional backup key. The primary key is used for both creating and validating cookies, while the optional backup key is only used for validating cookies. The purpose of the backup key is to maximize TFO validation when keys are rotated.

A randomly chosen primary key may be configured by the kernel if the `tcp_fastopen` sysctl is set to 0x400 (see above), or if the `TCP_FASTOPEN` `setsockopt()` optname is set and a key has not been previously configured via sysctl. If keys are configured via `setsockopt()` by using the `TCP_FASTOPEN_KEY` optname, then those per-socket keys will be used instead of any keys that are specified via sysctl.

A key is specified as 4 8-digit hexadecimal integers which are separated by a '-' as: xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx. Leading zeros may be omitted. A primary and a backup key may be specified by separating them by a comma. If only one key is specified, it becomes the primary key and any previously configured backup keys are removed.

tcp_syn_retries - INTEGER

Number of times initial SYNs for an active TCP connection attempt will be retransmitted. Should not be higher than 127. Default value is 6, which corresponds to 67seconds (with `tcp_syn_linear_timeouts = 4`) till the last retransmission with the current initial RTO of 1second. With this the final timeout for an active TCP connection attempt will happen after 131seconds.

tcp_timestamps - INTEGER

Enable timestamps as defined in RFC1323.

- 0: Disabled.
- 1: Enable timestamps as defined in RFC1323 and use random offset for each connection rather than only using the current time.
- 2: Like 1, but without random offsets.

Default: 1

tcp_min_tso_segs - INTEGER

Minimal number of segments per TSO frame.

Since linux-3.12, TCP does an automatic sizing of TSO frames, depending on flow rate, instead of filling 64Kbytes packets. For specific usages, it's possible to force TCP to build big TSO frames. Note that TCP stack might split too big TSO packets if available window is too small.

Default: 2

tcp_tso_rtt_log - INTEGER

Adjustment of TSO packet sizes based on min_rtt

Starting from linux-5.18, TCP autosizing can be tweaked for flows having small RTT.

Old autosizing was splitting the pacing budget to send 1024 TSO per second.

tso_packet_size = sk->sk_pacing_rate / 1024;

With the new mechanism, we increase this TSO sizing using:

```
distance = min_rtt_usecs / (2^tcp_tso_rtt_log) tso_packet_size += gso_max_size >> distance;
```

This means that flows between very close hosts can use bigger TSO packets, reducing their cpu costs.

If you want to use the old autosizing, set this sysctl to 0.

Default: 9 ($2^9 = 512$ usec)

tcp_pacing_ss_ratio - INTEGER

sk->sk_pacing_rate is set by TCP stack using a ratio applied to current rate. (current_rate = cwnd * mss / srtt) If TCP is in slow start, tcp_pacing_ss_ratio is applied to let TCP probe for bigger speeds, assuming cwnd can be doubled every other RTT.

Default: 200

tcp_pacing_ca_ratio - INTEGER

sk->sk_pacing_rate is set by TCP stack using a ratio applied to current rate. (current_rate = cwnd * mss / srtt) If TCP is in congestion avoidance phase, tcp_pacing_ca_ratio is applied to conservatively probe for bigger throughput.

Default: 120

tcp_syn_linear_timeouts - INTEGER

The number of times for an active TCP connection to retransmit SYNs with a linear backoff timeout before defaulting to an exponential backoff timeout. This has no effect on SYNACK at the passive TCP side.

With an initial RTO of 1 and `tcp_syn_linear_timeouts = 4` we would expect SYN RTOs to be: 1, 1, 1, 1, 1, 2, 4, ... (4 linear timeouts, and the first exponential backoff using $2^0 * \text{initial_RTO}$). Default: 4

tcp_tso_win_divisor - INTEGER

This allows control over what percentage of the congestion window can be consumed by a single TSO frame. The setting of this parameter is a choice between burstiness and building larger TSO frames.

Default: 3

tcp_tw_reuse - INTEGER

Enable reuse of TIME-WAIT sockets for new connections when it is safe from protocol viewpoint.

- 0 - disable
- 1 - global enable
- 2 - enable for loopback traffic only

It should not be changed without advice/request of technical experts.

Default: 2

tcp_window_scaling - BOOLEAN

Enable window scaling as defined in RFC1323.

tcp_shrink_window - BOOLEAN

This changes how the TCP receive window is calculated.

RFC 7323, section 2.4, says there are instances when a retracted window can be offered, and that TCP implementations MUST ensure that they handle a shrinking window, as specified in RFC 1122.

- 0 - Disabled. The window is never shrunk.
- 1 - Enabled. **The window is shrunk when necessary to remain within** the memory limit set by autotuning (sk_rcvbuf). This only occurs if a non-zero receive window scaling factor is also in effect.

Default: 0

tcp_wmem - vector of 3 INTEGERs: min, default, max

min: Amount of memory reserved for send buffers for TCP sockets. Each TCP socket has rights to use it due to fact of its birth.

Default: 4K

default: initial size of send buffer used by TCP sockets. This value overrides net.core.wmem_default used by other protocols.

It is usually lower than net.core.wmem_default.

Default: 16K

max: Maximal amount of memory allowed for automatically tuned send buffers for TCP sockets. This value does not override net.core.wmem_max. Calling setsockopt() with SO_SNDBUF disables automatic tuning of that socket's send buffer size, in which case this value is ignored.

Default: between 64K and 4MB, depending on RAM size.

tcp_notsent_lowat - UNSIGNED INTEGER

A TCP socket can control the amount of unsent bytes in its write queue, thanks to TCP_NOTSENT_LOWAT socket option. poll()/select()/epoll() reports POLLOUT events if the amount of unsent bytes is below a per socket value, and if the write queue is not full. sendmsg() will also not add new buffers if the limit is hit.

This global variable controls the amount of unsent data for sockets not using TCP_NOTSENT_LOWAT. For these sockets, a change to the global variable has immediate effect.

Default: UINT_MAX (0xFFFFFFFF)

tcp_workaround_signed_windows - BOOLEAN

If set, assume no receipt of a window scaling option means the remote TCP is broken and treats the window as a signed quantity. If unset, assume the remote TCP is not broken even if we do not receive a window scaling option from them.

Default: 0

tcp_thin_linear_timeouts - BOOLEAN

Enable dynamic triggering of linear timeouts for thin streams. If set, a check is performed upon retransmission by timeout to determine if the stream is thin (less than 4 packets in flight). As long as the stream is found to be thin, up to 6 linear timeouts may be performed before exponential backoff mode is initiated. This improves retransmission latency for non-aggressive thin streams, often found to be time-dependent. For more information on thin streams, see [Thin-streams and TCP](#)

Default: 0

tcp_limit_output_bytes - INTEGER

Controls TCP Small Queue limit per tcp socket. TCP bulk sender tends to increase packets in flight until it gets losses notifications. With SNDBUF autotuning, this can result in a large amount of packets queued on the local machine (e.g.: qdiscs, CPU backlog, or device) hurting latency of other flows, for typical pfifo_fast qdiscs. `tcp_limit_output_bytes` limits the number of bytes on qdisc or device to reduce artificial RTT/cwnd and reduce bufferbloat.

Default: 1048576 (16 * 65536)

tcp_challenge_ack_limit - INTEGER

Limits number of Challenge ACK sent per second, as recommended in RFC 5961 (Improving TCP's Robustness to Blind In-Window Attacks) Note that this per netns rate limit can allow some side channel attacks and probably should not be enabled. TCP stack implements per TCP socket limits anyway. Default: INT_MAX (unlimited)

tcp_ehash_entries - INTEGER

Show the number of hash buckets for TCP sockets in the current networking namespace.

A negative value means the networking namespace does not own its hash buckets and shares the initial networking namespace's one.

tcp_child_ehash_entries - INTEGER

Control the number of hash buckets for TCP sockets in the child networking namespace, which must be set before clone() or unshare().

If the value is not 0, the kernel uses a value rounded up to 2^n as the actual hash bucket size. 0 is a special value, meaning the child networking namespace will share the initial networking namespace's hash buckets.

Note that the child will use the global one in case the kernel fails to allocate enough memory. In addition, the global hash buckets are spread over available NUMA nodes, but the allocation of the child hash table depends on the current process's NUMA policy, which could result in performance differences.

Note also that the default value of `tcp_max_tw_buckets` and `tcp_max_syn_backlog` depend on the hash bucket size.

Possible values: 0, 2^n (n: 0 - 24 (16Mi))

Default: 0

`tcp_plb_enabled` - BOOLEAN

If set and the underlying congestion control (e.g. DCTCP) supports and enables PLB feature, TCP PLB (Protective Load Balancing) is enabled. PLB is described in the following paper: <https://doi.org/10.1145/3544216.3544226>. Based on PLB parameters, upon sensing sustained congestion, TCP triggers a change in flow label field for outgoing IPv6 packets. A change in flow label field potentially changes the path of outgoing packets for switches that use ECMP/WCMP for routing.

PLB changes socket txhash which results in a change in IPv6 Flow Label field, and currently no-op for IPv4 headers. It is possible to apply PLB for IPv4 with other network header fields (e.g. TCP or IPv4 options) or using encapsulation where outer header is used by switches to determine next hop. In either case, further host and switch side changes will be needed.

When set, PLB assumes that congestion signal (e.g. ECN) is made available and used by congestion control module to estimate a congestion measure (e.g. `ce_ratio`). PLB needs a congestion measure to make repathing decisions.

Default: FALSE

`tcp_plb_idle_rehash_rounds` - INTEGER

Number of consecutive congested rounds (RTT) seen after which a rehash can be performed, given there are no packets in flight. This is referred to as M in PLB paper: <https://doi.org/10.1145/3544216.3544226>.

Possible Values: 0 - 31

Default: 3

`tcp_plb_rehash_rounds` - INTEGER

Number of consecutive congested rounds (RTT) seen after which a forced rehash can be performed. Be careful when setting this parameter, as a small value increases the risk of retransmissions. This is referred to as N in PLB paper: <https://doi.org/10.1145/3544216.3544226>.

Possible Values: 0 - 31

Default: 12

`tcp_plb_suspend_rto_sec` - INTEGER

Time, in seconds, to suspend PLB in event of an RTO. In order to avoid having PLB repath onto a connectivity "black hole", after an RTO a TCP connection suspends PLB repathing for a random duration between 1x and 2x of this parameter. Randomness is added to avoid concurrent rehashing of multiple TCP connections. This should be set corresponding to the amount of time it takes to repair a failed link.

Possible Values: 0 - 255

Default: 60

`tcp_plb_cong_thresh` - INTEGER

Fraction of packets marked with congestion over a round (RTT) to tag that round as

congested. This is referred to as K in the PLB paper: <https://doi.org/10.1145/3544216.3544226>.

The 0-1 fraction range is mapped to 0-256 range to avoid floating point operations. For example, 128 means that if at least 50% of the packets in a round were marked as congested then the round will be tagged as congested.

Setting threshold to 0 means that PLB repaths every RTT regardless of congestion. This is not intended behavior for PLB and should be used only for experimentation purpose.

Possible Values: 0 - 256

Default: 128

tcp_pingpong_thresh - INTEGER

The number of estimated data replies sent for estimated incoming data requests that must happen before TCP considers that a connection is a "ping-pong" (request-response) connection for which delayed acknowledgments can provide benefits.

This threshold is 1 by default, but some applications may need a higher threshold for optimal performance.

Possible Values: 1 - 255

Default: 1

54.4 UDP variables

udp_l3mdev_accept - BOOLEAN

Enabling this option allows a "global" bound socket to work across L3 master domains (e.g., VRFs) with packets capable of being received regardless of the L3 domain in which they originated. Only valid when the kernel was compiled with CONFIG_NET_L3_MASTER_DEV.

Default: 0 (disabled)

udp_mem - vector of 3 INTEGERS: min, pressure, max

Number of pages allowed for queueing by all UDP sockets.

min: Number of pages allowed for queueing by all UDP sockets.

pressure: This value was introduced to follow format of tcp_mem.

max: This value was introduced to follow format of tcp_mem.

Default is calculated at boot time from amount of available memory.

udp_rmem_min - INTEGER

Minimal size of receive buffer used by UDP sockets in moderation. Each UDP socket is able to use the size for receiving data, even if total pages of UDP sockets exceed udp_mem pressure. The unit is byte.

Default: 4K

udp_wmem_min - INTEGER

UDP does not have tx memory accounting and this tunable has no effect.

udp_hash_entries - INTEGER

Show the number of hash buckets for UDP sockets in the current networking namespace.

A negative value means the networking namespace does not own its hash buckets and shares the initial networking namespace's one.

udp_child_ehash_entries - INTEGER

Control the number of hash buckets for UDP sockets in the child networking namespace, which must be set before clone() or unshare().

If the value is not 0, the kernel uses a value rounded up to 2^n as the actual hash bucket size. 0 is a special value, meaning the child networking namespace will share the initial networking namespace's hash buckets.

Note that the child will use the global one in case the kernel fails to allocate enough memory. In addition, the global hash buckets are spread over available NUMA nodes, but the allocation of the child hash table depends on the current process's NUMA policy, which could result in performance differences.

Possible values: 0, 2^n (n: 7 (128) - 16 (64K))

Default: 0

54.5 RAW variables

raw_l3mdev_accept - BOOLEAN

Enabling this option allows a "global" bound socket to work across L3 master domains (e.g., VRFs) with packets capable of being received regardless of the L3 domain in which they originated. Only valid when the kernel was compiled with CONFIG_NET_L3_MASTER_DEV.

Default: 1 (enabled)

54.6 CIPSOv4 Variables

cipso_cache_enable - BOOLEAN

If set, enable additions to and lookups from the CIPSO label mapping cache. If unset, additions are ignored and lookups always result in a miss. However, regardless of the setting the cache is still invalidated when required when means you can safely toggle this on and off and the cache will always be "safe".

Default: 1

cipso_cache_bucket_size - INTEGER

The CIPSO label cache consists of a fixed size hash table with each hash bucket containing a number of cache entries. This variable limits the number of entries in each hash bucket; the larger the value is, the more CIPSO label mappings that can be cached. When the number of entries in a given hash bucket reaches this limit adding new entries causes the oldest entry in the bucket to be removed to make room.

Default: 10

cipso_rbm_optfmt - BOOLEAN

Enable the "Optimized Tag 1 Format" as defined in section 3.4.2.6 of the CIPSO draft specification (see Documentation/netlabel for details). This means that when set the CIPSO tag will be padded with empty categories in order to make the packet data 32-bit aligned.

Default: 0

cipso_rbm_structvalid - BOOLEAN

If set, do a very strict check of the CIPSO option when ip_options_compile() is called. If unset, relax the checks done during ip_options_compile(). Either way is "safe" as errors are caught elsewhere in the CIPSO processing code but setting this to 0 (False) should result in less work (i.e. it should be faster) but could cause problems with other implementations that require strict checking.

Default: 0

54.7 IP Variables

ip_local_port_range - 2 INTEGERS

Defines the local port range that is used by TCP and UDP to choose the local port. The first number is the first, the second the last local port number. If possible, it is better these numbers have different parity (one even and one odd value). Must be greater than or equal to ip_unprivileged_port_start. The default values are 32768 and 60999 respectively.

ip_local_reserved_ports - list of comma separated ranges

Specify the ports which are reserved for known third-party applications. These ports will not be used by automatic port assignments (e.g. when calling connect() or bind() with port number 0). Explicit port allocation behavior is unchanged.

The format used for both input and output is a comma separated list of ranges (e.g. "1,2-4,10-10" for ports 1, 2, 3, 4 and 10). Writing to the file will clear all previously reserved ports and update the current list with the one given in the input.

Note that ip_local_port_range and ip_local_reserved_ports settings are independent and both are considered by the kernel when determining which ports are available for automatic port assignments.

You can reserve ports which are not in the current ip_local_port_range, e.g.:

```
$ cat /proc/sys/net/ipv4/ip_local_port_range
32000      60999
$ cat /proc/sys/net/ipv4/ip_local_reserved_ports
8080,9148
```

although this is redundant. However such a setting is useful if later the port range is changed to a value that will include the reserved ports. Also keep in mind, that overlapping of these ranges may affect probability of selecting ephemeral ports which are right after block of reserved ports.

Default: Empty

ip_unprivileged_port_start - INTEGER

This is a per-namespace sysctl. It defines the first unprivileged port in the network namespace. Privileged ports require root or CAP_NET_BIND_SERVICE in order to bind to them. To disable all privileged ports, set this to 0. They must not overlap with the ip_local_port_range.

Default: 1024

ip_nonlocal_bind - BOOLEAN

If set, allows processes to bind() to non-local IP addresses, which can be quite useful - but may break some applications.

Default: 0

ip_autobind_reuse - BOOLEAN

By default, bind() does not select the ports automatically even if the new socket and all sockets bound to the port have SO_REUSEADDR. ip_autobind_reuse allows bind() to reuse the port and this is useful when you use bind() + connect(), but may break some applications. The preferred solution is to use IP_BIND_ADDRESS_NO_PORT and this option should only be set by experts. Default: 0

ip_dynaddr - INTEGER

If set non-zero, enables support for dynamic addresses. If set to a non-zero value larger than 1, a kernel log message will be printed when dynamic address rewriting occurs.

Default: 0

ip_early_demux - BOOLEAN

Optimize input packet processing down to one demux for certain kinds of local sockets. Currently we only do this for established TCP and connected UDP sockets.

It may add an additional cost for pure routing workloads that reduces overall throughput, in such case you should disable it.

Default: 1

ping_group_range - 2 INTEGERS

Restrict ICMP_PROTO datagram sockets to users in the group range. The default is "1 0", meaning, that nobody (not even root) may create ping sockets. Setting it to "100 100" would grant permissions to the single group. "0 4294967294" would enable it for the world, "100 4294967294" would enable it for the users, but not daemons.

tcp_early_demux - BOOLEAN

Enable early demux for established TCP sockets.

Default: 1

udp_early_demux - BOOLEAN

Enable early demux for connected UDP sockets. Disable this if your system could experience more unconnected load.

Default: 1

icmp_echo_ignore_all - BOOLEAN

If set non-zero, then the kernel will ignore all ICMP ECHO requests sent to it.

Default: 0

icmp_echo_enable_probe - BOOLEAN

If set to one, then the kernel will respond to RFC 8335 PROBE requests sent to it.

Default: 0

icmp_echo_ignore_broadcasts - BOOLEAN

If set non-zero, then the kernel will ignore all ICMP ECHO and TIMESTAMP requests sent to it via broadcast/multicast.

Default: 1

icmp_ratelimit - INTEGER

Limit the maximal rates for sending ICMP packets whose type matches icmp_ratemask (see below) to specific targets. 0 to disable any limiting, otherwise the minimal space between responses in milliseconds. Note that another sysctl, icmp_msgs_per_sec limits the number of ICMP packets sent on all targets.

Default: 1000

icmp_msgs_per_sec - INTEGER

Limit maximal number of ICMP packets sent per second from this host. Only messages whose type matches icmp_ratemask (see below) are controlled by this limit. For security reasons, the precise count of messages per second is randomized.

Default: 1000

icmp_msgs_burst - INTEGER

icmp_msgs_per_sec controls number of ICMP packets sent per second, while icmp_msgs_burst controls the burst size of these packets. For security reasons, the precise burst size is randomized.

Default: 50

icmp_ratemask - INTEGER

Mask made of ICMP types for which rates are being limited.

Significant bits: IHGFEDCBA9876543210

Default mask: 0000001100000011000 (6168)

Bit definitions (see include/linux/icmp.h):

0	Echo Reply
3	Destination Unreachable ¹
4	Source Quench ¹
5	Redirect
8	Echo Request
B	Time Exceeded ¹
C	Parameter Problem ¹
D	Timestamp Request
E	Timestamp Reply
F	Info Request
G	Info Reply
H	Address Mask Request
I	Address Mask Reply

icmp_ignore_bogus_error_responses - BOOLEAN

Some routers violate RFC1122 by sending bogus responses to broadcast frames. Such violations are normally logged via a kernel warning. If this is set to TRUE, the kernel will not give such warnings, which will avoid log file clutter.

Default: 1

icmp_errors_use_inbound_ifaddr - BOOLEAN

¹ These are rate limited by default (see default mask above)

If zero, icmp error messages are sent with the primary address of the exiting interface.

If non-zero, the message will be sent with the primary address of the interface that received the packet that caused the icmp error. This is the behaviour many network administrators will expect from a router. And it can make debugging complicated network layouts much easier.

Note that if no primary address exists for the interface selected, then the primary address of the first non-loopback interface that has one will be used regardless of this setting.

Default: 0

igmp_max_memberships - INTEGER

Change the maximum number of multicast groups we can subscribe to. Default: 20

Theoretical maximum value is bounded by having to send a membership report in a single datagram (i.e. the report can't span multiple datagrams, or risk confusing the switch and leaving groups you don't intend to).

The number of supported groups 'M' is bounded by the number of group report entries you can fit into a single datagram of 65535 bytes.

$$M = 65536 - \text{sizeof(ip header)} / (\text{sizeof(Group record)})$$

Group records are variable length, with a minimum of 12 bytes. So net.ipv4.igmp_max_memberships should not be set higher than:

$$(65536 - 24) / 12 = 5459$$

The value 5459 assumes no IP header options, so in practice this number may be lower.

igmp_max_msf - INTEGER

Maximum number of addresses allowed in the source filter list for a multicast group.

Default: 10

igmp_qrv - INTEGER

Controls the IGMP query robustness variable (see RFC2236 8.1).

Default: 2 (as specified by RFC2236 8.1)

Minimum: 1 (as specified by RFC6636 4.5)

force_igmp_version - INTEGER

- 0 - (default) No enforcement of a IGMP version, IGMPv1/v2 fallback allowed. Will back to IGMPv3 mode again if all IGMPv1/v2 Querier Present timer expires.
- 1 - Enforce to use IGMP version 1. Will also reply IGMPv1 report if receive IGMPv2/v3 query.
- 2 - Enforce to use IGMP version 2. Will fallback to IGMPv1 if receive IGMPv1 query message. Will reply report if receive IGMPv3 query.
- 3 - Enforce to use IGMP version 3. The same react with default 0.

Note: this is not the same with force_mld_version because IGMPv3 RFC3376 Security Considerations does not have clear description that we could ignore other version messages completely as MLDv2 RFC3810. So make this value as default 0 is recommended.

conf/interface/*

changes special settings per interface (where interface" is the name of your network interface)

conf/all/*

is special, changes the settings for all interfaces

log_martians - BOOLEAN

Log packets with impossible addresses to kernel log. log_martians for the interface will be enabled if at least one of conf/{all,interface}/log_martians is set to TRUE, it will be disabled otherwise

accept_redirects - BOOLEAN

Accept ICMP redirect messages. accept_redirects for the interface will be enabled if:

- both conf/{all,interface}/accept_redirects are TRUE in the case forwarding for the interface is enabled

or

- at least one of conf/{all,interface}/accept_redirects is TRUE in the case forwarding for the interface is disabled

accept_redirects for the interface will be disabled otherwise

default:

- TRUE (host)
- FALSE (router)

forwarding - BOOLEAN

Enable IP forwarding on this interface. This controls whether packets received _on_ this interface can be forwarded.

mc_forwarding - BOOLEAN

Do multicast routing. The kernel needs to be compiled with CONFIG_MROUTE and a multicast routing daemon is required. conf/all/mc_forwarding must also be set to TRUE to enable multicast routing for the interface

medium_id - INTEGER

Integer value used to differentiate the devices by the medium they are attached to. Two devices can have different id values when the broadcast packets are received only on one of them. The default value 0 means that the device is the only interface to its medium, value of -1 means that medium is not known.

Currently, it is used to change the proxy_arp behavior: the proxy_arp feature is enabled for packets forwarded between two devices attached to different media.

proxy_arp - BOOLEAN

Do proxy arp.

proxy_arp for the interface will be enabled if at least one of conf/{all,interface}/proxy_arp is set to TRUE, it will be disabled otherwise

proxy_arp_pvlan - BOOLEAN

Private VLAN proxy arp.

Basically allow proxy arp replies back to the same interface (from which the ARP request/solicitation was received).

This is done to support (ethernet) switch features, like RFC 3069, where the individual ports are NOT allowed to communicate with each other, but they are allowed to talk to the upstream router. As described in RFC 3069, it is possible to allow these hosts to communicate through the upstream router by proxy_arp'ing. Don't need to be used together with proxy_arp.

This technology is known by different names:

In RFC 3069 it is called VLAN Aggregation. Cisco and Allied Telesyn call it Private VLAN. Hewlett-Packard call it Source-Port filtering or port-isolation. Ericsson call it MAC-Forced Forwarding (RFC Draft).

proxy_delay - INTEGER

Delay proxy response.

Delay response to a neighbor solicitation when proxy_arp or proxy_ndp is enabled. A random value between [0, proxy_delay) will be chosen, setting to zero means reply with no delay. Value in jiffies. Defaults to 80.

shared_media - BOOLEAN

Send(router) or accept(host) RFC1620 shared media redirects. Overrides secure_redirects.

shared_media for the interface will be enabled if at least one of conf/{all,interface}/shared_media is set to TRUE, it will be disabled otherwise

default TRUE

secure_redirects - BOOLEAN

Accept ICMP redirect messages only to gateways listed in the interface's current gateway list. Even if disabled, RFC1122 redirect rules still apply.

Overridden by shared_media.

secure_redirects for the interface will be enabled if at least one of conf/{all,interface}/secure_redirects is set to TRUE, it will be disabled otherwise

default TRUE

send_redirects - BOOLEAN

Send redirects, if router.

send_redirects for the interface will be enabled if at least one of conf/{all,interface}/send_redirects is set to TRUE, it will be disabled otherwise

Default: TRUE

bootp_relay - BOOLEAN

Accept packets with source address 0.b.c.d destined not to this host as local ones. It is supposed, that BOOTP relay daemon will catch and forward such packets. conf/all/bootp_relay must also be set to TRUE to enable BOOTP relay for the interface

default FALSE

Not Implemented Yet.

accept_source_route - BOOLEAN

Accept packets with SRR option. conf/all/accept_source_route must also be set to TRUE to accept packets with SRR option on the interface

default

- TRUE (router)
- FALSE (host)

accept_local - BOOLEAN

Accept packets with local source addresses. In combination with suitable routing, this can be used to direct packets between two local interfaces over the wire and have them accepted properly. default FALSE

route_localnet - BOOLEAN

Do not consider loopback addresses as martian source or destination while routing. This enables the use of 127/8 for local routing purposes.

default FALSE

rp_filter - INTEGER

- 0 - No source validation.
- 1 - Strict mode as defined in RFC3704 Strict Reverse Path Each incoming packet is tested against the FIB and if the interface is not the best reverse path the packet check will fail. By default failed packets are discarded.
- 2 - Loose mode as defined in RFC3704 Loose Reverse Path Each incoming packet's source address is also tested against the FIB and if the source address is not reachable via any interface the packet check will fail.

Current recommended practice in RFC3704 is to enable strict mode to prevent IP spoofing from DDos attacks. If using asymmetric routing or other complicated routing, then loose mode is recommended.

The max value from conf/{all,interface}/rp_filter is used when doing source validation on the {interface}.

Default value is 0. Note that some distributions enable it in startup scripts.

src_valid_mark - BOOLEAN

- 0 - The fwmark of the packet is not included in reverse path route lookup. This allows for asymmetric routing configurations utilizing the fwmark in only one direction, e.g., transparent proxying.
- 1 - The fwmark of the packet is included in reverse path route lookup. This permits rp_filter to function when the fwmark is used for routing traffic in both directions.

This setting also affects the utilization of fmwark when performing source address selection for ICMP replies, or determining addresses stored for the IPOPT_TS_TSANDADDR and IPOPT_RR IP options.

The max value from conf/{all,interface}/src_valid_mark is used.

Default value is 0.

arp_filter - BOOLEAN

- 1 - Allows you to have multiple network interfaces on the same subnet, and have the ARPs for each interface be answered based on whether or not the kernel would route a packet from the ARP'd IP out that interface (therefore you must use source based routing for this to work). In other words it allows control of which cards (usually 1) will respond to an arp request.

- 0 - (default) The kernel can respond to arp requests with addresses from other interfaces. This may seem wrong but it usually makes sense, because it increases the chance of successful communication. IP addresses are owned by the complete host on Linux, not by particular interfaces. Only for more complex setups like load-balancing, does this behaviour cause problems.

arp_filter for the interface will be enabled if at least one of conf/{all,interface}/arp_filter is set to TRUE, it will be disabled otherwise

arp_announce - INTEGER

Define different restriction levels for announcing the local source IP address from IP packets in ARP requests sent on interface:

- 0 - (default) Use any local address, configured on any interface
- 1 - Try to avoid local addresses that are not in the target's subnet for this interface. This mode is useful when target hosts reachable via this interface require the source IP address in ARP requests to be part of their logical network configured on the receiving interface. When we generate the request we will check all our subnets that include the target IP and will preserve the source address if it is from such subnet. If there is no such subnet we select source address according to the rules for level 2.
- 2 - Always use the best local address for this target. In this mode we ignore the source address in the IP packet and try to select local address that we prefer for talks with the target host. Such local address is selected by looking for primary IP addresses on all our subnets on the outgoing interface that include the target IP address. If no suitable local address is found we select the first local address we have on the outgoing interface or on all other interfaces, with the hope we will receive reply for our request and even sometimes no matter the source IP address we announce.

The max value from conf/{all,interface}/arp_announce is used.

Increasing the restriction level gives more chance for receiving answer from the resolved target while decreasing the level announces more valid sender's information.

arp_ignore - INTEGER

Define different modes for sending replies in response to received ARP requests that resolve local target IP addresses:

- 0 - (default): reply for any local target IP address, configured on any interface
- 1 - reply only if the target IP address is local address configured on the incoming interface
- 2 - reply only if the target IP address is local address configured on the incoming interface and both with the sender's IP address are part from same subnet on this interface
- 3 - do not reply for local addresses configured with scope host, only resolutions for global and link addresses are replied
- 4-7 - reserved
- 8 - do not reply for all local addresses

The max value from conf/{all,interface}/arp_ignore is used when ARP request is received on the {interface}

arp_notify - BOOLEAN

Define mode for notification of address and device changes.

0	(default): do nothing
1	Generate gratuitous arp requests when device is brought up or hardware address changes.

arp_accept - INTEGER

Define behavior for accepting gratuitous ARP (garp) frames from devices that are not already present in the ARP table:

- 0 - don't create new entries in the ARP table
- 1 - create new entries in the ARP table
- 2 - create new entries only if the source IP address is in the same subnet as an address configured on the interface that received the garp message.

Both replies and requests type gratuitous arp will trigger the ARP table to be updated, if this setting is on.

If the ARP table already contains the IP address of the gratuitous arp frame, the arp table will be updated regardless if this setting is on or off.

arp_evict_nocARRIER - BOOLEAN

Clears the ARP cache on NOCARRIER events. This option is important for wireless devices where the ARP cache should not be cleared when roaming between access points on the same network. In most cases this should remain as the default (1).

- 1 - (default): Clear the ARP cache on NOCARRIER events
- 0 - Do not clear ARP cache on NOCARRIER events

mcast_solicit - INTEGER

The maximum number of multicast probes in INCOMPLETE state, when the associated hardware address is unknown. Defaults to 3.

ucast_solicit - INTEGER

The maximum number of unicast probes in PROBE state, when the hardware address is being reconfirmed. Defaults to 3.

app_solicit - INTEGER

The maximum number of probes to send to the user space ARP daemon via netlink before dropping back to multicast probes (see mcast_resolicit). Defaults to 0.

mcast_resolicit - INTEGER

The maximum number of multicast probes after unicast and app probes in PROBE state. Defaults to 0.

disable_policy - BOOLEAN

Disable IPSEC policy (SPD) for this interface

disable_xfrm - BOOLEAN

Disable IPSEC encryption on this interface, whatever the policy

igmpv2_unsolicited_report_interval - INTEGER

The interval in milliseconds in which the next unsolicited IGMPv1 or IGMPv2 report retransmit will take place.

Default: 10000 (10 seconds)

igmpv3_unsolicited_report_interval - INTEGER

The interval in milliseconds in which the next unsolicited IGMPv3 report retransmit will take place.

Default: 1000 (1 seconds)

ignore_routes_with_linkdown - BOOLEAN

Ignore routes whose link is down when performing a FIB lookup.

promote_secondaries - BOOLEAN

When a primary IP address is removed from this interface promote a corresponding secondary IP address instead of removing all the corresponding secondary IP addresses.

drop_unicast_in_l2_multicast - BOOLEAN

Drop any unicast IP packets that are received in link-layer multicast (or broadcast) frames.

This behavior (for multicast) is actually a SHOULD in RFC 1122, but is disabled by default for compatibility reasons.

Default: off (0)

drop_gratuitous_arp - BOOLEAN

Drop all gratuitous ARP frames, for example if there's a known good ARP proxy on the network and such frames need not be used (or in the case of 802.11, must not be used to prevent attacks.)

Default: off (0)

tag - INTEGER

Allows you to write a number, which can be used as required.

Default value is 0.

xfrm4_gc_thresh - INTEGER

(Obsolete since linux-4.14) The threshold at which we will start garbage collecting for IPv4 destination cache entries. At twice this value the system will refuse new allocations.

igmp_link_local_mcast_reports - BOOLEAN

Enable IGMP reports for link local multicast groups in the 224.0.0.X range.

Default TRUE

Alexey Kuznetsov. kuznet@ms2.inr.ac.ru

Updated by:

- Andi Kleen ak@muc.de
- Nicolas Delon delon.nicolas@wanadoo.fr

54.8 /proc/sys/net/ipv6/* Variables

IPv6 has no global variables such as `tcp_*`. `tcp_*` settings under `ipv4/` also apply to IPv6 [XXX?].

bindv6only - BOOLEAN

Default value for `IPV6_V6ONLY` socket option, which restricts use of the IPv6 socket to IPv6 communication only.

- TRUE: disable IPv4-mapped address feature
- FALSE: enable IPv4-mapped address feature

Default: FALSE (as specified in RFC3493)

flowlabel_consistency - BOOLEAN

Protect the consistency (and unicity) of flow label. You have to disable it to use `IPV6_FL_F_REFLECT` flag on the flow label manager.

- TRUE: enabled
- FALSE: disabled

Default: TRUE

auto_flowlabels - INTEGER

Automatically generate flow labels based on a flow hash of the packet. This allows intermediate devices, such as routers, to identify packet flows for mechanisms like Equal Cost Multipath Routing (see RFC 6438).

0	automatic flow labels are completely disabled
1	automatic flow labels are enabled by default, they can be disabled on a per socket basis using the <code>IPV6_AUTOFLOWLABEL</code> socket option
2	automatic flow labels are allowed, they may be enabled on a per socket basis using the <code>IPV6_AUTOFLOWLABEL</code> socket option
3	automatic flow labels are enabled and enforced, they cannot be disabled by the socket option

Default: 1

flowlabel_state_ranges - BOOLEAN

Split the flow label number space into two ranges. 0-0x7FFF is reserved for the IPv6 flow manager facility, 0x80000-0xFFFF is reserved for stateless flow labels as described in RFC6437.

- TRUE: enabled
- FALSE: disabled

Default: true

flowlabel_reflect - INTEGER

Control flow label reflection. Needed for Path MTU Discovery to work with Equal Cost Multipath Routing in anycast environments. See RFC 7690 and: <https://tools.ietf.org/html/draft-wang-6man-flow-label-reflection-01>

This is a bitmask.

- 1: enabled for established flows

Note that this prevents automatic flowlabel changes, as done in "tcp: change IPv6 flow-label upon receiving spurious retransmission" and "tcp: Change txhash on every SYN and RTO retransmit"

- 2: enabled for TCP RESET packets (no active listener) If set, a RST packet sent in response to a SYN packet on a closed port will reflect the incoming flow label.
- 4: enabled for ICMPv6 echo reply messages.

Default: 0

fib_multipath_hash_policy - INTEGER

Controls which hash policy to use for multipath routes.

Default: 0 (Layer 3)

Possible values:

- 0 - Layer 3 (source and destination addresses plus flow label)
- 1 - Layer 4 (standard 5-tuple)
- 2 - Layer 3 or inner Layer 3 if present
- 3 - Custom multipath hash. Fields used for multipath hash calculation are determined by fib_multipath_hash_fields sysctl

fib_multipath_hash_fields - UNSIGNED INTEGER

When fib_multipath_hash_policy is set to 3 (custom multipath hash), the fields used for multipath hash calculation are determined by this sysctl.

This value is a bitmask which enables various fields for multipath hash calculation.

Possible fields are:

0x0001	Source IP address
0x0002	Destination IP address
0x0004	IP protocol
0x0008	Flow Label
0x0010	Source port
0x0020	Destination port
0x0040	Inner source IP address
0x0080	Inner destination IP address
0x0100	Inner IP protocol
0x0200	Inner Flow Label
0x0400	Inner source port
0x0800	Inner destination port

Default: 0x0007 (source IP, destination IP and IP protocol)

anycast_src_echo_reply - BOOLEAN

Controls the use of anycast addresses as source addresses for ICMPv6 echo reply

- TRUE: enabled
- FALSE: disabled

Default: FALSE

idgen_delay - INTEGER

Controls the delay in seconds after which time to retry privacy stable address generation if a DAD conflict is detected.

Default: 1 (as specified in RFC7217)

idgen_retries - INTEGER

Controls the number of retries to generate a stable privacy address if a DAD conflict is detected.

Default: 3 (as specified in RFC7217)

mld_qrv - INTEGER

Controls the MLD query robustness variable (see RFC3810 9.1).

Default: 2 (as specified by RFC3810 9.1)

Minimum: 1 (as specified by RFC6636 4.5)

max_dst_opts_number - INTEGER

Maximum number of non-padding TLVs allowed in a Destination options extension header. If this value is less than zero then unknown options are disallowed and the number of known TLVs allowed is the absolute value of this number.

Default: 8

max_hbh_opts_number - INTEGER

Maximum number of non-padding TLVs allowed in a Hop-by-Hop options extension header. If this value is less than zero then unknown options are disallowed and the number of known TLVs allowed is the absolute value of this number.

Default: 8

max_dst_opts_length - INTEGER

Maximum length allowed for a Destination options extension header.

Default: INT_MAX (unlimited)

max_hbh_length - INTEGER

Maximum length allowed for a Hop-by-Hop options extension header.

Default: INT_MAX (unlimited)

skip_notify_on_dev_down - BOOLEAN

Controls whether an RTM_DELROUTE message is generated for routes removed when a device is taken down or deleted. IPv4 does not generate this message; IPv6 does by default. Setting this sysctl to true skips the message, making IPv4 and IPv6 on par in relying on userspace caches to track link events and evict routes.

Default: false (generate message)

nexthop_compat_mode - BOOLEAN

New nexthop API provides a means for managing nexthops independent of prefixes. Backwards compatibility with old route format is enabled by default which means route dumps and notifications contain the new nexthop attribute but also the full, expanded nexthop definition. Further, updates or deletes of a nexthop configuration generate route notifications for each fib entry using the nexthop. Once a system understands the new API, this

sysctl can be disabled to achieve full performance benefits of the new API by disabling the nexthop expansion and extraneous notifications. Default: true (backward compat mode)

fib_notify_on_flag_change - INTEGER

Whether to emit RTM_NEWRROUTE notifications whenever RTM_F_OFFLOAD/RTM_F_TRAP/RTM_F_OFFLOAD_FAILED flags are changed.

After installing a route to the kernel, user space receives an acknowledgment, which means the route was installed in the kernel, but not necessarily in hardware. It is also possible for a route already installed in hardware to change its action and therefore its flags. For example, a host route that is trapping packets can be "promoted" to perform decapsulation following the installation of an IPinIP/VXLAN tunnel. The notifications will indicate to user-space the state of the route.

Default: 0 (Do not emit notifications.)

Possible values:

- 0 - Do not emit notifications.
- 1 - Emit notifications.
- 2 - Emit notifications only for RTM_F_OFFLOAD_FAILED flag change.

ioam6_id - INTEGER

Define the IOAM id of this node. Uses only 24 bits out of 32 in total.

Min: 0 Max: 0xFFFFFFF

Default: 0xFFFFFFF

ioam6_id_wide - LONG INTEGER

Define the wide IOAM id of this node. Uses only 56 bits out of 64 in total. Can be different from ioam6_id.

Min: 0 Max: 0xFFFFFFFFFFFFFF

Default: 0xFFFFFFFFFFFFFF

IPv6 Fragmentation:

ip6frag_high_thresh - INTEGER

Maximum memory used to reassemble IPv6 fragments. When ip6frag_high_thresh bytes of memory is allocated for this purpose, the fragment handler will toss packets until ip6frag_low_thresh is reached.

ip6frag_low_thresh - INTEGER

See ip6frag_high_thresh

ip6frag_time - INTEGER

Time in seconds to keep an IPv6 fragment in memory.

conf/default/*:

Change the interface-specific default settings.

These settings would be used during creating new interfaces.

conf/all/*:

Change all the interface-specific settings.

[XXX: Other special features than forwarding?]

conf/all/disable_ipv6 - BOOLEAN

Changing this value is same as changing `conf/default/disable_ipv6` setting and also all per-interface `disable_ipv6` settings to the same value.

Reading this value does not have any particular meaning. It does not say whether IPv6 support is enabled or disabled. Returned value can be 1 also in the case when some interface has `disable_ipv6` set to 0 and has configured IPv6 addresses.

conf/all/forwarding - BOOLEAN

Enable global IPv6 forwarding between all interfaces.

IPv4 and IPv6 work differently here; e.g. netfilter must be used to control which interfaces may forward packets and which not.

This also sets all interfaces' Host/Router setting 'forwarding' to the specified value. See below for details.

This referred to as global forwarding.

proxy_ndp - BOOLEAN

Do proxy ndp.

fwmark_reflect - BOOLEAN

Controls the fwmark of kernel-generated IPv6 reply packets that are not associated with a socket for example, TCP RSTs or ICMPv6 echo replies). If unset, these packets have a fwmark of zero. If set, they have the fwmark of the packet they are replying to.

Default: 0

conf/interface/*:

Change special settings per interface.

The functional behaviour for certain settings is different depending on whether local forwarding is enabled or not.

accept_ra - INTEGER

Accept Router Advertisements; autoconfigure using them.

It also determines whether or not to transmit Router Solicitations. If and only if the functional setting is to accept Router Advertisements, Router Solicitations will be transmitted.

Possible values are:

0	Do not accept Router Advertisements.
1	Accept Router Advertisements if forwarding is disabled.
2	Overrule forwarding behaviour. Accept Router Advertisements even if forwarding is enabled.

Functional default:

- enabled if local forwarding is disabled.
- disabled if local forwarding is enabled.

accept_ra_defrtr - BOOLEAN

Learn default router in Router Advertisement.

Functional default:

- enabled if accept_ra is enabled.
- disabled if accept_ra is disabled.

ra_defrtr_metric - UNSIGNED INTEGER

Route metric for default route learned in Router Advertisement. This value will be assigned as metric for the default route learned via IPv6 Router Advertisement. Takes affect only if accept_ra_defrtr is enabled.

Possible values:

1 to 0xFFFFFFFF

Default: IP6_RT_PRIO_USER i.e. 1024.

accept_ra_from_local - BOOLEAN

Accept RA with source-address that is found on local machine if the RA is otherwise proper and able to be accepted.

Default is to NOT accept these as it may be an un-intended network loop.

Functional default:

- enabled if accept_ra_from_local is enabled on a specific interface.
- disabled if accept_ra_from_local is disabled on a specific interface.

accept_ra_min_hop_limit - INTEGER

Minimum hop limit Information in Router Advertisement.

Hop limit Information in Router Advertisement less than this variable shall be ignored.

Default: 1

accept_ra_min_lft - INTEGER

Minimum acceptable lifetime value in Router Advertisement.

RA sections with a lifetime less than this value shall be ignored. Zero lifetimes stay unaffected.

Default: 0

accept_ra_pinfo - BOOLEAN

Learn Prefix Information in Router Advertisement.

Functional default:

- enabled if accept_ra is enabled.
- disabled if accept_ra is disabled.

ra_honor_pio_life - BOOLEAN

Whether to use RFC4862 Section 5.5.3e to determine the valid lifetime of an address matching a prefix sent in a Router Advertisement Prefix Information Option.

- If enabled, the PIO valid lifetime will always be honored.
- If disabled, RFC4862 section 5.5.3e is used to determine the valid lifetime of the address.

Default: 0 (disabled)

accept_ra_rt_info_min_plen - INTEGER

Minimum prefix length of Route Information in RA.

Route Information w/ prefix smaller than this variable shall be ignored.

Functional default:

- 0 if accept_ra_rtr_pref is enabled.
- -1 if accept_ra_rtr_pref is disabled.

accept_ra_rt_info_max_plen - INTEGER

Maximum prefix length of Route Information in RA.

Route Information w/ prefix larger than this variable shall be ignored.

Functional default:

- 0 if accept_ra_rtr_pref is enabled.
- -1 if accept_ra_rtr_pref is disabled.

accept_ra_rtr_pref - BOOLEAN

Accept Router Preference in RA.

Functional default:

- enabled if accept_ra is enabled.
- disabled if accept_ra is disabled.

accept_ra_mtu - BOOLEAN

Apply the MTU value specified in RA option 5 (RFC4861). If disabled, the MTU specified in the RA will be ignored.

Functional default:

- enabled if accept_ra is enabled.
- disabled if accept_ra is disabled.

accept_redirects - BOOLEAN

Accept Redirects.

Functional default:

- enabled if local forwarding is disabled.
- disabled if local forwarding is enabled.

accept_source_route - INTEGER

Accept source routing (routing extension header).

- ≥ 0 : Accept only routing header type 2.
- < 0 : Do not accept routing header.

Default: 0

autoconf - BOOLEAN

Autoconfigure addresses using Prefix Information in Router Advertisements.

Functional default:

- enabled if accept_ra_pinfo is enabled.
- disabled if accept_ra_pinfo is disabled.

dad_transmits - INTEGER

The amount of Duplicate Address Detection probes to send.

Default: 1

forwarding - INTEGER

Configure interface-specific Host/Router behaviour.

Note: It is recommended to have the same setting on all interfaces; mixed router/host scenarios are rather uncommon.

Possible values are:

- 0 Forwarding disabled
- 1 Forwarding enabled

FALSE (0):

By default, Host behaviour is assumed. This means:

1. IsRouter flag is not set in Neighbour Advertisements.
2. If accept_ra is TRUE (default), transmit Router Solicitations.
3. If accept_ra is TRUE (default), accept Router Advertisements (and do autoconfiguration).
4. If accept_redirects is TRUE (default), accept Redirects.

TRUE (1):

If local forwarding is enabled, Router behaviour is assumed. This means exactly the reverse from the above:

1. IsRouter flag is set in Neighbour Advertisements.
2. Router Solicitations are not sent unless accept_ra is 2.
3. Router Advertisements are ignored unless accept_ra is 2.
4. Redirects are ignored.

Default: 0 (disabled) if global forwarding is disabled (default), otherwise 1 (enabled).

hop_limit - INTEGER

Default Hop Limit to set.

Default: 64

mtu - INTEGER

Default Maximum Transfer Unit

Default: 1280 (IPv6 required minimum)

ip_nonlocal_bind - BOOLEAN

If set, allows processes to bind() to non-local IPv6 addresses, which can be quite useful - but may break some applications.

Default: 0

router_probe_interval - INTEGER

Minimum interval (in seconds) between Router Probing described in RFC4191.

Default: 60

router_solicitation_delay - INTEGER

Number of seconds to wait after interface is brought up before sending Router Solicitations.

Default: 1

router_solicitation_interval - INTEGER

Number of seconds to wait between Router Solicitations.

Default: 4

router_solicitations - INTEGER

Number of Router Solicitations to send until assuming no routers are present.

Default: 3

use_oif_addrs_only - BOOLEAN

When enabled, the candidate source addresses for destinations routed via this interface are restricted to the set of addresses configured on this interface (vis. RFC 6724, section 4).

Default: false

use_tempaddr - INTEGER

Preference for Privacy Extensions (RFC3041).

- <= 0 : disable Privacy Extensions
- == 1 : enable Privacy Extensions, but prefer public addresses over temporary addresses.
- > 1 : enable Privacy Extensions and prefer temporary addresses over public addresses.

Default:

- 0 (for most devices)
- -1 (for point-to-point devices and loopback devices)

temp_valid_lft - INTEGER

valid lifetime (in seconds) for temporary addresses. If less than the minimum required lifetime (typically 5 seconds), temporary addresses will not be created.

Default: 172800 (2 days)

temp_preferred_lft - INTEGER

Preferred lifetime (in seconds) for temporary addresses. If temp_preferred_lft is less than the minimum required lifetime (typically 5 seconds), temporary addresses will not be created. If temp_preferred_lft is greater than temp_valid_lft, the preferred lifetime is temp_valid_lft.

Default: 86400 (1 day)

keep_addr_on_down - INTEGER

Keep all IPv6 addresses on an interface down event. If set static global addresses with no expiration time are not flushed.

- >0 : enabled
- 0 : system default
- <0 : disabled

Default: 0 (addresses are removed)

max_desync_factor - INTEGER

Maximum value for DESYNC_FACTOR, which is a random value that ensures that clients don't synchronize with each other and generate new addresses at exactly the same time. value is in seconds.

Default: 600

regen_max_retry - INTEGER

Number of attempts before give up attempting to generate valid temporary addresses.

Default: 5

max_addresses - INTEGER

Maximum number of autoconfigured addresses per interface. Setting to zero disables the limitation. It is not recommended to set this value too large (or to zero) because it would be an easy way to crash the kernel by allowing too many addresses to be created.

Default: 16

disable_ipv6 - BOOLEAN

Disable IPv6 operation. If accept_dad is set to 2, this value will be dynamically set to TRUE if DAD fails for the link-local address.

Default: FALSE (enable IPv6 operation)

When this value is changed from 1 to 0 (IPv6 is being enabled), it will dynamically create a link-local address on the given interface and start Duplicate Address Detection, if necessary.

When this value is changed from 0 to 1 (IPv6 is being disabled), it will dynamically delete all addresses and routes on the given interface. From now on it will not possible to add addresses/routes to the selected interface.

accept_dad - INTEGER

Whether to accept DAD (Duplicate Address Detection).

0	Disable DAD
1	Enable DAD (default)
2	Enable DAD, and disable IPv6 operation if MAC-based duplicate link-local address has been found.

DAD operation and mode on a given interface will be selected according to the maximum value of conf/{all,interface}/accept_dad.

force_tllao - BOOLEAN

Enable sending the target link-layer address option even when responding to a unicast neighbor solicitation.

Default: FALSE

Quoting from RFC 2461, section 4.4, Target link-layer address:

"The option MUST be included for multicast solicitations in order to avoid infinite Neighbor Solicitation "recursion" when the peer node does not have a cache entry to return a Neighbor Advertisements message. When responding to unicast solicitations, the option can be omitted since the sender of the solicitation has the correct link-layer address; otherwise it would not have been able to send the unicast solicitation in the first place. However, including the link-layer address in this case adds little overhead and eliminates a potential race condition where the sender deletes the cached link-layer address prior to receiving a response to a previous solicitation."

ndisc_notify - BOOLEAN

Define mode for notification of address and device changes.

- 0 - (default): do nothing
- 1 - Generate unsolicited neighbour advertisements when device is brought up or hardware address changes.

ndisc_tclass - INTEGER

The IPv6 Traffic Class to use by default when sending IPv6 Neighbor Discovery (Router Solicitation, Router Advertisement, Neighbor Solicitation, Neighbor Advertisement, Redirect) messages. These 8 bits can be interpreted as 6 high order bits holding the DSCP value and 2 low order bits representing ECN (which you probably want to leave cleared).

- 0 - (default)

ndisc_evict_nocarrier - BOOLEAN

Clears the neighbor discovery table on NOCARRIER events. This option is important for wireless devices where the neighbor discovery cache should not be cleared when roaming between access points on the same network. In most cases this should remain as the default (1).

- 1 - (default): Clear neighbor discover cache on NOCARRIER events.
- 0 - Do not clear neighbor discovery cache on NOCARRIER events.

mldv1_unsolicited_report_interval - INTEGER

The interval in milliseconds in which the next unsolicited MLDv1 report retransmit will take place.

Default: 10000 (10 seconds)

mldv2_unsolicited_report_interval - INTEGER

The interval in milliseconds in which the next unsolicited MLDv2 report retransmit will take place.

Default: 1000 (1 second)

force_mld_version - INTEGER

- 0 - (default) No enforcement of a MLD version, MLDv1 fallback allowed
- 1 - Enforce to use MLD version 1
- 2 - Enforce to use MLD version 2

suppress_frag_ndisc - INTEGER

Control RFC 6980 (Security Implications of IPv6 Fragmentation with IPv6 Neighbor Discovery) behavior:

- 1 - (default) discard fragmented neighbor discovery packets
- 0 - allow fragmented neighbor discovery packets

optimistic_dad - BOOLEAN

Whether to perform Optimistic Duplicate Address Detection (RFC 4429).

- 0: disabled (default)
- 1: enabled

Optimistic Duplicate Address Detection for the interface will be enabled if at least one of conf/{all,interface}/optimistic_dad is set to 1, it will be disabled otherwise.

use_optimistic - BOOLEAN

If enabled, do not classify optimistic addresses as deprecated during source address selection. Preferred addresses will still be chosen before optimistic addresses, subject to other ranking in the source address selection algorithm.

- 0: disabled (default)
- 1: enabled

This will be enabled if at least one of conf/{all,interface}/use_optimistic is set to 1, disabled otherwise.

stable_secret - IPv6 address

This IPv6 address will be used as a secret to generate IPv6 addresses for link-local addresses and autoconfigured ones. All addresses generated after setting this secret will be stable privacy ones by default. This can be changed via the addrgenmode ip-link.conf/default/stable_secret is used as the secret for the namespace, the interface specific ones can overwrite that. Writes to conf/all/stable_secret are refused.

It is recommended to generate this secret during installation of a system and keep it stable after that.

By default the stable secret is unset.

addr_gen_mode - INTEGER

Defines how link-local and autoconf addresses are generated.

0	generate address based on EUI64 (default)
1	do no generate a link-local address, use EUI64 for addresses generated from autoconf
2	generate stable privacy addresses, using the secret from stable_secret (RFC7217)
3	generate stable privacy addresses, using a random secret if unset

drop_unicast_in_l2_multicast - BOOLEAN

Drop any unicast IPv6 packets that are received in link-layer multicast (or broadcast) frames.

By default this is turned off.

drop_unsolicited_na - BOOLEAN

Drop all unsolicited neighbor advertisements, for example if there's a known good NA proxy on the network and such frames need not be used (or in the case of 802.11, must not be used to prevent attacks.)

By default this is turned off.

accept_untracked_na - INTEGER

Define behavior for accepting neighbor advertisements from devices that are absent in the neighbor cache:

- 0 - (default) Do not accept unsolicited and untracked neighbor advertisements.
- 1 - Add a new neighbor cache entry in STALE state for routers on receiving a neighbor advertisement (either solicited or unsolicited) with target link-layer address option specified if no neighbor entry is already present for the advertised IPv6 address. Without this knob, NAs received for untracked addresses (absent in neighbor cache) are silently ignored.

This is as per router-side behavior documented in RFC9131.

This has lower precedence than drop_unsolicited_na.

This will optimize the return path for the initial off-link communication that is initiated by a directly connected host, by ensuring that the first-hop router which turns on this setting doesn't have to buffer the initial return packets to do neighbor-solicitation. The prerequisite is that the host is configured to send unsolicited neighbor advertisements on interface bringup. This setting should be used in conjunction with the ndisc_notify setting on the host to satisfy this prerequisite.

- 2 - Extend option (1) to add a new neighbor cache entry only if the source IP address is in the same subnet as an address configured on the interface that received the neighbor advertisement.

enhanced_dad - BOOLEAN

Include a nonce option in the IPv6 neighbor solicitation messages used for duplicate address detection per RFC7527. A received DAD NS will only signal a duplicate address if the nonce is different. This avoids any false detection of duplicates due to loopback of the NS messages that we send. The nonce option will be sent on an interface unless both of conf/{all,interface}/enhanced_dad are set to FALSE.

Default: TRUE

54.9 icmp/*:

ratelimit - INTEGER

Limit the maximal rates for sending ICMPv6 messages.

0 to disable any limiting, otherwise the minimal space between responses in milliseconds.

Default: 1000

ratemask - list of comma separated ranges

For ICMPv6 message types matching the ranges in the ratemask, limit the sending of the message according to ratelimit parameter.

The format used for both input and output is a comma separated list of ranges (e.g. "0-127,129" for ICMPv6 message type 0 to 127 and 129). Writing to the file will clear all previous ranges of ICMPv6 message types and update the current list with the input.

Refer to: <https://www.iana.org/assignments/icmpv6-parameters/icmpv6-parameters.xhtml> for numerical values of ICMPv6 message types, e.g. echo request is 128 and echo reply is 129.

Default: 0-1,3-127 (rate limit ICMPv6 errors except Packet Too Big)

echo_ignore_all - BOOLEAN

If set non-zero, then the kernel will ignore all ICMP ECHO requests sent to it over the IPv6 protocol.

Default: 0

echo_ignore_multicast - BOOLEAN

If set non-zero, then the kernel will ignore all ICMP ECHO requests sent to it over the IPv6 protocol via multicast.

Default: 0

echo_ignore_anycast - BOOLEAN

If set non-zero, then the kernel will ignore all ICMP ECHO requests sent to it over the IPv6 protocol destined to anycast address.

Default: 0

error_anycast_as_unicast - BOOLEAN

If set to 1, then the kernel will respond with ICMP Errors resulting from requests sent to it over the IPv6 protocol destined to anycast address essentially treating anycast as unicast.

Default: 0

xfrm6_gc_thresh - INTEGER

(Obsolete since linux-4.14) The threshold at which we will start garbage collecting for IPv6 destination cache entries. At twice this value the system will refuse new allocations.

IPv6 Update by: Pekka Savola <pekkas@netcore.fi> YOSHIFUJI Hideaki / USAGI Project
<yoshfiji@linux-ipv6.org>

54.10 /proc/sys/net/bridge/* Variables:

bridge-nf-call-arptables - BOOLEAN

- 1 : pass bridged ARP traffic to arptables' FORWARD chain.
- 0 : disable this.

Default: 1

bridge-nf-call-iptables - BOOLEAN

- 1 : pass bridged IPv4 traffic to iptables' chains.
- 0 : disable this.

Default: 1

bridge-nf-call-ip6tables - BOOLEAN

- 1 : pass bridged IPv6 traffic to ip6tables' chains.
- 0 : disable this.

Default: 1

bridge-nf-filter-vlan-tagged - BOOLEAN

- 1 : pass bridged vlan-tagged ARP/IP/IPv6 traffic to {arp,ip,ip6}tables.
- 0 : disable this.

Default: 0

bridge-nf-filter-pppoe-tagged - BOOLEAN

- 1 : pass bridged pppoe-tagged IP/IPv6 traffic to {ip,ip6}tables.
- 0 : disable this.

Default: 0

bridge-nf-pass-vlan-input-dev - BOOLEAN

- 1: if bridge-nf-filter-vlan-tagged is enabled, try to find a vlan interface on the bridge and set the netfilter input device to the vlan. This allows use of e.g. "iptables -i br0.1" and makes the REDIRECT target work with vlan-on-top-of-bridge interfaces. When no matching vlan interface is found, or this switch is off, the input device is set to the bridge interface.
- 0: disable bridge netfilter vlan interface lookup.

Default: 0

54.11 proc/sys/net/sctp/* Variables:

addip_enable - BOOLEAN

Enable or disable extension of Dynamic Address Reconfiguration (ADD-IP) functionality specified in RFC5061. This extension provides the ability to dynamically add and remove new addresses for the SCTP associations.

1: Enable extension.

0: Disable extension.

Default: 0

pf_enable - INTEGER

Enable or disable pf (pf is short for potentially failed) state. A value of pf_retrans > path_max_retrans also disables pf state. That is, one of both pf_enable and pf_retrans > path_max_retrans can disable pf state. Since pf_retrans and path_max_retrans can be changed by userspace application, sometimes user expects to disable pf state by the value of pf_retrans > path_max_retrans, but occasionally the value of pf_retrans or path_max_retrans is changed by the user application, this pf state is enabled. As such, it is necessary to add this to dynamically enable and disable pf state. See: <https://datatracker.ietf.org/doc/draft-ietf-tsvwg-sctp-failover> for details.

1: Enable pf.

0: Disable pf.

Default: 1

pf_expose - INTEGER

Unset or enable/disable pf (pf is short for potentially failed) state exposure. Applications can control the exposure of the PF path state in the SCTP_PEER_ADDR_CHANGE event and the SCTP_GET_PEER_ADDR_INFO sockopt. When it's unset, no

SCTP_PEER_ADDR_CHANGE event with SCTP_ADDR_PF state will be sent and a SCTP_PF-state transport info can be got via SCTP_GET_PEER_ADDR_INFO sockopt; When it's enabled, a SCTP_PEER_ADDR_CHANGE event will be sent for a transport becoming SCTP_PF state and a SCTP_PF-state transport info can be got via SCTP_GET_PEER_ADDR_INFO sockopt; When it's disabled, no SCTP_PEER_ADDR_CHANGE event will be sent and it returns -EACCES when trying to get a SCTP_PF-state transport info via SCTP_GET_PEER_ADDR_INFO sockopt.

0: Unset pf state exposure, Compatible with old applications.

1: Disable pf state exposure.

2: Enable pf state exposure.

Default: 0

addip_noauth_enable - BOOLEAN

Dynamic Address Reconfiguration (ADD-IP) requires the use of authentication to protect the operations of adding or removing new addresses. This requirement is mandated so that unauthorized hosts would not be able to hijack associations. However, older implementations may not have implemented this requirement while allowing the ADD-IP extension. For reasons of interoperability, we provide this variable to control the enforcement of the authentication requirement.

1	Allow ADD-IP extension to be used without authentication. This should only be set in a closed environment for interoperability with older implementations.
0	Enforce the authentication requirement

Default: 0

auth_enable - BOOLEAN

Enable or disable Authenticated Chunks extension. This extension provides the ability to send and receive authenticated chunks and is required for secure operation of Dynamic Address Reconfiguration (ADD-IP) extension.

- 1: Enable this extension.
- 0: Disable this extension.

Default: 0

prstcp_enable - BOOLEAN

Enable or disable the Partial Reliability extension (RFC3758) which is used to notify peers that a given DATA should no longer be expected.

- 1: Enable extension
- 0: Disable

Default: 1

max_burst - INTEGER

The limit of the number of new packets that can be initially sent. It controls how bursty the generated traffic can be.

Default: 4

association_max_retrans - INTEGER

Set the maximum number for retransmissions that an association can attempt deciding that the remote end is unreachable. If this value is exceeded, the association is terminated.

Default: 10

max_init_retransmits - INTEGER

The maximum number of retransmissions of INIT and COOKIE-ECHO chunks that an association will attempt before declaring the destination unreachable and terminating.

Default: 8

path_max_retrans - INTEGER

The maximum number of retransmissions that will be attempted on a given path. Once this threshold is exceeded, the path is considered unreachable, and new traffic will use a different path when the association is multihomed.

Default: 5

pf_retrans - INTEGER

The number of retransmissions that will be attempted on a given path before traffic is redirected to an alternate transport (should one exist). Note this is distinct from path_max_retrans, as a path that passes the pf_retrans threshold can still be used. Its only deprioritized when a transmission path is selected by the stack. This setting is primarily used to enable fast failover mechanisms without having to reduce path_max_retrans to a very low value. See: <http://www.ietf.org/id/draft-nishida-tsvwg-sctp-failover-05.txt> for details. Note also that a value of pf_retrans > path_max_retrans disables this feature. Since both pf_retrans and path_max_retrans can be changed by userspace application, a variable pf_enable is used to disable pf state.

Default: 0

ps_retrans - INTEGER

Primary.Switchover.Max.Retrans (PSMR), it's a tunable parameter coming from section-5 "Primary Path Switchover" in rfc7829. The primary path will be changed to another active path when the path error counter on the old primary path exceeds PSMR, so that "the SCTP sender is allowed to continue data transmission on a new working path even when the old primary destination address becomes active again". Note this feature is disabled by initializing 'ps_retrans' per netns as 0xffff by default, and its value can't be less than 'pf_retrans' when changing by sysctl.

Default: 0xffff

rto_initial - INTEGER

The initial round trip timeout value in milliseconds that will be used in calculating round trip times. This is the initial time interval for retransmissions.

Default: 3000

rto_max - INTEGER

The maximum value (in milliseconds) of the round trip timeout. This is the largest time interval that can elapse between retransmissions.

Default: 60000

rto_min - INTEGER

The minimum value (in milliseconds) of the round trip timeout. This is the smallest time interval the can elapse between retransmissions.

Default: 1000

hb_interval - INTEGER

The interval (in milliseconds) between HEARTBEAT chunks. These chunks are sent at the specified interval on idle paths to probe the state of a given path between 2 associations.

Default: 30000

sack_timeout - INTEGER

The amount of time (in milliseconds) that the implementation will wait to send a SACK.

Default: 200

valid_cookie_life - INTEGER

The default lifetime of the SCTP cookie (in milliseconds). The cookie is used during association establishment.

Default: 60000

cookie_preserve_enable - BOOLEAN

Enable or disable the ability to extend the lifetime of the SCTP cookie that is used during the establishment phase of SCTP association

- 1: Enable cookie lifetime extension.
- 0: Disable

Default: 1

cookie_hmac_alg - STRING

Select the hmac algorithm used when generating the cookie value sent by a listening sctp socket to a connecting client in the INIT-ACK chunk. Valid values are:

- md5
- sha1
- none

Ability to assign md5 or sha1 as the selected alg is predicated on the configuration of those algorithms at build time (CONFIG_CRYPTO_MD5 and CONFIG_CRYPTO_SHA1).

Default: Dependent on configuration. MD5 if available, else SHA1 if available, else none.

rcvbuf_policy - INTEGER

Determines if the receive buffer is attributed to the socket or to association. SCTP supports the capability to create multiple associations on a single socket. When using this capability, it is possible that a single stalled association that's buffering a lot of data may block other associations from delivering their data by consuming all of the receive buffer space. To work around this, the rcvbuf_policy could be set to attribute the receiver buffer space to each association instead of the socket. This prevents the described blocking.

- 1: rcvbuf space is per association
- 0: rcvbuf space is per socket

Default: 0

sndbuf_policy - INTEGER

Similar to rcvbuf_policy above, this applies to send buffer space.

- 1: Send buffer is tracked per association

- 0: Send buffer is tracked per socket.

Default: 0

sctp_mem - vector of 3 INTEGERs: min, pressure, max

Number of pages allowed for queueing by all SCTP sockets.

min: Below this number of pages SCTP is not bothered about its memory appetite. When amount of memory allocated by SCTP exceeds this number, SCTP starts to moderate memory usage.

pressure: This value was introduced to follow format of tcp_mem.

max: Number of pages allowed for queueing by all SCTP sockets.

Default is calculated at boot time from amount of available memory.

sctp_rmem - vector of 3 INTEGERs: min, default, max

Only the first value ("min") is used, "default" and "max" are ignored.

min: Minimal size of receive buffer used by SCTP socket. It is guaranteed to each SCTP socket (but not association) even under moderate memory pressure.

Default: 4K

sctp_wmem - vector of 3 INTEGERs: min, default, max

Only the first value ("min") is used, "default" and "max" are ignored.

min: Minimum size of send buffer that can be used by SCTP sockets. It is guaranteed to each SCTP socket (but not association) even under moderate memory pressure.

Default: 4K

addr_scope_policy - INTEGER

Control IPv4 address scoping - draft-stewart-tsvwg-sctp-ipv4-00

- 0 - Disable IPv4 address scoping
- 1 - Enable IPv4 address scoping
- 2 - Follow draft but allow IPv4 private addresses
- 3 - Follow draft but allow IPv4 link local addresses

Default: 1

udp_port - INTEGER

The listening port for the local UDP tunneling sock. Normally it's using the IANA-assigned UDP port number 9899 (sctp-tunneling).

This UDP sock is used for processing the incoming UDP-encapsulated SCTP packets (from RFC6951), and shared by all applications in the same net namespace. This UDP sock will be closed when the value is set to 0.

The value will also be used to set the src port of the UDP header for the outgoing UDP-encapsulated SCTP packets. For the dest port, please refer to 'encap_port' below.

Default: 0

encap_port - INTEGER

The default remote UDP encapsulation port.

This value is used to set the dest port of the UDP header for the outgoing UDP-encapsulated SCTP packets by default. Users can also change the value for each sock/asoc/transport by using setsockopt. For further information, please refer to RFC6951.

Note that when connecting to a remote server, the client should set this to the port that the UDP tunneling sock on the peer server is listening to and the local UDP tunneling sock on the client also must be started. On the server, it would get the encap_port from the incoming packet's source port.

Default: 0

plpmtud_probe_interval - INTEGER

The time interval (in milliseconds) for the PLPMTUD probe timer, which is configured to expire after this period to receive an acknowledgment to a probe packet. This is also the time interval between the probes for the current pmtu when the probe search is done.

PLPMTUD will be disabled when 0 is set, and other values for it must be ≥ 5000 .

Default: 0

reconf_enable - BOOLEAN

Enable or disable extension of Stream Reconfiguration functionality specified in RFC6525. This extension provides the ability to "reset" a stream, and it includes the Parameters of "Outgoing/Incoming SSN Reset", "SSN/TSN Reset" and "Add Outgoing/Incoming Streams".

- 1: Enable extension.
- 0: Disable extension.

Default: 0

intl_enable - BOOLEAN

Enable or disable extension of User Message Interleaving functionality specified in RFC8260. This extension allows the interleaving of user messages sent on different streams. With this feature enabled, I-DATA chunk will replace DATA chunk to carry user messages if also supported by the peer. Note that to use this feature, one needs to set this option to 1 and also needs to set socket options SCTP_FRAGMENT_INTERLEAVE to 2 and SCTP_INTERLEAVING_SUPPORTED to 1.

- 1: Enable extension.
- 0: Disable extension.

Default: 0

ecn_enable - BOOLEAN

Control use of Explicit Congestion Notification (ECN) by SCTP. Like in TCP, ECN is used only when both ends of the SCTP connection indicate support for it. This feature is useful in avoiding losses due to congestion by allowing supporting routers to signal congestion before having to drop packets.

1: Enable ecn. 0: Disable ecn.

Default: 1

l3mdev_accept - BOOLEAN

Enabling this option allows a "global" bound socket to work across L3 master domains (e.g., VRFs) with packets capable of being received regardless of the L3 do-

main in which they originated. Only valid when the kernel was compiled with CONFIG_NET_L3_MASTER_DEV.

Default: 1 (enabled)

54.12 /proc/sys/net/core/*

Please see: Documentation/admin-guide/sysctl/net.rst for descriptions of these entries.

54.13 /proc/sys/net/unix/*

max_dgram_qlen - INTEGER

The maximum length of dgram socket receive queue

Default: 10

IPV6

Options for the ipv6 module are supplied as parameters at load time.

Module options may be given as command line arguments to the insmod or modprobe command, but are usually specified in either /etc/modules.d/*.conf configuration files, or in a distro-specific configuration file.

The available ipv6 module parameters are listed below. If a parameter is not specified the default value is used.

The parameters are as follows:

disable

Specifies whether to load the IPv6 module, but disable all its functionality. This might be used when another module has a dependency on the IPv6 module being loaded, but no IPv6 addresses or operations are desired.

The possible values and their effects are:

0

IPv6 is enabled.

This is the default value.

1

IPv6 is disabled.

No IPv6 addresses will be added to interfaces, and it will not be possible to open an IPv6 socket.

A reboot is required to enable IPv6.

autoconf

Specifies whether to enable IPv6 address autoconfiguration on all interfaces. This might be used when one does not wish for addresses to be automatically generated from prefixes received in Router Advertisements.

The possible values and their effects are:

0

IPv6 address autoconfiguration is disabled on all interfaces.

Only the IPv6 loopback address (::1) and link-local addresses will be added to interfaces.

1

IPv6 address autoconfiguration is enabled on all interfaces.

This is the default value.

`disable_ipv6`

Specifies whether to disable IPv6 on all interfaces. This might be used when no IPv6 addresses are desired.

The possible values and their effects are:

0

IPv6 is enabled on all interfaces.

This is the default value.

1

IPv6 is disabled on all interfaces.

No IPv6 addresses will be added to interfaces.

IPVLAN DRIVER HOWTO

Initial Release:

Mahesh Bandewar <maheshb AT google.com>

56.1 1. Introduction:

This is conceptually very similar to the macvlan driver with one major exception of using L3 for mux-ing /demux-ing among slaves. This property makes the master device share the L2 with its slave devices. I have developed this driver in conjunction with network namespaces and not sure if there is use case outside of it.

56.2 2. Building and Installation:

In order to build the driver, please select the config item CONFIG_IPVLAN. The driver can be built into the kernel (CONFIG_IPVLAN=y) or as a module (CONFIG_IPVLAN=m).

56.3 3. Configuration:

There are no module parameters for this driver and it can be configured using IProute2/ip utility.

```
ip link add link <master> name <slave> type ipvlan [ mode MODE ] [ FLAGS ]
where
  MODE: l3 (default) | l3s | l2
  FLAGS: bridge (default) | private | vepa
```

e.g.

- (a) Following will create IPvlan link with eth0 as master in L3 bridge mode:

```
bash# ip link add link eth0 name ipvl0 type ipvlan
```

- (b) This command will create IPvlan link in L2 bridge mode:

```
bash# ip link add link eth0 name ipvl0 type ipvlan mode l2 bridge
```

- (c) This command will create an IPvlan device in L2 private mode:

```
bash# ip link add link eth0 name ipvlan type ipvlan mode l2 private
```

- (d) This command will create an IPvlan device in L2 vepa mode:

```
bash# ip link add link eth0 name ipvlan type ipvlan mode l2 vepa
```

56.4 4. Operating modes:

IPvlan has two modes of operation - L2 and L3. For a given master device, you can select one of these two modes and all slaves on that master will operate in the same (selected) mode. The RX mode is almost identical except that in L3 mode the slaves won't receive any multicast / broadcast traffic. L3 mode is more restrictive since routing is controlled from the other (mostly) default namespace.

56.4.1 4.1 L2 mode:

In this mode TX processing happens on the stack instance attached to the slave device and packets are switched and queued to the master device to send out. In this mode the slaves will RX/TX multicast and broadcast (if applicable) as well.

56.4.2 4.2 L3 mode:

In this mode TX processing up to L3 happens on the stack instance attached to the slave device and packets are switched to the stack instance of the master device for the L2 processing and routing from that instance will be used before packets are queued on the outbound device. In this mode the slaves will not receive nor can send multicast / broadcast traffic.

56.4.3 4.3 L3S mode:

This is very similar to the L3 mode except that iptables (conn-tracking) works in this mode and hence it is L3-symmetric (L3s). This will have slightly less performance but that shouldn't matter since you are choosing this mode over plain-L3 mode to make conn-tracking work.

56.5 5. Mode flags:

At this time following mode flags are available

56.5.1 5.1 bridge:

This is the default option. To configure the IPvlan port in this mode, user can choose to either add this option on the command-line or don't specify anything. This is the traditional mode where slaves can cross-talk among themselves apart from talking through the master device.

56.5.2 5.2 private:

If this option is added to the command-line, the port is set in private mode. i.e. port won't allow cross communication between slaves.

56.5.3 5.3 vepa:

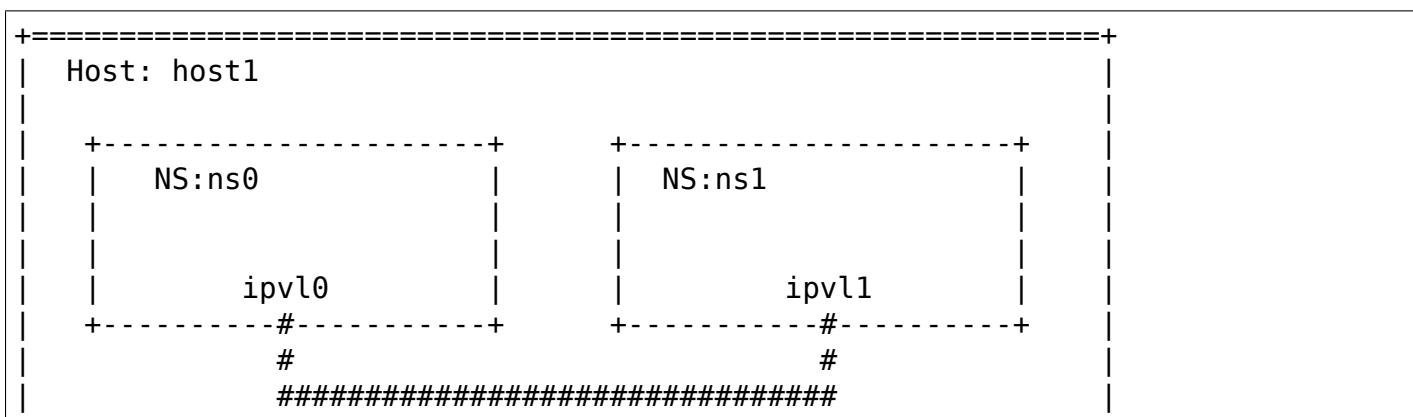
If this is added to the command-line, the port is set in VEPA mode. i.e. port will offload switching functionality to the external entity as described in 802.1Qbg Note: VEPA mode in IPvlan has limitations. IPvlan uses the mac-address of the master-device, so the packets which are emitted in this mode for the adjacent neighbor will have source and destination mac same. This will make the switch / router send the redirect message.

56.6 6. What to choose (macvlan vs. ipvlan)?

These two devices are very similar in many regards and the specific use case could very well define which device to choose. if one of the following situations defines your use case then you can choose to use ipvan:

- (a) The Linux host that is connected to the external switch / router has policy configured that allows only one mac per port.
 - (b) No of virtual devices created on a master exceed the mac capacity and puts the NIC in promiscuous mode and degraded performance is a concern.
 - (c) If the slave device is to be put into the hostile / untrusted network namespace where L2 on the slave could be changed / misused.

56.7 6. Example configuration:



```
| # eth0 |
+=====+=====+
```

- (a) Create two network namespaces - ns0, ns1:

```
ip netns add ns0
ip netns add ns1
```

- (b) Create two ipvlan slaves on eth0 (master device):

```
ip link add link eth0 ipvl0 type ipvlan mode l2
ip link add link eth0 ipvl1 type ipvlan mode l2
```

- (c) Assign slaves to the respective network namespaces:

```
ip link set dev ipvl0 netns ns0
ip link set dev ipvl1 netns ns1
```

- (d) Now switch to the namespace (ns0 or ns1) to configure the slave devices

- For ns0:

```
(1) ip netns exec ns0 bash
(2) ip link set dev ipvl0 up
(3) ip link set dev lo up
(4) ip -4 addr add 127.0.0.1 dev lo
(5) ip -4 addr add $IPADDR dev ipvl0
(6) ip -4 route add default via $ROUTER dev ipvl0
```

- For ns1:

```
(1) ip netns exec ns1 bash
(2) ip link set dev ipvl1 up
(3) ip link set dev lo up
(4) ip -4 addr add 127.0.0.1 dev lo
(5) ip -4 addr add $IPADDR dev ipvl1
(6) ip -4 route add default via $ROUTER dev ipvl1
```

IPVS-SYSCTL

57.1 /proc/sys/net/ipv4/vs/* Variables:

am_droprate - INTEGER

default 10

It sets the always mode drop rate, which is used in the mode 3 of the drop_rate defense.

amemthresh - INTEGER

default 1024

It sets the available memory threshold (in pages), which is used in the automatic modes of defense. When there is no enough available memory, the respective strategy will be enabled and the variable is automatically set to 2, otherwise the strategy is disabled and the variable is set to 1.

backup_only - BOOLEAN

- 0 - disabled (default)
- not 0 - enabled

If set, disable the director function while the server is in backup mode to avoid packet loops for DR/TUN methods.

conn_reuse_mode - INTEGER

1 - default

Controls how ipvs will deal with connections that are detected port reuse. It is a bitmap, with the values being:

0: disable any special handling on port reuse. The new connection will be delivered to the same real server that was servicing the previous connection.

bit 1: enable rescheduling of new connections when it is safe. That is, whenever expire_nodest_conn and for TCP sockets, when the connection is in TIME_WAIT state (which is only possible if you use NAT mode).

bit 2: it is bit 1 plus, for TCP connections, when connections are in FIN_WAIT state, as this is the last state seen by load balancer in Direct Routing mode. This bit helps on adding new real servers to a very busy cluster.

contrack - BOOLEAN

- 0 - disabled (default)
- not 0 - enabled

If set, maintain connection tracking entries for connections handled by IPVS.

This should be enabled if connections handled by IPVS are to be also handled by stateful firewall rules. That is, iptables rules that make use of connection tracking. It is a performance optimisation to disable this setting otherwise.

Connections handled by the IPVS FTP application module will have connection tracking entries regardless of this setting.

Only available when IPVS is compiled with CONFIG_IP_VS_NFCT enabled.

cache_bypass - BOOLEAN

- 0 - disabled (default)
- not 0 - enabled

If it is enabled, forward packets to the original destination directly when no cache server is available and destination address is not local (iph->daddr is RTN_UNICAST). It is mostly used in transparent web cache cluster.

debug_level - INTEGER

- 0 - transmission error messages (default)
- 1 - non-fatal error messages
- 2 - configuration
- 3 - destination trash
- 4 - drop entry
- 5 - service lookup
- 6 - scheduling
- 7 - connection new/expire, lookup and synchronization
- 8 - state transition
- 9 - binding destination, template checks and applications
- 10 - IPVS packet transmission
- 11 - IPVS packet handling (ip_vs_in/ip_vs_out)
- 12 or more - packet traversal

Only available when IPVS is compiled with CONFIG_IP_VS_DEBUG enabled.

Higher debugging levels include the messages for lower debugging levels, so setting debug level 2, includes level 0, 1 and 2 messages. Thus, logging becomes more and more verbose the higher the level.

drop_entry - INTEGER

- 0 - disabled (default)

The drop_entry defense is to randomly drop entries in the connection hash table, just in order to collect back some memory for new connections. In the current code, the drop_entry procedure can be activated every second, then it randomly scans 1/32 of the whole and drops entries that are in the SYN-RECV/SYNACK state, which should be effective against syn-flooding attack.

The valid values of drop_entry are from 0 to 3, where 0 means that this strategy is always disabled, 1 and 2 mean automatic modes (when there is no enough available memory, the strategy is enabled and the variable is automatically set to 2, otherwise the strategy is disabled and the variable is set to 1), and 3 means that the strategy is always enabled.

drop_packet - INTEGER

- 0 - disabled (default)

The drop_packet defense is designed to drop 1/rate packets before forwarding them to real servers. If the rate is 1, then drop all the incoming packets.

The value definition is the same as that of the drop_entry. In the automatic mode, the rate is determined by the follow formula: $\text{rate} = \text{amemthresh} / (\text{amemthresh} - \text{available_memory})$ when available memory is less than the available memory threshold. When the mode 3 is set, the always mode drop rate is controlled by the `/proc/sys/net/ipv4/vs/am_droprate`.

est_cpulist - CPULIST

Allowed CPUs for estimation kthreads

Syntax: standard cpulist format empty list - stop kthread tasks and estimation default - the system's housekeeping CPUs for kthreads

Example: "all": all possible CPUs "0-N": all possible CPUs, N denotes last CPU number "0,1-N:1/2": first and all CPUs with odd number "": empty list

est_nice - INTEGER

default 0 Valid range: -20 (more favorable) .. 19 (less favorable)

Niceness value to use for the estimation kthreads (scheduling priority)

expire_nodest_conn - BOOLEAN

- 0 - disabled (default)
- not 0 - enabled

The default value is 0, the load balancer will silently drop packets when its destination server is not available. It may be useful, when user-space monitoring program deletes the destination server (because of server overload or wrong detection) and add back the server later, and the connections to the server can continue.

If this feature is enabled, the load balancer will expire the connection immediately when a packet arrives and its destination server is not available, then the client program will be notified that the connection is closed. This is equivalent to the feature some people requires to flush connections when its destination is not available.

expire_quiescent_template - BOOLEAN

- 0 - disabled (default)
- not 0 - enabled

When set to a non-zero value, the load balancer will expire persistent templates when the destination server is quiescent. This may be useful, when a user makes a destination server quiescent by setting its weight to 0 and it is desired that subsequent otherwise persistent connections are sent to a different destination server. By default new persistent connections are allowed to quiescent destination servers.

If this feature is enabled, the load balancer will expire the persistence template if it is to be used to schedule a new connection and the destination server is quiescent.

ignore_tunneled - BOOLEAN

- 0 - disabled (default)
- not 0 - enabled

If set, ipvs will set the ipvs_property on all packets which are of unrecognized protocols. This prevents us from routing tunneled protocols like ipip, which is useful to prevent rescheduling packets that have been tunneled to the ipvs host (i.e. to prevent ipvs routing loops when ipvs is also acting as a real server).

nat_icmp_send - BOOLEAN

- 0 - disabled (default)
- not 0 - enabled

It controls sending icmp error messages (ICMP_DEST_UNREACH) for VS/NAT when the load balancer receives packets from real servers but the connection entries don't exist.

pmtu_disc - BOOLEAN

- 0 - disabled
- not 0 - enabled (default)

By default, reject with FRAG_NEEDED all DF packets that exceed the PMTU, irrespective of the forwarding method. For TUN method the flag can be disabled to fragment such packets.

secure_tcp - INTEGER

- 0 - disabled (default)

The secure_tcp defense is to use a more complicated TCP state transition table. For VS/NAT, it also delays entering the TCP ESTABLISHED state until the three way handshake is completed.

The value definition is the same as that of drop_entry and drop_packet.

sync_threshold - vector of 2 INTEGERS: sync_threshold, sync_period

default 3 50

It sets synchronization threshold, which is the minimum number of incoming packets that a connection needs to receive before the connection will be synchronized. A connection will be synchronized, every time the number of its incoming packets modulus sync_period equals the threshold. The range of the threshold is from 0 to sync_period.

When sync_period and sync_refresh_period are 0, send sync only for state changes or only once when pkts matches sync_threshold

sync_refresh_period - UNSIGNED INTEGER

default 0

In seconds, difference in reported connection timer that triggers new sync message. It can be used to avoid sync messages for the specified period (or half of the connection timeout if it is lower) if connection state is not changed since last sync.

This is useful for normal connections with high traffic to reduce sync rate. Additionally, retry sync_retries times with period of sync_refresh_period/8.

sync_retries - INTEGER

default 0

Defines sync retries with period of sync_refresh_period/8. Useful to protect against loss of sync messages. The range of the sync_retries is from 0 to 3.

sync_qlen_max - UNSIGNED LONG

Hard limit for queued sync messages that are not sent yet. It defaults to 1/32 of the memory pages but actually represents number of messages. It will protect us from allocating large parts of memory when the sending rate is lower than the queuing rate.

sync_sock_size - INTEGER

default 0

Configuration of SNDBUF (master) or RCVBUF (slave) socket limit. Default value is 0 (preserve system defaults).

sync_ports - INTEGER

default 1

The number of threads that master and backup servers can use for sync traffic. Every thread will use single UDP port, thread 0 will use the default port 8848 while last thread will use port 8848+sync_ports-1.

snat_reroute - BOOLEAN

- 0 - disabled
- not 0 - enabled (default)

If enabled, recalculate the route of SNATed packets from realservers so that they are routed as if they originate from the director. Otherwise they are routed as if they are forwarded by the director.

If policy routing is in effect then it is possible that the route of a packet originating from a director is routed differently to a packet being forwarded by the director.

If policy routing is not in effect then the recalculated route will always be the same as the original route so it is an optimisation to disable snat_reroute and avoid the recalculation.

sync.persist_mode - INTEGER

default 0

Controls the synchronisation of connections when using persistence

0: All types of connections are synchronised

1: Attempt to reduce the synchronisation traffic depending on the connection type. For persistent services avoid synchronisation for normal connections, do it only for persistence templates. In such case, for TCP and SCTP it may need enabling sloppy_tcp and sloppy_sctp flags on backup servers. For non-persistent services such optimization is not applied, mode 0 is assumed.

sync_version - INTEGER

default 1

The version of the synchronisation protocol used when sending synchronisation messages.

0 selects the original synchronisation protocol (version 0). This should be used when sending synchronisation messages to a legacy system that only understands the original synchronisation protocol.

1 selects the current synchronisation protocol (version 1). This should be used where possible.

Kernels with this sync_version entry are able to receive messages of both version 1 and version 2 of the synchronisation protocol.

run_estimation - BOOLEAN

0 - disabled not 0 - enabled (default)

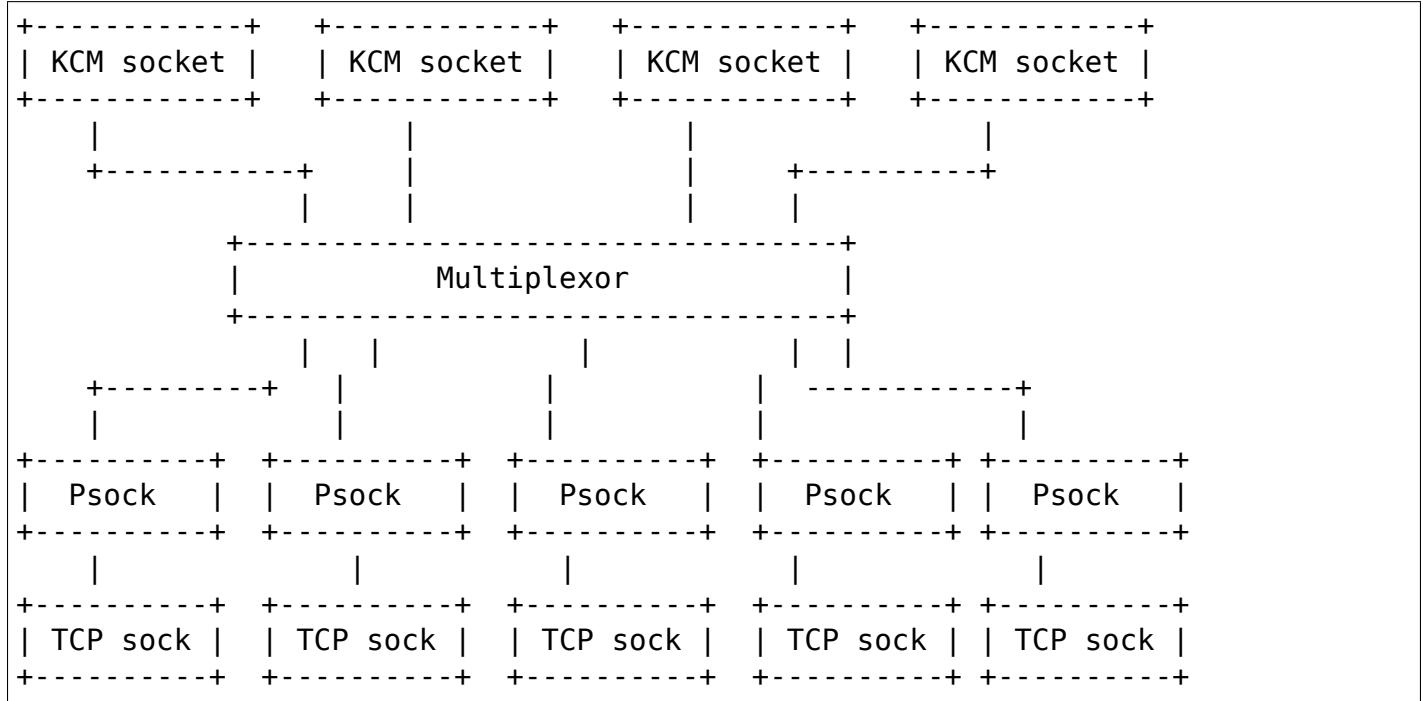
If disabled, the estimation will be suspended and kthread tasks stopped.

You can always re-enable estimation by setting this value to 1. But be careful, the first estimation after re-enable is not accurate.

KERNEL CONNECTION MULTIPLEXOR

Kernel Connection Multiplexor (KCM) is a mechanism that provides a message based interface over TCP for generic application protocols. With KCM an application can efficiently send and receive application protocol messages over TCP using datagram sockets.

KCM implements an NxM multiplexor in the kernel as diagrammed below:



58.1 KCM sockets

The KCM sockets provide the user interface to the multiplexor. All the KCM sockets bound to a multiplexor are considered to have equivalent function, and I/O operations in different sockets may be done in parallel without the need for synchronization between threads in userspace.

58.2 Multiplexor

The multiplexor provides the message steering. In the transmit path, messages written on a KCM socket are sent atomically on an appropriate TCP socket. Similarly, in the receive path, messages are constructed on each TCP socket (Psock) and complete messages are steered to a KCM socket.

58.3 TCP sockets & Psocks

TCP sockets may be bound to a KCM multiplexor. A Psock structure is allocated for each bound TCP socket, this structure holds the state for constructing messages on receive as well as other connection specific information for KCM.

58.4 Connected mode semantics

Each multiplexor assumes that all attached TCP connections are to the same destination and can use the different connections for load balancing when transmitting. The normal send and recv calls (include sendmmsg and recvmsg) can be used to send and receive messages from the KCM socket.

58.5 Socket types

KCM supports SOCK_DGRAM and SOCK_SEQPACKET socket types.

58.5.1 Message delineation

Messages are sent over a TCP stream with some application protocol message format that typically includes a header which frames the messages. The length of a received message can be deduced from the application protocol header (often just a simple length field).

A TCP stream must be parsed to determine message boundaries. Berkeley Packet Filter (BPF) is used for this. When attaching a TCP socket to a multiplexor a BPF program must be specified. The program is called at the start of receiving a new message and is given an skbuff that contains the bytes received so far. It parses the message header and returns the length of the message. Given this information, KCM will construct the message of the stated length and deliver it to a KCM socket.

58.5.2 TCP socket management

When a TCP socket is attached to a KCM multiplexor data ready (POLLIN) and write space available (POLLOUT) events are handled by the multiplexor. If there is a state change (disconnection) or other error on a TCP socket, an error is posted on the TCP socket so that a POLLERR event happens and KCM discontinues using the socket. When the application gets the error notification for a TCP socket, it should unattach the socket from KCM and then handle the error condition (the typical response is to close the socket and create a new connection if necessary).

KCM limits the maximum receive message size to be the size of the receive socket buffer on the attached TCP socket (the socket buffer size can be set by SO_RCVBUF). If the length of a new message reported by the BPF program is greater than this limit a corresponding error (EMSGSIZE) is posted on the TCP socket. The BPF program may also enforce a maximum messages size and report an error when it is exceeded.

A timeout may be set for assembling messages on a receive socket. The timeout value is taken from the receive timeout of the attached TCP socket (this is set by SO_RCVTIMEO). If the timer expires before assembly is complete an error (ETIMEDOUT) is posted on the socket.

58.6 User interface

58.6.1 Creating a multiplexor

A new multiplexor and initial KCM socket is created by a socket call:

```
socket(AF_KCM, type, protocol)
```

- type is either SOCK_DGRAM or SOCK_SEQPACKET
- protocol is KCMPROTO_CONNECTED

58.6.2 Cloning KCM sockets

After the first KCM socket is created using the socket call as described above, additional sockets for the multiplexor can be created by cloning a KCM socket. This is accomplished by an ioctl on a KCM socket:

```
/* From linux/kcm.h */
struct kcm_clone {
    int fd;
};

struct kcm_clone info;

memset(&info, 0, sizeof(info));

err = ioctl(kcmfd, SIOCKCMCLONE, &info);

if (!err)
    newkcmfd = info.fd;
```

58.6.3 Attach transport sockets

Attaching of transport sockets to a multiplexor is performed by calling an ioctl on a KCM socket for the multiplexor. e.g.:

```
/* From linux/kcm.h */
struct kcm_attach {
    int fd;
    int bpf_fd;
};

struct kcm_attach info;

memset(&info, 0, sizeof(info));

info.fd = tcpfd;
info.bpf_fd = bpf_prog_fd;

ioctl(kcmfd, SIOCKCMATTACH, &info);
```

The kcm_attach structure contains:

- fd: file descriptor for TCP socket being attached
- bpf_prog_fd: file descriptor for compiled BPF program downloaded

58.6.4 Unattach transport sockets

Unattaching a transport socket from a multiplexor is straightforward. An "unattach" ioctl is done with the kcm_unattach structure as the argument:

```
/* From linux/kcm.h */
struct kcm_unattach {
    int fd;
};

struct kcm_unattach info;

memset(&info, 0, sizeof(info));

info.fd = cfd;

ioctl(fd, SIOCKCMUNATTACH, &info);
```

58.6.5 Disabling receive on KCM socket

A setsockopt is used to disable or enable receiving on a KCM socket. When receive is disabled, any pending messages in the socket's receive buffer are moved to other sockets. This feature is useful if an application thread knows that it will be doing a lot of work on a request and won't be able to service new messages for a while. Example use:

```
int val = 1;

setsockopt(kcmfd, SOL_KCM, KCM_RECV_DISABLE, &val, sizeof(val))
```

58.6.6 BPF programs for message delineation

BPF programs can be compiled using the BPF LLVM backend. For example, the BPF program for parsing Thrift is:

```
#include "bpf.h" /* for __sk_buff */
#include "bpf_helpers.h" /* for load_word intrinsic */

SEC("socket_kcm")
int bpf_prog1(struct __sk_buff *skb)
{
    return load_word(skb, 0) + 4;
}

char _license[] SEC("license") = "GPL";
```

58.7 Use in applications

KCM accelerates application layer protocols. Specifically, it allows applications to use a message based interface for sending and receiving messages. The kernel provides necessary assurances that messages are sent and received atomically. This relieves much of the burden applications have in mapping a message based protocol onto the TCP stream. KCM also make application layer messages a unit of work in the kernel for the purposes of steering and scheduling, which in turn allows a simpler networking model in multithreaded applications.

58.7.1 Configurations

In an Nx1 configuration, KCM logically provides multiple socket handles to the same TCP connection. This allows parallelism between I/O operations on the TCP socket (for instance copyin and copyout of data is parallelized). In an application, a KCM socket can be opened for each processing thread and inserted into the epoll (similar to how SO_REUSEPORT is used to allow multiple listener sockets on the same port).

In a MxN configuration, multiple connections are established to the same destination. These are used for simple load balancing.

58.7.2 Message batching

The primary purpose of KCM is load balancing between KCM sockets and hence threads in a nominal use case. Perfect load balancing, that is steering each received message to a different KCM socket or steering each sent message to a different TCP socket, can negatively impact performance since this doesn't allow for affinities to be established. Balancing based on groups, or batches of messages, can be beneficial for performance.

On transmit, there are three ways an application can batch (pipeline) messages on a KCM socket.

- 1) Send multiple messages in a single sendmmsg.
- 2) Send a group of messages each with a sendmsg call, where all messages except the last have `MSG_BATCH` in the flags of sendmsg call.
- 3) Create "super message" composed of multiple messages and send this with a single sendmsg.

On receive, the KCM module attempts to queue messages received on the same KCM socket during each TCP ready callback. The targeted KCM socket changes at each receive ready callback on the KCM socket. The application does not need to configure this.

58.7.3 Error handling

An application should include a thread to monitor errors raised on the TCP connection. Normally, this will be done by placing each TCP socket attached to a KCM multiplexor in epoll set for POLLERR event. If an error occurs on an attached TCP socket, KCM sets an EPIPE on the socket thus waking up the application thread. When the application sees the error (which may just be a disconnect) it should unattach the socket from KCM and then close it. It is assumed that once an error is posted on the TCP socket the data stream is unrecoverable (i.e. an error may have occurred in the middle of receiving a message).

58.7.4 TCP connection monitoring

In KCM there is no means to correlate a message to the TCP socket that was used to send or receive the message (except in the case there is only one attached TCP socket). However, the application does retain an open file descriptor to the socket so it will be able to get statistics from the socket which can be used in detecting issues (such as high retransmissions on the socket).

L2TP

Layer 2 Tunneling Protocol (L2TP) allows L2 frames to be tunneled over an IP network.

This document covers the kernel's L2TP subsystem. It documents kernel APIs for application developers who want to use the L2TP subsystem and it provides some technical details about the internal implementation which may be useful to kernel developers and maintainers.

59.1 Overview

The kernel's L2TP subsystem implements the datapath for L2TPv2 and L2TPv3. L2TPv2 is carried over UDP. L2TPv3 is carried over UDP or directly over IP (protocol 115).

The L2TP RFCs define two basic kinds of L2TP packets: control packets (the "control plane"), and data packets (the "data plane"). The kernel deals only with data packets. The more complex control packets are handled by user space.

An L2TP tunnel carries one or more L2TP sessions. Each tunnel is associated with a socket. Each session is associated with a virtual netdevice, e.g. `pppN`, `l2tpethN`, through which data frames pass to/from L2TP. Fields in the L2TP header identify the tunnel or session and whether it is a control or data packet. When tunnels and sessions are set up using the Linux kernel API, we're just setting up the L2TP data path. All aspects of the control protocol are to be handled by user space.

This split in responsibilities leads to a natural sequence of operations when establishing tunnels and sessions. The procedure looks like this:

- 1) Create a tunnel socket. Exchange L2TP control protocol messages with the peer over that socket in order to establish a tunnel.
- 2) Create a tunnel context in the kernel, using information obtained from the peer using the control protocol messages.
- 3) Exchange L2TP control protocol messages with the peer over the tunnel socket in order to establish a session.
- 4) Create a session context in the kernel using information obtained from the peer using the control protocol messages.

59.2 L2TP APIs

This section documents each userspace API of the L2TP subsystem.

59.2.1 Tunnel Sockets

L2TPv2 always uses UDP. L2TPv3 may use UDP or IP encapsulation.

To create a tunnel socket for use by L2TP, the standard POSIX socket API is used.

For example, for a tunnel using IPv4 addresses and UDP encapsulation:

```
int sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

Or for a tunnel using IPv6 addresses and IP encapsulation:

```
int sockfd = socket(AF_INET6, SOCK_DGRAM, IPPROTO_L2TP);
```

UDP socket programming doesn't need to be covered here.

IPPROTO_L2TP is an IP protocol type implemented by the kernel's L2TP subsystem. The L2TPIP socket address is defined in struct sockaddr_l2tpip and struct sockaddr_l2tpip6 at [include/uapi/linux/l2tp.h](#). The address includes the L2TP tunnel (connection) id. To use L2TP IP encapsulation, an L2TPv3 application should bind the L2TPIP socket using the locally assigned tunnel id. When the peer's tunnel id and IP address is known, a connect must be done.

If the L2TP application needs to handle L2TPv3 tunnel setup requests from peers using L2TPIP, it must open a dedicated L2TPIP socket to listen for those requests and bind the socket using tunnel id 0 since tunnel setup requests are addressed to tunnel id 0.

An L2TP tunnel and all of its sessions are automatically closed when its tunnel socket is closed.

59.2.2 Netlink API

L2TP applications use netlink to manage L2TP tunnel and session instances in the kernel. The L2TP netlink API is defined in [include/uapi/linux/l2tp.h](#).

L2TP uses [Generic Netlink](#) (GENL). Several commands are defined: Create, Delete, Modify and Get for tunnel and session instances, e.g. L2TP_CMD_TUNNEL_CREATE. The API header lists the netlink attribute types that can be used with each command.

Tunnel and session instances are identified by a locally unique 32-bit id. L2TP tunnel ids are given by L2TP_ATTR_CONN_ID and L2TP_ATTR_PEER_CONN_ID attributes and L2TP session ids are given by L2TP_ATTR_SESSION_ID and L2TP_ATTR_PEER_SESSION_ID attributes. If netlink is used to manage L2TPv2 tunnel and session instances, the L2TPv2 16-bit tunnel/session id is cast to a 32-bit value in these attributes.

In the L2TP_CMD_TUNNEL_CREATE command, L2TP_ATTR_FD tells the kernel the tunnel socket fd being used. If not specified, the kernel creates a kernel socket for the tunnel, using IP parameters set in L2TP_ATTR_IP[6]_SADDR, L2TP_ATTR_IP[6]_DADDR, L2TP_ATTR_UDP_SPORT, L2TP_ATTR_UDP_DPORT attributes. Kernel sockets are used to implement unmanaged L2TPv3 tunnels (iproute2's "ip l2tp" commands). If L2TP_ATTR_FD is given, it must be a socket fd that is already bound and connected. There is more information about unmanaged tunnels later in this document.

`L2TP_CMD_TUNNEL_CREATE` attributes:-

Attribute	Required	Use
<code>CONN_ID</code>	Y	Sets the tunnel (connection) id.
<code>PEER_CONN_ID</code>	Y	Sets the peer tunnel (connection) id.
<code>PROTO_VERSION</code>	Y	Protocol version. 2 or 3.
<code>ENCAP_TYPE</code>	Y	Encapsulation type: UDP or IP.
<code>FD</code>	N	Tunnel socket file descriptor.
<code>UDP_CSUM</code>	N	Enable IPv4 UDP checksums. Used only if FD is not set.
<code>UDP_ZERO_CSUM6_TX</code>	N	Zero IPv6 UDP checksum on transmit. Used only if FD is not set.
<code>UDP_ZERO_CSUM6_RX</code>	N	Zero IPv6 UDP checksum on receive. Used only if FD is not set.
<code>IP_SADDR</code>	N	IPv4 source address. Used only if FD is not set.
<code>IP_DADDR</code>	N	IPv4 destination address. Used only if FD is not set.
<code>UDP_SPORT</code>	N	UDP source port. Used only if FD is not set.
<code>UDP_DPORT</code>	N	UDP destination port. Used only if FD is not set.
<code>IP6_SADDR</code>	N	IPv6 source address. Used only if FD is not set.
<code>IP6_DADDR</code>	N	IPv6 destination address. Used only if FD is not set.
<code>DEBUG</code>	N	Debug flags.

`L2TP_CMD_TUNNEL_DESTROY` attributes:-

Attribute	Required	Use
<code>CONN_ID</code>	Y	Identifies the tunnel id to be destroyed.

`L2TP_CMD_TUNNEL MODIFY` attributes:-

Attribute	Required	Use
<code>CONN_ID</code>	Y	Identifies the tunnel id to be modified.
<code>DEBUG</code>	N	Debug flags.

`L2TP_CMD_TUNNEL_GET` attributes:-

Attribute	Required	Use
<code>CONN_ID</code>	N	Identifies the tunnel id to be queried. Ignored in DUMP requests.

`L2TP_CMD_SESSION_CREATE` attributes:-

Attribute	Required	Use
CONN_ID	Y	The parent tunnel id.
SESSION_ID	Y	Sets the session id.
PEER_SESSION_ID	Y	Sets the parent session id.
PW_TYPE	Y	Sets the pseudowire type.
DEBUG	N	Debug flags.
RECV_SEQ	N	Enable rx data sequence numbers.
SEND_SEQ	N	Enable tx data sequence numbers.
LNS_MODE	N	Enable LNS mode (auto-enable data sequence numbers).
RECV_TIMEOUT	N	Timeout to wait when reordering received packets.
L2SPEC_TYPE	N	Sets layer2-specific-sublayer type (L2TPv3 only).
COOKIE	N	Sets optional cookie (L2TPv3 only).
PEER_COOKIE	N	Sets optional peer cookie (L2TPv3 only).
IFNAME	N	Sets interface name (L2TPv3 only).

For Ethernet session types, this will create an l2tpeth virtual interface which can then be configured as required. For PPP session types, a PPPoL2TP socket must also be opened and connected, mapping it onto the new session. This is covered in "PPPoL2TP Sockets" later.

L2TP_CMD_SESSION_DESTROY attributes:-

Attribute	Required	Use
CONN_ID	Y	Identifies the parent tunnel id of the session to be destroyed.
SESSION_ID	Y	Identifies the session id to be destroyed.
IFNAME	N	Identifies the session by interface name. If set, this overrides any CONN_ID and SESSION_ID attributes. Currently supported for L2TPv3 Ethernet sessions only.

L2TP_CMD_SESSION MODIFY attributes:-

Attribute	Required	Use
CONN_ID	Y	Identifies the parent tunnel id of the session to be modified.
SESSION_ID	Y	Identifies the session id to be modified.
IFNAME	N	Identifies the session by interface name. If set, this overrides any CONN_ID and SESSION_ID attributes. Currently supported for L2TPv3 Ethernet sessions only.
DEBUG	N	Debug flags.
RECV_SEQ	N	Enable rx data sequence numbers.
SEND_SEQ	N	Enable tx data sequence numbers.
LNS_MODE	N	Enable LNS mode (auto-enable data sequence numbers).
RECV_TIMEOUT	N	Timeout to wait when reordering received packets.

L2TP_CMD_SESSION_GET attributes:-

Attribute	Required	Use
CONN_ID	N	Identifies the tunnel id to be queried. Ignored for DUMP requests.
SESSION_ID	N	Identifies the session id to be queried. Ignored for DUMP requests.
IFNAME	N	Identifies the session by interface name. If set, this overrides any CONN_ID and SESSION_ID attributes. Ignored for DUMP requests. Currently supported for L2TPv3 Ethernet sessions only.

Application developers should refer to [include/uapi/linux/l2tp.h](#) for netlink command and attribute definitions.

Sample userspace code using [libmnl](#):

- Open L2TP netlink socket:

```
struct nl_sock *nl_sock;
int l2tp_nl_family_id;

nl_sock = nl_socket_alloc();
genl_connect(nl_sock);
genl_id = genl_ctrl_resolve(nl_sock, L2TP_GENL_NAME);
```

- Create a tunnel:

```
struct nlmsghdr *nlh;
struct genlmsghdr *gnlh;

nlh = mnl_nlmsg_put_header(buf);
nlh->nlmsg_type = genl_id; /* assigned to genl socket */
nlh->nlmsg_flags = NLM_F_REQUEST | NLM_F_ACK;
nlh->nlmsg_seq = seq;

gnlh = mnl_nlmsg_put_extra_header(nlh, sizeof(*gnlh));
gnlh->cmd = L2TP_CMD_TUNNEL_CREATE;
gnlh->version = L2TP_GENL_VERSION;
gnlh->reserved = 0;

mnl_attr_put_u32(nlh, L2TP_ATTR_FD, tunl_sock_fd);
mnl_attr_put_u32(nlh, L2TP_ATTR_CONN_ID, tid);
mnl_attr_put_u32(nlh, L2TP_ATTR_PEER_CONN_ID, peer_tid);
mnl_attr_put_u8(nlh, L2TP_ATTR_PROTO_VERSION, protocol_version);
mnl_attr_put_u16(nlh, L2TP_ATTR_ENCAP_TYPE, encaps);
```

- Create a session:

```
struct nlmsghdr *nlh;
struct genlmsghdr *gnlh;

nlh = mnl_nlmsg_put_header(buf);
nlh->nlmsg_type = genl_id; /* assigned to genl socket */
nlh->nlmsg_flags = NLM_F_REQUEST | NLM_F_ACK;
nlh->nlmsg_seq = seq;
```

```

gnlh = mnl_nlmsg_put_extra_header(nlh, sizeof(*gnlh));
gnlh->cmd = L2TP_CMD_SESSION_CREATE;
gnlh->version = L2TP_GENL_VERSION;
gnlh->reserved = 0;

mnl_attr_put_u32(nlh, L2TP_ATTR_CONN_ID, tid);
mnl_attr_put_u32(nlh, L2TP_ATTR_PEER_CONN_ID, peer_tid);
mnl_attr_put_u32(nlh, L2TP_ATTR_SESSION_ID, sid);
mnl_attr_put_u32(nlh, L2TP_ATTR_PEER_SESSION_ID, peer_sid);
mnl_attr_put_u16(nlh, L2TP_ATTR_PW_TYPE, pwtype);
/* there are other session options which can be set using netlink
 * attributes during session creation -- see l2tp.h
 */

```

- Delete a session:

```

struct nlmsghdr *nlh;
struct genlmsghdr *gnlh;

nlh = mnl_nlmsg_put_header(buf);
nlh->nlmsg_type = genl_id; /* assigned to genl socket */
nlh->nlmsg_flags = NLM_F_REQUEST | NLM_F_ACK;
nlh->nlmsg_seq = seq;

gnlh = mnl_nlmsg_put_extra_header(nlh, sizeof(*gnlh));
gnlh->cmd = L2TP_CMD_SESSION_DELETE;
gnlh->version = L2TP_GENL_VERSION;
gnlh->reserved = 0;

mnl_attr_put_u32(nlh, L2TP_ATTR_CONN_ID, tid);
mnl_attr_put_u32(nlh, L2TP_ATTR_SESSION_ID, sid);

```

- Delete a tunnel and all of its sessions (if any):

```

struct nlmsghdr *nlh;
struct genlmsghdr *gnlh;

nlh = mnl_nlmsg_put_header(buf);
nlh->nlmsg_type = genl_id; /* assigned to genl socket */
nlh->nlmsg_flags = NLM_F_REQUEST | NLM_F_ACK;
nlh->nlmsg_seq = seq;

gnlh = mnl_nlmsg_put_extra_header(nlh, sizeof(*gnlh));
gnlh->cmd = L2TP_CMD_TUNNEL_DELETE;
gnlh->version = L2TP_GENL_VERSION;
gnlh->reserved = 0;

mnl_attr_put_u32(nlh, L2TP_ATTR_CONN_ID, tid);

```

59.2.3 PPPoL2TP Session Socket API

For PPP session types, a PPPoL2TP socket must be opened and connected to the L2TP session.

When creating PPPoL2TP sockets, the application provides information to the kernel about the tunnel and session in a socket connect() call. Source and destination tunnel and session ids are provided, as well as the file descriptor of a UDP or L2TPIP socket. See struct pppol2tp_addr in [include/linux/if_pppol2tp.h](#). For historical reasons, there are unfortunately slightly different address structures for L2TPv2/L2TPv3 IPv4/IPv6 tunnels and userspace must use the appropriate structure that matches the tunnel socket type.

Userspace may control behavior of the tunnel or session using setsockopt and ioctl on the PPPoX socket. The following socket options are supported:-

DEBUG	bitmask of debug message categories. See below.
SENDSEQ	<ul style="list-style-type: none"> • 0 => don't send packets with sequence numbers • 1 => send packets with sequence numbers
RECVSEQ	<ul style="list-style-type: none"> • 0 => receive packet sequence numbers are optional • 1 => drop receive packets without sequence numbers
LNSMODE	<ul style="list-style-type: none"> • 0 => act as LAC. • 1 => act as LNS.
REORDERTO	reorder timeout (in millisecs). If 0, don't try to reorder.

In addition to the standard PPP ioctls, a PPPIOCGL2TPSTATS is provided to retrieve tunnel and session statistics from the kernel using the PPPoX socket of the appropriate tunnel or session.

Sample userspace code:

- Create session PPPoX data socket:

```
struct sockaddr_pppol2tp sax;
int fd;

/* Note, the tunnel socket must be bound already, else it
 * will not be ready
 */
sax.sa_family = AF_PPPOX;
sax.sa_protocol = PX_PROTO_OL2TP;
sax.pppol2tp.fd = tunnel_fd;
sax.pppol2tp.addr.sin_addr.s_addr = addr->sin_addr.s_addr;
sax.pppol2tp.addr.sin_port = addr->sin_port;
sax.pppol2tp.addr.sin_family = AF_INET;
```

```

sax.pppol2tp.s_tunnel = tunnel_id;
sax.pppol2tp.s_session = session_id;
sax.pppol2tp.d_tunnel = peer_tunnel_id;
sax.pppol2tp.d_session = peer_session_id;

/* session_fd is the fd of the session's PPPoL2TP socket.
 * tunnel_fd is the fd of the tunnel UDP / L2TPIP socket.
 */
fd = connect(session_fd, (struct sockaddr *)&sax, sizeof(sax));
if (fd < 0) {
    return -errno;
}
return 0;

```

59.2.4 Old L2TPv2-only API

When L2TP was first added to the Linux kernel in 2.6.23, it implemented only L2TPv2 and did not include a netlink API. Instead, tunnel and session instances in the kernel were managed directly using only PPPoL2TP sockets. The PPPoL2TP socket is used as described in section "PPPoL2TP Session Socket API" but tunnel and session instances are automatically created on a connect() of the socket instead of being created by a separate netlink request:

- Tunnels are managed using a tunnel management socket which is a dedicated PPPoL2TP socket, connected to (invalid) session id 0. The L2TP tunnel instance is created when the PPPoL2TP tunnel management socket is connected and is destroyed when the socket is closed.
- Session instances are created in the kernel when a PPPoL2TP socket is connected to a non-zero session id. Session parameters are set using setsockopt. The L2TP session instance is destroyed when the socket is closed.

This API is still supported but its use is discouraged. Instead, new L2TPv2 applications should use netlink to first create the tunnel and session, then create a PPPoL2TP socket for the session.

59.2.5 Unmanaged L2TPv3 tunnels

The kernel L2TP subsystem also supports static (unmanaged) L2TPv3 tunnels. Unmanaged tunnels have no userspace tunnel socket, and exchange no control messages with the peer to set up the tunnel; the tunnel is configured manually at each end of the tunnel. All configuration is done using netlink. There is no need for an L2TP userspace application in this case -- the tunnel socket is created by the kernel and configured using parameters sent in the L2TP_CMD_TUNNEL_CREATE netlink request. The ip utility of iproute2 has commands for managing static L2TPv3 tunnels; do ip l2tp help for more information.

59.2.6 Debugging

The L2TP subsystem offers a range of debugging interfaces through the debugfs filesystem.

To access these interfaces, the debugfs filesystem must first be mounted:

```
# mount -t debugfs debugfs /debug
```

Files under the l2tp directory can then be accessed, providing a summary of the current population of tunnel and session contexts existing in the kernel:

```
# cat /debug/l2tp/tunnels
```

The debugfs files should not be used by applications to obtain L2TP state information because the file format is subject to change. It is implemented to provide extra debug information to help diagnose problems. Applications should instead use the netlink API.

In addition the L2TP subsystem implements tracepoints using the standard kernel event tracing API. The available L2TP events can be reviewed as follows:

```
# find /debug/tracing/events/l2tp
```

Finally, /proc/net/pppol2tp is also provided for backwards compatibility with the original pppol2tp code. It lists information about L2TPv2 tunnels and sessions only. Its use is discouraged.

59.3 Internal Implementation

This section is for kernel developers and maintainers.

59.3.1 Sockets

UDP sockets are implemented by the networking core. When an L2TP tunnel is created using a UDP socket, the socket is set up as an encapsulated UDP socket by setting `encap_rcv` and `encap_destroy` callbacks on the UDP socket. `l2tp_udp_encap_recv` is called when packets are received on the socket. `l2tp_udp_encap_destroy` is called when userspace closes the socket.

L2TPIP sockets are implemented in `net/l2tp/l2tp_ip.c` and `net/l2tp/l2tp_ip6.c`.

59.3.2 Tunnels

The kernel keeps a struct `l2tp_tunnel` context per L2TP tunnel. The `l2tp_tunnel` is always associated with a UDP or L2TP/IP socket and keeps a list of sessions in the tunnel. When a tunnel is first registered with L2TP core, the reference count on the socket is increased. This ensures that the socket cannot be removed while L2TP's data structures reference it.

Tunnels are identified by a unique tunnel id. The id is 16-bit for L2TPv2 and 32-bit for L2TPv3. Internally, the id is stored as a 32-bit value.

Tunnels are kept in a per-net list, indexed by tunnel id. The tunnel id namespace is shared by L2TPv2 and L2TPv3. The tunnel context can be derived from the socket's `sk_user_data`.

Handling tunnel socket close is perhaps the most tricky part of the L2TP implementation. If userspace closes a tunnel socket, the L2TP tunnel and all of its sessions must be closed and destroyed. Since the tunnel context holds a ref on the tunnel socket, the socket's `sk_destruct` won't be called until the tunnel `sock_put`'s its socket. For UDP sockets, when userspace closes the tunnel socket, the socket's `encap_destroy` handler is invoked, which L2TP uses to initiate its tunnel close actions. For L2TPIP sockets, the socket's close handler initiates the same tunnel close actions. All sessions are first closed. Each session drops its tunnel ref. When the tunnel ref reaches zero, the tunnel puts its socket ref. When the socket is eventually destroyed, its `sk_destruct` finally frees the L2TP tunnel context.

59.3.3 Sessions

The kernel keeps a struct `l2tp_session` context for each session. Each session has private data which is used for data specific to the session type. With L2TPv2, the session always carries PPP traffic. With L2TPv3, the session can carry Ethernet frames (Ethernet pseudowire) or other data types such as PPP, ATM, HDLC or Frame Relay. Linux currently implements only Ethernet and PPP session types.

Some L2TP session types also have a socket (PPP pseudowires) while others do not (Ethernet pseudowires). We can't therefore use the socket reference count as the reference count for session contexts. The L2TP implementation therefore has its own internal reference counts on the session contexts.

Like tunnels, L2TP sessions are identified by a unique session id. Just as with tunnel ids, the session id is 16-bit for L2TPv2 and 32-bit for L2TPv3. Internally, the id is stored as a 32-bit value.

Sessions hold a ref on their parent tunnel to ensure that the tunnel stays extant while one or more sessions references it.

Sessions are kept in a per-tunnel list, indexed by session id. L2TPv3 sessions are also kept in a per-net list indexed by session id, because L2TPv3 session ids are unique across all tunnels and L2TPv3 data packets do not contain a tunnel id in the header. This list is therefore needed to find the session context associated with a received data packet when the tunnel context cannot be derived from the tunnel socket.

Although the L2TPv3 RFC specifies that L2TPv3 session ids are not scoped by the tunnel, the kernel does not police this for L2TPv3 UDP tunnels and does not add sessions of L2TPv3 UDP tunnels into the per-net session list. In the UDP receive code, we must trust that the tunnel can be identified using the tunnel socket's `sk_user_data` and lookup the session in the tunnel's session list instead of the per-net session list.

59.3.4 PPP

`net/l2tp/l2tp_ppp.c` implements the PPPoL2TP socket family. Each PPP session has a PPPoL2TP socket.

The PPPoL2TP socket's `sk_user_data` references the `l2tp_session`.

Userspace sends and receives PPP packets over L2TP using a PPPoL2TP socket. Only PPP control frames pass over this socket: PPP data packets are handled entirely by the kernel, passing between the L2TP session and its associated `pppN` netdev through the PPP channel interface of the kernel PPP subsystem.

The L2TP PPP implementation handles the closing of a PPPoL2TP socket by closing its corresponding L2TP session. This is complicated because it must consider racing with netlink session create/destroy requests and `pppol2tp_connect` trying to reconnect with a session that is in the process of being closed. Unlike tunnels, PPP sessions do not hold a ref on their associated socket, so code must be careful to `sock_hold` the socket where necessary. For all the details, see [commit 3d609342cc04129ff7568e19316ce3d7451a27e8](#).

59.3.5 Ethernet

`net/l2tp/l2tp_eth.c` implements L2TPv3 Ethernet pseudowires. It manages a netdev for each session.

L2TP Ethernet sessions are created and destroyed by netlink request, or are destroyed when the tunnel is destroyed. Unlike PPP sessions, Ethernet sessions do not have an associated socket.

59.4 Miscellaneous

59.4.1 RFCs

The kernel code implements the datapath features specified in the following RFCs:

RFC2661	L2TPv2	https://tools.ietf.org/html/rfc2661
RFC3931	L2TPv3	https://tools.ietf.org/html/rfc3931
RFC4719	L2TPv3 Ethernet	https://tools.ietf.org/html/rfc4719

59.4.2 Implementations

A number of open source applications use the L2TP kernel subsystem:

iproute2	https://github.com/shemminger/iproute2
go-l2tp	https://github.com/katalix/go-l2tp
tunneldigger	https://github.com/wlanslovenija/tunneldigger
xl2tpd	https://github.com/xelerance/xl2tpd

59.4.3 Limitations

The current implementation has a number of limitations:

- 1) Multiple UDP sockets with the same 5-tuple address cannot be used. The kernel's tunnel context is identified using private data associated with the socket so it is important that each socket is uniquely identified by its address.
- 2) Interfacing with openvswitch is not yet implemented. It may be useful to map OVS Ethernet and VLAN ports into L2TPv3 tunnels.
- 3) VLAN pseudowires are implemented using an `l2tpethN` interface configured with a VLAN sub-interface. Since L2TPv3 VLAN pseudowires carry one and only one VLAN, it may be better to use a single netdevice rather than an `l2tpethN` and `l2tpethN:M` pair per VLAN

session. The netlink attribute `L2TP_ATTR_VLAN_ID` was added for this, but it was never implemented.

59.4.4 Testing

Unmanaged L2TPv3 Ethernet features are tested by the kernel's built-in selftests. See [tools/testing/selftests/net/l2tp.sh](#).

Another test suite, [l2tp-ktest](#), covers all of the L2TP APIs and tunnel/session types. This may be integrated into the kernel's built-in L2TP selftests in the future.

THE LINUX LAPB MODULE INTERFACE

Version 1.3

Jonathan Naylor 29.12.96

Changed (Henner Eisen, 2000-10-29): int return value for data_indication()

The LAPB module will be a separately compiled module for use by any parts of the Linux operating system that require a LAPB service. This document defines the interfaces to, and the services provided by this module. The term module in this context does not imply that the LAPB module is a separately loadable module, although it may be. The term module is used in its more standard meaning.

The interface to the LAPB module consists of functions to the module, callbacks from the module to indicate important state changes, and structures for getting and setting information about the module.

60.1 Structures

Probably the most important structure is the skbuff structure for holding received and transmitted data, however it is beyond the scope of this document.

The two LAPB specific structures are the LAPB initialisation structure and the LAPB parameter structure. These will be defined in a standard header file, <linux/lapb.h>. The header file <net/lapb.h> is internal to the LAPB module and is not for use.

60.2 LAPB Initialisation Structure

This structure is used only once, in the call to lapb_register (see below). It contains information about the device driver that requires the services of the LAPB module:

```
struct lapb_register_struct {
    void (*connect_confirmation)(int token, int reason);
    void (*connect_indication)(int token, int reason);
    void (*disconnect_confirmation)(int token, int reason);
    void (*disconnect_indication)(int token, int reason);
    int (*data_indication)(int token, struct sk_buff *skb);
    void (*data_transmit)(int token, struct sk_buff *skb);
};
```

Each member of this structure corresponds to a function in the device driver that is called when a particular event in the LAPB module occurs. These will be described in detail below. If a callback is not required (!!) then a NULL may be substituted.

60.3 LAPB Parameter Structure

This structure is used with the `lapb_getparms` and `lapb_setparms` functions (see below). They are used to allow the device driver to get and set the operational parameters of the LAPB implementation for a given connection:

```
struct lapb_parms_struct {
    unsigned int t1;
    unsigned int t1timer;
    unsigned int t2;
    unsigned int t2timer;
    unsigned int n2;
    unsigned int n2count;
    unsigned int window;
    unsigned int state;
    unsigned int mode;
};
```

T1 and T2 are protocol timing parameters and are given in units of 100ms. N2 is the maximum number of tries on the link before it is declared a failure. The window size is the maximum number of outstanding data packets allowed to be unacknowledged by the remote end, the value of the window is between 1 and 7 for a standard LAPB link, and between 1 and 127 for an extended LAPB link.

The mode variable is a bit field used for setting (at present) three values. The bit fields have the following meanings:

Bit	Meaning
0	LAPB operation (0=LAPB_STANDARD 1=LAPB_EXTENDED).
1	[SM]LP operation (0=LAPB_SLP 1=LAPB=MLP).
2	DTE/DCE operation (0=LAPB_DTE 1=LAPB_DCE)
3-31	Reserved, must be 0.

Extended LAPB operation indicates the use of extended sequence numbers and consequently larger window sizes, the default is standard LAPB operation. MLP operation is the same as SLP operation except that the addresses used by LAPB are different to indicate the mode of operation, the default is Single Link Procedure. The difference between DCE and DTE operation is (i) the addresses used for commands and responses, and (ii) when the DCE is not connected, it sends DM without polls set, every T1. The upper case constant names will be defined in the public LAPB header file.

60.4 Functions

The LAPB module provides a number of function entry points.

```
int lapb_register(void *token, struct lapb_register_struct);
```

This must be called before the LAPB module may be used. If the call is successful then LAPB_OK is returned. The token must be a unique identifier generated by the device driver to allow for the unique identification of the instance of the LAPB link. It is returned by the LAPB module in all of the callbacks, and is used by the device driver in all calls to the LAPB module. For multiple LAPB links in a single device driver, multiple calls to lapb_register must be made. The format of the lapb_register_struct is given above. The return values are:

LAPB_OK	LAPB registered successfully.
LAPB_BADTOKEN	Token is already registered.
LAPB_NOMEM	Out of memory

```
int lapb_unregister(void *token);
```

This releases all the resources associated with a LAPB link. Any current LAPB link will be abandoned without further messages being passed. After this call, the value of token is no longer valid for any calls to the LAPB function. The valid return values are:

LAPB_OK	LAPB unregistered successfully.
LAPB_BADTOKEN	Invalid/unknown LAPB token.

```
int lapb_getparms(void *token, struct lapb_parms_struct *parms);
```

This allows the device driver to get the values of the current LAPB variables, the lapb_parms_struct is described above. The valid return values are:

LAPB_OK	LAPB getparms was successful.
LAPB_BADTOKEN	Invalid/unknown LAPB token.

```
int lapb_setparms(void *token, struct lapb_parms_struct *parms);
```

This allows the device driver to set the values of the current LAPB variables, the lapb_parms_struct is described above. The values of t1timer, t2timer and n2count are ignored, likewise changing the mode bits when connected will be ignored. An error implies that none of the values have been changed. The valid return values are:

LAPB_OK	LAPB getparms was successful.
LAPB_BADTOKEN	Invalid/unknown LAPB token.
LAPB_INVALUE	One of the values was out of its allowable range.

```
int lapb_connect_request(void *token);
```

Initiate a connect using the current parameter settings. The valid return values are:

LAPB_OK	LAPB is starting to connect.
LAPB_BADTOKEN	Invalid/unknown LAPB token.
LAPB_CONNECTED	LAPB module is already connected.

```
int lapb_disconnect_request(void *token);
```

Initiate a disconnect. The valid return values are:

LAPB_OK	LAPB is starting to disconnect.
LAPB_BADTOKEN	Invalid/unknown LAPB token.
LAPB_NOTCONNECTED	LAPB module is not connected.

```
int lapb_data_request(void *token, struct sk_buff *skb);
```

Queue data with the LAPB module for transmitting over the link. If the call is successful then the skbuff is owned by the LAPB module and may not be used by the device driver again. The valid return values are:

LAPB_OK	LAPB has accepted the data.
LAPB_BADTOKEN	Invalid/unknown LAPB token.
LAPB_NOTCONNECTED	LAPB module is not connected.

```
int lapb_data_received(void *token, struct sk_buff *skb);
```

Queue data with the LAPB module which has been received from the device. It is expected that the data passed to the LAPB module has skb->data pointing to the beginning of the LAPB data. If the call is successful then the skbuff is owned by the LAPB module and may not be used by the device driver again. The valid return values are:

LAPB_OK	LAPB has accepted the data.
LAPB_BADTOKEN	Invalid/unknown LAPB token.

60.5 Callbacks

These callbacks are functions provided by the device driver for the LAPB module to call when an event occurs. They are registered with the LAPB module with `lapb_register` (see above) in the structure `lapb_register_struct` (see above).

```
void (*connect_confirmation)(void *token, int reason);
```

This is called by the LAPB module when a connection is established after being requested by a call to `lapb_connect_request` (see above). The reason is always `LAPB_OK`.

```
void (*connect_indication)(void *token, int reason);
```

This is called by the LAPB module when the link is established by the remote system. The value of reason is always LAPB_OK.

```
void (*disconnect_confirmation)(void *token, int reason);
```

This is called by the LAPB module when an event occurs after the device driver has called `lapb_disconnect_request` (see above). The reason indicates what has happened. In all cases the LAPB link can be regarded as being terminated. The values for reason are:

LAPB_OK	The LAPB link was terminated normally.
LAPB_NOTCONNECTED	The remote system was not connected.
LAPB_TIMEDOUT	No response was received in N2 tries from the remote system.

```
void (*disconnect_indication)(void *token, int reason);
```

This is called by the LAPB module when the link is terminated by the remote system or another event has occurred to terminate the link. This may be returned in response to a `lapb_connect_request` (see above) if the remote system refused the request. The values for reason are:

LAPB_OK	The LAPB link was terminated normally by the remote system.
LAPB_REFUSED	The remote system refused the connect request.
LAPB_NOTCONNECTED	The remote system was not connected.
LAPB_TIMEDOUT	No response was received in N2 tries from the remote system.

```
int (*data_indication)(void *token, struct sk_buff *skb);
```

This is called by the LAPB module when data has been received from the remote system that should be passed onto the next layer in the protocol stack. The skbuff becomes the property of the device driver and the LAPB module will not perform any more actions on it. The `skb->data` pointer will be pointing to the first byte of data after the LAPB header.

This method should return `NET_RX_DROP` (as defined in the header file `include/linux/netdevice.h`) if and only if the frame was dropped before it could be delivered to the upper layer.

```
void (*data_transmit)(void *token, struct sk_buff *skb);
```

This is called by the LAPB module when data is to be transmitted to the remote system by the device driver. The skbuff becomes the property of the device driver and the LAPB module will not perform any more actions on it. The `skb->data` pointer will be pointing to the first byte of the LAPB header.

HOW TO USE PACKET INJECTION WITH MAC80211

mac80211 now allows arbitrary packets to be injected down any Monitor Mode interface from userland. The packet you inject needs to be composed in the following format:

```
[ radiotap header ]  
[ ieee80211 header ]  
[ payload ]
```

The radiotap format is discussed in [./Documentation/networking/radiotap-headers.rst](#).

Despite many radiotap parameters being currently defined, most only make sense to appear on received packets. The following information is parsed from the radiotap headers and used to control injection:

- IEEE80211_RADIOTAP_FLAGS

IEEE80211_RADIOTAP_F_FCS	FCS will be removed and recalculated
IEEE80211_RADIOTAP_F_WEP	frame will be encrypted if key available
IEEE80211_RADIOTAP_F_FRAG	frame will be fragmented if longer than the current fragmentation threshold.

- IEEE80211_RADIOTAP_TX_FLAGS

IEEE80211_RADIOTAP_F_TX_NOACK	frame should be sent without waiting for an ACK even if it is a unicast frame
-------------------------------	---

- IEEE80211_RADIOTAP_RATE

legacy rate for the transmission (only for devices without own rate control)

- IEEE80211_RADIOTAP_MCS

HT rate for the transmission (only for devices without own rate control). Also some flags are parsed

IEEE80211_RADIOTAP_MCS_SGI	use short guard interval
IEEE80211_RADIOTAP_MCS_BW_40	send in HT40 mode

- IEEE80211_RADIOTAP_DATA_RETRIES

number of retries when either IEEE80211_RADIOTAP_RATE or IEEE80211_RADIOTAP_MCS was used

- IEEE80211_RADIOTAP_VHT

VHT mcs and number of streams used in the transmission (only for devices without own rate control). Also other fields are parsed

flags field

IEEE80211_RADIOTAP_VHT_FLAG_SGI: use short guard interval

bandwidth field

- 1: send using 40MHz channel width
- 4: send using 80MHz channel width
- 11: send using 160MHz channel width

The injection code can also skip all other currently defined radiotap fields facilitating replay of captured radiotap headers directly.

Here is an example valid radiotap header defining some parameters:

```
0x00, 0x00, // <-- radiotap version
0x0b, 0x00, // <- radiotap header length
0x04, 0x0c, 0x00, 0x00, // <-- bitmap
0x6c, // <-- rate
0x0c, //<-- tx power
0x01 //<-- antenna
```

The ieee80211 header follows immediately afterwards, looking for example like this:

```
0x08, 0x01, 0x00, 0x00,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0x13, 0x22, 0x33, 0x44, 0x55, 0x66,
0x13, 0x22, 0x33, 0x44, 0x55, 0x66,
0x10, 0x86
```

Then lastly there is the payload.

After composing the packet contents, it is sent by send()-ing it to a logical mac80211 interface that is in Monitor mode. Libpcap can also be used, (which is easier than doing the work to bind the socket to the right interface), along the following lines::

```
ppcap = pcap_open_live(szInterfaceName, 800, 1, 20, szErrbuf);
...
r = pcap_inject(ppcap, u8aSendBuffer, nLength);
```

You can also find a link to a complete inject application here:

<https://wireless.wiki.kernel.org/en/users/Documentation/packetspammer>

Andy Green <andy@warmcat.com>

MANAGEMENT COMPONENT TRANSPORT PROTOCOL (MCTP)

net/mctp/ contains protocol support for MCTP, as defined by DMTF standard DSP0236. Physical interface drivers ("bindings" in the specification) are provided in drivers/net/mctp/.

The core code provides a socket-based interface to send and receive MCTP messages, through an AF_MCTP, SOCK_DGRAM socket.

62.1 Structure: interfaces & networks

The kernel models the local MCTP topology through two items: interfaces and networks.

An interface (or "link") is an instance of an MCTP physical transport binding (as defined by DSP0236, section 3.2.47), likely connected to a specific hardware device. This is represented as a `struct netdevice`.

A network defines a unique address space for MCTP endpoints by endpoint-ID (described by DSP0236, section 3.2.31). A network has a user-visible identifier to allow references from userspace. Route definitions are specific to one network.

Interfaces are associated with one network. A network may be associated with one or more interfaces.

If multiple networks are present, each may contain endpoint IDs (EIDs) that are also present on other networks.

62.2 Sockets API

62.2.1 Protocol definitions

MCTP uses AF_MCTP / PF_MCTP for the address- and protocol- families. Since MCTP is message-based, only SOCK_DGRAM sockets are supported.

```
int sd = socket(AF_MCTP, SOCK_DGRAM, 0);
```

The only (current) value for the protocol argument is 0.

As with all socket address families, source and destination addresses are specified with a `sockaddr` type, with a single-byte endpoint address:

```

typedef __u8 mctp_eid_t;

struct mctp_addr {
    mctp_eid_t s_addr;
};

struct sockaddr_mctp {
    __kernel_sa_family_t smctp_family;
    unsigned int smctp_network;
    struct mctp_addr smctp_addr;
    __u8 smctp_type;
    __u8 smctp_tag;
};

#define MCTP_NET_ANY 0x0
#define MCTP_ADDR_ANY 0xff

```

62.2.2 Syscall behaviour

The following sections describe the MCTP-specific behaviours of the standard socket system calls. These behaviours have been chosen to map closely to the existing sockets APIs.

bind() : set local socket address

Sockets that receive incoming request packets will bind to a local address, using the `bind()` syscall.

```

struct sockaddr_mctp addr;

addr.smctp_family = AF_MCTP;
addr.smctp_network = MCTP_NET_ANY;
addr.smctp_addr.s_addr = MCTP_ADDR_ANY;
addr.smctp_type = MCTP_TYPE_PLDM;
addr.smctp_tag = MCTP_TAG_OWNER;

int rc = bind(sd, (struct sockaddr *)&addr, sizeof(addr));

```

This establishes the local address of the socket. Incoming MCTP messages that match the network, address, and message type will be received by this socket. The reference to 'incoming' is important here; a bound socket will only receive messages with the TO bit set, to indicate an incoming request message, rather than a response.

The `smctp_tag` value will configure the tags accepted from the remote side of this socket. Given the above, the only valid value is `MCTP_TAG_OWNER`, which will result in remotely "owned" tags being routed to this socket. Since `MCTP_TAG_OWNER` is set, the 3 least-significant bits of `smctp_tag` are not used; callers must set them to zero.

A `smctp_network` value of `MCTP_NET_ANY` will configure the socket to receive incoming packets from any locally-connected network. A specific network value will cause the socket to only receive incoming messages from that network.

The `smctp_addr` field specifies a local address to bind to. A value of `MCTP_ADDR_ANY` configures the socket to receive messages addressed to any local destination EID.

The `smctp_type` field specifies which message types to receive. Only the lower 7 bits of the type is matched on incoming messages (ie., the most-significant IC bit is not part of the match). This results in the socket receiving packets with and without a message integrity check footer.

`sendto()`, `sendmsg()`, `send()` : transmit an MCTP message

An MCTP message is transmitted using one of the `sendto()`, `sendmsg()` or `send()` syscalls. Using `sendto()` as the primary example:

```
struct sockaddr_mctp addr;
char buf[14];
ssize_t len;

/* set message destination */
addr.smctp_family = AF_MCTP;
addr.smctp_network = 0;
addr.smctp_addr.s_addr = 8;
addr.smctp_tag = MCTP_TAG_OWNER;
addr.smctp_type = MCTP_TYPE_ECHO;

/* arbitrary message to send, with message-type header */
buf[0] = MCTP_TYPE_ECHO;
memcpy(buf + 1, "hello, world!", sizeof(buf) - 1);

len = sendto(sd, buf, sizeof(buf), 0,
             (struct sockaddr_mctp *)&addr, sizeof(addr));
```

The network and address fields of `addr` define the remote address to send to. If `smctp_tag` has the `MCTP_TAG_OWNER`, the kernel will ignore any bits set in `MCTP_TAG_VALUE`, and generate a tag value suitable for the destination EID. If `MCTP_TAG_OWNER` is not set, the message will be sent with the tag value as specified. If a tag value cannot be allocated, the system call will report an `errno` of `EAGAIN`.

The application must provide the message type byte as the first byte of the message buffer passed to `sendto()`. If a message integrity check is to be included in the transmitted message, it must also be provided in the message buffer, and the most-significant bit of the message type byte must be 1.

The `sendmsg()` system call allows a more compact argument interface, and the message buffer to be specified as a scatter-gather list. At present no ancillary message types (used for the `msg_control` data passed to `sendmsg()`) are defined.

Transmitting a message on an unconnected socket with `MCTP_TAG_OWNER` specified will cause an allocation of a tag, if no valid tag is already allocated for that destination. The (destination-eid,tag) tuple acts as an implicit local socket address, to allow the socket to receive responses to this outgoing message. If any previous allocation has been performed (to for a different remote EID), that allocation is lost.

Sockets will only receive responses to requests they have sent (with `TO=1`) and may only respond (with `TO=0`) to requests they have received.

recvfrom(), recvmsg(), recv() : receive an MCTP message

An MCTP message can be received by an application using one of the `recvfrom()`, `recvmsg()`, or `recv()` system calls. Using `recvfrom()` as the primary example:

```
struct sockaddr_mctp addr;
socklen_t addrlen;
char buf[14];
ssize_t len;

addrlen = sizeof(addr);

len = recvfrom(sd, buf, sizeof(buf), 0,
               (struct sockaddr_mctp *)&addr, &addrlen);

/* We can expect addr to describe an MCTP address */
assert(addrlen >= sizeof(buf));
assert(addr.smctp_family == AF_MCTP);

printf("received %zd bytes from remote EID %d\n", rc, addr.smctp_addr);
```

The address argument to `recvfrom` and `recvmsg` is populated with the remote address of the incoming message, including tag value (this will be needed in order to reply to the message).

The first byte of the message buffer will contain the message type byte. If an integrity check follows the message, it will be included in the received buffer.

The `recv()` system call behaves in a similar way, but does not provide a remote address to the application. Therefore, these are only useful if the remote address is already known, or the message does not require a reply.

Like the send calls, sockets will only receive responses to requests they have sent (TO=1) and may only respond (TO=0) to requests they have received.

ioctl(SI0CMCTPALLOCTAG) and ioctl(SI0CMCTPDROPTAG)

These tags give applications more control over MCTP message tags, by allocating (and dropping) tag values explicitly, rather than the kernel automatically allocating a per-message tag at `sendmsg()` time.

In general, you will only need to use these ioctls if your MCTP protocol does not fit the usual request/response model. For example, if you need to persist tags across multiple requests, or a request may generate more than one response. In these cases, the ioctls allow you to decouple the tag allocation (and release) from individual message send and receive operations.

Both ioctls are passed a pointer to a `struct mctp_ioc_tag_ctl`:

```
struct mctp_ioc_tag_ctl {
    mctp_eid_t      peer_addr;
    __u8            tag;
    __u16           flags;
};
```

`SIOCMCTPALLOCATE` allocates a tag for a specific peer, which an application can use in future `sendmsg()` calls. The application populates the `peer_addr` member with the remote EID. Other fields must be zero.

On return, the `tag` member will be populated with the allocated tag value. The allocated tag will have the following tag bits set:

- `MCTP_TAG_OWNER`: it only makes sense to allocate tags if you're the tag owner
- `MCTP_TAG_PREALLOC`: to indicate to `sendmsg()` that this is a preallocated tag.
- ... and the actual tag value, within the least-significant three bits (`MCTP_TAG_MASK`). Note that zero is a valid tag value.

The tag value should be used as-is for the `smctp_tag` member of `struct sockaddr_mctp`.

`SIOCMCTPDROPTAG` releases a tag that has been previously allocated by a `SIOCMCTPALLOCATE` ioctl. The `peer_addr` must be the same as used for the allocation, and the `tag` value must match exactly the tag returned from the allocation (including the `MCTP_TAG_OWNER` and `MCTP_TAG_PREALLOC` bits). The `flags` field must be zero.

62.3 Kernel internals

There are a few possible packet flows in the MCTP stack:

1. local TX to remote endpoint, message <= MTU:

```
sendmsg()
-> mctp_local_output()
: route lookup
-> rt->output() (== mctp_route_output)
-> dev_queue_xmit()
```

2. local TX to remote endpoint, message > MTU:

```
sendmsg()
-> mctp_local_output()
-> mctp_do_fragment_route()
: creates packet-sized skbs. For each new skb:
-> rt->output() (== mctp_route_output)
-> dev_queue_xmit()
```

3. remote TX to local endpoint, single-packet message:

```
mctp_pktype_receive()
: route lookup
-> rt->output() (== mctp_route_input)
: sk_key lookup
-> sock_queue_rcv_skb()
```

4. remote TX to local endpoint, multiple-packet message:

```
mctp_pktype_receive()
: route lookup
```

```
-> rt->output() (== mctp_route_input)
: sk_key lookup
: stores skb in struct sk_key->reasmb_head

mctp_pktype_receive()
: route lookup
-> rt->output() (== mctp_route_input)
: sk_key lookup
: finds existing reassembly in sk_key->reasmb_head
: appends new fragment
-> sock_queue_rcv_skb()
```

62.3.1 Key refcounts

- keys are refed by:
 - a skb: during route output, stored in skb->cb.
 - netns and sock lists.
- keys can be associated with a device, in which case they hold a reference to the dev (set through key->dev, counted through dev->key_count). Multiple keys can reference the device.

MPLS SYSFS VARIABLES

63.1 /proc/sys/net/mpls/* Variables:

platform_labels - INTEGER

Number of entries in the platform label table. It is not possible to configure forwarding for label values equal to or greater than the number of platform labels.

A dense utilization of the entries in the platform label table is possible and expected as the platform labels are locally allocated.

If the number of platform label table entries is set to 0 no label will be recognized by the kernel and mpls forwarding will be disabled.

Reducing this value will remove all label routing entries that no longer fit in the table.

Possible values: 0 - 1048575

Default: 0

ip_ttl_propagate - BOOL

Control whether TTL is propagated from the IPv4/IPv6 header to the MPLS header on imposing labels and propagated from the MPLS header to the IPv4/IPv6 header on popping the last label.

If disabled, the MPLS transport network will appear as a single hop to transit traffic.

- 0 - disabled / RFC 3443 [Short] Pipe Model
- 1 - enabled / RFC 3443 Uniform Model (default)

default_ttl - INTEGER

Default TTL value to use for MPLS packets where it cannot be propagated from an IP header, either because one isn't present or ip_ttl_propagate has been disabled.

Possible values: 1 - 255

Default: 255

conf/<interface>/input - BOOL

Control whether packets can be input on this interface.

If disabled, packets will be discarded without further processing.

- 0 - disabled (default)
- not 0 - enabled

MPTCP SYSFS VARIABLES

64.1 /proc/sys/net/mptcp/* Variables

enabled - BOOLEAN

Control whether MPTCP sockets can be created.

MPTCP sockets can be created if the value is 1. This is a per-namespace sysctl.

Default: 1 (enabled)

add_addr_timeout - INTEGER (seconds)

Set the timeout after which an ADD_ADDR control message will be resent to an MPTCP peer that has not acknowledged a previous ADD_ADDR message.

The default value matches TCP_RTO_MAX. This is a per-namespace sysctl.

Default: 120

close_timeout - INTEGER (seconds)

Set the make-after-break timeout: in absence of any close or shutdown syscall, MPTCP sockets will maintain the status unchanged for such time, after the last subflow removal, before moving to TCP_CLOSE.

The default value matches TCP_TIMEWAIT_LEN. This is a per-namespace sysctl.

Default: 60

checksum_enabled - BOOLEAN

Control whether DSS checksum can be enabled.

DSS checksum can be enabled if the value is nonzero. This is a per-namespace sysctl.

Default: 0

allow_join_initial_addr_port - BOOLEAN

Allow peers to send join requests to the IP address and port number used by the initial subflow if the value is 1. This controls a flag that is sent to the peer at connection time, and whether such join requests are accepted or denied.

Joins to addresses advertised with ADD_ADDR are not affected by this value.

This is a per-namespace sysctl.

Default: 1

pm_type - INTEGER

Set the default path manager type to use for each new MPTCP socket. In-kernel path management will control subflow connections and address advertisements according to per-namespace values configured over the MPTCP netlink API. Userspace path management puts per-MPTCP-connection subflow connection decisions and address advertisements under control of a privileged userspace program, at the cost of more netlink traffic to propagate all of the related events and commands.

This is a per-namespace sysctl.

- 0 - In-kernel path manager
- 1 - Userspace path manager

Default: 0

stale_loss_cnt - INTEGER

The number of MPTCP-level retransmission intervals with no traffic and pending outstanding data on a given subflow required to declare it stale. The packet scheduler ignores stale subflows. A low stale_loss_cnt value allows for fast active-backup switch-over, an high value maximize links utilization on edge scenarios e.g. lossy link with high BER or peer pausing the data processing.

This is a per-namespace sysctl.

Default: 4

scheduler - STRING

Select the scheduler of your choice.

Support for selection of different schedulers. This is a per-namespace sysctl.

Default: "default"

HOWTO FOR MULTIQUEUE NETWORK DEVICE SUPPORT

65.1 Section 1: Base driver requirements for implementing multiqueue support

65.1.1 Intro: Kernel support for multiqueue devices

Kernel support for multiqueue devices is always present.

Base drivers are required to use the new alloc_etherdev_mq() or alloc_netdev_mq() functions to allocate the subqueues for the device. The underlying kernel API will take care of the allocation and deallocation of the subqueue memory, as well as netdev configuration of where the queues exist in memory.

The base driver will also need to manage the queues as it does the global netdev->queue_lock today. Therefore base drivers should use the netif_{start|stop|wake}_subqueue() functions to manage each queue while the device is still operational. netdev->queue_lock is still used when the device comes online or when it's completely shut down ([unregister_netdev\(\)](#), etc.).

65.2 Section 2: Qdisc support for multiqueue devices

Currently two qdiscs are optimized for multiqueue devices. The first is the default pfifo_fast qdisc. This qdisc supports one qdisc per hardware queue. A new round-robin qdisc, sch_multiq also supports multiple hardware queues. The qdisc is responsible for classifying the skb's and then directing the skb's to bands and queues based on the value in skb->queue_mapping. Use this field in the base driver to determine which queue to send the skb to.

sch_multiq has been added for hardware that wishes to avoid head-of-line blocking. It will cycle through the bands and verify that the hardware queue associated with the band is not stopped prior to dequeuing a packet.

On qdisc load, the number of bands is based on the number of queues on the hardware. Once the association is made, any skb with skb->queue_mapping set, will be queued to the band associated with the hardware queue.

65.3 Section 3: Brief howto using MULTIQ for multiqueue devices

The userspace command 'tc,' part of the iproute2 package, is used to configure qdiscs. To add the MULTIQ qdisc to your network device, assuming the device is called eth0, run the following command:

```
# tc qdisc add dev eth0 root handle 1: multiq
```

The qdisc will allocate the number of bands to equal the number of queues that the device reports, and bring the qdisc online. Assuming eth0 has 4 Tx queues, the band mapping would look like:

```
band 0 => queue 0
band 1 => queue 1
band 2 => queue 2
band 3 => queue 3
```

Traffic will begin flowing through each queue based on either the simple_tx_hash function or based on netdev->select_queue() if you have it defined.

The behavior of tc filters remains the same. However a new tc action, skbedit, has been added. Assuming you wanted to route all traffic to a specific host, for example 192.168.0.3, through a specific queue you could use this action and establish a filter such as:

```
tc filter add dev eth0 parent 1: protocol ip prio 1 u32 \
    match ip dst 192.168.0.3 \
    action skbedit queue_mapping 3
```

Author

Alexander Duyck <alexander.h.duyck@intel.com>

Original Author

Peter P. Waskiewicz Jr. <peter.p.waskiewicz.jr@intel.com>

NAPI

NAPI is the event handling mechanism used by the Linux networking stack. The name NAPI no longer stands for anything in particular¹.

In basic operation the device notifies the host about new events via an interrupt. The host then schedules a NAPI instance to process the events. The device may also be polled for events via NAPI without receiving interrupts first (*busy polling*).

NAPI processing usually happens in the software interrupt context, but there is an option to use *separate kernel threads* for NAPI processing.

All in all NAPI abstracts away from the drivers the context and configuration of event (packet Rx and Tx) processing.

66.1 Driver API

The two most important elements of NAPI are the struct napi_struct and the associated poll method. struct napi_struct holds the state of the NAPI instance while the method is the driver-specific event handler. The method will typically free Tx packets that have been transmitted and process newly received packets.

66.1.1 Control API

`netif_napi_add()` and `netif_napi_del()` add/remove a NAPI instance from the system. The instances are attached to the netdevice passed as argument (and will be deleted automatically when netdevice is unregistered). Instances are added in a disabled state.

`napi_enable()` and `napi_disable()` manage the disabled state. A disabled NAPI can't be scheduled and its poll method is guaranteed to not be invoked. `napi_disable()` waits for ownership of the NAPI instance to be released.

The control APIs are not idempotent. Control API calls are safe against concurrent use of datapath APIs but an incorrect sequence of control API calls may result in crashes, deadlocks, or race conditions. For example, calling `napi_disable()` multiple times in a row will deadlock.

¹ NAPI was originally referred to as New API in 2.4 Linux.

66.1.2 Datapath API

`napi_schedule()` is the basic method of scheduling a NAPI poll. Drivers should call this function in their interrupt handler (see *Scheduling and IRQ masking* for more info). A successful call to `napi_schedule()` will take ownership of the NAPI instance.

Later, after NAPI is scheduled, the driver's poll method will be called to process the events/packets. The method takes a budget argument - drivers can process completions for any number of Tx packets but should only process up to budget number of Rx packets. Rx processing is usually much more expensive.

In other words for Rx processing the budget argument limits how many packets driver can process in a single poll. Rx specific APIs like page pool or XDP cannot be used at all when budget is 0. skb Tx processing should happen regardless of the budget, but if the argument is 0 driver cannot call any XDP (or page pool) APIs.

Warning: The budget argument may be 0 if core tries to only process skb Tx completions and no Rx or XDP packets.

The poll method returns the amount of work done. If the driver still has outstanding work to do (e.g. budget was exhausted) the poll method should return exactly budget. In that case, the NAPI instance will be serviced/polled again (without the need to be scheduled).

If event processing has been completed (all outstanding packets processed) the poll method should call `napi_complete_done()` before returning. `napi_complete_done()` releases the ownership of the instance.

Warning: The case of finishing all events and using exactly budget must be handled carefully. There is no way to report this (rare) condition to the stack, so the driver must either not call `napi_complete_done()` and wait to be called again, or return budget - 1.

If the budget is 0 `napi_complete_done()` should never be called.

66.1.3 Call sequence

Drivers should not make assumptions about the exact sequencing of calls. The poll method may be called without the driver scheduling the instance (unless the instance is disabled). Similarly, it's not guaranteed that the poll method will be called, even if `napi_schedule()` succeeded (e.g. if the instance gets disabled).

As mentioned in the *Control API* section - `napi_disable()` and subsequent calls to the poll method only wait for the ownership of the instance to be released, not for the poll method to exit. This means that drivers should avoid accessing any data structures after calling `napi_complete_done()`.

66.1.4 Scheduling and IRQ masking

Drivers should keep the interrupts masked after scheduling the NAPI instance - until NAPI polling finishes any further interrupts are unnecessary.

Drivers which have to mask the interrupts explicitly (as opposed to IRQ being auto-masked by the device) should use the `napi_schedule_prep()` and `_napi_schedule()` calls:

```
if (napi_schedule_prep(&v->napi)) {
    mydrv_mask_rxtx_irq(v->idx);
    /* schedule after masking to avoid races */
    __napi_schedule(&v->napi);
}
```

IRQ should only be unmasked after a successful call to `napi_complete_done()`:

```
if (budget && napi_complete_done(&v->napi, work_done)) {
    mydrv_unmask_rxtx_irq(v->idx);
    return min(work_done, budget - 1);
}
```

`napi_schedule_irqoff()` is a variant of `napi_schedule()` which takes advantage of guarantees given by being invoked in IRQ context (no need to mask interrupts). Note that PREEMPT_RT forces all interrupts to be threaded so the interrupt may need to be marked IRQF_NO_THREAD to avoid issues on real-time kernel configurations.

66.1.5 Instance to queue mapping

Modern devices have multiple NAPI instances (struct `napi_struct`) per interface. There is no strong requirement on how the instances are mapped to queues and interrupts. NAPI is primarily a polling/processing abstraction without specific user-facing semantics. That said, most networking devices end up using NAPI in fairly similar ways.

NAPI instances most often correspond 1:1:1 to interrupts and queue pairs (queue pair is a set of a single Rx and single Tx queue).

In less common cases a NAPI instance may be used for multiple queues or Rx and Tx queues can be serviced by separate NAPI instances on a single core. Regardless of the queue assignment, however, there is usually still a 1:1 mapping between NAPI instances and interrupts.

It's worth noting that the ethtool API uses a "channel" terminology where each channel can be either rx, tx or combined. It's not clear what constitutes a channel; the recommended interpretation is to understand a channel as an IRQ/NAPI which services queues of a given type. For example, a configuration of 1 rx, 1 tx and 1 combined channel is expected to utilize 3 interrupts, 2 Rx and 2 Tx queues.

66.2 User API

User interactions with NAPI depend on NAPI instance ID. The instance IDs are only visible to the user thru the `SO_INCOMING_NAPI_ID` socket option. It's not currently possible to query IDs used by a given device.

66.2.1 Software IRQ coalescing

NAPI does not perform any explicit event coalescing by default. In most scenarios batching happens due to IRQ coalescing which is done by the device. There are cases where software coalescing is helpful.

NAPI can be configured to arm a repoll timer instead of unmasking the hardware interrupts as soon as all packets are processed. The `gro_flush_timeout` sysfs configuration of the netdevice is reused to control the delay of the timer, while `napi_defer_hard_irqs` controls the number of consecutive empty polls before NAPI gives up and goes back to using hardware IRQs.

66.2.2 Busy polling

Busy polling allows a user process to check for incoming packets before the device interrupt fires. As is the case with any busy polling it trades off CPU cycles for lower latency (production uses of NAPI busy polling are not well known).

Busy polling is enabled by either setting `SO_BUSY_POLL` on selected sockets or using the global `net.core.busy_poll` and `net.core.busy_read` sysctls. An `io_uring` API for NAPI busy polling also exists.

66.2.3 IRQ mitigation

While busy polling is supposed to be used by low latency applications, a similar mechanism can be used for IRQ mitigation.

Very high request-per-second applications (especially routing/forwarding applications and especially applications using `AF_XDP` sockets) may not want to be interrupted until they finish processing a request or a batch of packets.

Such applications can pledge to the kernel that they will perform a busy polling operation periodically, and the driver should keep the device IRQs permanently masked. This mode is enabled by using the `SO_PREFER_BUSY_POLL` socket option. To avoid system misbehavior the pledge is revoked if `gro_flush_timeout` passes without any busy poll call.

The NAPI budget for busy polling is lower than the default (which makes sense given the low latency intention of normal busy polling). This is not the case with IRQ mitigation, however, so the budget can be adjusted with the `SO_BUSY_POLL_BUDGET` socket option.

66.2.4 Threaded NAPI

Threaded NAPI is an operating mode that uses dedicated kernel threads rather than software IRQ context for NAPI processing. The configuration is per netdevice and will affect all NAPI instances of that device. Each NAPI instance will spawn a separate thread (called `napi/${ifc-name}-${napi-id}`).

It is recommended to pin each kernel thread to a single CPU, the same CPU as the CPU which services the interrupt. Note that the mapping between IRQs and NAPI instances may not be trivial (and is driver dependent). The NAPI instance IDs will be assigned in the opposite order than the process IDs of the kernel threads.

Threaded NAPI is controlled by writing 0/1 to the `threaded` file in netdev's sysfs directory.

COMMON NETWORKING STRUCT CACHELINES

67.1 `inet_connection_sock` struct fast path usage breakdown

Type	Name	fastpath_tx_access	fastpath_rx_access	comment	..struct		
.inet_connection_sock	struct_inet_sock	icsk_inet	read_mostly	read_mostly			
tcp_init_buffer_space,tcp_init_transfer,tcp_finish_connect,tcp_connect,tcp_send_rcvq,tcp_send_syn_d	struct_request_sock_queue	icsk_accept_queue	- - -	struct_inet_bind_bucket			
icsk_bind_hash	read_mostly	-	tcp_set_state	struct_inet_bind2_bucket	icsk_bind2_hash		
read_mostly	-	tcp_set_state,inet_put_port	unsigned_long	icsk_timeout	read_mostly		
-	inet_csk_reset_xmit_timer,tcp_connect	struct_timer_list	icsk_retransmit_timer				
read_mostly	-	inet_csk_reset_xmit_timer,tcp_connect	struct_timer_list	icsk_delack_timer			
read_mostly	-	inet_csk_reset_xmit_timer,tcp_connect	u32	icsk_rto	read_write		
tcp_cwnd_validate,tcp_schedule_loss_probe,tcp_connect_init,tcp_connect,tcp_write_xmit,tcp_push_on	u32	icsk_rto_min	- -	u32	icsk_pmtu_cookie	read_write	
-	tcp_sync_mss,tcp_current_mss,tcp_send_syn_data,tcp_connect_init,tcp_connect						
struct_tcp_congestion_ops	icsk_ca_ops	read_write	-	tcp_cwnd_validate,tcp_tso_segs,tcp_ca_dst_init,tcp			
struct_inet_connection_sock_af_ops	icsk_af_ops	read_mostly	-	tcp_finish_connect,tcp_send_syn_data,to			
struct_tcp_ulp_ops*	icsk_ulp_ops	- - void*	icsk_ulp_data	- - u8:5	icsk_ca_state	read_write	
-	tcp_cwnd_application_limited,tcp_set_ca_state,tcp_enter_cwr,tcp_tso_should_defer,tcp_mtu_probe,tcp						
u8:1	icsk_ca_initialized	read_write	-	tcp_init_transfer,tcp_init congestion_control,tcp_init_transfer,tcp			
u8:1	icsk_ca_setssockopt	- - -	u8:1	icsk_ca_dst_locked	write_mostly		
-	tcp_ca_dst_init,tcp_connect_init,tcp_connect		u8	icsk_retransmits			
write_mostly	-	tcp_connect_init,tcp_connect	u8	icsk_pending	read_write		
inet_csk_reset_xmit_timer,tcp_connect,tcp_check_probe_timer,_tcp_push_pending_frames,tcp_rearm	u8	icsk_backoff	write_mostly	-	tcp_write_queue_purge,tcp_connect_init	u8	
icsk_syn_retries	- -	u8	icsk_probes_out	- -	u16	icsk_ext_hdr_len	read_mostly
-	_tcp_mtu_to_mss,tcp_mtu_to_rss,tcp_mtu_probe,tcp_write_xmit,tcp_mtu_to_mss,						
struct_ic_sk_ack_u8	pending read	write	read_write	inet_csk_ack_scheduled,_tcp_cleanup_rbuf,tcp_cle			
sent,inet_csk_reset_xmit_timer	struct_ic_sk_ack_u8	quick	read_write	write_mostly			
tcp_dec_quickack_mode,tcp_event_ack_sent,_tcp_transmit_skb,_tcp_select_window,_tcp_cleanup_r							
struct_ic_sk_ack_u8	pingpong	- -	struct_ic_sk_ack_u8	retry	write_mostly	read_write	
inet_csk_clear_xmit_timer,tcp_rearm_rto,tcp_event_new_data_sent,tcp_write_xmit,_tcp_send_ack,tcp							
struct_ic_sk_ack_u8	ato read	mostly	write	mostly	tcp_dec_quickack_mode,tcp_event_ack_sent,_tcp_tr		
struct_ic_sk_ack_un	signed_long	timeout	read_write	read_write	inet_csk_reset_xmit_timer,tcp_connect		
struct_ic_sk_ack_u32	rcv_time	read_write	-	tcp_finish_connect,tcp_connect,tcp_event_data_sent,_tcp_t			
struct_ic_sk_mtup_int	search_high	read_write	-	tcp_mtup_init,tcp_sync_mss,tcp_connect_init,tcp_mtu_o			
struct_ic_sk_mtup_int	search_low	read_write	-	tcp_mtu_probe,tcp_mtu_check_reprobe,tcp_write_xmit,			
struct_ic_sk_mtup_u32:31	probe_size	read_write	-	tcp_mtup_init,tcp_connect_init,_tcp_transmit_skb			
struct_ic_sk_mtup_u32:1	enabled	read_write	-	tcp_mtup_init,tcp_sync_mss,tcp_connect_init,tcp_mtu_pr			

```
struct _icsk_mtup_u32 probe_timestamp read_write - tcp_mtup_init,tcp_connect_init,tcp_mtu_check_re
u32 icsk_probes_tstamp -- u32 icsk_user_timeout -- u64[104/sizeof(u64)] icsk_ca_priv --
```

67.2 `inet_sock` struct fast path usage breakdown

```
Type Name fastpath_tx_access fastpath_rx_access comment ..struct ..inet_sock struct_sock sk
read_mostly read_mostly tcp_init_buffer_space,tcp_init_transfer,tcp_finish_connect,tcp_connect,tcp_s
struct_ipv6_pinfo* pinet6 -- be16 inet_sport read_mostly - __tcp_transmit_skb be32
inet_daddr read_mostly - ip_select_ident_segs be32 inet_rcv_saddr -- be16 inet_dport
read_mostly - __tcp_transmit_skb u16 inet_num -- be32 inet_saddr -- s16 uc_ttl read_mostly
- __ip_queue_xmit/ip_select_ttl u16 cmsg_flags -- struct_ip_options_rcu* inet_opt read_mostly
- __ip_queue_xmit u16 inet_id read_mostly - ip_select_ident_segs u8 tos read_mostly
- ip_queue_xmit u8 min_ttl -- u8 mc_ttl -- u8 pmtudisc -- u8:1 recvverr -- u8:1 is_icsk --
u8:1 freebind -- u8:1 hdrincl -- u8:1 mc_loop -- u8:1 transparent -- u8:1 mc_all -- u8:1 node-
frag -- u8:1 bind_address_no_port -- u8:1 recvverr_rfc4884 -- u8:1 defer_connect read_mostly
- tcp_sendmsg_fastopen u8 rcv_tos -- u8 convert_csum -- int uc_index -- int mc_index --
be32 mc_addr -- struct_ip_mc_socklist* mc_list -- struct_inet_cork_full cork read_mostly
- __tcp_transmit_skb struct local_port_range --
```

67.3 `net_device` struct fast path usage breakdown

```
Type Name fastpath_tx_access fastpath_rx_access Comments ..struct ..net_device
char name[16] -- struct_netdev_name_node* name_node struct_dev_ifalias*
ifalias unsigned_long mem_end unsigned_long mem_start unsigned_long
base_addr unsigned_long state struct_list_head dev_list struct_list_head napi_list
struct_list_head unreg_list struct_list_head close_list struct_list_head ptype_all
read_mostly - dev_nit_active(tx) struct_list_head ptype_specific read_mostly
deliver_ptype_list_skb/_netif_receive_skb_core(rx) struct adj_list unsigned_int flags read_mostly
read_mostly _dev_queue_xmit,_dev_xmit_skb,ip6_output,_ip6_finish_output(tx);ip6_rcv_core(rx)
xdp_features_t xdp_features unsigned_long long priv_flags read_mostly - _dev_queue_xmit(tx)
struct_net_device_ops* netdev_ops read_mostly - netdev_core_pick_tx,netdev_start_xmit(tx)
struct_xdp_metadata_ops* xdp_metadata_ops int ifindex - read_mostly ip6_rcv_core
unsigned_short gflags unsigned_short hard_header_len read_mostly read_mostly
ip6_xmit(tx);gro_list_prepare(rx) unsigned_int mtu read_mostly - ip_finish_output2 un-
signed_short needed_headroom read_mostly - LL_RESERVED_SPACE/ip_finish_output2
unsigned_short needed_tailroom netdev_features_t features read_mostly read_mostly
HARD_TX_LOCK,netif_skb_features,sk_setup_caps(tx);netif_elide_gro(rx) netdev_features_t
hw_features netdev_features_t wanted_features netdev_features_t vlan_features net-
dev_features_t hw_enc_features - - netif_skb_features netdev_features_t mpls_features net-
dev_features_t gso_partial_features read_mostly gso_features_check unsigned_int min_mtu
unsigned_int max_mtu unsigned_short type unsigned_char min_header_len unsigned_char
name_assign_type int group struct_net_device_stats stats struct_net_device_core_stats*
core_stats atomic_t carrier_up_count atomic_t carrier_down_count struct_iw_handler_def*
wireless_handlers struct_iw_public_data* wireless_data struct_ethtool_ops* ethtool_ops
struct_l3mdev_ops* l3mdev_ops struct_ndisc_ops* ndisc_ops struct_xfrmdev_ops* xfr-
mdev_ops struct_tlsdev_ops* tlsdev_ops struct_header_ops* header_ops read_mostly -
ip_finish_output2,ip6_finish_output2(tx) unsigned_char operstate unsigned_char link_mode
unsigned_char if_port unsigned_char dma unsigned_char perm_addr[32] unsigned_char
```

```

addr_assign_type unsigned_char addr_len unsigned_char upper_level unsigned_char
lower_level unsigned_short neigh_priv_len unsigned_short padded unsigned_short dev_id
unsigned_short dev_port spinlock_t addr_list_lock int irq struct_netdev_hw_addr_list uc
struct_netdev_hw_addr_list mc struct_netdev_hw_addr_list dev_addrs struct_kset* queues_kset
struct_list_head unlink_list unsigned_int promiscuity unsigned_int allmulti bool uc_promisc
unsigned_char nested_level struct_in_device* ip_ptr read_mostly read_mostly __in_dev_get
struct_inet6_dev* ip6_ptr read_mostly read_mostly __in6_dev_get struct_vlan_info* vlan_info
struct_dsa_port* dsa_ptr struct_tipc_bearer* tipc_ptr void* atalk_ptr void* ax25_ptr
struct_wireless_dev* ieee80211_ptr struct_wpan_dev* ieee802154_ptr struct_mpls_dev*
mpls_ptr struct_mctp_dev* mctp_ptr unsigned_char* dev_addr struct_netdev_queue*
_rx read_mostly - netdev_get_rx_queue(rx) unsigned_int num_rx_queues unsigned_int
real_num_rx_queues - read_mostly get_rps_cpu struct_bpf_prog* xdp_prog - read_mostly
netif_elide_gro() unsigned_long gro_flush_timeout - read_mostly napi_complete_done
int napi_defer_hard_irqs - read_mostly napi_complete_done unsigned_int gro_max_size -
read_mostly skb_gro_receive unsigned_int gro_ipv4_max_size - read_mostly skb_gro_receive
rx_handler_func_t* rx_handler read_mostly - __netif_receive_skb_core void* rx_handler_data
read_mostly - struct_netdev_queue* ingress_queue read_mostly - struct_bpf_mprog_entry
tcx_ingress - read_mostly sch_handle_ingress struct_nf_hook_entries* nf_hooks_ingress
unsigned_char broadcast[32] struct_cpu_rmap* rx_cpu_rmap struct_hlist_node index_hlist
struct_netdev_queue* _tx read_mostly - netdev_get_tx_queue(tx) unsigned_int num_tx_queues
- - unsigned_int real_num_tx_queues read_mostly - skb_tx_hash,netdev_core_pick_tx(tx)
unsigned_int tx_queue_len spinlock_t tx_global_lock struct_xdp_dev_bulk_queue_percpu*
xdp_bulkq struct_xps_dev_maps* xps_maps[2] read_mostly - __netif_set_xps_queue
struct_bpf_mprog_entry tcx_egress read_mostly - sch_handle_egress struct_nf_hook_entries*
nf_hooks_egress read_mostly - struct_hlist_head qdisc_hash[16] struct_timer_list watchdog_timer
int watchdog_timeo u32 proto_down_reason struct_list_head todo_list
int_percpu* pc当地_refcnt refcount_t dev_refcnt struct_ref_tracker_dir refcnt_tracker
struct_list_head link_watch_list enum:8 reg_state bool dismantle enum:16 rtnl_link_state
bool needs_free_netdev void*priv_destructor struct_net_device struct_netpoll_info*
npinfo - read_mostly napi_poll/napi_poll_lock possible_net_t nd_net - read_mostly
(dev_net)napi_busy_loop,tcp_v(4/6)_recv,ip(v6)_recv,ip(6)_input,ip(6)_input_finish void*
ml_priv enum_netdev_ml_priv_type ml_priv_type struct_pc当地_lstats_percpu* lstats
read_mostly dev_lstats_add() struct_pc当地_sw_netstats_percpu* tstats read_mostly
dev_sw_netstats_tx_add() struct_pc当地_dstats_percpu* dstats struct_garp_port* garp_port
struct_mrp_port* mrp_port struct_dm_hw_stat_delta* dm_private struct_device dev -
- struct_attribute_group* sysfs_groups[4] struct_attribute_group* sysfs_rx_queue_group
struct_rtnl_link_ops* rtnl_link_ops unsigned_int gso_max_size read_mostly -
sk_dst_gso_max_size unsigned_int tso_max_size u16 gso_max_segs read_mostly - gso_max_segs
u16 tso_max_segs unsigned_int gso_ipv4_max_size read_mostly - sk_dst_gso_max_size
struct_dcbnl_rtnl_ops* dcbnl_ops s16 num_tc read_mostly - skb_tx_hash struct_netdev_tc_txq
tc_to_txq[16] read_mostly - skb_tx_hash u8 prio_tc_map[16] unsigned_int fcoe_ddp_xid
struct_netprio_map* priomap struct_phy_device* phydev struct_sfp_bus* sfp_bus
struct_lock_class_key* qdisc_tx_busylock bool proto_down unsigned:1 wol_enabled
unsigned:1 threaded - - napi_poll(napi_enable,dev_set_threaded) struct_list_head net_notifier_list
struct_macsec_ops* macsec_ops struct_udp_tunnel_nic_info* udp_tunnel_nic_info
struct_udp_tunnel_nic* udp_tunnel_nic unsigned_int xdp_zc_max_segs struct_bpf_xdp_entity
xdp_state[3] u8 dev_addr_shadow[32] netdevice_tracker linkwatch_dev_tracker
netdevice_tracker watchdog_dev_tracker netdevice_tracker dev_registered_tracker
struct_rtnl_hw_stats64* offload_xstats_l3 struct_devlink_port* devlink_port struct_dpll_pin*
dpll_pin

```

67.4 netns_ipv4 struct fast path usage breakdown

```
Type  Name  fastpath_tx_access  fastpath_rx_access  comment  ..struct  ..netns_ipv4
struct_inet_timewait_death_row      tcp_death_row      struct_udp_table*    udp_table
struct_ctl_table_header* forw_hdr struct_ctl_table_header* frags_hdr struct_ctl_table_header*
ipv4_hdr   struct_ctl_table_header* route_hdr   struct_ctl_table_header* xfrm4_hdr
struct_ipv4_devconf* devconf_all  struct_ipv4_devconf* devconf_dflt  struct_ip_ra_chain
ra_chain  struct_mutex  ra_mutex  struct_fib_rules_ops* rules_ops  struct_fib_table
fib_main   struct_fib_table  fib_default  unsigned_int  fib_rules_require_fldissect  bool
fib_has_custom_rules  bool  fib_has_custom_local_routes  bool  fib_offload_disabled  atomic_t
fib_num_tcclassid_users  struct_hlist_head*  fib_table_hash  struct_sock*  fibnl  struct_sock*
mc_autojoin_sk  struct_inet_peer_base* peers  struct_fqdir*  fqdir  u8  sysctl_icmp_echo_ignore_all
u8  sysctl_icmp_echo_enable_probe  u8  sysctl_icmp_echo_ignore_broadcasts  u8
sysctl_icmp_ignore_bogus_error_responses  u8  sysctl_icmp_errors_use_inbound_ifaddr
int  sysctl_icmp_ratelimit  int  sysctl_icmp_ratemask  u32  ip_rt_min_pmtu  - -  int
ip_rt_mtu_expires  - -  int  ip_rt_min_advms  - -  struct_local_ports  ip_local_ports
- -  u8  sysctl_tcp_ecn  - -  u8  sysctl_tcp_ecnFallback  - -  u8  sysctl_ip_default_ttl  - -
ip4_dst_hoplimit/ip_select_ttl  u8  sysctl_ip_no_pmtu_disc  - -  u8  sysctl_ip_fwd_use_pmtu
read_mostly  - ip_dst_mtu_maybe_forward/ip_skb_dst_mtu  u8  sysctl_ip_fwd_update_priority
- -  ip_forward  u8  sysctl_ip_nonlocal_bind  - -  u8  sysctl_ip_autobind_reuse  - -  u8
sysctl_ip_dynaddr  - -  u8  sysctl_ip_early_demux  -  read_mostly  ip(6)_recv_finish_core
u8  sysctl_raw_l3mdev_accept  - -  u8  sysctl_tcp_early_demux  -  read_mostly
ip(6)_recv_finish_core  u8  sysctl_udp_early_demux  u8  sysctl_nexthop_compat_mode  - -  u8
sysctl_fwmark_reflect  - -  u8  sysctl_tcp_fwmark_accept  - -  u8  sysctl_tcp_l3mdev_accept
- -  u8  sysctl_tcp_mtu_probing  - -  int  sysctl_tcp_mtu_probe_floor  - -  int  sysctl_tcp_base_mss
- -  int  sysctl_tcp_min_snd_mss  read_mostly  -  _tcp_mtu_to_mss(tcp_write_xmit)  int
sysctl_tcp_probe_threshold  - -  tcp_mtu_probe(tcp_write_xmit)  u32  sysctl_tcp_probe_interval
- -  tcp_mtu_check_reprobe(tcp_write_xmit)  int  sysctl_tcp_keepalive_time  - -  int
sysctl_tcp_keepalive_intvl  - -  u8  sysctl_tcp_keepalive_probes  - -  u8  sysctl_tcp_syn_retries
- -  u8  sysctl_tcp_synack_retries  - -  u8  sysctl_tcp_syncookies  - -  generated_on_syn  u8
sysctl_tcp_migrate_req  - -  reuseport  u8  sysctl_tcp_comp_sack_nr  - -  _tcp_ack_snd_check
int  sysctl_tcp_reordering  -  read_mostly  tcp_may_raise_cwnd/tcp_cong_control  u8
sysctl_tcp_retries1  - -  u8  sysctl_tcp_retries2  - -  u8  sysctl_tcp_orphan_retries
- -  u8  sysctl_tcp_tw_reuse  - -  timewait_sock_ops  int  sysctl_tcp_fin_timeout  - -
TCP_LAST_ACK/tcp_rcv_state_process  unsigned_int  sysctl_tcp_notsent_lowat  read_mostly
-  tcp_notsent_lowat/tcp_stream_memory_free  u8  sysctl_tcp_sack  - -  tcp_syn_options  u8
sysctl_tcp_window_scaling  - -  tcp_syn_options,tcp_parse_options  u8  sysctl_tcp_timestamps
u8  sysctl_tcp_early_retrans  read_mostly  -  tcp_schedule_loss_probe(tcp_write_xmit)
u8  sysctl_tcp_recovery  - -  tcp_fastretrans_alert  u8  sysctl_tcp_thin_linear_timeouts
- -  tcp_retrans_timer(on_thin_streams)  u8  sysctl_tcp_slow_start_after_idle  - -
unlikely(tcp_cwnd_validate-network-not-starved)  u8  sysctl_tcp_retransCollapse  - -
u8  sysctl_tcp_stdurg  - -  unlikely(tcp_check_urg)  u8  sysctl_tcp_rfc1337  - -  u8
sysctl_tcp_abort_on_overflow  - -  u8  sysctl_tcp_fack  - -  int  sysctl_tcp_max_reordering
-  tcp_check_sack_reordering  int  sysctl_tcp_adv_win_scale  - -  tcp_init_buffer_space  u8
sysctl_tcp_dsack  - -  partial_packet_or_retrans_in_tcp_data_queue  u8  sysctl_tcp_app_win
-  tcp_win_from_space  u8  sysctl_tcp_frto  - -  tcp_enter_loss  u8  sysctl_tcp_nometrics_save
- -  TCP_LAST_ACK/tcp_update_metrics  u8  sysctl_tcp_no_ssthresh_metrics_save
- -  TCP_LAST_ACK/tcp_(update/init)_metrics  u8  sysctl_tcp_moderate_rcvbuf
read_mostly  read_mostly  tcp_tso_should_defer(tx);tcp_rcv_space_adjust(rx)  u8
sysctl_tcp_tso_win_divisor  read_mostly  -  tcp_tso_should_defer(tcp_write_xmit)  u8
sysctl_tcp_workaround_signed_windows  - -  tcp_select_window  int  sysctl_tcp_limit_output_bytes
```

```

read_mostly - tcp_small_queue_check(tcp_write_xmit) int sysctl_tcp_challenge_ack_limit --
int sysctl_tcp_min_rtt_wlen read_mostly - tcp_ack_update_rtt u8 sysctl_tcp_min_tso_segs
-- unlikely(icsk_ca_ops-written) u8 sysctl_tcp_tso_rtt_log read_mostly - tcp_tso_autosize
u8 sysctl_tcp_autocorking read_mostly - tcp_push/tcp_should_autocork u8
sysctl_tcp_reflect_tos - - - tcp_v(4/6)_send_synack int sysctl_tcp_invalid_ratelimit
- - int sysctl_tcp_pacing_ss_ratio - - default_cong_cont(tcp_update_pacing_rate)
int sysctl_tcp_pacing_ca_ratio - - default_cong_cont(tcp_update_pacing_rate) int
sysctl_tcp_wmem[3] read_mostly - tcp_wmem_schedule(sendmsg/sendpage) int
sysctl_tcp_rmem[3] - read_mostly _tcp_grow_window(tx),tcp_rcv_space_adjust(rx) un-
signed_int sysctl_tcp_child_ehash_entries unsigned_long sysctl_tcp_comp_sack_delay_ns -
- _tcp_ack_snd_check unsigned_long sysctl_tcp_comp_sack_slack_ns - - _tcp_ack_snd_check
int sysctl_max_syn_backlog - - int sysctl_tcp_fastopen - - struct_tcp_congestion_ops
tcp_congestion_control - - init_cc struct_tcp_fastopen_context tcp_fastopen_ctx - -
unsigned_int sysctl_tcp_fastopen_blackhole_timeout - - atomic_t tfo_active_disable_times
- - unsigned_long tfo_active_disable_stamp - - u32 tcp_challenge_timestamp - - u32
tcp_challenge_count - - u8 sysctl_tcp_plb_enabled - - u8 sysctl_tcp_plb_idle_rehash_rounds
- - u8 sysctl_tcp_plb_rehash_rounds - - u8 sysctl_tcp_plb_suspend_rto_sec - -
int sysctl_tcp_plbCong_thresh - - int sysctl_udp_wmem_min int sysctl_udp_rmem_min u8
sysctl_fib_notify_on_flag_change u8 sysctl_udp_l3mdev_accept u8 sysctl_igmp_llm_reports
int sysctl_igmp_max_memberships int sysctl_igmp_max_msf int sysctl_igmp_qrv
struct_ping_group_range ping_group_range atomic_t dev_addr_genid unsigned_int
sysctl_udp_child_hash_entries unsigned_long* sysctl_local_reserved_ports int
sysctl_ip_prot_sock struct_mr_table* mrt struct_list_head mr_tables struct_fib_rules_ops*
mr_rules_ops u32 sysctl_fib_multipath_hash_fields u8 sysctl_fib_multipath_use_neigh
u8 sysctl_fib_multipath_hash_policy struct_fib_notifier_ops* notifier_ops unsigned_int
fib_seq struct_fib_notifier_ops* ipmr_notifier_ops unsigned_int ipmr_seq atomic_t rt_genid
siphash_key_t ip_id_key

```

67.5 netns_ipv4 enum fast path usage breakdown

Type	Name	fastpath_tx_access	fastpath_rx_access	comment ..enum	unsigned_long
LINUX_MIB_TCPKEEPALIVE	write_mostly	-	tcp_keepalive_timer	unsigned_long	
LINUX_MIB_DELAYEDACKS	write_mostly	-	tcp_delack_timer_handler,tcp_delack_timer		
unsigned_long	LINUX_MIB_DELAYEDACKLOCKED		write_mostly	-	
tcp_delack_timer_handler,tcp_delack_timer	unsigned_long	LINUX_MIB_TCPAUTOCORKING			
write_mostly	-tcp_push,tcp_sendmsg	locked	unsigned_long	LINUX_MIB_TCPCFROMZEROWINDOWAII	
write_mostly	-	tcp_select_window,tcp_transmit-skb		unsigned_long	
LINUX_MIB_TCPTOZEROWINDOWADV	write_mostly	-	tcp_select_window,tcp_transmit-		
skb	unsigned_long	LINUX_MIB_TCPWANTZEROWINDOWADV	write_mostly	-	
tcp_select_window,tcp_transmit-skb	unsigned_long	LINUX_MIB_TCPORIGDATASENT			
write_mostly	-	tcp_write_xmit	unsigned_long	LINUX_MIB_TCPHPHITS	-
write_mostly	top_rcv_established,tcp_v4_do_rcv,tcp_v6_do_rcv		unsigned_long		
LINUX_MIB_TCPRCVCOALESCE	write_mostly	tcp_try_coalesce,tcp_queue_rcv,tcp_rcv_established			
unsigned_long	LINUX_MIB_TCPPUREACKS	- write_mostly	tcp_ack,tcp_rcv_established		
unsigned_long	LINUX_MIB_TCPHPACKS	- write_mostly	tcp_ack,tcp_rcv_established		
unsigned_long	LINUX_MIB_TCPDELIVERED	- write_mostly	tcp_newly_delivered,tcp_ack,tcp_rcv_establi		
unsigned_long	LINUX_MIB_SYNCOOKIESSENT	unsigned_long	LINUX_MIB_SYNCOOKIESRECV		
unsigned_long	LINUX_MIB_SYNCOOKIESFAILED	unsigned_long	LINUX_MIB_EMBRYONICRSTS		
unsigned_long	LINUX_MIB_PRUNECALLED	unsigned_long	LINUX_MIB_RCVPRUNED		
unsigned_long	LINUX_MIB_OFOPRUNED	unsigned_long	LINUX_MIB_OUTOFWINDOWICMPS		


```

unsigned long          LINUX_MIB_TCPACKSKIPPEDFINWAIT2           unsigned long
LINUX_MIB_TCPACKSKIPPEDTIMEWAIT unsigned long LINUX_MIB_TCPACKSKIPPEDCHALLENGE
unsigned long          LINUX_MIB_TCPWINPROBE      unsigned long LINUX_MIB_TCPMTUPFAIL
unsigned long          LINUX_MIB_TCPMTUPSUCCESS unsigned long LINUX_MIB_TCPDELIVEREDCE
unsigned long          LINUX_MIB_TCPACKCOMPRESSED        unsigned long
LINUX_MIB_TCPZEROWINDOWDROP   unsigned long LINUX_MIB_TCPRCVQDROP
unsigned long          LINUX_MIB_TCPWQUEUETOOBIG unsigned long LINUX_MIB_TCPFASTOPENPASSIVEA
unsigned long          LINUX_MIB_TCPTIMEOUTREHASH        unsigned long
LINUX_MIB_TCPCDUPLICATEDATAREHASH unsigned long LINUX_MIB_TCPSACKRECVSEGS
unsigned long          LINUX_MIB_TCPSACKIGNOREDDUBIOUS       unsigned long
LINUX_MIB_TCPCMIGRATEREQSUCCESS unsigned long LINUX_MIB_TCPCMIGRATEREQFAILURE
unsigned long          __LINUX_MIB_MAX

```

67.6 tcp_sock struct fast path usage breakdown

Type	Name	fastpath_tx_access	fastpath_rx_access	Comments	..struct ..tcp_sock
struct_inet_connection_sock	inet_conn	u16	tcp_header_len	read_mostly	
read_mostly		tcp_bound_to_half_wnd,tcp_current_mss(tx);tcp_rcv_established(rx)			
u16	gso_segs	read_mostly	-	tcp_xmit_size_goal _be32 pred_flags	read_write
read_mostly		tcp_select_window(tx);tcp_rcv_established(rx)	u64	bytes_received	-
read_write	tcp_rcv_nxt_update(rx)	u32	segs_in	-	read_write
tcp_v6_rcv(rx)		u32	read_write	tcp_v6_rcv(rx)	u32
data_segs_in	-	read_write	tcp_v6_rcv(rx)	u32	rcv_nxt
read_mostly		rcv_nxt	read_mostly	read_write	
tcp_cleanup_rbuf,tcp_send_ack,tcp_inq_hint,tcp_transmit_skb,tcp_receive_window(tx);tcp_v6_do_rcv,					
u32 copied_seq	-	read_mostly	tcp_cleanup_rbuf,tcp_rcv_space_adjust,tcp_inq_hint	u32 rcv_wup	
-	tcp_cleanup_rbuf,tcp_receive_window,tcp_receive_established	u32	tcp_transmit_skb,tcp_event_new_data_sent(wri	u32 snd_nxt	
read_write	read_mostly	tcp_rate_check_app_limited,			
u32 segs_out	read_write	-	tcp_transmit_skb	u32 data_segs_out	
read_write	-	tcp_transmit_skb,tcp_update_skb_after_send	u64 bytes_sent		
read_write	-	tcp_transmit_skb	u64 bytes_acked	-	read_write
tcp_snd_una_update/tcp_ack	u32 dsack_dups	u32 snd_una	read_mostly	read_write	
tcp_wnd_end,tcp_urg_mode,tcp_minshall_check,tcp_cwnd_validate(tx);tcp_ack,tcp_may_update_wind					
u32 snd_sml	read_write	-	tcp_minshall_check,tcp_minshall_update		
u32 rcv_tstamp	-	read_mostly	tcp_ack	u32 lsndtime	read_write
tcp_slow_start_after_idle_check,tcp_event_data_sent				u32 last_oow_ack_time	
u32 compressed_ack_rcv_nxt	u32 tsoffset	read_mostly	read_mostly		
tcp_established_options(tx);tcp_fast_parse_options(rx)		struct_list_head	tsq_node	-	-
struct_list_head	tsorted_sent_queue	read_write	-	tcp_update_skb_after_send	u32
snd_wl1	-	read_mostly	tcp_update_skb_after_send	u32 max_window	read_mostly
tcp_wnd_end,tcp_tso_should_defer(tx);tcp_fast_path_on(rx)	u32 mss_cache	read_mostly	read_mostly	-	-
-	tcp_bound_to_half_wnd,forced_push	u32 mss_cache	read_mostly	read_mostly	
tcp_rate_check_app_limited,tcp_current_mss,tcp_sync_mss,tcp_sndbuf_expand,tcp_tso_should_defer					
u32 window_clamp	read_mostly	read_write	tcp_rcv_space_adjust,_tcp_select_window		
u32 rcv_ssthresh	read_mostly	-	tcp_select_window	u8 scaling_ratio	read_mostly
read_mostly	tcp_win_from_space	struct	tcp_rack	u16 advmss	-
tcp_rcv_space_adjust	u8 compressed_ack	u8:2 dup_ack_counter	u8:1 tlp_retrans		
u8:1 tcp_usec_ts	read_mostly	read_mostly	u32 chrono_start	read_write	-
tcp_chrono_start/stop(tcp_write_xmit,tcp_cwnd_validate,tcp_send_syn_data)				u32[3]	
chrono_stat	read_write	-	tcp_chrono_start/stop(tcp_write_xmit,tcp_cwnd_validate,tcp_send_syn_data)		
u8:2 chrono_type	read_write	-	tcp_chrono_start/stop(tcp_write_xmit,tcp_cwnd_validate,tcp_send_syn_data)		
u8:1 rate_app_limited	-	read_write	tcp_rate_gen	u8:1 fastopen_connect	u8:1

```

fastopen_no_cookie u8:1 is_sack_reneg - read_mostly tcp_skb_entail,tcp_ack u8:2
fastopen_client_fail u8:4 nonagle read_write - tcp_skb_entail,tcp_push_pending_frames
u8:1 thin_lto u8:1 recvmsg_inq u8:1 repair read_mostly - tcp_write_xmit u8:1
frto u8 repair_queue - - u8:2 save_syn u8:1 syn_data u8:1 syn_fastopen
u8:1 syn_fastopen_exp u8:1 syn_fastopen_ch u8:1 syn_data_acked u8:1
is_cwnd_limited read_mostly - tcp_cwnd_validate,tcp_is_cwnd_limited u32 tlp_high_seq
- read_mostly tcp_ack u32 tcp_tx_delay u64 tcp_wstamp_ns read_write -
tcp_pacing_check,tcp_tso_should_defer,tcp_update_skb_after_send u64 tcp_clock_cache
read_write read_write tcp_mstamp_refresh(tcp_write_xmit/tcp_rcv_space_adjust),_tcp_transmit_skb,
u64 tcp_mstamp read_write read_write tcp_mstamp_refresh(tcp_write_xmit/tcp_rcv_space_adjust)(tx),
u32 srtt_us read_mostly read_write tcp_tso_should_defer(tx);tcp_update_pacing_rate,_tcp_set_rto,tcp_
u32 mdev_us read_write - tcp_rtt_estimator u32 mdev_max_us u32 rttvar_us - read_mostly
_tcp_set_rto u32 rtt_seq read_write tcp_rtt_estimator struct_minmax rtt_min - read_mostly
tcp_min_rtt/tcp_rate_gen,tcp_min_rttcp_update_rtt_min u32 packets_out read_write
read_write tcp_packets_in_flight(tx/rx);tcp_slow_start_after_idle_check,tcp_nagle_check,tcp_rate_skb
u32 retrans_out - read_mostly tcp_packets_in_flight,tcp_rate_check_app_limited
u32 max_packets_out - read_write tcp_cwnd_validate u32 cwnd_usage_seq -
read_write tcp_cwnd_validate u16 urg_data - read_mostly tcp_fast_path_check
u8 ecn_flags read_write - tcp_ecn_send u8 keepalive_probes u32 reordering
read_mostly - tcp_sndbuf_expand u32 reord_seen u32 snd_up read_write
read_mostly tcp_mark_urg,tcp_urg_mode,_tcp_transmit_skb(tx);tcp_clean_rtx_queue(rx)
struct_tcp_options_received rx_opt read_mostly read_write tcp_established_options(tx);tcp_fast_path_
u32 snd_ssthresh - read_mostly tcp_update_pacing_rate u32 snd_cwnd read_mostly
read_mostly tcp_snd_cwnd,tcp_rate_check_app_limited,tcp_tso_should_defer(tx);tcp_update_pacing_
u32 snd_cwnd_cnt u32 snd_cwnd_clamp u32 snd_cwnd_used u32 snd_cwnd_stamp
u32 prior_cwnd u32 prr_delivered u32 prr_out read_mostly read_mostly
tcp_rate_skb_sent,tcp_newly_delivered(tx);tcp_ack,tcp_rate_gen,tcp_clean_rtx_queue(rx)
u32 delivered read_mostly read_write tcp_rate_skb_sent, tcp_newly_delivered(tx);tcp_ack,
tcp_rate_gen, tcp_clean_rtx_queue (rx) u32 delivered_ce read_mostly read_write
tcp_rate_skb_sent(tx);tcp_rate_gen(rx) u32 lost - read_mostly tcp_ack u32 app_limited
read_write read_mostly tcp_rate_check_app_limited,tcp_rate_skb_sent(tx);tcp_rate_gen(rx)
u64 first_tx_mstamp read_write - tcp_rate_skb_sent u64 delivered_mstamp read_write
- tcp_rate_skb_sent u32 rate_delivered - read_mostly tcp_rate_gen u32 rate_interval_us
- read_mostly rate_delivered,rate_app_limited u32 rcv wnd read_write read_mostly
tcp_select_window,tcp_receive_window,tcp_fast_path_check u32 write_seq read_write -
tcp_rate_check_app_limited,tcp_write_queue_empty,tcp_skb_entail,forced_push,tcp_mark_push
u32 notsent_lowat read_mostly - tcp_stream_memory_free u32 pushed_seq
read_write - tcp_mark_push,forced_push u32 lost_out read_mostly read_mostly
tcp_left_out(tx);tcp_packets_in_flight(tx/rx);tcp_rate_check_app_limited(rx) u32 sacked_out
read_mostly read_mostly tcp_left_out(tx);tcp_packets_in_flight(tx/rx);tcp_clean_rtx_queue(rx)
struct_hrtimer pacing_timer struct_hrtimer compressed_ack_timer struct_sk_buff*
lost_skb_hint read_mostly tcp_clean_rtx_queue struct_sk_buff* retransmit_skb_hint
read_mostly - tcp_clean_rtx_queue struct_rb_root out_of_order_queue - read_mostly
tcp_data_queue,tcp_fast_path_check struct_sk_buff* ooo_last_skb struct_tcp_sack_block[1] du-
plicate_sack struct_tcp_sack_block[4] selective_acks struct_tcp_sack_block[4] recv_sack_cache
struct_sk_buff* highest_sack read_write - tcp_event_new_data_sent int lost_cnt_hint u32
prior_ssthresh u32 high_seq u32 retrans_stamp u32 undo_marker int undo_retrans u64
bytes_retrans u32 total_retrans u32 rto_stamp u16 total_rto u16 total_rto_recoveries u32 to-
tal_rto_time u32 urg_seq -- unsigned_int keepalive_time unsigned_int keepalive_intvl int linger2
u8 bpf_sock_ops_cb_flags u8:1 bpf_chg_cc_inprogress u16 timeout_rehash u32 rcv_oopack
u32 rcv_rtt_last_tsecr struct rcv_rtt_est - read_write tcp_rcv_space_adjust,tcp_rcv_established

```

```
struct rcvq_space - read_write tcp_rcv_space_adjust struct mtu_probe u32 plb_rehash
u32 mtu_info bool is_mptcp bool smc_hs_congested bool syn_smc struct_tcp_sock_af_ops*
af_specific struct_tcp_md5sig_info* md5sig_info struct_tcp_fastopen_request* fastopen_req
struct_request_sock* fastopen_rsk struct_saved_syn* saved_syn
```


NETCONSOLE

started by Ingo Molnar <mingo@redhat.com>, 2001.09.17

2.6 port and netpoll api by Matt Mackall <mpm@selenic.com>, Sep 9 2003

IPv6 support by Cong Wang <xiyou.wangcong@gmail.com>, Jan 1 2013

Extended console support by Tejun Heo <tj@kernel.org>, May 1 2015

Release prepend support by Breno Leitao <leitao@debian.org>, Jul 7 2023

Please send bug reports to Matt Mackall <mpm@selenic.com> Satyam Sharma <satyam.sharma@gmail.com>, and Cong Wang <xiyou.wangcong@gmail.com>

68.1 Introduction:

This module logs kernel printk messages over UDP allowing debugging of problem where disk logging fails and serial consoles are impractical.

It can be used either built-in or as a module. As a built-in, netconsole initializes immediately after NIC cards and will bring up the specified interface as soon as possible. While this doesn't allow capture of early kernel panics, it does capture most of the boot process.

68.2 Sender and receiver configuration:

It takes a string configuration parameter "netconsole" in the following format:

```
netconsole=[+][r][src-port]@[src-ip]/[<dev>],[tgt-port]@<tgt-ip>/[tgt-macaddr]  
where  
+           if present, enable extended console support  
r           if present, prepend kernel version (release) to the message  
message  
src-port    source for UDP packets (defaults to 6665)  
src-ip     source IP to use (interface address)  
dev        network interface (eth0)  
tgt-port    port for logging agent (6666)  
tgt-ip     IP address for logging agent  
tgt-macaddr ethernet MAC address for logging agent (broadcast)
```

Examples:

```
linux netconsole=4444@10.0.0.1/eth1,9353@10.0.0.2/12:34:56:78:9a:bc
```

or:

```
insmod netconsole netconsole=@/,@10.0.0.2/
```

or using IPv6:

```
insmod netconsole netconsole=@/,@fd00:1:2:3::1/
```

It also supports logging to multiple remote agents by specifying parameters for the multiple agents separated by semicolons and the complete string enclosed in "quotes", thusly:

```
modprobe netconsole netconsole="@/,@10.0.0.2/;@/eth1,6892@10.0.0.3/"
```

Built-in netconsole starts immediately after the TCP stack is initialized and attempts to bring up the supplied dev at the supplied address.

The remote host has several options to receive the kernel messages, for example:

1) syslogd

2) netcat

On distributions using a BSD-based netcat version (e.g. Fedora, openSUSE and Ubuntu) the listening port must be specified without the -p switch:

```
nc -u -l <port> / 'nc -u -l <port>
```

or::

```
netcat -u -l <port> / 'netcat -u -l <port>
```

3) socat

```
socat udp-recv:<port> -
```

68.3 Dynamic reconfiguration:

Dynamic reconfigurability is a useful addition to netconsole that enables remote logging targets to be dynamically added, removed, or have their parameters reconfigured at runtime from a configfs-based userspace interface.

To include this feature, select CONFIG_NETCONSOLE_DYNAMIC when building the netconsole module (or kernel, if netconsole is built-in).

Some examples follow (where configfs is mounted at the /sys/kernel/config mountpoint).

To add a remote logging target (target names can be arbitrary):

```
cd /sys/kernel/config/netconsole/
mkdir target1
```

Note that newly created targets have default parameter values (as mentioned above) and are disabled by default -- they must first be enabled by writing "1" to the "enabled" attribute (usually after setting parameters accordingly) as described below.

To remove a target:

```
rmdir /sys/kernel/config/netconsole/othertarget/
```

The interface exposes these parameters of a netconsole target to userspace:

enabled	Is this target currently enabled?	(read-write)
extended	Extended mode enabled	(read-write)
release	Prepend kernel release to message	(read-write)
dev_name	Local network interface name	(read-write)
local_port	Source UDP port to use	(read-write)
remote_port	Remote agent's UDP port	(read-write)
local_ip	Source IP address to use	(read-write)
remote_ip	Remote agent's IP address	(read-write)
local_mac	Local interface's MAC address	(read-only)
remote_mac	Remote agent's MAC address	(read-write)

The "enabled" attribute is also used to control whether the parameters of a target can be updated or not -- you can modify the parameters of only disabled targets (i.e. if "enabled" is 0).

To update a target's parameters:

```
cat enabled                                # check if enabled is 1
echo 0 > enabled                            # disable the target (if required)
echo eth2 > dev_name                         # set local interface
echo 10.0.0.4 > remote_ip                   # update some parameter
echo cb:a9:87:65:43:21 > remote_mac        # update more parameters
echo 1 > enabled                             # enable target again
```

You can also update the local interface dynamically. This is especially useful if you want to use interfaces that have newly come up (and may not have existed when netconsole was loaded / initialized).

Netconsole targets defined at boot time (or module load time) with the *netconsole=* param are assigned the name *cmdline<index>*. For example, the first target in the parameter is named *cmdline0*. You can control and modify these targets by creating configfs directories with the matching name.

Let's suppose you have two netconsole targets defined at boot time:

```
netconsole=4444@10.0.0.1/eth1,9353@10.0.0.2/12:34:56:78:9a:bc;4444@10.0.0.1/
→eth1,9353@10.0.0.3/12:34:56:78:9a:bc
```

You can modify these targets in runtime by creating the following targets:

```
mkdir cmdline0
cat cmdline0/remote_ip
10.0.0.2
```

```
mkdir cmdline1
cat cmdline1/remote_ip
10.0.0.3
```

68.4 Extended console:

If '+' is prefixed to the configuration line or "extended" config file is set to 1, extended console support is enabled. An example boot param follows:

```
linux netconsole=+4444@10.0.0.1/eth1,9353@10.0.0.2/12:34:56:78:9a:bc
```

Log messages are transmitted with extended metadata header in the following format which is the same as /dev/kmsg:

```
<level>,<seqnum>,<timestamp>,<contflag>;<message text>
```

If 'r' (release) feature is enabled, the kernel release version is prepended to the start of the message. Example:

```
6.4.0,6,444,501151268,-;netconsole: network logging started
```

Non printable characters in <message text> are escaped using "xff" notation. If the message contains optional dictionary, verbatim newline is used as the delimiter.

If a message doesn't fit in certain number of bytes (currently 1000), the message is split into multiple fragments by netconsole. These fragments are transmitted with "ncfrag" header field added:

```
ncfrag=<byte-offset>/<total-bytes>
```

For example, assuming a lot smaller chunk size, a message "the first chunk, the 2nd chunk." may be split as follows:

```
6,416,1758426,-,ncfrag=0/31;the first chunk,
6,416,1758426,-,ncfrag=16/31; the 2nd chunk.
```

68.5 Miscellaneous notes:

Warning: the default target ethernet setting uses the broadcast ethernet address to send packets, which can cause increased load on other systems on the same ethernet segment.

Tip: some LAN switches may be configured to suppress ethernet broadcasts so it is advised to explicitly specify the remote agents' MAC addresses from the config parameters passed to netconsole.

Tip: to find out the MAC address of, say, 10.0.0.2, you may try using:

```
ping -c 1 10.0.0.2 ; /sbin/arp -n | grep 10.0.0.2
```

Tip: in case the remote logging agent is on a separate LAN subnet than the sender, it is suggested to try specifying the MAC address of the default gateway (you may use /sbin/route -n to find it out) as the remote MAC address instead.

Note: the network device (eth1 in the above case) can run any kind of other network traffic, netconsole is not intrusive. Netconsole might cause slight delays in other traffic if the volume of kernel messages is high, but should have no other impact.

Note: if you find that the remote logging agent is not receiving or printing all messages from the sender, it is likely that you have set the "console_loglevel" parameter (on the sender) to only send high priority messages to the console. You can change this at runtime using:

```
dmesg -n 8
```

or by specifying "debug" on the kernel command line at boot, to send all kernel messages to the console. A specific value for this parameter can also be set using the "loglevel" kernel boot option. See the dmesg(8) man page and Documentation/admin-guide/kernel-parameters.rst for details.

Netconsole was designed to be as instantaneous as possible, to enable the logging of even the most critical kernel bugs. It works from IRQ contexts as well, and does not enable interrupts while sending packets. Due to these unique needs, configuration cannot be more automatic, and some fundamental limitations will remain: only IP networks, UDP packets and ethernet devices are supported.

NETDEV FEATURES MESS AND HOW TO GET OUT FROM IT ALIVE

Author:

Michał Mirosław <mirq-linux@rere.qmwm.pl>

69.1 Part I: Feature sets

Long gone are the days when a network card would just take and give packets verbatim. Today's devices add multiple features and bugs (read: offloads) that relieve an OS of various tasks like generating and checking checksums, splitting packets, classifying them. Those capabilities and their state are commonly referred to as netdev features in Linux kernel world.

There are currently three sets of features relevant to the driver, and one used internally by network core:

1. netdev->hw_features set contains features whose state may possibly be changed (enabled or disabled) for a particular device by user's request. This set should be initialized in ndo_init callback and not changed later.
2. netdev->features set contains features which are currently enabled for a device. This should be changed only by network core or in error paths of ndo_set_features callback.
3. netdev->vlan_features set contains features whose state is inherited by child VLAN devices (limits netdev->features set). This is currently used for all VLAN devices whether tags are stripped or inserted in hardware or software.
4. netdev->wanted_features set contains feature set requested by user. This set is filtered by ndo_fix_features callback whenever it or some device-specific conditions change. This set is internal to networking core and should not be referenced in drivers.

69.2 Part II: Controlling enabled features

When current feature set (netdev->features) is to be changed, new set is calculated and filtered by calling ndo_fix_features callback and netdev_fix_features(). If the resulting set differs from current set, it is passed to ndo_set_features callback and (if the callback returns success) replaces value stored in netdev->features. NETDEV_FEAT_CHANGE notification is issued after that whenever current set might have changed.

The following events trigger recalculation:

1. device's registration, after ndo_init returned success
2. user requested changes in features state

3. `netdev_update_features()` is called

`ndo_*_features` callbacks are called with `rtnl_lock` held. Missing callbacks are treated as always returning success.

A driver that wants to trigger recalculation must do so by calling `netdev_update_features()` while holding `rtnl_lock`. This should not be done from `ndo_*_features` callbacks. `netdev->features` should not be modified by driver except by means of `ndo_fix_features` callback.

69.3 Part III: Implementation hints

- `ndo_fix_features`:

All dependencies between features should be resolved here. The resulting set can be reduced further by networking core imposed limitations (as coded in `netdev_fix_features()`). For this reason it is safer to disable a feature when its dependencies are not met instead of forcing the dependency on.

This callback should not modify hardware nor driver state (should be stateless). It can be called multiple times between successive `ndo_set_features` calls.

Callback must not alter features contained in `NETIF_F_SOFT_FEATURES` or `NETIF_F_NEVER_CHANGE` sets. The exception is `NETIF_F_VLAN_CHALLENGED` but care must be taken as the change won't affect already configured VLANs.

- `ndo_set_features`:

Hardware should be reconfigured to match passed feature set. The set should not be altered unless some error condition happens that can't be reliably detected in `ndo_fix_features`. In this case, the callback should update `netdev->features` to match resulting hardware state. Errors returned are not (and cannot be) propagated anywhere except dmesg. (Note: successful return is zero, >0 means silent error.)

69.4 Part IV: Features

For current list of features, see `include/linux/netdev_features.h`. This section describes semantics of some of them.

- Transmit checksumming

For complete description, see comments near the top of `include/linux/skbuff.h`.

Note: `NETIF_F_HW_CSUM` is a superset of `NETIF_F_IP_CSUM` + `NETIF_F_IPV6_CSUM`. It means that device can fill TCP/UDP-like checksum anywhere in the packets whatever headers there might be.

- Transmit TCP segmentation offload

`NETIF_F_TSO_ECN` means that hardware can properly split packets with CWR bit set, be it TCPv4 (when `NETIF_F_TSO` is enabled) or TCPv6 (`NETIF_F_TSO6`).

- Transmit UDP segmentation offload

`NETIF_F_GSO_UDP_L4` accepts a single UDP header with a payload that exceeds `gso_size`. On segmentation, it segments the payload on `gso_size` boundaries and replicates the network and UDP headers (fixing up the last one if less than `gso_size`).

- Transmit DMA from high memory

On platforms where this is relevant, NETIF_F_HIGHDMA signals that ndo_start_xmit can handle skbs with frags in high memory.

- Transmit scatter-gather

Those features say that ndo_start_xmit can handle fragmented skbs: NETIF_F_SG --- paged skbs (skb_shinfo()->frags), NETIF_F_FRAGLIST --- chained skbs (skb->next/prev list).

- Software features

Features contained in NETIF_F_SOFT_FEATURES are features of networking stack. Driver should not change behaviour based on them.

- LLTX driver (deprecated for hardware drivers)

NETIF_F_LLTX is meant to be used by drivers that don't need locking at all, e.g. software tunnels.

This is also used in a few legacy drivers that implement their own locking, don't use it for new (hardware) drivers.

- netns-local device

NETIF_F_NETNS_LOCAL is set for devices that are not allowed to move between network namespaces (e.g. loopback).

Don't use it in drivers.

- VLAN challenged

NETIF_F_VLAN_CHALLENGED should be set for devices which can't cope with VLAN headers. Some drivers set this because the cards can't handle the bigger MTU. [FIXME: Those cases could be fixed in VLAN code by allowing only reduced-MTU VLANs. This may be not useful, though.]

- rx-fcs

This requests that the NIC append the Ethernet Frame Checksum (FCS) to the end of the skb data. This allows sniffers and other tools to read the CRC recorded by the NIC on receipt of the packet.

- rx-all

This requests that the NIC receive all possible frames, including errored frames (such as bad FCS, etc). This can be helpful when sniffing a link with bad packets on it. Some NICs may receive more packets if also put into normal PROMISC mode.

- rx-gro-hw

This requests that the NIC enables Hardware GRO (generic receive offload). Hardware GRO is basically the exact reverse of TSO, and is generally stricter than Hardware LRO. A packet stream merged by Hardware GRO must be re-segmentable by GSO or TSO back to the exact original packet stream. Hardware GRO is dependent on RXCSUM since every packet successfully merged by hardware must also have the checksum verified by hardware.

- hsr-tag-ins-offload

This should be set for devices which insert an HSR (High-availability Seamless Redundancy) or PRP (Parallel Redundancy Protocol) tag automatically.

- `hsr-tag-rm-offload`

This should be set for devices which remove HSR (High-availability Seamless Redundancy) or PRP (Parallel Redundancy Protocol) tags automatically.

- `hsr-fwd-offload`

This should be set for devices which forward HSR (High-availability Seamless Redundancy) frames from one port to another in hardware.

- `hsr-dup-offload`

This should be set for devices which duplicate outgoing HSR (High-availability Seamless Redundancy) or PRP (Parallel Redundancy Protocol) tags automatically in hardware.

NETWORK DEVICES, THE KERNEL, AND YOU!

70.1 Introduction

The following is a random collection of documentation regarding network devices.

70.2 struct net_device lifetime rules

Network device structures need to persist even after module is unloaded and must be allocated with `alloc_netdev_mqs()` and friends. If device has registered successfully, it will be freed on last use by `free_netdev()`. This is required to handle the pathological case cleanly (example: `rmmmod mydriver </sys/class/net/myeth/mtu`)

`alloc_netdev_mqs()` / `alloc_netdev()` reserve extra space for driver private data which gets freed when the network device is freed. If separately allocated data is attached to the network device (`netdev_priv()`) then it is up to the module exit handler to free that.

There are two groups of APIs for registering `struct net_device`. First group can be used in normal contexts where `rtnl_lock` is not already held: `register_netdev()`, `unregister_netdev()`. Second group can be used when `rtnl_lock` is already held: `register_netdevice()`, `unregister_netdevice()`, `free_netdevice()`.

70.2.1 Simple drivers

Most drivers (especially device drivers) handle lifetime of `struct net_device` in context where `rtnl_lock` is not held (e.g. driver probe and remove paths).

In that case the `struct net_device` registration is done using the `register_netdev()`, and `unregister_netdev()` functions:

```
int probe()
{
    struct my_device_priv *priv;
    int err;

    dev = alloc_netdev_mqs(...);
    if (!dev)
        return -ENOMEM;
    priv = netdev_priv(dev);
```

```

/* ... do all device setup before calling register_netdev() ...
 */

err = register_netdev(dev);
if (err)
    goto err_undo;

/* net_device is visible to the user! */

err_undo:
/* ... undo the device setup ... */
free_netdev(dev);
return err;
}

void remove()
{
    unregister_netdev(dev);
    free_netdev(dev);
}

```

Note that after calling `register_netdev()` the device is visible in the system. Users can open it and start sending / receiving traffic immediately, or run any other callback, so all initialization must be done prior to registration.

`unregister_netdev()` closes the device and waits for all users to be done with it. The memory of `struct net_device` itself may still be referenced by sysfs but all operations on that device will fail.

`free_netdev()` can be called after `unregister_netdev()` returns on when `register_netdev()` failed.

70.2.2 Device management under RTNL

Registering `struct net_device` while in context which already holds the `rtnl_lock` requires extra care. In those scenarios most drivers will want to make use of `struct net_device`'s `needs_free_netdev` and `priv_destructor` members for freeing of state.

Example flow of netdev handling under `rtnl_lock`:

```

static void my_setup(struct net_device *dev)
{
    dev->needs_free_netdev = true;
}

static void my_destructor(struct net_device *dev)
{
    some_obj_destroy(priv->obj);
    some_uninit(priv);
}

int create_link()

```

```
{
    struct my_device_priv *priv;
    int err;

    ASSERT_RTNL();

    dev = alloc_netdev(sizeof(*priv), "net%d", NET_NAME_UNKNOWN, my_setup);
    if (!dev)
        return -ENOMEM;
    priv = netdev_priv(dev);

    /* Implicit constructor */
    err = some_init(priv);
    if (err)
        goto err_free_dev;

    priv->obj = some_obj_create();
    if (!priv->obj) {
        err = -ENOMEM;
        goto err_some_uninit;
    }
    /* End of constructor, set the destructor: */
    dev->priv_destructor = my_destructor;

    err = register_netdevice(dev);
    if (err)
        /* register_netdevice() calls destructor on failure */
        goto err_free_dev;

    /* If anything fails now unregister_netdevice() (or unregister_netdev())
     * will take care of calling my_destructor and free_netdev().
     */
}

return 0;

err_some_uninit:
    some_uninit(priv);
err_free_dev:
    free_netdev(dev);
    return err;
}
```

If `struct net_device.priv_destructor` is set it will be called by the core some time after `unregister_netdevice()`, it will also be called if `register_netdevice()` fails. The callback may be invoked with or without `rtnl_lock` held.

There is no explicit constructor callback, driver "constructs" the private netdev state after allocating it and before registration.

Setting `struct net_device.needs_free_netdev` makes core call `free_netdevice()` automatically after `unregister_netdevice()` when all references to the device are gone. It only takes effect after a successful call to `register_netdevice()` so if `register_netdevice()` fails driver is

responsible for calling `free_netdev()`.

`free_netdev()` is safe to call on error paths right after `unregister_netdevice()` or when `register_netdevice()` fails. Parts of netdev (de)registration process happen after `rtnl_lock` is released, therefore in those cases `free_netdev()` will defer some of the processing until `rtnl_lock` is released.

Devices spawned from struct `rtnl_link_ops` should never free the `struct net_device` directly.

.ndo_init and .ndo_uninit

`.ndo_init` and `.ndo_uninit` callbacks are called during `net_device` registration and de-registration, under `rtnl_lock`. Drivers can use those e.g. when parts of their init process need to run under `rtnl_lock`.

`.ndo_init` runs before device is visible in the system, `.ndo_uninit` runs during de-registering after device is closed but other subsystems may still have outstanding references to the netdevice.

70.3 MTU

Each network device has a Maximum Transfer Unit. The MTU does not include any link layer protocol overhead. Upper layer protocols must not pass a socket buffer (skb) to a device to transmit with more data than the mtu. The MTU does not include link layer header overhead, so for example on Ethernet if the standard MTU is 1500 bytes used, the actual skb will contain up to 1514 bytes because of the Ethernet header. Devices should allow for the 4 byte VLAN header as well.

Segmentation Offload (GSO, TSO) is an exception to this rule. The upper layer protocol may pass a large socket buffer to the device transmit routine, and the device will break that up into separate packets based on the current MTU.

MTU is symmetrical and applies both to receive and transmit. A device must be able to receive at least the maximum size packet allowed by the MTU. A network device may use the MTU as mechanism to size receive buffers, but the device should allow packets with VLAN header. With standard Ethernet mtu of 1500 bytes, the device should allow up to 1518 byte packets (1500 + 14 header + 4 tag). The device may either: drop, truncate, or pass up oversize packets, but dropping oversize packets is preferred.

70.4 struct net_device synchronization rules

ndo_open:

Synchronization: `rtnl_lock()` semaphore. Context: process

ndo_stop:

Synchronization: `rtnl_lock()` semaphore. Context: process Note: `netif_running()` is guaranteed false

ndo_do_ioctl:

Synchronization: `rtnl_lock()` semaphore. Context: process

This is only called by network subsystems internally, not by user space calling ioctl as it was in before linux-5.14.

ndo_siocbond:

Synchronization: rtnl_lock() semaphore. Context: process

Used by the bonding driver for the SIOCBOND family of ioctl commands.

ndo_siocwandeV:

Synchronization: rtnl_lock() semaphore. Context: process

Used by the drivers/net/wan framework to handle the SIOCWANDEV ioctl with the if_settings structure.

ndo_siocdevprivate:

Synchronization: rtnl_lock() semaphore. Context: process

This is used to implement SIOCDEVPRIVATE ioctl helpers. These should not be added to new drivers, so don't use.

ndo_eth_ioctl:

Synchronization: rtnl_lock() semaphore. Context: process

ndo_get_stats:

Synchronization: rtnl_lock() semaphore, dev_base_lock rwlock, or RCU. Context: atomic (can't sleep under rwlock or RCU)

ndo_start_xmit:

Synchronization: __netif_tx_lock spinlock.

When the driver sets NETIF_F_LLTX in dev->features this will be called without holding netif_tx_lock. In this case the driver has to lock by itself when needed. The locking there should also properly protect against set_rx_mode. WARNING: use of NETIF_F_LLTX is deprecated. Don't use it for new drivers.

Context: Process with BHs disabled or BH (timer),

will be called with interrupts disabled by netconsole.

Return codes:

- NETDEV_TX_OK everything ok.
- NETDEV_TX_BUSY Cannot transmit packet, try later Usually a bug, means queue start/stop flow control is broken in the driver. Note: the driver must NOT put the skb in its DMA ring.

ndo_tx_timeout:

Synchronization: netif_tx_lock spinlock; all TX queues frozen. Context: BHs disabled
Notes: [*netif_queue_stopped\(\)*](#) is guaranteed true

ndo_set_rx_mode:

Synchronization: netif_addr_lock spinlock. Context: BHs disabled

70.5 struct napi_struct synchronization rules

napi->poll:

Synchronization:

NAPI_STATE_SCHED bit in napi->state. Device driver's ndo_stop method will invoke `napi_disable()` on all NAPI instances which will do a sleeping poll on the NAPI_STATE_SCHED napi->state bit, waiting for all pending NAPI activity to cease.

Context:

softirq will be called with interrupts disabled by netconsole.

NETFILTER SYSFS VARIABLES

71.1 /proc/sys/net/netfilter/* Variables:

nf_log_all_netns - BOOLEAN

- 0 - disabled (default)
- not 0 - enabled

By default, only init_net namespace can log packets into kernel log with LOG target; this aims to prevent containers from flooding host kernel log. If enabled, this target also works in other network namespaces. This variable is only accessible from init_net.

NETIF MSG LEVEL

The design of the network interface message level setting.

72.1 History

The design of the debugging message interface was guided and constrained by backwards compatibility previous practice. It is useful to understand the history and evolution in order to understand current practice and relate it to older driver source code.

From the beginning of Linux, each network device driver has had a local integer variable that controls the debug message level. The message level ranged from 0 to 7, and monotonically increased in verbosity.

The message level was not precisely defined past level 3, but were always implemented within +1 of the specified level. Drivers tended to shed the more verbose level messages as they matured.

- 0 Minimal messages, only essential information on fatal errors.
- 1 Standard messages, initialization status. No run-time messages
- 2 Special media selection messages, generally timer-driver.
- 3 Interface starts and stops, including normal status messages
- 4 Tx and Rx frame error messages, and abnormal driver operation
- 5 Tx packet queue information, interrupt events.
- 6 Status on each completed Tx packet and received Rx packets
- 7 Initial contents of Tx and Rx packets

Initially this message level variable was uniquely named in each driver e.g. "lance_debug", so that a kernel symbolic debugger could locate and modify the setting. When kernel modules became common, the variables were consistently renamed to "debug" and allowed to be set as a module parameter.

This approach worked well. However there is always a demand for additional features. Over the years the following emerged as reasonable and easily implemented enhancements

- Using an ioctl() call to modify the level.
- Per-interface rather than per-driver message level setting.

- More selective control over the type of messages emitted.

The netif_msg recommendation adds these features with only a minor complexity and code size increase.

The recommendation is the following points

- Retaining the per-driver integer variable "debug" as a module parameter with a default level of '1'.
- Adding a per-interface private variable named "msg_enable". The variable is a bit map rather than a level, and is initialized as:

```
1 << debug
```

Or more precisely:

```
debug < 0 ? 0 : 1 << min(sizeof(int)-1, debug)
```

Messages should changes from:

```
if (debug > 1)
    printk(MSG_DEBUG "%s: ...
```

to:

```
if (np->msg_enable & NETIF_MSG_LINK)
    printk(MSG_DEBUG "%s: ...
```

The set of message levels is named

Old level	Name	Bit position
0	NETIF_MSG_DRV	0x0001
1	NETIF_MSG_PROBE	0x0002
2	NETIF_MSG_LINK	0x0004
2	NETIF_MSG_TIMER	0x0004
3	NETIF_MSG_IFDOWN	0x0008
3	NETIF_MSG_IFUP	0x0008
4	NETIF_MSG_RX_ERR	0x0010
4	NETIF_MSG_TX_ERR	0x0010
5	NETIF_MSG_TX queued	0x0020
5	NETIF_MSG_INTR	0x0020
6	NETIF_MSG_TX_DONE	0x0040
6	NETIF_MSG_RX_STATUS	0x0040
7	NETIF_MSG_PKTDATA	0x0080

RESILIENT NEXT-HOP GROUPS

Resilient groups are a type of next-hop group that is aimed at minimizing disruption in flow routing across changes to the group composition and weights of constituent next hops.

The idea behind resilient hashing groups is best explained in contrast to the legacy multipath next-hop group, which uses the hash-threshold algorithm, described in RFC 2992.

To select a next hop, hash-threshold algorithm first assigns a range of hashes to each next hop in the group, and then selects the next hop by comparing the SKB hash with the individual ranges. When a next hop is removed from the group, the ranges are recomputed, which leads to reassignment of parts of hash space from one next hop to another. RFC 2992 illustrates it thus:

+	-	-	-	-	-	-	-			
	1		2		3		4		5	
+	-	-	-	-	-	-	-	-	-	-
	1		2		4		5			
+	-	-	-	-	-	-	-	-	-	-

Before and after deletion of next hop 3 under the hash-threshold algorithm.

Note how next hop 2 gave up part of the hash space in favor of next hop 1, and 4 in favor of 5. While there will usually be some overlap between the previous and the new distribution, some traffic flows change the next hop that they resolve to.

If a multipath group is used for load-balancing between multiple servers, this hash space reassignment causes an issue that packets from a single flow suddenly end up arriving at a server that does not expect them. This can result in TCP connections being reset.

If a multipath group is used for load-balancing among available paths to the same server, the issue is that different latencies and reordering along the way causes the packets to arrive in the wrong order, resulting in degraded application performance.

To mitigate the above-mentioned flow redirection, resilient next-hop groups insert another layer of indirection between the hash space and its constituent next hops: a hash table. The selection algorithm uses SKB hash to choose a hash table bucket, then reads the next hop that this bucket contains, and forwards traffic there.

This indirection brings an important feature. In the hash-threshold algorithm, the range of hashes associated with a next hop must be continuous. With a hash table, mapping between the hash table buckets and the individual next hops is arbitrary. Therefore when a next hop is deleted the buckets that held it are simply reassigned to other next hops:

```
+---+---+---+---+---+---+---+---+---+---+---+
|1|1|1|1|2|2|2|2|3|3|3|3|4|4|4|4|5|5|5|5|
+---+---+---+---+---+---+---+---+---+---+---+
          V V V V
+---+---+---+---+---+---+---+---+---+---+---+
|1|1|1|1|2|2|2|2|1|2|4|5|4|4|4|4|5|5|5|5|
+---+---+---+---+---+---+---+---+---+---+---+
```

Before and after deletion of next hop 3
under the resilient hashing algorithm.

When weights of next hops in a group are altered, it may be possible to choose a subset of buckets that are currently not used for forwarding traffic, and use those to satisfy the new next-hop distribution demands, keeping the "busy" buckets intact. This way, established flows are ideally kept being forwarded to the same endpoints through the same paths as before the next-hop group change.

73.1 Algorithm

In a nutshell, the algorithm works as follows. Each next hop deserves a certain number of buckets, according to its weight and the number of buckets in the hash table. In accordance with the source code, we will call this number a "wants count" of a next hop. In case of an event that might cause bucket allocation change, the wants counts for individual next hops are updated.

Next hops that have fewer buckets than their wants count, are called "underweight". Those that have more are "overweight". If there are no overweight (and therefore no underweight) next hops in the group, it is said to be "balanced".

Each bucket maintains a last-used timer. Every time a packet is forwarded through a bucket, this timer is updated to current jiffies value. One attribute of a resilient group is then the "idle timer", which is the amount of time that a bucket must not be hit by traffic in order for it to be considered "idle". Buckets that are not idle are busy.

After assigning wants counts to next hops, an "upkeep" algorithm runs. For buckets:

- 1) that have no assigned next hop, or
- 2) whose next hop has been removed, or
- 3) that are idle and their next hop is overweight,

upkeep changes the next hop that the bucket references to one of the underweight next hops. If, after considering all buckets in this manner, there are still underweight next hops, another upkeep run is scheduled to a future time.

There may not be enough "idle" buckets to satisfy the updated wants counts of all next hops. Another attribute of a resilient group is the "unbalanced timer". This timer can be set to 0, in which case the table will stay out of balance until idle buckets do appear, possibly never. If set to a non-zero value, the value represents the period of time that the table is permitted to stay out of balance.

With this in mind, we update the above list of conditions with one more item. Thus buckets:

- 4) whose next hop is overweight, and the amount of time that the table has been out of balance exceeds the unbalanced timer, if that is non-zero,
... are migrated as well.

73.2 Offloading & Driver Feedback

When offloading resilient groups, the algorithm that distributes buckets among next hops is still the one in SW. Drivers are notified of updates to next hop groups in the following three ways:

- Full group notification with the type `NH_NOTIFIER_INFO_TYPE_RES_TABLE`. This is used just after the group is created and buckets populated for the first time.
- Single-bucket notifications of the type `NH_NOTIFIER_INFO_TYPE_RES_BUCKET`, which is used for notifications of individual migrations within an already-established group.
- Pre-replace notification, `NEXTHOP_EVENT_RES_TABLE_PRE_REPLACE`. This is sent before the group is replaced, and is a way for the driver to veto the group before committing anything to the HW.

Some single-bucket notifications are forced, as indicated by the "force" flag in the notification. Those are used for the cases where e.g. the next hop associated with the bucket was removed, and the bucket really must be migrated.

Non-forced notifications can be overridden by the driver by returning an error code. The use case for this is that the driver notifies the HW that a bucket should be migrated, but the HW discovers that the bucket has in fact been hit by traffic.

A second way for the HW to report that a bucket is busy is through the `nexthop_res_grp_activity_update()` API. The buckets identified this way as busy are treated as if traffic hit them.

Offloaded buckets should be flagged as either "offload" or "trap". This is done through the `nexthop_bucket_set_hw_flags()` API.

73.3 Netlink UAPI

73.3.1 Resilient Group Replacement

Resilient groups are configured using the `RTM_NEWNEXTHOP` message in the same manner as other multipath groups. The following changes apply to the attributes passed in the netlink message:

<code>NHA_GROUP_TYPE</code>	Should be <code>NEXTHOP_GRP_TYPE_RES</code> for resilient group.
<code>NHA_RES_GROUP</code>	A nest that contains attributes specific to resilient groups.

`NHA_RES_GROUP` payload:

<code>NHA_RES_GROUP_BUCKETS</code>	Number of buckets in the hash table.
<code>NHA_RES_GROUP_IDLE_TIMER</code>	Idle timer in units of <code>clock_t</code> .
<code>NHA_RES_GROUP_UNBALANCED_TIMER</code>	Unbalanced timer in units of <code>clock_t</code> .

73.3.2 Next Hop Get

Requests to get resilient next-hop groups use the RTM_GETNEXTHOP message in exactly the same way as other next hop get requests. The response attributes match the replacement attributes cited above, except NHA_RES_GROUP payload will include the following attribute:

NHA_RES_GROUP_UNBALANCED_TIME	How long has the resilient group been out of balance, in units of clock_t.
-------------------------------	--

73.3.3 Bucket Get

The message RTM_GETNEXTHOPBUCKET without the NLM_F_DUMP flag is used to request a single bucket. The attributes recognized at get requests are:

NHA_ID	ID of the next-hop group that the bucket belongs to.
NHA_RES_BUCKET	A nest that contains attributes specific to bucket.

NHA_RES_BUCKET payload:

NHA_RES_BUCKET_INDEX	Index of bucket in the resilient table.
----------------------	---

73.3.4 Bucket Dumps

The message RTM_GETNEXTHOPBUCKET with the NLM_F_DUMP flag is used to request a dump of matching buckets. The attributes recognized at dump requests are:

NHA_ID	If specified, limits the dump to just the next-hop group with this ID.
NHA_OIF	If specified, limits the dump to buckets that contain next hops that use the device with this ifindex.
NHA_MASTER	If specified, limits the dump to buckets that contain next hops that use a device in the VRF with this ifindex.
NHA_RES_BUCKET	A nest that contains attributes specific to bucket.

NHA_RES_BUCKET payload:

NHA_RES_BUCKET_NH_ID	If specified, limits the dump to just the buckets that contain the next hop with this ID.
----------------------	---

73.4 Usage

To illustrate the usage, consider the following commands:

```
# ip nexthop add id 1 via 192.0.2.2 dev eth0
# ip nexthop add id 2 via 192.0.2.3 dev eth0
# ip nexthop add id 10 group 1/2 type resilient \
    buckets 8 idle_timer 60 unbalanced_timer 300
```

The last command creates a resilient next-hop group. It will have 8 buckets (which is unusually low number, and used here for demonstration purposes only), each bucket will be considered idle when no traffic hits it for at least 60 seconds, and if the table remains out of balance for 300 seconds, it will be forcefully brought into balance.

Changing next-hop weights leads to change in bucket allocation:

```
# ip nexthop replace id 10 group 1,3/2 type resilient
```

This can be confirmed by looking at individual buckets:

```
# ip nexthop bucket show id 10
id 10 index 0 idle_time 5.59 nhid 1
id 10 index 1 idle_time 5.59 nhid 1
id 10 index 2 idle_time 8.74 nhid 2
id 10 index 3 idle_time 8.74 nhid 2
id 10 index 4 idle_time 8.74 nhid 1
id 10 index 5 idle_time 8.74 nhid 1
id 10 index 6 idle_time 8.74 nhid 1
id 10 index 7 idle_time 8.74 nhid 1
```

Note the two buckets that have a shorter idle time. Those are the ones that were migrated after the next-hop replace command to satisfy the new demand that next hop 1 be given 6 buckets instead of 4.

73.5 Netdevsim

The netdevsim driver implements a mock offload of resilient groups, and exposes debugfs interface that allows marking individual buckets as busy. For example, the following will mark bucket 23 in next-hop group 10 as active:

```
# echo 10 23 > /sys/kernel/debug/netdevsim/netdevsim10/fib/nexthop_bucket_
→activity
```

In addition, another debugfs interface can be used to configure that the next attempt to migrate a bucket should fail:

```
# echo 1 > /sys/kernel/debug/netdevsim/netdevsim10/fib/fail_nexthop_bucket_
→replace
```

Besides serving as an example, the interfaces that netdevsim exposes are useful in automated testing, and tools/testing/selftests/drivers/net/netdevsim/nexthop.sh makes use of

them to test the algorithm.

NETFILTER CONNTRACK SYSFS VARIABLES

74.1 /proc/sys/net/netfilter/nf_conntrack_* Variables:

nf_conntrack_acct - BOOLEAN

- 0 - disabled (default)
- not 0 - enabled

Enable connection tracking flow accounting. 64-bit byte and packet counters per flow are added.

nf_conntrack_buckets - INTEGER

Size of hash table. If not specified as parameter during module loading, the default size is calculated by dividing total memory by 16384 to determine the number of buckets. The hash table will never have fewer than 1024 and never more than 262144 buckets. This sysctl is only writeable in the initial net namespace.

nf_conntrack_checksum - BOOLEAN

- 0 - disabled
- not 0 - enabled (default)

Verify checksum of incoming packets. Packets with bad checksums are in INVALID state. If this is enabled, such packets will not be considered for connection tracking.

nf_conntrack_count - INTEGER (read-only)

Number of currently allocated flow entries.

nf_conntrack_events - BOOLEAN

- 0 - disabled
- 1 - enabled
- 2 - auto (default)

If this option is enabled, the connection tracking code will provide userspace with connection tracking events via ctnetlink. The default allocates the extension if a userspace program is listening to ctnetlink events.

nf_conntrack_expect_max - INTEGER

Maximum size of expectation table. Default value is nf_conntrack_buckets / 256. Minimum is 1.

nf_conntrack_frag6_high_thresh - INTEGER

default 262144

Maximum memory used to reassemble IPv6 fragments. When nf_conntrack_frag6_high_thresh bytes of memory is allocated for this purpose, the fragment handler will toss packets until nf_conntrack_frag6_low_thresh is reached.

nf_conntrack_frag6_low_thresh - INTEGER

default 196608

See nf_conntrack_frag6_low_thresh

nf_conntrack_frag6_timeout - INTEGER (seconds)

default 60

Time to keep an IPv6 fragment in memory.

nf_conntrack_generic_timeout - INTEGER (seconds)

default 600

Default for generic timeout. This refers to layer 4 unknown/unsupported protocols.

nf_conntrack_icmp_timeout - INTEGER (seconds)

default 30

Default for ICMP timeout.

nf_conntrack_icmpv6_timeout - INTEGER (seconds)

default 30

Default for ICMP6 timeout.

nf_conntrack_log_invalid - INTEGER

- 0 - disable (default)
- 1 - log ICMP packets
- 6 - log TCP packets
- 17 - log UDP packets
- 33 - log DCCP packets
- 41 - log ICMPv6 packets
- 136 - log UDPLITE packets
- 255 - log packets of any protocol

Log invalid packets of a type specified by value.

nf_conntrack_max - INTEGER

Maximum number of allowed connection tracking entries. This value is set to nf_conntrack_buckets by default. Note that connection tracking entries are added to the table twice -- once for the original direction and once for the reply direction (i.e., with the reversed address). This means that with default settings a maxed-out table will have an average hash chain length of 2, not 1.

nf_conntrack_tcp_be Liberal - BOOLEAN

- 0 - disabled (default)
- not 0 - enabled

Be conservative in what you do, be liberal in what you accept from others. If it's non-zero, we mark only out of window RST segments as INVALID.

nf_conntrack_tcp_ignore_invalid_rst - BOOLEAN

- 0 - disabled (default)
- 1 - enabled

If it's 1, we don't mark out of window RST segments as INVALID.

nf_conntrack_tcp_loose - BOOLEAN

- 0 - disabled
- not 0 - enabled (default)

If it is set to zero, we disable picking up already established connections.

nf_conntrack_tcp_max_retrans - INTEGER

default 3

Maximum number of packets that can be retransmitted without received an (acceptable) ACK from the destination. If this number is reached, a shorter timer will be started.

nf_conntrack_tcp_timeout_close - INTEGER (seconds)

default 10

nf_conntrack_tcp_timeout_close_wait - INTEGER (seconds)

default 60

nf_conntrack_tcp_timeout_established - INTEGER (seconds)

default 432000 (5 days)

nf_conntrack_tcp_timeout_fin_wait - INTEGER (seconds)

default 120

nf_conntrack_tcp_timeout_last_ack - INTEGER (seconds)

default 30

nf_conntrack_tcp_timeout_max_retrans - INTEGER (seconds)

default 300

nf_conntrack_tcp_timeout_syn_recv - INTEGER (seconds)

default 60

nf_conntrack_tcp_timeout_syn_sent - INTEGER (seconds)

default 120

nf_conntrack_tcp_timeout_time_wait - INTEGER (seconds)

default 120

nf_conntrack_tcp_timeout_unacknowledged - INTEGER (seconds)

default 300

nf_conntrack_timestamp - BOOLEAN

- 0 - disabled (default)
- not 0 - enabled

Enable connection tracking flow timestamping.

nf_conntrack_sctp_timeout_closed - INTEGER (seconds)

default 10

nf_conntrack_sctp_timeout_cookie_wait - INTEGER (seconds)

default 3

nf_conntrack_sctp_timeout_cookie_echoed - INTEGER (seconds)

default 3

nf_conntrack_sctp_timeout_established - INTEGER (seconds)

default 210

Default is set to (hb_interval * path_max_retrans + rto_max)

nf_conntrack_sctp_timeout_shutdown_sent - INTEGER (seconds)

default 3

nf_conntrack_sctp_timeout_shutdown_recd - INTEGER (seconds)

default 3

nf_conntrack_sctp_timeout_shutdown_ack_sent - INTEGER (seconds)

default 3

nf_conntrack_sctp_timeout_heartbeat_sent - INTEGER (seconds)

default 30

This timeout is used to setup conntrack entry on secondary paths. Default is set to hb_interval.

nf_conntrack_udp_timeout - INTEGER (seconds)

default 30

nf_conntrack_udp_timeout_stream - INTEGER (seconds)

default 120

This extended timeout will be used in case there is an UDP stream detected.

nf_conntrack_gre_timeout - INTEGER (seconds)

default 30

nf_conntrack_gre_timeout_stream - INTEGER (seconds)

default 180

This extended timeout will be used in case there is an GRE stream detected.

nf_hooks_lwtunnel - BOOLEAN

- 0 - disabled (default)
- not 0 - enabled

If this option is enabled, the lightweight tunnel netfilter hooks are enabled. This option cannot be disabled once it is enabled.

nf_flowtable_tcp_timeout - INTEGER (seconds)

default 30

Control offload timeout for tcp connections. TCP connections may be offloaded from nf conntrack to nf flow table. Once aged, the connection is returned to nf conntrack with tcp pickup timeout.

nf_flowtable_udp_timeout - INTEGER (seconds)

default 30

Control offload timeout for udp connections. UDP connections may be offloaded from nf conntrack to nf flow table. Once aged, the connection is returned to nf conntrack with udp pickup timeout.

NETFILTER'S FLOWTABLE INFRASTRUCTURE

This documentation describes the Netfilter flowtable infrastructure which allows you to define a fastpath through the flowtable datapath. This infrastructure also provides hardware offload support. The flowtable supports for the layer 3 IPv4 and IPv6 and the layer 4 TCP and UDP protocols.

75.1 Overview

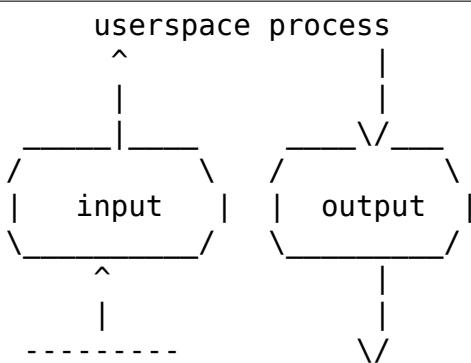
Once the first packet of the flow successfully goes through the IP forwarding path, from the second packet on, you might decide to offload the flow to the flowtable through your ruleset. The flowtable infrastructure provides a rule action that allows you to specify when to add a flow to the flowtable.

A packet that finds a matching entry in the flowtable (ie. flowtable hit) is transmitted to the output netdevice via `neigh_xmit()`, hence, packets bypass the classic IP forwarding path (the visible effect is that you do not see these packets from any of the Netfilter hooks coming after ingress). In case that there is no matching entry in the flowtable (ie. flowtable miss), the packet follows the classic IP forwarding path.

The flowtable uses a resizable hashtable. Lookups are based on the following n-tuple selectors: layer 2 protocol encapsulation (VLAN and PPPoE), layer 3 source and destination, layer 4 source and destination ports and the input interface (useful in case there are several conntrack zones in place).

The 'flow add' action allows you to populate the flowtable, the user selectively specifies what flows are placed into the flowtable. Hence, packets follow the classic IP forwarding path unless the user explicitly instruct flows to use this new alternative forwarding path via policy.

The flowtable datapath is represented in Fig.1, which describes the classic IP forwarding path including the Netfilter hooks and the flowtable fastpath bypass.



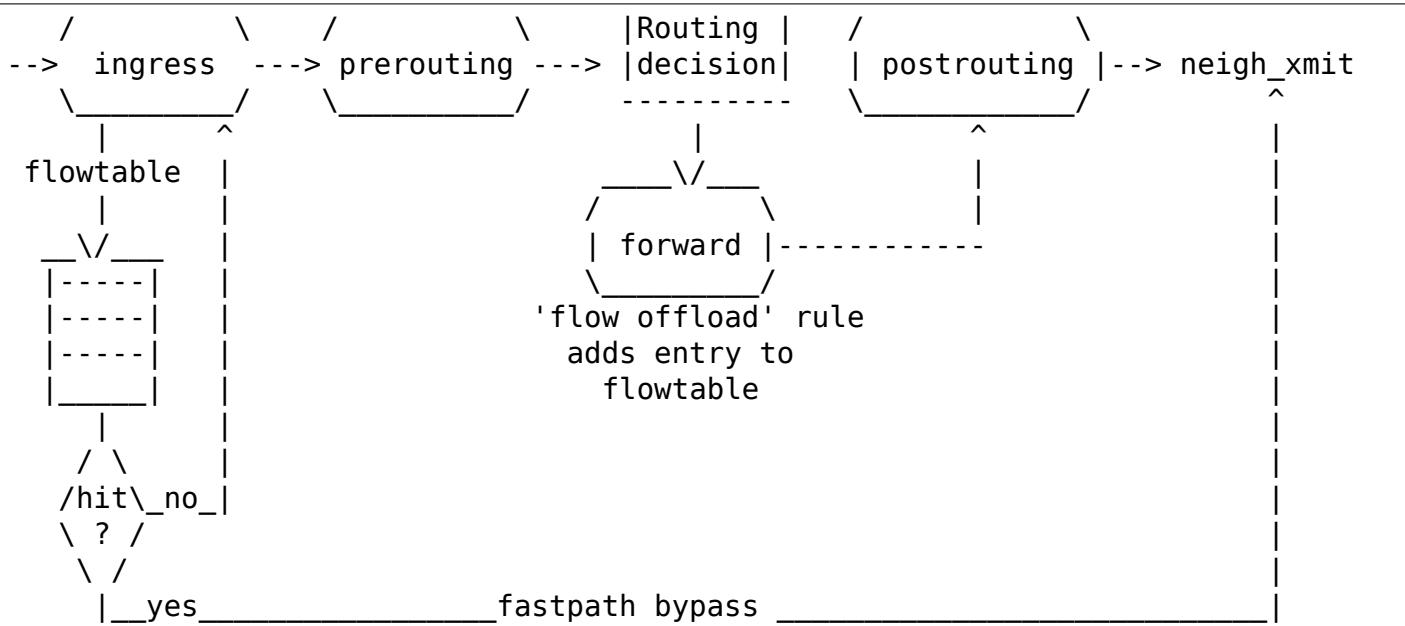


Fig.1 Netfilter hooks and flowtable interactions

The flowtable entry also stores the NAT configuration, so all packets are mangled according to the NAT policy that is specified from the classic IP forwarding path. The TTL is decremented before calling `neigh_xmit()`. Fragmented traffic is passed up to follow the classic IP forwarding path given that the transport header is missing, in this case, flowtable lookups are not possible. TCP RST and FIN packets are also passed up to the classic IP forwarding path to release the flow gracefully. Packets that exceed the MTU are also passed up to the classic forwarding path to report packet-too-big ICMP errors to the sender.

75.2 Example configuration

Enabling the flowtable bypass is relatively easy, you only need to create a flowtable and add one rule to your forward chain:

```

table inet x {
    flowtable f {
        hook ingress priority 0; devices = { eth0, eth1 };
    }
    chain y {
        type filter hook forward priority 0; policy accept;
        ip protocol tcp flow add @f
        counter packets 0 bytes 0
    }
}
  
```

This example adds the flowtable '`f`' to the ingress hook of the `eth0` and `eth1` netdevices. You can create as many flowtables as you want in case you need to perform resource partitioning. The flowtable priority defines the order in which hooks are run in the pipeline, this is convenient in case you already have a nftables ingress chain (make sure the flowtable priority is smaller than the nftables ingress chain hence the flowtable runs before in the pipeline).

The 'flow offload' action from the forward chain 'y' adds an entry to the flowtable for the TCP syn-ack packet coming in the reply direction. Once the flow is offloaded, you will observe that the counter rule in the example above does not get updated for the packets that are being forwarded through the forwarding bypass.

You can identify offloaded flows through the [OFFLOAD] tag when listing your connection tracking table.

```
# conntrack -L
tcp      6 src=10.141.10.2 dst=192.168.10.2 sport=52728 dport=5201 src=192.168.
         ↳ 10.2 dst=192.168.10.1 sport=5201 dport=52728 [OFFLOAD] mark=0 use=2
```

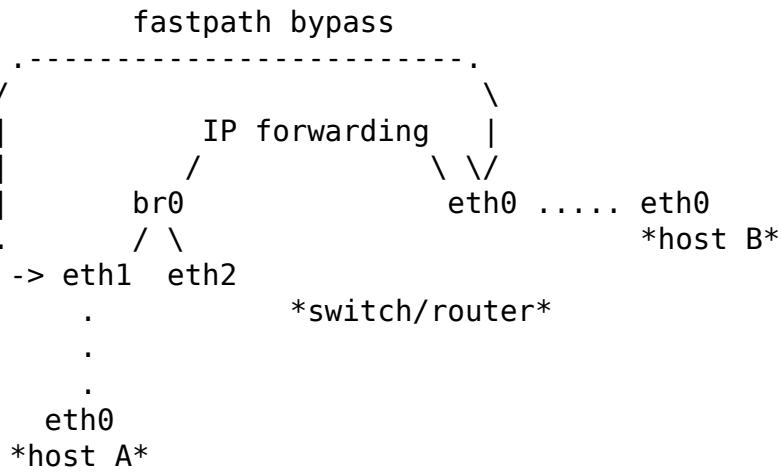
75.3 Layer 2 encapsulation

Since Linux kernel 5.13, the flowtable infrastructure discovers the real netdevice behind VLAN and PPPoE netdevices. The flowtable software datapath parses the VLAN and PPPoE layer 2 headers to extract the ethertype and the VLAN ID / PPPoE session ID which are used for the flowtable lookups. The flowtable datapath also deals with layer 2 decapsulation.

You do not need to add the PPPoE and the VLAN devices to your flowtable, instead the real device is sufficient for the flowtable to track your flows.

75.4 Bridge and IP forwarding

Since Linux kernel 5.13, you can add bridge ports to the flowtable. The flowtable infrastructure discovers the topology behind the bridge device. This allows the flowtable to define a fastpath bypass between the bridge ports (represented as eth1 and eth2 in the example figure below) and the gateway device (represented as eth0) in your switch/router.



The flowtable infrastructure also supports for bridge VLAN filtering actions such as PVID and untagged. You can also stack a classic VLAN device on top of your bridge port.

If you would like that your flowtable defines a fastpath between your bridge ports and your IP forwarding path, you have to add your bridge ports (as represented by the real netdevice) to your flowtable definition.

75.5 Counters

The flowtable can synchronize packet and byte counters with the existing connection tracking entry by specifying the counter statement in your flowtable definition, e.g.

```
table inet x {
    flowtable f {
        hook ingress priority 0; devices = { eth0, eth1 };
        counter
    }
}
```

Counter support is available since Linux kernel 5.7.

75.6 Hardware offload

If your network device provides hardware offload support, you can turn it on by means of the 'offload' flag in your flowtable definition, e.g.

```
table inet x {
    flowtable f {
        hook ingress priority 0; devices = { eth0, eth1 };
        flags offload;
    }
}
```

There is a workqueue that adds the flows to the hardware. Note that a few packets might still run over the flowtable software path until the workqueue has a chance to offload the flow to the network device.

You can identify hardware offloaded flows through the [HW_OFFLOAD] tag when listing your connection tracking table. Please, note that the [OFFLOAD] tag refers to the software offload mode, so there is a distinction between [OFFLOAD] which refers to the software flowtable fastpath and [HW_OFFLOAD] which refers to the hardware offload datapath being used by the flow.

The flowtable hardware offload infrastructure also supports for the DSA (Distributed Switch Architecture).

75.7 Limitations

The flowtable behaves like a cache. The flowtable entries might get stale if either the destination MAC address or the egress netdevice that is used for transmission changes.

This might be a problem if:

- You run the flowtable in software mode and you combine bridge and IP forwarding in your setup.
- Hardware offload is enabled.

75.8 More reading

This documentation is based on the LWN.net articles¹². Rafal Milecki also made a very complete and comprehensive summary called "A state of network acceleration" that describes how things were before this infrastructure was mainlined³ and it also makes a rough summary of this work⁴.

¹ <https://lwn.net/Articles/738214/>

² <https://lwn.net/Articles/742164/>

³ <http://lists.infradead.org/pipermail/lede-dev/2018-January/010830.html>

⁴ <http://lists.infradead.org/pipermail/lede-dev/2018-January/010829.html>

OPEN VSWITCH DATAPATH DEVELOPER DOCUMENTATION

The Open vSwitch kernel module allows flexible userspace control over flow-level packet processing on selected network devices. It can be used to implement a plain Ethernet switch, network device bonding, VLAN processing, network access control, flow-based network control, and so on.

The kernel module implements multiple "datapaths" (analogous to bridges), each of which can have multiple "vports" (analogous to ports within a bridge). Each datapath also has associated with it a "flow table" that userspace populates with "flows" that map from keys based on packet headers and metadata to sets of actions. The most common action forwards the packet to another vport; other actions are also implemented.

When a packet arrives on a vport, the kernel module processes it by extracting its flow key and looking it up in the flow table. If there is a matching flow, it executes the associated actions. If there is no match, it queues the packet to userspace for processing (as part of its processing, userspace will likely set up a flow to handle further packets of the same type entirely in-kernel).

76.1 Flow key compatibility

Network protocols evolve over time. New protocols become important and existing protocols lose their prominence. For the Open vSwitch kernel module to remain relevant, it must be possible for newer versions to parse additional protocols as part of the flow key. It might even be desirable, someday, to drop support for parsing protocols that have become obsolete. Therefore, the Netlink interface to Open vSwitch is designed to allow carefully written userspace applications to work with any version of the flow key, past or future.

To support this forward and backward compatibility, whenever the kernel module passes a packet to userspace, it also passes along the flow key that it parsed from the packet. Userspace then extracts its own notion of a flow key from the packet and compares it against the kernel-provided version:

- If userspace's notion of the flow key for the packet matches the kernel's, then nothing special is necessary.
- If the kernel's flow key includes more fields than the userspace version of the flow key, for example if the kernel decoded IPv6 headers but userspace stopped at the Ethernet type (because it does not understand IPv6), then again nothing special is necessary. Userspace can still set up a flow in the usual way, as long as it uses the kernel-provided flow key to do it.
- If the userspace flow key includes more fields than the kernel's, for example if userspace decoded an IPv6 header but the kernel stopped at the Ethernet type, then userspace can

forward the packet manually, without setting up a flow in the kernel. This case is bad for performance because every packet that the kernel considers part of the flow must go to userspace, but the forwarding behavior is correct. (If userspace can determine that the values of the extra fields would not affect forwarding behavior, then it could set up a flow anyway.)

How flow keys evolve over time is important to making this work, so the following sections go into detail.

76.2 Flow key format

A flow key is passed over a Netlink socket as a sequence of Netlink attributes. Some attributes represent packet metadata, defined as any information about a packet that cannot be extracted from the packet itself, e.g. the vport on which the packet was received. Most attributes, however, are extracted from headers within the packet, e.g. source and destination addresses from Ethernet, IP, or TCP headers.

The `<linux/openvswitch.h>` header file defines the exact format of the flow key attributes. For informal explanatory purposes here, we write them as comma-separated strings, with parentheses indicating arguments and nesting. For example, the following could represent a flow key corresponding to a TCP packet that arrived on vport 1:

```
in_port(1), eth(src=e0:91:f5:21:d0:b2, dst=00:02:e3:0f:80:a4),  
eth_type(0x0800), ipv4(src=172.16.0.20, dst=172.18.0.52, proto=17, tos=0,  
frag=no), tcp(src=49163, dst=80)
```

Often we ellipsize arguments not important to the discussion, e.g.:

```
in_port(1), eth(...), eth_type(0x0800), ipv4(...), tcp(...)
```

76.3 Wildcarded flow key format

A wildcarded flow is described with two sequences of Netlink attributes passed over the Netlink socket. A flow key, exactly as described above, and an optional corresponding flow mask.

A wildcarded flow can represent a group of exact match flows. Each '1' bit in the mask specifies a exact match with the corresponding bit in the flow key. A '0' bit specifies a don't care bit, which will match either a '1' or '0' bit of a incoming packet. Using wildcarded flow can improve the flow set up rate by reduce the number of new flows need to be processed by the user space program.

Support for the mask Netlink attribute is optional for both the kernel and user space program. The kernel can ignore the mask attribute, installing an exact match flow, or reduce the number of don't care bits in the kernel to less than what was specified by the user space program. In this case, variations in bits that the kernel does not implement will simply result in additional flow setups. The kernel module will also work with user space programs that neither support nor supply flow mask attributes.

Since the kernel may ignore or modify wildcard bits, it can be difficult for the userspace program to know exactly what matches are installed. There are two possible approaches: reactively

install flows as they miss the kernel flow table (and therefore not attempt to determine wildcard changes at all) or use the kernel's response messages to determine the installed wildcards.

When interacting with userspace, the kernel should maintain the match portion of the key exactly as originally installed. This will provide a handle to identify the flow for all future operations. However, when reporting the mask of an installed flow, the mask should include any restrictions imposed by the kernel.

The behavior when using overlapping wildcared flows is undefined. It is the responsibility of the user space program to ensure that any incoming packet can match at most one flow, wildcared or not. The current implementation performs best-effort detection of overlapping wildcared flows and may reject some but not all of them. However, this behavior may change in future versions.

76.4 Unique flow identifiers

An alternative to using the original match portion of a key as the handle for flow identification is a unique flow identifier, or "UFID". UFIDs are optional for both the kernel and user space program.

User space programs that support UFID are expected to provide it during flow setup in addition to the flow, then refer to the flow using the UFID for all future operations. The kernel is not required to index flows by the original flow key if a UFID is specified.

76.5 Basic rule for evolving flow keys

Some care is needed to really maintain forward and backward compatibility for applications that follow the rules listed under "Flow key compatibility" above.

The basic rule is obvious:

```
=====
New network protocol support must only supplement existing flow
key attributes. It must not change the meaning of already defined
flow key attributes.
=====
```

This rule does have less-obvious consequences so it is worth working through a few examples. Suppose, for example, that the kernel module did not already implement VLAN parsing. Instead, it just interpreted the 802.1Q TPID (0x8100) as the Ethertype then stopped parsing the packet. The flow key for any packet with an 802.1Q header would look essentially like this, ignoring metadata:

```
eth(...), eth_type(0x8100)
```

Naively, to add VLAN support, it makes sense to add a new "vlan" flow key attribute to contain the VLAN tag, then continue to decode the encapsulated headers beyond the VLAN tag using the existing field definitions. With this change, a TCP packet in VLAN 10 would have a flow key much like this:

```
eth(...), vlan(vid=10, pcp=0), eth_type(0x0800), ip(proto=6, ...), tcp(...)
```

But this change would negatively affect a userspace application that has not been updated to understand the new "vlan" flow key attribute. The application could, following the flow compatibility rules above, ignore the "vlan" attribute that it does not understand and therefore assume that the flow contained IP packets. This is a bad assumption (the flow only contains IP packets if one parses and skips over the 802.1Q header) and it could cause the application's behavior to change across kernel versions even though it follows the compatibility rules.

The solution is to use a set of nested attributes. This is, for example, why 802.1Q support uses nested attributes. A TCP packet in VLAN 10 is actually expressed as:

```
eth(...), eth_type(0x8100), vlan(vid=10, pcp=0), encaps(eth_type(0x0800),  
ip(proto=6, ...), tcp(...))
```

Notice how the "eth_type", "ip", and "tcp" flow key attributes are nested inside the "encap" attribute. Thus, an application that does not understand the "vlan" key will not see either of those attributes and therefore will not misinterpret them. (Also, the outer eth_type is still 0x8100, not changed to 0x0800.)

76.6 Handling malformed packets

Don't drop packets in the kernel for malformed protocol headers, bad checksums, etc. This would prevent userspace from implementing a simple Ethernet switch that forwards every packet.

Instead, in such a case, include an attribute with "empty" content. It doesn't matter if the empty content could be valid protocol values, as long as those values are rarely seen in practice, because userspace can always forward all packets with those values to userspace and handle them individually.

For example, consider a packet that contains an IP header that indicates protocol 6 for TCP, but which is truncated just after the IP header, so that the TCP header is missing. The flow key for this packet would include a tcp attribute with all-zero src and dst, like this:

```
eth(...), eth_type(0x0800), ip(proto=6, ...), tcp(src=0, dst=0)
```

As another example, consider a packet with an Ethernet type of 0x8100, indicating that a VLAN TCI should follow, but which is truncated just after the Ethernet type. The flow key for this packet would include an all-zero-bits vlan and an empty encaps attribute, like this:

```
eth(...), eth_type(0x8100), vlan(0), encaps()
```

Unlike a TCP packet with source and destination ports 0, an all-zero-bits VLAN TCI is not that rare, so the CFI bit (aka VLAN_TAG_PRESENT inside the kernel) is ordinarily set in a vlan attribute expressly to allow this situation to be distinguished. Thus, the flow key in this second example unambiguously indicates a missing or malformed VLAN TCI.

76.7 Other rules

The other rules for flow keys are much less subtle:

- Duplicate attributes are not allowed at a given nesting level.
- Ordering of attributes is not significant.
- When the kernel sends a given flow key to userspace, it always composes it the same way. This allows userspace to hash and compare entire flow keys that it may not be able to fully interpret.

OPERATIONAL STATES

77.1 1. Introduction

Linux distinguishes between administrative and operational state of an interface. Administrative state is the result of "ip link set dev <dev> up or down" and reflects whether the administrator wants to use the device for traffic.

However, an interface is not usable just because the admin enabled it - ethernet requires to be plugged into the switch and, depending on a site's networking policy and configuration, an 802.1X authentication to be performed before user data can be transferred. Operational state shows the ability of an interface to transmit this user data.

Thanks to 802.1X, userspace must be granted the possibility to influence operational state. To accommodate this, operational state is split into two parts: Two flags that can be set by the driver only, and a RFC2863 compatible state that is derived from these flags, a policy, and changeable from userspace under certain rules.

77.2 2. Querying from userspace

Both admin and operational state can be queried via the netlink operation RTM_GETLINK. It is also possible to subscribe to RTNLGRP_LINK to be notified of updates while the interface is admin up. This is important for setting from userspace.

These values contain interface state:

ifinfomsg::if_flags & IFF_UP:

Interface is admin up

ifinfomsg::if_flags & IFF_RUNNING:

Interface is in RFC2863 operational state UP or UNKNOWN. This is for backward compatibility, routing daemons, dhcp clients can use this flag to determine whether they should use the interface.

ifinfomsg::if_flags & IFF_LOWER_UP:

Driver has signaled *netif_carrier_on()*

ifinfomsg::if_flags & IFF_DORMANT:

Driver has signaled *netif_dormant_on()*

77.2.1 TLV IFLA_OPERSTATE

contains RFC2863 state of the interface in numeric representation:

IF_OPER_UNKNOWN (0):

Interface is in unknown state, neither driver nor userspace has set operational state. Interface must be considered for user data as setting operational state has not been implemented in every driver.

IF_OPER_NOTPRESENT (1):

Unused in current kernel (notpresent interfaces normally disappear), just a numerical placeholder.

IF_OPER_DOWN (2):

Interface is unable to transfer data on L1, f.e. ethernet is not plugged or interface is ADMIN down.

IF_OPER_LOWERLAYERDOWN (3):

Interfaces stacked on an interface that is IF_OPER_DOWN show this state (f.e. VLAN).

IF_OPER_TESTING (4):

Interface is in testing mode, for example executing driver self-tests or media (cable) test. It can't be used for normal traffic until tests complete.

IF_OPER_DORMANT (5):

Interface is L1 up, but waiting for an external event, f.e. for a protocol to establish. (802.1X)

IF_OPER_UP (6):

Interface is operational up and can be used.

This TLV can also be queried via sysfs.

77.2.2 TLV IFLA_LINKMODE

contains link policy. This is needed for userspace interaction described below.

This TLV can also be queried via sysfs.

77.3 3. Kernel driver API

Kernel drivers have access to two flags that map to IFF_LOWER_UP and IFF_DORMANT. These flags can be set from everywhere, even from interrupts. It is guaranteed that only the driver has write access, however, if different layers of the driver manipulate the same flag, the driver has to provide the synchronisation needed.

`_LINK_STATE_NOCARRIER`, maps to !IFF_LOWER_UP:

The driver uses `netif_carrier_on()` to clear and `netif_carrier_off()` to set this flag. On `netif_carrier_off()`, the scheduler stops sending packets. The name 'carrier' and the inversion are historical, think of it as lower layer.

Note that for certain kind of soft-devices, which are not managing any real hardware, it is possible to set this bit from userspace. One should use TLV IFLA_CARRIER to do so.

`netif_carrier_ok()` can be used to query that bit.

`_LINK_STATE_DORMANT`, maps to IFF_DORMANT:

Set by the driver to express that the device cannot yet be used because some driver controlled protocol establishment has to complete. Corresponding functions are `netif_dormant_on()` to set the flag, `netif_dormant_off()` to clear it and `netif_dormant()` to query.

On device allocation, both flags `_LINK_STATE_NO_CARRIER` and `_LINK_STATE_DORMANT` are cleared, so the effective state is equivalent to `netif_carrier_ok()` and `!netif_dormant()`.

Whenever the driver CHANGES one of these flags, a workqueue event is scheduled to translate the flag combination to IFLA_OPERSTATE as follows:

`!netif_carrier_ok()`:

IF_OPER_LOWER_LAYERDOWN if the interface is stacked, IF_OPER_DOWN otherwise.
Kernel can recognise stacked interfaces because their ifindex != iflink.

`netif_carrier_ok() && netif_dormant()`:

IF_OPER_DORMANT

`netif_carrier_ok() && !netif_dormant()`:

IF_OPER_UP if userspace interaction is disabled. Otherwise IF_OPER_DORMANT with the possibility for userspace to initiate the IF_OPER_UP transition afterwards.

77.4 4. Setting from userspace

Applications have to use the netlink interface to influence the RFC2863 operational state of an interface. Setting IFLA_LINKMODE to 1 via RTM_SETLINK instructs the kernel that an interface should go to IF_OPER_DORMANT instead of IF_OPER_UP when the combination `netif_carrier_ok() && !netif_dormant()` is set by the driver. Afterwards, the userspace application can set IFLA_OPERSTATE to IF_OPER_DORMANT or IF_OPER_UP as long as the driver does not set `netif_carrier_off()` or `netif_dormant_on()`. Changes made by userspace are multicasted on the netlink group RTNLGRP_LINK.

So basically a 802.1X supplicant interacts with the kernel like this:

- subscribe to RTNLGRP_LINK
- set IFLA_LINKMODE to 1 via RTM_SETLINK
- query RTM_GETLINK once to get initial state
- if initial flags are not (IFF_LOWER_UP && !IFF_DORMANT), wait until netlink multicast signals this state
- do 802.1X, eventually abort if flags go down again
- send RTM_SETLINK to set operstate to IF_OPER_UP if authentication succeeds, IF_OPER_DORMANT otherwise
- see how operstate and IFF_RUNNING is echoed via netlink multicast
- set interface back to IF_OPER_DORMANT if 802.1X reauthentication fails
- restart if kernel changes IFF_LOWER_UP or IFF_DORMANT flag

if supplicant goes down, bring back IFLA_LINKMODE to 0 and IFLA_OPERSTATE to a sane value.

A routing daemon or dhcp client just needs to care for IFF_RUNNING or waiting for operstate to go IF_OPER_UP/IF_OPER_UNKNOWN before considering the interface / querying a DHCP address.

For technical questions and/or comments please e-mail to Stefan Rompf (stefan at loplof.de).

PACKET MMAP

78.1 Abstract

This file documents the mmap() facility available with the PACKET socket interface. This type of sockets is used for

- i) capture network traffic with utilities like tcpdump,
- ii) transmit network traffic, or any other that needs raw access to network interface.

Howto can be found at:

<https://sites.google.com/site/packetmmap/>

Please send your comments to

- Ulisses Alonso Camaró <uaca@i.hate.spam.alumni.uv.es>
- Johann Baudy

78.2 Why use PACKET_MMAP

Non PACKET_MMAP capture process (plain AF_PACKET) is very inefficient. It uses very limited buffers and requires one system call to capture each packet, it requires two if you want to get packet's timestamp (like libpcap always does).

On the other hand PACKET_MMAP is very efficient. PACKET_MMAP provides a size configurable circular buffer mapped in user space that can be used to either send or receive packets. This way reading packets just needs to wait for them, most of the time there is no need to issue a single system call. Concerning transmission, multiple packets can be sent through one system call to get the highest bandwidth. By using a shared buffer between the kernel and the user also has the benefit of minimizing packet copies.

It's fine to use PACKET_MMAP to improve the performance of the capture and transmission process, but it isn't everything. At least, if you are capturing at high speeds (this is relative to the cpu speed), you should check if the device driver of your network interface card supports some sort of interrupt load mitigation or (even better) if it supports NAPI, also make sure it is enabled. For transmission, check the MTU (Maximum Transmission Unit) used and supported by devices of your network. CPU IRQ pinning of your network interface card can also be an advantage.

78.3 How to use mmap() to improve capture process

From the user standpoint, you should use the higher level libpcap library, which is a de facto standard, portable across nearly all operating systems including Win32.

Packet MMAP support was integrated into libpcap around the time of version 1.3.0; TPACKET_V3 support was added in version 1.5.0

78.4 How to use mmap() directly to improve capture process

From the system calls stand point, the use of PACKET_MMAP involves the following process:

[setup]	socket() -----> creation of the capture socket
	setsockopt() ---> allocation of the circular buffer (ring)
	option: PACKET_RX_RING
	mmap() -----> mapping of the allocated buffer to the
	user process
[capture]	poll() -----> to wait for incoming packets
[shutdown]	close() -----> destruction of the capture socket and
	deallocation of all associated
	resources.

socket creation and destruction is straight forward, and is done the same way with or without PACKET_MMAP:

```
int fd = socket(PF_PACKET, mode, htons(ETH_P_ALL));
```

where mode is SOCK_RAW for the raw interface where link level information can be captured or SOCK_DGRAM for the cooked interface where link level information capture is not supported and a link level pseudo-header is provided by the kernel.

The destruction of the socket and all associated resources is done by a simple call to close(fd).

Similarly as without PACKET_MMAP, it is possible to use one socket for capture and transmission. This can be done by mapping the allocated RX and TX buffer ring with a single mmap() call. See "Mapping and use of the circular buffer (ring)".

Next I will describe PACKET_MMAP settings and its constraints, also the mapping of the circular buffer in the user process and the use of this buffer.

78.5 How to use mmap() directly to improve transmission process

Transmission process is similar to capture as shown below:

[setup]	socket() -----> creation of the transmission socket setsockopt() ---> allocation of the circular buffer (ring) option: PACKET_TX_RING bind() -----> bind transmission socket with a network <u>interface</u>
	mmap() -----> mapping of the allocated buffer to the user process
[transmission]	poll() -----> wait for free packets (optional) send() -----> send all packets that are set as ready in the ring The flag MSG_DONTWAIT can be used to return before end of transfer.
[shutdown]	close() -----> destruction of the transmission socket and deallocation of all associated resources.

Socket creation and destruction is also straight forward, and is done the same way as in capturing described in the previous paragraph:

```
int fd = socket(PF_PACKET, mode, 0);
```

The protocol can optionally be 0 in case we only want to transmit via this socket, which avoids an expensive call to packet_rcv(). In this case, you also need to bind(2) the TX_RING with sll_protocol = 0 set. Otherwise, htons(ETH_P_ALL) or any other protocol, for example.

Binding the socket to your network interface is mandatory (with zero copy) to know the header size of frames used in the circular buffer.

As capture, each frame contains two parts:

```
-----  
| struct tpacket_hdr | Header. It contains the status of  
|                   | of this frame  
|-----|  
| data buffer       | . Data that will be sent over the network interface.  
|                   | .  
-----  
  
bind() associates the socket to your network interface thanks to  
sll_ifindex parameter of struct sockaddr_ll.
```

Initialization example::

```
struct sockaddr_ll my_addr;  
struct ifreq s_ifr;
```

```

...
strncpy_pad (s_ifr.ifr_name, "eth0", sizeof(s_ifr.ifr_name));

/* get interface index of eth0 */
ioctl(this->socket, SIOCGIFINDEX, &s_ifr);

/* fill sockaddr_ll struct to prepare binding */
my_addr.sll_family = AF_PACKET;
my_addr.sll_protocol = htons(ETH_P_ALL);
my_addr.sll_ifindex = s_ifr.ifr_ifindex;

/* bind socket to eth0 */
bind(this->socket, (struct sockaddr *)&my_addr, sizeof(struct sockaddr_ll));

```

A complete tutorial is available at: <https://sites.google.com/site/packet mmap/>

By default, the user should put data at:

```
frame base + TPACKET_HDRLEN - sizeof(struct sockaddr_ll)
```

So, whatever you choose for the socket mode (SOCK_DGRAM or SOCK_RAW), the beginning of the user data will be at:

```
frame base + TPACKET_ALIGN(sizeof(struct tpacket_hdr))
```

If you wish to put user data at a custom offset from the beginning of the frame (for payload alignment with SOCK_RAW mode for instance) you can set tp_net (with SOCK_DGRAM) or tp_mac (with SOCK_RAW). In order to make this work it must be enabled previously with setsockopt() and the PACKET_TX_HAS_OFF option.

78.6 PACKET_MMAP settings

To setup PACKET_MMAP from user level code is done with a call like

- Capture process:

```
setsockopt(fd, SOL_PACKET, PACKET_RX_RING, (void *) &req, sizeof(req))
```

- Transmission process:

```
setsockopt(fd, SOL_PACKET, PACKET_TX_RING, (void *) &req, sizeof(req))
```

The most significant argument in the previous call is the req parameter, this parameter must have the following structure:

```

struct tpacket_req
{
    unsigned int      tp_block_size; /* Minimal size of contiguous block */
    unsigned int      tp_block_nr;   /* Number of blocks */
    unsigned int      tp_frame_size; /* Size of frame */

```

```
    unsigned int      tp_frame_nr;      /* Total number of frames */
};
```

This structure is defined in /usr/include/linux/if_packet.h and establishes a circular buffer (ring) of unswappable memory. Being mapped in the capture process allows reading the captured frames and related meta-information like timestamps without requiring a system call.

Frames are grouped in blocks. Each block is a physically contiguous region of memory and holds tp_block_size/tp_frame_size frames. The total number of blocks is tp_block_nr. Note that tp_frame_nr is a redundant parameter because:

```
frames_per_block = tp_block_size/tp_frame_size
```

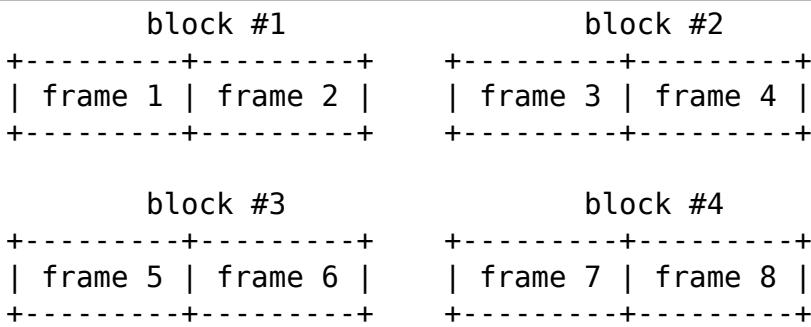
indeed, packet_set_ring checks that the following condition is true:

```
frames_per_block * tp_block_nr == tp_frame_nr
```

Lets see an example, with the following values:

```
tp_block_size= 4096
tp_frame_size= 2048
tp_block_nr  = 4
tp_frame_nr  = 8
```

we will get the following buffer structure:



A frame can be of any size with the only condition it can fit in a block. A block can only hold an integer number of frames, or in other words, a frame cannot be spawned across two blocks, so there are some details you have to take into account when choosing the frame_size. See "Mapping and use of the circular buffer (ring)".

78.7 PACKET_MMAP setting constraints

In kernel versions prior to 2.4.26 (for the 2.4 branch) and 2.6.5 (2.6 branch), the PACKET_MMAP buffer could hold only 32768 frames in a 32 bit architecture or 16384 in a 64 bit architecture.

78.7.1 Block size limit

As stated earlier, each block is a contiguous physical region of memory. These memory regions are allocated with calls to the `_get_free_pages()` function. As the name indicates, this function allocates pages of memory, and the second argument is "order" or a power of two number of pages, that is (for `PAGE_SIZE == 4096`) `order=0 ==> 4096 bytes`, `order=1 ==> 8192 bytes`, `order=2 ==> 16384 bytes`, etc. The maximum size of a region allocated by `_get_free_pages` is determined by the `MAX_PAGE_ORDER` macro. More precisely the limit can be calculated as:

```
PAGE_SIZE << MAX_PAGE_ORDER
```

```
In a i386 architecture PAGE_SIZE is 4096 bytes
In a 2.4/i386 kernel MAX_PAGE_ORDER is 10
In a 2.6/i386 kernel MAX_PAGE_ORDER is 11
```

So `get_free_pages` can allocate as much as 4MB or 8MB in a 2.4/2.6 kernel respectively, with an i386 architecture.

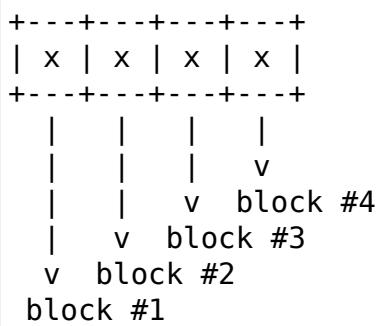
User space programs can include `/usr/include/sys/user.h` and `/usr/include/linux/mmzone.h` to get `PAGE_SIZE` `MAX_PAGE_ORDER` declarations.

The pagesize can also be determined dynamically with the `getpagesize(2)` system call.

78.7.2 Block number limit

To understand the constraints of `PACKET_MMAP`, we have to see the structure used to hold the pointers to each block.

Currently, this structure is a dynamically allocated vector with `kmalloc` called `pg_vec`, its size limits the number of blocks that can be allocated:



`kmalloc` allocates any number of bytes of physically contiguous memory from a pool of pre-determined sizes. This pool of memory is maintained by the slab allocator which is at the end the responsible for doing the allocation and hence which imposes the maximum memory that `kmalloc` can allocate.

In a 2.4/2.6 kernel and the i386 architecture, the limit is 131072 bytes. The predetermined sizes that `kmalloc` uses can be checked in the "size-<bytes>" entries of `/proc/slabinfo`

In a 32 bit architecture, pointers are 4 bytes long, so the total number of pointers to blocks is:

```
131072/4 = 32768 blocks
```

78.8 PACKET_MMAP buffer size calculator

Definitions:

<size-max>	is the maximum size of allocable with kmalloc (see /proc/slabinfo)
<pointer size>	depends on the architecture -- sizeof(void *)
<page size>	depends on the architecture -- PAGE_SIZE or getpagesize (2)
<max-order>	is the value defined with MAX_PAGE_ORDER
<frame size>	it's an upper bound of frame's capture size (more on this later)

from these definitions we will derive:

```
<block number> = <size-max>/<pointer size>
<block size> = <pagesize> << <max-order>
```

so, the max buffer size is:

```
<block number> * <block size>
```

and, the number of frames be:

```
<block number> * <block size> / <frame size>
```

Suppose the following parameters, which apply for 2.6 kernel and an i386 architecture:

```
<size-max> = 131072 bytes
<pointer size> = 4 bytes
<pagesize> = 4096 bytes
<max-order> = 11
```

and a value for <frame size> of 2048 bytes. These parameters will yield:

```
<block number> = 131072/4 = 32768 blocks
<block size> = 4096 << 11 = 8 MiB.
```

and hence the buffer will have a 262144 MiB size. So it can hold 262144 MiB / 2048 bytes = 134217728 frames

Actually, this buffer size is not possible with an i386 architecture. Remember that the memory is allocated in kernel space, in the case of an i386 kernel's memory size is limited to 1GiB.

All memory allocations are not freed until the socket is closed. The memory allocations are done with GFP_KERNEL priority, this basically means that the allocation can wait and swap other process' memory in order to allocate the necessary memory, so normally limits can be reached.

78.8.1 Other constraints

If you check the source code you will see that what I draw here as a frame is not only the link level frame. At the beginning of each frame there is a header called struct tpacket_hdr used in PACKET_MMAP to hold link level's frame meta information like timestamp. So what we draw here a frame it's really the following (from include/linux/if_packet.h):

```
/*
Frame structure:

- Start. Frame must be aligned to TPACKET_ALIGNMENT=16
- struct tpacket_hdr
- pad to TPACKET_ALIGNMENT=16
- struct sockaddr_ll
- Gap, chosen so that packet data (Start+tp_net) aligns to
  TPACKET_ALIGNMENT=16
- Start+tp_mac: [ Optional MAC header ]
- Start+tp_net: Packet data, aligned to TPACKET_ALIGNMENT=16.
- Pad to align to TPACKET_ALIGNMENT=16
*/

```

The following are conditions that are checked in packet_set_ring

- tp_block_size must be a multiple of PAGE_SIZE (1)
- tp_frame_size must be greater than TPACKET_HDRLEN (obvious)
- tp_frame_size must be a multiple of TPACKET_ALIGNMENT
- tp_frame_nr must be exactly frames_per_block*tp_block_nr

Note that tp_block_size should be chosen to be a power of two or there will be a waste of memory.

78.8.2 Mapping and use of the circular buffer (ring)

The mapping of the buffer in the user process is done with the conventional mmap function. Even the circular buffer is compound of several physically discontiguous blocks of memory, they are contiguous to the user space, hence just one call to mmap is needed:

```
mmap(0, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

If tp_frame_size is a divisor of tp_block_size frames will be contiguously spaced by tp_frame_size bytes. If not, each tp_block_size/tp_frame_size frames there will be a gap between the frames. This is because a frame cannot be spanned across two blocks.

To use one socket for capture and transmission, the mapping of both the RX and TX buffer ring has to be done with one call to mmap:

```
...
setsockopt(fd, SOL_PACKET, PACKET_RX_RING, &foo, sizeof(foo));
setsockopt(fd, SOL_PACKET, PACKET_TX_RING, &bar, sizeof(bar));
...
```

```
rx_ring = mmap(0, size * 2, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
tx_ring = rx_ring + size;
```

RX must be the first as the kernel maps the TX ring memory right after the RX one.

At the beginning of each frame there is an status field (see struct tpacket_hdr). If this field is 0 means that the frame is ready to be used for the kernel, If not, there is a frame the user can read and the following flags apply:

Capture process

From include/linux/if_packet.h:

```
#define TP_STATUS_COPY          (1 << 1)
#define TP_STATUS_LOSING         (1 << 2)
#define TP_STATUS_CSUMNOTREADY   (1 << 3)
#define TP_STATUS_CSUM_VALID      (1 << 7)
```

TP_STATUS_COPY	This flag indicates that the frame (and associated meta information) has been truncated because it's larger than tp_frame_size. This packet can be read entirely with recvfrom(). In order to make this work it must be enabled previously with setsockopt() and the PACKET_COPY_THRESH option. The number of frames that can be buffered to be read with recvfrom is limited like a normal socket. See the SO_RCVBUF option in the socket (7) man page.
TP_STATUS_LOSING	indicates there were packet drops from last time statistics where checked with getsockopt() and the PACKET_STATISTICS option.
TP_STATUS_CSUMNOTREADY	currently it's used for outgoing IP packets which its checksum will be done in hardware. So while reading the packet we should not try to check the checksum.
TP_STATUS_CSUM_VALID	This flag indicates that at least the transport header checksum of the packet has been already validated on the kernel side. If the flag is not set then we are free to check the checksum by ourselves provided that TP_STATUS_CSUMNOTREADY is also not set.

for convenience there are also the following defines:

```
#define TP_STATUS_KERNEL    0
#define TP_STATUS_USER       1
```

The kernel initializes all frames to TP_STATUS_KERNEL, when the kernel receives a packet it puts in the buffer and updates the status with at least the TP_STATUS_USER flag. Then the user can read the packet, once the packet is read the user must zero the status field, so the kernel can use again that frame buffer.

The user can use poll (any other variant should apply too) to check if new packets are in the ring:

```

struct pollfd pfd;

pfd.fd = fd;
pfd.revents = 0;
pfd.events = POLLIN|POLLRDNORM|POLLERR;

if (status == TP_STATUS_KERNEL)
    retval = poll(&pfd, 1, timeout);

```

It doesn't incur in a race condition to first check the status value and then poll for frames.

Transmission process

Those defines are also used for transmission:

#define TP_STATUS_AVAILABLE	0 // Frame is available
#define TP_STATUS_SEND_REQUEST	1 // Frame will be sent on next send()
#define TP_STATUS_SENDING	2 // Frame is currently in transmission
#define TP_STATUS_WRONG_FORMAT	4 // Frame format is not correct

First, the kernel initializes all frames to TP_STATUS_AVAILABLE. To send a packet, the user fills a data buffer of an available frame, sets tp_len to current data buffer size and sets its status field to TP_STATUS_SEND_REQUEST. This can be done on multiple frames. Once the user is ready to transmit, it calls send(). Then all buffers with status equal to TP_STATUS_SEND_REQUEST are forwarded to the network device. The kernel updates each status of sent frames with TP_STATUS_SENDING until the end of transfer.

At the end of each transfer, buffer status returns to TP_STATUS_AVAILABLE.

```

header->tp_len = in_i_size;
header->tp_status = TP_STATUS_SEND_REQUEST;
retval = send(this->socket, NULL, 0, 0);

```

The user can also use poll() to check if a buffer is available:

(status == TP_STATUS_SENDING)

```

struct pollfd pfd;
pfd.fd = fd;
pfd.revents = 0;
pfd.events = POLLOUT;
retval = poll(&pfd, 1, timeout);

```

78.9 What TPACKET versions are available and when to use them?

```
int val = tpocket_version;
setsockopt(fd, SOL_PACKET, PACKET_VERSION, &val, sizeof(val));
getsockopt(fd, SOL_PACKET, PACKET_VERSION, &val, sizeof(val));
```

where 'tpacket_version' can be TPACKET_V1 (default), TPACKET_V2, TPACKET_V3.

TPACKET_V1:

- Default if not otherwise specified by setsockopt(2)
- RX_RING, TX_RING available

TPACKET_V1 --> TPACKET_V2:

- Made 64 bit clean due to unsigned long usage in TPACKET_V1 structures, thus this also works on 64 bit kernel with 32 bit userspace and the like
- Timestamp resolution in nanoseconds instead of microseconds
- RX_RING, TX_RING available
- VLAN metadata information available for packets (TP_STATUS_VLAN_VALID, TP_STATUS_VLAN_TPID_VALID), in the tpocket2_hdr structure:
 - TP_STATUS_VLAN_VALID bit being set into the tp_status field indicates that the tp_vlan_tci field has valid VLAN TCI value
 - TP_STATUS_VLAN_TPID_VALID bit being set into the tp_status field indicates that the tp_vlan_tpid field has valid VLAN TPID value
- How to switch to TPACKET_V2:
 1. Replace struct tpocket_hdr by struct tpocket2_hdr
 2. Query header len and save
 3. Set protocol version to 2, set up ring as usual
 4. For getting the sockaddr_ll, use (void *)hdr + TPACKET_ALIGN(hdrlen) instead of (void *)hdr + TPACKET_ALIGN(sizeof(struct tpocket_hdr))

TPACKET_V2 --> TPACKET_V3:

- **Flexible buffer implementation for RX_RING:**
 1. Blocks can be configured with non-static frame-size
 2. Read/poll is at a block-level (as opposed to packet-level)
 3. Added poll timeout to avoid indefinite user-space wait on idle links
 4. Added user-configurable knobs:
 - 4.1 block::timeout
 - 4.2 tpkt_hdr::sk_rxhash
- RX Hash data available in user space
- TX_RING semantics are conceptually similar to TPACKET_V2; use tpocket3_hdr instead of tpocket2_hdr, and TPACKET3_HDRLEN instead of TPACKET2_HDRLEN. In the current implementation, the tp_next_offset field in the tpocket3_hdr MUST be set

to zero, indicating that the ring does not hold variable sized frames. Packets with non-zero values of tp_next_offset will be dropped.

78.10 AF_PACKET fanout mode

In the AF_PACKET fanout mode, packet reception can be load balanced among processes. This also works in combination with mmap(2) on packet sockets.

Currently implemented fanout policies are:

- PACKET_FANOUT_HASH: schedule to socket by skb's packet hash
- PACKET_FANOUT_LB: schedule to socket by round-robin
- PACKET_FANOUT_CPU: schedule to socket by CPU packet arrives on
- PACKET_FANOUT_RND: schedule to socket by random selection
- PACKET_FANOUT_ROLLOVER: if one socket is full, rollover to another
- PACKET_FANOUT_QM: schedule to socket by skbs recorded queue_mapping

Minimal example code by David S. Miller (try things like "./test eth0 hash", "./test eth0 lb", etc.):

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <sys/types.h>
#include <sys/wait.h>
#include <sys/socket.h>
#include <sys/ioctl.h>

#include <unistd.h>

#include <linux/if_ether.h>
#include <linux/if_packet.h>

#include <net/if.h>

static const char *device_name;
static int fanout_type;
static int fanout_id;

#ifndef PACKET_FANOUT
#define PACKET_FANOUT          18
#define PACKET_FANOUT_HASH      0
#define PACKET_FANOUT_LB        1
#endif

static int setup_socket(void)
{
```

```

int err, fd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_IP));
struct sockaddr_ll ll;
struct ifreq ifr;
int fanout_arg;

if (fd < 0) {
    perror("socket");
    return EXIT_FAILURE;
}

memset(&ifr, 0, sizeof(ifr));
strcpy(ifr.ifr_name, device_name);
err = ioctl(fd, SIOCGIFINDEX, &ifr);
if (err < 0) {
    perror("SIOCGIFINDEX");
    return EXIT_FAILURE;
}

memset(&ll, 0, sizeof(ll));
ll.sll_family = AF_PACKET;
ll.sll_ifindex = ifr.ifr_ifindex;
err = bind(fd, (struct sockaddr *) &ll, sizeof(ll));
if (err < 0) {
    perror("bind");
    return EXIT_FAILURE;
}

fanout_arg = (fanout_id | (fanout_type << 16));
err = setsockopt(fd, SOL_PACKET, PACKET_FANOUT,
                 &fanout_arg, sizeof(fanout_arg));
if (err) {
    perror("setsockopt");
    return EXIT_FAILURE;
}

return fd;
}

static void fanout_thread(void)
{
    int fd = setup_socket();
    int limit = 10000;

    if (fd < 0)
        exit(fd);

    while (limit-- > 0) {
        char buf[1600];
        int err;

```

```

        err = read(fd, buf, sizeof(buf));
        if (err < 0) {
                perror("read");
                exit(EXIT_FAILURE);
        }
        if ((limit % 10) == 0)
                fprintf(stdout, "(%d) \n", getpid());
}

fprintf(stdout, "%d: Received 10000 packets\n", getpid());

close(fd);
exit(0);
}

int main(int argc, char **argp)
{
    int fd, err;
    int i;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s INTERFACE {hash|lb}\n", argp[0]);
        return EXIT_FAILURE;
    }

    if (!strcmp(argp[2], "hash"))
        fanout_type = PACKET_FANOUT_HASH;
    else if (!strcmp(argp[2], "lb"))
        fanout_type = PACKET_FANOUT_LB;
    else {
        fprintf(stderr, "Unknown fanout type [%s]\n", argp[2]);
        exit(EXIT_FAILURE);
    }

    device_name = argp[1];
    fanout_id = getpid() & 0xffff;

    for (i = 0; i < 4; i++) {
        pid_t pid = fork();

        switch (pid) {
        case 0:
            fanout_thread();

        case -1:
            perror("fork");
            exit(EXIT_FAILURE);
        }
    }
}

```

```

        for (i = 0; i < 4; i++) {
            int status;

            wait(&status);
        }

        return 0;
}

```

78.11 AF_PACKET TPACKET_V3 example

AF_PACKET's TPACKET_V3 ring buffer can be configured to use non-static frame sizes by doing its own memory management. It is based on blocks where polling works on a per block basis instead of per ring as in TPACKET_V2 and predecessor.

It is said that TPACKET_V3 brings the following benefits:

- ~15% - 20% reduction in CPU-usage
- ~20% increase in packet capture rate
- ~2x increase in packet density
- Port aggregation analysis
- Non static frame size to capture entire packet payload

So it seems to be a good candidate to be used with packet fanout.

Minimal example code by Daniel Borkmann based on Chetan Loke's lolpcap (compile it with gcc -Wall -O2 blob.c, and try things like "./a.out eth0", etc.):

```

/* Written from scratch, but kernel-to-user space API usage
 * dissected from lolpcap:
 * Copyright 2011, Chetan Loke <loke.chetan@gmail.com>
 * License: GPL, version 2.0
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <assert.h>
#include <net/if.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <poll.h>
#include <unistd.h>
#include <signal.h>
#include <inttypes.h>
#include <sys/socket.h>
#include <sys/mman.h>
#include <linux/if_packet.h>

```

```

#include <linux/if_ether.h>
#include <linux/ip.h>

#ifndef likely
# define likely(x)      __builtin_expect(!!(x), 1)
#endif
#ifndef unlikely
# define unlikely(x)    __builtin_expect(!!(x), 0)
#endif

struct block_desc {
    uint32_t version;
    uint32_t offset_to_priv;
    struct tpacket_hdr_v1 h1;
};

struct ring {
    struct iovec *rd;
    uint8_t *map;
    struct tpacket_req3 req;
};
static unsigned long packets_total = 0, bytes_total = 0;
static sig_atomic_t sigint = 0;

static void sighandler(int num)
{
    sigint = 1;
}

static int setup_socket(struct ring *ring, char *netdev)
{
    int err, i, fd, v = TPACKET_V3;
    struct sockaddr_ll ll;
    unsigned int blocksiz = 1 << 22, framesiz = 1 << 11;
    unsigned int blocknum = 64;

    fd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
    if (fd < 0) {
        perror("socket");
        exit(1);
    }

    err = setsockopt(fd, SOL_PACKET, PACKET_VERSION, &v, sizeof(v));
    if (err < 0) {
        perror("setsockopt");
        exit(1);
    }

    memset(&ring->req, 0, sizeof(ring->req));
}

```

```

ring->req.tp_block_size = blocksiz;
ring->req.tp_frame_size = framesiz;
ring->req.tp_block_nr = blocknum;
ring->req.tp_frame_nr = (blocksiz * blocknum) / framesiz;
ring->req.tp_retire_blk_tov = 60;
ring->req.tp_feature_req_word = TP_FT_REQ_FILL_RXHASH;

err = setsockopt(fd, SOL_PACKET, PACKET_RX_RING, &ring->req,
                 sizeof(ring->req));
if (err < 0) {
    perror("setsockopt");
    exit(1);
}

ring->map = mmap(NULL, ring->req.tp_block_size * ring->req.tp_block_nr,
                  PROT_READ | PROT_WRITE, MAP_SHARED | MAP_LOCKED, fd, u
-0);
if (ring->map == MAP_FAILED) {
    perror("mmap");
    exit(1);
}

ring->rd = malloc(ring->req.tp_block_nr * sizeof(*ring->rd));
assert(ring->rd);
for (i = 0; i < ring->req.tp_block_nr; ++i) {
    ring->rd[i].iov_base = ring->map + (i * ring->req.tp_block_
-size);
    ring->rd[i].iov_len = ring->req.tp_block_size;
}

memset(&ll, 0, sizeof(ll));
ll.sll_family = PF_PACKET;
ll.sll_protocol = htons(ETH_P_ALL);
ll.sll_ifindex = if_nametoindex(netdev);
ll.sll_hatype = 0;
ll.sll_pktype = 0;
ll.sll_halen = 0;

err = bind(fd, (struct sockaddr *) &ll, sizeof(ll));
if (err < 0) {
    perror("bind");
    exit(1);
}

return fd;
}

static void display(struct tpacket3_hdr *ppd)
{
    struct ethhdr *eth = (struct ethhdr *) ((uint8_t *) ppd + ppd->tp_mac);

```

```

        struct iphdr *ip = (struct iphdr *) ((uint8_t *) eth + ETH_HLEN);

        if (eth->h_proto == htons(ETH_P_IP)) {
                struct sockaddr_in ss, sd;
                char sbuf[NI_MAXHOST], dbuf[NI_MAXHOST];

                memset(&ss, 0, sizeof(ss));
                ss.sin_family = PF_INET;
                ss.sin_addr.s_addr = ip->saddr;
                getnameinfo((struct sockaddr *) &ss, sizeof(ss),
                            sbuf, sizeof(sbuf), NULL, 0, NI_NUMERICHOST);

                memset(&sd, 0, sizeof(sd));
                sd.sin_family = PF_INET;
                sd.sin_addr.s_addr = ip->daddr;
                getnameinfo((struct sockaddr *) &sd, sizeof(sd),
                            dbuf, sizeof(dbuf), NULL, 0, NI_NUMERICHOST);

                printf("%s -> %s, ", sbuf, dbuf);
        }

        printf("rxhash: 0x%x\n", ppd->hv1.tp_rxhash);
}

static void walk_block(struct block_desc *pbd, const int block_num)
{
        int num_pkts = pbd->h1.num_pkts, i;
        unsigned long bytes = 0;
        struct tpacket3_hdr *ppd;

        ppd = (struct tpacket3_hdr *) ((uint8_t *) pbd +
                                       pbd->h1.offset_to_first_pkt);
        for (i = 0; i < num_pkts; ++i) {
                bytes += ppd->tp_snaplen;
                display(ppd);

                ppd = (struct tpacket3_hdr *) ((uint8_t *) ppd +
                                               ppd->tp_next_offset);
        }

        packets_total += num_pkts;
        bytes_total += bytes;
}

static void flush_block(struct block_desc *pbd)
{
        pbd->h1.block_status = TP_STATUS_KERNEL;
}

static void teardown_socket(struct ring *ring, int fd)

```

```
{
    munmap(ring->map, ring->req.tp_block_size * ring->req.tp_block_nr);
    free(ring->rd);
    close(fd);
}

int main(int argc, char **argp)
{
    int fd, err;
    socklen_t len;
    struct ring ring;
    struct pollfd pfd;
    unsigned int block_num = 0, blocks = 64;
    struct block_desc *pbd;
    struct tpacket_stats_v3 stats;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s INTERFACE\n", argp[0]);
        return EXIT_FAILURE;
    }

    signal(SIGINT, sighandler);

    memset(&ring, 0, sizeof(ring));
    fd = setup_socket(&ring, argp[argc - 1]);
    assert(fd > 0);

    memset(&pfd, 0, sizeof(pfd));
    pfd.fd = fd;
    pfd.events = POLLIN | POLLERR;
    pfd.revents = 0;

    while (likely(!sigint)) {
        pbd = (struct block_desc *) ring.rd[block_num].iov_base;

        if (((pbd->h1.block_status & TP_STATUS_USER) == 0) {
            poll(&pfd, 1, -1);
            continue;
        }

        walk_block(pbd, block_num);
        flush_block(pbd);
        block_num = (block_num + 1) % blocks;
    }

    len = sizeof(stats);
    err = getsockopt(fd, SOL_PACKET, PACKET_STATISTICS, &stats, &len);
    if (err < 0) {
        perror("getsockopt");
        exit(1);
    }
}
```

```

    }

    fflush(stdout);
    printf("\nReceived %u packets, %lu bytes, %u dropped, freeze_q_cnt: %u\
→n",
           stats.tp_packets, bytes_total, stats.tp_drops,
           stats.tp_freeze_q_cnt);

    teardown_socket(&ring, fd);
    return 0;
}

```

78.12 PACKET_QDISC_BYPASS

If there is a requirement to load the network with many packets in a similar fashion as pktgen does, you might set the following option after socket creation:

```
int one = 1;
setsockopt(fd, SOL_PACKET, PACKET_QDISC_BYPASS, &one, sizeof(one));
```

This has the side-effect, that packets sent through PF_PACKET will bypass the kernel's qdisc layer and are forcedly pushed to the driver directly. Meaning, packet are not buffered, tc disciplines are ignored, increased loss can occur and such packets are also not visible to other PF_PACKET sockets anymore. So, you have been warned; generally, this can be useful for stress testing various components of a system.

On default, PACKET_QDISC_BYPASS is disabled and needs to be explicitly enabled on PF_PACKET sockets.

78.13 PACKET_TIMESTAMP

The PACKET_TIMESTAMP setting determines the source of the timestamp in the packet meta information for mmap(2)ed RX_RING and TX_RINGS. If your NIC is capable of timestamping packets in hardware, you can request those hardware timestamps to be used. Note: you may need to enable the generation of hardware timestamps with SIOCSHWTSTAMP (see related information from [Timestamping](#)).

PACKET_TIMESTAMP accepts the same integer bit field as SO_TIMESTAMPING:

```
int req = SOF_TIMESTAMPING_RAW_HARDWARE;
setsockopt(fd, SOL_PACKET, PACKET_TIMESTAMP, (void *) &req, sizeof(req))
```

For the mmap(2)ed ring buffers, such timestamps are stored in the tpacket{,2,3}_hdr structure's tp_sec and tp_{n,u}sec members. To determine what kind of timestamp has been reported, the tp_status field is binary or'ed with the following possible bits ...

```
TP_STATUS_TS_RAW_HARDWARE
TP_STATUS_TS_SOFTWARE
```

... that are equivalent to its `SOF_TIMESTAMPING_*` counterparts. For the `RX_RING`, if neither is set (i.e. `PACKET_TIMESTAMP` is not set), then a software fallback was invoked *within* `PF_PACKET`'s processing code (less precise).

Getting timestamps for the `TX_RING` works as follows: i) fill the ring frames, ii) call `sendto()` e.g. in blocking mode, iii) wait for status of relevant frames to be updated resp. the frame handed over to the application, iv) walk through the frames to pick up the individual hw/sw timestamps.

Only (!) if transmit timestamping is enabled, then these bits are combined with binary | with `TP_STATUS_AVAILABLE`, so you must check for that in your application (e.g. `!(tp_status & (TP_STATUS_SEND_REQUEST | TP_STATUS_SENDING))`) in a first step to see if the frame belongs to the application, and then one can extract the type of timestamp in a second step from `tp_status`!

If you don't care about them, thus having it disabled, checking for `TP_STATUS_AVAILABLE` resp. `TP_STATUS_WRONG_FORMAT` is sufficient. If in the `TX_RING` part only `TP_STATUS_AVAILABLE` is set, then the `tp_sec` and `tp_{n,u}sec` members do not contain a valid value. For `TX_RINGS`, by default no timestamp is generated!

See `include/linux/net_tstamp.h` and [Timestamping](#) for more information on hardware timestamps.

78.14 Miscellaneous bits

- Packet sockets work well together with Linux socket filters, thus you also might want to have a look at [Linux Socket Filtering aka Berkeley Packet Filter \(BPF\)](#)

78.15 THANKS

Jesse Brandenburg, for fixing my grammatical/spelling errors

LINUX PHONET PROTOCOL FAMILY

79.1 Introduction

Phonet is a packet protocol used by Nokia cellular modems for both IPC and RPC. With the Linux Phonet socket family, Linux host processes can receive and send messages from/to the modem, or any other external device attached to the modem. The modem takes care of routing.

Phonet packets can be exchanged through various hardware connections depending on the device, such as:

- USB with the CDC Phonet interface,
- infrared,
- Bluetooth,
- an RS232 serial port (with a dedicated "FBUS" line discipline),
- the SSI bus with some TI OMAP processors.

79.2 Packets format

Phonet packets have a common header as follows:

```
struct phonethdr {  
    uint8_t  pn_media; /* Media type (link-layer identifier) */  
    uint8_t  pn_rdev;  /* Receiver device ID */  
    uint8_t  pn_sdev;  /* Sender device ID */  
    uint8_t  pn_res;   /* Resource ID or function */  
    uint16_t pn_length; /* Big-endian message byte length (minus 6) */  
    uint8_t  pn_robj;  /* Receiver object ID */  
    uint8_t  pn_sobj;  /* Sender object ID */  
};
```

On Linux, the link-layer header includes the pn_media byte (see below). The next 7 bytes are part of the network-layer header.

The device ID is split: the 6 higher-order bits constitute the device address, while the 2 lower-order bits are used for multiplexing, as are the 8-bit object identifiers. As such, Phonet can be considered as a network layer with 6 bits of address space and 10 bits for transport protocol (much like port numbers in IP world).

The modem always has address number zero. All other device have a their own 6-bit address.

79.3 Link layer

Phonet links are always point-to-point links. The link layer header consists of a single Phonet media type byte. It uniquely identifies the link through which the packet is transmitted, from the modem's perspective. Each Phonet network device shall prepend and set the media type byte as appropriate. For convenience, a common `phonet_header_ops` link-layer header operations structure is provided. It sets the media type according to the network device hardware address.

Linux Phonet network interfaces support a dedicated link layer packets type (`ETH_P_PHONET`) which is out of the Ethernet type range. They can only send and receive Phonet packets.

The virtual TUN tunnel device driver can also be used for Phonet. This requires `IFF_TUN` mode, `_without_` the `IFF_NO_PI` flag. In this case, there is no link-layer header, so there is no Phonet media type byte.

Note that Phonet interfaces are not allowed to re-order packets, so only the (default) Linux FIFO qdisc should be used with them.

79.4 Network layer

The Phonet socket address family maps the Phonet packet header:

```
struct sockaddr_pn {
    sa_family_t spn_family;      /* AF_PHONET */
    uint8_t     spn_obj;         /* Object ID */
    uint8_t     spn_dev;         /* Device ID */
    uint8_t     spn_resource;    /* Resource or function */
    uint8_t     spn_zero[...];   /* Padding */
};
```

The resource field is only used when sending and receiving; It is ignored by `bind()` and `getsockname()`.

79.5 Low-level datagram protocol

Applications can send Phonet messages using the Phonet datagram socket protocol from the `PF_PHONET` family. Each socket is bound to one of the 2^{10} object IDs available, and can send and receive packets with any other peer.

```
struct sockaddr_pn addr = { .spn_family = AF_PHONET, };
ssize_t len;
socklen_t addrlen = sizeof(addr);
int fd;

fd = socket(PF_PHONET, SOCK_DGRAM, 0);
bind(fd, (struct sockaddr *)&addr, sizeof(addr));
/* ... */

sendto(fd, msg, msglen, 0, (struct sockaddr *)&addr, sizeof(addr));
```

```
len = recvfrom(fd, buf, sizeof(buf), 0,
               (struct sockaddr *)&addr, &addrlen);
```

This protocol follows the SOCK_DGRAM connection-less semantics. However, connect() and getpeername() are not supported, as they did not seem useful with Phonet usages (could be added easily).

79.6 Resource subscription

A Phonet datagram socket can be subscribed to any number of 8-bits Phonet resources, as follow:

```
uint32_t res = 0xXX;
ioctl(fd, SIOCPNADDRESOURCE, &res);
```

Subscription is similarly cancelled using the SIOCPNDELRESOURCE I/O control request, or when the socket is closed.

Note that no more than one socket can be subscribed to any given resource at a time. If not, ioctl() will return EBUSY.

79.7 Phonet Pipe protocol

The Phonet Pipe protocol is a simple sequenced packets protocol with end-to-end congestion control. It uses the passive listening socket paradigm. The listening socket is bound to an unique free object ID. Each listening socket can handle up to 255 simultaneous connections, one per accept()'d socket.

```
int lfd, cfd;

lfd = socket(PF_PHONET, SOCK_SEQPACKET, PN_PROTO_PIPE);
listen (lfd, INT_MAX);

/* ... */
cfid = accept(lfd, NULL, NULL);
for (;;)
{
    char buf[...];
    ssize_t len = read(cfid, buf, sizeof(buf));

    /* ... */

    write(cfid, msg, msglen);
}
```

Connections are traditionally established between two endpoints by a "third party" application. This means that both endpoints are passive.

As of Linux kernel version 2.6.39, it is also possible to connect two endpoints directly, using connect() on the active side. This is intended to support the newer Nokia Wireless Modem API, as found in e.g. the Nokia Slim Modem in the ST-Ericsson U8500 platform:

```
struct sockaddr_spn spn;
int fd;

fd = socket(PF_PHONET, SOCK_SEQPACKET, PN_PROTO_PIPE);
memset(&spn, 0, sizeof(spn));
spn.spn_family = AF_PHONET;
spn.spn_obj = ...;
spn.spn_dev = ...;
spn.spn_resource = 0xD9;
connect(fd, (struct sockaddr *)&spn, sizeof(spn));
/* normal I/O here ... */
close(fd);
```

The pipe protocol provides two socket options at the SOL_PNPipe level:

PNPIPE_ENCAP accepts one integer value (int) of:

PNPIPE_ENCAP_NONE:

The socket operates normally (default).

PNPIPE_ENCAP_IP:

The socket is used as a backend for a virtual IP interface. This requires CAP_NET_ADMIN capability. GPRS data support on Nokia modems can use this. Note that the socket cannot be reliably poll()'d or read() from while in this mode.

PNPIPE_IFINDEX

is a read-only integer value. It contains the interface index of the network interface created by PNPIPE_ENCAP, or zero if encapsulation is off.

PNPIPE_HANDLE

is a read-only integer value. It contains the underlying identifier ("pipe handle") of the pipe. This is only defined for socket descriptors that are already connected or being connected.

79.8 Authors

Linux Phonet was initially written by Sakari Ailus.

Other contributors include Mikä Liljeberg, Andras Domokos, Carlos Chinea and Rémi Denis-Courmont.

Copyright © 2008 Nokia Corporation.

HOWTO FOR THE LINUX PACKET GENERATOR

Enable CONFIG_NET_PKTGEN to compile and build pktgen either in-kernel or as a module. A module is preferred; modprobe pktgen if needed. Once running, pktgen creates a thread for each CPU with affinity to that CPU. Monitoring and controlling is done via /proc. It is easiest to select a suitable sample script and configure that.

On a dual CPU:

```
ps aux | grep pkt
root      129  0.3  0.0      0      0 ?          SW    2003 523:20 [kpktgend_0]
root      130  0.3  0.0      0      0 ?          SW    2003 509:50 [kpktgend_1]
```

For monitoring and control pktgen creates:

```
/proc/net/pktgen/pgctrl
/proc/net/pktgen/kpktgend_X
/proc/net/pktgen/ethX
```

80.1 Tuning NIC for max performance

The default NIC settings are (likely) not tuned for pktgen's artificial overload type of benchmarking, as this could hurt the normal use-case.

Specifically increasing the TX ring buffer in the NIC:

```
# ethtool -G ethX tx 1024
```

A larger TX ring can improve pktgen's performance, while it can hurt in the general case, 1) because the TX ring buffer might get larger than the CPU's L1/L2 cache, 2) because it allows more queueing in the NIC HW layer (which is bad for bufferbloat).

One should hesitate to conclude that packets/descriptors in the HW TX ring cause delay. Drivers usually delay cleaning up the ring-buffers for various performance reasons, and packets stalling the TX ring might just be waiting for cleanup.

This cleanup issue is specifically the case for the driver ixgbe (Intel 82599 chip). This driver (ixgbe) combines TX+RX ring cleanups, and the cleanup interval is affected by the ethtool --coalesce setting of parameter "rx-usecs".

For ixgbe use e.g. "30" resulting in approx 33K interrupts/sec ($1/30*10^6$):

```
# ethtool -C ethX rx-usecs 30
```

80.2 Kernel threads

Pktgen creates a thread for each CPU with affinity to that CPU. Which is controlled through procfile /proc/net/pktgen/kpktgend_X.

Example: /proc/net/pktgen/kpktgend_0:

```
Running:  
Stopped: eth4@0  
Result: 0K: add_device=eth4@0
```

Most important are the devices assigned to the thread.

The two basic thread commands are:

- add_device DEVICE@NAME -- adds a single device
- rem_device_all -- remove all associated devices

When adding a device to a thread, a corresponding procfile is created which is used for configuring this device. Thus, device names need to be unique.

To support adding the same device to multiple threads, which is useful with multi queue NICs, the device naming scheme is extended with "@": device@something

The part after "@" can be anything, but it is custom to use the thread number.

80.3 Viewing devices

The Params section holds configured information. The Current section holds running statistics. The Result is printed after a run or after interruption. Example:

```
/proc/net/pktgen/eth4@0

Params: count 100000 min_pkt_size: 60 max_pkt_size: 60
        frags: 0 delay: 0 clone_skb: 64 ifname: eth4@0
        flows: 0 flowlen: 0
        queue_map_min: 0 queue_map_max: 0
        dst_min: 192.168.81.2 dst_max:
        src_min: src_max:
        src_mac: 90:e2:ba:0a:56:b4 dst_mac: 00:1b:21:3c:9d:f8
        udp_src_min: 9 udp_src_max: 109 udp_dst_min: 9 udp_dst_max: 9
        src_mac_count: 0 dst_mac_count: 0
        Flags: UDPSRC_RND NO_TIMESTAMP QUEUE_MAP_CPU

Current:
        pkts-sofar: 100000 errors: 0
        started: 623913381008us stopped: 623913396439us idle: 25us
        seq_num: 100001 cur_dst_mac_offset: 0 cur_src_mac_offset: 0
        cur_saddr: 192.168.8.3 cur_daddr: 192.168.81.2
```

```

cur_udp_dst: 9 cur_udp_src: 42
cur_queue_map: 0
flows: 0
Result: OK: 15430(c15405+d25) usec, 100000 (60byte,0frags)
6480562pps 3110Mb/sec (3110669760bps) errors: 0

```

80.4 Configuring devices

This is done via the /proc interface, and most easily done via pgset as defined in the sample scripts. You need to specify PGDEV environment variable to use functions from sample scripts, i.e.:

```

export PGDEV=/proc/net/pktgen/eth4@0
source samples/pktgen/functions.sh

```

Examples:

pg_ctrl start	starts injection.
pg_ctrl stop	aborts injection. Also, ^C aborts generator.
pgset "clone_skb 1"	sets the number of copies of the same packet
pgset "clone_skb 0"	use single SKB for all transmits
pgset "burst 8"	uses xmit_more API to queue 8 copies of the same packet and update HW tx queue tail pointer once. "burst 1" is the default
pgset "pkt_size 9014"	sets packet size to 9014
pgset "frags 5"	packet will consist of 5 fragments
pgset "count 200000"	sets number of packets to send, set to zero for continuous sends until explicitly stopped.
pgset "delay 5000"	adds delay to hard_start_xmit(). nanoseconds
pgset "dst 10.0.0.1"	sets IP destination address (BEWARE! This generator is very aggressive!)
pgset "dst_min 10.0.0.1"	Same as dst
pgset "dst_max 10.0.0.254"	Set the maximum destination IP.
pgset "src_min 10.0.0.1"	Set the minimum (or only) source IP.
pgset "src_max 10.0.0.254"	Set the maximum source IP.
pgset "dst6 fec0::1"	IPV6 destination address
pgset "src6 fec0::2"	IPV6 source address
pgset "dstmac 00:00:00:00:00:00"	sets MAC destination address
pgset "srcmac 00:00:00:00:00:00"	sets MAC source address
pgset "queue_map_min 0"	Sets the min value of tx queue interval
pgset "queue_map_max 7"	Sets the max value of tx queue interval, for ↴ multiqueue devices
	To select queue 1 of a given device, use queue_map_min=1 and queue_map_max=1

```
pgset "src_mac_count 1" Sets the number of MACs we'll range through.  
The 'minimum' MAC is what you set with srcmac.  
  
pgset "dst_mac_count 1" Sets the number of MACs we'll range through.  
The 'minimum' MAC is what you set with dstmac.  
  
pgset "flag [name]"      Set a flag to determine behaviour. Current flags  
                        are: IPSRC_RND # IP source is random (between min/max)  
                           IPDST_RND # IP destination is random  
                           UDPSRC_RND, UDPDST_RND,  
                           MACSRC_RND, MACDST_RND  
                           TXSIZE_RND, IPV6,  
                           MPLS_RND, VID_RND, SVID_RND  
                           FLOW_SEQ,  
                           QUEUE_MAP_RND # queue map random  
                           QUEUE_MAP_CPU # queue map mirrors smp_processor_  
→id()  
                           UDPCSUM,  
                           IPSEC # IPsec encapsulation (needs CONFIG_XFRM)  
                           NODE_ALLOC # node specific memory allocation  
                           NO_TIMESTAMP # disable timestamping  
                           SHARED # enable shared SKB  
pgset 'flag ![name]'    Clear a flag to determine behaviour.  
                        Note that you might need to use single quote in  
                        interactive mode, so that your shell wouldn't expand  
                        the specified flag as a history command.  
  
pgset "spi [SPI_VALUE]" Set specific SA used to transform packet.  
  
pgset "udp_src_min 9"   set UDP source port min, If < udp_src_max, then  
                        cycle through the port range.  
  
pgset "udp_src_max 9"   set UDP source port max.  
pgset "udp_dst_min 9"   set UDP destination port min, If < udp_dst_max, then  
                        cycle through the port range.  
pgset "udp_dst_max 9"   set UDP destination port max.  
  
pgset "mpls 0001000a,0002000a,0000000a" set MPLS labels (in this example  
                                         outer label=16,middle label=32,  
                                         inner label=0 (IPv4 NULL)) Note that  
                                         there must be no spaces between the  
                                         arguments. Leading zeros are required.  
                                         Do not set the bottom of stack bit,  
                                         that's done automatically. If you do  
                                         set the bottom of stack bit, that  
                                         indicates that you want to randomly  
                                         generate that address and the flag  
                                         MPLS_RND will be turned on. You  
                                         can have any mix of random and fixed
```

```

labels in the label stack.

pgset "mpls 0"           turn off mpls (or any invalid argument works too!)

pgset "vlan_id 77"        set VLAN ID 0-4095
pgset "vlan_p 3"          set priority bit 0-7 (default 0)
pgset "vlan_cfi 0"        set canonical format identifier 0-1 (default 0)

pgset "svlan_id 22"       set SVLAN ID 0-4095
pgset "svlan_p 3"          set priority bit 0-7 (default 0)
pgset "svlan_cfi 0"        set canonical format identifier 0-1 (default 0)

pgset "vlan_id 9999"      > 4095 remove vlan and svlan tags
pgset "svlan 9999"         > 4095 remove svlan tag

pgset "tos XX"            set former IPv4 TOS field (e.g. "tos 28" for AF11 no_U
                           ↳ ECN, default 00)
pgset "traffic_class XX"  set former IPv6 TRAFFIC CLASS (e.g. "traffic_class B8
                           ↳" for EF no ECN, default 00)

pgset "rate 300M"          set rate to 300 Mb/s
pgset "ratep 1000000"      set rate to 1Mpps

pgset "xmit_mode netif_receive" RX inject into stack netif_receive_skb()
                               Works with "burst" but not with "clone_skb".
                               Default xmit_mode is "start_xmit".

```

80.5 Sample scripts

A collection of tutorial scripts and helpers for pktgen is in the samples/pktgen directory. The helper parameters.sh file support easy and consistent parameter parsing across the sample scripts.

Usage example and help:

```
./pktgen_sample01_simple.sh -i eth4 -m 00:1B:21:3C:9D:F8 -d 192.168.8.2
```

Usage::

```
./pktgen_sample01_simple.sh [-vx] -i ethX

-i : ($DEV)          output interface/device (required)
-s : ($PKT_SIZE)    packet size
-d : ($DEST_IP)     destination IP. CIDR (e.g. 198.18.0.0/15) is also allowed
-m : ($DST_MAC)     destination MAC-addr
-p : ($DST_PORT)    destination PORT range (e.g. 433-444) is also allowed
-t : ($THREADS)     threads to start
-f : ($F_THREAD)    index of first thread (zero indexed CPU number)
-c : ($SKB_CLONE)   SKB clones send before alloc new SKB
```

```
-n : ($COUNT)      num messages to send per thread, 0 means indefinitely
-b : ($BURST)     HW level bursting of SKBs
-v : ($VERBOSE)   verbose
-x : ($DEBUG)     debug
-6 : ($IP6)       IPv6
-w : ($DELAY)     Tx Delay value (ns)
-a : ($APPEND)    Script will not reset generator's state, but will append its u
-config
```

The global variables being set are also listed. E.g. the required interface/device parameter "-i" sets variable \$DEV. Copy the pktgen_sampleXX scripts and modify them to fit your own needs.

80.6 Interrupt affinity

Note that when adding devices to a specific CPU it is a good idea to also assign /proc/irq/XX/smp_affinity so that the TX interrupts are bound to the same CPU. This reduces cache bouncing when freeing skbs.

Plus using the device flag QUEUE_MAP_CPU, which maps the SKBs TX queue to the running threads CPU (directly from smp_processor_id()).

80.7 Enable IPsec

Default IPsec transformation with ESP encapsulation plus transport mode can be enabled by simply setting:

```
pgset "flag IPSEC"
pgset "flows 1"
```

To avoid breaking existing testbed scripts for using AH type and tunnel mode, you can use "pgset spi SPI_VALUE" to specify which transformation mode to employ.

80.8 Disable shared SKB

By default, SKBs sent by pktgen are shared (user count > 1). To test with non-shared SKBs, remove the "SHARED" flag by simply setting:

```
pg_set "flag !SHARED"
```

However, if the "clone_skb" or "burst" parameters are configured, the skb still needs to be held by pktgen for further access. Hence the skb must be shared.

80.9 Current commands and configuration options

Pgcontrol commands:

```
start  
stop  
reset
```

Thread commands:

```
add_device  
rem_device_all
```

Device commands:

```
count  
clone_skb  
burst  
debug  
  
frags  
delay  
  
src_mac_count  
dst_mac_count  
  
pkt_size  
min_pkt_size  
max_pkt_size  
  
queue_map_min  
queue_map_max  
skb_priority  
  
tos          (ipv4)  
traffic_class (ipv6)  
  
mpls  
  
udp_src_min  
udp_src_max  
  
udp_dst_min  
udp_dst_max  
  
node  
  
flag  
IPSRC_RND  
IPDST_RND  
UDPSRC_RND
```

```
UDPDST_RND
MACSRC_RND
MACDST_RND
TXSIZE_RND
IPV6
MPLS_RND
VID_RND
SVID_RND
FLOW_SEQ
QUEUE_MAP_RND
QUEUE_MAP_CPU
UDPCSUM
IPSEC
NODE_ALLOC
NO_TIMESTAMP
SHARED

spi (ipsec)

dst_min
dst_max

src_min
src_max

dst_mac
src_mac

clear_counters

src6
dst6
dst6_max
dst6_min

flows
flowlen

rate
ratep

xmit_mode <start_xmit|netif_receive>

vlan_cfi
vlan_id
vlan_p

svlan_cfi
svlan_id
svlan_p
```

References:

- <ftp://robur.slu.se/pub/Linux/net-development/pktgen-testing/>
- <ftp://robur.slu.se/pub/Linux/net-development/pktgen-testing/examples/>

Paper from Linux-Kongress in Erlangen 2004. - ftp://robur.slu.se/pub/Linux/net-development/pktgen-testing/pktgen_paper.pdf

Thanks to:

Grant Grundler for testing on IA-64 and parisc, Harald Welte, Lennert Buytenhek Stephen Hemminger, Andi Kleen, Dave Miller and many others.

Good luck with the linux net-development.

PLIP: THE PARALLEL LINE INTERNET PROTOCOL DEVICE

Donald Becker (becker@super.org) I.D.A. Supercomputing Research Center, Bowie MD 20715
At some point T. Thorn will probably contribute text, Tommy Thorn (tthorn@daimi.aau.dk)

81.1 PLIP Introduction

This document describes the parallel port packet pusher for Net/LGX. This device interface allows a point-to-point connection between two parallel ports to appear as a IP network interface.

81.1.1 What is PLIP?

PLIP is Parallel Line IP, that is, the transportation of IP packages over a parallel port. In the case of a PC, the obvious choice is the printer port. PLIP is a non-standard, but [can use] uses the standard LapLink null-printer cable [can also work in turbo mode, with a PLIP cable]. [The protocol used to pack IP packages, is a simple one initiated by Crynwr.]

81.1.2 Advantages of PLIP

It's cheap, it's available everywhere, and it's easy.

The PLIP cable is all that's needed to connect two Linux boxes, and it can be built for very few bucks.

Connecting two Linux boxes takes only a second's decision and a few minutes' work, no need to search for a [supported] netcard. This might even be especially important in the case of notebooks, where netcards are not easily available.

Not requiring a netcard also means that apart from connecting the cables, everything else is software configuration [which in principle could be made very easy.]

81.1.3 Disadvantages of PLIP

Doesn't work over a modem, like SLIP and PPP. Limited range, 15 m. Can only be used to connect three (?) Linux boxes. Doesn't connect to an existing Ethernet. Isn't standard (not even de facto standard, like SLIP).

81.1.4 Performance

PLIP easily outperforms Ethernet cards....(ups, I was dreaming, but it *is* getting late. EOB)

81.2 PLIP driver details

The Linux PLIP driver is an implementation of the original Crynwr protocol, that uses the parallel port subsystem of the kernel in order to properly share parallel ports between PLIP and other services.

81.2.1 IRQs and trigger timeouts

When a parallel port used for a PLIP driver has an IRQ configured to it, the PLIP driver is signaled whenever data is sent to it via the cable, such that when no data is available, the driver isn't being used.

However, on some machines it is hard, if not impossible, to configure an IRQ to a certain parallel port, mainly because it is used by some other device. On these machines, the PLIP driver can be used in IRQ-less mode, where the PLIP driver would constantly poll the parallel port for data waiting, and if such data is available, process it. This mode is less efficient than the IRQ mode, because the driver has to check the parallel port many times per second, even when no data at all is sent. Some rough measurements indicate that there isn't a noticeable performance drop when using IRQ-less mode as compared to IRQ mode as far as the data transfer speed is involved. There is a performance drop on the machine hosting the driver.

When the PLIP driver is used in IRQ mode, the timeout used for triggering a data transfer (the maximal time the PLIP driver would allow the other side before announcing a timeout, when trying to handshake a transfer of some data) is, by default, 500usec. As IRQ delivery is more or less immediate, this timeout is quite sufficient.

When in IRQ-less mode, the PLIP driver polls the parallel port HZ times per second (where HZ is typically 100 on most platforms, and 1024 on an Alpha, as of this writing). Between two such polls, there are $10^6/HZ$ usecs. On an i386, for example, $10^6/100 = 10000$ usec. It is easy to see that it is quite possible for the trigger timeout to expire between two such polls, as the timeout is only 500usec long. As a result, it is required to change the trigger timeout on the *other* side of a PLIP connection, to about $10^6/HZ$ usecs. If both sides of a PLIP connection are used in IRQ-less mode, this timeout is required on both sides.

It appears that in practice, the trigger timeout can be shorter than in the above calculation. It isn't an important issue, unless the wire is faulty, in which case a long timeout would stall the machine when, for whatever reason, bits are dropped.

A utility that can perform this change in Linux is plipconfig, which is part of the net-tools package (its location can be found in the Documentation/Changes file). An example command would be 'plipconfig plipX trigger 10000', where plipX is the appropriate PLIP device.

81.3 PLIP hardware interconnection

PLIP uses several different data transfer methods. The first (and the only one implemented in the early version of the code) uses a standard printer "null" cable to transfer data four bits at a time using data bit outputs connected to status bit inputs.

The second data transfer method relies on both machines having bi-directional parallel ports, rather than output-only printer ports. This allows byte-wide transfers and avoids reconstructing nibbles into bytes, leading to much faster transfers.

81.3.1 Parallel Transfer Mode 0 Cable

The cable for the first transfer mode is a standard printer "null" cable which transfers data four bits at a time using data bit outputs of the first port (machine T) connected to the status bit inputs of the second port (machine R). There are five status inputs, and they are used as four data inputs and a clock (data strobe) input, arranged so that the data input bits appear as contiguous bits with standard status register implementation.

A cable that implements this protocol is available commercially as a "Null Printer" or "Turbo Laplink" cable. It can be constructed with two DB-25 male connectors symmetrically connected as follows:

STROBE output	1*	
D0->ERROR	2 - 15	15 - 2
D1->SLCT	3 - 13	13 - 3
D2->PAPOUT	4 - 12	12 - 4
D3->ACK	5 - 10	10 - 5
D4->BUSY	6 - 11	11 - 6
D5,D6,D7	are 7*, 8*, 9*	
AUTOFD output	14*	
INIT output	16*	
SLCTIN	17 - 17	
extra grounds	are 18*, 19*, 20*, 21*, 22*, 23*, 24*	
GROUND	25 - 25	

* Do not connect these pins on either end

If the cable you are using has a metallic shield it should be connected to the metallic DB-25 shell at one end only.

81.3.2 Parallel Transfer Mode 1

The second data transfer method relies on both machines having bi-directional parallel ports, rather than output-only printer ports. This allows byte-wide transfers, and avoids reconstructing nibbles into bytes. This cable should not be used on unidirectional printer (as opposed to parallel) ports or when the machine isn't configured for PLIP, as it will result in output driver conflicts and the (unlikely) possibility of damage.

The cable for this transfer mode should be constructed as follows:

```

STROBE->BUSY 1 - 11
D0->D0      2 - 2
D1->D1      3 - 3
D2->D2      4 - 4
D3->D3      5 - 5
D4->D4      6 - 6
D5->D5      7 - 7
D6->D6      8 - 8
D7->D7      9 - 9
INIT -> ACK 16 - 10
AUTOFD->PAPOUT 14 - 12
SLCT->SLCTIN 13 - 17
GND->ERROR 18 - 15
extra grounds are 19*,20*,21*,22*,23*,24*
GROUND      25 - 25

```

* Do not connect these pins on either end

Once again, if the cable you are using has a metallic shield it should be connected to the metallic DB-25 shell at one end only.

81.3.3 PLIP Mode 0 transfer protocol

The PLIP driver is compatible with the "Crynwyr" parallel port transfer standard in Mode 0. That standard specifies the following protocol:

```

send header nibble '0x8'
count-low octet
count-high octet
... data octets
checksum octet

```

Each octet is sent as:

```

<wait for rx. '0x1?'>  <send 0x10+(octet&0x0F)>
<wait for rx. '0x0?'>  <send 0x00+((octet>>4)&0x0F)>

```

To start a transfer the transmitting machine outputs a nibble 0x08. That raises the ACK line, triggering an interrupt in the receiving machine. The receiving machine disables interrupts and raises its own ACK line.

Restated:

```

(OUT is bit 0-4, OUT.j is bit j from OUT. IN likewise)
Send_Byte:
    OUT := low nibble, OUT.4 := 1
    WAIT FOR IN.4 = 1
    OUT := high nibble, OUT.4 := 0
    WAIT FOR IN.4 = 0

```

PPP GENERIC DRIVER AND CHANNEL INTERFACE

Paul Mackerras paulus@samba.org

7 Feb 2002

The generic PPP driver in linux-2.4 provides an implementation of the functionality which is of use in any PPP implementation, including:

- the network interface unit (ppp0 etc.)
- the interface to the networking code
- PPP multilink: splitting datagrams between multiple links, and ordering and combining received fragments
- the interface to pppd, via a /dev/ppp character device
- packet compression and decompression
- TCP/IP header compression and decompression
- detecting network traffic for demand dialling and for idle timeouts
- simple packet filtering

For sending and receiving PPP frames, the generic PPP driver calls on the services of PPP channels. A PPP channel encapsulates a mechanism for transporting PPP frames from one machine to another. A PPP channel implementation can be arbitrarily complex internally but has a very simple interface with the generic PPP code: it merely has to be able to send PPP frames, receive PPP frames, and optionally handle ioctl requests. Currently there are PPP channel implementations for asynchronous serial ports, synchronous serial ports, and for PPP over ethernet.

This architecture makes it possible to implement PPP multilink in a natural and straightforward way, by allowing more than one channel to be linked to each ppp network interface unit. The generic layer is responsible for splitting datagrams on transmit and recombining them on receive.

82.1 PPP channel API

See `include/linux/ppp_channel.h` for the declaration of the types and functions used to communicate between the generic PPP layer and PPP channels.

Each channel has to provide two functions to the generic PPP layer, via the `ppp_channel.ops` pointer:

- `start_xmit()` is called by the generic layer when it has a frame to send. The channel has the option of rejecting the frame for flow-control reasons. In this case, `start_xmit()` should return 0 and the channel should call the `ppp_output_wakeup()` function at a later time when it can accept frames again, and the generic layer will then attempt to retransmit the rejected frame(s). If the frame is accepted, the `start_xmit()` function should return 1.
- `ioctl()` provides an interface which can be used by a user-space program to control aspects of the channel's behaviour. This procedure will be called when a user-space program does an ioctl system call on an instance of `/dev/ppp` which is bound to the channel. (Usually it would only be `pppd` which would do this.)

The generic PPP layer provides seven functions to channels:

- `ppp_register_channel()` is called when a channel has been created, to notify the PPP generic layer of its presence. For example, setting a serial port to the PPPDISC line discipline causes the `ppp_async` channel code to call this function.
- `ppp_unregister_channel()` is called when a channel is to be destroyed. For example, the `ppp_async` channel code calls this when a hangup is detected on the serial port.
- `ppp_output_wakeup()` is called by a channel when it has previously rejected a call to its `start_xmit` function, and can now accept more packets.
- `ppp_input()` is called by a channel when it has received a complete PPP frame.
- `ppp_input_error()` is called by a channel when it has detected that a frame has been lost or dropped (for example, because of a FCS (frame check sequence) error).
- `ppp_channel_index()` returns the channel index assigned by the PPP generic layer to this channel. The channel should provide some way (e.g. an ioctl) to transmit this back to user-space, as user-space will need it to attach an instance of `/dev/ppp` to this channel.
- `ppp_unit_number()` returns the unit number of the ppp network interface to which this channel is connected, or -1 if the channel is not connected.

Connecting a channel to the ppp generic layer is initiated from the channel code, rather than from the generic layer. The channel is expected to have some way for a user-level process to control it independently of the ppp generic layer. For example, with the `ppp_async` channel, this is provided by the file descriptor to the serial port.

Generally a user-level process will initialize the underlying communications medium and prepare it to do PPP. For example, with an async tty, this can involve setting the tty speed and modes, issuing modem commands, and then going through some sort of dialog with the remote system to invoke PPP service there. We refer to this process as *discovery*. Then the user-level process tells the medium to become a PPP channel and register itself with the generic PPP layer. The channel then has to report the channel number assigned to it back to the user-level process. From that point, the PPP negotiation code in the PPP daemon (`pppd`) can take over and perform the PPP negotiation, accessing the channel through the `/dev/ppp` interface.

At the interface to the PPP generic layer, PPP frames are stored in skbuff structures and start with the two-byte PPP protocol number. The frame does *not* include the 0xff address byte or the 0x03 control byte that are optionally used in async PPP. Nor is there any escaping of control characters, nor are there any FCS or framing characters included. That is all the responsibility of the channel code, if it is needed for the particular medium. That is, the skbuffs presented to the `start_xmit()` function contain only the 2-byte protocol number and the data, and the skbuffs presented to `ppp_input()` must be in the same format.

The channel must provide an instance of a `ppp_channel` struct to represent the channel. The channel is free to use the `private` field however it wishes. The channel should initialize the `mtu` and `hdrlen` fields before calling `ppp_register_channel()` and not change them until after `ppp_unregister_channel()` returns. The `mtu` field represents the maximum size of the data part of the PPP frames, that is, it does not include the 2-byte protocol number.

If the channel needs some headroom in the skbuffs presented to it for transmission (i.e., some space free in the skbuff data area before the start of the PPP frame), it should set the `hdrlen` field of the `ppp_channel` struct to the amount of headroom required. The generic PPP layer will attempt to provide that much headroom but the channel should still check if there is sufficient headroom and copy the skbuff if there isn't.

On the input side, channels should ideally provide at least 2 bytes of headroom in the skbuffs presented to `ppp_input()`. The generic PPP code does not require this but will be more efficient if this is done.

82.2 Buffering and flow control

The generic PPP layer has been designed to minimize the amount of data that it buffers in the transmit direction. It maintains a queue of transmit packets for the PPP unit (network interface device) plus a queue of transmit packets for each attached channel. Normally the transmit queue for the unit will contain at most one packet; the exceptions are when `pppd` sends packets by writing to `/dev/ppp`, and when the core networking code calls the generic layer's `start_xmit()` function with the queue stopped, i.e. when the generic layer has called `netif_stop_queue()`, which only happens on a transmit timeout. The `start_xmit` function always accepts and queues the packet which it is asked to transmit.

Transmit packets are dequeued from the PPP unit transmit queue and then subjected to TCP/IP header compression and packet compression (Deflate or BSD-Compress compression), as appropriate. After this point the packets can no longer be reordered, as the decompression algorithms rely on receiving compressed packets in the same order that they were generated.

If multilink is not in use, this packet is then passed to the attached channel's `start_xmit()` function. If the channel refuses to take the packet, the generic layer saves it for later transmission. The generic layer will call the channel's `start_xmit()` function again when the channel calls `ppp_output_wakeup()` or when the core networking code calls the generic layer's `start_xmit()` function again. The generic layer contains no timeout and retransmission logic; it relies on the core networking code for that.

If multilink is in use, the generic layer divides the packet into one or more fragments and puts a multilink header on each fragment. It decides how many fragments to use based on the length of the packet and the number of channels which are potentially able to accept a fragment at the moment. A channel is potentially able to accept a fragment if it doesn't have any fragments currently queued up for it to transmit. The channel may still refuse a fragment; in this case the fragment is queued up for the channel to transmit later. This scheme has the effect that more

fragments are given to higher-bandwidth channels. It also means that under light load, the generic layer will tend to fragment large packets across all the channels, thus reducing latency, while under heavy load, packets will tend to be transmitted as single fragments, thus reducing the overhead of fragmentation.

82.3 SMP safety

The PPP generic layer has been designed to be SMP-safe. Locks are used around accesses to the internal data structures where necessary to ensure their integrity. As part of this, the generic layer requires that the channels adhere to certain requirements and in turn provides certain guarantees to the channels. Essentially the channels are required to provide the appropriate locking on the `ppp_channel` structures that form the basis of the communication between the channel and the generic layer. This is because the channel provides the storage for the `ppp_channel` structure, and so the channel is required to provide the guarantee that this storage exists and is valid at the appropriate times.

The generic layer requires these guarantees from the channel:

- The `ppp_channel` object must exist from the time that `ppp_register_channel()` is called until after the call to `ppp_unregister_channel()` returns.
- No thread may be in a call to any of `ppp_input()`, `ppp_input_error()`, `ppp_output_wakeup()`, `ppp_channel_index()` or `ppp_unit_number()` for a channel at the time that `ppp_unregister_channel()` is called for that channel.
- `ppp_register_channel()` and `ppp_unregister_channel()` must be called from process context, not interrupt or softirq/BH context.
- The remaining generic layer functions may be called at softirq/BH level but must not be called from a hardware interrupt handler.
- The generic layer may call the channel `start_xmit()` function at softirq/BH level but will not call it at interrupt level. Thus the `start_xmit()` function may not block.
- The generic layer will only call the channel `ioctl()` function in process context.

The generic layer provides these guarantees to the channels:

- The generic layer will not call the `start_xmit()` function for a channel while any thread is already executing in that function for that channel.
- The generic layer will not call the `ioctl()` function for a channel while any thread is already executing in that function for that channel.
- By the time a call to `ppp_unregister_channel()` returns, no thread will be executing in a call from the generic layer to that channel's `start_xmit()` or `ioctl()` function, and the generic layer will not call either of those functions subsequently.

82.4 Interface to pppd

The PPP generic layer exports a character device interface called `/dev/ppp`. This is used by `pppd` to control PPP interface units and channels. Although there is only one `/dev/ppp`, each open instance of `/dev/ppp` acts independently and can be attached either to a PPP unit or a PPP channel. This is achieved using the `file->private_data` field to point to a separate object for each open instance of `/dev/ppp`. In this way an effect similar to Solaris' `clone open` is obtained, allowing us to control an arbitrary number of PPP interfaces and channels without having to fill up `/dev` with hundreds of device names.

When `/dev/ppp` is opened, a new instance is created which is initially unattached. Using an `ioctl` call, it can then be attached to an existing unit, attached to a newly-created unit, or attached to an existing channel. An instance attached to a unit can be used to send and receive PPP control frames, using the `read()` and `write()` system calls, along with `poll()` if necessary. Similarly, an instance attached to a channel can be used to send and receive PPP frames on that channel.

In multilink terms, the unit represents the bundle, while the channels represent the individual physical links. Thus, a PPP frame sent by a `write` to the unit (i.e., to an instance of `/dev/ppp` attached to the unit) will be subject to bundle-level compression and to fragmentation across the individual links (if multilink is in use). In contrast, a PPP frame sent by a `write` to the channel will be sent as-is on that channel, without any multilink header.

A channel is not initially attached to any unit. In this state it can be used for PPP negotiation but not for the transfer of data packets. It can then be connected to a PPP unit with an `ioctl` call, which makes it available to send and receive data packets for that unit.

The `ioctl` calls which are available on an instance of `/dev/ppp` depend on whether it is unattached, attached to a PPP interface, or attached to a PPP channel. The `ioctl` calls which are available on an unattached instance are:

- `PPPIOCNEWUNIT` creates a new PPP interface and makes this `/dev/ppp` instance the "owner" of the interface. The argument should point to an `int` which is the desired unit number if $>= 0$, or -1 to assign the lowest unused unit number. Being the owner of the interface means that the interface will be shut down if this instance of `/dev/ppp` is closed.
- `PPPIOCATTACH` attaches this instance to an existing PPP interface. The argument should point to an `int` containing the unit number. This does not make this instance the owner of the PPP interface.
- `PPPIOCATTCHAN` attaches this instance to an existing PPP channel. The argument should point to an `int` containing the channel number.

The `ioctl` calls available on an instance of `/dev/ppp` attached to a channel are:

- `PPPIOCONNECT` connects this channel to a PPP interface. The argument should point to an `int` containing the interface unit number. It will return an `EINVAL` error if the channel is already connected to an interface, or `ENXIO` if the requested interface does not exist.
- `PPPIOCDISCONN` disconnects this channel from the PPP interface that it is connected to. It will return an `EINVAL` error if the channel is not connected to an interface.
- `PPPIOCBRIDGECHAN` bridges a channel with another. The argument should point to an `int` containing the channel number of the channel to bridge to. Once two channels are bridged, frames presented to one channel by `ppp_input()` are passed to the bridge instance for onward transmission. This allows frames to be switched from one channel into another: for example, to pass PPPoE frames into a PPPoL2TP session. Since channel bridging interrupts

the normal `ppp_input()` path, a given channel may not be part of a bridge at the same time as being part of a unit. This ioctl will return an `EALREADY` error if the channel is already part of a bridge or unit, or `ENXIO` if the requested channel does not exist.

- `PPPIOCUNBRIDGECHAN` performs the inverse of `PPPIOCBRIDGECHAN`, unbridging a channel pair. This ioctl will return an `EINVAL` error if the channel does not form part of a bridge.
- All other ioctl commands are passed to the channel ioctl() function.

The ioctl calls that are available on an instance that is attached to an interface unit are:

- `PPPIOCSMRU` sets the MRU (maximum receive unit) for the interface. The argument should point to an int containing the new MRU value.
- `PPPIOCSFLAGS` sets flags which control the operation of the interface. The argument should be a pointer to an int containing the new flags value. The bits in the flags value that can be set are:

<code>SC_COMP_TCP</code>	enable transmit TCP header compression
<code>SC_NO_TCP_CCID</code>	disable connection-id compression for TCP header compression
<code>SC_REJ_COMP_TCP</code>	disable receive TCP header decompression
<code>SC_CCP_OPEN</code>	Compression Control Protocol (CCP) is open, so inspect CCP packets
<code>SC_CCP_UP</code>	CCP is up, may (de)compress packets
<code>SC_LOOP_TRAFFIC</code>	send IP traffic to <code>pppd</code>
<code>SC_MULTILINK</code>	enable PPP multilink fragmentation on transmitted packets
<code>SC_MP_SHORTSEQ</code>	expect short multilink sequence numbers on received multilink fragments
<code>SC_MP_XSHORTSEQ</code>	transmit short multilink sequence nos.

The values of these flags are defined in `<linux/ppp-ioctl.h>`. Note that the values of the `SC_MULTILINK`, `SC_MP_SHORTSEQ` and `SC_MP_XSHORTSEQ` bits are ignored if the `CONFIG_PPP_MULTILINK` option is not selected.

- `PPPIOCGFLAGS` returns the value of the status/control flags for the interface unit. The argument should point to an int where the ioctl will store the flags value. As well as the values listed above for `PPPIOCSFLAGS`, the following bits may be set in the returned value:

<code>SC_COMP_RUN</code>	CCP compressor is running
<code>SC_DECOMP_RUN</code>	CCP decompressor is running
<code>SC_DC_ERROR</code>	CCP decompressor detected non-fatal error
<code>SC_DC_FERROR</code>	CCP decompressor detected fatal error

- `PPPIOCSCOMPRESS` sets the parameters for packet compression or decompression. The argument should point to a `ppp_option_data` structure (defined in `<linux/ppp-ioctl.h>`), which contains a pointer/length pair which should describe a block of memory containing a CCP option specifying a compression method and its parameters. The `ppp_option_data` struct also contains a `transmit` field. If this is 0, the ioctl will affect the receive path, otherwise the transmit path.

- PPPIOCGUNIT returns, in the int pointed to by the argument, the unit number of this interface unit.
- PPPIOCSDEBUG sets the debug flags for the interface to the value in the int pointed to by the argument. Only the least significant bit is used; if this is 1 the generic layer will print some debug messages during its operation. This is only intended for debugging the generic PPP layer code; it is generally not helpful for working out why a PPP connection is failing.
- PPPIOCGDEBUG returns the debug flags for the interface in the int pointed to by the argument.
- PPPIOCGIDLE returns the time, in seconds, since the last data packets were sent and received. The argument should point to a ppp_idle structure (defined in <linux/ppp_defs.h>). If the CONFIG_PPP_FILTER option is enabled, the set of packets which reset the transmit and receive idle timers is restricted to those which pass the active packet filter. Two versions of this command exist, to deal with user space expecting times as either 32-bit or 64-bit time_t seconds.
- PPPIOCSMAXCID sets the maximum connection-ID parameter (and thus the number of connection slots) for the TCP header compressor and decompressor. The lower 16 bits of the int pointed to by the argument specify the maximum connection-ID for the compressor. If the upper 16 bits of that int are non-zero, they specify the maximum connection-ID for the decompressor, otherwise the decompressor's maximum connection-ID is set to 15.
- PPPIOCSNPMODE sets the network-protocol mode for a given network protocol. The argument should point to an npioctl struct (defined in <linux/ppp-ioctl.h>). The protocol field gives the PPP protocol number for the protocol to be affected, and the mode field specifies what to do with packets for that protocol:

NPMODE_PASS	normal operation, transmit and receive packets
NPMODE_DROP	silently drop packets for this protocol
NPMODE_ERROR	drop packets and return an error on transmit
NPMODE_QUEUE	queue up packets for transmit, drop received packets

At present NPMODE_ERROR and NPMODE_QUEUE have the same effect as NPMODE_DROP.

- PPPIOCGNPMODE returns the network-protocol mode for a given protocol. The argument should point to an npioctl struct with the protocol field set to the PPP protocol number for the protocol of interest. On return the mode field will be set to the network-protocol mode for that protocol.
- PPPIOCSPASS and PPPIOCSACTIVE set the pass and active packet filters. These ioctls are only available if the CONFIG_PPP_FILTER option is selected. The argument should point to a sock_fprog structure (defined in <linux/filter.h>) containing the compiled BPF instructions for the filter. Packets are dropped if they fail the pass filter; otherwise, if they fail the active filter they are passed but they do not reset the transmit or receive idle timer.
- PPPIOCMRRU enables or disables multilink processing for received packets and sets the multilink MRRU (maximum reconstructed receive unit). The argument should point to an int containing the new MRRU value. If the MRRU value is 0, processing of received multilink fragments is disabled. This ioctl is only available if the CONFIG_PPP_MULTILINK option is selected.

Last modified: 7-feb-2002

THE PROC/NET/TCP AND PROC/NET/TCP6 VARIABLES

This document describes the interfaces /proc/net/tcp and /proc/net/tcp6. Note that these interfaces are deprecated in favor of tcp_diag.

These /proc interfaces provide information about currently active TCP connections, and are implemented by tcp4_seq_show() in net/ipv4/tcp_ipv4.c and tcp6_seq_show() in net/ipv6/tcp_ipv6.c, respectively.

It will first list all listening TCP sockets, and next list all established TCP connections. A typical entry of /proc/net/tcp would look like this (split up into 3 parts because of the length of the line):

```
46: 010310AC:9C4C 030310AC:1770 01
| | | | | --> connection state
| | | | |-----> remote TCP port number
| | | | |-----> remote IPv4 address
| | | | |-----> local TCP port number
| | | | |-----> local IPv4 address
| | | | |-----> number of entry

00000150:00000000 01:00000019 00000000
| | | | |-----> number of unrecovered RT0 timeouts
| | | | |-----> number of jiffies until timer expires
| | | | |-----> timer_active (see below)
| | | | |-----> receive-queue
| | | | |-----> transmit-queue

1000      0 54165785 4 cd1e6040 25 4 27 3 -1
| | | | | | | | |-----> slow start size threshold,
| | | | | | | | |-----> or -1 if the threshold
| | | | | | | | |-----> is >= 0xFFFF
| | | | | | | | |-----> sending congestion window
| | | | | | | | |-----> (ack.quick<<1)|ack.pingpong
| | | | | | | | |-----> Predicted tick of soft clock
| | | | | | | | |-----> (delayed ACK control data)
| | | | | | | | |-----> retransmit timeout
| | | | | | | | |-----> location of socket in memory
| | | | | | | | |-----> socket reference count
| | | | | | | | |-----> inode
| | | | | | | | |-----> unanswered 0-window probes
| | | | | | | | |-----> uid
```

timer_active:

0	no timer is pending
1	retransmit-timer is pending
2	another timer (e.g. delayed ack or keepalive) is pending
3	this is a socket in TIME_WAIT state. Not all fields will contain data (or even exist)
4	zero window probe timer is pending

HOW TO USE RADIOTAP HEADERS

84.1 Pointer to the radiotap include file

Radiotap headers are variable-length and extensible, you can get most of the information you need to know on them from:

```
./include/net/ieee80211_radiotap.h
```

This document gives an overview and warns on some corner cases.

84.2 Structure of the header

There is a fixed portion at the start which contains a u32 bitmap that defines if the possible argument associated with that bit is present or not. So if b0 of the it_present member of ieee80211_radiotap_header is set, it means that the header for argument index 0 (IEEE80211_RADIOTAP_TSFT) is present in the argument area.

```
< 8-byte ieee80211_radiotap_header >
[ <possible argument bitmap extensions ... > ]
[ <argument> ... ]
```

At the moment there are only 13 possible argument indexes defined, but in case we run out of space in the u32 it_present member, it is defined that b31 set indicates that there is another u32 bitmap following (shown as "possible argument bitmap extensions..." above), and the start of the arguments is moved forward 4 bytes each time.

Note also that the it_len member __le16 is set to the total number of bytes covered by the ieee80211_radiotap_header and any arguments following.

84.3 Requirements for arguments

After the fixed part of the header, the arguments follow for each argument index whose matching bit is set in the it_present member of ieee80211_radiotap_header.

- the arguments are all stored little-endian!
- the argument payload for a given argument index has a fixed size. So IEEE80211_RADIOTAP_TSFT being present always indicates an 8-byte argument is

present. See the comments in `./include/net/ieee80211_radiotap.h` for a nice breakdown of all the argument sizes

- the arguments must be aligned to a boundary of the argument size using padding. So a u16 argument must start on the next u16 boundary if it isn't already on one, a u32 must start on the next u32 boundary and so on.
- "alignment" is relative to the start of the `ieee80211_radiotap_header`, ie, the first byte of the radiotap header. The absolute alignment of that first byte isn't defined. So even if the whole radiotap header is starting at, eg, address `0x00000003`, still the first byte of the radiotap header is treated as 0 for alignment purposes.
- the above point that there may be no absolute alignment for multibyte entities in the fixed radiotap header or the argument region means that you have to take special evasive action when trying to access these multibyte entities. Some arches like Blackfin cannot deal with an attempt to dereference, eg, a u16 pointer that is pointing to an odd address. Instead you have to use a kernel API `get_unaligned()` to dereference the pointer, which will do it bytewise on the arches that require that.
- The arguments for a given argument index can be a compound of multiple types together. For example `IEEE80211_RADIOTAP_CHANNEL` has an argument payload consisting of two u16s of total length 4. When this happens, the padding rule is applied dealing with a u16, NOT dealing with a 4-byte single entity.

84.4 Example valid radiotap header

```
0x00, 0x00, // <-- radiotap version + pad byte
0x0b, 0x00, // <- radiotap header length
0x04, 0x0c, 0x00, 0x00, // <-- bitmap
0x6c, // <-- rate (in 500kHz units)
0x0c, //<-- tx power
0x01 //<-- antenna
```

84.5 Using the Radiotap Parser

If you are having to parse a radiotap struct, you can radically simplify the job by using the radiotap parser that lives in `net/wireless/radiotap.c` and has its prototypes available in `include/net/cfg80211.h`. You use it like this:

```
#include <net/cfg80211.h>

/* buf points to the start of the radiotap header part */

int MyFunction(u8 * buf, int buflen)
{
    int pkt_rate_100kHz = 0, antenna = 0, pwr = 0;
    struct ieee80211_radiotap_iterator iterator;
    int ret = ieee80211_radiotap_iterator_init(&iterator, buf, buflen);

    while (!ret) {
```

```

    ret = ieee80211_radiotap_iterator_next(&iterator);

    if (ret)
        continue;

    /* see if this argument is something we can use */

    switch (iterator.this_arg_index) {
    /*
     * You must take care when dereferencing iterator.this_arg
     * for multibyte types... the pointer is not aligned. Use
     * get_unaligned((type *)iterator.this_arg) to dereference
     * iterator.this_arg for type "type" safely on all arches.
     */
    case IEEE80211_RADIOTAP_RATE:
        /* radiotap "rate" u8 is in
         * 500kbps units, eg, 0x02=1Mbps
         */
        pkt_rate_100kHz = (*iterator.this_arg) * 5;
        break;

    case IEEE80211_RADIOTAP_ANTENNA:
        /* radiotap uses 0 for 1st ant */
        antenna = *iterator.this_arg);
        break;

    case IEEE80211_RADIOTAP_DBM_TX_POWER:
        pwr = *iterator.this_arg;
        break;

    default:
        break;
    }
} /* while more rt headers */

if (ret != -ENOENT)
    return TXRX_DROP;

/* discard the radiotap header part */
buf += iterator.max_length;
buflen -= iterator.max_length;

...
}

```

Andy Green <andy@warmcat.com>

85.1 Overview

This readme tries to provide some background on the hows and whys of RDS, and will hopefully help you find your way around the code.

In addition, please see this email about RDS origins: <http://oss.oracle.com/pipermail/rds-devel/2007-November/000228.html>

85.2 RDS Architecture

RDS provides reliable, ordered datagram delivery by using a single reliable connection between any two nodes in the cluster. This allows applications to use a single socket to talk to any other process in the cluster - so in a cluster with N processes you need N sockets, in contrast to N*N if you use a connection-oriented socket transport like TCP.

RDS is not Infiniband-specific; it was designed to support different transports. The current implementation used to support RDS over TCP as well as IB.

The high-level semantics of RDS from the application's point of view are

- Addressing

RDS uses IPv4 addresses and 16bit port numbers to identify the end point of a connection. All socket operations that involve passing addresses between kernel and user space generally use a struct sockaddr_in.

The fact that IPv4 addresses are used does not mean the underlying transport has to be IP-based. In fact, RDS over IB uses a reliable IB connection; the IP address is used exclusively to locate the remote node's GID (by ARPing for the given IP).

The port space is entirely independent of UDP, TCP or any other protocol.

- Socket interface

RDS sockets work *mostly* as you would expect from a BSD socket. The next section will cover the details. At any rate, all I/O is performed through the standard BSD socket API. Some additions like zero-copy support are implemented through control messages, while other extensions use the getsockopt/ setsockopt calls.

Sockets must be bound before you can send or receive data. This is needed because binding also selects a transport and attaches it to the socket. Once bound, the transport as-

signment does not change. RDS will tolerate IPs moving around (eg in a active-active HA scenario), but only as long as the address doesn't move to a different transport.

- sysctls

RDS supports a number of sysctls in /proc/sys/net/rds

85.3 Socket Interface

AF_RDS, PF_RDS, SOL_RDS

AF_RDS and PF_RDS are the domain type to be used with socket(2) to create RDS sockets. SOL_RDS is the socket-level to be used with setsockopt(2) and getsockopt(2) for RDS specific socket options.

fd = socket(PF_RDS, SOCK_SEQPACKET, 0);

This creates a new, unbound RDS socket.

setsockopt(SOL_SOCKET): send and receive buffer size

RDS honors the send and receive buffer size socket options. You are not allowed to queue more than SO_SNDSIZE bytes to a socket. A message is queued when sendmsg is called, and it leaves the queue when the remote system acknowledges its arrival.

The SO_RCVSIZE option controls the maximum receive queue length. This is a soft limit rather than a hard limit - RDS will continue to accept and queue incoming messages, even if that takes the queue length over the limit. However, it will also mark the port as "congested" and send a congestion update to the source node. The source node is supposed to throttle any processes sending to this congested port.

bind(fd, &sockaddr_in, ...)

This binds the socket to a local IP address and port, and a transport, if one has not already been selected via the SO_RDS_TRANSPORT socket option

sendmsg(fd, ...)

Sends a message to the indicated recipient. The kernel will transparently establish the underlying reliable connection if it isn't up yet.

An attempt to send a message that exceeds SO_SNDSIZE will return with -EMSGSIZE

An attempt to send a message that would take the total number of queued bytes over the SO_SNDSIZE threshold will return EAGAIN.

An attempt to send a message to a destination that is marked as "congested" will return ENOBUFS.

recvmsg(fd, ...)

Receives a message that was queued to this socket. The sockets recv queue accounting is adjusted, and if the queue length drops below SO_SNDSIZE, the port is marked uncongested, and a congestion update is sent to all peers.

Applications can ask the RDS kernel module to receive notifications via control messages (for instance, there is a notification when a congestion update arrived, or when a RDMA operation completes). These notifications are received through

the msg.msg_control buffer of struct msghdr. The format of the messages is described in manpages.

poll(fd)

RDS supports the poll interface to allow the application to implement async I/O.

POLLIN handling is pretty straightforward. When there's an incoming message queued to the socket, or a pending notification, we signal POLLIN.

POLLOUT is a little harder. Since you can essentially send to any destination, RDS will always signal POLLOUT as long as there's room on the send queue (ie the number of bytes queued is less than the sendbuf size).

However, the kernel will refuse to accept messages to a destination marked congested - in this case you will loop forever if you rely on poll to tell you what to do. This isn't a trivial problem, but applications can deal with this - by using congestion notifications, and by checking for ENOBUFS errors returned by sendmsg.

setsockopt(SOL_RDS, RDS_CANCEL_SENT_TO, &sockaddr_in)

This allows the application to discard all messages queued to a specific destination on this particular socket.

This allows the application to cancel outstanding messages if it detects a timeout. For instance, if it tried to send a message, and the remote host is unreachable, RDS will keep trying forever. The application may decide it's not worth it, and cancel the operation. In this case, it would use RDS_CANCEL_SENT_TO to nuke any pending messages.

setsockopt(fd, SOL_RDS, SO_RDS_TRANSPORT, (int *)&transport . .), getsockopt(fd, SOL_RDS, SO_RDS_TRANSPORT, (int *)&transport . .)

Set or read an integer defining the underlying encapsulating transport to be used for RDS packets on the socket. When setting the option, integer argument may be one of RDS_TRANS_TCP or RDS_TRANS_IB. When retrieving the value, RDS_TRANS_NONE will be returned on an unbound socket. This socket option may only be set exactly once on the socket, prior to binding it via the bind(2) system call. Attempts to set SO_RDS_TRANSPORT on a socket for which the transport has been previously attached explicitly (by SO_RDS_TRANSPORT) or implicitly (via bind(2)) will return an error of EOPNOTSUPP. An attempt to set SO_RDS_TRANSPORT to RDS_TRANS_NONE will always return EINVAL.

85.4 RDMA for RDS

see rds-rdma(7) manpage (available in rds-tools)

85.5 Congestion Notifications

see rds(7) manpage

85.6 RDS Protocol

Message header

The message header is a 'struct rds_header' (see rds.h):

Fields:

h_sequence:

per-packet sequence number

h_ack:

piggybacked acknowledgment of last packet received

h_len:

length of data, not including header

h_sport:

source port

h_dport:

destination port

h_flags:

Can be:

CONG_BITMAP	this is a congestion update bitmap
ACK_REQUIRED	receiver must ack this packet
RETRANSMITTED	packet has previously been sent

h_credit:

indicate to other end of connection that it has more credits available (i.e. there is more send room)

h_padding[4]:

unused, for future use

h_csum:

header checksum

h_exthdr:

optional data can be passed here. This is currently used for passing RDMA-related information.

ACK and retransmit handling

One might think that with reliable IB connections you wouldn't need to ack messages that have been received. The problem is that IB hardware generates an ack message before it has DMAed the message into memory. This creates a potential message loss if the HCA is disabled for any reason between when it sends the ack and before the message is DMAed and processed. This is only a potential issue if another HCA is available for fail-over.

Sending an ack immediately would allow the sender to free the sent message from their send queue quickly, but could cause excessive traffic to be used for acks. RDS piggybacks acks on sent data packets. Ack-only packets are reduced by only allowing one to be in flight at a time, and by the sender only asking for acks when its send buffers start to fill up. All retransmissions are also acked.

Flow Control

RDS's IB transport uses a credit-based mechanism to verify that there is space in the peer's receive buffers for more data. This eliminates the need for hardware retries on the connection.

Congestion

Messages waiting in the receive queue on the receiving socket are accounted against the sockets SO_RCVBUF option value. Only the payload bytes in the message are accounted for. If the number of bytes queued equals or exceeds rcvbuf then the socket is congested. All sends attempted to this socket's address should return block or return -EWOULDBLOCK.

Applications are expected to be reasonably tuned such that this situation very rarely occurs. An application encountering this "back-pressure" is considered a bug.

This is implemented by having each node maintain bitmaps which indicate which ports on bound addresses are congested. As the bitmap changes it is sent through all the connections which terminate in the local address of the bitmap which changed.

The bitmaps are allocated as connections are brought up. This avoids allocation in the interrupt handling path which queues sages on sockets. The dense bitmaps let transports send the entire bitmap on any bitmap change reasonably efficiently. This is much easier to implement than some finer-grained communication of per-port congestion. The sender does a very inexpensive bit test to test if the port it's about to send to is congested or not.

85.7 RDS Transport Layer

As mentioned above, RDS is not IB-specific. Its code is divided into a general RDS layer and a transport layer.

The general layer handles the socket API, congestion handling, loopback, stats, user-mem pinning, and the connection state machine.

The transport layer handles the details of the transport. The IB transport, for example, handles all the queue pairs, work requests, CM event handlers, and other Infiniband details.

85.8 RDS Kernel Structures

struct rds_message

aka possibly "rds_outgoing", the generic RDS layer copies data to be sent and sets header fields as needed, based on the socket API. This is then queued for the individual connection and sent by the connection's transport.

struct rds_incoming

a generic struct referring to incoming data that can be handed from the transport to the general code and queued by the general code while the socket is awoken. It is then passed back to the transport code to handle the actual copy-to-user.

struct rds_socket

per-socket information

struct rds_connection

per-connection information

struct rds_transport

pointers to transport-specific functions

struct rds_statistics

non-transport-specific statistics

struct rds_cong_map

wraps the raw congestion bitmap, contains rbnodes, waitq, etc.

85.9 Connection management

Connections may be in UP, DOWN, CONNECTING, DISCONNECTING, and ERROR states.

The first time an attempt is made by an RDS socket to send data to a node, a connection is allocated and connected. That connection is then maintained forever -- if there are transport errors, the connection will be dropped and re-established.

Dropping a connection while packets are queued will cause queued or partially-sent datagrams to be retransmitted when the connection is re-established.

85.10 The send path

rds_sendmsg()

- struct rds_message built from incoming data
- CMSGs parsed (e.g. RDMA ops)
- transport connection allocated and connected if not already
- rds_message placed on send queue
- send worker awoken

rds_send_worker()

- calls rds_send_xmit() until queue is empty

rds_send_xmit()

- transmits congestion map if one is pending
- may set ACK_REQUIRED
- calls transport to send either non-RDMA or RDMA message (RDMA ops never retransmitted)

rds_ib_xmit()

- allocs work requests from send ring
- adds any new send credits available to peer (h_credits)
- maps the rds_message's sg list
- piggybacks ack
- populates work requests
- post send to connection's queue pair

85.11 The recv path

rds_ib_recv_cq_comp_handler()

- looks at write completions
- unmaps recv buffer from device
- no errors, call rds_ib_process_recv()
- refill recv ring

rds_ib_process_recv()

- validate header checksum
- copy header to rds_ib_incoming struct if start of a new datagram
- add to ibinc's fraglist
- **if competed datagram:**
 - update cong map if datagram was cong update
 - call rds_recv_incoming() otherwise
 - note if ack is required

rds_recv_incoming()

- drop duplicate packets
- respond to pings
- find the sock associated with this datagram
- add to sock queue
- wake up sock

- do some congestion calculations

rds_recvmsg

- copy data into user iovec
- handle CMSGs
- return to application

85.12 Multipath RDS (mprds)

Mprds is multipathed-RDS, primarily intended for RDS-over-TCP (though the concept can be extended to other transports). The classical implementation of RDS-over-TCP is implemented by demultiplexing multiple PF_RDS sockets between any 2 endpoints (where endpoint == [IP address, port]) over a single TCP socket between the 2 IP addresses involved. This has the limitation that it ends up funneling multiple RDS flows over a single TCP flow, thus it is (a) upper-bounded to the single-flow bandwidth, (b) suffers from head-of-line blocking for all the RDS sockets.

Better throughput (for a fixed small packet size, MTU) can be achieved by having multiple TCP/IP flows per rds/tcp connection, i.e., multipathed RDS (mprds). Each such TCP/IP flow constitutes a path for the rds/tcp connection. RDS sockets will be attached to a path based on some hash (e.g., of local address and RDS port number) and packets for that RDS socket will be sent over the attached path using TCP to segment/reassemble RDS datagrams on that path.

Multipathed RDS is implemented by splitting the struct rds_connection into a common (to all paths) part, and a per-path struct rds_conn_path. All I/O workqs and reconnect threads are driven from the rds_conn_path. Transports such as TCP that are multipath capable may then set up a TCP socket per rds_conn_path, and this is managed by the transport via the transport privatee cp_transport_data pointer.

Transports announce themselves as multipath capable by setting the t_mp_capable bit during registration with the rds core module. When the transport is multipath-capable, rds_sendmsg() hashes outgoing traffic across multiple paths. The outgoing hash is computed based on the local address and port that the PF_RDS socket is bound to.

Additionally, even if the transport is MP capable, we may be peering with some node that does not support mprds, or supports a different number of paths. As a result, the peering nodes need to agree on the number of paths to be used for the connection. This is done by sending out a control packet exchange before the first data packet. The control packet exchange must have completed prior to outgoing hash completion in rds_sendmsg() when the transport is mutlipath capable.

The control packet is an RDS ping packet (i.e., packet to rds dest port 0) with the ping packet having a rds extension header option of type RDS_EXTHDR_NPATHS, length 2 bytes, and the value is the number of paths supported by the sender. The "probe" ping packet will get sent from some reserved port, RDS_FLAG_PROBE_PORT (in <linux/rds.h>) The receiver of a ping from RDS_FLAG_PROBE_PORT will thus immediately be able to compute the min(sender_paths, rcvr_paths). The pong sent in response to a probe-ping should contain the rcvr's npaths when the rcvr is mprds-capable.

If the rcvr is not mprds-capable, the exthdr in the ping will be ignored. In this case the pong will not have any exthdrs, so the sender of the probe-ping can default to single-path mprds.

LINUX WIRELESS REGULATORY DOCUMENTATION

This document gives a brief review over how the Linux wireless regulatory infrastructure works. More up to date information can be obtained at the project's web page:
<https://wireless.wiki.kernel.org/en/developers/Regulatory>

86.1 Keeping regulatory domains in userspace

Due to the dynamic nature of regulatory domains we keep them in userspace and provide a framework for userspace to upload to the kernel one regulatory domain to be used as the central core regulatory domain all wireless devices should adhere to.

86.2 How to get regulatory domains to the kernel

When the regulatory domain is first set up, the kernel will request a database file (regulatory.db) containing all the regulatory rules. It will then use that database when it needs to look up the rules for a given country.

86.3 How to get regulatory domains to the kernel (old CRDA solution)

Userspace gets a regulatory domain in the kernel by having a userspace agent build it and send it via nl80211. Only expected regulatory domains will be respected by the kernel.

A currently available userspace agent which can accomplish this is CRDA - central regulatory domain agent. Its documented here:

<https://wireless.wiki.kernel.org/en/developers/Regulatory/CRDA>

Essentially the kernel will send a udev event when it knows it needs a new regulatory domain. A udev rule can be put in place to trigger crda to send the respective regulatory domain for a specific ISO/IEC 3166 alpha2.

Below is an example udev rule which can be used:

```
# Example file, should be put in /etc/udev/rules.d/regulatory.rules KERNEL=="regulatory*", ACTION=="change", SUBSYSTEM=="platform", RUN+="/sbin/crda"
```

The alpha2 is passed as an environment variable under the variable COUNTRY.

86.4 Who asks for regulatory domains?

- Users

Users can use iw:

<https://wireless.wiki.kernel.org/en/users/Documentation/iw>

An example:

```
# set regulatory domain to "Costa Rica"
iw reg set CR
```

This will request the kernel to set the regulatory domain to the specified alpha2. The kernel in turn will then ask userspace to provide a regulatory domain for the alpha2 specified by the user by sending a uevent.

- Wireless subsystems for Country Information elements

The kernel will send a uevent to inform userspace a new regulatory domain is required. More on this to be added as its integration is added.

- Drivers

If drivers determine they need a specific regulatory domain set they can inform the wireless core using regulatory_hint(). They have two options -- they either provide an alpha2 so that crda can provide back a regulatory domain for that country or they can build their own regulatory domain based on internal custom knowledge so the wireless core can respect it.

Most drivers will rely on the first mechanism of providing a regulatory hint with an alpha2. For these drivers there is an additional check that can be used to ensure compliance based on custom EEPROM regulatory data. This additional check can be used by drivers by registering on its struct wiphy a reg_notifier() callback. This notifier is called when the core's regulatory domain has been changed. The driver can use this to review the changes made and also review who made them (driver, user, country IE) and determine what to allow based on its internal EEPROM data. Devices drivers wishing to be capable of world roaming should use this callback. More on world roaming will be added to this document when its support is enabled.

Device drivers who provide their own built regulatory domain do not need a callback as the channels registered by them are the only ones that will be allowed and therefore *additional* channels cannot be enabled.

86.5 Example code - drivers hinting an alpha2:

This example comes from the zd1211rw device driver. You can start by having a mapping of your device's EEPROM country/regulatory domain value to a specific alpha2 as follows:

```
static struct zd_reg_alpha2_map reg_alpha2_map[] = {
    { ZD_REGDOMAIN_FCC, "US" },
    { ZD_REGDOMAIN_IC, "CA" },
    { ZD_REGDOMAIN_ETSI, "DE" }, /* Generic ETSI, use most restrictive */
    { ZD_REGDOMAIN_JAPAN, "JP" },
    { ZD_REGDOMAIN_JAPAN_ADD, "JP" },
```

```
{ ZD_REGDOMAIN_SPAIN, "ES" },
{ ZD_REGDOMAIN_FRANCE, "FR" },
```

Then you can define a routine to map your read EEPROM value to an alpha2, as follows:

```
static int zd_reg2alpha2(u8 regdomain, char *alpha2)
{
    unsigned int i;
    struct zd_reg_alpha2_map *reg_map;
    for (i = 0; i < ARRAY_SIZE(reg_alpha2_map); i++) {
        reg_map = &reg_alpha2_map[i];
        if (regdomain == reg_map->reg) {
            alpha2[0] = reg_map->alpha2[0];
            alpha2[1] = reg_map->alpha2[1];
            return 0;
        }
    }
    return 1;
}
```

Lastly, you can then hint to the core of your discovered alpha2, if a match was found. You need to do this after you have registered your wiphy. You are expected to do this during initialization.

```
r = zd_reg2alpha2(mac->regdomain, alpha2);
if (!r)
    regulatory_hint(hw->wiphy, alpha2);
```

86.6 Example code - drivers providing a built in regulatory domain:

[NOTE: This API is not currently available, it can be added when required]

If you have regulatory information you can obtain from your driver and you *need* to use this we let you build a regulatory domain structure and pass it to the wireless core. To do this you should kcalloc() a structure big enough to hold your regulatory domain structure and you should then fill it with your data. Finally you simply call regulatory_hint() with the regulatory domain structure in it.

Below is a simple example, with a regulatory domain cached using the stack. Your implementation may vary (read EEPROM cache instead, for example).

Example cache of some regulatory domain:

```
struct ieee80211_regdomain mydriver_jp_regdom = {
    .n_reg_rules = 3,
    .alpha2 = "JP",
    //.alpha2 = "99", /* If I have no alpha2 to map it to */
    .reg_rules = {
        /* IEEE 802.11b/g, channels 1..14 */
        REG_RULE(2412-10, 2484+10, 40, 6, 20, 0),
```

```

    /* IEEE 802.11a, channels 34..48 */
    REG_RULE(5170-10, 5240+10, 40, 6, 20,
              NL80211_RRF_NO_IR),
    /* IEEE 802.11a, channels 52..64 */
    REG_RULE(5260-10, 5320+10, 40, 6, 20,
              NL80211_RRF_NO_IR|
              NL80211_RRF_DFS),
}
};
```

Then in some part of your code after your wiphy has been registered:

```

struct ieee80211_regdomain *rd;
int size_of_regd;
int num_rules = mydriver_jp_regdom.n_reg_rules;
unsigned int i;

size_of_regd = sizeof(struct ieee80211_regdomain) +
               (num_rules * sizeof(struct ieee80211_reg_rule));

rd = kzalloc(size_of_regd, GFP_KERNEL);
if (!rd)
    return -ENOMEM;

memcpy(rd, &mydriver_jp_regdom, sizeof(struct ieee80211_regdomain));

for (i=0; i < num_rules; i++)
    memcpy(&rd->reg_rules[i],
           &mydriver_jp_regdom.reg_rules[i],
           sizeof(struct ieee80211_reg_rule));
regulatory_struct_hint(rd);
```

86.7 Statically compiled regulatory database

When a database should be fixed into the kernel, it can be provided as a firmware file at build time that is then linked into the kernel.

NETWORK FUNCTION REPRESENTORS

This document describes the semantics and usage of representor netdevices, as used to control internal switching on SmartNICs. For the closely-related port representors on physical (multi-port) switches, see [Documentation/networking/switchdev.rst](#).

87.1 Motivation

Since the mid-2010s, network cards have started offering more complex virtualisation capabilities than the legacy SR-IOV approach (with its simple MAC/VLAN-based switching model) can support. This led to a desire to offload software-defined networks (such as OpenVSwitch) to these NICs to specify the network connectivity of each function. The resulting designs are variously called SmartNICs or DPUs.

Network function representors bring the standard Linux networking stack to virtual switches and IOV devices. Just as each physical port of a Linux-controlled switch has a separate netdev, so does each virtual port of a virtual switch. When the system boots, and before any offload is configured, all packets from the virtual functions appear in the networking stack of the PF via the representors. The PF can thus always communicate freely with the virtual functions. The PF can configure standard Linux forwarding between representors, the uplink or any other netdev (routing, bridging, TC classifiers).

Thus, a representor is both a control plane object (representing the function in administrative commands) and a data plane object (one end of a virtual pipe). As a virtual link endpoint, the representor can be configured like any other netdevice; in some cases (e.g. link state) the representee will follow the representor's configuration, while in others there are separate APIs to configure the representee.

87.2 Definitions

This document uses the term "switchdev function" to refer to the PCIe function which has administrative control over the virtual switch on the device. Typically, this will be a PF, but conceivably a NIC could be configured to grant these administrative privileges instead to a VF or SF (subfunction). Depending on NIC design, a multi-port NIC might have a single switchdev function for the whole device or might have a separate virtual switch, and hence switchdev function, for each physical network port. If the NIC supports nested switching, there might be separate switchdev functions for each nested switch, in which case each switchdev function should only create representors for the ports on the (sub-)switch it directly administers.

A "representee" is the object that a representor represents. So for example in the case of a VF representor, the representee is the corresponding VF.

87.3 What does a representor do?

A representor has three main roles.

1. It is used to configure the network connection the representee sees, e.g. link up/down, MTU, etc. For instance, bringing the representor administratively UP should cause the representee to see a link up / carrier on event.
2. It provides the slow path for traffic which does not hit any offloaded fast-path rules in the virtual switch. Packets transmitted on the representor netdevice should be delivered to the representee; packets transmitted by the representee which fail to match any switching rule should be received on the representor netdevice. (That is, there is a virtual pipe connecting the representor to the representee, similar in concept to a veth pair.) This allows software switch implementations (such as OpenVSwitch or a Linux bridge) to forward packets between representees and the rest of the network.
3. It acts as a handle by which switching rules (such as TC filters) can refer to the representee, allowing these rules to be offloaded.

The combination of 2) and 3) means that the behaviour (apart from performance) should be the same whether a TC filter is offloaded or not. E.g. a TC rule on a VF representor applies in software to packets received on that representor netdevice, while in hardware offload it would apply to packets transmitted by the representee VF. Conversely, a mirrored egress redirect to a VF representor corresponds in hardware to delivery directly to the representee VF.

87.4 What functions should have a representor?

Essentially, for each virtual port on the device's internal switch, there should be a representor. Some vendors have chosen to omit representors for the uplink and the physical network port, which can simplify usage (the uplink netdev becomes in effect the physical port's representor) but does not generalise to devices with multiple ports or uplinks.

Thus, the following should all have representors:

- VFs belonging to the switchdev function.
- Other PFs on the local PCIe controller, and any VFs belonging to them.
- PFs and VFs on external PCIe controllers on the device (e.g. for any embedded System-on-Chip within the SmartNIC).
- PFs and VFs with other personalities, including network block devices (such as a vDPA virtio-blk PF backed by remote/distributed storage), if (and only if) their network access is implemented through a virtual switch port.¹ Note that such functions can require a representor despite the representee not having a netdev.

¹ The concept here is that a hardware IP stack in the device performs the translation between block DMA requests and network packets, so that only network packets pass through the virtual port onto the switch. The network access that the IP stack "sees" would then be configurable through tc rules; e.g. its traffic might all be wrapped in a specific VLAN or VxLAN. However, any needed configuration of the block device *qua* block device, not being a networking entity, would not be appropriate for the representor and would thus use some other channel such as devlink. Contrast this with the case of a virtio-blk implementation which forwards the DMA requests unchanged to

- Subfunctions (SFs) belonging to any of the above PFs or VFs, if they have their own port on the switch (as opposed to using their parent PF's port).
- Any accelerators or plugins on the device whose interface to the network is through a virtual switch port, even if they do not have a corresponding PCIe PF or VF.

This allows the entire switching behaviour of the NIC to be controlled through representor TC rules.

It is a common misunderstanding to conflate virtual ports with PCIe virtual functions or their netdevs. While in simple cases there will be a 1:1 correspondence between VF netdevices and VF representors, more advanced device configurations may not follow this. A PCIe function which does not have network access through the internal switch (not even indirectly through the hardware implementation of whatever services the function provides) should *not* have a representor (even if it has a netdev). Such a function has no switch virtual port for the representor to configure or to be the other end of the virtual pipe. The representor represents the virtual port, not the PCIe function nor the 'end user' netdevice.

87.5 How are representors created?

The driver instance attached to the switchdev function should, for each virtual port on the switch, create a pure-software netdevice which has some form of in-kernel reference to the switchdev function's own netdevice or driver private data (`netdev_priv()`). This may be by enumerating ports at probe time, reacting dynamically to the creation and destruction of ports at run time, or a combination of the two.

The operations of the representor netdevice will generally involve acting through the switchdev function. For example, `ndo_start_xmit()` might send the packet through a hardware TX queue attached to the switchdev function, with either packet metadata or queue configuration marking it for delivery to the representee.

87.6 How are representors identified?

The representor netdevice should *not* directly refer to a PCIe device (e.g. through `net_dev->dev.parent / SET_NETDEV_DEV()`), either of the representee or of the switchdev function. Instead, the driver should use the `SET_NETDEV_DEVLINK_PORT` macro to assign a devlink port instance to the netdevice before registering the netdevice; the kernel uses the devlink port to provide the `phys_switch_id` and `phys_port_name` sysfs nodes. (Some legacy drivers implement `ndo_get_port_parent_id()` and `ndo_get_phys_port_name()` directly, but this is deprecated.) See [Documentation/networking/devlink/devlink-port.rst](#) for the details of this API.

It is expected that userland will use this information (e.g. through udev rules) to construct an appropriately informative name or alias for the netdevice. For instance if the switchdev function is `eth4` then a representor with a `phys_port_name` of `p0pf1vf2` might be renamed `eth4pf1vf2rep`.

There are as yet no established conventions for naming representors which do not correspond to PCIe functions (e.g. accelerators and plugins).

another PF whose driver then initiates and terminates IP traffic in software; in that case the DMA traffic would *not* run over the virtual switch and the virtio-blk PF should thus *not* have a representor.

87.7 How do representors interact with TC rules?

Any TC rule on a representor applies (in software TC) to packets received by that representor netdevice. Thus, if the delivery part of the rule corresponds to another port on the virtual switch, the driver may choose to offload it to hardware, applying it to packets transmitted by the representee.

Similarly, since a TC mirrored egress action targeting the representor would (in software) send the packet through the representor (and thus indirectly deliver it to the representee), hardware offload should interpret this as delivery to the representee.

As a simple example, if `PORT_DEV` is the physical port representor and `REP_DEV` is a VF representor, the following rules:

```
tc filter add dev $REP_DEV parent ffff: protocol ipv4 flower \
    action mirrored egress redirect dev $PORT_DEV
tc filter add dev $PORT_DEV parent ffff: protocol ipv4 flower skip_sw \
    action mirrored egress mirror dev $REP_DEV
```

would mean that all IPv4 packets from the VF are sent out the physical port, and all IPv4 packets received on the physical port are delivered to the VF in addition to `PORT_DEV`. (Note that without `skip_sw` on the second rule, the VF would get two copies, as the packet reception on `PORT_DEV` would trigger the TC rule again and mirror the packet to `REP_DEV`.)

On devices without separate port and uplink representors, `PORT_DEV` would instead be the switchdev function's own uplink netdevice.

Of course the rules can (if supported by the NIC) include packet-modifying actions (e.g. VLAN push/pop), which should be performed by the virtual switch.

Tunnel encapsulation and decapsulation are rather more complicated, as they involve a third netdevice (a tunnel netdev operating in metadata mode, such as a VxLAN device created with `ip link add vxlan0 type vxlan external`) and require an IP address to be bound to the underlay device (e.g. switchdev function uplink netdev or port representor). TC rules such as:

```
tc filter add dev $REP_DEV parent ffff: flower \
    action tunnel_key set id $VNI src_ip $LOCAL_IP dst_ip $REMOTE_IP \
        dst_port 4789 \
    action mirrored egress redirect dev vxlan0
tc filter add dev vxlan0 parent ffff: flower enc_src_ip $REMOTE_IP \
    enc_dst_ip $LOCAL_IP enc_key_id $VNI enc_dst_port 4789 \
    action tunnel_key unset action mirrored egress redirect dev $REP_DEV
```

where `LOCAL_IP` is an IP address bound to `PORT_DEV`, and `REMOTE_IP` is another IP address on the same subnet, mean that packets sent by the VF should be VxLAN encapsulated and sent out the physical port (the driver has to deduce this by a route lookup of `LOCAL_IP` leading to `PORT_DEV`, and also perform an ARP/neighbour table lookup to find the MAC addresses to use in the outer Ethernet frame), while UDP packets received on the physical port with UDP port 4789 should be parsed as VxLAN and, if their VSID matches `$VNI`, decapsulated and forwarded to the VF.

If this all seems complicated, just remember the 'golden rule' of TC offload: the hardware should ensure the same final results as if the packets were processed through the slow path,

traversed software TC (except ignoring any `skip_hw` rules and applying any `skip_sw` rules) and were transmitted or received through the representor netdevices.

87.8 Configuring the representee's MAC

The representee's link state is controlled through the representor. Setting the representor administratively UP or DOWN should cause carrier ON or OFF at the representee.

Setting an MTU on the representor should cause that same MTU to be reported to the representee. (On hardware that allows configuring separate and distinct MTU and MRU values, the representor MTU should correspond to the representee's MRU and vice-versa.)

Currently there is no way to use the representor to set the station permanent MAC address of the representee; other methods available to do this include:

- legacy SR-IOV (`ip link set DEVICE vf NUM mac LLADDR`)
- devlink port function (see **devlink-port(8)** and [Documentation/networking/devlink/devlink-port.rst](#))

RXRPC NETWORK PROTOCOL

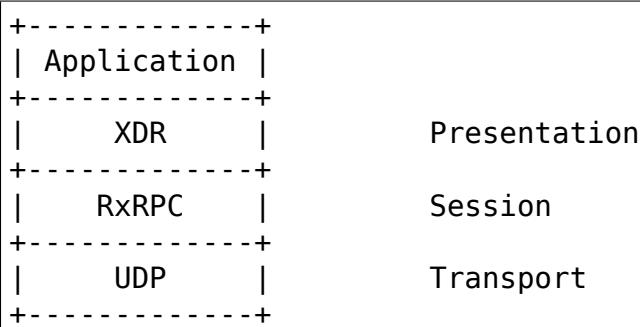
The RxRPC protocol driver provides a reliable two-phase transport on top of UDP that can be used to perform RxRPC remote operations. This is done over sockets of AF_RXRPC family, using sendmsg() and recvmsg() with control data to send and receive data, aborts and errors.

Contents of this document:

- (1) Overview.
- (2) RxRPC protocol summary.
- (3) AF_RXRPC driver model.
- (4) Control messages.
- (5) Socket options.
- (6) Security.
- (7) Example client usage.
- (8) Example server usage.
- (9) AF_RXRPC kernel interface.
- (10) Configurable parameters.

88.1 Overview

RxRPC is a two-layer protocol. There is a session layer which provides reliable virtual connections using UDP over IPv4 (or IPv6) as the transport layer, but implements a real network protocol; and there's the presentation layer which renders structured data to binary blobs and back again using XDR (as does SunRPC):



AF_RXRPC provides:

- (1) Part of an RxRPC facility for both kernel and userspace applications by making the session part of it a Linux network protocol (AF_RXRPC).
- (2) A two-phase protocol. The client transmits a blob (the request) and then receives a blob (the reply), and the server receives the request and then transmits the reply.
- (3) Retention of the reusable bits of the transport system set up for one call to speed up subsequent calls.
- (4) A secure protocol, using the Linux kernel's key retention facility to manage security on the client end. The server end must of necessity be more active in security negotiations.

AF_RXRPC does not provide XDR marshalling/presentation facilities. That is left to the application. AF_RXRPC only deals in blobs. Even the operation ID is just the first four bytes of the request blob, and as such is beyond the kernel's interest.

Sockets of AF_RXRPC family are:

- (1) created as type SOCK_DGRAM;
- (2) provided with a protocol of the type of underlying transport they're going to use - currently only PF_INET is supported.

The Andrew File System (AFS) is an example of an application that uses this and that has both kernel (filesystem) and userspace (utility) components.

88.2 RxRPC Protocol Summary

An overview of the RxRPC protocol:

- (1) RxRPC sits on top of another networking protocol (UDP is the only option currently), and uses this to provide network transport. UDP ports, for example, provide transport endpoints.
- (2) RxRPC supports multiple virtual "connections" from any given transport endpoint, thus allowing the endpoints to be shared, even to the same remote endpoint.
- (3) Each connection goes to a particular "service". A connection may not go to multiple services. A service may be considered the RxRPC equivalent of a port number. AF_RXRPC permits multiple services to share an endpoint.
- (4) Client-originating packets are marked, thus a transport endpoint can be shared between client and server connections (connections have a direction).
- (5) Up to a billion connections may be supported concurrently between one local transport endpoint and one service on one remote endpoint. An RxRPC connection is described by seven numbers:

```
Local address  }
Local port      } Transport (UDP) address
Remote address  }
Remote port     }
Direction
Connection ID
Service ID
```

- (6) Each RxRPC operation is a "call". A connection may make up to four billion calls, but only up to four calls may be in progress on a connection at any one time.
- (7) Calls are two-phase and asymmetric: the client sends its request data, which the service receives; then the service transmits the reply data which the client receives.
- (8) The data blobs are of indefinite size, the end of a phase is marked with a flag in the packet. The number of packets of data making up one blob may not exceed 4 billion, however, as this would cause the sequence number to wrap.
- (9) The first four bytes of the request data are the service operation ID.
- (10) Security is negotiated on a per-connection basis. The connection is initiated by the first data packet on it arriving. If security is requested, the server then issues a "challenge" and then the client replies with a "response". If the response is successful, the security is set for the lifetime of that connection, and all subsequent calls made upon it use that same security. In the event that the server lets a connection lapse before the client, the security will be renegotiated if the client uses the connection again.
- (11) Calls use ACK packets to handle reliability. Data packets are also explicitly sequenced per call.
- (12) There are two types of positive acknowledgment: hard-ACKs and soft-ACKs. A hard-ACK indicates to the far side that all the data received to a point has been received and processed; a soft-ACK indicates that the data has been received but may yet be discarded and re-requested. The sender may not discard any transmittable packets until they've been hard-ACK'd.
- (13) Reception of a reply data packet implicitly hard-ACK's all the data packets that make up the request.
- (14) An call is complete when the request has been sent, the reply has been received and the final hard-ACK on the last packet of the reply has reached the server.
- (15) An call may be aborted by either end at any time up to its completion.

88.3 AF_RXRPC Driver Model

About the AF_RXRPC driver:

- (1) The AF_RXRPC protocol transparently uses internal sockets of the transport protocol to represent transport endpoints.
- (2) AF_RXRPC sockets map onto RxRPC connection bundles. Actual RxRPC connections are handled transparently. One client socket may be used to make multiple simultaneous calls to the same service. One server socket may handle calls from many clients.
- (3) Additional parallel client connections will be initiated to support extra concurrent calls, up to a tunable limit.
- (4) Each connection is retained for a certain amount of time [tunable] after the last call currently using it has completed in case a new call is made that could reuse it.
- (5) Each internal UDP socket is retained [tunable] for a certain amount of time [tunable] after the last connection using it discarded, in case a new connection is made that could use it.

- (6) A client-side connection is only shared between calls if they have the same key struct describing their security (and assuming the calls would otherwise share the connection). Non-secured calls would also be able to share connections with each other.
- (7) A server-side connection is shared if the client says it is.
- (8) ACK'ing is handled by the protocol driver automatically, including ping replying.
- (9) SO_KEEPALIVE automatically pings the other side to keep the connection alive [TODO].
- (10) If an ICMP error is received, all calls affected by that error will be aborted with an appropriate network error passed through recvmsg().

Interaction with the user of the RxRPC socket:

- (1) A socket is made into a server socket by binding an address with a non-zero service ID.
- (2) In the client, sending a request is achieved with one or more sendmsgs, followed by the reply being received with one or more recvmsgs.
- (3) The first sendmsg for a request to be sent from a client contains a tag to be used in all other sendmsgs or recvmsgs associated with that call. The tag is carried in the control data.
- (4) connect() is used to supply a default destination address for a client socket. This may be overridden by supplying an alternate address to the first sendmsg() of a call (struct msghdr::msg_name).
- (5) If connect() is called on an unbound client, a random local port will bind before the operation takes place.
- (6) A server socket may also be used to make client calls. To do this, the first sendmsg() of the call must specify the target address. The server's transport endpoint is used to send the packets.
- (7) Once the application has received the last message associated with a call, the tag is guaranteed not to be seen again, and so it can be used to pin client resources. A new call can then be initiated with the same tag without fear of interference.
- (8) In the server, a request is received with one or more recvmsgs, then the the reply is transmitted with one or more sendmsgs, and then the final ACK is received with a last recvmsg.
- (9) When sending data for a call, sendmsg is given MSG_MORE if there's more data to come on that call.
- (10) When receiving data for a call, recvmsg flags MSG_MORE if there's more data to come for that call.
- (11) When receiving data or messages for a call, MSG_EOR is flagged by recvmsg to indicate the terminal message for that call.
- (12) A call may be aborted by adding an abort control message to the control data. Issuing an abort terminates the kernel's use of that call's tag. Any messages waiting in the receive queue for that call will be discarded.
- (13) Aborts, busy notifications and challenge packets are delivered by recvmsg, and control data messages will be set to indicate the context. Receiving an abort or a busy message terminates the kernel's use of that call's tag.
- (14) The control data part of the msghdr struct is used for a number of things:

- (1) The tag of the intended or affected call.
 - (2) Sending or receiving errors, aborts and busy notifications.
 - (3) Notifications of incoming calls.
 - (4) Sending debug requests and receiving debug replies [TODO].
- (15) When the kernel has received and set up an incoming call, it sends a message to server application to let it know there's a new call awaiting its acceptance [recvmsg reports a special control message]. The server application then uses sendmsg to assign a tag to the new call. Once that is done, the first part of the request data will be delivered by recvmsg.
- (16) The server application has to provide the server socket with a keyring of secret keys corresponding to the security types it permits. When a secure connection is being set up, the kernel looks up the appropriate secret key in the keyring and then sends a challenge packet to the client and receives a response packet. The kernel then checks the authorisation of the packet and either aborts the connection or sets up the security.
- (17) The name of the key a client will use to secure its communications is nominated by a socket option.

Notes on sendmsg:

- (1) MSG_WAITALL can be set to tell sendmsg to ignore signals if the peer is making progress at accepting packets within a reasonable time such that we manage to queue up all the data for transmission. This requires the client to accept at least one packet per 2*RTT time period.

If this isn't set, sendmsg() will return immediately, either returning EINTR/ERESTARTSYS if nothing was consumed or returning the amount of data consumed.

Notes on recvmsg:

- (1) If there's a sequence of data messages belonging to a particular call on the receive queue, then recvmsg will keep working through them until:
 - (a) it meets the end of that call's received data,
 - (b) it meets a non-data message,
 - (c) it meets a message belonging to a different call, or
 - (d) it fills the user buffer.

If recvmsg is called in blocking mode, it will keep sleeping, awaiting the reception of further data, until one of the above four conditions is met.

- (2) MSG_PEEK operates similarly, but will return immediately if it has put any data in the buffer rather than sleeping until it can fill the buffer.
- (3) If a data message is only partially consumed in filling a user buffer, then the remainder of that message will be left on the front of the queue for the next taker. MSG_TRUNC will never be flagged.
- (4) If there is more data to be had on a call (it hasn't copied the last byte of the last data message in that phase yet), then MSG_MORE will be flagged.

88.4 Control Messages

AF_RXRPC makes use of control messages in sendmsg() and recvmsg() to multiplex calls, to invoke certain actions and to report certain conditions. These are:

MESSAGE ID	SRT	DATA	MEANING
RXRPC_USER_CALL_ID	sr-	User ID	App's call specifier
RXRPC_ABORT	srt	Abort code	Abort code to issue/received
RXRPC_ACK	-rt	n/a	Final ACK received
RXRPC_NET_ERROR	-rt	error num	Network error on call
RXRPC_BUSY	-rt	n/a	Call rejected (server busy)
RXRPC_LOCAL_ERROR	-rt	error num	Local error encountered
RXRPC_NEW_CALL	-r-	n/a	New call received
RXRPC_ACCEPT	s--	n/a	Accept new call
RXRPC_EXCLUSIVE_CALL	s--	n/a	Make an exclusive client call
RXRPC_UPGRADE_SERVICES	s--	n/a	Client call can be upgraded
RXRPC_TX_LENGTH	s--	data len	Total length of Tx data

(SRT = usable in Sendmsg / delivered by Recvmsg / Terminal message)

(1) RXRPC_USER_CALL_ID

This is used to indicate the application's call ID. It's an unsigned long that the app specifies in the client by attaching it to the first data message or in the server by passing it in association with an RXRPC_ACCEPT message. recvmsg() passes it in conjunction with all messages except those of the RXRPC_NEW_CALL message.

(2) RXRPC_ABORT

This can be used by an application to abort a call by passing it to sendmsg, or it can be delivered by recvmsg to indicate a remote abort was received. Either way, it must be associated with an RXRPC_USER_CALL_ID to specify the call affected. If an abort is being sent, then error EBADSLT will be returned if there is no call with that user ID.

(3) RXRPC_ACK

This is delivered to a server application to indicate that the final ACK of a call was received from the client. It will be associated with an RXRPC_USER_CALL_ID to indicate the call that's now complete.

(4) RXRPC_NET_ERROR

This is delivered to an application to indicate that an ICMP error message was encountered in the process of trying to talk to the peer. An errno-class integer value will be included in the control message data indicating the problem, and an RXRPC_USER_CALL_ID will indicate the call affected.

(5) RXRPC_BUSY

This is delivered to a client application to indicate that a call was rejected by the server due to the server being busy. It will be associated with an RXRPC_USER_CALL_ID to indicate the rejected call.

(6) RXRPC_LOCAL_ERROR

This is delivered to an application to indicate that a local error was encountered and that a call has been aborted because of it. An errno-class integer value will be included in the control message data indicating the problem, and an RXRPC_USER_CALL_ID will indicate the call affected.

(7) RXRPC_NEW_CALL

This is delivered to indicate to a server application that a new call has arrived and is awaiting acceptance. No user ID is associated with this, as a user ID must subsequently be assigned by doing an RXRPC_ACCEPT.

(8) RXRPC_ACCEPT

This is used by a server application to attempt to accept a call and assign it a user ID. It should be associated with an RXRPC_USER_CALL_ID to indicate the user ID to be assigned. If there is no call to be accepted (it may have timed out, been aborted, etc.), then sendmsg will return error ENODATA. If the user ID is already in use by another call, then error EBADSLT will be returned.

(9) RXRPC_EXCLUSIVE_CALL

This is used to indicate that a client call should be made on a one-off connection. The connection is discarded once the call has terminated.

(10) RXRPC_UPGRADE_SERVICE

This is used to make a client call to probe if the specified service ID may be upgraded by the server. The caller must check msg_name returned to recvmsg() for the service ID actually in use. The operation probed must be one that takes the same arguments in both services.

Once this has been used to establish the upgrade capability (or lack thereof) of the server, the service ID returned should be used for all future communication to that server and RXRPC_UPGRADE_SERVICE should no longer be set.

(11) RXRPC_TX_LENGTH

This is used to inform the kernel of the total amount of data that is going to be transmitted by a call (whether in a client request or a service response). If given, it allows the kernel to encrypt from the userspace buffer directly to the packet buffers, rather than copying into the buffer and then encrypting in place. This may only be given with the first sendmsg() providing data for a call. EMSGSIZE will be generated if the amount of data actually given is different.

This takes a parameter of `_s64` type that indicates how much will be transmitted. This may not be less than zero.

The symbol RXRPC_SUPPORTED is defined as one more than the highest control message type supported. At run time this can be queried by means of the RXRPC_SUPPORTED_CMSG socket option (see below).

SOCKET OPTIONS

AF_RXRPC sockets support a few socket options at the SOL_RXRPC level:

(1) RXRPC_SECURITY_KEY

This is used to specify the description of the key to be used. The key is extracted from the calling process's keyrings with request_key() and should be of "rxrpc" type.

The optval pointer points to the description string, and optlen indicates how long the string is, without the NUL terminator.

(2) RXRPC_SECURITY_KEYRING

Similar to above but specifies a keyring of server secret keys to use (key type "keyring"). See the "Security" section.

(3) RXRPC_EXCLUSIVE_CONNECTION

This is used to request that new connections should be used for each call made subsequently on this socket. optval should be NULL and optlen 0.

(4) RXRPC_MIN_SECURITY_LEVEL

This is used to specify the minimum security level required for calls on this socket. optval must point to an int containing one of the following values:

(a) RXRPC_SECURITY_PLAIN

Encrypted checksum only.

(b) RXRPC_SECURITY_AUTH

Encrypted checksum plus packet padded and first eight bytes of packet encrypted - which includes the actual packet length.

(c) RXRPC_SECURITY_ENCRYPT

Encrypted checksum plus entire packet padded and encrypted, including actual packet length.

(5) RXRPC_UPGRADEABLE_SERVICE

This is used to indicate that a service socket with two bindings may upgrade one bound service to the other if requested by the client. optval must point to an array of two unsigned short ints. The first is the service ID to upgrade from and the second the service ID to upgrade to.

(6) RXRPC_SUPPORTED_CMSG

This is a read-only option that writes an int into the buffer indicating the highest control message type supported.

SECURITY

Currently, only the kerberos 4 equivalent protocol has been implemented (security index 2 - rxkad). This requires the rxkad module to be loaded and, on the client, tickets of the appropriate type to be obtained from the AFS kaserver or the kerberos server and installed as "rxrpc" type keys. This is normally done using the klog program. An example simple klog program can be found at:

<http://people.redhat.com/~dhowells/rxrpc/klog.c>

The payload provided to add_key() on the client should be of the following form:

```
struct rxrpc_key_sec2_v1 {
    uint16_t      security_index; /* 2 */
    uint16_t      ticket_length;  /* length of ticket[] */
    uint32_t      expiry;        /* time at which expires */
    uint8_t       kvno;          /* key version number */
    uint8_t       __pad[3];
    uint8_t      session_key[8]; /* DES session key */
    uint8_t      ticket[0];     /* the encrypted ticket */
};
```

Where the ticket blob is just appended to the above structure.

For the server, keys of type "rxrpc_s" must be made available to the server. They have a description of "<serviceID>:<securityIndex>" (eg: "52:2" for an rxkad key for the AFS VL service). When such a key is created, it should be given the server's secret key as the instantiation data (see the example below).

```
add_key("rxrpc_s", "52:2", secret_key, 8, keyring);
```

A keyring is passed to the server socket by naming it in a sockopt. The server socket then looks the server secret keys up in this keyring when secure incoming connections are made. This can be seen in an example program that can be found at:

<http://people.redhat.com/~dhowells/rxrpc/listen.c>

EXAMPLE CLIENT USAGE

A client would issue an operation by:

- (1) An RxRPC socket is set up by:

```
client = socket(AF_RXRPC, SOCK_DGRAM, PF_INET);
```

Where the third parameter indicates the protocol family of the transport socket used - usually IPv4 but it can also be IPv6 [TODO].

- (2) A local address can optionally be bound:

```
struct sockaddr_rxrpc srx = {
    .srx_family      = AF_RXRPC,
    .srx_service     = 0, /* we're a client */
    .transport_type  = SOCK_DGRAM, /* type of transport socket */
    .transport.sin_family = AF_INET,
    .transport.sin_port   = htons(7000), /* AFS callback */
    .transport.sin_address = 0, /* all local interfaces */
};
bind(client, &srx, sizeof(srx));
```

This specifies the local UDP port to be used. If not given, a random non-privileged port will be used. A UDP port may be shared between several unrelated RxRPC sockets. Security is handled on a basis of per-RxRPC virtual connection.

- (3) The security is set:

```
const char *key = "AFS:cambridge.redhat.com";
setsockopt(client, SOL_RXRPC, RXRPC_SECURITY_KEY, key, strlen(key));
```

This issues a request_key() to get the key representing the security context. The minimum security level can be set:

```
unsigned int sec = RXRPC_SECURITY_ENCRYPT;
setsockopt(client, SOL_RXRPC, RXRPC_MIN_SECURITY_LEVEL,
           &sec, sizeof(sec));
```

- (4) The server to be contacted can then be specified (alternatively this can be done through sendmsg):

```
struct sockaddr_rxrpc srx = {
    .srx_family      = AF_RXRPC,
```

```

.srx_service      = VL_SERVICE_ID,
.transport_type = SOCK_DGRAM, /* type of transport socket */
.transport.sin_family   = AF_INET,
.transport.sin_port     = htons(7005), /* AFS volume manager */
.transport.sin_address = ...,
};

connect(client, &srx, sizeof(srx));

```

- (5) The request data should then be posted to the server socket using a series of sendmsg() calls, each with the following control message attached:

RXRPC_USER_CALL_ID	specifies the user ID for this call
--------------------	-------------------------------------

MSG_MORE should be set in msghdr::msg_flags on all but the last part of the request. Multiple requests may be made simultaneously.

An RXRPC_TX_LENGTH control message can also be specified on the first sendmsg() call.

If a call is intended to go to a destination other than the default specified through connect(), then msghdr::msg_name should be set on the first request message of that call.

- (6) The reply data will then be posted to the server socket for recvmsg() to pick up. MSG_MORE will be flagged by recvmsg() if there's more reply data for a particular call to be read. MSG_EOR will be set on the terminal read for a call.

All data will be delivered with the following control message attached:

RXRPC_USER_CALL_ID - specifies the user ID for this call

If an abort or error occurred, this will be returned in the control data buffer instead, and MSG_EOR will be flagged to indicate the end of that call.

A client may ask for a service ID it knows and ask that this be upgraded to a better service if one is available by supplying RXRPC_UPGRADE_SERVICE on the first sendmsg() of a call. The client should then check srx_service in the msg_name filled in by recvmsg() when collecting the result. srx_service will hold the same value as given to sendmsg() if the upgrade request was ignored by the service - otherwise it will be altered to indicate the service ID the server upgraded to. Note that the upgraded service ID is chosen by the server. The caller has to wait until it sees the service ID in the reply before sending any more calls (further calls to the same destination will be blocked until the probe is concluded).

91.1 Example Server Usage

A server would be set up to accept operations in the following manner:

- (1) An RxRPC socket is created by:

```
server = socket(AF_RXRPC, SOCK_DGRAM, PF_INET);
```

Where the third parameter indicates the address type of the transport socket used - usually IPv4.

- (2) Security is set up if desired by giving the socket a keyring with server secret keys in it:

```

keyring = add_key("keyring", "AFSkeys", NULL, 0,
                  KEY_SPEC_PROCESS_KEYRING);

const char secret_key[8] = {
    0xa7, 0x83, 0x8a, 0xcb, 0xc7, 0x83, 0xec, 0x94 };
add_key("rxrpc_s", "52:2", secret_key, 8, keyring);

setsockopt(server, SOL_RXRPC, RXRPC_SECURITY_KEYRING, "AFSkeys", 7);

```

The keyring can be manipulated after it has been given to the socket. This permits the server to add more keys, replace keys, etc. while it is live.

- (3) A local address must then be bound:

```

struct sockaddr_rxrpc srx = {
    .srx_family      = AF_RXRPC,
    .srx_service     = VL_SERVICE_ID, /* RxRPC service ID */
    .transport_type  = SOCK_DGRAM,   /* type of transport socket */
    .transport.sin_family = AF_INET,
    .transport.sin_port   = htons(7000), /* AFS callback */
    .transport.sin_address = 0, /* all local interfaces */
};

bind(server, &srx, sizeof(srx));

```

More than one service ID may be bound to a socket, provided the transport parameters are the same. The limit is currently two. To do this, bind() should be called twice.

- (4) If service upgrading is required, first two service IDs must have been bound and then the following option must be set:

```

unsigned short service_ids[2] = { from_ID, to_ID };
setsockopt(server, SOL_RXRPC, RXRPC_UPGRADEABLE_SERVICE,
            service_ids, sizeof(service_ids));

```

This will automatically upgrade connections on service from_ID to service to_ID if they request it. This will be reflected in msg_name obtained through recvmsg() when the request data is delivered to userspace.

- (5) The server is then set to listen out for incoming calls:

```
listen(server, 100);
```

- (6) The kernel notifies the server of pending incoming connections by sending it a message for each. This is received with recvmsg() on the server socket. It has no data, and has a single dataless control message attached:

```
RXRPC_NEW_CALL
```

The address that can be passed back by recvmsg() at this point should be ignored since the call for which the message was posted may have gone by the time it is accepted - in which case the first call still on the queue will be accepted.

- (7) The server then accepts the new call by issuing a sendmsg() with two pieces of control data and no actual data:

RXRPC_ACCEPT	indicate connection acceptance
RXRPC_USER_CALL_ID	specify user ID for this call

- (8) The first request data packet will then be posted to the server socket for recvmsg() to pick up. At that point, the RxRPC address for the call can be read from the address fields in the msghdr struct.

Subsequent request data will be posted to the server socket for recvmsg() to collect as it arrives. All but the last piece of the request data will be delivered with MSG_MORE flagged.

All data will be delivered with the following control message attached:

RXRPC_USER_CALL_ID	specifies the user ID for this call
--------------------	-------------------------------------

- (9) The reply data should then be posted to the server socket using a series of sendmsg() calls, each with the following control messages attached:

RXRPC_USER_CALL_ID	specifies the user ID for this call
--------------------	-------------------------------------

MSG_MORE should be set in msghdr::msg_flags on all but the last message for a particular call.

- (10) The final ACK from the client will be posted for retrieval by recvmsg() when it is received. It will take the form of a dataless message with two control messages attached:

RXRPC_USER_CALL_ID	specifies the user ID for this call
RXRPC_ACK	indicates final ACK (no data)

MSG_EOR will be flagged to indicate that this is the final message for this call.

- (11) Up to the point the final packet of reply data is sent, the call can be aborted by calling sendmsg() with a dataless message with the following control messages attached:

RXRPC_USER_CALL_ID	specifies the user ID for this call
RXRPC_ABORT	indicates abort code (4 byte data)

Any packets waiting in the socket's receive queue will be discarded if this is issued.

Note that all the communications for a particular service take place through the one server socket, using control messages on sendmsg() and recvmsg() to determine the call affected.

91.2 AF_RXRPC Kernel Interface

The AF_RXRPC module also provides an interface for use by in-kernel utilities such as the AFS filesystem. This permits such a utility to:

- (1) Use different keys directly on individual client calls on one socket rather than having to open a whole slew of sockets, one for each key it might want to use.
- (2) Avoid having RxRPC call request_key() at the point of issue of a call or opening of a socket. Instead the utility is responsible for requesting a key at the appropriate point. AFS, for instance, would do this during VFS operations such as open() or unlink(). The key is then handed through when the call is initiated.
- (3) Request the use of something other than GFP_KERNEL to allocate memory.
- (4) Avoid the overhead of using the recvmsg() call. RxRPC messages can be intercepted before they get put into the socket Rx queue and the socket buffers manipulated directly.

To use the RxRPC facility, a kernel utility must still open an AF_RXRPC socket, bind an address as appropriate and listen if it's to be a server socket, but then it passes this to the kernel interface functions.

The kernel interface functions are as follows:

- (1) Begin a new client call:

```
struct rxrpc_call *
rxrpc_kernel_begin_call(struct socket *sock,
                      struct sockaddr_rxrpc *srx,
                      struct key *key,
                      unsigned long user_call_ID,
                      s64 tx_total_len,
                      gfp_t gfp,
                      rxrpc_notify_rx_t notify_rx,
                      bool upgrade,
                      bool intr,
                      unsigned int debug_id);
```

This allocates the infrastructure to make a new RxRPC call and assigns call and connection numbers. The call will be made on the UDP port that the socket is bound to. The call will go to the destination address of a connected client socket unless an alternative is supplied (srx is non-NULL).

If a key is supplied then this will be used to secure the call instead of the key bound to the socket with the RXRPC_SECURITY_KEY sockopt. Calls secured in this way will still share connections if at all possible.

The user_call_ID is equivalent to that supplied to sendmsg() in the control data buffer. It is entirely feasible to use this to point to a kernel data structure.

tx_total_len is the amount of data the caller is intending to transmit with this call (or -1 if unknown at this point). Setting the data size allows the kernel to encrypt directly to the packet buffers, thereby saving a copy. The value may not be less than -1.

notify_rx is a pointer to a function to be called when events such as incoming data packets or remote aborts happen.

upgrade should be set to true if a client operation should request that the server upgrade the service to a better one. The resultant service ID is returned by rxrpc_kernel_recv_data().

intr should be set to true if the call should be interruptible. If this is not set, this function may not return until a channel has been allocated; if it is set, the function may return -ERESTARTSYS.

debug_id is the call debugging ID to be used for tracing. This can be obtained by atomically incrementing rxrpc_debug_id.

If this function is successful, an opaque reference to the RxRPC call is returned. The caller now holds a reference on this and it must be properly ended.

- (2) Shut down a client call:

```
void rxrpc_kernel_shutdown_call(struct socket *sock,
                                struct rxrpc_call *call);
```

This is used to shut down a previously begun call. The user_call_ID is expunged from AF_RXRPC's knowledge and will not be seen again in association with the specified call.

- (3) Release the ref on a client call:

```
void rxrpc_kernel_put_call(struct socket *sock,
                           struct rxrpc_call *call);
```

This is used to release the caller's ref on an rxrpc call.

- (4) Send data through a call:

```
typedef void (*rxrpc_notify_end_tx_t)(struct sock *sk,
                                       unsigned long user_call_ID,
                                       struct sk_buff *skb);

int rxrpc_kernel_send_data(struct socket *sock,
                           struct rxrpc_call *call,
                           struct msghdr *msg,
                           size_t len,
                           rxrpc_notify_end_tx_t notify_end_rx);
```

This is used to supply either the request part of a client call or the reply part of a server call. msg.msg iovlen and msg.msg iov specify the data buffers to be used. msg iov may not be NULL and must point exclusively to in-kernel virtual addresses. msg.msg_flags may be given MSG_MORE if there will be subsequent data sends for this call.

The msg must not specify a destination address, control data or any flags other than MSG_MORE. len is the total amount of data to transmit.

notify_end_rx can be NULL or it can be used to specify a function to be called when the call changes state to end the Tx phase. This function is called with a spinlock held to prevent the last DATA packet from being transmitted until the function returns.

- (5) Receive data from a call:

```
int rxrpc_kernel_recv_data(struct socket *sock,
                           struct rxrpc_call *call,
```

```
void *_buf,
size_t size,
size_t *_offset,
bool want_more,
u32 *_abort,
u16 *_service)
```

This is used to receive data from either the reply part of a client call or the request part of a service call. `buf` and `size` specify how much data is desired and where to store it. `*_offset` is added on to `buf` and subtracted from `size` internally; the amount copied into the buffer is added to `*_offset` before returning.

`want_more` should be true if further data will be required after this is satisfied and false if this is the last item of the receive phase.

There are three normal returns: 0 if the buffer was filled and `want_more` was true; 1 if the buffer was filled, the last DATA packet has been emptied and `want_more` was false; and -EAGAIN if the function needs to be called again.

If the last DATA packet is processed but the buffer contains less than the amount requested, EBADMSG is returned. If `want_more` wasn't set, but more data was available, EMSGSIZE is returned.

If a remote ABORT is detected, the abort code received will be stored in ```*_abort``` and ECONNABORTED will be returned.

The service ID that the call ended up with is returned into `*_service`. This can be used to see if a call got a service upgrade.

(6) Abort a call??

```
void rxrpc_kernel_abort_call(struct socket *sock,
                           struct rxrpc_call *call,
                           u32 abort_code);
```

This is used to abort a call if it's still in an abortable state. The abort code specified will be placed in the ABORT message sent.

(7) Intercept received RxRPC messages:

```
typedef void (*rxrpc_interceptor_t)(struct sock *sk,
                                    unsigned long user_call_ID,
                                    struct sk_buff *skb);

void
rxrpc_kernel_intercept_rx_messages(struct socket *sock,
                                   rxrpc_interceptor_t interceptor);
```

This installs an interceptor function on the specified AF_RXRPC socket. All messages that would otherwise wind up in the socket's Rx queue are then diverted to this function. Note

that care must be taken to process the messages in the right order to maintain DATA message sequentiality.

The interceptor function itself is provided with the address of the socket and handling the incoming message, the ID assigned by the kernel utility to the call and the socket buffer containing the message.

The skb->mark field indicates the type of message:

Mark	Meaning
RXRPC_SKB_MARK_DATA	Data message
RXRPC_SKB_MARK_FINAL_ACK	Final ACK received for an incoming call
RXRPC_SKB_MARK_BUSY	Client call rejected as server busy
RXRPC_SKB_MARK_REMOTE_ABORT	Call aborted by peer
RXRPC_SKB_MARK_NET_ERROR	Network error detected
RXRPC_SKB_MARK_LOCAL_ERROR	Local error encountered
RXRPC_SKB_MARK_NEW_CALL	New incoming call awaiting acceptance

The remote abort message can be probed with rxrpc_kernel_get_abort_code(). The two error messages can be probed with rxrpc_kernel_get_error_number(). A new call can be accepted with rxrpc_kernel_accept_call().

Data messages can have their contents extracted with the usual bunch of socket buffer manipulation functions. A data message can be determined to be the last one in a sequence with rxrpc_kernel_is_data_last(). When a data message has been used up, rxrpc_kernel_data_consumed() should be called on it.

Messages should be handled to rxrpc_kernel_free_skb() to dispose of. It is possible to get extra refs on all types of message for later freeing, but this may pin the state of a call until the message is finally freed.

(8) Accept an incoming call:

```
struct rxrpc_call *
rxrpc_kernel_accept_call(struct socket *sock,
                        unsigned long user_call_ID);
```

This is used to accept an incoming call and to assign it a call ID. This function is similar to rxrpc_kernel_begin_call() and calls accepted must be ended in the same way.

If this function is successful, an opaque reference to the RxRPC call is returned. The caller now holds a reference on this and it must be properly ended.

(9) Reject an incoming call:

```
int rxrpc_kernel_reject_call(struct socket *sock);
```

This is used to reject the first incoming call on the socket's queue with a BUSY message. -ENODATA is returned if there were no incoming calls. Other errors may be returned if the call had been aborted (-ECONNABORTED) or had timed out (-ETIME).

(10) Allocate a null key for doing anonymous security:

```
struct key *rpxrpc_get_null_key(const char *keyname);
```

This is used to allocate a null RxRPC key that can be used to indicate anonymous security for a particular domain.

- (11) Get the peer address of a call:

```
void rpxrpc_kernel_get_peer(struct socket *sock, struct rpxrpc_call *call,
                           struct sockaddr_rpxrpc *_srx);
```

This is used to find the remote peer address of a call.

- (12) Set the total transmit data size on a call:

```
void rpxrpc_kernel_set_tx_length(struct socket *sock,
                                 struct rpxrpc_call *call,
                                 s64 tx_total_len);
```

This sets the amount of data that the caller is intending to transmit on a call. It's intended to be used for setting the reply size as the request size should be set when the call is begun. tx_total_len may not be less than zero.

- (13) Get call RTT:

```
u64 rpxrpc_kernel_get_rtt(struct socket *sock, struct rpxrpc_call *call);
```

Get the RTT time to the peer in use by a call. The value returned is in nanoseconds.

- (14) Check call still alive:

```
bool rpxrpc_kernel_check_life(struct socket *sock,
                             struct rpxrpc_call *call,
                             u32 *_life);
void rpxrpc_kernel_probe_life(struct socket *sock,
                            struct rpxrpc_call *call);
```

The first function passes back in *_life a number that is updated when ACKs are received from the peer (notably including PING RESPONSE ACKs which we can elicit by sending PING ACKs to see if the call still exists on the server). The caller should compare the numbers of two calls to see if the call is still alive after waiting for a suitable interval. It also returns true as long as the call hasn't yet reached the completed state.

This allows the caller to work out if the server is still contactable and if the call is still alive on the server while waiting for the server to process a client operation.

The second function causes a ping ACK to be transmitted to try to provoke the peer into responding, which would then cause the value returned by the first function to change. Note that this must be called in TASK_RUNNING state.

- (15) Get remote client epoch:

```
u32 rpxrpc_kernel_get_epoch(struct socket *sock,
                           struct rpxrpc_call *call)
```

This allows the epoch that's contained in packets of an incoming client call to be queried. This value is returned. The function always successful if the call is still in progress. It

shouldn't be called once the call has expired. Note that calling this on a local client call only returns the local epoch.

This value can be used to determine if the remote client has been restarted as it shouldn't change otherwise.

- (16) Set the maximum lifespan on a call:

```
void rxrpc_kernel_set_max_life(struct socket *sock,
                               struct rxrpc_call *call,
                               unsigned long hard_timeout)
```

This sets the maximum lifespan on a call to `hard_timeout` (which is in jiffies). In the event of the timeout occurring, the call will be aborted and -ETIME or -ETIMEDOUT will be returned.

- (17) Apply the RXRPC_MIN_SECURITY_LEVEL sockopt to a socket from within in the kernel:

```
int rxrpc_sock_set_min_security_level(struct sock *sk,
                                       unsigned int val);
```

This specifies the minimum security level required for calls on this socket.

91.3 Configurable Parameters

The RxRPC protocol driver has a number of configurable parameters that can be adjusted through sysctls in `/proc/net/rxrpc/`:

- (1) `req_ack_delay`

The amount of time in milliseconds after receiving a packet with the request-ack flag set before we honour the flag and actually send the requested ack.

Usually the other side won't stop sending packets until the advertised reception window is full (to a maximum of 255 packets), so delaying the ACK permits several packets to be ACK'd in one go.

- (2) `soft_ack_delay`

The amount of time in milliseconds after receiving a new packet before we generate a soft-ACK to tell the sender that it doesn't need to resend.

- (3) `idle_ack_delay`

The amount of time in milliseconds after all the packets currently in the received queue have been consumed before we generate a hard-ACK to tell the sender it can free its buffers, assuming no other reason occurs that we would send an ACK.

- (4) `resend_timeout`

The amount of time in milliseconds after transmitting a packet before we transmit it again, assuming no ACK is received from the receiver telling us they got it.

- (5) `max_call_lifetime`

The maximum amount of time in seconds that a call may be in progress before we preemptively kill it.

(6) `dead_call_expiry`

The amount of time in seconds before we remove a dead call from the call list. Dead calls are kept around for a little while for the purpose of repeating ACK and ABORT packets.

(7) `connection_expiry`

The amount of time in seconds after a connection was last used before we remove it from the connection list. While a connection is in existence, it serves as a placeholder for negotiated security; when it is deleted, the security must be renegotiated.

(8) `transport_expiry`

The amount of time in seconds after a transport was last used before we remove it from the transport list. While a transport is in existence, it serves to anchor the peer data and keeps the connection ID counter.

(9) `rxrpc_rx_window_size`

The size of the receive window in packets. This is the maximum number of unconsumed received packets we're willing to hold in memory for any particular call.

(10) `rxrpc_rx_mtu`

The maximum packet MTU size that we're willing to receive in bytes. This indicates to the peer whether we're willing to accept jumbo packets.

(11) `rxrpc_rx_jumbo_max`

The maximum number of packets that we're willing to accept in a jumbo packet. Non-terminal packets in a jumbo packet must contain a four byte header plus exactly 1412 bytes of data. The terminal packet must contain a four byte header plus any amount of data. In any event, a jumbo packet may not exceed `rxrpc_rx_mtu` in size.

LINUX KERNEL SCTP

This is the current BETA release of the Linux Kernel SCTP reference implementation.

SCTP (Stream Control Transmission Protocol) is a IP based, message oriented, reliable transport protocol, with congestion control, support for transparent multi-homing, and multiple ordered streams of messages. RFC2960 defines the core protocol. The IETF SIGTRAN working group originally developed the SCTP protocol and later handed the protocol over to the Transport Area (TSVWG) working group for the continued evolution of SCTP as a general purpose transport.

See the IETF website (<http://www.ietf.org>) for further documents on SCTP. See <http://www.ietf.org/rfc/rfc2960.txt>

The initial project goal is to create an Linux kernel reference implementation of SCTP that is RFC 2960 compliant and provides an programming interface referred to as the UDP-style API of the Sockets Extensions for SCTP, as proposed in IETF Internet-Drafts.

92.1 Caveats

- lksctp can be built as statically or as a module. However, be aware that module removal of lksctp is not yet a safe activity.
- There is tentative support for IPv6, but most work has gone towards implementation and testing lksctp on IPv4.

For more information, please visit the lksctp project website:

<http://www.sf.net/projects/lksctp>

Or contact the lksctp developers through the mailing list:

<linux-sctp@vger.kernel.org>

LSM/SELINUX SECID

flowi structure:

The secid member in the flow structure is used in LSMs (e.g. SELinux) to indicate the label of the flow. This label of the flow is currently used in selecting matching labeled xfrm(s).

If this is an outbound flow, the label is derived from the socket, if any, or the incoming packet this flow is being generated as a response to (e.g. tcp resets, timewait ack, etc.). It is also conceivable that the label could be derived from other sources such as process context, device, etc., in special cases, as may be appropriate.

If this is an inbound flow, the label is derived from the IPSec security associations, if any, used by the packet.

SEG6 SYSFS VARIABLES

94.1 /proc/sys/net/conf/<iface>/seg6_* variables:

seg6_enabled - BOOL

Accept or drop SR-enabled IPv6 packets on this interface.

Relevant packets are those with SRH present and DA = local.

- 0 - disabled (default)
- not 0 - enabled

seg6_require_hmac - INTEGER

Define HMAC policy for ingress SR-enabled packets on this interface.

- -1 - Ignore HMAC field
- 0 - Accept SR packets without HMAC, validate SR packets with HMAC
- 1 - Drop SR packets without HMAC, validate SR packets with HMAC

Default is 0.

seg6_flowlabel - INTEGER

Controls the behaviour of computing the flowlabel of outer IPv6 header in case of SR T.encaps

-1	set flowlabel to zero.
0	copy flowlabel from Inner packet in case of Inner IPv6 (Set flowlabel to 0 in case IPv4/L2)
1	Compute the flowlabel using seg6_make_flowlabel()

Default is 0.

STRUCT SK_BUFF

sk_buff is the main networking structure representing a packet.

95.1 Basic *sk_buff* geometry

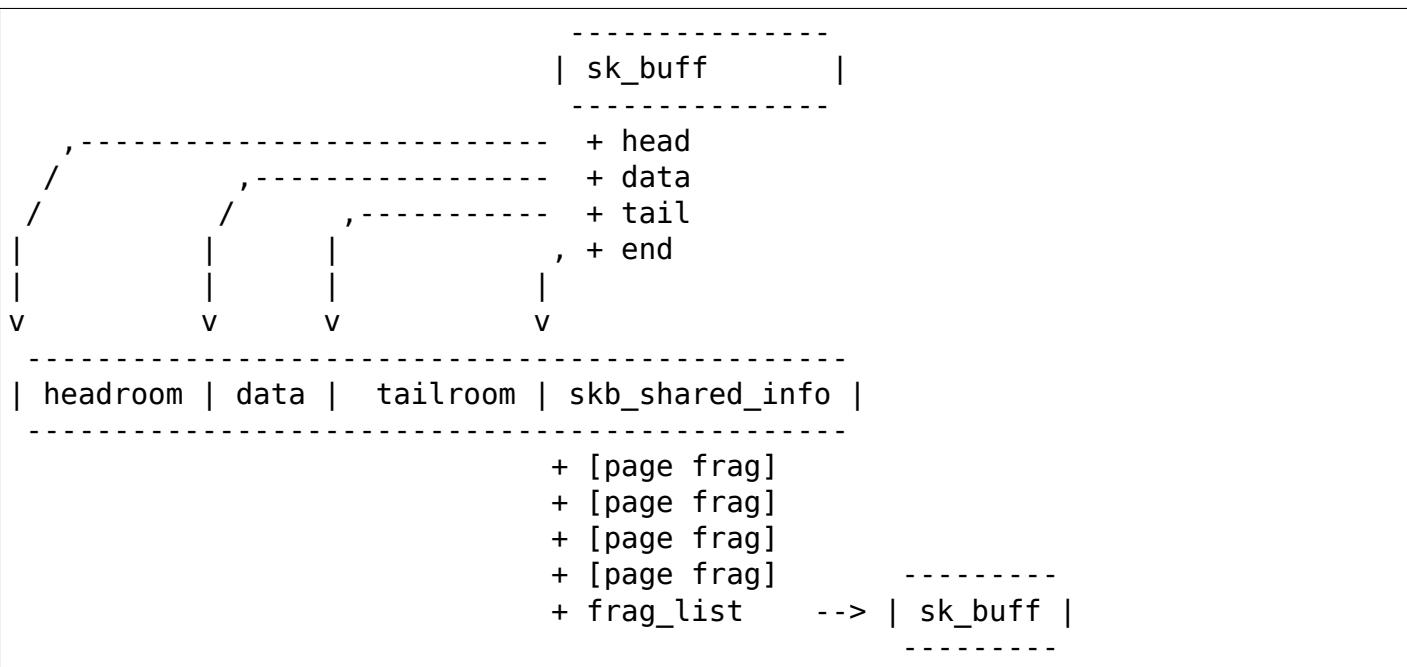
struct sk_buff itself is a metadata structure and does not hold any packet data. All the data is held in associated buffers.

sk_buff.head points to the main "head" buffer. The head buffer is divided into two parts:

- data buffer, containing headers and sometimes payload; this is the part of the skb operated on by the common helpers such as *skb_put()* or *skb_pull()*;
- shared info (*struct skb_shared_info*) which holds an array of pointers to read-only data in the (page, offset, length) format.

Optionally *skb_shared_info.frag_list* may point to another skb.

Basic diagram may look like this:



95.2 Shared skbs and skb clones

`sk_buff.users` is a simple refcount allowing multiple entities to keep a `struct sk_buff` alive. Skbs with a `sk_buff.users != 1` are referred to as shared skbs (see `skb_shared()`).

`skb_clone()` allows for fast duplication of skbs. None of the data buffers get copied, but caller gets a new metadata struct (`struct sk_buff`). `&skb_shared_info.refcount` indicates the number of skbs pointing at the same packet data (i.e. clones).

95.3 dataref and headerless skbs

Transport layers send out clones of payload skbs they hold for retransmissions. To allow lower layers of the stack to prepend their headers we split `skb_shared_info.dataref` into two halves. The lower 16 bits count the overall number of references. The higher 16 bits indicate how many of the references are payload-only. `skb_header_cloned()` checks if skb is allowed to add / write the headers.

The creator of the skb (e.g. TCP) marks its skb as `sk_buff.nohdr` (via `__skb_header_release()`). Any clone created from marked skb will get `sk_buff.hdr_len` populated with the available headroom. If there's the only clone in existence it's able to modify the headroom at will. The sequence of calls inside the transport layer is:

```
<alloc skb>
skb_reserve()
__skb_header_release()
skb_clone()
// send the clone down the stack
```

This is not a very generic construct and it depends on the transport layers doing the right thing. In practice there's usually only one payload-only skb. Having multiple payload-only skbs with different lengths of `hdr_len` is not possible. The payload-only skbs should never leave their owner.

95.4 Checksum information

The interface for checksum offload between the stack and networking drivers is as follows...

95.4.1 IP checksum related features

Drivers advertise checksum offload capabilities in the features of a device. From the stack's point of view these are capabilities offered by the driver. A driver typically only advertises features that it is capable of offloading to its device.

Table 1: Checksum related device features

NETIF_F_HW_CSUM	The driver (or its device) is able to compute one IP (one's complement) checksum for any combination of protocols or protocol layering. The checksum is computed and set in a packet per the CHECKSUM_PARTIAL interface (see below).
NETIF_F_IP_CSUM	Driver (device) is only able to checksum plain TCP or UDP packets over IPv4. These are specifically unencapsulated packets of the form IPv4 TCP or IPv4 UDP where the Protocol field in the IPv4 header is TCP or UDP. The IPv4 header may contain IP options. This feature cannot be set in features for a device with NETIF_F_HW_CSUM also set. This feature is being DEPRECATED (see below).
NETIF_F_IPV6_CSUM	Driver (device) is only able to checksum plain TCP or UDP packets over IPv6. These are specifically unencapsulated packets of the form IPv6 TCP or IPv6 UDP where the Next Header field in the IPv6 header is either TCP or UDP. IPv6 extension headers are not supported with this feature. This feature cannot be set in features for a device with NETIF_F_HW_CSUM also set. This feature is being DEPRECATED (see below).
NETIF_F_RXCSUM	Driver (device) performs receive checksum offload. This flag is only used to disable the RX checksum feature for a device. The stack will accept receive checksum indication in packets received on a device regardless of whether NETIF_F_RXCSUM is set.

95.4.2 Checksumming of received packets by device

Indication of checksum verification is set in `sk_buff.ip_summed`. Possible values are:

- `CHECKSUM_NONE`

Device did not checksum this packet e.g. due to lack of capabilities. The packet contains full (though not verified) checksum in packet but not in `skb->csum`. Thus, `skb->csum` is undefined in this case.

- `CHECKSUM_UNNECESSARY`

The hardware you're dealing with doesn't calculate the full checksum (as in `CHECKSUM_COMPLETE`), but it does parse headers and verify checksums for specific protocols. For such packets it will set `CHECKSUM_UNNECESSARY` if their checksums are okay. `sk_buff.csum` is still undefined in this case though. A driver or device must never modify the checksum field in the packet even if checksum is verified.

`CHECKSUM_UNNECESSARY` is applicable to following protocols:

- TCP: IPv6 and IPv4.
- UDP: IPv4 and IPv6. A device may apply `CHECKSUM_UNNECESSARY` to a zero UDP checksum for either IPv4 or IPv6, the networking stack may perform further validation in this case.
- GRE: only if the checksum is present in the header.
- SCTP: indicates the CRC in SCTP header has been validated.
- FCOE: indicates the CRC in FC frame has been validated.

`sk_buff.csum_level` indicates the number of consecutive checksums found in the packet minus one that have been verified as `CHECKSUM_UNNECESSARY`. For instance if a device receives an IPv6->UDP->GRE->IPv4->TCP packet and a device is able to verify the checksums for UDP (possibly zero), GRE (checksum flag is set) and TCP, `sk_buff.csum_level` would be set to two. If the device were only able to verify the UDP checksum and not GRE, either because it doesn't support GRE checksum or because GRE checksum is bad, `skb->csum_level` would be set to zero (TCP checksum is not considered in this case).

- `CHECKSUM_COMPLETE`

This is the most generic way. The device supplied checksum of the `_whole_` packet as seen by `netif_rx()` and fills in `sk_buff.csum`. This means the hardware doesn't need to parse L3/L4 headers to implement this.

Notes:

- Even if device supports only some protocols, but is able to produce `skb->csum`, it MUST use `CHECKSUM_COMPLETE`, not `CHECKSUM_UNNECESSARY`.
- `CHECKSUM_COMPLETE` is not applicable to SCTP and FCoE protocols.

- `CHECKSUM_PARTIAL`

A checksum is set up to be offloaded to a device as described in the output description for `CHECKSUM_PARTIAL`. This may occur on a packet received directly from another Linux OS, e.g., a virtualized Linux kernel on the same host, or it may be set in the input path in GRO or remote checksum offload. For the purposes of checksum verification, the checksum referred to by `skb->csum_start + skb->csum_offset` and any preceding checksums in the packet are considered verified. Any checksums in the packet that are after the checksum being offloaded are not considered to be verified.

95.4.3 Checksumming on transmit for non-GSO

The stack requests checksum offload in the `sk_buff.ip_summed` for a packet. Values are:

- `CHECKSUM_PARTIAL`

The driver is required to checksum the packet as seen by `hard_start_xmit()` from `sk_buff.csum_start` up to the end, and to record/write the checksum at offset `sk_buff.csum_start + sk_buff.csum_offset`. A driver may verify that the `csum_start` and `csum_offset` values are valid values given the length and offset of the packet, but it should not attempt to validate that the checksum refers to a legitimate transport layer checksum -- it is the purview of the stack to validate that `csum_start` and `csum_offset` are set correctly.

When the stack requests checksum offload for a packet, the driver MUST ensure that the checksum is set correctly. A driver can either offload the checksum calculation to the device, or call `skb_checksum_help` (in the case that the device does not support offload for a particular checksum).

`NETIF_F_IP_CSUM` and `NETIF_F_IPV6_CSUM` are being deprecated in favor of `NETIF_F_HW_CSUM`. New devices should use `NETIF_F_HW_CSUM` to indicate checksum offload capability. `skb_csum_hwoffload_help()` can be called to resolve `CHECKSUM_PARTIAL` based on network device checksumming capabilities: if a packet does not match them, `skb_checksum_help()` or `skb_crc32c_help()` (depending on the value of `sk_buff.csum_not_inet`, see [Non-IP checksum \(CRC\) offloads](#)) is called to resolve the checksum.

- `CHECKSUM_NONE`

The skb was already checksummed by the protocol, or a checksum is not required.

- `CHECKSUM_UNNECESSARY`

This has the same meaning as `CHECKSUM_NONE` for checksum offload on output.

- `CHECKSUM_COMPLETE`

Not used in checksum output. If a driver observes a packet with this value set in skbuff, it should treat the packet as if `CHECKSUM_NONE` were set.

95.4.4 Non-IP checksum (CRC) offloads

<code>NETIF_F_SCTP_CRC</code>	This feature indicates that a device is capable of offloading the SCTP CRC in a packet. To perform this offload the stack will set <code>csum_start</code> and <code>csum_offset</code> accordingly, set <code>ip_summed</code> to <code>CHECKSUM_PARTIAL</code> and set <code>csum_not_inet</code> to 1, to provide an indication in the skbuff that the <code>CHECKSUM_PARTIAL</code> refers to CRC32c. A driver that supports both IP checksum offload and SCTP CRC32c offload must verify which offload is configured for a packet by testing the value of <code>sk_buff.csum_not_inet</code> ; <code>skb_crc32c_csum_help()</code> is provided to resolve <code>CHECKSUM_PARTIAL</code> on skb's where <code>csum_not_inet</code> is set to 1.
<code>NETIF_F_FCOE_CRC</code>	This feature indicates that a device is capable of offloading the FCOE CRC in a packet. To perform this offload the stack will set <code>ip_summed</code> to <code>CHECKSUM_PARTIAL</code> and set <code>csum_start</code> and <code>csum_offset</code> accordingly. Note that there is no indication in the skbuff that the <code>CHECKSUM_PARTIAL</code> refers to an FCOE checksum, so a driver that supports both IP checksum offload and FCOE CRC offload must verify which offload is configured for a packet, presumably by inspecting packet headers.

95.4.5 Checksumming on output with GSO

In the case of a GSO packet (`skb_is_gso()` is true), checksum offload is implied by the `SKB_GSO_*` flags in `gso_type`. Most obviously, if the `gso_type` is `SKB_GSO_TCPV4` or `SKB_GSO_TCPV6`, TCP checksum offload as part of the GSO operation is implied. If a checksum is being offloaded with GSO then `ip_summed` is `CHECKSUM_PARTIAL`, and both `csum_start` and `csum_offset` are set to refer to the outermost checksum being offloaded (two offloaded checksums are possible with UDP encapsulation).

SMC SYSCTL

96.1 /proc/sys/net/smci/* Variables

autocorking_size - INTEGER

Setting SMC auto corking size: SMC auto corking is like TCP auto corking from the application's perspective of view. When applications do consecutive small write()/sendmsg() system calls, we try to coalesce these small writes as much as possible, to lower total amount of CDC and RDMA Write been sent. autocorking_size limits the maximum corked bytes that can be sent to the under device in 1 single sending. If set to 0, the SMC auto corking is disabled. Applications can still use TCP_CORK for optimal behavior when they know how/when to uncork their sockets.

Default: 64K

smcr_buf_type - INTEGER

Controls which type of sndbufs and RMBs to use in later newly created SMC-R link group. Only for SMC-R.

Default: 0 (physically contiguous sndbufs and RMBs)

Possible values:

- 0 - Use physically contiguous buffers
- 1 - Use virtually contiguous buffers
- 2 - Mixed use of the two types. Try physically contiguous buffers first. If not available, use virtually contiguous buffers then.

smcr_testlink_time - INTEGER

How frequently SMC-R link sends out TEST_LINK LLC messages to confirm viability, after the last activity of connections on it. Value 0 means disabling TEST_LINK.

Default: 30 seconds.

wmem - INTEGER

Initial size of send buffer used by SMC sockets.

The minimum value is 16KiB and there is no hard limit for max value, but only allowed 512KiB for SMC-R and 1MiB for SMC-D.

Default: 64KiB

rmem - INTEGER

Initial size of receive buffer (RMB) used by SMC sockets.

The minimum value is 16KiB and there is no hard limit for max value, but only allowed 512KiB for SMC-R and 1MiB for SMC-D.

Default: 64KiB

smcr_max_links_per_lgr - INTEGER

Controls the max number of links can be added to a SMC-R link group. Notice that the actual number of the links added to a SMC-R link group depends on the number of RDMA devices exist in the system. The acceptable value ranges from 1 to 2. Only for SMC-R v2.1 and later.

Default: 2

smcr_max_conns_per_lgr - INTEGER

Controls the max number of connections can be added to a SMC-R link group. The acceptable value ranges from 16 to 255. Only for SMC-R v2.1 and later.

Default: 255

INTERFACE STATISTICS

97.1 Overview

This document is a guide to Linux network interface statistics.

There are three main sources of interface statistics in Linux:

- standard interface statistics based on *struct rtnl_link_stats64*;
- protocol-specific statistics; and
- driver-defined statistics available via ethtool.

97.1.1 Standard interface statistics

There are multiple interfaces to reach the standard statistics. Most commonly used is the *ip* command from *iproute2*:

```
$ ip -s -s link show dev ens4ulul
6: ens4ulul: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP
    link/ether 48:2a:e3:4c:b1:d1 brd ff:ff:ff:ff:ff:ff
    RX: bytes packets errors dropped overrun mcast
        74327665117 69016965 0          0          0          0
    RX errors: length crc frame fifo missed
        0          0          0          0          0
    TX: bytes packets errors dropped carrier collsns
        21405556176 44608960 0          0          0          0
    TX errors: aborted fifo window heartbeat transns
        0          0          0          0          128
    altname enp58s0ulul
```

Note that *-s* has been specified twice to see all members of *struct rtnl_link_stats64*. If *-s* is specified once the detailed errors won't be shown.

ip supports JSON formatting via the *-j* option.

97.1.2 Protocol-specific statistics

Protocol-specific statistics are exposed via relevant interfaces, the same interfaces as are used to configure them.

ethtool

Ethtool exposes common low-level statistics. All the standard statistics are expected to be maintained by the device, not the driver (as opposed to driver-defined stats described in the next section which mix software and hardware stats). For devices which contain unmanaged switches (e.g. legacy SR-IOV or multi-host NICs) the events counted may not pertain exclusively to the packets destined to the local host interface. In other words the events may be counted at the network port (MAC/PHY blocks) without separation for different host side (PCIe) devices. Such ambiguity must not be present when internal switch is managed by Linux (so called switchdev mode for NICs).

Standard ethtool statistics can be accessed via the interfaces used for configuration. For example ethtool interface used to configure pause frames can report corresponding hardware counters:

```
$ ethtool --include-statistics -a eth0
Pause parameters for eth0:
Autonegotiate:          on
RX:                     on
TX:                     on
Statistics:
  tx_pause_frames: 1
  rx_pause_frames: 1
```

General Ethernet statistics not associated with any particular functionality are exposed via `ethtool -S $ifc` by specifying the `--groups` parameter:

```
$ ethtool -S eth0 --groups eth-phy eth-mac eth-ctrl rmon
Stats for eth0:
eth-phy-SymbolErrorDuringCarrier: 0
eth-mac-FramesTransmittedOK: 1
eth-mac-FrameTooLongErrors: 1
eth-ctrl-MACControlFramesTransmitted: 1
eth-ctrl-MACControlFramesReceived: 0
eth-ctrl-UnsupportedOpcodesReceived: 1
rmon-etherStatsUndersizePkts: 1
rmon-etherStatsJabbers: 0
rmon-rx-etherStatsPkts640ctets: 1
rmon-rx-etherStatsPkts65to1270ctets: 0
rmon-rx-etherStatsPkts128to2550ctets: 0
rmon-tx-etherStatsPkts640ctets: 2
rmon-tx-etherStatsPkts65to1270ctets: 3
rmon-tx-etherStatsPkts128to2550ctets: 0
```

97.1.3 Driver-defined statistics

Driver-defined ethtool statistics can be dumped using `ethtool -S $ifc`, e.g.:

```
$ ethtool -S ens4ulul
NIC statistics:
  tx_single_collisions: 0
  tx_multi_collisions: 0
```

97.2 uAPIs

97.2.1 procfs

The historical `/proc/net/dev` text interface gives access to the list of interfaces as well as their statistics.

Note that even though this interface is using `struct rtnl_link_stats64` internally it combines some of the fields.

97.2.2 sysfs

Each device directory in sysfs contains a `statistics` directory (e.g. `/sys/class/net/lo/statistics/`) with files corresponding to members of `struct rtnl_link_stats64`.

This simple interface is convenient especially in constrained/embedded environments without access to tools. However, it's inefficient when reading multiple stats as it internally performs a full dump of `struct rtnl_link_stats64` and reports only the stat corresponding to the accessed file.

Sysfs files are documented in *Documentation/ABI/testing/sysfs-class-net-statistics*.

97.2.3 netlink

`rtnetlink (NETLINK_ROUTE)` is the preferred method of accessing `struct rtnl_link_stats64` stats.

Statistics are reported both in the responses to link information requests (`RTM_GETLINK`) and statistic requests (`RTM_GETSTATS`, when `IFLA_STATS_LINK_64` bit is set in the `.filter_mask` of the request).

97.2.4 ethtool

Ethtool IOCTL interface allows drivers to report implementation specific statistics. Historically it has also been used to report statistics for which other APIs did not exist, like per-device-queue statistics, or standard-based statistics (e.g. RFC 2863).

Statistics and their string identifiers are retrieved separately. Identifiers via `ETH_TOOL_GSTRINGS` with `string_set` set to `ETH_SS_STATS`, and values via `ETH_TOOL_GSTATS`. User space should use `ETHTOOL_GDRVINFO` to retrieve the number of statistics (`.n_stats`).

97.2.5 ethtool-netlink

Ethtool netlink is a replacement for the older IOCTL interface.

Protocol-related statistics can be requested in get commands by setting the *ETHTOOL_FLAG_STATS* flag in *ETHTOOL_A_HEADER_FLAGS*. Currently statistics are supported in the following commands:

- *ETHTOOL_MSG_PAUSE_GET*
- *ETHTOOL_MSG_FEC_GET*
- *ETHTOOL_MSG_MM_GET*

97.2.6 debugfs

Some drivers expose extra statistics via *debugfs*.

97.3 struct rtnl_link_stats64

struct **rtnl_link_stats64**

The main device statistics structure.

Definition:

```
struct rtnl_link_stats64 {
    __u64 rx_packets;
    __u64 tx_packets;
    __u64 rx_bytes;
    __u64 tx_bytes;
    __u64 rx_errors;
    __u64 tx_errors;
    __u64 rx_dropped;
    __u64 tx_dropped;
    __u64 multicast;
    __u64 collisions;
    __u64 rx_length_errors;
    __u64 rx_over_errors;
    __u64 rx_crc_errors;
    __u64 rx_frame_errors;
    __u64 rx_fifo_errors;
    __u64 rx_missed_errors;
    __u64 tx_aborted_errors;
    __u64 tx_carrier_errors;
    __u64 tx_fifo_errors;
    __u64 tx_heartbeat_errors;
    __u64 tx_window_errors;
    __u64 rx_compressed;
    __u64 tx_compressed;
    __u64 rx_nohandler;
    __u64 rx_otherhost_dropped;
};
```

Members

rx_packets

Number of good packets received by the interface. For hardware interfaces counts all good packets received from the device by the host, including packets which host had to drop at various stages of processing (even in the driver).

tx_packets

Number of packets successfully transmitted. For hardware interfaces counts packets which host was able to successfully hand over to the device, which does not necessarily mean that packets had been successfully transmitted out of the device, only that device acknowledged it copied them out of host memory.

rx_bytes

Number of good received bytes, corresponding to **rx_packets**.

For IEEE 802.3 devices should count the length of Ethernet Frames excluding the FCS.

tx_bytes

Number of good transmitted bytes, corresponding to **tx_packets**.

For IEEE 802.3 devices should count the length of Ethernet Frames excluding the FCS.

rx_errors

Total number of bad packets received on this network device. This counter must include events counted by **rx_length_errors**, **rx_crc_errors**, **rx_frame_errors** and other errors not otherwise counted.

tx_errors

Total number of transmit problems. This counter must include events counter by **tx_aborted_errors**, **tx_carrier_errors**, **tx_fifo_errors**, **tx_heartbeat_errors**, **tx_window_errors** and other errors not otherwise counted.

rx_dropped

Number of packets received but not processed, e.g. due to lack of resources or unsupported protocol. For hardware interfaces this counter may include packets discarded due to L2 address filtering but should not include packets dropped by the device due to buffer exhaustion which are counted separately in **rx_missed_errors** (since procfs folds those two counters together).

tx_dropped

Number of packets dropped on their way to transmission, e.g. due to lack of resources.

multicast

Multicast packets received. For hardware interfaces this statistic is commonly calculated at the device level (unlike **rx_packets**) and therefore may include packets which did not reach the host.

For IEEE 802.3 devices this counter may be equivalent to:

- 30.3.1.1.21 aMulticastFramesReceivedOK

collisions

Number of collisions during packet transmissions.

rx_length_errors

Number of packets dropped due to invalid length. Part of aggregate "frame" errors in

/proc/net/dev.

For IEEE 802.3 devices this counter should be equivalent to a sum of the following attributes:

- 30.3.1.1.23 aInRangeLengthErrors
- 30.3.1.1.24 aOutOfRangeLengthField
- 30.3.1.1.25 aFrameTooLongErrors

rx_over_errors

Receiver FIFO overflow event counter.

Historically the count of overflow events. Such events may be reported in the receive descriptors or via interrupts, and may not correspond one-to-one with dropped packets.

The recommended interpretation for high speed interfaces is - number of packets dropped because they did not fit into buffers provided by the host, e.g. packets larger than MTU or next buffer in the ring was not available for a scatter transfer.

Part of aggregate "frame" errors in */proc/net/dev*.

This statistics was historically used interchangeably with **rx_fifo_errors**.

This statistic corresponds to hardware events and is not commonly used on software devices.

rx_crc_errors

Number of packets received with a CRC error. Part of aggregate "frame" errors in */proc/net/dev*.

For IEEE 802.3 devices this counter must be equivalent to:

- 30.3.1.1.6 aFrameCheckSequenceErrors

rx_frame_errors

Receiver frame alignment errors. Part of aggregate "frame" errors in */proc/net/dev*.

For IEEE 802.3 devices this counter should be equivalent to:

- 30.3.1.1.7 aAlignmentErrors

rx_fifo_errors

Receiver FIFO error counter.

Historically the count of overflow events. Those events may be reported in the receive descriptors or via interrupts, and may not correspond one-to-one with dropped packets.

This statistics was used interchangeably with **rx_over_errors**. Not recommended for use in drivers for high speed interfaces.

This statistic is used on software devices, e.g. to count software packet queue overflow (can) or sequencing errors (GRE).

rx_missed_errors

Count of packets missed by the host. Folded into the "drop" counter in */proc/net/dev*.

Counts number of packets dropped by the device due to lack of buffer space. This usually indicates that the host interface is slower than the network interface, or host is not keeping up with the receive packet rate.

This statistic corresponds to hardware events and is not used on software devices.

tx_aborted_errors

Part of aggregate "carrier" errors in */proc/net/dev*. For IEEE 802.3 devices capable of half-duplex operation this counter must be equivalent to:

- 30.3.1.1.11 aFramesAbortedDueToXSColls

High speed interfaces may use this counter as a general device discard counter.

tx_carrier_errors

Number of frame transmission errors due to loss of carrier during transmission. Part of aggregate "carrier" errors in */proc/net/dev*.

For IEEE 802.3 devices this counter must be equivalent to:

- 30.3.1.1.13 aCarrierSenseErrors

tx_fifo_errors

Number of frame transmission errors due to device FIFO underrun / underflow. This condition occurs when the device begins transmission of a frame but is unable to deliver the entire frame to the transmitter in time for transmission. Part of aggregate "carrier" errors in */proc/net/dev*.

tx_heartbeat_errors

Number of Heartbeat / SQE Test errors for old half-duplex Ethernet. Part of aggregate "carrier" errors in */proc/net/dev*.

For IEEE 802.3 devices possibly equivalent to:

- 30.3.2.1.4 aSQETestErrors

tx_window_errors

Number of frame transmission errors due to late collisions (for Ethernet - after the first 64B of transmission). Part of aggregate "carrier" errors in */proc/net/dev*.

For IEEE 802.3 devices this counter must be equivalent to:

- 30.3.1.1.10 aLateCollisions

rx_compressed

Number of correctly received compressed packets. This counters is only meaningful for interfaces which support packet compression (e.g. CSLIP, PPP).

tx_compressed

Number of transmitted compressed packets. This counters is only meaningful for interfaces which support packet compression (e.g. CSLIP, PPP).

rx_nohandler

Number of packets received on the interface but dropped by the networking stack because the device is not designated to receive packets (e.g. backup link in a bond).

rx_otherhost_dropped

Number of packets dropped due to mismatch in destination MAC address.

97.4 Notes for driver authors

Drivers should report all statistics which have a matching member in `struct rtnl_link_stats64` exclusively via `.ndo_get_stats64`. Reporting such standard stats via ethtool or debugfs will not be accepted.

Drivers must ensure best possible compliance with `struct rtnl_link_stats64`. Please note for example that detailed error statistics must be added into the general `rx_error / tx_error` counters.

The `.ndo_get_stats64` callback can not sleep because of accesses via `/proc/net/dev`. If driver may sleep when retrieving the statistics from the device it should do so periodically asynchronously and only return a recent copy from `.ndo_get_stats64`. Ethtool interrupt coalescing interface allows setting the frequency of refreshing statistics, if needed.

Retrieving ethtool statistics is a multi-syscall process, drivers are advised to keep the number of statistics constant to avoid race conditions with user space trying to read them.

Statistics must persist across routine operations like bringing the interface down and up.

97.4.1 Kernel-internal data structures

The following structures are internal to the kernel, their members are translated to netlink attributes when dumped. Drivers must not overwrite the statistics they don't report with 0.

- `ethtool_pause_stats()`
- `ethtool_fec_stats()`

STREAM PARSER (STRPARSER)

98.1 Introduction

The stream parser (strparser) is a utility that parses messages of an application layer protocol running over a data stream. The stream parser works in conjunction with an upper layer in the kernel to provide kernel support for application layer messages. For instance, Kernel Connection Multiplexor (KCM) uses the Stream Parser to parse messages using a BPF program.

The strparser works in one of two modes: receive callback or general mode.

In receive callback mode, the strparser is called from the `data_ready` callback of a TCP socket. Messages are parsed and delivered as they are received on the socket.

In general mode, a sequence of skbs are fed to strparser from an outside source. Message are parsed and delivered as the sequence is processed. This modes allows strparser to be applied to arbitrary streams of data.

98.2 Interface

The API includes a context structure, a set of callbacks, utility functions, and a `data_ready` function for receive callback mode. The callbacks include a `parse_msg` function that is called to perform parsing (e.g. BPF parsing in case of KCM), and a `recv_msg` function that is called when a full message has been completed.

98.3 Functions

```
strp_init(struct strparser *strp, struct sock *sk,  
          const struct strp_callbacks *cb)
```

Called to initialize a stream parser. `strp` is a struct of type `strparser` that is allocated by the upper layer. `sk` is the TCP socket associated with the stream parser for use with receive callback mode; in general mode this is set to `NULL`. Callbacks are called by the stream parser (the callbacks are listed below).

```
void strp_pause(struct strparser *strp)
```

Temporarily pause a stream parser. Message parsing is suspended and no new messages are delivered to the upper layer.

```
void strp_unpause(struct strparser *strp)
```

Unpause a paused stream parser.

```
void strp_stop(struct strparser *strp);
```

strp_stop is called to completely stop stream parser operations. This is called internally when the stream parser encounters an error, and it is called from the upper layer to stop parsing operations.

```
void strp_done(struct strparser *strp);
```

strp_done is called to release any resources held by the stream parser instance. This must be called after the stream processor has been stopped.

```
int strp_process(struct strparser *strp, struct sk_buff *orig_
→skb,
                  unsigned int orig_offset, size_t orig_len,
                  size_t max_msg_size, long timeo)
```

strp_process is called in general mode for a stream parser to parse an sk_buff. The number of bytes processed or a negative error number is returned. Note that strp_process does not consume the sk_buff. max_msg_size is maximum size the stream parser will parse. timeo is timeout for completing a message.

```
void strp_data_ready(struct strparser *strp);
```

The upper layer calls strp_tcp_data_ready when data is ready on the lower socket for strparser to process. This should be called from a data_ready callback that is set on the socket. Note that maximum messages size is the limit of the receive socket buffer and message timeout is the receive timeout for the socket.

```
void strp_check_rcv(struct strparser *strp);
```

strp_check_rcv is called to check for new messages on the socket. This is normally called at initialization of a stream parser instance or after strp_unpause.

98.4 Callbacks

There are six callbacks:

```
int (*parse_msg)(struct strparser *strp, struct sk_buff *skb);
```

parse_msg is called to determine the length of the next message in the stream. The upper layer must implement this function. It should parse the sk_buff as containing the headers for the next application layer message in the stream.

The skb->cb in the input skb is a struct strp_msg. Only the offset field is relevant in parse_msg and gives the offset where the message starts in the skb.

The return values of this function are:

>0	indicates length of successfully parsed message
0	indicates more data must be received to parse the message
-ESTRPIPE	current message should not be processed by the kernel, return control of the socket to userspace which can proceed to read the messages itself
other < 0	Error in parsing, give control back to userspace assuming that synchronization is lost and the stream is unrecoverable (application expected to close TCP socket)

In the case that an error is returned (return value is less than zero) and the parser is in receive callback mode, then it will set the error on TCP socket and wake it up. If `parse_msg` returned -ESTRPIPE and the stream parser had previously read some bytes for the current message, then the error set on the attached socket is ENODATA since the stream is unrecoverable in that case.

```
void (*lock)(struct strparser *strup)
```

The lock callback is called to lock the strp structure when the strparser is performing an asynchronous operation (such as processing a timeout). In receive callback mode the default function is to `lock_sock` for the associated socket. In general mode the callback must be set appropriately.

```
void (*unlock)(struct strparser *strup)
```

The unlock callback is called to release the lock obtained by the lock callback. In receive callback mode the default function is `release_sock` for the associated socket. In general mode the callback must be set appropriately.

```
void (*recv_msg)(struct strparser *strup, struct sk_buff *skb);
```

`recv_msg` is called when a full message has been received and is queued. The callee must consume the `sk_buff`; it can call `strup_pause` to prevent any further messages from being received in `recv_msg` (see `strup_pause` above). This callback must be set.

The `skb->cb` in the input `skb` is a `struct strp_msg`. This struct contains two fields: `offset` and `full_len`. `Offset` is where the message starts in the `skb`, and `full_len` is the the length of the message. `skb->len - offset` may be greater then `full_len` since strparser does not trim the `skb`.

```
int (*read_sock_done)(struct strparser *strup, int err);

read_sock_done is called when the stream parser is done reading
the TCP socket in receive callback mode. The stream parser may
read multiple messages in a loop and this function allows cleanup
to occur when exiting the loop. If the callback is not set (NULL
in strup_init) a default function is used.

::

void (*abort_parser)(struct strparser *strup, int err);
```

This function is called when stream parser encounters an error in parsing. The default function stops the stream parser and sets the error in the socket if the parser is in receive callback mode. The default function can be changed by setting the callback to non-NUL in strp_init.

98.5 Statistics

Various counters are kept for each stream parser instance. These are in the strp_stats structure. strp_aggr_stats is a convenience structure for accumulating statistics for multiple stream parser instances. save_strp_stats and aggregate_strp_stats are helper functions to save and aggregate statistics.

98.6 Message assembly limits

The stream parser provide mechanisms to limit the resources consumed by message assembly.

A timer is set when assembly starts for a new message. In receive callback mode the message timeout is taken from rcvtime for the associated TCP socket. In general mode, the timeout is passed as an argument in strp_process. If the timer fires before assembly completes the stream parser is aborted and the ETIMEDOUT error is set on the TCP socket if in receive callback mode.

In receive callback mode, message length is limited to the receive buffer size of the associated TCP socket. If the length returned by parse_msg is greater than the socket buffer size then the stream parser is aborted with EMSGSIZE error set on the TCP socket. Note that this makes the maximum size of receive skbuffs for a socket with a stream parser to be 2*sk_rcvbuf of the TCP socket.

In general mode the message length limit is passed in as an argument to strp_process.

98.7 Author

Tom Herbert (tom@quantonium.net)

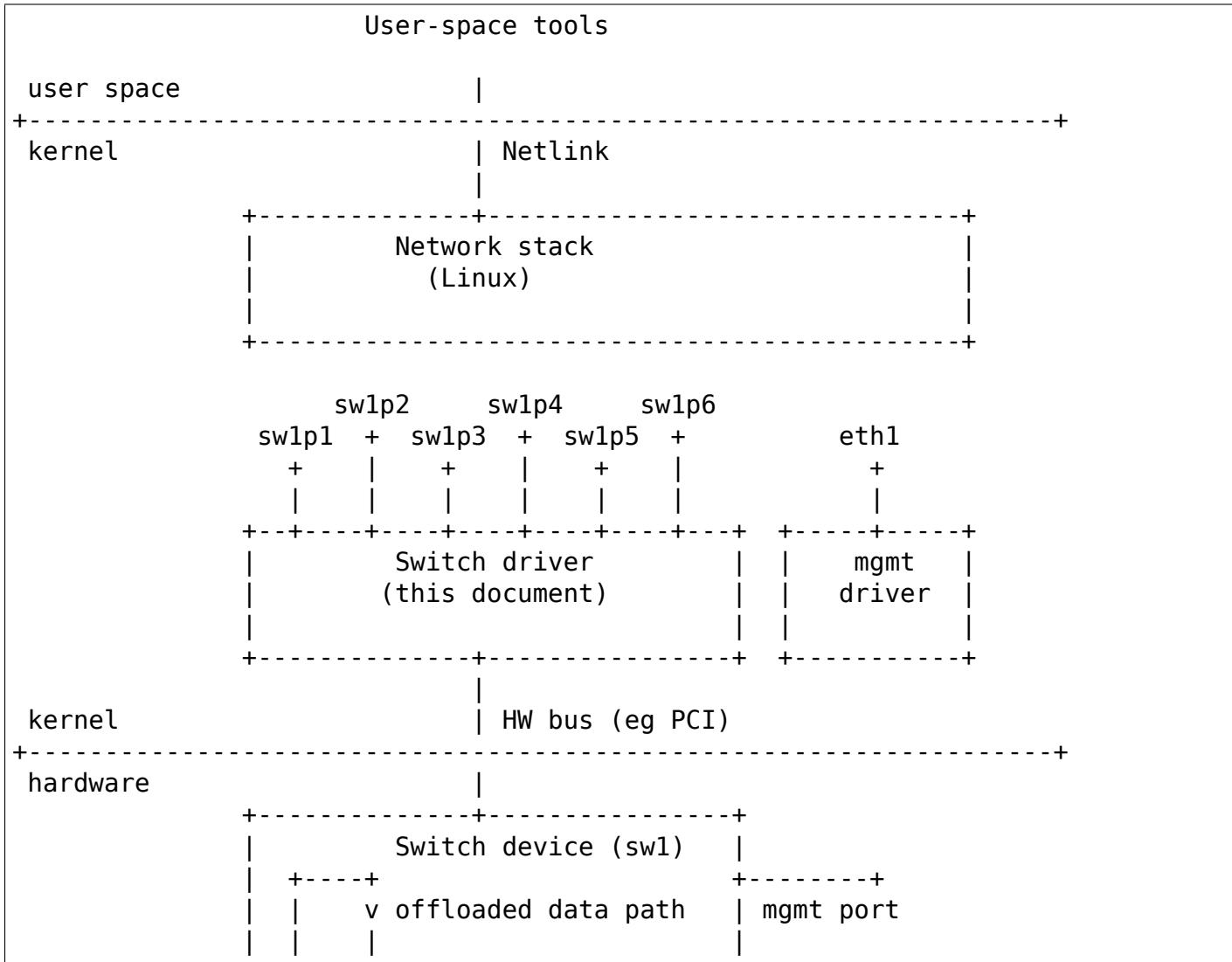
ETHERNET SWITCH DEVICE DRIVER MODEL (SWITCHDEV)

Copyright © 2014 Jiri Pirko <jiri@resnulli.us>

Copyright © 2014-2015 Scott Feldman <sfeldma@gmail.com>

The Ethernet switch device driver model (switchdev) is an in-kernel driver model for switch devices which offload the forwarding (data) plane from the kernel.

Figure 1 is a block diagram showing the components of the switchdev model for an example setup using a data-center-class switch ASIC chip. Other setups with SR-IOV or soft switches, such as OVS, are possible.



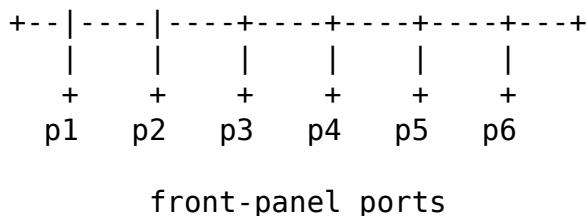


Fig 1.

99.1 Include Files

```
#include <linux/netdevice.h>
#include <net/switchdev.h>
```

99.2 Configuration

Use "depends NET_SWITCHDEV" in driver's Kconfig to ensure switchdev model support is built for driver.

99.3 Switch Ports

On switchdev driver initialization, the driver will allocate and register a *struct net_device* (using *register_netdev()*) for each enumerated physical switch port, called the port netdev. A port netdev is the software representation of the physical port and provides a conduit for control traffic to/from the controller (the kernel) and the network, as well as an anchor point for higher level constructs such as bridges, bonds, VLANs, tunnels, and L3 routers. Using standard netdev tools (iproute2, ethtool, etc), the port netdev can also provide to the user access to the physical properties of the switch port such as PHY link state and I/O statistics.

There is (currently) no higher-level kernel object for the switch beyond the port netdevs. All of the switchdev driver ops are netdev ops or switchdev ops.

A switch management port is outside the scope of the switchdev driver model. Typically, the management port is not participating in offloaded data plane and is loaded with a different driver, such as a NIC driver, on the management port device.

99.3.1 Switch ID

The switchdev driver must implement the net_device operation ndo_get_port_parent_id for each port netdev, returning the same physical ID for each port of a switch. The ID must be unique between switches on the same system. The ID does not need to be unique between switches on different systems.

The switch ID is used to locate ports on a switch and to know if aggregated ports belong to the same switch.

99.3.2 Port Netdev Naming

Udev rules should be used for port netdev naming, using some unique attribute of the port as a key, for example the port MAC address or the port PHYS name. Hard-coding of kernel netdev names within the driver is discouraged; let the kernel pick the default netdev name, and let udev set the final name based on a port attribute.

Using port PHYS name (ndo_get_phys_port_name) for the key is particularly useful for dynamically-named ports where the device names its ports based on external configuration. For example, if a physical 40G port is split logically into 4 10G ports, resulting in 4 port netdevs, the device can give a unique name for each port using port PHYS name. The udev rule would be:

```
SUBSYSTEM=="net", ACTION=="add", ATTR{phys_switch_id}=="<phys_switch_id>", \
ATTR{phys_port_name}!="", NAME="swX$attr{phys_port_name}"
```

Suggested naming convention is "swXpYsZ", where X is the switch name or ID, Y is the port name or ID, and Z is the sub-port name or ID. For example, sw1p1s0 would be sub-port 0 on port 1 on switch 1.

99.3.3 Port Features

NETIF_F_NETNS_LOCAL

If the switchdev driver (and device) only supports offloading of the default network namespace (netns), the driver should set this feature flag to prevent the port netdev from being moved out of the default netns. A netns-aware driver/device would not set this flag and be responsible for partitioning hardware to preserve netns containment. This means hardware cannot forward traffic from a port in one namespace to another port in another namespace.

99.3.4 Port Topology

The port netdevs representing the physical switch ports can be organized into higher-level switching constructs. The default construct is a standalone router port, used to offload L3 forwarding. Two or more ports can be bonded together to form a LAG. Two or more ports (or LAGs) can be bridged to bridge L2 networks. VLANs can be applied to sub-divide L2 networks. L2-over-L3 tunnels can be built on ports. These constructs are built using standard Linux tools such as the bridge driver, the bonding/team drivers, and netlink-based tools such as iproute2.

The switchdev driver can know a particular port's position in the topology by monitoring NETDEV_CHANGEUPPER notifications. For example, a port moved into a bond will see its upper

master change. If that bond is moved into a bridge, the bond's upper master will change. And so on. The driver will track such movements to know what position a port is in in the overall topology by registering for netdevice events and acting on NETDEV_CHANGEUPPER.

99.4 L2 Forwarding Offload

The idea is to offload the L2 data forwarding (switching) path from the kernel to the switchdev device by mirroring bridge FDB entries down to the device. An FDB entry is the {port, MAC, VLAN} tuple forwarding destination.

To offloading L2 bridging, the switchdev driver/device should support:

- Static FDB entries installed on a bridge port
- Notification of learned/forgotten src mac/vlans from device
- STP state changes on the port
- VLAN flooding of multicast/broadcast and unknown unicast packets

99.4.1 Static FDB Entries

A driver which implements the `ndo_fdb_add`, `ndo_fdb_del` and `ndo_fdb_dump` operations is able to support the command below, which adds a static bridge FDB entry:

```
bridge fdb add dev DEV ADDRESS [vlan VID] [self] static
```

(the "static" keyword is non-optional: if not specified, the entry defaults to being "local", which means that it should not be forwarded)

The "self" keyword (optional because it is implicit) has the role of instructing the kernel to fulfill the operation through the `ndo_fdb_add` implementation of the DEV device itself. If DEV is a bridge port, this will bypass the bridge and therefore leave the software database out of sync with the hardware one.

To avoid this, the "master" keyword can be used:

```
bridge fdb add dev DEV ADDRESS [vlan VID] master static
```

The above command instructs the kernel to search for a master interface of DEV and fulfill the operation through the `ndo_fdb_add` method of that. This time, the bridge generates a `SWITCHDEV_FDB_ADD_TO_DEVICE` notification which the port driver can handle and use it to program its hardware table. This way, the software and the hardware database will both contain this static FDB entry.

Note: for new switchdev drivers that offload the Linux bridge, implementing the `ndo_fdb_add` and `ndo_fdb_del` bridge bypass methods is strongly discouraged: all static FDB entries should be added on a bridge port using the "master" flag. The `ndo_fdb_dump` is an exception and can be implemented to visualize the hardware tables, if the device does not have an interrupt for notifying the operating system of newly learned/forgotten dynamic FDB addresses. In that case, the hardware FDB might end up having entries that the software FDB does not, and implementing `ndo_fdb_dump` is the only way to see them.

Note: by default, the bridge does not filter on VLAN and only bridges untagged traffic. To enable VLAN support, turn on VLAN filtering:

```
echo 1 >/sys/class/net/<bridge>/bridge/vlan_filtering
```

99.4.2 Notification of Learned/Forgotten Source MAC/VLANs

The switch device will learn/forget source MAC address/VLAN on ingress packets and notify the switch driver of the mac/vlan/port tuples. The switch driver, in turn, will notify the bridge driver using the switchdev notifier call:

```
err = call_switchdev_notifiers(val, dev, info, extack);
```

Where val is SWITCHDEV_FDB_ADD when learning and SWITCHDEV_FDB_DEL when forgetting, and info points to a struct switchdev_notifier_fdb_info. On SWITCHDEV_FDB_ADD, the bridge driver will install the FDB entry into the bridge's FDB and mark the entry as NTF_EXT_LEARNED. The iproute2 bridge command will label these entries "offload":

```
$ bridge fdb
52:54:00:12:35:01 dev sw1p1 master br0 permanent
00:02:00:00:02:00 dev sw1p1 master br0 offload
00:02:00:00:02:00 dev sw1p1 self
52:54:00:12:35:02 dev sw1p2 master br0 permanent
00:02:00:00:03:00 dev sw1p2 master br0 offload
00:02:00:00:03:00 dev sw1p2 self
33:33:00:00:00:01 dev eth0 self permanent
01:00:5e:00:00:01 dev eth0 self permanent
33:33:ff:00:00:00 dev eth0 self permanent
01:80:c2:00:00:0e dev eth0 self permanent
33:33:00:00:00:01 dev br0 self permanent
01:00:5e:00:00:01 dev br0 self permanent
33:33:ff:12:35:01 dev br0 self permanent
```

Learning on the port should be disabled on the bridge using the bridge command:

```
bridge link set dev DEV learning off
```

Learning on the device port should be enabled, as well as learning_sync:

```
bridge link set dev DEV learning on self
bridge link set dev DEV learning_sync on self
```

Learning_sync attribute enables syncing of the learned/forgotten FDB entry to the bridge's FDB. It's possible, but not optimal, to enable learning on the device port and on the bridge port, and disable learning_sync.

To support learning, the driver implements switchdev op switchdev_port_attr_set for SWITCHDEV_ATTR_PORT_ID_{PRE}_BRIDGE_FLAGS.

99.4.3 FDB Ageing

The bridge will skip ageing FDB entries marked with NTF_EXT_LEARNED and it is the responsibility of the port driver/device to age out these entries. If the port device supports ageing, when the FDB entry expires, it will notify the driver which in turn will notify the bridge with SWITCHDEV_FDB_DEL. If the device does not support ageing, the driver can simulate ageing using a garbage collection timer to monitor FDB entries. Expired entries will be notified to the bridge using SWITCHDEV_FDB_DEL. See rocker driver for example of driver running ageing timer.

To keep an NTF_EXT_LEARNED entry "alive", the driver should refresh the FDB entry by calling call_switchdev_notifiers(SWITCHDEV_FDB_ADD, ...). The notification will reset the FDB entry's last-used time to now. The driver should rate limit refresh notifications, for example, no more than once a second. (The last-used time is visible using the bridge -s fdb option).

99.4.4 STP State Change on Port

Internally or with a third-party STP protocol implementation (e.g. mstpd), the bridge driver maintains the STP state for ports, and will notify the switch driver of STP state change on a port using the switchdev op switchdev_attr_port_set for SWITCHDEV_ATTR_PORT_ID_STP_UPDATE.

State is one of BR_STATE_*. The switch driver can use STP state updates to update ingress packet filter list for the port. For example, if port is DISABLED, no packets should pass, but if port moves to BLOCKED, then STP BPDUs and other IEEE 01:80:c2:xx:xx link-local multicast packets can pass.

Note that STP BDPUs are untagged and STP state applies to all VLANs on the port so packet filters should be applied consistently across untagged and tagged VLANs on the port.

99.4.5 Flooding L2 domain

For a given L2 VLAN domain, the switch device should flood multicast/broadcast and unknown unicast packets to all ports in domain, if allowed by port's current STP state. The switch driver, knowing which ports are within which vlan L2 domain, can program the switch device for flooding. The packet may be sent to the port netdev for processing by the bridge driver. The bridge should not reflood the packet to the same ports the device flooded, otherwise there will be duplicate packets on the wire.

To avoid duplicate packets, the switch driver should mark a packet as already forwarded by setting the skb->offload_fwd_mark bit. The bridge driver will mark the skb using the ingress bridge port's mark and prevent it from being forwarded through any bridge port with the same mark.

It is possible for the switch device to not handle flooding and push the packets up to the bridge driver for flooding. This is not ideal as the number of ports scale in the L2 domain as the device is much more efficient at flooding packets that software.

If supported by the device, flood control can be offloaded to it, preventing certain netdevs from flooding unicast traffic for which there is no FDB entry.

99.4.6 IGMP Snooping

In order to support IGMP snooping, the port netdevs should trap to the bridge driver all IGMP join and leave messages. The bridge multicast module will notify port netdevs on every multicast group changed whether it is static configured or dynamically joined/leave. The hardware implementation should be forwarding all registered multicast traffic groups only to the configured ports.

99.5 L3 Routing Offload

Offloading L3 routing requires that device be programmed with FIB entries from the kernel, with the device doing the FIB lookup and forwarding. The device does a longest prefix match (LPM) on FIB entries matching route prefix and forwards the packet to the matching FIB entry's nexthop(s) egress ports.

To program the device, the driver has to register a FIB notifier handler using `register_fib_notifier`. The following events are available:

<code>FIB_EVENT_ENTRY_ADD</code>	used for both adding a new FIB entry to the device, or modifying an existing entry on the device.
<code>FIB_EVENT_ENTRY_DEL</code>	used for removing a FIB entry
<code>FIB_EVENT_RULE_ADD,</code>	
<code>FIB_EVENT_RULE_DEL</code>	used to propagate FIB rule changes

`FIB_EVENT_ENTRY_ADD` and `FIB_EVENT_ENTRY_DEL` events pass:

```
struct fib_entry_notifier_info {
    struct fib_notifier_info info; /* must be first */
    u32 dst;
    int dst_len;
    struct fib_info *fi;
    u8 tos;
    u8 type;
    u32 tb_id;
    u32 nlflags;
};
```

to add/modify/delete IPv4 dst/dest_len prefix on table tb_id. The `*fi` structure holds details on the route and route's nexthops. `*dev` is one of the port netdevs mentioned in the route's next hop list.

Routes offloaded to the device are labeled with "offload" in the ip route listing:

```
$ ip route show
default via 192.168.0.2 dev eth0
11.0.0.0/30 dev sw1p1 proto kernel scope link src 11.0.0.2 offload
11.0.0.4/30 via 11.0.0.1 dev sw1p1 proto zebra metric 20 offload
11.0.0.8/30 dev sw1p2 proto kernel scope link src 11.0.0.10 offload
11.0.0.12/30 via 11.0.0.9 dev sw1p2 proto zebra metric 20 offload
12.0.0.2 proto zebra metric 30 offload
    nexthop via 11.0.0.1 dev sw1p1 weight 1
```

```
nexthop via 11.0.0.9 dev swlp2 weight 1
12.0.0.3 via 11.0.0.1 dev swlp1 proto zebra metric 20 offload
12.0.0.4 via 11.0.0.9 dev swlp2 proto zebra metric 20 offload
192.168.0.0/24 dev eth0 proto kernel scope link src 192.168.0.15
```

The "offload" flag is set in case at least one device offloads the FIB entry.

XXX: add/mod/del IPv6 FIB API

99.5.1 Nexthop Resolution

The FIB entry's nexthop list contains the nexthop tuple (gateway, dev), but for the switch device to forward the packet with the correct dst mac address, the nexthop gateways must be resolved to the neighbor's mac address. Neighbor mac address discovery comes via the ARP (or ND) process and is available via the arp_tbl neighbor table. To resolve the routes nexthop gateways, the driver should trigger the kernel's neighbor resolution process. See the rocker driver's rocker_port_ipv4_resolve() for an example.

The driver can monitor for updates to arp_tbl using the netevent notifier NETEVENT_NEIGH_UPDATE. The device can be programmed with resolved nexthops for the routes as arp_tbl updates. The driver implements ndo_neigh_destroy to know when arp_tbl neighbor entries are purged from the port.

99.6 Device driver expected behavior

Below is a set of defined behavior that switchdev enabled network devices must adhere to.

99.6.1 Configuration-less state

Upon driver bring up, the network devices must be fully operational, and the backing driver must configure the network device such that it is possible to send and receive traffic to this network device and it is properly separated from other network devices/ports (e.g.: as is frequent with a switch ASIC). How this is achieved is heavily hardware dependent, but a simple solution can be to use per-port VLAN identifiers unless a better mechanism is available (proprietary metadata for each network port for instance).

The network device must be capable of running a full IP protocol stack including multicast, DHCP, IPv4/6, etc. If necessary, it should program the appropriate filters for VLAN, multicast, unicast etc. The underlying device driver must effectively be configured in a similar fashion to what it would do when IGMP snooping is enabled for IP multicast over these switchdev network devices and unsolicited multicast must be filtered as early as possible in the hardware.

When configuring VLANs on top of the network device, all VLANs must be working, irrespective of the state of other network devices (e.g.: other ports being part of a VLAN-aware bridge doing ingress VID checking). See below for details.

If the device implements e.g.: VLAN filtering, putting the interface in promiscuous mode should allow the reception of all VLAN tags (including those not present in the filter(s)).

99.6.2 Bridged switch ports

When a switchdev enabled network device is added as a bridge member, it should not disrupt any functionality of non-bridged network devices and they should continue to behave as normal network devices. Depending on the bridge configuration knobs below, the expected behavior is documented.

99.6.3 Bridge VLAN filtering

The Linux bridge allows the configuration of a VLAN filtering mode (statically, at device creation time, and dynamically, during run time) which must be observed by the underlying switchdev network device/hardware:

- with VLAN filtering turned off: the bridge is strictly VLAN unaware and its data path will process all Ethernet frames as if they are VLAN-untagged. The bridge VLAN database can still be modified, but the modifications should have no effect while VLAN filtering is turned off. Frames ingressing the device with a VID that is not programmed into the bridge/switch's VLAN table must be forwarded and may be processed using a VLAN device (see below).
- with VLAN filtering turned on: the bridge is VLAN-aware and frames ingressing the device with a VID that is not programmed into the bridges/switch's VLAN table must be dropped (strict VID checking).

When there is a VLAN device (e.g: sw0p1.100) configured on top of a switchdev network device which is a bridge port member, the behavior of the software network stack must be preserved, or the configuration must be refused if that is not possible.

- with VLAN filtering turned off, the bridge will process all ingress traffic for the port, except for the traffic tagged with a VLAN ID destined for a VLAN upper. The VLAN upper interface (which consumes the VLAN tag) can even be added to a second bridge, which includes other switch ports or software interfaces. Some approaches to ensure that the forwarding domain for traffic belonging to the VLAN upper interfaces are managed properly:
 - If forwarding destinations can be managed per VLAN, the hardware could be configured to map all traffic, except the packets tagged with a VID belonging to a VLAN upper interface, to an internal VID corresponding to untagged packets. This internal VID spans all ports of the VLAN-unaware bridge. The VID corresponding to the VLAN upper interface spans the physical port of that VLAN interface, as well as the other ports that might be bridged with it.
 - Treat bridge ports with VLAN upper interfaces as standalone, and let forwarding be handled in the software data path.
- with VLAN filtering turned on, these VLAN devices can be created as long as the bridge does not have an existing VLAN entry with the same VID on any bridge port. These VLAN devices cannot be enslaved into the bridge since they duplicate functionality/use case with the bridge's VLAN data path processing.

Non-bridged network ports of the same switch fabric must not be disturbed in any way by the enabling of VLAN filtering on the bridge device(s). If the VLAN filtering setting is global to the entire chip, then the standalone ports should indicate to the network stack that VLAN filtering is required by setting 'rx-vlan-filter: on [fixed]' in the ethtool features.

Because VLAN filtering can be turned on/off at runtime, the switchdev driver must be able to reconfigure the underlying hardware on the fly to honor the toggling of that option and behave appropriately. If that is not possible, the switchdev driver can also refuse to support dynamic toggling of the VLAN filtering knob at runtime and require a destruction of the bridge device(s) and creation of new bridge device(s) with a different VLAN filtering value to ensure VLAN awareness is pushed down to the hardware.

Even when VLAN filtering in the bridge is turned off, the underlying switch hardware and driver may still configure itself in a VLAN-aware mode provided that the behavior described above is observed.

The VLAN protocol of the bridge plays a role in deciding whether a packet is treated as tagged or not: a bridge using the 802.1ad protocol must treat both VLAN-untagged packets, as well as packets tagged with 802.1Q headers, as untagged.

The 802.1p (VID 0) tagged packets must be treated in the same way by the device as untagged packets, since the bridge device does not allow the manipulation of VID 0 in its database.

When the bridge has VLAN filtering enabled and a PVID is not configured on the ingress port, untagged and 802.1p tagged packets must be dropped. When the bridge has VLAN filtering enabled and a PVID exists on the ingress port, untagged and priority-tagged packets must be accepted and forwarded according to the bridge's port membership of the PVID VLAN. When the bridge has VLAN filtering disabled, the presence/lack of a PVID should not influence the packet forwarding decision.

99.6.4 Bridge IGMP snooping

The Linux bridge allows the configuration of IGMP snooping (statically, at interface creation time, or dynamically, during runtime) which must be observed by the underlying switchdev network device/hardware in the following way:

- when IGMP snooping is turned off, multicast traffic must be flooded to all ports within the same bridge that have `mcast_flood=true`. The CPU/management port should ideally not be flooded (unless the ingress interface has `IFF_ALLMULTI` or `IFF_PROMISC`) and continue to learn multicast traffic through the network stack notifications. If the hardware is not capable of doing that then the CPU/management port must also be flooded and multicast filtering happens in software.
- when IGMP snooping is turned on, multicast traffic must selectively flow to the appropriate network ports (including CPU/management port). Flooding of unknown multicast should be only towards the ports connected to a multicast router (the local device may also act as a multicast router).

The switch must adhere to RFC 4541 and flood multicast traffic accordingly since that is what the Linux bridge implementation does.

Because IGMP snooping can be turned on/off at runtime, the switchdev driver must be able to reconfigure the underlying hardware on the fly to honor the toggling of that option and behave appropriately.

A switchdev driver can also refuse to support dynamic toggling of the multicast snooping knob at runtime and require the destruction of the bridge device(s) and creation of a new bridge device(s) with a different multicast snooping value.

SYSFS TAGGING

(Taken almost verbatim from Eric Biederman's netns tagging patch commit msg)

The problem. Network devices show up in sysfs and with the network namespace active multiple devices with the same name can show up in the same directory, ouch!

To avoid that problem and allow existing applications in network namespaces to see the same interface that is currently presented in sysfs, sysfs now has tagging directory support.

By using the network namespace pointers as tags to separate out the sysfs directory entries we ensure that we don't have conflicts in the directories and applications only see a limited set of the network devices.

Each sysfs directory entry may be tagged with a namespace via the `void *ns` member of its `kernfs_node`. If a directory entry is tagged, then `kernfs_node->flags` will have a flag between `KOBJ_NS_TYPE_NONE` and `KOBJ_NS_TYPES`, and `ns` will point to the namespace to which it belongs.

Each sysfs superblock's `kernfs_super_info` contains an array `void *ns[KOBJ_NS_TYPES]`. When a task in a tagging namespace `kobj_ns_type` first mounts sysfs, a new superblock is created. It will be differentiated from other sysfs mounts by having its `s_fs_info->ns[kobj_ns_type]` set to the new namespace. Note that through bind mounting and mounts propagation, a task can easily view the contents of other namespaces' sysfs mounts. Therefore, when a namespace exits, it will call `kobj_ns_exit()` to invalidate any `kernfs_node->ns` pointers pointing to it.

Users of this interface:

- define a type in the `kobj_ns_type` enumeration.
- call `kobj_ns_type_register()` with its `kobj_ns_type_operations` which has
 - `current_ns()` which returns current's namespace
 - `netlink_ns()` which returns a socket's namespace
 - `initial_ns()` which returns the initial namespace
- call `kobj_ns_exit()` when an individual tag is no longer valid

**CHAPTER
ONE**

TC ACTIONS - ENVIRONMENTAL RULES

The "environmental" rules for authors of any new tc actions are:

- 1) If you stealeth or borroweth any packet thou shalt be branching from the righteous path and thou shalt cloneth.
For example if your action queues a packet to be processed later, or intentionally branches by redirecting a packet, then you need to clone the packet.
- 2) If you munge any packet thou shalt call pskb_expand_head in the case someone else is referencing the skb. After that you "own" the skb.
- 3) Dropping packets you don't own is a no-no. You simply return TC_ACT_SHOT to the caller and they will drop it.

The "environmental" rules for callers of actions (qdiscs etc) are:

- 1) Thou art responsible for freeing anything returned as being TC_ACT_SHOT/STOLEN/QUEUED. If none of TC_ACT_SHOT/STOLEN/QUEUED is returned, then all is great and you don't need to do anything.

Post on netdev if something is unclear.

TC QUEUE BASED FILTERING

TC can be used for directing traffic to either a set of queues or to a single queue on both the transmit and receive side.

On the transmit side:

- 1) TC filter directing traffic to a set of queues is achieved using the action `skbedit priority` for Tx priority selection, the priority maps to a traffic class (set of queues) when the queue-sets are configured using `mqprio`.
- 2) TC filter directs traffic to a transmit queue with the action `skbedit queue_mapping $tx_qid`. The action `skbedit queue_mapping` for transmit queue is executed in software only and cannot be offloaded.

Likewise, on the receive side, the two filters for selecting set of queues and/or a single queue are supported as below:

- 1) TC flower filter directs incoming traffic to a set of queues using the '`hw_tc`' option. `hw_tc $TCID` - Specify a hardware traffic class to pass matching packets on to. TCID is in the range 0 through 15.
- 2) TC filter with action `skbedit queue_mapping $rx_qid` selects a receive queue. The action `skbedit queue_mapping` for receive queue is supported only in hardware. Multiple filters may compete in the hardware for queue selection. In such case, the hardware pipeline resolves conflicts based on priority. On Intel E810 devices, TC filter directing traffic to a queue have higher priority over flow director filter assigning a queue. The hash filter has lowest priority.

TCP AUTHENTICATION OPTION LINUX IMPLEMENTATION (RFC5925)

TCP Authentication Option (TCP-AO) provides a TCP extension aimed at verifying segments between trusted peers. It adds a new TCP header option with a Message Authentication Code (MAC). MACs are produced from the content of a TCP segment using a hashing function with a password known to both peers. The intent of TCP-AO is to deprecate TCP-MD5 providing better security, key rotation and support for variety of hashing algorithms.

103.1 1. Introduction

Table 1: Short and Limited Comparison of TCP-AO and TCP-MD5

	TCP-MD5	TCP-AO
Supported hashing algorithms	MD5 (cryptographically weak)	Must support HMAC-SHA1 (chosen-prefix attacks) and CMAC-AES-128 (only side-channel attacks). May support any hashing algorithm.
Length of MACs (bytes)	16	Typically 12-16. Other variants that fit TCP header permitted.
Number of keys per TCP connection	1	Many
Possibility to change an active key	Non-practical (both peers have to change them during MSL)	Supported by protocol
Protection against ICMP 'hard errors'	No	Yes: ignoring them by default on established connections
Protection against traffic-crossing attack	No	Yes: pseudo-header includes TCP ports.
Protection against replayed TCP segments	No	Sequence Number Extension (SNE) and Initial Sequence Numbers (ISNs)
Supports Connectionless Resets	Yes	No. ISNs+SNE are needed to correctly sign RST.
Standards	RFC 2385	RFC 5925, RFC 5926

103.1.1 1.1 Frequently Asked Questions (FAQ) with references to RFC 5925

Q: Can either SendID or RecvID be non-unique for the same 4-tuple (srcaddr, srcport, dstaddr, dstport)?

A: No [3.1]:

>> The IDs of MKTs MUST NOT overlap where their TCP connection identifiers overlap.

Q: Can Master Key Tuple (MKT) for an active connection be removed?

A: No, unless it's copied to Transport Control Block (TCB) [3.1]:

It is presumed that an MKT affecting a particular connection cannot be destroyed during an active connection -- or, equivalently, that its parameters are copied to an area local to the connection (i.e., instantiated) and so changes would affect only new connections.

Q: If an old MKT needs to be deleted, how should it be done in order to not remove it for an active connection? (As it can be still in use at any moment later)

A: Not specified by RFC 5925, seems to be a problem for key management to ensure that no one uses such MKT before trying to remove it.

Q: Can an old MKT exist forever and be used by another peer?

A: It can, it's a key management task to decide when to remove an old key [6.1]:

Deciding when to start using a key is a performance issue. Deciding when to remove an MKT is a security issue. Invalid MKTs are expected to be removed. TCP-AO provides no mechanism to coordinate their removal, as we consider this a key management operation.

also [6.1]:

The only way to avoid reuse of previously used MKTs is to remove the MKT when it is no longer considered permitted.

Linux TCP-AO will try its best to prevent you from removing a key that's being used, considering it a key management failure. But since keeping an outdated key may become a security issue and as a peer may unintentionally prevent the removal of an old key by always setting it as RNextKeyID - a forced key removal mechanism is provided, where userspace has to supply KeyID to use instead of the one that's being removed and the kernel will atomically delete the old key, even if the peer is still requesting it. There are no guarantees for force-delete as the peer may yet not have the new key - the TCP connection may just break. Alternatively, one may choose to shut down the socket.

Q: What happens when a packet is received on a new connection with no known MKT's RecvID?

A: RFC 5925 specifies that by default it is accepted with a warning logged, but the behaviour can be configured by the user [7.5.1.a]:

If the segment is a SYN, then this is the first segment of a new connection. Find the matching MKT for this segment, using the segment's socket pair and its TCP-AO KeyID, matched against the MKT's TCP connection

identifier and the MKT's RecvID.

- i. If there is no matching MKT, remove TCP-AO from the segment.

Proceed with further TCP handling of the segment.

NOTE: this presumes that connections that do not match any MKT should be silently accepted, as noted in Section 7.3.

[7.3]:

>> A TCP-AO implementation MUST allow for configuration of the behavior of segments with TCP-AO but that do not match an MKT. The initial default of this configuration SHOULD be to silently accept such connections. If this is not the desired case, an MKT can be included to match such connections, or the connection can indicate that TCP-AO is required. Alternately, the configuration can be changed to discard segments with the AO option not matching an MKT.

[10.2.b]:

Connections not matching any MKT do not require TCP-AO. Further, incoming segments with TCP-AO are not discarded solely because they include the option, provided they do not match any MKT.

Note that Linux TCP-AO implementation differs in this aspect. Currently, TCP-AO segments with unknown key signatures are discarded with warnings logged.

Q: Does the RFC imply centralized kernel key management in any way? (i.e. that a key on all connections MUST be rotated at the same time?)

A: Not specified. MKTs can be managed in userspace, the only relevant part to key changes is [7.3]:

>> All TCP segments MUST be checked against the set of MKTs for matching TCP connection identifiers.

Q: What happens when RNextKeyID requested by a peer is unknown? Should the connection be reset?

A: It should not, no action needs to be performed [7.5.2.e]:

ii. If they differ, determine whether the RNextKeyID MKT is ready.

1. If the MKT corresponding to the segment's socket pair and RNextKeyID is not available, no action is required (RNextKeyID of a received segment needs to match the MKT's SendID).

Q: How current_key is set and when does it change? It is a user-triggered change, or is it by a request from the remote peer? Is it set by the user explicitly, or by a matching rule?

A: current_key is set by RNextKeyID [6.1]:

Rnext_key is changed only by manual user intervention or MKT management protocol operation. It is not manipulated by TCP-AO. Current_key is updated by TCP-AO when processing received TCP segments as discussed in the segment

processing description in Section 7.5. Note that the algorithm allows the current_key to change to a new MKT, then change back to a previously used MKT (known as "backing up"). This can occur during an MKT change when segments are received out of order, and is considered a feature of TCP-AO, because reordering does not result in drops.

[7.5.2.e.ii]:

2. If the matching MKT corresponding to the segment's socket pair and RNextKeyID is available:

a. Set current_key to the RNextKeyID MKT.

Q: If both peers have multiple MKTs matching the connection's socket pair (with different KeyIDs), how should the sender/receiver pick KeyID to use?

A: Some mechanism should pick the "desired" MKT [3.3]:

Multiple MKTs may match a single outgoing segment, e.g., when MKTs are being changed. Those MKTs cannot have conflicting IDs (as noted elsewhere), and some mechanism must determine which MKT to use for each given outgoing segment.

>> An outgoing TCP segment MUST match at most one desired MKT, indicated by the segment's socket pair. The segment MAY match multiple MKTs, provided that exactly one MKT is indicated as desired. Other information in the segment MAY be used to determine the desired MKT when multiple MKTs match; such information MUST NOT include values in any TCP option fields.

Q: Can TCP-MD5 connection migrate to TCP-AO (and vice-versa):

A: No [1]:

TCP MD5-protected connections cannot be migrated to TCP-AO because TCP MD5 does not support any changes to a connection's security algorithm once established.

Q: If all MKTs are removed on a connection, can it become a non-TCP-AO signed connection?

A: [7.5.2] doesn't have the same choice as SYN packet handling in [7.5.1.i] that would allow accepting segments without a sign (which would be insecure). While switching to non-TCP-AO connection is not prohibited directly, it seems what the RFC means. Also, there's a requirement for TCP-AO connections to always have one current_key [3.3]:

TCP-AO requires that every protected TCP segment match exactly one MKT.

[3.3]:

>> An incoming TCP segment including TCP-AO MUST match exactly one MKT, indicated solely by the segment's socket pair and its TCP-AO KeyID.

[4.4]:

One or more MKTs. These are the MKTs that match this connection's socket pair.

Q: Can a non-TCP-AO connection become a TCP-AO-enabled one?

A: No: for already established non-TCP-AO connection it would be impossible to switch using TCP-AO as the traffic key generation requires the initial sequence numbers. Paraphrasing, starting using TCP-AO would require re-establishing the TCP connection.

103.2 2. In-kernel MKTs database vs database in userspace

Linux TCP-AO support is implemented using `setsockopt()`s, in a similar way to TCP-MD5. It means that a userspace application that wants to use TCP-AO should perform `setsockopt()` on a TCP socket when it wants to add, remove or rotate MKTs. This approach moves the key management responsibility to userspace as well as decisions on corner cases, i.e. what to do if the peer doesn't respect RNextKeyID; moving more code to userspace, especially responsible for the policy decisions. Besides, it's flexible and scales well (with less locking needed than in the case of an in-kernel database). One also should keep in mind that mainly intended users are BGP processes, not any random applications, which means that compared to IPsec tunnels, no transparency is really needed and modern BGP daemons already have `setsockopt()`s for TCP-MD5 support.

Table 2: Considered pros and cons of the approaches

	<code>setsockopt()</code>	in-kernel DB
Extendability	<code>setsockopt()</code> commands should be extendable syscalls	Netlink messages are simple and extendable
Required userspace changes	BGP or any application that wants TCP-AO needs to perform <code>setsockopt()</code> s and do key management	could be transparent as tunnels, providing something like <code>ip tcpao add key (delete/show/rotate)</code>
MKts removal or adding	harder for userspace	harder for kernel
Dump-ability	<code>getsockopt()</code>	Netlink <code>.dump()</code> callback
Limits on kernel resources/memory	equal	
Scalability	contention on TCP_LISTEN sockets	contention on the whole database
Monitoring & warnings	<code>TCP_DIAG</code>	same Netlink socket
Matching of MKTs	half-problem: only listen sockets	hard

103.3 3. uAPI

Linux provides a set of `setsockopt()`s and `getsockopt()`s that let userspace manage TCP-AO on a per-socket basis. In order to add/delete MKTs `TCP_AO_ADD_KEY` and `TCP_AO_DEL_KEY` TCP socket options must be used. It is not allowed to add a key on an established non-TCP-AO connection as well as to remove the last key from TCP-AO connection.

`setsockopt(TCP_AO_DEL_KEY)` command may specify `tcp_ao_del::current_key + tcp_ao_del::set_current` and/or `tcp_ao_del::rnext + tcp_ao_del::set_rnext` which makes such delete "forced": it provides userspace a way to delete a key that's being used and atomically set another one instead. This is not intended for normal use and should be used only when the peer ignores RNextKeyID and keeps requesting/using an old key. It provides a way to force-delete a key that's not trusted but may break the TCP-AO connection.

The usual/normal key-rotation can be performed with `setsockopt(TCP_AO_INFO)`. It also provides a uAPI to change per-socket TCP-AO settings, such as ignoring ICMPs, as well as clear per-socket TCP-AO packet counters. The corresponding `getsockopt(TCP_AO_INFO)` can be used to get those per-socket TCP-AO settings.

Another useful command is `getsockopt(TCP_AO_GET_KEYS)`. One can use it to list all MKTs on a TCP socket or use a filter to get keys for a specific peer and/or sndid/rcvid, VRF L3 interface or get current_key/rnext_key.

To repair TCP-AO connections `setsockopt(TCP_AO_REPAIR)` is available, provided that the user previously has checkpointed/dumped the socket with `getsockopt(TCP_AO_REPAIR)`.

A tip here for scaled TCP_LISTEN sockets, that may have some thousands TCP-AO keys, is: use filters in `getsockopt(TCP_AO_GET_KEYS)` and asynchronous delete with `setsockopt(TCP_AO_DEL_KEY)`.

Linux TCP-AO also provides a bunch of segment counters that can be helpful with troubleshooting/debugging issues. Every MKT has good/bad counters that reflect how many packets passed/failed verification. Each TCP-AO socket has the following counters: - for good segments (properly signed) - for bad segments (failed TCP-AO verification) - for segments with unknown keys - for segments where an AO signature was expected, but wasn't found - for the number of ignored ICMPs

TCP-AO per-socket counters are also duplicated with per-netns counters, exposed with SNMP. Those are `TCPA0Good`, `TCPA0Bad`, `TCPA0KeyNotFound`, `TCPA0Required` and `TCPA0DroppedIcmps`.

RFC 5925 very permissively specifies how TCP port matching can be done for MKTs:

TCP connection identifier. A TCP socket pair, i.e., a local IP address, a remote IP address, a TCP local port, and a TCP remote port. Values can be partially specified using ranges (e.g., 2-30), masks (e.g., 0xF0), wildcards (e.g., "*"), or any other suitable indication.

Currently Linux TCP-AO implementation doesn't provide any TCP port matching. Probably, port ranges are the most flexible for uAPI, but so far not implemented.

103.4 4. setsockopt() vs accept() race

In contrast with TCP-MD5 established connection which has just one key, TCP-AO connections may have many keys, which means that accepted connections on a listen socket may have any amount of keys as well. As copying all those keys on a first properly signed SYN would make the request socket bigger, that would be undesirable. Currently, the implementation doesn't copy keys to request sockets, but rather look them up on the "parent" listener socket.

The result is that when userspace removes TCP-AO keys, that may break not-yet-established connections on request sockets as well as not removing keys from sockets that were already established, but not yet `accept()`'ed, hanging in the `accept` queue.

The reverse is valid as well: if userspace adds a new key for a peer on a listener socket, the established sockets in `accept` queue won't have the new keys.

At this moment, the resolution for the two races: `setsockopt(TCP_AO_ADD_KEY)` vs `accept()` and `setsockopt(TCP_AO_DEL_KEY)` vs `accept()` is delegated to userspace. This means that it's expected that userspace would check the MTKs on the socket that was returned by `accept()` to verify that any key rotation that happened on listen socket is reflected on the newly established connection.

This is a similar "do-nothing" approach to TCP-MD5 from the kernel side and may be changed later by introducing new flags to `tcp_ao_add` and `tcp_ao_del`.

Note that this race is rare for it needs TCP-AO key rotation to happen during the 3-way handshake for the new TCP connection.

103.5 5. Interaction with TCP-MD5

A TCP connection can not migrate between TCP-AO and TCP-MD5 options. The established sockets that have either AO or MD5 keys are restricted for adding keys of the other option.

For listening sockets the picture is different: BGP server may want to receive both TCP-AO and (deprecated) TCP-MD5 clients. As a result, both types of keys may be added to `TCP_CLOSED` or `TCP_LISTEN` sockets. It's not allowed to add different types of keys for the same peer.

103.6 6. SNE Linux implementation

RFC 5925 [6.2] describes the algorithm of how to extend TCP sequence numbers with SNE. In short: TCP has to track the previous sequence numbers and set `sne_flag` when the current SEQ number rolls over. The flag is cleared when both current and previous SEQ numbers cross 0x7fff, which is 32Kb.

In times when `sne_flag` is set, the algorithm compares SEQ for each packet with 0x7fff and if it's higher than 32Kb, it assumes that the packet should be verified with SNE before the increment. As a result, there's this [0; 32Kb] window, when packets with (SNE - 1) can be accepted.

Linux implementation simplifies this a bit: as the network stack already tracks the first SEQ byte that ACK is wanted for (`snd_una`) and the next SEQ byte that is wanted (`rcv_nxt`) - that's enough information for a rough estimation on where in the 4GB SEQ number space both sender and receiver are. When they roll over to zero, the corresponding SNE gets incremented.

`tcp_ao_compute_sne()` is called for each TCP-AO segment. It compares SEQ numbers from the segment with `snd_una` or `rcv_nxt` and fits the result into a 2GB window around them, detecting SEQ numbers rolling over. That simplifies the code a lot and only requires SNE numbers to be stored on every TCP-AO socket.

The 2GB window at first glance seems much more permissive compared to RFC 5926. But that is only used to pick the correct SNE before/after a rollover. It allows more TCP segment replays, but yet all regular TCP checks in `tcp_sequence()` are applied on the verified segment. So, it trades a bit more permissive acceptance of replayed/retransmitted segments for the simplicity of the algorithm and what seems better behaviour for large TCP windows.

103.7 7. Links

RFC 5925 The TCP Authentication Option

<https://www.rfc-editor.org/rfc/pdfrfc/rfc5925.txt.pdf>

RFC 5926 Cryptographic Algorithms for the TCP Authentication Option (TCP-AO)

<https://www.rfc-editor.org/rfc/pdfrfc/rfc5926.txt.pdf>

Draft "SHA-2 Algorithm for the TCP Authentication Option (TCP-AO)"

<https://datatracker.ietf.org/doc/html/draft-nayak-tcp-sha2-03>

RFC 2385 Protection of BGP Sessions via the TCP MD5 Signature Option

<https://www.rfc-editor.org/rfc/pdfrfc/rfc2385.txt.pdf>

Author

Dmitry Safonov <dima@arista.com>

THIN-STREAMS AND TCP

A wide range of Internet-based services that use reliable transport protocols display what we call thin-stream properties. This means that the application sends data with such a low rate that the retransmission mechanisms of the transport protocol are not fully effective. In time-dependent scenarios (like online games, control systems, stock trading etc.) where the user experience depends on the data delivery latency, packet loss can be devastating for the service quality. Extreme latencies are caused by TCP's dependency on the arrival of new data from the application to trigger retransmissions effectively through fast retransmit instead of waiting for long timeouts.

After analysing a large number of time-dependent interactive applications, we have seen that they often produce thin streams and also stay with this traffic pattern throughout its entire lifespan. The combination of time-dependency and the fact that the streams provoke high latencies when using TCP is unfortunate.

In order to reduce application-layer latency when packets are lost, a set of mechanisms has been made, which address these latency issues for thin streams. In short, if the kernel detects a thin stream, the retransmission mechanisms are modified in the following manner:

- 1) If the stream is thin, fast retransmit on the first dupACK.
- 2) If the stream is thin, do not apply exponential backoff.

These enhancements are applied only if the stream is detected as thin. This is accomplished by defining a threshold for the number of packets in flight. If there are less than 4 packets in flight, fast retransmissions can not be triggered, and the stream is prone to experience high retransmission latencies.

Since these mechanisms are targeted at time-dependent applications, they must be specifically activated by the application using the `TCP_THIN_LINEAR_TIMEOUTS` and `TCP_THIN_DUPACK_IOCTL`s or the `tcp_thin_linear_timeouts` and `tcp_thin_dupack sysctl`s. Both modifications are turned off by default.

104.1 References

More information on the modifications, as well as a wide range of experimental data can be found here:

"Improving latency for interactive, thin-stream applications over reliable transport" http://simula.no/research/nd/publications/Simula.nd.477/simula_pdf_file

**CHAPTER
FIVE**

TEAM

Team devices are driven from userspace via libteam library which is here:

<https://github.com/jpirko/libteam>

TIMESTAMPING

106.1 1. Control Interfaces

The interfaces for receiving network packages timestamps are:

SO_TIMESTAMP

Generates a timestamp for each incoming packet in (not necessarily monotonic) system time. Reports the timestamp via `recvmsg()` in a control message in usec resolution. `SO_TIMESTAMP` is defined as `SO_TIMESTAMP_NEW` or `SO_TIMESTAMP_OLD` based on the architecture type and `time_t` representation of libc. Control message format is in `struct __kernel_old_timeval` for `SO_TIMESTAMP_OLD` and in `struct __kernel_sock_timeval` for `SO_TIMESTAMP_NEW` options respectively.

SO_TIMESTAMPNS

Same timestamping mechanism as `SO_TIMESTAMP`, but reports the timestamp as `struct timespec` in nsec resolution. `SO_TIMESTAMPNS` is defined as `SO_TIMESTAMPNS_NEW` or `SO_TIMESTAMPNS_OLD` based on the architecture type and `time_t` representation of libc. Control message format is in `struct timespec` for `SO_TIMESTAMPNS_OLD` and in `struct __kernel_timespec` for `SO_TIMESTAMPNS_NEW` options respectively.

IP_MULTICAST_LOOP + SO_TIMESTAMP[NS]

Only for multicast:approximate transmit timestamp obtained by reading the looped packet receive timestamp.

SO_TIMESTAMPING

Generates timestamps on reception, transmission or both. Supports multiple timestamp sources, including hardware. Supports generating timestamps for stream sockets.

106.1.1 1.1 **SO_TIMESTAMP (also **SO_TIMESTAMP_OLD** and **SO_TIMESTAMP_NEW**)**

This socket option enables timestamping of datagrams on the reception path. Because the destination socket, if any, is not known early in the network stack, the feature has to be enabled for all packets. The same is true for all early receive timestamp options.

For interface details, see *man 7 socket*.

Always use `SO_TIMESTAMP_NEW` timestamp to always get timestamp in `struct __kernel_sock_timeval` format.

`SO_TIMESTAMP_OLD` returns incorrect timestamps after the year 2038 on 32 bit machines.

106.1.2 1.2 SO_TIMESTAMPNS (also SO_TIMESTAMPNS_OLD and SO_TIMESTAMPNS_NEW)

This option is identical to SO_TIMESTAMP except for the returned data type. Its struct timespec allows for higher resolution (ns) timestamps than the timeval of SO_TIMESTAMP (ms).

Always use SO_TIMESTAMPNS_NEW timestamp to always get timestamp in struct __kernel_timespec format.

SO_TIMESTAMPNS_OLD returns incorrect timestamps after the year 2038 on 32 bit machines.

106.1.3 1.3 SO_TIMESTAMPING (also SO_TIMESTAMPING_OLD and SO_TIMESTAMPING_NEW)

Supports multiple types of timestamp requests. As a result, this socket option takes a bitmap of flags, not a boolean. In:

```
err = setsockopt(fd, SOL_SOCKET, SO_TIMESTAMPING, &val, sizeof(val));
```

val is an integer with any of the following bits set. Setting other bit returns EINVAL and does not change the current state.

The socket option configures timestamp generation for individual sk_buffs (1.3.1), timestamp reporting to the socket's error queue (1.3.2) and options (1.3.3). Timestamp generation can also be enabled for individual sendmsg calls using cmsg (1.3.4).

1.3.1 Timestamp Generation

Some bits are requests to the stack to try to generate timestamps. Any combination of them is valid. Changes to these bits apply to newly created packets, not to packets already in the stack. As a result, it is possible to selectively request timestamps for a subset of packets (e.g., for sampling) by embedding an send() call within two setsockopt calls, one to enable timestamp generation and one to disable it. Timestamps may also be generated for reasons other than being requested by a particular socket, such as when receive timestamping is enabled system wide, as explained earlier.

SOF_TIMESTAMPING_RX_HARDWARE:

Request rx timestamps generated by the network adapter.

SOF_TIMESTAMPING_RX_SOFTWARE:

Request rx timestamps when data enters the kernel. These timestamps are generated just after a device driver hands a packet to the kernel receive stack.

SOF_TIMESTAMPING_TX_HARDWARE:

Request tx timestamps generated by the network adapter. This flag can be enabled via both socket options and control messages.

SOF_TIMESTAMPING_TX_SOFTWARE:

Request tx timestamps when data leaves the kernel. These timestamps are generated in the device driver as close as possible, but always prior to, passing the packet to the network interface. Hence, they require driver support and may not be available for all devices. This flag can be enabled via both socket options and control messages.

SOF_TIMESTAMPING_TX_SCHED:

Request tx timestamps prior to entering the packet scheduler. Kernel transmit latency is, if long, often dominated by queuing delay. The difference between this timestamp and one taken at SOF_TIMESTAMPING_TX_SOFTWARE will expose this latency independent of protocol processing. The latency incurred in protocol processing, if any, can be computed by subtracting a userspace timestamp taken immediately before send() from this timestamp. On machines with virtual devices where a transmitted packet travels through multiple devices and, hence, multiple packet schedulers, a timestamp is generated at each layer. This allows for fine grained measurement of queuing delay. This flag can be enabled via both socket options and control messages.

SOF_TIMESTAMPING_TX_ACK:

Request tx timestamps when all data in the send buffer has been acknowledged. This only makes sense for reliable protocols. It is currently only implemented for TCP. For that protocol, it may over-report measurement, because the timestamp is generated when all data up to and including the buffer at send() was acknowledged: the cumulative acknowledgement. The mechanism ignores SACK and FACK. This flag can be enabled via both socket options and control messages.

1.3.2 Timestamp Reporting

The other three bits control which timestamps will be reported in a generated control message. Changes to the bits take immediate effect at the timestamp reporting locations in the stack. Timestamps are only reported for packets that also have the relevant timestamp generation request set.

SOF_TIMESTAMPING_SOFTWARE:

Report any software timestamps when available.

SOF_TIMESTAMPING_SYS_HARDWARE:

This option is deprecated and ignored.

SOF_TIMESTAMPING_RAW_HARDWARE:

Report hardware timestamps as generated by SOF_TIMESTAMPING_TX_HARDWARE when available.

1.3.3 Timestamp Options

The interface supports the options

SOF_TIMESTAMPING_OPT_ID:

Generate a unique identifier along with each packet. A process can have multiple concurrent timestamping requests outstanding. Packets can be reordered in the transmit path, for instance in the packet scheduler. In that case timestamps will be queued onto the error queue out of order from the original send() calls. It is not always possible to uniquely match timestamps to the original send() calls based on timestamp order or payload inspection alone, then.

This option associates each packet at send() with a unique identifier and returns that along with the timestamp. The identifier is derived from a per-socket u32 counter (that wraps). For datagram sockets, the counter increments with each sent packet. For stream sockets, it increments with every byte. For stream sockets, also set SOF_TIMESTAMPING_OPT_ID_TCP, see the section below.

The counter starts at zero. It is initialized the first time that the socket option is enabled. It is reset each time the option is enabled after having been disabled. Resetting the counter does not change the identifiers of existing packets in the system.

This option is implemented only for transmit timestamps. There, the timestamp is always looped along with a struct `sock_extended_err`. The option modifies field `ee_data` to pass an id that is unique among all possibly concurrently outstanding timestamp requests for that socket.

SOF_TIMESTAMPING_OPT_ID_TCP:

Pass this modifier along with `SOF_TIMESTAMPING_OPT_ID` for new TCP timestamping applications. `SOF_TIMESTAMPING_OPT_ID` defines how the counter increments for stream sockets, but its starting point is not entirely trivial. This option fixes that.

For stream sockets, if `SOF_TIMESTAMPING_OPT_ID` is set, this should always be set too. On datagram sockets the option has no effect.

A reasonable expectation is that the counter is reset to zero with the system call, so that a subsequent `write()` of N bytes generates a timestamp with counter N-1. `SOF_TIMESTAMPING_OPT_ID_TCP` implements this behavior under all conditions.

`SOF_TIMESTAMPING_OPT_ID` without modifier often reports the same, especially when the socket option is set when no data is in transmission. If data is being transmitted, it may be off by the length of the output queue (`SIOCOUTQ`).

The difference is due to being based on `snd_una` versus `write_seq`. `snd_una` is the offset in the stream acknowledged by the peer. This depends on factors outside of process control, such as network RTT. `write_seq` is the last byte written by the process. This offset is not affected by external inputs.

The difference is subtle and unlikely to be noticed when configured at initial socket creation, when no data is queued or sent. But `SOF_TIMESTAMPING_OPT_ID_TCP` behavior is more robust regardless of when the socket option is set.

SOF_TIMESTAMPING_OPT_CMSG:

Support `recv()` `cmsg` for all timestamped packets. Control messages are already supported unconditionally on all packets with receive timestamps and on IPv6 packets with transmit timestamp. This option extends them to IPv4 packets with transmit timestamp. One use case is to correlate packets with their egress device, by enabling socket option `IP_PKTINFO` simultaneously.

SOF_TIMESTAMPING_OPT_TSONLY:

Applies to transmit timestamps only. Makes the kernel return the timestamp as a `cmsg` alongside an empty packet, as opposed to alongside the original packet. This reduces the amount of memory charged to the socket's receive budget (`SO_RCVBUF`) and delivers the timestamp even if `sysctl net.core.tstamp_allow_data` is 0. This option disables `SOF_TIMESTAMPING_OPT_CMSG`.

SOF_TIMESTAMPING_OPT_STATS:

Optional stats that are obtained along with the transmit timestamps. It must be used together with `SOF_TIMESTAMPING_OPT_TSONLY`. When the transmit timestamp is available, the stats are available in a separate control message of type `SCM_TIMESTAMPING_OPT_STATS`, as a list of TLVs (struct `nlaattr`) of types. These stats allow the application to associate various transport layer stats with the transmit timestamps, such as how long a certain block of data was limited by peer's receiver window.

SOF_TIMESTAMPING_OPT_PKTINFO:

Enable the SCM_TIMESTAMPING_PKTINFO control message for incoming packets with hardware timestamps. The message contains struct scm_ts_pktinfo, which supplies the index of the real interface which received the packet and its length at layer 2. A valid (non-zero) interface index will be returned only if CONFIG_NET_RX_BUSY_POLL is enabled and the driver is using NAPI. The struct contains also two other fields, but they are reserved and undefined.

SOF_TIMESTAMPING_OPT_TX_SWHW:

Request both hardware and software timestamps for outgoing packets when SOF_TIMESTAMPING_TX_HARDWARE and SOF_TIMESTAMPING_TX_SOFTWARE are enabled at the same time. If both timestamps are generated, two separate messages will be looped to the socket's error queue, each containing just one timestamp.

New applications are encouraged to pass SOF_TIMESTAMPING_OPT_ID to disambiguate timestamps and SOF_TIMESTAMPING_OPT_TSONLY to operate regardless of the setting of sysctl net.core.timestamp_allow_data.

An exception is when a process needs additional cmsg data, for instance SOL_IP/IP_PKTINFO to detect the egress network interface. Then pass option SOF_TIMESTAMPING_OPT_CMSG. This option depends on having access to the contents of the original packet, so cannot be combined with SOF_TIMESTAMPING_OPT_TSONLY.

1.3.4. Enabling timestamps via control messages

In addition to socket options, timestamp generation can be requested per write via cmsg, only for SOF_TIMESTAMPING_TX_*(see Section 1.3.1). Using this feature, applications can sample timestamps per sendmsg() without paying the overhead of enabling and disabling timestamps via setsockopt:

```
struct msghdr *msg;
...
cmsg = CMSG_FIRSTHDR(msg);
cmsg->cmsg_level = SOL_SOCKET;
cmsg->cmsg_type = SO_TIMESTAMPING;
cmsg->cmsg_len = CMSG_LEN(sizeof(__u32));
*((__u32 *) CMSG_DATA(cmsg)) = SOF_TIMESTAMPING_TX_SCHED |
                                SOF_TIMESTAMPING_TX_SOFTWARE |
                                SOF_TIMESTAMPING_TX_ACK;
err = sendmsg(fd, msg, 0);
```

The SOF_TIMESTAMPING_TX_* flags set via cmsg will override the SOF_TIMESTAMPING_TX_* flags set via setsockopt.

Moreover, applications must still enable timestamp reporting via setsockopt to receive timestamps:

```
__u32 val = SOF_TIMESTAMPING_SOFTWARE |
            SOF_TIMESTAMPING_OPT_ID /* or any other flag */;
err = setsockopt(fd, SOL_SOCKET, SO_TIMESTAMPING, &val, sizeof(val));
```

106.1.4 1.4 ByteStream Timestamps

The SO_TIMESTAMPING interface supports timestamping of bytes in a bytestream. Each request is interpreted as a request for when the entire contents of the buffer has passed a timestamping point. That is, for streams option SOF_TIMESTAMPING_TX_SOFTWARE will record when all bytes have reached the device driver, regardless of how many packets the data has been converted into.

In general, bytestreams have no natural delimiters and therefore correlating a timestamp with data is non-trivial. A range of bytes may be split across segments, any segments may be merged (possibly coalescing sections of previously segmented buffers associated with independent send() calls). Segments can be reordered and the same byte range can coexist in multiple segments for protocols that implement retransmissions.

It is essential that all timestamps implement the same semantics, regardless of these possible transformations, as otherwise they are incomparable. Handling "rare" corner cases differently from the simple case (a 1:1 mapping from buffer to skb) is insufficient because performance debugging often needs to focus on such outliers.

In practice, timestamps can be correlated with segments of a bytestream consistently, if both semantics of the timestamp and the timing of measurement are chosen correctly. This challenge is no different from deciding on a strategy for IP fragmentation. There, the definition is that only the first fragment is timestamped. For bytestreams, we chose that a timestamp is generated only when all bytes have passed a point. SOF_TIMESTAMPING_TX_ACK as defined is easy to implement and reason about. An implementation that has to take into account SACK would be more complex due to possible transmission holes and out of order arrival.

On the host, TCP can also break the simple 1:1 mapping from buffer to skbuff as a result of Nagle, cork, autocork, segmentation and GSO. The implementation ensures correctness in all cases by tracking the individual last byte passed to send(), even if it is no longer the last byte after an skbuff extend or merge operation. It stores the relevant sequence number in skb_shinfo(skb)->tskey. Because an skbuff has only one such field, only one timestamp can be generated.

In rare cases, a timestamp request can be missed if two requests are collapsed onto the same skb. A process can detect this situation by enabling SOF_TIMESTAMPING_OPT_ID and comparing the byte offset at send time with the value returned for each timestamp. It can prevent the situation by always flushing the TCP stack in between requests, for instance by enabling TCP_NODELAY and disabling TCP_CORK and autocork. After linux-4.7, a better way to prevent coalescing is to use MSG_EOR flag at sendmsg() time.

These precautions ensure that the timestamp is generated only when all bytes have passed a timestamp point, assuming that the network stack itself does not reorder the segments. The stack indeed tries to avoid reordering. The one exception is under administrator control: it is possible to construct a packet scheduler configuration that delays segments from the same stream differently. Such a setup would be unusual.

106.2 2 Data Interfaces

Timestamps are read using the ancillary data feature of `recvmsg()`. See *man 3 cmsg* for details of this interface. The socket manual page (*man 7 socket*) describes how timestamps generated with `SO_TIMESTAMP` and `SO_TIMESTAMPNS` records can be retrieved.

106.2.1 2.1 SCM_TIMESTAMPING records

These timestamps are returned in a control message with `cmsg_level` `SOL_SOCKET`, `cmsg_type` `SCM_TIMESTAMPING`, and payload of type

For `SO_TIMESTAMPING_OLD`:

```
struct scm_timestamping {
    struct timespec ts[3];
};
```

For `SO_TIMESTAMPING_NEW`:

```
struct scm_timestamping64 {
    struct __kernel_timespec ts[3];
```

Always use `SO_TIMESTAMPING_NEW` timestamp to always get timestamp in `struct scm_timestamping64` format.

`SO_TIMESTAMPING_OLD` returns incorrect timestamps after the year 2038 on 32 bit machines.

The structure can return up to three timestamps. This is a legacy feature. At least one field is non-zero at any time. Most timestamps are passed in `ts[0]`. Hardware timestamps are passed in `ts[2]`.

`ts[1]` used to hold hardware timestamps converted to system time. Instead, expose the hardware clock device on the NIC directly as a HW PTP clock source, to allow time conversion in userspace and optionally synchronize system time with a userspace PTP stack such as `linuxptp`. For the PTP clock API, see `Documentation/driver-api/ptp.rst`.

Note that if the `SO_TIMESTAMP` or `SO_TIMESTAMPNS` option is enabled together with `SO_TIMESTAMPING` using `SOF_TIMESTAMPING_SOFTWARE`, a false software timestamp will be generated in the `recvmsg()` call and passed in `ts[0]` when a real software timestamp is missing. This happens also on hardware transmit timestamps.

2.1.1 Transmit timestamps with MSG_ERRQUEUE

For transmit timestamps the outgoing packet is looped back to the socket's error queue with the send timestamp(s) attached. A process receives the timestamps by calling `recvmsg()` with flag `MSG_ERRQUEUE` set and with a `msg_control` buffer sufficiently large to receive the relevant metadata structures. The `recvmsg` call returns the original outgoing data packet with two ancillary messages attached.

A message of `cm_level` `SOL_IP(V6)` and `cm_type` `IP(V6)_RECVERR` embeds a `struct sock_extended_err`. This defines the error type. For timestamps, the `ee_errno` field is `ENOMSG`. The other ancillary message will have `cm_level` `SOL_SOCKET` and `cm_type` `SCM_TIMESTAMPING`. This embeds the `struct scm_timestamping`.

2.1.1.2 Timestamp types

The semantics of the three struct timespec are defined by field ee_info in the extended error structure. It contains a value of type SCM_TIMESTAMP_* to define the actual timestamp passed in scm_timestamping.

The SCM_TIMESTAMP_* types are 1:1 matches to the SOF_TIMESTAMPING_* control fields discussed previously, with one exception. For legacy reasons, SCM_TIMESTAMP_SND is equal to zero and can be set for both SOF_TIMESTAMPING_TX_HARDWARE and SOF_TIMESTAMPING_TX_SOFTWARE. It is the first if ts[2] is non-zero, the second otherwise, in which case the timestamp is stored in ts[0].

2.1.1.3 Fragmentation

Fragmentation of outgoing datagrams is rare, but is possible, e.g., by explicitly disabling PMTU discovery. If an outgoing packet is fragmented, then only the first fragment is timestamped and returned to the sending socket.

2.1.1.4 Packet Payload

The calling application is often not interested in receiving the whole packet payload that it passed to the stack originally: the socket error queue mechanism is just a method to piggyback the timestamp on. In this case, the application can choose to read datagrams with a smaller buffer, possibly even of length 0. The payload is truncated accordingly. Until the process calls recvmsg() on the error queue, however, the full packet is queued, taking up budget from SO_RCVBUF.

2.1.1.5 Blocking Read

Reading from the error queue is always a non-blocking operation. To block waiting on a timestamp, use poll or select. poll() will return POLLERR in pollfd.revents if any data is ready on the error queue. There is no need to pass this flag in pollfd.events. This flag is ignored on request. See also *man 2 poll*.

2.1.2 Receive timestamps

On reception, there is no reason to read from the socket error queue. The SCM_TIMESTAMPING ancillary data is sent along with the packet data on a normal recvmsg(). Since this is not a socket error, it is not accompanied by a message SOL_IP(V6)/IP(V6)_RECVERROR. In this case, the meaning of the three fields in struct scm_timestamping is implicitly defined. ts[0] holds a software timestamp if set, ts[1] is again deprecated and ts[2] holds a hardware timestamp if set.

106.3 3. Hardware Timestamping configuration: SIOCSHWTSTAMP and SIOCGHWTSTAMP

Hardware time stamping must also be initialized for each device driver that is expected to do hardware time stamping. The parameter is defined in include/uapi/linux/net_tstamp.h as:

```
struct hwtstamp_config {
    int flags;          /* no flags defined right now, must be zero */
    int tx_type;        /* HWTSTAMP_TX_* */
    int rx_filter;      /* HWTSTAMP_FILTER_* */
};
```

Desired behavior is passed into the kernel and to a specific device by calling ioctl(SIOCSHWTSTAMP) with a pointer to a struct ifreq whose ifr_data points to a struct hwtstamp_config. The tx_type and rx_filter are hints to the driver what it is expected to do. If the requested fine-grained filtering for incoming packets is not supported, the driver may time stamp more than just the requested types of packets.

Drivers are free to use a more permissive configuration than the requested configuration. It is expected that drivers should only implement directly the most generic mode that can be supported. For example if the hardware can support HWTSTAMP_FILTER_PTP_V2_EVENT, then it should generally always upscale HWTSTAMP_FILTER_PTP_V2_L2_SYNC, and so forth, as HWTSTAMP_FILTER_PTP_V2_EVENT is more generic (and more useful to applications).

A driver which supports hardware time stamping shall update the struct with the actual, possibly more permissive configuration. If the requested packets cannot be time stamped, then nothing should be changed and ERANGE shall be returned (in contrast to EINVAL, which indicates that SIOCSHWTSTAMP is not supported at all).

Only a processes with admin rights may change the configuration. User space is responsible to ensure that multiple processes don't interfere with each other and that the settings are reset.

Any process can read the actual configuration by passing this structure to ioctl(SIOCGHWTSTAMP) in the same way. However, this has not been implemented in all drivers.

```
/* possible values for hwtstamp_config->tx_type */
enum {
    /*
     * no outgoing packet will need hardware time stamping;
     * should a packet arrive which asks for it, no hardware
     * time stamping will be done
     */
    HWTSTAMP_TX_OFF,

    /*
     * enables hardware time stamping for outgoing packets;
     * the sender of the packet decides which are to be
     * time stamped by setting SOF_TIMESTAMPING_TX_SOFTWARE
     * before sending the packet
     */
    HWTSTAMP_TX_ON,
};
```

```
/* possible values for hwtstamp_config->rx_filter */
enum {
    /* time stamp no incoming packet at all */
    HWTSTAMP_FILTER_NONE,

    /* time stamp any incoming packet */
    HWTSTAMP_FILTER_ALL,

    /* return value: time stamp all packets requested plus some others */
    HWTSTAMP_FILTER_SOME,

    /* PTP v1, UDP, any kind of event packet */
    HWTSTAMP_FILTER_PTP_V1_L4_EVENT,

    /* for the complete list of values, please check
     * the include file include/uapi/linux/net_tstamp.h
     */
};

};
```

106.3.1 3.1 Hardware Timestamping Implementation: Device Drivers

A driver which supports hardware time stamping must support the SIOCSHWTSTAMP ioctl and update the supplied struct hwtstamp_config with the actual values as described in the section on SIOCSHWTSTAMP. It should also support SIOCGHWTSTAMP.

Time stamps for received packets must be stored in the skb. To get a pointer to the shared time stamp structure of the skb call skb_hwtstamps(). Then set the time stamps in the structure:

```
struct skb_shared_hwtstamps {
    /* hardware time stamp transformed into duration
     * since arbitrary point in time
     */
    ktime_t      hwtstamp;
};
```

Time stamps for outgoing packets are to be generated as follows:

- In hard_start_xmit(), check if (skb_shinfo(skb)->tx_flags & SKBTX_HW_TSTAMP) is set no-zero. If yes, then the driver is expected to do hardware time stamping.
- If this is possible for the skb and requested, then declare that the driver is doing the time stamping by setting the flag SKBTX_IN_PROGRESS in skb_shinfo(skb)->tx_flags , e.g. with:

```
skb_shinfo(skb)->tx_flags |= SKBTX_IN_PROGRESS;
```

You might want to keep a pointer to the associated skb for the next step and not free the skb. A driver not supporting hardware time stamping doesn't do that. A driver must never touch sk_buff::tstamp! It is used to store software generated time stamps by the network subsystem.

- Driver should call `skb_tx_timestamp()` as close to passing `sk_buff` to hardware as possible. `skb_tx_timestamp()` provides a software time stamp if requested and hardware timestamping is not possible (SKBTX_IN_PROGRESS not set).
- As soon as the driver has sent the packet and/or obtained a hardware time stamp for it, it passes the time stamp back by calling `skb_tstamp_tx()` with the original skb, the raw hardware time stamp. `skb_tstamp_tx()` clones the original skb and adds the timestamps, therefore the original skb has to be freed now. If obtaining the hardware time stamp somehow fails, then the driver should not fall back to software time stamping. The rationale is that this would occur at a later time in the processing pipeline than other software time stamping and therefore could lead to unexpected deltas between time stamps.

106.3.2 3.2 Special considerations for stacked PTP Hardware Clocks

There are situations when there may be more than one PHC (PTP Hardware Clock) in the data path of a packet. The kernel has no explicit mechanism to allow the user to select which PHC to use for timestamping Ethernet frames. Instead, the assumption is that the outermost PHC is always the most preferable, and that kernel drivers collaborate towards achieving that goal. Currently there are 3 cases of stacked PHCs, detailed below:

3.2.1 DSA (Distributed Switch Architecture) switches

These are Ethernet switches which have one of their ports connected to an (otherwise completely unaware) host Ethernet interface, and perform the role of a port multiplier with optional forwarding acceleration features. Each DSA switch port is visible to the user as a standalone (virtual) network interface, and its network I/O is performed, under the hood, indirectly through the host interface (redirecting to the host port on TX, and intercepting frames on RX).

When a DSA switch is attached to a host port, PTP synchronization has to suffer, since the switch's variable queuing delay introduces a path delay jitter between the host port and its PTP partner. For this reason, some DSA switches include a timestamping clock of their own, and have the ability to perform network timestamping on their own MAC, such that path delays only measure wire and PHY propagation latencies. Timestamping DSA switches are supported in Linux and expose the same ABI as any other network interface (save for the fact that the DSA interfaces are in fact virtual in terms of network I/O, they do have their own PHC). It is typical, but not mandatory, for all interfaces of a DSA switch to share the same PHC.

By design, PTP timestamping with a DSA switch does not need any special handling in the driver for the host port it is attached to. However, when the host port also supports PTP timestamping, DSA will take care of intercepting the `.ndo_eth_ioctl` calls towards the host port, and block attempts to enable hardware timestamping on it. This is because the SO_TIMESTAMPING API does not allow the delivery of multiple hardware timestamps for the same packet, so anybody else except for the DSA switch port must be prevented from doing so.

In the generic layer, DSA provides the following infrastructure for PTP timestamping:

- `.port_txstamp()`: a hook called prior to the transmission of packets with a hardware TX timestamping request from user space. This is required for two-step timestamping, since the hardware timestamp becomes available after the actual MAC transmission, so the driver must be prepared to correlate the timestamp with the original packet so that it can re-enqueue the packet back into the socket's error queue. To save the packet for when the timestamp becomes available, the driver can call `skb_clone_sk`, save the clone pointer

in `skb->cb` and enqueue a tx skb queue. Typically, a switch will have a PTP TX timestamp register (or sometimes a FIFO) where the timestamp becomes available. In case of a FIFO, the hardware might store key-value pairs of PTP sequence ID/message type/domain number and the actual timestamp. To perform the correlation correctly between the packets in a queue waiting for timestamping and the actual timestamps, drivers can use a BPF classifier (`ptp_classify_raw`) to identify the PTP transport type, and `ptp_parse_header` to interpret the PTP header fields. There may be an IRQ that is raised upon this timestamp's availability, or the driver might have to poll after invoking `dev_queue_xmit()` towards the host interface. One-step TX timestamping do not require packet cloning, since there is no follow-up message required by the PTP protocol (because the TX timestamp is embedded into the packet by the MAC), and therefore user space does not expect the packet annotated with the TX timestamp to be re-enqueued into its socket's error queue.

- `.port_rxtstamp()`: On RX, the BPF classifier is run by DSA to identify PTP event messages (any other packets, including PTP general messages, are not timestamped). The original (and only) timestampable skb is provided to the driver, for it to annotate it with a timestamp, if that is immediately available, or defer to later. On reception, timestamps might either be available in-band (through metadata in the DSA header, or attached in other ways to the packet), or out-of-band (through another RX timestamping FIFO). Deferral on RX is typically necessary when retrieving the timestamp needs a sleepable context. In that case, it is the responsibility of the DSA driver to call `netif_rx()` on the freshly timestamped skb.

3.2.2 Ethernet PHYs

These are devices that typically fulfill a Layer 1 role in the network stack, hence they do not have a representation in terms of a network interface as DSA switches do. However, PHYs may be able to detect and timestamp PTP packets, for performance reasons: timestamps taken as close as possible to the wire have the potential to yield a more stable and precise synchronization.

A PHY driver that supports PTP timestamping must create a `struct mii_timestamper` and add a pointer to it in `phydev->mii_ts`. The presence of this pointer will be checked by the networking stack.

Since PHYs do not have network interface representations, the timestamping and ethtool ioctl operations for them need to be mediated by their respective MAC driver. Therefore, as opposed to DSA switches, modifications need to be done to each individual MAC driver for PHY timestamping support. This entails:

- Checking, in `.ndo_eth_ioctl`, whether `phy_has_hwstamp(netdev->phydev)` is true or not. If it is, then the MAC driver should not process this request but instead pass it on to the PHY using `phy_mii_ioctl()`.
- On RX, special intervention may or may not be needed, depending on the function used to deliver skb's up the network stack. In the case of plain `netif_rx()` and similar, MAC drivers must check whether `skb_defer_rx_timestamp(skb)` is necessary or not - and if it is, don't call `netif_rx()` at all. If `CONFIG_NETWORK_PHY_TIMESTAMPING` is enabled, and `skb->dev->phydev->mii_ts` exists, its `.rxtstamp()` hook will be called now, to determine, using logic very similar to DSA, whether deferral for RX timestamping is necessary. Again like DSA, it becomes the responsibility of the PHY driver to send the packet up the stack when the timestamp is available.

For other skb receive functions, such as `napi_gro_receive` and `netif_receive_skb`, the stack automatically checks whether `skb_defer_rx_timestamp()` is necessary, so this

check is not needed inside the driver.

- On TX, again, special intervention might or might not be needed. The function that calls the `mii_ts->txtstamp()` hook is named `skb_clone_tx_timestamp()`. This function can either be called directly (case in which explicit MAC driver support is indeed needed), but the function also piggybacks from the `skb_tx_timestamp()` call, which many MAC drivers already perform for software timestamping purposes. Therefore, if a MAC supports software timestamping, it does not need to do anything further at this stage.

3.2.3 MII bus snooping devices

These perform the same role as timestamping Ethernet PHYs, save for the fact that they are discrete devices and can therefore be used in conjunction with any PHY even if it doesn't support timestamping. In Linux, they are discoverable and attachable to a `struct phy_device` through Device Tree, and for the rest, they use the same `mii_ts` infrastructure as those. See Documentation/devicetree/bindings/ptp/timestamper.txt for more details.

3.2.4 Other caveats for MAC drivers

Stacked PHCs, especially DSA (but not only) - since that doesn't require any modification to MAC drivers, so it is more difficult to ensure correctness of all possible code paths - is that they uncover bugs which were impossible to trigger before the existence of stacked PTP clocks. One example has to do with this line of code, already presented earlier:

```
skb_shinfo(skb)->tx_flags |= SKBTX_IN_PROGRESS;
```

Any TX timestamping logic, be it a plain MAC driver, a DSA switch driver, a PHY driver or a MII bus snooping device driver, should set this flag. But a MAC driver that is unaware of PHC stacking might get tripped up by somebody other than itself setting this flag, and deliver a duplicate timestamp. For example, a typical driver design for TX timestamping might be to split the transmission part into 2 portions:

1. "TX": checks whether PTP timestamping has been previously enabled through the `ndo_eth_ioctl ("priv->hwtstamp_tx_enabled == true")` and the current skb requires a TX timestamp ("`skb_shinfo(skb)->tx_flags & SKBTX_HW_TSTAMP`"). If this is true, it sets the "`skb_shinfo(skb)->tx_flags |= SKBTX_IN_PROGRESS`" flag. Note: as described above, in the case of a stacked PHC system, this condition should never trigger, as this MAC is certainly not the outermost PHC. But this is not where the typical issue is. Transmission proceeds with this packet.
2. "TX confirmation": Transmission has finished. The driver checks whether it is necessary to collect any TX timestamp for it. Here is where the typical issues are: the MAC driver takes a shortcut and only checks whether "`skb_shinfo(skb)->tx_flags & SKBTX_IN_PROGRESS`" was set. With a stacked PHC system, this is incorrect because this MAC driver is not the only entity in the TX data path who could have enabled `SKBTX_IN_PROGRESS` in the first place.

The correct solution for this problem is for MAC drivers to have a compound check in their "TX confirmation" portion, not only for "`skb_shinfo(skb)->tx_flags & SKBTX_IN_PROGRESS`", but also for "`priv->hwtstamp_tx_enabled == true`". Because the rest of the system ensures that PTP timestamping is not enabled for anything other than the outermost PHC, this enhanced check will avoid delivering a duplicated TX timestamp to user space.

LINUX KERNEL TIPC

107.1 Introduction

TIPC (Transparent Inter Process Communication) is a protocol that is specially designed for intra-cluster communication. It can be configured to transmit messages either on UDP or directly across Ethernet. Message delivery is sequence guaranteed, loss free and flow controlled. Latency times are shorter than with any other known protocol, while maximal throughput is comparable to that of TCP.

107.1.1 TIPC Features

- Cluster wide IPC service

Have you ever wished you had the convenience of Unix Domain Sockets even when transmitting data between cluster nodes? Where you yourself determine the addresses you want to bind to and use? Where you don't have to perform DNS lookups and worry about IP addresses? Where you don't have to start timers to monitor the continuous existence of peer sockets? And yet without the downsides of that socket type, such as the risk of lingering inodes?

Welcome to the Transparent Inter Process Communication service, TIPC in short, which gives you all of this, and a lot more.

- Service Addressing

A fundamental concept in TIPC is that of Service Addressing which makes it possible for a programmer to chose his own address, bind it to a server socket and let client programs use only that address for sending messages.

- Service Tracking

A client wanting to wait for the availability of a server, uses the Service Tracking mechanism to subscribe for binding and unbinding/close events for sockets with the associated service address.

The service tracking mechanism can also be used for Cluster Topology Tracking, i.e., subscribing for availability/non-availability of cluster nodes.

Likewise, the service tracking mechanism can be used for Cluster Connectivity Tracking, i.e., subscribing for up/down events for individual links between cluster nodes.

- Transmission Modes

Using a service address, a client can send datagram messages to a server socket.

Using the same address type, it can establish a connection towards an accepting server socket.

It can also use a service address to create and join a Communication Group, which is the TIPC manifestation of a brokerless message bus.

Multicast with very good performance and scalability is available both in datagram mode and in communication group mode.

- Inter Node Links

Communication between any two nodes in a cluster is maintained by one or two Inter Node Links, which both guarantee data traffic integrity and monitor the peer node's availability.

- Cluster Scalability

By applying the Overlapping Ring Monitoring algorithm on the inter node links it is possible to scale TIPC clusters up to 1000 nodes with a maintained neighbor failure discovery time of 1-2 seconds. For smaller clusters this time can be made much shorter.

- Neighbor Discovery

Neighbor Node Discovery in the cluster is done by Ethernet broadcast or UDP multicast, when any of those services are available. If not, configured peer IP addresses can be used.

- Configuration

When running TIPC in single node mode no configuration whatsoever is needed. When running in cluster mode TIPC must as a minimum be given a node address (before Linux 4.17) and told which interface to attach to. The "tipc" configuration tool makes it possible to add and maintain many more configuration parameters.

- Performance

TIPC message transfer latency times are better than in any other known protocol. Maximal byte throughput for inter-node connections is still somewhat lower than for TCP, while they are superior for intra-node and inter-container throughput on the same host.

- Language Support

The TIPC user API has support for C, Python, Perl, Ruby, D and Go.

107.1.2 More Information

- How to set up TIPC:

http://tipc.io/getting_started.html

- How to program with TIPC:

<http://tipc.io/programming.html>

- How to contribute to TIPC:

- <http://tipc.io/contacts.html>

- More details about TIPC specification:

<http://tipc.io/protocol.html>

107.2 Implementation

TIPC is implemented as a kernel module in net/tipc/ directory.

107.2.1 TIPC Base Types

struct tipc_subscription
TIPC network topology subscription object

Definition:

```
struct tipc_subscription {
    struct tipc_subscr s;
    struct tipc_event evt;
    struct kref kref;
    struct net *net;
    struct timer_list timer;
    struct list_head service_list;
    struct list_head sub_list;
    int conid;
    bool inactive;
    spinlock_t lock;
};
```

Members

s host-endian copy of the user subscription
evt template for events generated by subscription
kref reference count for this subscription
net network namespace associated with subscription
timer timer governing subscription duration (optional)
service_list adjacent subscriptions in name sequence's subscription list
sub_list adjacent subscriptions in subscriber's subscription list
conid connection identifier of topology server
inactive true if this subscription is inactive
lock serialize up/down and timer events

struct tipc_media_addr
destination address used by TIPC bearers

Definition:

```
struct tipc_media_addr {
    u8 value[TIPC_MEDIA_INFO_SIZE];
    u8 media_id;
    u8 broadcast;
};
```

Members

value
address info (format defined by media)

media_id
TIPC media type identifier

broadcast
non-zero if address is a broadcast address

struct tipc_media
Media specific info exposed to generic bearer layer

Definition:

```
struct tipc_media {
    int (*send_msg)(struct net *net, struct sk_buff *buf, struct tipc_bearer *b,
→     struct tipc_media_addr *dest);
    int (*enable_media)(struct net *net, struct tipc_bearer *b, struct nlattr *attr[]);
    void (*disable_media)(struct tipc_bearer *b);
    int (*addr2str)(struct tipc_media_addr *addr, char *strbuf, int bufsz);
    int (*addr2msg)(char *msg, struct tipc_media_addr *addr);
    int (*msg2addr)(struct tipc_bearer *b, struct tipc_media_addr *addr, char *msg);
    int (*raw2addr)(struct tipc_bearer *b, struct tipc_media_addr *addr, const char *raw);
    u32 priority;
    u32 tolerance;
    u32 min_win;
    u32 max_win;
    u32 mtu;
    u32 type_id;
    u32 hwaddr_len;
    char name[TIPC_MAX_MEDIA_NAME];
};
```

Members

send_msg
routine which handles buffer transmission

enable_media

routine which enables a media

disable_media

routine which disables a media

addr2str

convert media address format to string

addr2msg

convert from media addr format to discovery msg addr format

msg2addr

convert from discovery msg addr format to media addr format

raw2addr

convert from raw addr format to media addr format

priority

default link (and bearer) priority

tolerance

default time (in ms) before declaring link failure

min_win

minimum window (in packets) before declaring link congestion

max_win

maximum window (in packets) before declaring link congestion

mtu

max packet size bearer can support for media type not dependent on underlying device MTU

type_id

TIPC media identifier

hwaddr_len

TIPC media address len

name

media name

struct tipc_bearer

Generic TIPC bearer structure

Definition:

```
struct tipc_bearer {
    void __rcu *media_ptr;
    u32 mtu;
    struct tipc_media_addr addr;
    char name[TIPC_MAX_BEARER_NAME];
    struct tipc_media *media;
    struct tipc_media_addr bcast_addr;
    struct packet_type pt;
    struct rcu_head rcu;
    u32 priority;
    u32 min_win;
```

```
u32 max_win;
u32 tolerance;
u32 domain;
u32 identity;
struct tipc_discoverer *disc;
char net_plane;
u16 encaps_hlen;
unsigned long up;
refcount_t refcnt;
};
```

Members

media_ptr

pointer to additional media-specific information about bearer

mtu

max packet size bearer can support

addr

media-specific address associated with bearer

name

bearer name (format = media:interface)

media

ptr to media structure associated with bearer

bcast_addr

media address used in broadcasting

pt

packet type for bearer

rcu

rcu struct for tipc_bearer

priority

default link priority for bearer

min_win

minimum window (in packets) before declaring link congestion

max_win

maximum window (in packets) before declaring link congestion

tolerance

default link tolerance for bearer

domain

network domain to which links can be established

identity

array index of this bearer within TIPC bearer array

disc

ptr to link setup request

net_plane

network plane ('A' through 'H') currently associated with bearer

encap_hlen

encap headers length

up

bearer up flag (bit 0)

refcnt

tipc_bearer reference counter

Note

media-specific code is responsible for initialization of the fields indicated below when a bearer is enabled; TIPC's generic bearer code takes care of initializing all other fields.

struct publication

info about a published service address or range

Definition:

```
struct publication {
    struct tipc_service_range sr;
    struct tipc_socket_addr sk;
    u16 scope;
    u32 key;
    u32 id;
    struct list_head binding_node;
    struct list_head binding_sock;
    struct list_head local_publ;
    struct list_head all_publ;
    struct list_head list;
    struct rcu_head rcu;
};
```

Members**sr**

service range represented by this publication

sk

address of socket bound to this publication

scope

scope of publication, TIPC_NODE_SCOPE or TIPC_CLUSTER_SCOPE

key

publication key, unique across the cluster

id

publication id

binding_node

all publications from the same node which bound this one - Remote publications: in node->publ_list; Used by node/name distr to withdraw publications when node is lost - Local/node scope publications: in name_table->node_scope list - Local/cluster scope publications: in name_table->cluster_scope list

binding_sock

all publications from the same socket which bound this one Used by socket to withdraw publications when socket is unbound/released

local_publ

list of identical publications made from this node Used by closest_first and multicast receive lookup algorithms

all_publ

all publications identical to this one, whatever node and scope Used by round-robin lookup algorithm

list

to form a list of publications in temporal order

rcu

RCU callback head used for deferred freeing

struct name_table

table containing all existing port name publications

Definition:

```
struct name_table {
    struct hlist_head services[TIPC_NAMETBL_SIZE];
    struct list_head node_scope;
    struct list_head cluster_scope;
    rwlock_t cluster_scope_lock;
    u32 local_publ_count;
    u32 rc_dests;
    u32 snd_nxt;
};
```

Members

services

name sequence hash lists

node_scope

all local publications with node scope - used by name_distr during re-init of name table

cluster_scope

all local publications with cluster scope - used by name_distr to send bulk updates to new nodes - used by name_distr during re-init of name table

cluster_scope_lock

lock for accessing **cluster_scope**

local_publ_count

number of publications issued by this node

rc_dests

destination node counter

snd_nxt

next sequence number to be used

struct distr_item

publication info distributed to other nodes

Definition:

```
struct distr_item {
    __be32 type;
    __be32 lower;
    __be32 upper;
    __be32 port;
    __be32 key;
};
```

Members**type**

name sequence type

lower

name sequence lower bound

upper

name sequence upper bound

port

publishing port reference

key

publication key

Description

====> All fields are stored in network byte order. <====

First 3 fields identify (name or) name sequence being published. Reference field uniquely identifies port that published name sequence. Key field uniquely identifies publication, in the event a port has multiple publications of the same name sequence.

Note

There is no field that identifies the publishing node because it is the same for all items contained within a publication message.

struct tipc_bc_base

base structure for keeping broadcast send state

Definition:

```
struct tipc_bc_base {
    struct tipc_link *link;
    struct sk_buff_head inputq;
    int dests[MAX_BEARERS];
    int primary_bearer;
    bool bcast_support;
    bool force_bcast;
    bool rcast_support;
    bool force_rcast;
    int rc_ratio;
```

```
    int bc_threshold;  
};
```

Members

link

broadcast send link structure

inputq

data input queue; will only carry SOCK_WAKEUP messages

dests

array keeping number of reachable destinations per bearer

primary_bearer

a bearer having links to all broadcast destinations, if any

bcast_support

indicates if primary bearer, if any, supports broadcast

force_bcast

forces broadcast for multicast traffic

rcast_support

indicates if all peer nodes support replicast

force_rcast

forces replicast for multicast traffic

rc_ratio

dest count as percentage of cluster size where send method changes

bc_threshold

calculated from rc_ratio; if dests > threshold use broadcast

107.2.2 TIPC Bearer Interfaces

```
struct tipc_media *tipc_media_find(const char *name)
```

locates specified media object by name

Parameters

const char *name

name to locate

```
struct tipc_media *media_find_id(u8 type)
```

locates specified media object by type identifier

Parameters

u8 type

type identifier to locate

```
int tipc_media_addr_printf(char *buf, int len, struct tipc_media_addr *a)
```

record media address in print buffer

Parameters

```
char *buf
    output buffer

int len
    output buffer size remaining

struct tipc_media_addr *a
    input media address

int bearer_name_validate(const char *name, struct tipc_bearer_names *name_parts)
    validate & (optionally) deconstruct bearer name
```

Parameters

```
const char *name
    ptr to bearer name string

struct tipc_bearer_names *name_parts
    ptr to area for bearer name components (or NULL if not needed)
```

Return

1 if bearer name is valid, otherwise 0.

```
struct tipc_bearer *tipc_bearer_find(struct net *net, const char *name)
    locates bearer object with matching bearer name
```

Parameters

```
struct net *net
    the applicable net namespace

const char *name
    bearer name to locate

int tipc_enable_bearer(struct net *net, const char *name, u32 disc_domain, u32 prio, struct nla_attr *attr[], struct netlink_ext_ack *extack)
    enable bearer with the given name
```

Parameters

```
struct net *net
    the applicable net namespace

const char *name
    bearer name to enable

u32 disc_domain
    bearer domain

u32 prio
    bearer priority

struct nla_attr *attr[]
    nla_attr array

struct netlink_ext_ack *extack
    netlink extended ack
```

```
int tipc_reset_bearer(struct net *net, struct tipc_bearer *b)
```

Reset all links established over this bearer

Parameters

struct net *net

the applicable net namespace

struct tipc_bearer *b

the target bearer

```
void bearer_disable(struct net *net, struct tipc_bearer *b)
```

disable this bearer

Parameters

struct net *net

the applicable net namespace

struct tipc_bearer *b

the bearer to disable

Note

This routine assumes caller holds RTNL lock.

```
int tipc_l2_send_msg(struct net *net, struct sk_buff *skb, struct tipc_bearer *b, struct tipc_media_addr *dest)
```

send a TIPC packet out over an L2 interface

Parameters

struct net *net

the associated network namespace

struct sk_buff *skb

the packet to be sent

struct tipc_bearer *b

the bearer through which the packet is to be sent

struct tipc_media_addr *dest

peer destination address

```
int tipc_l2_rcv_msg(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt,  
                     struct net_device *orig_dev)
```

handle incoming TIPC message from an interface

Parameters

struct sk_buff *skb

the received message

struct net_device *dev

the net device that the packet was received on

struct packet_type *pt

the packet_type structure which was used to register this handler

struct net_device *orig_dev

the original receive net device in case the device is a bond

Description

Accept only packets explicitly sent to this node, or broadcast packets; ignores packets sent using interface multicast, and traffic sent to other nodes (which can happen if interface is running in promiscuous mode).

```
int tipc_l2_device_event(struct notifier_block *nb, unsigned long evt, void *ptr)
```

handle device events from network device

Parameters

struct notifier_block *nb

the context of the notification

unsigned long evt

the type of event

void *ptr

the net device that the event was on

Description

This function is called by the Ethernet driver in case of link change event.

struct udp_media_addr

IP/UDP addressing information

Definition:

```
struct udp_media_addr {
    __be16 proto;
    __be16 port;
    union {
        struct in_addr ipv4;
        struct in6_addr ipv6;
    };
};
```

Members

proto

Ethernet protocol in use

port

port being used

{unnamed_union}

anonymous

ipv4

IPv4 address of neighbor

ipv6

IPv6 address of neighbor

Description

This is the bearer level originating address used in neighbor discovery messages, and all fields should be in network byte order

struct udp_bearer
ip/udp bearer data structure

Definition:

```
struct udp_bearer {  
    struct tipc_bearer __rcu *bearer;  
    struct socket *ubsock;  
    u32 ifindex;  
    struct work_struct work;  
    struct udp_replicast rcast;  
};
```

Members

bearer
associated generic tipc bearer

ubsock
bearer associated socket

ifindex
local address scope

work
used to schedule deferred work on a bearer

rcast
associated udp_replicast container

int tipc_parse_udp_addr(struct nla *nla, struct udp_media_addr *addr, u32 *scope_id)
build udp media address from netlink data

Parameters

struct nla *nla
netlink attribute containing sockaddr storage aligned address

struct udp_media_addr *addr
tipc media address to fill with address, port and protocol type

u32 *scope_id
IPv6 scope id pointer, not NULL indicates it's required

int tipc_udp_enable(struct net *net, struct tipc_bearer *b, struct nla *attrs[])
callback to create a new udp bearer instance

Parameters

struct net *net
network namespace

struct tipc_bearer *b
pointer to generic tipc_bearer

struct nla *attrs[]
netlink bearer configuration

Description

validate the bearer parameters and initialize the udp bearer rtnl_lock should be held

107.2.3 TIPC Crypto Interfaces

struct tipc_tfm

TIPC TFM structure to form a list of TFMs

Definition:

```
struct tipc_tfm {
    struct crypto_aead *tfm;
    struct list_head list;
};
```

Members

tfm

cipher handle/key

list

linked list of TFMs

struct tipc_aead

TIPC AEAD key structure

Definition:

```
struct tipc_aead {
#define TIPC_AEAD_HINT_LEN (5);
    struct tipc_tfm * __percpu *tfm_entry;
    struct tipc_crypto *crypto;
    struct tipc_aead *cloned;
    atomic_t users;
    u32 salt;
    u8 authsize;
    u8 mode;
    char hint[2 * TIPC_AEAD_HINT_LEN + 1];
    struct rcu_head rCU;
    struct tipc_aead_key *key;
    u16 gen;
    atomic64_t seqno ;
    refcount_t refcnt ;
};
```

Members

tfm_entry

per-cpu pointer to one entry in TFM list

crypto

TIPC crypto owns this key

cloned

reference to the source key in case cloning

users

the number of the key users (TX/RX)

salt

the key's SALT value

authsize

authentication tag size (max = 16)

mode

crypto mode is applied to the key

hint

a hint for user key

rcu

struct rcu_head

key

the aead key

gen

the key's generation

seqno

the key seqno (cluster scope)

refcnt

the key reference counter

struct tipc_crypto_stats

TIPC Crypto statistics

Definition:

```
struct tipc_crypto_stats {  
    unsigned int stat[MAX_STATS];  
};
```

Members**stat**

array of crypto statistics

struct tipc_crypto

TIPC TX/RX crypto structure

Definition:

```
struct tipc_crypto {  
    struct net *net;  
    struct tipc_node *node;  
    struct tipc_aead __rcu *aead[KEY_MAX + 1];  
    atomic_t peer_rx_active;  
    u16 key_gen;  
    struct tipc_key key;
```

```

u8 skey_mode;
struct tipc_aead_key *skey;
struct workqueue_struct *wq;
struct delayed_work work;
#define KEY_DISTR_SCHED          1;
#define KEY_DISTR_COMPL         2;
atomic_t key_distr;
u32 rekeying_intv;
struct tipc_crypto_stats __percpu *stats;
char name[48];
atomic64_t sndnxt ;
unsigned long timer1;
unsigned long timer2;
union {
    struct {
        u8 working:1;
        u8 key_master:1;
        u8 legacy_user:1;
        u8 nokey: 1;
    };
    u8 flags;
};
spinlock_t lock;
};

```

Members

net

struct net

node

TIPC node (RX)

aead

array of pointers to AEAD keys for encryption/decryption

peer_rx_active

replicated peer RX active key index

key_gen

TX/RX key generation

key

the key states

skey_mode

session key's mode

skey

received session key

wq

common workqueue on TX crypto

work

delayed work sched for TX/RX

key_distr

key distributing state

rekeying_intv

rekeying interval (in minutes)

stats

the crypto statistics

name

the crypto name

sndnxt

the per-peer sndnxt (TX)

timer1

general timer 1 (jiffies)

timer2

general timer 2 (jiffies)

{unnamed_union}

anonymous

{unnamed_struct}

anonymous

working

the crypto is working or not

key_master

flag indicates if master key exists

legacy_user

flag indicates if a peer joins w/o master key (for bwd comp.)

nokey

no key indication

flags

combined flags field

lock

tipc_key lock

int **tipc_aead_key_validate**(struct tipc_aead_key *ukey, struct genl_info *info)

Validate a AEAD user key

Parameters

struct tipc_aead_key *ukey

pointer to user key data

struct genl_info *info

netlink info pointer

int **tipc_aead_key_generate**(struct tipc_aead_key *skey)

Generate new session key

Parameters

struct tipc_aead_key *skey
input/output key with new content

Return

0 in case of success, otherwise < 0

void tipc_aead_free(struct rcu_head *rp)
Release AEAD key incl. all the TFM's in the list

Parameters

struct rcu_head *rp
rcu head pointer

struct crypto_aead *tipc_aead_tfm_next(struct tipc_aead *aead)
Move TFM entry to the next one in list and return it

Parameters

struct tipc_aead *aead
the AEAD key pointer

int tipc_aead_init(struct tipc_aead **aead, struct tipc_aead_key *ukey, u8 mode)
Initiate TIPC AEAD

Parameters

struct tipc_aead **aead
returned new TIPC AEAD key handle pointer

struct tipc_aead_key *ukey
pointer to user key data

u8 mode
the key mode

Description

Allocate a (list of) new cipher transformation (TFM) with the specific user key data if valid. The number of the allocated TFM's can be set via the sysfs "net/tipc/max_tfms" first. Also, all the other AEAD data are also initialized.

Return

0 if the initiation is successful, otherwise: < 0

int tipc_aead_clone(struct tipc_aead **dst, struct tipc_aead *src)
Clone a TIPC AEAD key

Parameters

struct tipc_aead **dst
dest key for the cloning

struct tipc_aead *src
source key to clone from

Description

Make a "copy" of the source AEAD key data to the dest, the TFM's list is common for the keys. A reference to the source is held in the "cloned" pointer for the later freeing purposes.

Note

this must be done in cluster-key mode only!

Return

0 in case of success, otherwise < 0

```
void *tipc_aead_mem_alloc(struct crypto_aead *tfm, unsigned int crypto_ctx_size, u8 **iv,
                           struct aead_request **req, struct scatterlist **sg, int nsg)
```

Allocate memory for AEAD request operations

Parameters

struct crypto_aead *tfm

cipher handle to be registered with the request

unsigned int crypto_ctx_size

size of crypto context for callback

u8 **iv

returned pointer to IV data

struct aead_request **req

returned pointer to AEAD request data

struct scatterlist **sg

returned pointer to SG lists

int nsg

number of SG lists to be allocated

Description

Allocate memory to store the crypto context data, AEAD request, IV and SG lists, the memory layout is as follows: crypto_ctx || iv || aead_req || sg[]

Return

the pointer to the memory areas in case of success, otherwise NULL

```
int tipc_aead_encrypt(struct tipc_aead *aead, struct sk_buff *skb, struct tipc_bearer *b,
                      struct tipc_media_addr *dst, struct tipc_node * __dnode)
```

Encrypt a message

Parameters

struct tipc_aead *aead

TIPC AEAD key for the message encryption

struct sk_buff *skb

the input/output skb

struct tipc_bearer *b

TIPC bearer where the message will be delivered after the encryption

struct tipc_media_addr *dst

the destination media address

struct tipc_node * __dnode

TIPC dest node if "known"

Return

- 0 : if the encryption has completed
- -EINPROGRESS/-EBUSY : if a callback will be performed
- < 0 : the encryption has failed

int tipc_aead_decrypt(struct net *net, struct tipc_aead *aead, struct sk_buff *skb, struct tipc_bearer *b)

Decrypt an encrypted message

Parameters

struct net *net

struct net

struct tipc_aead *aead

TIPC AEAD for the message decryption

struct sk_buff *skb

the input/output skb

struct tipc_bearer *b

TIPC bearer where the message has been received

Return

- 0 : if the decryption has completed
- -EINPROGRESS/-EBUSY : if a callback will be performed
- < 0 : the decryption has failed

bool tipc_ehdr_validate(struct sk_buff *skb)

Validate an encryption message

Parameters

struct sk_buff *skb

the message buffer

Return

"true" if this is a valid encryption message, otherwise "false"

int tipc_ehdr_build(struct net *net, struct tipc_aead *aead, u8 tx_key, struct sk_buff *skb, struct tipc_crypto *_rx)

Build TIPC encryption message header

Parameters

struct net *net

struct net

struct tipc_aead *aead

TX AEAD key to be used for the message encryption

u8 tx_key

key id used for the message encryption

struct sk_buff *skb

input/output message skb

struct tipc_crypto * rx
RX crypto handle if dest is "known"

Return

the header size if the building is successful, otherwise < 0

int tipc_crypto_key_init(struct tipc_crypto *c, struct tipc_aead_key *ukey, u8 mode, bool master_key)

Initiate a new user / AEAD key

Parameters

struct tipc_crypto *c
TIPC crypto to which new key is attached

struct tipc_aead_key *ukey
the user key

u8 mode
the key mode (CLUSTER_KEY or PER_NODE_KEY)

bool master_key
specify this is a cluster master key

Description

A new TIPC AEAD key will be allocated and initiated with the specified user key, then attached to the TIPC crypto.

Return

new key id in case of success, otherwise: < 0

int tipc_crypto_key_attach(struct tipc_crypto *c, struct tipc_aead *aead, u8 pos, bool master_key)

Attach a new AEAD key to TIPC crypto

Parameters

struct tipc_crypto *c
TIPC crypto to which the new AEAD key is attached

struct tipc_aead *aead
the new AEAD key pointer

u8 pos
desired slot in the crypto key array, = 0 if any!

bool master_key
specify this is a cluster master key

Return

new key id in case of success, otherwise: -EBUSY

bool tipc_crypto_key_try_align(struct tipc_crypto *rx, u8 new_pending)
Align RX keys if possible

Parameters

```
struct tipc_crypto *rx
    RX crypto handle

u8 new_pending
    new pending slot if aligned (= TX key from peer)
```

Description

Peer has used an unknown key slot, this only happens when peer has left and rejoined, or we are newcomer. That means, there must be no active key but a pending key at unaligned slot. If so, we try to move the pending key to the new slot.

Note

A potential passive key can exist, it will be shifted correspondingly!

Return

"true" if key is successfully aligned, otherwise "false"

```
struct tipc_aead *tipc_crypto_key_pick_tx(struct tipc_crypto *tx, struct tipc_crypto *rx,
                                         struct sk_buff *skb, u8 tx_key)
```

Pick one TX key for message decryption

Parameters

```
struct tipc_crypto *tx
    TX crypto handle

struct tipc_crypto *rx
    RX crypto handle (can be NULL)

struct sk_buff *skb
    the message skb which will be decrypted later

u8 tx_key
    peer TX key id
```

Description

This function looks up the existing TX keys and pick one which is suitable for the message decryption, that must be a cluster key and not used before on the same message (i.e. recursive).

Return

the TX AEAD key handle in case of success, otherwise NULL

```
void tipc_crypto_key_synth(struct tipc_crypto *rx, struct sk_buff *skb)
    Synch own key data according to peer key status
```

Parameters

```
struct tipc_crypto *rx
    RX crypto handle

struct sk_buff *skb
    TIPCV2 message buffer (incl. the ehdr from peer)
```

Description

This function updates the peer node related data as the peer RX active key has changed, so the number of TX keys' users on this node are increased and decreased correspondingly.

It also considers if peer has no key, then we need to make own master key (if any) taking over i.e. starting grace period and also trigger key distributing process.

The "per-peer" sndnxt is also reset when the peer key has switched.

```
int tipc_crypto_xmit(struct net *net, struct sk_buff **skb, struct tipc_bearer *b, struct tipc_media_addr *dst, struct tipc_node * __dnode)
```

Build & encrypt TIPC message for xmit

Parameters

struct net *net
struct net

struct sk_buff **skb
input/output message skb pointer

struct tipc_bearer *b
bearer used for xmit later

struct tipc_media_addr *dst
destination media address

struct tipc_node * __dnode
destination node for reference if any

Description

First, build an encryption message header on the top of the message, then encrypt the original TIPC message by using the pending, master or active key with this preference order. If the encryption is successful, the encrypted skb is returned directly or via the callback. Otherwise, the skb is freed!

Return

- 0 : the encryption has succeeded (or no encryption)
- -EINPROGRESS/-EBUSY : the encryption is ongoing, a callback will be made
- **-ENOKEK** : the encryption has failed due to no key
- **-EKEYREVOKED** : the encryption has failed due to key revoked
- **-ENOMEM** : the encryption has failed due to no memory
- < 0 : the encryption has failed due to other reasons

```
int tipc_crypto_rcv(struct net *net, struct tipc_crypto *rx, struct sk_buff **skb, struct tipc_bearer *b)
```

Decrypt an encrypted TIPC message from peer

Parameters

struct net *net
struct net

struct tipc_crypto *rx
RX crypto handle

struct sk_buff **skb
input/output message skb pointer

```
struct tipc_bearer *b
    bearer where the message has been received
```

Description

If the decryption is successful, the decrypted skb is returned directly or as the callback, the encryption header and auth tag will be trimed out before forwarding to [tipc_rcv\(\)](#) via the tipc_crypto_rcv_complete(). Otherwise, the skb will be freed!

Note

RX key(s) can be re-aligned, or in case of no key suitable, TX cluster key(s) can be taken for decryption (- recursive).

Return

- 0 : the decryption has successfully completed
- -EINPROGRESS/-EBUSY : the decryption is ongoing, a callback will be made
- **-ENOKEY** : the decryption has failed due to no key
- **-EBADMSG** : the decryption has failed due to bad message
- **-ENOMEM** : the decryption has failed due to no memory
- < 0 : the decryption has failed due to other reasons

```
void tipc_crypto_msg_rcv(struct net *net, struct sk_buff *skb)
    Common 'MSG_CRYPTO' processing point
```

Parameters

```
struct net *net
    the struct net
```

```
struct sk_buff *skb
    the receiving message buffer
```

```
int tipc_crypto_key_distr(struct tipc_crypto *tx, u8 key, struct tipc_node *dest)
    Distribute a TX key
```

Parameters

```
struct tipc_crypto *tx
    the TX crypto
```

```
u8 key
    the key's index
```

```
struct tipc_node *dest
    the destination tipc node, = NULL if distributing to all nodes
```

Return

0 in case of success, otherwise < 0

```
int tipc_crypto_key_xmit(struct net *net, struct tipc_aead_key *skey, u16 gen, u8 mode, u32
                           dnnode)
```

Send a session key

Parameters

struct net *net
the struct net

struct tipc_aead_key *skey
the session key to be sent

u16 gen
the key's generation

u8 mode
the key's mode

u32 dnode
the destination node address, = 0 if broadcasting to all nodes

Description

The session key 'skey' is packed in a TIPC v2 'MSG_CRYPTO/KEY_DISTR_MSG' as its data section, then xmit-ed through the uc/bc link.

Return

0 in case of success, otherwise < 0

bool tipc_crypto_key_rcv(struct *tipc_crypto* *rx, struct *tipc_msg* *hdr)
Receive a session key

Parameters

struct tipc_crypto *rx
the RX crypto

struct tipc_msg *hdr
the TIPC v2 message incl. the receiving session key in its data

Description

This function retrieves the session key in the message from peer, then schedules a RX work to attach the key to the corresponding RX crypto.

Return

"true" if the key has been scheduled for attaching, otherwise "false".

void tipc_crypto_work_rx(struct work_struct *work)
Scheduled RX works handler

Parameters

struct work_struct *work
the struct RX work

Description

The function processes the previous scheduled works i.e. distributing TX key or attaching a received session key on RX crypto.

void tipc_crypto_rekeying_sched(struct *tipc_crypto* *tx, bool changed, u32 new_intv)
(Re)schedule rekeying w/o new interval

Parameters

```

struct tipc_crypto *tx
    TX crypto

bool changed
    if the rekeying needs to be rescheduled with new interval

u32 new_intv
    new rekeying interval (when "changed" = true)

void tipc_crypto_work_tx(struct work_struct *work)
    Scheduled TX works handler

```

Parameters

```

struct work_struct *work
    the struct TX work

```

Description

The function processes the previous scheduled work, i.e. key rekeying, by generating a new session key based on current one, then attaching it to the TX crypto and finally distributing it to peers. It also re-schedules the rekeying if needed.

107.2.4 TIPC Discoverer Interfaces

```

struct tipc_discoverer
    information about an ongoing link setup request

```

Definition:

```

struct tipc_discoverer {
    u32 bearer_id;
    struct tipc_media_addr dest;
    struct net *net;
    u32 domain;
    int num_nodes;
    spinlock_t lock;
    struct sk_buff *skb;
    struct timer_list timer;
    unsigned long timer_intv;
};

```

Members

bearer_id
identity of bearer issuing requests

dest
destination address for request messages

net
network namespace instance

domain
network domain to which links can be established

num_nodes

number of nodes currently discovered (i.e. with an active link)

lock

spinlock for controlling access to requests

skb

request message to be (repeatedly) sent

timer

timer governing period between requests

timer_intv

current interval between requests (in ms)

```
void tipc_disc_init_msg(struct net *net, struct sk_buff *skb, u32 mtyp, struct tipc_bearer *b)
```

initialize a link setup message

Parameters

struct net *net

the applicable net namespace

struct sk_buff *skb

buffer containing message

u32 mtyp

message type (request or response)

struct tipc_bearer *b

ptr to bearer issuing message

```
void disc_dupl_alert(struct tipc_bearer *b, u32 node_addr, struct tipc_media_addr *media_addr)
```

issue node address duplication alert

Parameters

struct tipc_bearer *b

pointer to bearer detecting duplication

u32 node_addr

duplicated node address

struct tipc_media_addr *media_addr

media address advertised by duplicated node

```
void tipc_disc_rcv(struct net *net, struct sk_buff *skb, struct tipc_bearer *b)
```

handle incoming discovery message (request or response)

Parameters

struct net *net

applicable net namespace

struct sk_buff *skb

buffer containing message

struct tipc_bearer *b

bearer that message arrived on

```
int tipc_disc_create(struct net *net, struct tipc_bearer *b, struct tipc_media_addr *dest,
                     struct sk_buff **skb)
```

create object to send periodic link setup requests

Parameters

struct net *net

the applicable net namespace

struct tipc_bearer *b

ptr to bearer issuing requests

struct tipc_media_addr *dest

destination address for request messages

struct sk_buff **skb

pointer to created frame

Return

0 if successful, otherwise -errno.

```
void tipc_disc_delete(struct tipc_disc discoverer *d)
```

destroy object sending periodic link setup requests

Parameters

struct tipc_disc discoverer *d

ptr to link dest structure

```
void tipc_disc_reset(struct net *net, struct tipc_bearer *b)
```

reset object to send periodic link setup requests

Parameters

struct net *net

the applicable net namespace

struct tipc_bearer *b

ptr to bearer issuing requests

107.2.5 TIPC Link Interfaces

struct tipc_link

TIPC link data structure

Definition:

```
struct tipc_link {
    u32 addr;
    char name[TIPC_MAX_LINK_NAME];
    struct net *net;
    u16 peer_session;
    u16 session;
    u16 snd_nxt_state;
    u16 rcv_nxt_state;
    u32 peer_bearer_id;
```

```
u32 bearer_id;
u32 tolerance;
u32 abort_limit;
u32 state;
u16 peer_caps;
bool in_session;
bool active;
u32 silent_intv_cnt;
char if_name[TIPC_MAX_IF_NAME];
u32 priority;
char net_plane;
struct tipc_mon_state mon_state;
u16 rst_cnt;
u16 drop_point;
struct sk_buff *failover_reasm_skb;
struct sk_buff_head failover_deferdq;
u16 mtu;
u16 advertised_mtu;
struct sk_buff_head transmq;
struct sk_buff_head backlogq;
struct {
    u16 len;
    u16 limit;
    struct sk_buff *target_bskb;
} backlog[5];
u16 snd_nxt;
u16 rcv_nxt;
u32 rcv_unacked;
struct sk_buff_head deferdq;
struct sk_buff_head *inputq;
struct sk_buff_head *namedq;
struct sk_buff_head wakeupq;
u16 window;
u16 min_win;
u16 ssthresh;
u16 max_win;
u16 cong_acks;
u16 checkpoint;
struct sk_buff *reasm_buf;
struct sk_buff *reasm_tnlmsg;
u16 ackers;
u16 acked;
u16 last_gap;
struct tipc_gap_ack_blk *last_ga;
struct tipc_link *bc_rcvlink;
struct tipc_link *bc_sndlink;
u8 nack_state;
bool bc_peer_is_up;
struct tipc_stats stats;
};
```

Members**addr**

network address of link's peer node

name

link name character string

net

pointer to namespace struct

peer_session

link session # being used by peer end of link

session

session to be used by link

snd_nxt_state

next send seq number

rcv_nxt_state

next rcv seq number

peer_bearer_id

bearer id used by link's peer endpoint

bearer_id

local bearer id used by link

tolerance

minimum link continuity loss needed to reset link [in ms]

abort_limit

of unacknowledged continuity probes needed to reset link

state

current state of link FSM

peer_caps

bitmap describing capabilities of peer node

in_session

have received ACTIVATE_MSG from peer

active

link is active

silent_intv_cnt

of timer intervals without any reception from peer

if_name

associated interface name

priority

current link priority

net_plane

current link network plane ('A' through 'H')

mon_state

cookie with information needed by link monitor

rst_cnt
link reset counter

drop_point
seq number for failover handling (FIXME)

failover_reasm_skb
saved failover msg ptr (FIXME)

failover_deferdq
deferred message queue for failover processing (FIXME)

mtu
current maximum packet size for this link

advertised_mtu
advertised own mtu when link is being established

transmq
the link's transmit queue

backlogq
queue for messages waiting to be sent

backlog
link's backlog by priority (importance)

snd_nxt
next sequence number to be used

rcv_nxt
next sequence number to expect for inbound messages

rcv_unacked
messages read by user, but not yet acked back to peer

deferdq
deferred receive queue

inputq
buffer queue for messages to be delivered upwards

namedq
buffer queue for name table messages to be delivered upwards

wakeupq
linked list of wakeup msgs waiting for link congestion to abate

window
sliding window size for congestion handling

min_win
minimal send window to be used by link

ssthresh
slow start threshold for congestion handling

max_win
maximal send window to be used by link

cong_acks

congestion acks for congestion avoidance (FIXME)

checkpoint

seq number for congestion window size handling

reasm_buf

head of partially reassembled inbound message fragments

reasm_tnlmsg

fragmentation/reassembly area for tunnel protocol message

ackers

of peers that needs to ack each packet before it can be released

acked

last packet acked by a certain peer. Used for broadcast.

last_gap

last gap ack blocks for bcast (FIXME)

last_ga

ptr to gap ack blocks

bc_rcvlink

the peer specific link used for broadcast reception

bc_sndlink

the namespace global link used for broadcast sending

nack_state

bcast nack state

bc_peer_is_up

peer has acked the bcast init msg

stats

collects statistics regarding link activity

```
bool tipc_link_create(struct net *net, char *if_name, int bearer_id, int tolerance, char
                      net_plane, u32 mtu, int priority, u32 min_win, u32 max_win, u32
                      session, u32 self, u32 peer, u8 *peer_id, u16 peer_caps, struct
                      tipc_link *bc_sndlink, struct tipc_link *bc_rcvlink, struct sk_buff_head
                      *inputq, struct sk_buff_head *namedq, struct tipc_link **link)
```

create a new link

Parameters**struct net *net**

pointer to associated network namespace

char *if_name

associated interface name

int bearer_id

id (index) of associated bearer

int tolerance

link tolerance to be used by link

```
char net_plane
    network plane (A,B,c..) this link belongs to

u32 mtu
    mtu to be advertised by link

int priority
    priority to be used by link

u32 min_win
    minimal send window to be used by link

u32 max_win
    maximal send window to be used by link

u32 session
    session to be used by link

u32 self
    local unicast link id

u32 peer
    node id of peer node

u8 *peer_id
    128-bit ID of peer

u16 peer_caps
    bitmap describing peer node capabilities

struct tipc_link *bc_sndlink
    the namespace global link used for broadcast sending

struct tipc_link *bc_rcvlink
    the peer specific link used for broadcast reception

struct sk_buff_head *inputq
    queue to put messages ready for delivery

struct sk_buff_head *namedq
    queue to put binding table update messages ready for delivery

struct tipc_link **link
    return value, pointer to put the created link
```

Return

true if link was created, otherwise false

```
bool tipc_link_bc_create(struct net *net, u32 ownnode, u32 peer, u8 *peer_id, int mtu, u32
                           min_win, u32 max_win, u16 peer_caps, struct sk_buff_head
                           *inputq, struct sk_buff_head *namedq, struct tipc_link
                           *bc_sndlink, struct tipc_link **link)
```

create new link to be used for broadcast

Parameters

```
struct net *net
    pointer to associated network namespace
```

u32 ownnode
identity of own node

u32 peer
node id of peer node

u8 *peer_id
128-bit ID of peer

int mtu
mtu to be used initially if no peers

u32 min_win
minimal send window to be used by link

u32 max_win
maximal send window to be used by link

u16 peer_caps
bitmap describing peer node capabilities

struct sk_buff_head *inputq
queue to put messages ready for delivery

struct sk_buff_head *namedq
queue to put binding table update messages ready for delivery

struct tipc_link *bc_sndlink
the namespace global link used for broadcast sending

struct tipc_link **link
return value, pointer to put the created link

Return

true if link was created, otherwise false

int tipc_link_fsm_evt(struct tipc_link *l, int evt)
link finite state machine

Parameters

struct tipc_link *l
pointer to link

int evt
state machine event to be processed

bool tipc_link_too_silent(struct tipc_link *l)
check if link is "too silent"

Parameters

struct tipc_link *l
tipc link to be checked

Return

true if the link 'silent_intv_cnt' is about to reach the 'abort_limit' value, otherwise false

```
int link_schedule_user(struct tipc_link *l, struct tipc_msg *hdr)
    schedule a message sender for wakeup after congestion
```

Parameters

struct tipc_link *l
congested link

struct tipc_msg *hdr
header of message that is being sent Create pseudo msg to send back to user when con-gestion abates

```
void link_prepare_wakeup(struct tipc_link *l)
    prepare users for wakeup after congestion
```

Parameters

struct tipc_link *l
congested link Wake up a number of waiting users, as permitted by available space in the send queue

```
void tipc_link_set_skb_retransmit_time(struct sk_buff *skb, struct tipc_link *l)
    set the time at which retransmission of the given skb should be next attempted
```

Parameters

struct sk_buff *skb
skb to set a future retransmission time for

struct tipc_link *l
link the skb will be transmitted on

```
int tipc_link_xmit(struct tipc_link *l, struct sk_buff_head *list, struct sk_buff_head *xmitq)
    enqueue buffer list according to queue situation
```

Parameters

struct tipc_link *l
link to use

struct sk_buff_head *list
chain of buffers containing message

struct sk_buff_head *xmitq
returned list of packets to be sent by caller

Description

Consumes the buffer chain. Messages at TIPC_SYSTEM_IMPORTANCE are always accepted

Return

0 if success, or errno: -ELINKCONG, -EMSGSIZE or -ENOBUFS

```
bool link_retransmit_failure(struct tipc_link *l, struct tipc_link *r, int *rc)
    Detect repeated retransmit failures
```

Parameters

struct tipc_link *l
tipc link sender

struct tipc_link *r
tipc link receiver (= 1 in case of unicast)

int *rc
returned code

Return

true if the repeated retransmit failures happens, otherwise false

u16 tipc_get_gap_ack_blocks(struct tipc_gap_ack_blocks **ga, struct tipc_link *l, struct tipc_msg *hdr, bool uc)

get Gap ACK blocks from PROTOCOL/STATE_MSG

Parameters

struct tipc_gap_ack_blocks **ga
returned pointer to the Gap ACK blocks if any

struct tipc_link *l
the tipc link

struct tipc_msg *hdr
the PROTOCOL/STATE_MSG header

bool uc
desired Gap ACK blocks type, i.e. unicast (= 1) or broadcast (= 0)

Return

the total Gap ACK blocks size

void tipc_link_failover_prepare(struct tipc_link *l, struct tipc_link *tnl, struct sk_buff_head *xmitq)

prepare tnl for link failover

Parameters

struct tipc_link *l
failover link

struct tipc_link *tnl
tunnel link

struct sk_buff_head *xmitq
queue for messages to be xmitted

Description

This is a special version of the precursor - tipc_link_tnl_prepare(), see the [tipc_node_link_failover\(\)](#) for details

void tipc_link_reset_stats(struct tipc_link *l)
reset link statistics

Parameters

struct tipc_link *l
pointer to link

```
int tipc_link_dump(struct tipc_link *l, u16 dqueues, char *buf)
    dump TIPC link data
```

Parameters

struct tipc_link *l
tipc link to be dumped

u16 dqueues

bitmask to decide if any link queue to be dumped? - TIPC_DUMP_NONE: don't dump link queues - TIPC_DUMP_TRANSMQ: dump link transmq queue - TIPC_DUMP_BACKLOGQ: dump link backlog queue - TIPC_DUMP_DEFERDQ: dump link deferd queue - TIPC_DUMP_INPUTQ: dump link input queue - TIPC_DUMP_WAKEUP: dump link wakeup queue - TIPC_DUMP_ALL: dump all the link queues above

char *buf

returned buffer of dump data in format

107.2.6 TIPC msg Interfaces

```
struct sk_buff *tipc_buf_acquire(u32 size, gfp_t gfp)
    creates a TIPC message buffer
```

Parameters

u32 size
message size (including TIPC header)

gfp_t gfp
memory allocation flags

Return

a new buffer with data pointers set to the specified size.

NOTE

Headroom is reserved to allow prepending of a data link header. There may also be unrequested tailroom present at the buffer's end.

```
int tipc_msg_append(struct tipc_msg *_hdr, struct msghdr *m, int dlen, int mss, struct
                     sk_buff_head *txq)
```

Append data to tail of an existing buffer queue

Parameters

struct tipc_msg *_hdr
header to be used

struct msghdr *m
the data to be appended

int dlen
size of data to be appended

int mss
max allowable size of buffer

struct sk_buff_head *txq
queue to append to

Return

the number of 1k blocks appended or errno value

int tipc_msg_fragment(struct sk_buff *skb, const struct tipc_msg *hdr, int pktmax, struct sk_buff_head *frags)

build a fragment skb list for TIPC message

Parameters

struct sk_buff *skb
TIPC message skb

const struct tipc_msg *hdr
internal msg header to be put on the top of the fragments

int pktmax
max size of a fragment incl. the header

struct sk_buff_head *frags
returned fragment skb list

Return

0 if the fragmentation is successful, otherwise: -EINVAL or -ENOMEM

int tipc_msg_build(struct tipc_msg *mhdr, struct msghdr *m, int offset, int dsz, int pktmax, struct sk_buff_head *list)

create buffer chain containing specified header and data

Parameters

struct tipc_msg *mhdr
Message header, to be prepended to data

struct msghdr *m
User message

int offset
buffer offset for fragmented messages (FIXME)

int dsz
Total length of user data

int pktmax
Max packet size that can be used

struct sk_buff_head *list
Buffer or chain of buffers to be returned to caller

Description

Note that the recursive call we are making here is safe, since it can logically go only one further level down.

Return

message data size or errno: -ENOMEM, -EFAULT

bool tipc_msg_bundle(struct sk_buff *bskb, struct tipc_msg *msg, u32 max)

Append contents of a buffer to tail of an existing one

Parameters

struct sk_buff *bskb

the bundle buffer to append to

struct tipc_msg *msg

message to be appended

u32 max

max allowable size for the bundle buffer

Return

"true" if bundling has been performed, otherwise "false"

bool tipc_msg_try_bundle(struct sk_buff *tskb, struct sk_buff **skb, u32 mss, u32 dnode, bool *new_bundle)

Try to bundle a new message to the last one

Parameters

struct sk_buff *tskb

the last/target message to which the new one will be appended

struct sk_buff **skb

the new message skb pointer

u32 mss

max message size (header inclusive)

u32 dnode

destination node for the message

bool *new_bundle

if this call made a new bundle or not

Return

"true" if the new message skb is potential for bundling this time or later, in the case a bundling has been done this time, the skb is consumed (the skb pointer = NULL). Otherwise, "false" if the skb cannot be bundled at all.

bool tipc_msg_extract(struct sk_buff *skb, struct sk_buff **iskb, int *pos)

extract bundled inner packet from buffer

Parameters

struct sk_buff *skb

buffer to be extracted from.

struct sk_buff **iskb

extracted inner buffer, to be returned

int *pos

position in outer message of msg to be extracted. Returns position of next msg. Consumes outer buffer when last packet extracted

Return

true when there is an extracted buffer, otherwise false

```
bool tipc_msg_reverse(u32 own_node, struct sk_buff **skb, int err)
    swap source and destination addresses and add error code
```

Parameters

u32 own_node

originating node id for reversed message

struct sk_buff **skb

buffer containing message to be reversed; will be consumed

int err

error code to be set in message, if any Replaces consumed buffer with new one when successful

Return

true if success, otherwise false

```
bool tipc_msg_lookup_dest(struct net *net, struct sk_buff *skb, int *err)
    try to find new destination for named message
```

Parameters

struct net *net

pointer to associated network namespace

struct sk_buff *skb

the buffer containing the message.

int *err

error code to be used by caller if lookup fails Does not consume buffer

Return

true if a destination is found, false otherwise

107.2.7 TIPC Name Interfaces

struct service_range

container for all bindings of a service range

Definition:

```
struct service_range {
    u32 lower;
    u32 upper;
    struct rb_node tree_node;
    u32 max;
    struct list_head local_publ;
    struct list_head all_publ;
};
```

Members

lower

service range lower bound

upper

service range upper bound

tree_node

member of service range RB tree

max

largest 'upper' in this node subtree

local_publ

list of identical publications made from this node Used by closest_first lookup and multicast lookup algorithm

all_publ

all publications identical to this one, whatever node and scope Used by round-robin lookup algorithm

struct tipc_service

container for all published instances of a service type

Definition:

```
struct tipc_service {  
    u32 type;  
    u32 publ_cnt;  
    struct rb_root ranges;  
    struct hlist_node service_list;  
    struct list_head subscriptions;  
    spinlock_t lock;  
    struct rcu_head rcu;  
};
```

Members**type**

32 bit 'type' value for service

publ_cnt

increasing counter for publications in this service

ranges

rb tree containing all service ranges for this service

service_list

links to adjacent name ranges in hash chain

subscriptions

list of subscriptions for this service type

lock

spinlock controlling access to pertaining service ranges/publications

rcu

RCU callback head used for deferred freeing

service_range_foreach_match

```
service_range_foreach_match (sr, sc, start, end)
```

iterate over tipc service rbtree for each range match

Parameters**sr**

the service range pointer as a loop cursor

sc

the pointer to tipc service which holds the service range rbtree

start

beginning of the search range (end >= start) for matching

end

end of the search range (end >= start) for matching

```
struct service_range *service_range_match_first(struct rb_node *n, u32 start, u32 end)
```

find first service range matching a range

Parameters**struct rb_node *n**

the root node of service range rbtree for searching

u32 start

beginning of the search range (end >= start) for matching

u32 end

end of the search range (end >= start) for matching

Return

the leftmost service range node in the rbtree that overlaps the specific range if any. Otherwise, returns NULL.

```
struct service_range *service_range_match_next(struct rb_node *n, u32 start, u32 end)
```

find next service range matching a range

Parameters**struct rb_node *n**

a node in service range rbtree from which the searching starts

u32 start

beginning of the search range (end >= start) for matching

u32 end

end of the search range (end >= start) for matching

Return

the next service range node to the given node in the rbtree that overlaps the specific range if any. Otherwise, returns NULL.

```
struct publication *tipc_publ_create(struct tipc_uaddr *ua, struct tipc_socket_addr *sk, u32 key)
```

create a publication structure

Parameters

struct tipc_uaddr *ua

the service range the user is binding to

struct tipc_socket_addr *sk

the address of the socket that is bound

u32 key

publication key

struct *tipc_service* ***tipc_service_create**(struct *net* *net, struct tipc_uaddr *ua)

create a service structure for the specified 'type'

Parameters

struct net *net

network namespace

struct tipc_uaddr *ua

address representing the service to be bound

Description

Allocates a single range structure and sets it to all 0's.

struct *publication* ***tipc_service_remove_publ**(struct *service_range* *r, struct
tipc_socket_addr *sk, u32 key)

remove a publication from a service

Parameters

struct service_range *r

service_range to remove publication from

struct tipc_socket_addr *sk

address publishing socket

u32 key

target publication key

void **tipc_service_subscribe**(struct *tipc_service* *service, struct *tipc_subscription* *sub)

attach a subscription, and optionally issue the prescribed number of events if there is any
service range overlapping with the requested range

Parameters

struct tipc_service *service

the tipc_service to attach the **sub** to

struct tipc_subscription *sub

the subscription to attach

bool **tipc_nametbl_lookup_anycast**(struct *net* *net, struct tipc_uaddr *ua, struct
tipc_socket_addr *sk)

perform service instance to socket translation

Parameters

struct net *net

network namespace

```
struct tipc_uaddr *ua
    service address to look up

struct tipc_socket_addr *sk
    address to socket we want to find
```

Description

On entry, a non-zero 'sk->node' indicates the node where we want lookup to be performed, which may not be this one.

On exit:

- If lookup is deferred to another node, leave 'sk->node' unchanged and return 'true'.
- If lookup is successful, set the 'sk->node' and 'sk->ref' (== portid) which represent the bound socket and return 'true'.
- If lookup fails, return 'false'

Note that for legacy users (node configured with Z.C.N address format) the 'closest-first' lookup algorithm must be maintained, i.e., if sk.node is 0 we must look in the local binding list first

```
void tipc_nametbl_withdraw(struct net *net, struct tipc_uaddr *ua, struct tipc_socket_addr
                           *sk, u32 key)
```

withdraw a service binding

Parameters

```
struct net *net
    network namespace
```

```
struct tipc_uaddr *ua
    service address/range being unbound
```

```
struct tipc_socket_addr *sk
    address of the socket being unbound from
```

```
u32 key
    target publication key
```

```
bool tipc_nametbl_subscribe(struct tipc_subscription *sub)
    add a subscription object to the name table
```

Parameters

```
struct tipc_subscription *sub
    subscription to add
```

```
void tipc_nametbl_unsubscribe(struct tipc_subscription *sub)
    remove a subscription object from name table
```

Parameters

```
struct tipc_subscription *sub
    subscription to remove
```

```
void tipc_service_delete(struct net *net, struct tipc_service *sc)
    purge all publications for a service and delete it
```

Parameters

```
struct net *net
    the associated network namespace

struct tipc_service *sc
    tipc_service to delete

void publ_to_item(struct distr_item *i, struct publication *p)
    add publication info to a publication message
```

Parameters

```
struct distr_item *i
    location of item in the message

struct publication *p
    publication info

struct sk_buff *named_prepare_buf(struct net *net, u32 type, u32 size, u32 dest)
    allocate & initialize a publication message
```

Parameters

```
struct net *net
    the associated network namespace

u32 type
    message type

u32 size
    payload size

u32 dest
    destination node
```

Description

The buffer returned is of size INT_H_SIZE + payload size

```
struct sk_buff *tipc_named_publish(struct net *net, struct publication *p)
    tell other nodes about a new publication by this node
```

Parameters

```
struct net *net
    the associated network namespace

struct publication *p
    the new publication
```

```
struct sk_buff *tipc_named_withdraw(struct net *net, struct publication *p)
    tell other nodes about a withdrawn publication by this node
```

Parameters

```
struct net *net
    the associated network namespace

struct publication *p
    the withdrawn publication
```

```
void named_distribute(struct net *net, struct sk_buff_head *list, u32 dnode, struct list_head *pls, u16 seqno)
```

prepare name info for bulk distribution to another node

Parameters

struct net *net

the associated network namespace

struct sk_buff_head *list

list of messages (buffers) to be returned from this function

u32 dnode

node to be updated

struct list_head *pls

linked list of publication items to be packed into buffer chain

u16 seqno

sequence number for this message

```
void tipc_named_node_up(struct net *net, u32 dnode, u16 capabilities)
```

tell specified node about all publications by this node

Parameters

struct net *net

the associated network namespace

u32 dnode

destination node

u16 capabilities

peer node's capabilities

```
void tipc_publ_purge(struct net *net, struct publication *p, u32 addr)
```

remove publication associated with a failed node

Parameters

struct net *net

the associated network namespace

struct publication *p

the publication to remove

u32 addr

failed node's address

Description

Invoked for each publication issued by a newly failed node. Removes publication structure from name table & deletes it.

```
bool tipc_update_namtbl(struct net *net, struct distr_item *i, u32 node, u32 dtype)
```

try to process a nametable update and notify subscribers

Parameters

struct net *net

the associated network namespace

struct distr_item *i
location of item in the message

u32 node
node address

u32 dtype
name distributor message type

Description

tipc_nametbl_lock must be held.

Return

the publication item if successful, otherwise NULL.

void tipc_named_rcv(struct net *net, struct sk_buff_head *namedq, u16 *rcv_nxt, bool *open)
process name table update messages sent by another node

Parameters

struct net *net
the associated network namespace

struct sk_buff_head *namedq
queue to receive from

u16 *rcv_nxt
store last received seqno here

bool *open
last bulk msg was received (FIXME)

void tipc_named_reinit(struct net *net)
re-initialize local publications

Parameters

struct net *net
the associated network namespace

Description

This routine is called whenever TIPC networking is enabled. All name table entries published by this node are updated to reflect the node's new network address.

107.2.8 TIPC Node Management Interfaces

struct tipc_node
TIPC node structure

Definition:

```
struct tipc_node {  
    u32 addr;  
    struct kref kref;  
    rwlock_t lock;  
    struct net *net;
```

```

struct hlist_node hash;
int active_links[2];
struct tipc_link_entry links[MAX_BEARERS];
struct tipc_bmlink_entry bc_entry;
int action_flags;
struct list_head list;
int state;
bool preliminary;
bool failover_sent;
u16 sync_point;
int link_cnt;
u16 working_links;
u16 capabilities;
u32 signature;
u32 link_id;
u8 peer_id[16];
char peer_id_string[NODE_ID_STR_LEN];
struct list_head publ_list;
struct list_head conn_sks;
unsigned long keepalive_intv;
struct timer_list timer;
struct rcu_head rcu;
unsigned long delete_at;
struct net *peer_net;
u32 peer_hash_mix;
#ifndef CONFIG_TIPC_CRYPTO
    struct tipc_crypto *crypto_rx;
#endif;
};

```

Members

addr

network address of node

kref

reference counter to node object

lock

rwlock governing access to structure

net

the applicable net namespace

hash

links to adjacent nodes in unsorted hash chain

active_links

bearer ids of active links, used as index into links[] array

links

array containing references to all links to node

bc_entry

broadcast link entry

action_flags

bit mask of different types of node actions

list

links to adjacent nodes in sorted list of cluster's nodes

state

connectivity state vs peer node

preliminary

a preliminary node or not

failover_sent

failover sent or not

sync_point

sequence number where synch/failover is finished

link_cnt

number of links to node

working_links

number of working links to node (both active and standby)

capabilities

bitmap, indicating peer node's functional capabilities

signature

node instance identifier

link_id

local and remote bearer ids of changing link, if any

peer_id

128-bit ID of peer

peer_id_string

ID string of peer

publ_list

list of publications

conn_sks

list of connections (FIXME)

keepalive_intv

keepalive interval in milliseconds

timer

node's keepalive timer

rcu

rcu struct for tipc_node

delete_at

indicates the time for deleting a down node

peer_net

peer's net namespace

peer hash mix

hash for this peer (FIXME)

crypto rx

RX crypto handler

```
struct tipc_crypto *tipc_node_crypto_rx(struct tipc_node *_n)
```

Retrieve crypto RX handle from node

Parameters

struct tipc_node * __n
target tipc node

Note

node ref counter must be held first!

```
void __tipc_node_link_up(struct tipc_node *n, int bearer_id, struct sk_buff_head *xmitq)
    handle addition of link
```

Parameters

struct tipc_node *
target tipc node

int bearer_id
id of the bearer

```
struct sk_buff_head *xmitq
```

queue for messages to be xmited on Node lock must be held by caller Link becomes active (alone or shared) or standby, depending on its priority.

```
void tipc_node_link_up(struct tipc_node *n, int bearer_id, struct sk_buff_head *xmitq)
    handle addition of link
```

Parameters

```
struct tipc_node *n  
    target tipc node
```

int bearer_id
id of the bearer

```
struct sk_buff_head *xmitq
```

queue for messages to be xmited on

Description

Link becomes active (alone or shared) or standby, depending on its priority

```
void tipc_node_link_failover(struct tipc_node *n, struct tipc_link *l, struct tipc_link *tnl,  
                           struct sk_buff_head *xmitq)
```

start failover in case "half-failover"

Parameters

struct tipc_node *
tipc node structure

```
struct tipc_link *l
    link peer endpoint failingover (- can be NULL)

struct tipc_link *tnl
    tunnel link

struct sk_buff_head *xmitq
    queue for messages to be xmited on tnl link later
```

Description

This function is only called in a very special situation where link failover can be already started on peer node but not on this node. This can happen when e.g.:

1. Both links <1A-2A>, <1B-2B> down
2. Link endpoint 2A up, but 1A still down (e.g. due to network disturbance, wrong session, etc.)
3. Link <1B-2B> up
4. Link endpoint 2A down (e.g. due to link tolerance timeout)
5. Node 2 starts failover onto link <1B-2B>

==> Node 1 does never start link/node failover!

```
void __tipc_node_link_down(struct tipc_node *n, int *bearer_id, struct sk_buff_head *xmitq,
                           struct tipc_media_addr **maddr)
```

handle loss of link

Parameters

```
struct tipc_node *n
    target tipc_node

int *bearer_id
    id of the bearer

struct sk_buff_head *xmitq
    queue for messages to be xmited on

struct tipc_media_addr **maddr
    output media address of the bearer

int tipc_node_get_linkname(struct net *net, u32 bearer_id, u32 addr, char *linkname, size_t len)
```

get the name of a link

Parameters

```
struct net *net
    the applicable net namespace

u32 bearer_id
    id of the bearer

u32 addr
    peer node address

char *linkname
    link name output buffer
```

size_t len
 size of **linkname** output buffer

Return

0 on success

int tipc_node_xmit(struct net *net, struct sk_buff_head *list, u32 dnode, int selector)
 general link level function for message sending

Parameters

struct net *net
 the applicable net namespace

struct sk_buff_head *list
 chain of buffers containing message

u32 dnode
 address of destination node

int selector
 a number used for deterministic link selection Consumes the buffer chain.

Return

0 if success, otherwise: -ELINKCONG,-EHOSTUNREACH,-EMSGSIZE,-ENOBUF

void tipc_node_bc_rcv(struct net *net, struct sk_buff *skb, int bearer_id)
 process TIPC broadcast packet arriving from off-node

Parameters

struct net *net
 the applicable net namespace

struct sk_buff *skb
 TIPC packet

int bearer_id
 id of bearer message arrived on

Description

Invoked with no locks held.

bool tipc_node_check_state(struct tipc_node *n, struct sk_buff *skb, int bearer_id, struct sk_buff_head *xmitq)

check and if necessary update node state

Parameters

struct tipc_node *n
 target tipc_node

struct sk_buff *skb
 TIPC packet

int bearer_id
 identity of bearer delivering the packet

struct sk_buff_head *xmitq
queue for messages to be xmitted on

Return

true if state and msg are ok, otherwise false

void tipc_rcv(struct net *net, struct sk_buff *skb, struct tipc_bearer *b)
process TIPC packets/messages arriving from off-node

Parameters

struct net *net
the applicable net namespace

struct sk_buff *skb
TIPC packet

struct tipc_bearer *b
pointer to bearer message arrived on

Description

Invoked with no locks held. Bearer pointer must point to a valid bearer structure (i.e. cannot be NULL), but bearer can be inactive.

int tipc_node_dump(struct tipc_node *n, bool more, char *buf)
dump TIPC node data

Parameters

struct tipc_node *n
tipc node to be dumped

bool more
dump more? - false: dump only tipc node data - true: dump node link data as well

char *buf
returned buffer of dump data in format

107.2.9 TIPC Socket Interfaces

struct tipc_sock
TIPC socket structure

Definition:

```
struct tipc_sock {
    struct sock sk;
    u32 max_pkt;
    u32 maxnagle;
    u32 portid;
    struct tipc_msg phdr;
    struct list_head cong_links;
    struct list_head publications;
    u32 pub_count;
    atomic_t dupl_rcvcnt;
```

```

u16 conn_timeout;
bool probe_unacked;
u16 cong_link_cnt;
u16 snt_unacked;
u16 snd_win;
u16 peer_caps;
u16 rcv_unacked;
u16 rcv_win;
struct sockaddr_tipc peer;
struct rhash_head node;
struct tipc_mc_method mc_method;
struct rcu_head rcu;
struct tipc_group *group;
u32 oneway;
u32 nagle_start;
u16 snd_backlog;
u16 msg_acc;
u16 pkt_cnt;
bool expect_ack;
bool nodelay;
bool group_is_open;
bool published;
u8 conn_addrtype;
};


```

Members

sk

socket - interacts with 'port' and with user via the socket API

max_pkt

maximum packet size "hint" used when building messages sent by port

maxnagle

maximum size of msg which can be subject to nagle

portid

unique port identity in TIPC socket hash table

phdr

preformatted message header used when sending messages

cong_links

list of congested links

publications

list of publications for port

pub_count

total # of publications port has made during its lifetime

dupl_rcvcnt

number of bytes counted twice, in both backlog and rcv queue

conn_timeout

the time we can wait for an unresponded setup request

probe_unacked

probe has not received ack yet

cong_link_cnt

number of congested links

snt_unacked

messages sent by socket, and not yet acked by peer

snd_win

send window size

peer_caps

peer capabilities mask

rcv_unacked

messages read by user, but not yet acked back to peer

rcv_win

receive window size

peer

'connected' peer for dgram/rdm

node

hash table node

mc_method

cookie for use between socket and broadcast layer

rcu

rcu struct for tipc_sock

group

TIPC communications group

oneway

message count in one direction (FIXME)

nagle_start

current nagle value

snd_backlog

send backlog count

msg_acc

messages accepted; used in managing backlog and nagle

pkt_cnt

TIPC socket packet count

expect_ack

whether this TIPC socket is expecting an ack

nodeelay

setsockopt() TIPC_NODELAY setting

group_is_open

TIPC socket group is fully open (FIXME)

published

true if port has one or more associated names

conn_addrtype

address type used when establishing connection

void tsk_advance_rx_queue(struct *sock* *sk)

discard first buffer in socket receive queue

Parameters

struct sock *sk

network socket

Description

Caller must hold socket lock

void tsk_rej_rx_queue(struct *sock* *sk, int error)

reject all buffers in socket receive queue

Parameters

struct sock *sk

network socket

int error

response error code

Description

Caller must hold socket lock

int tipc_sk_create(struct *net* *net, struct *socket* *sock, int protocol, int kern)

create a TIPC socket

Parameters

struct net *net

network namespace (must be default network)

struct socket *sock

pre-allocated socket structure

int protocol

protocol indicator (must be 0)

int kern

caused by kernel or by userspace?

Description

This routine creates additional data structures used by the TIPC socket, initializes them, and links them together.

Return

0 on success, errno otherwise

int tipc_release(struct *socket* *sock)

destroy a TIPC socket

Parameters

struct socket *sock
socket to destroy

Description

This routine cleans up any messages that are still queued on the socket. For DGRAM and RDM socket types, all queued messages are rejected. For SEQPACKET and STREAM socket types, the first message is rejected and any others are discarded. (If the first message on a STREAM socket is partially-read, it is discarded and the next one is rejected instead.)

NOTE

Rejected messages are not necessarily returned to the sender! They are returned or discarded according to the "destination droppable" setting specified for the message by the sender.

Return

0 on success, errno otherwise

```
int __tipc_bind(struct socket *sock, struct sockaddr *skaddr, int alen)  
    associate or disassociate TIPC name(s) with a socket
```

Parameters

struct socket *sock
socket structure

struct sockaddr *skaddr
socket address describing name(s) and desired operation

int alen
size of socket address data structure

Description

Name and name sequence binding are indicated using a positive scope value; a negative scope value unbinds the specified name. Specifying no name (i.e. a socket address length of 0) unbinds all names from the socket.

Return

0 on success, errno otherwise

NOTE

This routine doesn't need to take the socket lock since it doesn't access any non-constant socket information.

```
int tipc_getname(struct socket *sock, struct sockaddr *uaddr, int peer)  
    get port ID of socket or peer socket
```

Parameters

struct socket *sock
socket structure

struct sockaddr *uaddr
area for returned socket address

int peer

0 = own ID, 1 = current peer ID, 2 = current/former peer ID

Return

0 on success, errno otherwise

NOTE**This routine doesn't need to take the socket lock since it only**

accesses socket information that is unchanging (or which changes in a completely predictable manner).

`_poll_t tipc_poll(struct file *file, struct socket *sock, poll_table *wait)`

read and possibly block on pollmask

Parameters**struct file *file**

file structure associated with the socket

struct socket *sock

socket for which to calculate the poll bits

poll_table *wait

???

Return

pollmask value

Description

COMMENTARY: It appears that the usual socket locking mechanisms are not useful here since the pollmask info is potentially out-of-date the moment this routine exits. TCP and other protocols seem to rely on higher level poll routines to handle any preventable race conditions, so TIPC will do the same ...

IMPORTANT: The fact that a read or write operation is indicated does NOT imply that the operation will succeed, merely that it should be performed and will not block.

`int tipc_sendmcast(struct socket *sock, struct tipc_uaddr *ua, struct msghdr *msg, size_t dlen, long timeout)`

send multicast message

Parameters**struct socket *sock**

socket structure

struct tipc_uaddr *ua

destination address struct

struct msghdr *msg

message to send

size_t dlen

length of data to send

long timeout

timeout to wait for wakeup

Description

Called from function `tipc_sendmsg()`, which has done all sanity checks

Return

the number of bytes sent on success, or errno

```
int tipc_send_group_msg(struct net *net, struct tipc_sock *tsk, struct msghdr *m, struct tipc_member *mb, u32 dnode, u32 dport, int dlen)
```

send a message to a member in the group

Parameters

struct net *net

network namespace

struct tipc_sock *tsk

tipc socket

struct msghdr *m

message to send

struct tipc_member *mb

group member

u32 dnode

destination node

u32 dport

destination port

int dlen

total length of message data

```
int tipc_send_group_unicast(struct socket *sock, struct msghdr *m, int dlen, long timeout)
```

send message to a member in the group

Parameters

struct socket *sock

socket structure

struct msghdr *m

message to send

int dlen

total length of message data

long timeout

timeout to wait for wakeup

Description

Called from function `tipc_sendmsg()`, which has done all sanity checks

Return

the number of bytes sent on success, or errno

```
int tipc_send_group_anycast(struct socket *sock, struct msghdr *m, int dlen, long timeout)
    send message to any member with given identity
```

Parameters

struct socket *sock
socket structure

struct msghdr *m
message to send

int dlen
total length of message data

long timeout
timeout to wait for wakeup

Description

Called from function [tipc_sendmsg\(\)](#), which has done all sanity checks

Return

the number of bytes sent on success, or errno

```
int tipc_send_group_bcast(struct socket *sock, struct msghdr *m, int dlen, long timeout)
    send message to all members in communication group
```

Parameters

struct socket *sock
socket structure

struct msghdr *m
message to send

int dlen
total length of message data

long timeout
timeout to wait for wakeup

Description

Called from function [tipc_sendmsg\(\)](#), which has done all sanity checks

Return

the number of bytes sent on success, or errno

```
int tipc_send_group_mcast(struct socket *sock, struct msghdr *m, int dlen, long timeout)
    send message to all members with given identity
```

Parameters

struct socket *sock
socket structure

struct msghdr *m
message to send

int dlen
total length of message data

long timeout
timeout to wait for wakeup

Description

Called from function `tipc_sendmsg()`, which has done all sanity checks

Return

the number of bytes sent on success, or errno

void tipc_sk_mcast_rcv(struct net *net, struct sk_buff_head *arrvq, struct sk_buff_head *inputq)

Deliver multicast messages to all destination sockets

Parameters

struct net *net
the associated network namespace

struct sk_buff_head *arrvq
queue with arriving messages, to be cloned after destination lookup

struct sk_buff_head *inputq
queue with cloned messages, delivered to socket after dest lookup

Description

Multi-threaded: parallel calls with reference to same queues may occur

void tipc_sk_conn_proto_rcv(struct tipc_sock *tsk, struct sk_buff *skb, struct sk_buff_head *inputq, struct sk_buff_head *xmitq)

receive a connection mng protocol message

Parameters

struct tipc_sock *tsk
receiving socket

struct sk_buff *skb
pointer to message buffer.

struct sk_buff_head *inputq
buffer list containing the buffers

struct sk_buff_head *xmitq
output message area

int tipc_sendmsg(struct socket *sock, struct msghdr *m, size_t dsz)
send message in connectionless manner

Parameters

struct socket *sock
socket structure

struct msghdr *m
message to send

size_t dsz
amount of user data to be sent

Description

Message must have an destination specified explicitly. Used for SOCK_RDM and SOCK_DGRAM messages, and for 'SYN' messages on SOCK_SEQPACKET and SOCK_STREAM connections. (Note: 'SYN+' is prohibited on SOCK_STREAM.)

Return

the number of bytes sent on success, or errno otherwise

```
int tipc_sendstream(struct socket *sock, struct msghdr *m, size_t dsz)
    send stream-oriented data
```

Parameters

struct socket *sock
socket structure

struct msghdr *m
data to send

size_t dsz
total length of data to be transmitted

Description

Used for SOCK_STREAM data.

Return

the number of bytes sent on success (or partial success), or errno if no data sent

```
int tipc_send_packet(struct socket *sock, struct msghdr *m, size_t dsz)
    send a connection-oriented message
```

Parameters

struct socket *sock
socket structure

struct msghdr *m
message to send

size_t dsz
length of data to be transmitted

Description

Used for SOCK_SEQPACKET messages.

Return

the number of bytes sent on success, or errno otherwise

```
void tipc_sk_set_orig_addr(struct msghdr *m, struct sk_buff *skb)
    capture sender's address for received message
```

Parameters

struct msghdr *m
descriptor for message info

struct sk_buff *skb
received message

Note

Address is not captured if not requested by receiver.

int tipc_sk_anc_data_recv(struct msghdr *m, struct sk_buff *skb, struct tipc_sock *tsk)
optionally capture ancillary data for received message

Parameters

struct msghdr *m
descriptor for message info

struct sk_buff *skb
received message buffer

struct tipc_sock *tsk
TIPC port associated with message

Note

Ancillary data is not captured if not requested by receiver.

Return

0 if successful, otherwise errno

int tipc_recvmsg(struct socket *sock, struct msghdr *m, size_t buflen, int flags)
receive packet-oriented message

Parameters

struct socket *sock
network socket

struct msghdr *m
descriptor for message info

size_t buflen
length of user buffer area

int flags
receive flags

Description

Used for SOCK_DGRAM, SOCK_RDM, and SOCK_SEQPACKET messages. If the complete message doesn't fit in user area, truncate it.

Return

size of returned message data, errno otherwise

int tipc_recvstream(struct socket *sock, struct msghdr *m, size_t buflen, int flags)
receive stream-oriented data

Parameters

struct socket *sock
network socket

struct msghdr *m
 descriptor for message info

size_t buflen
 total size of user buffer area

int flags
 receive flags

Description

Used for SOCK_STREAM messages only. If not enough data is available will optionally wait for more; never truncates data.

Return

size of returned message data, errno otherwise

void tipc_write_space(struct sock *sk)
 wake up thread if port congestion is released

Parameters

struct sock *sk
 socket

void tipc_data_ready(struct sock *sk)
 wake up threads to indicate messages have been received

Parameters

struct sock *sk
 socket

bool tipc_sk_filter_connect(struct tipc_sock *tsk, struct sk_buff *skb, struct sk_buff_head *xmitq)

check incoming message for a connection-based socket

Parameters

struct tipc_sock *tsk
 TIPC socket

struct sk_buff *skb
 pointer to message buffer.

struct sk_buff_head *xmitq
 for Nagle ACK if any

Return

true if message should be added to receive queue, false otherwise

unsigned int rcvbuf_limit(struct sock *sk, struct sk_buff *skb)
 get proper overload limit of socket receive queue

Parameters

struct sock *sk
 socket

struct sk_buff *skb
message

Description

For connection oriented messages, irrespective of importance, default queue limit is 2 MB.

For connectionless messages, queue limits are based on message importance as follows:

TIPC_LOW_IMPORTANCE	(2 MB)	TIPC_MEDIUM_IMPORTANCE	(4 MB)
TIPC_HIGH_IMPORTANCE	(8 MB)	TIPC_CRITICAL_IMPORTANCE	(16 MB)

Return

overload limit according to corresponding message importance

void tipc_sk_filter_rcv(struct sock *sk, struct sk_buff *skb, struct sk_buff_head *xmitq)
validate incoming message

Parameters

struct sock *sk
socket

struct sk_buff *skb
pointer to message.

struct sk_buff_head *xmitq
output message area (FIXME)

Description

Enqueues message on receive queue if acceptable; optionally handles disconnect indication for a connected socket.

Called with socket lock already taken

int tipc_sk_backlog_rcv(struct sock *sk, struct sk_buff *skb)
handle incoming message from backlog queue

Parameters

struct sock *sk
socket

struct sk_buff *skb
message

Description

Caller must hold socket lock

void tipc_sk_enqueue(struct sk_buff_head *inputq, struct sock *sk, u32 dport, struct sk_buff_head *xmitq)
extract all buffers with destination 'dport' from inputq and try adding them to socket or backlog queue

Parameters

struct sk_buff_head *inputq
list of incoming buffers with potentially different destinations

struct sock *sk
socket where the buffers should be enqueued

u32 dport
port number for the socket

struct sk_buff_head *xmitq
output queue

Description

Caller must hold socket lock

void tipc_sk_rcv(struct net *net, struct sk_buff_head *inputq)
handle a chain of incoming buffers

Parameters

struct net *net
the associated network namespace

struct sk_buff_head *inputq
buffer list containing the buffers Consumes all buffers in list until inputq is empty

Note

may be called in multiple threads referring to the same queue

int tipc_connect(struct socket *sock, struct sockaddr *dest, int destlen, int flags)
establish a connection to another TIPC port

Parameters

struct socket *sock
socket structure

struct sockaddr *dest
socket address for destination port

int destlen
size of socket address data structure

int flags
file-related flags associated with socket

Return

0 on success, errno otherwise

int tipc_listen(struct socket *sock, int len)
allow socket to listen for incoming connections

Parameters

struct socket *sock
socket structure

int len
(unused)

Return

0 on success, errno otherwise

```
int tipc_accept(struct socket *sock, struct socket *new_sock, int flags, bool kern)
    wait for connection request
```

Parameters

struct socket *sock
listening socket

struct socket *new_sock
new socket that is to be connected

int flags
file-related flags associated with socket

bool kern
caused by kernel or by userspace?

Return

0 on success, errno otherwise

```
int tipc_shutdown(struct socket *sock, int how)
    shutdown socket connection
```

Parameters

struct socket *sock
socket structure

int how
direction to close (must be SHUT_RDWR)

Description

Terminates connection (if necessary), then purges socket's receive queue.

Return

0 on success, errno otherwise

```
int tipc_setsockopt(struct socket *sock, int lvl, int opt, sockptr_t ov, unsigned int ol)
    set socket option
```

Parameters

struct socket *sock
socket structure

int lvl
option level

int opt
option identifier

sockptr_t ov
pointer to new option value

unsigned int ol
length of option value

Description

For stream sockets only, accepts and ignores all IPPROTO_TCP options (to ease compatibility).

Return

0 on success, errno otherwise

```
int tipc_getsockopt(struct socket *sock, int lvl, int opt, char __user *ov, int __user *ol)
    get socket option
```

Parameters

struct socket *sock
socket structure

int lvl
option level

int opt
option identifier

char __user *ov
receptacle for option value

int __user *ol
receptacle for length of option value

Description

For stream sockets only, returns 0 length result for all IPPROTO_TCP options (to ease compatibility).

Return

0 on success, errno otherwise

```
int tipc_socket_init(void)
    initialize TIPC socket interface
```

Parameters

void
no arguments

Return

0 on success, errno otherwise

```
void tipc_socket_stop(void)
    stop TIPC socket interface
```

Parameters

void
no arguments

```
bool tipc_sk_filtering(struct sock *sk)
    check if a socket should be traced
```

Parameters

struct sock *sk
the socket to be examined

Description

sysctl_tipc_sk_filter is used as the socket tuple for filtering: (portid, sock type, name type, name lower, name upper)

Return

true if the socket meets the socket tuple data (value 0 = 'any') or when there is no tuple set (all = 0), otherwise false

bool tipc_sk_overlimit1(struct *sock* *sk, struct *sk_buff* *skb)

check if socket rx queue is about to be overloaded, both the rcv and backlog queues are considered

Parameters

struct sock *sk

tipc sk to be checked

struct sk_buff *skb

tipc msg to be checked

Return

true if the socket rx queue allocation is > 90%, otherwise false

bool tipc_sk_overlimit2(struct *sock* *sk, struct *sk_buff* *skb)

check if socket rx queue is about to be overloaded, only the rcv queue is considered

Parameters

struct sock *sk

tipc sk to be checked

struct sk_buff *skb

tipc msg to be checked

Return

true if the socket rx queue allocation is > 90%, otherwise false

int tipc_sk_dump(struct *sock* *sk, u16 dqueues, char *buf)

dump TIPC socket

Parameters

struct sock *sk

tipc sk to be dumped

u16 dqueues

bitmask to decide if any socket queue to be dumped? - TIPC_DUMP_NONE: don't dump socket queues - TIPC_DUMP_SK_SNDQ: dump socket send queue - TIPC_DUMP_SK_RCVQ: dump socket rcv queue - TIPC_DUMP_SK_BKLQ: dump socket backlog queue - TIPC_DUMP_ALL: dump all the socket queues above

char *buf

returned buffer of dump data in format

107.2.10 TIPC Network Topology Interfaces

```
bool tipc_sub_check_overlap(struct tipc_service_range *subscribed, struct
                           tipc_service_range *found)
```

test for subscription overlap with the given values

Parameters

struct tipc_service_range *subscribed

the service range subscribed for

struct tipc_service_range *found

the service range we are checking for match

Description

Returns true if there is overlap, otherwise false.

107.2.11 TIPC Server Interfaces

struct tipc_topsrv

TIPC server structure

Definition:

```
struct tipc_topsrv {
    struct idr conn_idr;
    spinlock_t idr_lock;
    int idr_in_use;
    struct net *net;
    struct work_struct awork;
    struct workqueue_struct *recv_wq;
    struct workqueue_struct *send_wq;
    struct socket *listener;
    char name[TIPC_SERVER_NAME_LEN];
};
```

Members

conn_idr

identifier set of connection

idr_lock

protect the connection identifier set

idr_in_use

amount of allocated identifier entry

net

network namespace instance

awork

accept work item

recv_wq

receive workqueue

send_wq
send workqueue

listener
topsrv listener socket

name
server name

struct tipc_conn
TIPC connection structure

Definition:

```
struct tipc_conn {  
    struct kref kref;  
    int conid;  
    struct socket *sock;  
    unsigned long flags;  
    struct tipc_topsrv *server;  
    struct list_head sub_list;  
    spinlock_t sub_lock;  
    struct work_struct rwork;  
    struct list_head outqueue;  
    spinlock_t outqueue_lock;  
    struct work_struct swork;  
};
```

Members

kref
reference counter to connection object

conid
connection identifier

sock
socket handler associated with connection

flags
indicates connection state

server
pointer to connected server

sub_list
list to all pertaining subscriptions

sub_lock
lock protecting the subscription list

rwork
receive work item

outqueue
pointer to first outbound message in queue

outqueue_lock
control access to the outqueue

swork
send work item

107.2.12 TIPC Trace Interfaces

int tipc_skb_dump(struct sk_buff *skb, bool more, char *buf)
dump TIPC skb data

Parameters

struct sk_buff *skb
skb to be dumped

bool more
dump more? - false: dump only tipc msg data - true: dump kernel-related skb data and tipc cb[] array as well

char *buf
returned buffer of dump data in format

int tipc_list_dump(struct sk_buff_head *list, bool more, char *buf)
dump TIPC skb list/queue

Parameters

struct sk_buff_head *list
list of skbs to be dumped

bool more
dump more? - false: dump only the head & tail skbs - true: dump the first & last 5 skbs

char *buf
returned buffer of dump data in format

TRANSPARENT PROXY SUPPORT

This feature adds Linux 2.2-like transparent proxy support to current kernels. To use it, enable the socket match and the TPROXY target in your kernel config. You will need policy routing too, so be sure to enable that as well.

From Linux 4.18 transparent proxy support is also available in nf_tables.

108.1 1. Making non-local sockets work

The idea is that you identify packets with destination address matching a local socket on your box, set the packet mark to a certain value:

```
# iptables -t mangle -N DIVERT
# iptables -t mangle -A PREROUTING -p tcp -m socket -j DIVERT
# iptables -t mangle -A DIVERT -j MARK --set-mark 1
# iptables -t mangle -A DIVERT -j ACCEPT
```

Alternatively you can do this in nft with the following commands:

```
# nft add table filter
# nft add chain filter divert "{ type filter hook prerouting priority -150; }"
# nft add rule filter divert meta l4proto tcp socket transparent 1 meta mark
  ↳ set 1 accept
```

And then match on that value using policy routing to have those packets delivered locally:

```
# ip rule add fwmark 1 lookup 100
# ip route add local 0.0.0.0/0 dev lo table 100
```

Because of certain restrictions in the IPv4 routing output code you'll have to modify your application to allow it to send datagrams _from_ non-local IP addresses. All you have to do is enable the (SOL_IP, IP_TRANSPARENT) socket option before calling bind:

```
fd = socket(AF_INET, SOCK_STREAM, 0);
/* - 8< */
int value = 1;
setsockopt(fd, SOL_IP, IP_TRANSPARENT, &value, sizeof(value));
/* - 8< */
name.sin_family = AF_INET;
name.sin_port = htons(0xCAFE);
```

```
name.sin_addr.s_addr = htonl(0xDEADBEEF);  
bind(fd, &name, sizeof(name));
```

A trivial patch for netcat is available here: http://people.netfilter.org/hidden/tproxy/netcat-ip_transparent-support.patch

108.2 2. Redirecting traffic

Transparent proxying often involves "intercepting" traffic on a router. This is usually done with the iptables REDIRECT target; however, there are serious limitations of that method. One of the major issues is that it actually modifies the packets to change the destination address -- which might not be acceptable in certain situations. (Think of proxying UDP for example: you won't be able to find out the original destination address. Even in case of TCP getting the original destination address is racy.)

The 'TPROXY' target provides similar functionality without relying on NAT. Simply add rules like this to the iptables ruleset above:

```
# iptables -t mangle -A PREROUTING -p tcp --dport 80 -j TPROXY \  
--tproxy-mark 0x1/0x1 --on-port 50080
```

Or the following rule to nft:

```
# nft add rule filter divert tcp dport 80 tproxy to :50080 meta mark set 1 accept
```

Note that for this to work you'll have to modify the proxy to enable (SOL_IP, IP_TRANSPARENT) for the listening socket.

As an example implementation, tcprdr is available here: <https://git.breakpoint.cc/cgit/fw/tcprdr.git/> This tool is written by Florian Westphal and it was used for testing during the nf_tables implementation.

108.3 3. Iptables and nf_tables extensions

To use tproxy you'll need to have the following modules compiled for iptables:

- NETFILTER_XT_MATCH_SOCKET
- NETFILTER_XT_TARGET_TPROXY

Or the flowing modules for nf_tables:

- NFT_SOCKET
- NFT_TPROXY

108.4 4. Application support

108.4.1 4.1. Squid

Squid 3.HEAD has support built-in. To use it, pass '--enable-linux-netfilter' to configure and set the 'tproxy' option on the HTTP listener you redirect traffic to with the TPROXY iptables target.

For more information please consult the following page on the Squid wiki: <http://wiki.squid-cache.org/Features/Tproxy4>

UNIVERSAL TUN/TAP DEVICE DRIVER

Copyright © 1999-2000 Maxim Krasnyansky <max_mk@yahoo.com>

Linux, Solaris drivers Copyright © 1999-2000 Maxim Krasnyansky
<max_mk@yahoo.com>

FreeBSD TAP driver Copyright © 1999-2000 Maksim Yevmenkin
<m_evmenkin@yahoo.com>

Revision of this document 2002 by Florian Thiel <florian.thiel@gmx.net>

109.1 1. Description

TUN/TAP provides packet reception and transmission for user space programs. It can be seen as a simple Point-to-Point or Ethernet device, which, instead of receiving packets from physical media, receives them from user space program and instead of sending packets via physical media writes them to the user space program.

In order to use the driver a program has to open /dev/net/tun and issue a corresponding ioctl() to register a network device with the kernel. A network device will appear as tunXX or tapXX, depending on the options chosen. When the program closes the file descriptor, the network device and all corresponding routes will disappear.

Depending on the type of device chosen the userspace program has to read/write IP packets (with tun) or ethernet frames (with tap). Which one is being used depends on the flags given with the ioctl().

The package from <http://vtun.sourceforge.net/tun> contains two simple examples for how to use tun and tap devices. Both programs work like a bridge between two network interfaces. br_select.c - bridge based on select system call. br_sigio.c - bridge based on async io and SIGIO signal. However, the best example is VTun <http://vtun.sourceforge.net> :))

109.2 2. Configuration

Create device node:

```
mkdir /dev/net (if it doesn't exist already)
mknod /dev/net/tun c 10 200
```

Set permissions:

```
e.g. chmod 0666 /dev/net/tun
```

There's no harm in allowing the device to be accessible by non-root users, since CAP_NET_ADMIN is required for creating network devices or for connecting to network devices which aren't owned by the user in question. If you want to create persistent devices and give ownership of them to unprivileged users, then you need the /dev/net/tun device to be usable by those users.

Driver module autoloading

Make sure that "Kernel module loader" - module auto-loading support is enabled in your kernel. The kernel should load it on first access.

Manual loading

insert the module by hand:

```
modprobe tun
```

If you do it the latter way, you have to load the module every time you need it, if you do it the other way it will be automatically loaded when /dev/net/tun is being opened.

109.3 3. Program interface

109.3.1 3.1 Network device allocation

char *dev should be the name of the device with a format string (e.g. "tun%d"), but (as far as I can see) this can be any valid network device name. Note that the character pointer becomes overwritten with the real device name (e.g. "tun0"):

```
#include <linux/if.h>
#include <linux/if_tun.h>

int tun_alloc(char *dev)
{
    struct ifreq ifr;
    int fd, err;

    if( (fd = open("/dev/net/tun", O_RDWR)) < 0 )
        return tun_alloc_old(dev);

    memset(&ifr, 0, sizeof(ifr));
```

```

/* Flags: IFF_TUN    - TUN device (no Ethernet headers)
 *        IFF_TAP    - TAP device
 *
 *        IFF_NO_PI - Do not provide packet information
 */
ifr.ifr_flags = IFF_TUN;
if( *dev )
    strncpy_pad(ifr.ifr_name, dev, IFNAMSIZ);

if( (err = ioctl(fd, TUNSETIFF, (void *) &ifr)) < 0 ){
    close(fd);
    return err;
}
strcpy(dev, ifr.ifr_name);
return fd;
}

```

109.3.2 3.2 Frame format

If flag IFF_NO_PI is not set each frame format is:

```

Flags [2 bytes]
Proto [2 bytes]
Raw protocol(IP, IPv6, etc) frame.

```

109.3.3 3.3 Multiqueue tun/tap interface

From version 3.8, Linux supports multiqueue tun/tap which can uses multiple file descriptors (queues) to parallelize packets sending or receiving. The device allocation is the same as before, and if user wants to create multiple queues, TUNSETIFF with the same device name must be called many times with IFF_MULTI_QUEUE flag.

char *dev should be the name of the device, queues is the number of queues to be created, fds is used to store and return the file descriptors (queues) created to the caller. Each file descriptor were served as the interface of a queue which could be accessed by userspace.

```

#include <linux/if.h>
#include <linux/if_tun.h>

int tun_alloc_mq(char *dev, int queues, int *fds)
{
    struct ifreq ifr;
    int fd, err, i;

    if (!dev)
        return -1;

    memset(&ifr, 0, sizeof(ifr));
    /* Flags: IFF_TUN    - TUN device (no Ethernet headers)
     *        IFF_TAP    - TAP device
     *
     *        IFF_NO_PI - Do not provide packet information
     */
    ifr.ifr_flags = IFF_TUN;
    if( *dev )
        strncpy_pad(ifr.ifr_name, dev, IFNAMSIZ);

    if( (err = ioctl(fd, TUNSETIFF, (void *) &ifr)) < 0 ){
        close(fd);
        return err;
    }
    strcpy(dev, ifr.ifr_name);
    return fd;
}

```

```

*           IFF_TAP      - TAP device
*
*           IFF_NO_PI   - Do not provide packet information
*           IFF_MULTI_QUEUE - Create a queue of multiqueue device
*/
ifr.ifr_flags = IFF_TAP | IFF_NO_PI | IFF_MULTI_QUEUE;
strcpy(ifr.ifr_name, dev);

for (i = 0; i < queues; i++) {
    if ((fd = open("/dev/net/tun", O_RDWR)) < 0)
        goto err;
    err = ioctl(fd, TUNSETIFF, (void *)&ifr);
    if (err) {
        close(fd);
        goto err;
    }
    fds[i] = fd;
}

return 0;
err:
for (--i; i >= 0; i--)
    close(fds[i]);
return err;
}

```

A new ioctl(TUNSETQUEUE) were introduced to enable or disable a queue. When calling it with IFF_DETACH_QUEUE flag, the queue were disabled. And when calling it with IFF_ATTACH_QUEUE flag, the queue were enabled. The queue were enabled by default after it was created through TUNSETIFF.

fd is the file descriptor (queue) that we want to enable or disable, when enable is true we enable it, otherwise we disable it:

```

#include <linux/if.h>
#include <linux/if_tun.h>

int tun_set_queue(int fd, int enable)
{
    struct ifreq ifr;

    memset(&ifr, 0, sizeof(ifr));

    if (enable)
        ifr.ifr_flags = IFF_ATTACH_QUEUE;
    else
        ifr.ifr_flags = IFF_DETACH_QUEUE;

    return ioctl(fd, TUNSETQUEUE, (void *)&ifr);
}

```

109.4 Universal TUN/TAP device driver Frequently Asked Question

1. What platforms are supported by TUN/TAP driver ?

Currently driver has been written for 3 Unices:

- Linux kernels 2.2.x, 2.4.x
- FreeBSD 3.x, 4.x, 5.x
- Solaris 2.6, 7.0, 8.0

2. What is TUN/TAP driver used for?

As mentioned above, main purpose of TUN/TAP driver is tunneling. It is used by VTun (<http://vtun.sourceforge.net>).

Another interesting application using TUN/TAP is pipsecd (<http://perso.enst.fr/~beyssac/pipsec/>), a userspace IPSec implementation that can use complete kernel routing (unlike FreeS/WAN).

3. How does Virtual network device actually work ?

Virtual network device can be viewed as a simple Point-to-Point or Ethernet device, which instead of receiving packets from a physical media, receives them from user space program and instead of sending packets via physical media sends them to the user space program.

Let's say that you configured IPv6 on the tap0, then whenever the kernel sends an IPv6 packet to tap0, it is passed to the application (VTun for example). The application encrypts, compresses and sends it to the other side over TCP or UDP. The application on the other side decompresses and decrypts the data received and writes the packet to the TAP device, the kernel handles the packet like it came from real physical device.

4. What is the difference between TUN driver and TAP driver?

TUN works with IP frames. TAP works with Ethernet frames.

This means that you have to read/write IP packets when you are using tun and ethernet frames when using tap.

5. What is the difference between BPF and TUN/TAP driver?

BPF is an advanced packet filter. It can be attached to existing network interface. It does not provide a virtual network interface. A TUN/TAP driver does provide a virtual network interface and it is possible to attach BPF to this interface.

6. Does TAP driver support kernel Ethernet bridging?

Yes. Linux and FreeBSD drivers support Ethernet bridging.

THE UDP-LITE PROTOCOL (RFC 3828)

UDP-Lite is a Standards-Track IETF transport protocol whose characteristic is a variable-length checksum. This has advantages for transport of multimedia (video, VoIP) over wireless networks, as partly damaged packets can still be fed into the codec instead of being discarded due to a failed checksum test.

This file briefly describes the existing kernel support and the socket API. For in-depth information, you can consult:

- The UDP-Lite Homepage: <http://web.archive.org/web/%2E/http://www.erg.abdn.ac.uk/users/gerrit/udp-lite/>

From here you can also download some example application source code.

- The UDP-Lite HOWTO on <http://web.archive.org/web/%2E/http://www.erg.abdn.ac.uk/users/gerrit/udp-lite/files/UDP-Lite-HOWTO.txt>
- The Wireshark UDP-Lite WiKi (with capture files): https://wiki.wireshark.org/Lightweight_User_Datagram_Protocol
- The Protocol Spec, RFC 3828, <http://www.ietf.org/rfc/rfc3828.txt>

110.1 1. Applications

Several applications have been ported successfully to UDP-Lite. Ethereal (now called wireshark) has UDP-Litev4/v6 support by default.

Porting applications to UDP-Lite is straightforward: only socket level and IPPROTO need to be changed; senders additionally set the checksum coverage length (default = header length = 8). Details are in the next section.

110.2 2. Programming API

UDP-Lite provides a connectionless, unreliable datagram service and hence uses the same socket type as UDP. In fact, porting from UDP to UDP-Lite is very easy: simply add IPPROTO_UDPLITE as the last argument of the socket(2) call so that the statement looks like:

```
s = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDPLITE);
```

or, respectively,

```
s = socket(PF_INET6, SOCK_DGRAM, IPPROTO_UDPLITE);
```

With just the above change you are able to run UDP-Lite services or connect to UDP-Lite servers. The kernel will assume that you are not interested in using partial checksum coverage and so emulate UDP mode (full coverage).

To make use of the partial checksum coverage facilities requires setting a single socket option, which takes an integer specifying the coverage length:

- Sender checksum coverage: UDPLITE_SEND_CSCOV

For example:

```
int val = 20;
setsockopt(s, SOL_UDPLITE, UDPLITE_SEND_CSCOV, &val, sizeof(int));
```

sets the checksum coverage length to 20 bytes (12b data + 8b header). Of each packet only the first 20 bytes (plus the pseudo-header) will be checksummed. This is useful for RTP applications which have a 12-byte base header.

- Receiver checksum coverage: UDPLITE_RECV_CSCOV

This option is the receiver-side analogue. It is truly optional, i.e. not required to enable traffic with partial checksum coverage. Its function is that of a traffic filter: when enabled, it instructs the kernel to drop all packets which have a coverage *less* than this value. For example, if RTP and UDP headers are to be protected, a receiver can enforce that only packets with a minimum coverage of 20 are admitted:

```
int min = 20;
setsockopt(s, SOL_UDPLITE, UDPLITE_RECV_CSCOV, &min, sizeof(int));
```

The calls to `getsockopt(2)` are analogous. Being an extension and not a stand-alone protocol, all socket options known from UDP can be used in exactly the same manner as before, e.g. `UDP_CORK` or `UDP_ENCAP`.

A detailed discussion of UDP-Lite checksum coverage options is in section IV.

110.3 Header Files

The socket API requires support through header files in `/usr/include`:

- `/usr/include/netinet/in.h` to define `IPPROTO_UDPLITE`
- `/usr/include/netinet/udplite.h` for UDP-Lite header fields and protocol constants

For testing purposes, the following can serve as a `mini` header file:

<code>#define IPPROTO_UDPLITE</code>	136
<code>#define SOL_UDPLITE</code>	136
<code>#define UDPLITE_SEND_CSCOV</code>	10
<code>#define UDPLITE_RECV_CSCOV</code>	11

Ready-made header files for various distros are in the UDP-Lite tarball.

110.4 4. Kernel Behaviour with Regards to the Various Socket Options

To enable debugging messages, the log level need to be set to 8, as most messages use the KERN_DEBUG level (7).

1) Sender Socket Options

If the sender specifies a value of 0 as coverage length, the module assumes full coverage, transmits a packet with coverage length of 0 and according checksum. If the sender specifies a coverage < 8 and different from 0, the kernel assumes 8 as default value. Finally, if the specified coverage length exceeds the packet length, the packet length is used instead as coverage length.

2) Receiver Socket Options

The receiver specifies the minimum value of the coverage length it is willing to accept. A value of 0 here indicates that the receiver always wants the whole of the packet covered. In this case, all partially covered packets are dropped and an error is logged.

It is not possible to specify illegal values (<0 and <8); in these cases the default of 8 is assumed.

All packets arriving with a coverage value less than the specified threshold are discarded, these events are also logged.

3) Disabling the Checksum Computation

On both sender and receiver, checksumming will always be performed and cannot be disabled using SO_NO_CHECK. Thus:

```
setsockopt(sockfd, SOL_SOCKET, SO_NO_CHECK, ... );
```

will always will be ignored, while the value of:

```
getsockopt(sockfd, SOL_SOCKET, SO_NO_CHECK, &value, ... );
```

is meaningless (as in TCP). Packets with a zero checksum field are illegal (cf. RFC 3828, sec. 3.1) and will be silently discarded.

4) Fragmentation

The checksum computation respects both buffersize and MTU. The size of UDP-Lite packets is determined by the size of the send buffer. The minimum size of the send buffer is 2048 (defined as SOCK_MIN_SNDBUF in include/net/sock.h), the default value is configurable as net.core.wmem_default or via setting the SO_SNDBUF socket(7) option. The maximum upper bound for the send buffer is determined by net.core.wmem_max.

Given a payload size larger than the send buffer size, UDP-Lite will split the payload into several individual packets, filling up the send buffer size in each case.

The precise value also depends on the interface MTU. The interface MTU, in turn, may trigger IP fragmentation. In this case, the generated UDP-Lite packet is split into several IP packets, of which only the first one contains the L4 header.

The send buffer size has implications on the checksum coverage length. Consider the following example:

Payload: 1536 bytes	Send Buffer: 1024 bytes
MTU: 1500 bytes	Coverage Length: 856 bytes

UDP-Lite will ship the 1536 bytes in two separate packets:

Packet 1: 1024 payload + 8 byte header + 20 byte IP header = 1052 bytes
Packet 2: 512 payload + 8 byte header + 20 byte IP header = 540 bytes

The coverage packet covers the UDP-Lite header and 848 bytes of the payload in the first packet, the second packet is fully covered. Note that for the second packet, the coverage length exceeds the packet length. The kernel always re-adjusts the coverage length to the packet length in such cases.

As an example of what happens when one UDP-Lite packet is split into several tiny fragments, consider the following example:

Payload: 1024 bytes	Send buffer size: 1024 bytes
MTU: 300 bytes	Coverage length: 575 bytes
<pre> +-----+-----+-----+ 8 272 280 280 280 +-----+-----+-----+ 280 560 840 1032 ^ *****checksum coverage***** </pre>	

The UDP-Lite module generates one 1032 byte packet (1024 + 8 byte header). According to the interface MTU, these are split into 4 IP packets (280 byte IP payload + 20 byte IP header). The kernel module sums the contents of the entire first two packets, plus 15 bytes of the last packet before releasing the fragments to the IP module.

To see the analogous case for IPv6 fragmentation, consider a link MTU of 1280 bytes and a write buffer of 3356 bytes. If the checksum coverage is less than 1232 bytes (MTU minus IPv6/fragment header lengths), only the first fragment needs to be considered. When using larger checksum coverage lengths, each eligible fragment needs to be checksummed. Suppose we have a checksum coverage of 3062. The buffer of 3356 bytes will be split into the following fragments:

Fragment 1: 1280 bytes carrying 1232 bytes of UDP-Lite data
Fragment 2: 1280 bytes carrying 1232 bytes of UDP-Lite data
Fragment 3: 948 bytes carrying 900 bytes of UDP-Lite data

The first two fragments have to be checksummed in full, of the last fragment only 598 (= 3062 - 2*1232) bytes are checksummed.

While it is important that such cases are dealt with correctly, they are (annoyingly) rare: UDP-Lite is designed for optimising multimedia performance over wireless (or generally noisy) links and thus smaller coverage lengths are likely to be expected.

110.5 5. UDP-Lite Runtime Statistics and their Meaning

Exceptional and error conditions are logged to syslog at the KERN_DEBUG level. Live statistics about UDP-Lite are available in /proc/net/snmp and can (with newer versions of netstat) be viewed using:

```
netstat -svu
```

This displays UDP-Lite statistics variables, whose meaning is as follows.

InDatagrams	The total number of datagrams delivered to users.
NoPorts	Number of packets received to an unknown port. These cases are counted separately (not as In-Errors).
InErrors	Number of erroneous UDP-Lite packets. Errors include: <ul style="list-style-type: none"> • internal socket queue receive errors • packet too short (less than 8 bytes or stated coverage length exceeds received length) • xfrm4_policy_check() returned with error • application has specified larger min. coverage length than that of incoming packet • checksum coverage violated • bad checksum
OutDatagrams	Total number of sent datagrams.

These statistics derive from the UDP MIB (RFC 2013).

110.6 6. IPtables

There is packet match support for UDP-Lite as well as support for the LOG target. If you copy and paste the following line into /etc/protocols:

```
udplite 136      UDP-Lite      # UDP-Lite [RFC 3828]
```

then:

```
iptables -A INPUT -p udplite -j LOG
```

will produce logging output to syslog. Dropping and rejecting packets also works.

110.7 7. Maintainer Address

The UDP-Lite patch was developed at

University of Aberdeen Electronics Research Group Department of Engineering
Fraser Noble Building Aberdeen AB24 3UE; UK

The current maintainer is Gerrit Renker, <gerrit@erg.abdn.ac.uk>. Initial code was developed by William Stanislaus, <wiliam@erg.abdn.ac.uk>.

VIRTUAL ROUTING AND FORWARDING (VRF)

111.1 The VRF Device

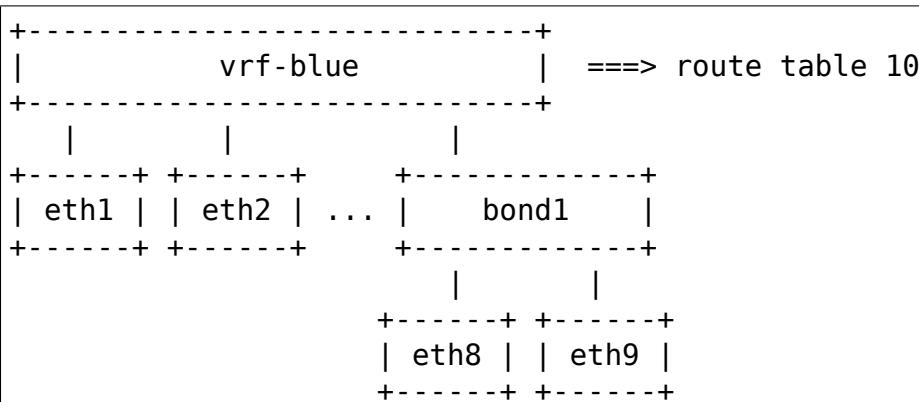
The VRF device combined with ip rules provides the ability to create virtual routing and forwarding domains (aka VRFs, VRF-lite to be specific) in the Linux network stack. One use case is the multi-tenancy problem where each tenant has their own unique routing tables and in the very least need different default gateways.

Processes can be "VRF aware" by binding a socket to the VRF device. Packets through the socket then use the routing table associated with the VRF device. An important feature of the VRF device implementation is that it impacts only Layer 3 and above so L2 tools (e.g., LLDP) are not affected (ie., they do not need to be run in each VRF). The design also allows the use of higher priority ip rules (Policy Based Routing, PBR) to take precedence over the VRF device rules directing specific traffic as desired.

In addition, VRF devices allow VRFs to be nested within namespaces. For example network namespaces provide separation of network interfaces at the device layer, VLANs on the interfaces within a namespace provide L2 separation and then VRF devices provide L3 separation.

111.1.1 Design

A VRF device is created with an associated route table. Network interfaces are then enslaved to a VRF device:



Packets received on an enslaved device and are switched to the VRF device in the IPv4 and IPv6 processing stacks giving the impression that packets flow through the VRF device. Similarly on egress routing rules are used to send packets to the VRF device driver before getting sent out the actual interface. This allows tcpdump on a VRF device to capture all packets into and out

of the VRF as a whole¹. Similarly, netfilter² and tc rules can be applied using the VRF device to specify rules that apply to the VRF domain as a whole.

111.1.2 Setup

1. VRF device is created with an association to a FIB table. e.g.:

```
ip link add vrf-blue type vrf table 10
ip link set dev vrf-blue up
```

2. An l3mdev FIB rule directs lookups to the table associated with the device. A single l3mdev rule is sufficient for all VRFs. The VRF device adds the l3mdev rule for IPv4 and IPv6 when the first device is created with a default preference of 1000. Users may delete the rule if desired and add with a different priority or install per-VRF rules.

Prior to the v4.8 kernel iif and oif rules are needed for each VRF device:

```
ip ru add oif vrf-blue table 10
ip ru add iif vrf-blue table 10
```

3. Set the default route for the table (and hence default route for the VRF):

```
ip route add table 10 unreachable default metric 4278198272
```

This high metric value ensures that the default unreachable route can be overridden by a routing protocol suite. FRRouting interprets kernel metrics as a combined admin distance (upper byte) and priority (lower 3 bytes). Thus the above metric translates to [255/8192].

4. Enslave L3 interfaces to a VRF device:

```
ip link set dev eth1 master vrf-blue
```

Local and connected routes for enslaved devices are automatically moved to the table associated with VRF device. Any additional routes depending on the enslaved device are dropped and will need to be reinserted to the VRF FIB table following the enslavement.

The IPv6 sysctl option `keep_addr_on_down` can be enabled to keep IPv6 global addresses as VRF enslavement changes:

```
sysctl -w net.ipv6.conf.all.keep_addr_on_down=1
```

5. Additional VRF routes are added to associated table:

```
ip route add table 10 ...
```

¹ Packets in the forwarded state do not flow through the device, so those packets are not seen by tcpdump. Will revisit this limitation in a future release.

² Iptables on ingress supports PREROUTING with skb->dev set to the real ingress device and both INPUT and PREROUTING rules with skb->dev set to the VRF device. For egress POSTROUTING and OUTPUT rules can be written using either the VRF device or real egress device.

111.1.3 Applications

Applications that are to work within a VRF need to bind their socket to the VRF device:

```
setsockopt(sd, SOL_SOCKET, SO_BINDTODEVICE, dev, strlen(dev)+1);
```

or to specify the output device using cmsg and IP_PKTINFO.

By default the scope of the port bindings for unbound sockets is limited to the default VRF. That is, it will not be matched by packets arriving on interfaces enslaved to an l3mdev and processes may bind to the same port if they bind to an l3mdev.

TCP & UDP services running in the default VRF context (ie., not bound to any VRF device) can work across all VRF domains by enabling the `tcp_l3mdev_accept` and `udp_l3mdev_accept` sysctl options:

```
sysctl -w net.ipv4.tcp_l3mdev_accept=1
sysctl -w net.ipv4.udp_l3mdev_accept=1
```

These options are disabled by default so that a socket in a VRF is only selected for packets in that VRF. There is a similar option for RAW sockets, which is enabled by default for reasons of backwards compatibility. This is so as to specify the output device with cmsg and IP_PKTINFO, but using a socket not bound to the corresponding VRF. This allows e.g. older ping implementations to be run with specifying the device but without executing it in the VRF. This option can be disabled so that packets received in a VRF context are only handled by a raw socket bound to the VRF, and packets in the default VRF are only handled by a socket not bound to any VRF:

```
sysctl -w net.ipv4.raw_l3mdev_accept=0
```

netfilter rules on the VRF device can be used to limit access to services running in the default VRF context as well.

Using VRF-aware applications (applications which simultaneously create sockets outside and inside VRFs) in conjunction with `net.ipv4.tcp_l3mdev_accept=1` is possible but may lead to problems in some situations. With that sysctl value, it is unspecified which listening socket will be selected to handle connections for VRF traffic; ie. either a socket bound to the VRF or an unbound socket may be used to accept new connections from a VRF. This somewhat unexpected behavior can lead to problems if sockets are configured with extra options (ex. TCP MD5 keys) with the expectation that VRF traffic will exclusively be handled by sockets bound to VRFs, as would be the case with `net.ipv4.tcp_l3mdev_accept=0`. Finally and as a reminder, regardless of which listening socket is selected, established sockets will be created in the VRF based on the ingress interface, as documented earlier.

111.2 Using iproute2 for VRFs

iproute2 supports the vrf keyword as of v4.7. For backwards compatibility this section lists both commands where appropriate -- with the vrf keyword and the older form without it.

1. Create a VRF

To instantiate a VRF device and associate it with a table:

```
$ ip link add dev NAME type vrf table ID
```

As of v4.8 the kernel supports the l3mdev FIB rule where a single rule covers all VRFs. The l3mdev rule is created for IPv4 and IPv6 on first device create.

2. List VRFs

To list VRFs that have been created:

```
$ ip [-d] link show type vrf
NOTE: The -d option is needed to show the table id
```

For example:

```
$ ip -d link show type vrf
11: mgmt: <NOARP,MASTER,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    ↳ mode DEFAULT group default qlen 1000
        link/ether 72:b3:ba:91:e2:24 brd ff:ff:ff:ff:ff:ff promiscuity 0
        vrf table 1 addrgenmode eui64
12: red: <NOARP,MASTER,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    ↳ mode DEFAULT group default qlen 1000
        link/ether b6:6f:6e:f6:da:73 brd ff:ff:ff:ff:ff:ff promiscuity 0
        vrf table 10 addrgenmode eui64
13: blue: <NOARP,MASTER,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    ↳ mode DEFAULT group default qlen 1000
        link/ether 36:62:e8:7d:bb:8c brd ff:ff:ff:ff:ff:ff promiscuity 0
        vrf table 66 addrgenmode eui64
14: green: <NOARP,MASTER,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    ↳ mode DEFAULT group default qlen 1000
        link/ether e6:28:b8:63:70:bb brd ff:ff:ff:ff:ff:ff promiscuity 0
        vrf table 81 addrgenmode eui64
```

Or in brief output:

```
$ ip -br link show type vrf
mgmt      UP          72:b3:ba:91:e2:24 <NOARP,MASTER,UP,LOWER_UP>
red       UP          b6:6f:6e:f6:da:73 <NOARP,MASTER,UP,LOWER_UP>
blue      UP          36:62:e8:7d:bb:8c <NOARP,MASTER,UP,LOWER_UP>
green     UP          e6:28:b8:63:70:bb <NOARP,MASTER,UP,LOWER_UP>
```

3. Assign a Network Interface to a VRF

Network interfaces are assigned to a VRF by enslaving the netdevice to a VRF device:

```
$ ip link set dev NAME master NAME
```

On enslavement connected and local routes are automatically moved to the table associated with the VRF device.

For example:

```
$ ip link set dev eth0 master mgmt
```

4. Show Devices Assigned to a VRF

To show devices that have been assigned to a specific VRF add the master option to the ip command:

```
$ ip link show vrf NAME
$ ip link show master NAME
```

For example:

```
$ ip link show vrf red
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
  ↳ master red state UP mode DEFAULT group default qlen 1000
    link/ether 02:00:00:00:02:02 brd ff:ff:ff:ff:ff:ff
4: eth2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
  ↳ master red state UP mode DEFAULT group default qlen 1000
    link/ether 02:00:00:00:02:03 brd ff:ff:ff:ff:ff:ff
7: eth5: <BROADCAST,MULTICAST> mtu 1500 qdisc noop master red state DOWN
  ↳ mode DEFAULT group default qlen 1000
    link/ether 02:00:00:00:02:06 brd ff:ff:ff:ff:ff:ff
```

Or using the brief output:

```
$ ip -br link show vrf red
eth1           UP      02:00:00:00:02:02 <BROADCAST,MULTICAST,UP,
  ↳ LOWER_UP>
eth2           UP      02:00:00:00:02:03 <BROADCAST,MULTICAST,UP,
  ↳ LOWER_UP>
eth5          DOWN    02:00:00:00:02:06 <BROADCAST,MULTICAST>
```

5. Show Neighbor Entries for a VRF

To list neighbor entries associated with devices enslaved to a VRF device add the master option to the ip command:

```
$ ip [-6] neigh show vrf NAME
$ ip [-6] neigh show master NAME
```

For example:

```
$ ip neigh show vrf red
10.2.1.254 dev eth1 lladdr a6:d9:c7:4f:06:23 REACHABLE
10.2.2.254 dev eth2 lladdr 5e:54:01:6a:ee:80 REACHABLE
```

```
$ ip -6 neigh show vrf red  
2002:1::64 dev eth1 lladdr a6:d9:c7:4f:06:23 REACHABLE
```

6. Show Addresses for a VRF

To show addresses for interfaces associated with a VRF add the master option to the ip command:

```
$ ip addr show vrf NAME  
$ ip addr show master NAME
```

For example:

```
$ ip addr show vrf red  
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast  
    ↳master red state UP group default qlen 1000  
        link/ether 02:00:00:00:02:02 brd ff:ff:ff:ff:ff:ff  
        inet 10.2.1.2/24 brd 10.2.1.255 scope global eth1  
            valid_lft forever preferred_lft forever  
        inet6 2002:1::2/120 scope global  
            valid_lft forever preferred_lft forever  
        inet6 fe80::ff:fe00:202/64 scope link  
            valid_lft forever preferred_lft forever  
4: eth2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast  
    ↳master red state UP group default qlen 1000  
        link/ether 02:00:00:00:02:03 brd ff:ff:ff:ff:ff:ff  
        inet 10.2.2.2/24 brd 10.2.2.255 scope global eth2  
            valid_lft forever preferred_lft forever  
        inet6 2002:2::2/120 scope global  
            valid_lft forever preferred_lft forever  
        inet6 fe80::ff:fe00:203/64 scope link  
            valid_lft forever preferred_lft forever  
7: eth5: <BROADCAST,MULTICAST> mtu 1500 qdisc noop master red state DOWN  
    ↳group default qlen 1000  
        link/ether 02:00:00:00:02:06 brd ff:ff:ff:ff:ff:ff
```

Or in brief format:

```
$ ip -br addr show vrf red  
eth1           UP      10.2.1.2/24 2002:1::2/120  
  ↳fe80::ff:fe00:202/64  
eth2           UP      10.2.2.2/24 2002:2::2/120  
  ↳fe80::ff:fe00:203/64  
eth5          DOWN
```

7. Show Routes for a VRF

To show routes for a VRF use the ip command to display the table associated with the VRF device:

```
$ ip [-6] route show vrf NAME  
$ ip [-6] route show table ID
```

For example:

```
$ ip route show vrf red
unreachable default metric 4278198272
broadcast 10.2.1.0 dev eth1 proto kernel scope link src 10.2.1.2
10.2.1.0/24 dev eth1 proto kernel scope link src 10.2.1.2
local 10.2.1.2 dev eth1 proto kernel scope host src 10.2.1.2
broadcast 10.2.1.255 dev eth1 proto kernel scope link src 10.2.1.2
broadcast 10.2.2.0 dev eth2 proto kernel scope link src 10.2.2.2
10.2.2.0/24 dev eth2 proto kernel scope link src 10.2.2.2
local 10.2.2.2 dev eth2 proto kernel scope host src 10.2.2.2
broadcast 10.2.2.255 dev eth2 proto kernel scope link src 10.2.2.2

$ ip -6 route show vrf red
local 2002:1:: dev lo proto none metric 0 pref medium
local 2002:1::2 dev lo proto none metric 0 pref medium
2002:1::/120 dev eth1 proto kernel metric 256 pref medium
local 2002:2:: dev lo proto none metric 0 pref medium
local 2002:2::2 dev lo proto none metric 0 pref medium
2002:2::/120 dev eth2 proto kernel metric 256 pref medium
local fe80:: dev lo proto none metric 0 pref medium
local fe80:: dev lo proto none metric 0 pref medium
local fe80::ff:fe00:202 dev lo proto none metric 0 pref medium
local fe80::ff:fe00:203 dev lo proto none metric 0 pref medium
fe80::/64 dev eth1 proto kernel metric 256 pref medium
fe80::/64 dev eth2 proto kernel metric 256 pref medium
ff00::/8 dev red metric 256 pref medium
ff00::/8 dev eth1 metric 256 pref medium
ff00::/8 dev eth2 metric 256 pref medium
unreachable default dev lo metric 4278198272 error -101 pref medium
```

8. Route Lookup for a VRF

A test route lookup can be done for a VRF:

```
$ ip [-6] route get vrf NAME ADDRESS
$ ip [-6] route get oif NAME ADDRESS
```

For example:

```
$ ip route get 10.2.1.40 vrf red
10.2.1.40 dev eth1 table red src 10.2.1.2
cache

$ ip -6 route get 2002:1::32 vrf red
2002:1::32 from :: dev eth1 table red proto kernel src 2002:1::2 ↴
metric 256 pref medium
```

9. Removing Network Interface from a VRF

Network interfaces are removed from a VRF by breaking the enslavement to the VRF device:

```
$ ip link set dev NAME nomaster
```

Connected routes are moved back to the default table and local entries are moved to the local table.

For example:

```
$ ip link set dev eth0 nomaster
```

Commands used in this example:

```
cat >> /etc/iproute2/rt_tables.d/vrf.conf <<EOF
1 mgmt
10 red
66 blue
81 green
EOF

function vrf_create
{
    VRF=$1
    TBID=$2

    # create VRF device
    ip link add ${VRF} type vrf table ${TBID}

    if [ "${VRF}" != "mgmt" ]; then
        ip route add table ${TBID} unreachable default metric 4278198272
    fi
    ip link set dev ${VRF} up
}

vrf_create mgmt 1
ip link set dev eth0 master mgmt

vrf_create red 10
ip link set dev eth1 master red
ip link set dev eth2 master red
ip link set dev eth5 master red

vrf_create blue 66
ip link set dev eth3 master blue

vrf_create green 81
ip link set dev eth4 master green
```

Interface addresses from /etc/network/interfaces:

```
auto eth0
iface eth0 inet static
```

```
address 10.0.0.2
netmask 255.255.255.0
gateway 10.0.0.254

iface eth0 inet6 static
    address 2000:1::2
    netmask 120

auto eth1
iface eth1 inet static
    address 10.2.1.2
    netmask 255.255.255.0

iface eth1 inet6 static
    address 2002:1::2
    netmask 120

auto eth2
iface eth2 inet static
    address 10.2.2.2
    netmask 255.255.255.0

iface eth2 inet6 static
    address 2002:2::2
    netmask 120

auto eth3
iface eth3 inet static
    address 10.2.3.2
    netmask 255.255.255.0

iface eth3 inet6 static
    address 2002:3::2
    netmask 120

auto eth4
iface eth4 inet static
    address 10.2.4.2
    netmask 255.255.255.0

iface eth4 inet6 static
    address 2002:4::2
    netmask 120
```


VIRTUAL EXTENSIBLE LOCAL AREA NETWORKING DOCUMENTATION

The VXLAN protocol is a tunnelling protocol designed to solve the problem of limited VLAN IDs (4096) in IEEE 802.1q. With VXLAN the size of the identifier is expanded to 24 bits (16777216).

VXLAN is described by IETF RFC 7348, and has been implemented by a number of vendors. The protocol runs over UDP using a single destination port. This document describes the Linux kernel tunnel device, there is also a separate implementation of VXLAN for Openvswitch.

Unlike most tunnels, a VXLAN is a 1 to N network, not just point to point. A VXLAN device can learn the IP address of the other endpoint either dynamically in a manner similar to a learning bridge, or make use of statically-configured forwarding entries.

The management of vxlan is done in a manner similar to its two closest neighbors GRE and VLAN. Configuring VXLAN requires the version of iproute2 that matches the kernel release where VXLAN was first merged upstream.

1. Create vxlan device:

```
# ip link add vxlan0 type vxlan id 42 group 239.1.1.1 dev eth1 dstport 4789
```

This creates a new device named vxlan0. The device uses the multicast group 239.1.1.1 over eth1 to handle traffic for which there is no entry in the forwarding table. The destination port number is set to the IANA-assigned value of 4789. The Linux implementation of VXLAN predates the IANA's selection of a standard destination port number and uses the Linux-selected value by default to maintain backwards compatibility.

2. Delete vxlan device:

```
# ip link delete vxlan0
```

3. Show vxlan info:

```
# ip -d link show vxlan0
```

It is possible to create, destroy and display the vxlan forwarding table using the new bridge command.

1. Create forwarding table entry:

```
# bridge fdb add to 00:17:42:8a:b4:05 dst 192.19.0.2 dev vxlan0
```

2. Delete forwarding table entry:

```
# bridge fdb delete 00:17:42:8a:b4:05 dev vxlan0
```

3. Show forwarding table:

```
# bridge fdb show dev vxlan0
```

The following NIC features may indicate support for UDP tunnel-related offloads (most commonly VXLAN features, but support for a particular encapsulation protocol is NIC specific):

- *tx-udp_tnl-segmentation*
- ***tx-udp_tnl-csum-segmentation***
ability to perform TCP segmentation offload of UDP encapsulated frames
- ***rx-udp_tunnel-port-offload***
receive side parsing of UDP encapsulated frames which allows NICs to perform protocol-aware offloads, like checksum validation offload of inner frames (only needed by NICs without protocol-agnostic offloads)

For devices supporting *rx-udp_tunnel-port-offload* the list of currently offloaded ports can be interrogated with *ethtool*:

```
$ ethtool --show-tunnels eth0
Tunnel information for eth0:
  UDP port table 0:
    Size: 4
    Types: vxlan
    No entries
  UDP port table 1:
    Size: 4
    Types: geneve, vxlan-gpe
    Entries (1):
      port 1230, vxlan-gpe
```

LINUX X.25 PROJECT

As my third year dissertation at University I have taken it upon myself to write an X.25 implementation for Linux. My aim is to provide a complete X.25 Packet Layer and a LAPB module to allow for "normal" X.25 to be run using Linux. There are two sorts of X.25 cards available, intelligent ones that implement LAPB on the card itself, and unintelligent ones that simply do framing, bit-stuffing and checksumming. These both need to be handled by the system.

I therefore decided to write the implementation such that as far as the Packet Layer is concerned, the link layer was being performed by a lower layer of the Linux kernel and therefore it did not concern itself with implementation of LAPB. Therefore the LAPB modules would be called by unintelligent X.25 card drivers and not by intelligent ones, this would provide a uniform device driver interface, and simplify configuration.

To confuse matters a little, an 802.2 LLC implementation is also possible which could allow X.25 to be run over an Ethernet (or Token Ring) and conform with the JNT "Pink Book", this would have a different interface to the Packet Layer but there would be no confusion since the class of device being served by the LLC would be completely separate from LAPB.

Just when you thought that it could not become more confusing, another option appeared, XOT. This allows X.25 Packet Layer frames to operate over the Internet using TCP/IP as a reliable link layer. RFC1613 specifies the format and behaviour of the protocol. If time permits this option will also be actively considered.

A linux-x25 mailing list has been created at vger.kernel.org to support the development and use of Linux X.25. It is early days yet, but interested parties are welcome to subscribe to it. Just send a message to majordomo@vger.kernel.org with the following in the message body:

subscribe linux-x25 end

The contents of the Subject line are ignored.

Jonathan

g4klx@g4klx.demon.co.uk

X.25 DEVICE DRIVER INTERFACE

Version 1.1

Jonathan Naylor 26.12.96

This is a description of the messages to be passed between the X.25 Packet Layer and the X.25 device driver. They are designed to allow for the easy setting of the LAPB mode from within the Packet Layer.

The X.25 device driver will be coded normally as per the Linux device driver standards. Most X.25 device drivers will be moderately similar to the already existing Ethernet device drivers. However unlike those drivers, the X.25 device driver has a state associated with it, and this information needs to be passed to and from the Packet Layer for proper operation.

All messages are held in `sk_buff`'s just like real data to be transmitted over the LAPB link. The first byte of the `skbuff` indicates the meaning of the rest of the `skbuff`, if any more information does exist.

114.1 Packet Layer to Device Driver

First Byte = 0x00 (`X25_IFACE_DATA`)

This indicates that the rest of the `skbuff` contains data to be transmitted over the LAPB link. The LAPB link should already exist before any data is passed down.

First Byte = 0x01 (`X25_IFACE_CONNECT`)

Establish the LAPB link. If the link is already established then the connect confirmation message should be returned as soon as possible.

First Byte = 0x02 (`X25_IFACE_DISCONNECT`)

Terminate the LAPB link. If it is already disconnected then the disconnect confirmation message should be returned as soon as possible.

First Byte = 0x03 (`X25_IFACE_PARAMS`)

LAPB parameters. To be defined.

114.2 Device Driver to Packet Layer

First Byte = 0x00 (X25_IFACE_DATA)

This indicates that the rest of the skbuff contains data that has been received over the LAPB link.

First Byte = 0x01 (X25_IFACE_CONNECT)

LAPB link has been established. The same message is used for both a LAPB link connect_confirmation and a connect_indication.

First Byte = 0x02 (X25_IFACE_DISCONNECT)

LAPB link has been terminated. This same message is used for both a LAPB link disconnect_confirmation and a disconnect_indication.

First Byte = 0x03 (X25_IFACE_PARAMS)

LAPB parameters. To be defined.

114.3 Requirements for the device driver

Packets should not be reordered or dropped when delivering between the Packet Layer and the device driver.

To avoid packets from being reordered or dropped when delivering from the device driver to the Packet Layer, the device driver should not call "netif_rx" to deliver the received packets. Instead, it should call "netif_receive_skb_core" from softirq context to deliver them.

XFRM DEVICE - OFFLOADING THE IPSEC COMPUTATIONS

Shannon Nelson <shannon.nelson@oracle.com> Leon Romanovsky <leonro@nvidia.com>

115.1 Overview

IPsec is a useful feature for securing network traffic, but the computational cost is high: a 10Gbps link can easily be brought down to under 1Gbps, depending on the traffic and link configuration. Luckily, there are NICs that offer a hardware based IPsec offload which can radically increase throughput and decrease CPU utilization. The XFRM Device interface allows NIC drivers to offer to the stack access to the hardware offload.

Right now, there are two types of hardware offload that kernel supports.

- IPsec crypto offload: * NIC performs encrypt/decrypt * Kernel does everything else
- IPsec packet offload: * NIC performs encrypt/decrypt * NIC does encapsulation * Kernel and NIC have SA and policy in-sync * NIC handles the SA and policies states * The Kernel talks to the keymanager

Userland access to the offload is typically through a system such as libreswan or KAME/raccoon, but the iproute2 'ip xfrm' command set can be handy when experimenting. An example command might look something like this for crypto offload:

```
ip x s add proto esp dst 14.0.0.70 src 14.0.0.52 spi 0x07 mode transport
    reqid      0x07      replay-window      32      aead      'rfc4106(gcm(aes))'
    0x44434241343332312423222114131211f4f3f2f1 128 sel src 14.0.0.52/24
    dst 14.0.0.70/24 proto tcp offload dev eth4 dir in
```

and for packet offload

```
ip x s add proto esp dst 14.0.0.70 src 14.0.0.52 spi 0x07 mode transport
    reqid      0x07      replay-window      32      aead      'rfc4106(gcm(aes))'
    0x44434241343332312423222114131211f4f3f2f1 128 sel src 14.0.0.52/24
    dst 14.0.0.70/24 proto tcp offload packet dev eth4 dir in
```

```
ip x p add src 14.0.0.70 dst 14.0.0.52 offload packet dev eth4 dir in tmpl src 14.0.0.70
dst 14.0.0.52 proto esp reqid 10000 mode transport
```

Yes, that's ugly, but that's what shell scripts and/or libreswan are for.

115.2 Callbacks to implement

```
/* from include/linux/netdevice.h */
struct xfrmdev_ops {
    /* Crypto and Packet offload callbacks */
    int (*ndo_dev_state_add) (struct xfrm_state *x, struct netlink_ext_
->ack *extack);
    void (*ndo_dev_state_delete) (struct xfrm_state *x);
    void (*ndo_dev_state_free) (struct xfrm_state *x);
    bool (*ndo_dev_offload_ok) (struct sk_buff *skb,
                                struct xfrm_state *x);
    void (*ndo_dev_state_advance_esn) (struct xfrm_state *x);

    /* Solely packet offload callbacks */
    void (*ndo_dev_state_update_curlft) (struct xfrm_state *x);
    int (*ndo_dev_policy_add) (struct xfrm_policy *x, struct netlink_ext_
->ack *extack);
    void (*ndo_dev_policy_delete) (struct xfrm_policy *x);
    void (*ndo_dev_policy_free) (struct xfrm_policy *x);
};
```

The NIC driver offering ipsec offload will need to implement callbacks relevant to supported offload to make the offload available to the network stack's XFRM subsystem. Additionally, the feature bits NETIF_F_HW_ESP and NETIF_F_HW_ESP_TX_CSUM will signal the availability of the offload.

115.3 Flow

At probe time and before the call to `register_netdev()`, the driver should set up local data structures and XFRM callbacks, and set the feature bits. The XFRM code's listener will finish the setup on NETDEV_REGISTER.

```
adapter->netdev->xfrmdev_ops = &ixgbe_xfrmdev_ops;
adapter->netdev->features |= NETIF_F_HW_ESP;
adapter->netdev->hw_enc_features |= NETIF_F_HW_ESP;
```

When new SAs are set up with a request for "offload" feature, the driver's `xdo_dev_state_add()` will be given the new SA to be offloaded and an indication of whether it is for Rx or Tx. The driver should

- verify the algorithm is supported for offloads
- store the SA information (key, salt, target-ip, protocol, etc)
- enable the HW offload of the SA
- return status value:

0	success
-EOPNETSUPP	offload not supported, try SW IPsec, not applicable for packet offload mode
other	fail the request

The driver can also set an offload_handle in the SA, an opaque void pointer that can be used to convey context into the fast-path offload requests:

```
xs->xso.offload_handle = context;
```

When the network stack is preparing an IPsec packet for an SA that has been setup for offload, it first calls into xdo_dev_offload_ok() with the skb and the intended offload state to ask the driver if the offload will serviceable. This can check the packet information to be sure the offload can be supported (e.g. IPv4 or IPv6, no IPv4 options, etc) and return true or false to signify its support.

Crypto offload mode: When ready to send, the driver needs to inspect the Tx packet for the offload information, including the opaque context, and set up the packet send accordingly:

```
xs = xfrm_input_state(skb);
context = xs->xso.offload_handle;
set up HW for send
```

The stack has already inserted the appropriate IPsec headers in the packet data, the offload just needs to do the encryption and fix up the header values.

When a packet is received and the HW has indicated that it offloaded a decryption, the driver needs to add a reference to the decoded SA into the packet's skb. At this point the data should be decrypted but the IPsec headers are still in the packet data; they are removed later up the stack in xfrm_input().

find and hold the SA that was used to the Rx skb:

```
get spi, protocol, and destination IP from packet headers
xs = find xs from (spi, protocol, dest_IP)
xfrm_state_hold(xs);
```

store the state information into the skb:

```
sp = secpAth_set(skb);
if (!sp) return;
sp->xvec[sp->len++] = xs;
sp->olen++;
```

indicate the success and/or error status of the offload:

```
xo = xfrm_offload(skb);
xo->flags = CRYPTO_DONE;
xo->status = crypto_status;
```

hand the packet to napi_gro_receive() as usual

In ESN mode, xdo_dev_state_advance_esn() is called from xfrm_replay_advance_esn(). Driver will check packet seq number and update HW ESN state machine if needed.

Packet offload mode: HW adds and deletes XFRM headers. So in RX path, XFRM stack is bypassed if HW reported success. In TX path, the packet leaves kernel without extra header and not encrypted, the HW is responsible to perform it.

When the SA is removed by the user, the driver's `xdo_dev_state_delete()` and `xdo_dev_policy_delete()` are asked to disable the offload. Later, `xdo_dev_state_free()` and `xdo_dev_policy_free()` are called from a garbage collection routine after all reference counts to the state and policy have been removed and any remaining resources can be cleared for the offload state. How these are used by the driver will depend on specific hardware needs.

As a netdev is set to DOWN the XFRM stack's netdev listener will call `xdo_dev_state_delete()`, `xdo_dev_policy_delete()`, `xdo_dev_state_free()` and `xdo_dev_policy_free()` on any remaining offloaded states.

Outcome of HW handling packets, the XFRM core can't count hard, soft limits. The HW/driver are responsible to perform it and provide accurate data when `xdo_dev_state_update_curlft()` is called. In case of one of these limits occurred, the driver needs to call to `xfrm_state_check_expire()` to make sure that XFRM performs rekeying sequence.

XFRM PROC - /PROC/NET/XFRM_* FILES

Masahide NAKAMURA <nakam@linux-ipv6.org>

116.1 Transformation Statistics

The xfrm_proc code is a set of statistics showing numbers of packets dropped by the transformation code and why. These counters are defined as part of the linux private MIB. These counters can be viewed in /proc/net/xfrm_stat.

116.1.1 Inbound errors

XfrmInError:

All errors which is not matched others

XfrmInBufferError:

No buffer is left

XfrmInHdrError:

Header error

XfrmInNoStates:

No state is found i.e. Either inbound SPI, address, or IPsec protocol at SA is wrong

XfrmInStateProtoError:

Transformation protocol specific error e.g. SA key is wrong

XfrmInStateModelError:

Transformation mode specific error

XfrmInStateSeqError:

Sequence error i.e. Sequence number is out of window

XfrmInStateExpired:

State is expired

XfrmInStateMismatch:

State has mismatch option e.g. UDP encapsulation type is mismatch

XfrmInStateInvalid:

State is invalid

XfrmInTmplMismatch:

No matching template for states e.g. Inbound SAs are correct but SP rule is wrong

XfrmInNoPols:

No policy is found for states e.g. Inbound SAs are correct but no SP is found

XfrmInPolBlock:

Policy discards

XfrmInPolError:

Policy error

XfrmAcquireError:

State hasn't been fully acquired before use

XfrmFwdHdrError:

Forward routing of a packet is not allowed

116.1.2 Outbound errors

XfrmOutError:

All errors which is not matched others

XfrmOutBundleGenError:

Bundle generation error

XfrmOutBundleCheckError:

Bundle check error

XfrmOutNoStates:

No state is found

XfrmOutStateProtoError:

Transformation protocol specific error

XfrmOutStateModeError:

Transformation mode specific error

XfrmOutStateSeqError:

Sequence error i.e. Sequence number overflow

XfrmOutStateExpired:

State is expired

XfrmOutPolBlock:

Policy discards

XfrmOutPolDead:

Policy is dead

XfrmOutPolError:

Policy error

XfrmOutStateInvalid:

State is invalid, perhaps expired

The sync patches work is based on initial patches from Krisztian <hidden@balabit.hu> and others and additional patches from Jamal <hadi@cyberus.ca>.

The end goal for syncing is to be able to insert attributes + generate events so that the SA can be safely moved from one machine to another for HA purposes. The idea is to synchronize the SA so that the takeover machine can do the processing of the SA as accurate as possible if it has access to it.

We already have the ability to generate SA add/del/upd events. These patches add ability to sync and have accurate lifetime byte (to ensure proper decay of SAs) and replay counters to avoid replay attacks with as minimal loss at failover time. This way a backup stays as closely up-to-date as an active member.

Because the above items change for every packet the SA receives, it is possible for a lot of the events to be generated. For this reason, we also add a nagle-like algorithm to restrict the events. i.e we are going to set thresholds to say "let me know if the replay sequence threshold is reached or 10 secs have passed" These thresholds are set system-wide via sysctls or can be updated per SA.

The identified items that need to be synchronized are: - the lifetime byte counter note that: lifetime time limit is not important if you assume the failover machine is known ahead of time since the decay of the time countdown is not driven by packet arrival. - the replay sequence for both inbound and outbound

117.1 1) Message Structure

nlmsghdr:aevent_id:optional-TLVs.

The netlink message types are:

XFRM_MSG_NEWAE and XFRM_MSG_GETAE.

A XFRM_MSG_GETAE does not have TLVs.

A XFRM_MSG_NEWAE will have at least two TLVs (as is discussed further below).

aevent_id structure looks like:

```
struct xfrm_aevent_id {
    struct xfrm_usersa_id           sa_id;
    xfrm_address_t                  saddr;
    __u32                           flags;
```

```
};                                __u32 reqid;
```

The unique SA is identified by the combination of xfrm_usersa_id, reqid and saddr. flags are used to indicate different things. The possible flags are:

```
XFRM_AE_RTHR=1, /* replay threshold*/
XFRM_AE_RVAL=2, /* replay value */
XFRM_AE_LVAL=4, /* lifetime value */
XFRM_AE_ETHR=8, /* expiry timer threshold */
XFRM_AE_CR=16, /* Event cause is replay update */
XFRM_AE_CE=32, /* Event cause is timer expiry */
XFRM_AE_CU=64, /* Event cause is policy update */
```

How these flags are used is dependent on the direction of the message (kernel<->user) as well the cause (config, query or event). This is described below in the different messages.

The pid will be set appropriately in netlink to recognize direction (0 to the kernel and pid = processid that created the event when going from kernel to user space)

A program needs to subscribe to multicast group XFRMNLGRP_AEVENTS to get notified of these events.

117.2 2) TLVS reflect the different parameters:

a) byte value (XFRM_AE_LTVAL)

This TLV carries the running/current counter for byte lifetime since last event.

b) replay value (XFRM_AE_RVAL)

This TLV carries the running/current counter for replay sequence since last event.

c) replay threshold (XFRM_AE_RTHR)

This TLV carries the threshold being used by the kernel to trigger events when the replay sequence is exceeded.

d) expiry timer (XFRM_AE_ETIMER_THRESH)

This is a timer value in milliseconds which is used as the nagle value to rate limit the events.

117.3 3) Default configurations for the parameters:

By default these events should be turned off unless there is at least one listener registered to listen to the multicast group XFRMNLGRP_AEVENTS.

Programs installing SAs will need to specify the two thresholds, however, in order to not change existing applications such as racoon we also provide default threshold values for these different parameters in case they are not specified.

the two sysctls/proc entries are:

- a) /proc/sys/net/core/sysctl_xfrm_aevent_etime used to provide default values for the XFRMA_ETIMER_THRESH in incremental units of time of 100ms. The default is 10 (1 second)
- b) /proc/sys/net/core/sysctl_xfrm_aevent_rseqth used to provide default values for XFRMA_REPLY_THRESH parameter in incremental packet count. The default is two packets.

117.4 4) Message types

- a) XFRM_MSG_GETAE issued by user-->kernel. XFRM_MSG_GETAE does not carry any TLVs.

The response is a XFRM_MSG_NEWAE which is formatted based on what XFRM_MSG_GETAE queried for.

The response will always have XFRMA_LTIME_VAL and XFRMA_REPLY_VAL TLVs. * if XFRM_AE_RTHR flag is set, then XFRMA_REPLY_THRESH is also retrieved * if XFRM_AE_ETHR flag is set, then XFRMA_ETIMER_THRESH is also retrieved

- b) XFRM_MSG_NEWAE is issued by either user space to configure or kernel to announce events or respond to a XFRM_MSG_GETAE.

- i) user --> kernel to configure a specific SA.

any of the values or threshold parameters can be updated by passing the appropriate TLV.

A response is issued back to the sender in user space to indicate success or failure.

In the case of success, additionally an event with XFRM_MSG_NEWAE is also issued to any listeners as described in iii).

- ii) kernel->user direction as a response to XFRM_MSG_GETAE

The response will always have XFRMA_LTIME_VAL and XFRMA_REPLY_VAL TLVs.

The threshold TLVs will be included if explicitly requested in the XFRM_MSG_GETAE message.

- iii) kernel->user to report as event if someone sets any values or thresholds for an SA using XFRM_MSG_NEWAE (as described in #i above). In such a case XFRM_AE_CU flag is set to inform the user that the change happened as a result of an update. The message will always have XFRMA_LTIME_VAL and XFRMA_REPLY_VAL TLVs.

- iv) kernel->user to report event when replay threshold or a timeout is exceeded.

In such a case either XFRM_AE_CR (replay exceeded) or XFRM_AE_CE (timeout happened) is set to inform the user what happened. Note the two flags are mutually exclusive. The message will always have XFRMA_LTIME_VAL and XFRMA_REPLY_VAL TLVs.

117.5 Exceptions to threshold settings

If you have an SA that is getting hit by traffic in bursts such that there is a period where the timer threshold expires with no packets seen, then an odd behavior is seen as follows: The first packet arrival after a timer expiry will trigger a timeout event; i.e we don't wait for a timeout period or a packet threshold to be reached. This is done for simplicity and efficiency reasons.

-JHS

XFRM SYSCALL

118.1 /proc/sys/net/core/xfrm_* Variables:

xfrm_acq_expires - INTEGER

default 30 - hard timeout in seconds for acquire requests

XDP RX METADATA

This document describes how an eXpress Data Path (XDP) program can access hardware metadata related to a packet using a set of helper functions, and how it can pass that metadata on to other consumers.

119.1 General Design

XDP has access to a set of kfuncs to manipulate the metadata in an XDP frame. Every device driver that wishes to expose additional packet metadata can implement these kfuncs. The set of kfuncs is declared in `include/net/xdp.h` via `XDP_METADATA_KFUNC_xxx`.

Currently, the following kfuncs are supported. In the future, as more metadata is supported, this set will grow:

`_bpf_kfunc int bpf_xdp_metadata_rx_timestamp(const struct xdp_md *ctx, u64 *timestamp)`
Read XDP frame RX timestamp.

Parameters

const struct xdp_md *ctx
XDP context pointer.

u64 *timestamp
Return value pointer.

Return

- Returns 0 on success or `-errno` on error.
- `-EOPNOTSUPP` : means device driver does not implement kfunc
- `-ENODATA` : means no RX-timestamp available for this frame

`_bpf_kfunc int bpf_xdp_metadata_rx_hash(const struct xdp_md *ctx, u32 *hash, enum xdp_rss_hash_type *rss_type)`

Read XDP frame RX hash.

Parameters

const struct xdp_md *ctx
XDP context pointer.

u32 *hash
Return value pointer.

enum xdp_rss_hash_type *rss_type
Return value pointer for RSS type.

Description

The RSS hash type (**rss_type**) specifies what portion of packet headers NIC hardware used when calculating RSS hash value. The RSS type can be decoded via enum `xdp_rss_hash_type` either matching on individual L3/L4 bits `XDP_RSS_L*` or by combined traditional *RSS Hashing Types* `XDP_RSS_TYPE_L*`.

Return

- Returns 0 on success or -errno on error.
- -EOPNOTSUPP : means device driver doesn't implement kfunc
- -ENODATA : means no RX-hash available for this frame

`_bpf_kfunc int bpf_xdp_metadata_rx_vlan_tag(const struct xdp_md *ctx, __be16 *vlan_proto, u16 *vlan_tci)`

Get XDP packet outermost VLAN tag

Parameters

const struct xdp_md *ctx
XDP context pointer.

__be16 *vlan_proto
Destination pointer for VLAN Tag protocol identifier (TPID).

u16 *vlan_tci
Destination pointer for VLAN TCI (VID + DEI + PCP)

Description

In case of success, `vlan_proto` contains *Tag protocol identifier (TPID)*, usually `ETH_P_8021Q` or `ETH_P_8021AD`, but some networks can use custom TPIDs. `vlan_proto` is stored in **network byte order (BE)** and should be used as follows: `if (vlan_proto == bpf_htons(ETH_P_8021Q)) do_something();`

`vlan_tci` contains the remaining 16 bits of a VLAN tag. Driver is expected to provide those in **host byte order (usually LE)**, so the bpf program should not perform byte conversion. According to 802.1Q standard, *VLAN TCI (Tag control information)* is a bit field that contains: *VLAN identifier (VID)* that can be read with `vlan_tci & 0xffff`, *Drop eligible indicator (DEI)* - 1 bit, *Priority code point (PCP)* - 3 bits. For detailed meaning of DEI and PCP, please refer to other sources.

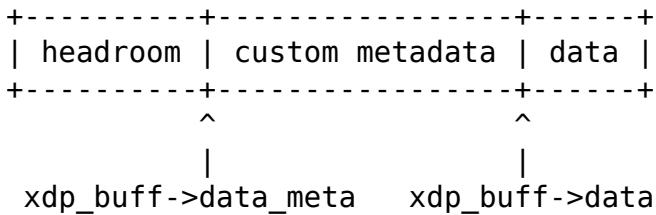
Return

- Returns 0 on success or -errno on error.
- -EOPNOTSUPP : device driver doesn't implement kfunc
- -ENODATA : VLAN tag was not stripped or is not available

An XDP program can use these kfuncs to read the metadata into stack variables for its own consumption. Or, to pass the metadata on to other consumers, an XDP program can store it into the metadata area carried ahead of the packet. Not all packets will necessarily have the requested metadata available in which case the driver returns -ENODATA.

Not all kfuncs have to be implemented by the device driver; when not implemented, the default ones that return -EOPNOTSUPP will be used to indicate the device driver have not implemented this kfunc.

Within an XDP frame, the metadata layout (accessed via `xdp_buff`) is as follows:

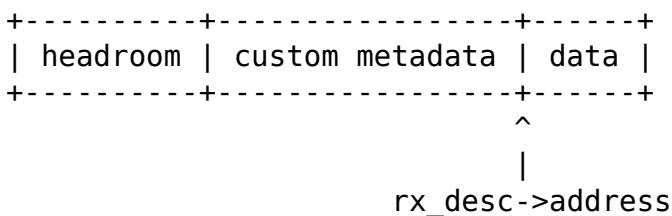


An XDP program can store individual metadata items into this `data_meta` area in whichever format it chooses. Later consumers of the metadata will have to agree on the format by some out of band contract (like for the AF_XDP use case, see below).

119.2 AF_XDP

AF_XDP use-case implies that there is a contract between the BPF program that redirects XDP frames into the AF_XDP socket (XSK) and the final consumer. Thus the BPF program manually allocates a fixed number of bytes out of metadata via `bpf_xdp_adjust_meta` and calls a subset of kfuncs to populate it. The userspace XSK consumer computes `xsk_umem_get_data()` - `METADATA_SIZE` to locate that metadata. Note, `xsk_umem_get_data` is defined in `libxdp` and `METADATA_SIZE` is an application-specific constant (AF_XDP receive descriptor does *not* explicitly carry the size of the metadata).

Here is the AF_XDP consumer layout (note missing `data_meta` pointer):



119.3 XDP_PASS

This is the path where the packets processed by the XDP program are passed into the kernel. The kernel creates the `skb` out of the `xdp_buff` contents. Currently, every driver has custom kernel code to parse the descriptors and populate `skb` metadata when doing this `xdp_buff->skb` conversion, and the XDP metadata is not used by the kernel when building `skbs`. However, TC-BPF programs can access the XDP metadata area using the `data_meta` pointer.

In the future, we'd like to support a case where an XDP program can override some of the metadata used for building `skbs`.

119.4 bpf_redirect_map

`bpf_redirect_map` can redirect the frame to a different device. Some devices (like virtual ethernet links) support running a second XDP program after the redirect. However, the final consumer doesn't have access to the original hardware descriptor and can't access any of the original metadata. The same applies to XDP programs installed into devmaps and cpumaps.

This means that for redirected packets only custom metadata is currently supported, which has to be prepared by the initial XDP program before redirect. If the frame is eventually passed to the kernel, the `skb` created from such a frame won't have any hardware metadata populated in its `skb`. If such a packet is later redirected into an XSK, that will also only have access to the custom metadata.

119.5 bpf_tail_call

Adding programs that access metadata kfuncs to the `BPF_MAP_TYPE_PROG_ARRAY` is currently not supported.

119.6 Supported Devices

It is possible to query which kfunc the particular netdev implements via netlink. See `xdp-rx-metadata-features` attribute set in `Documentation/netlink/specs/netdev.yaml`.

119.7 Example

See `tools/testing/selftests/bpf/progs/xdp_metadata.c` and `tools/testing/selftests/bpf/prog_tests/xdp_metadata.c` for an example of BPF program that handles XDP metadata.

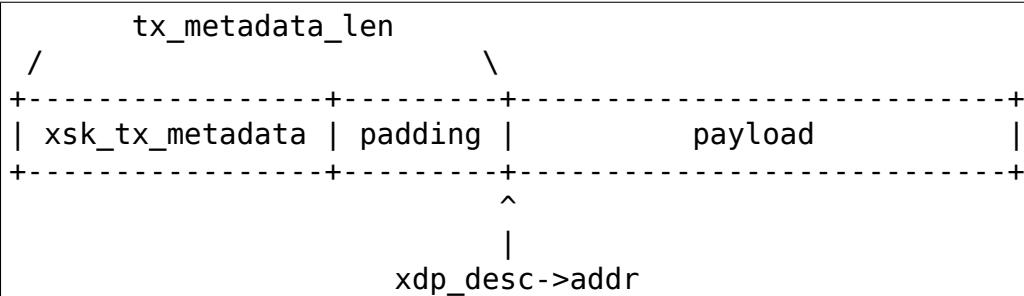
AF_XDP TX METADATA

This document describes how to enable offloads when transmitting packets via [AF_XDP](#). Refer to [XDP RX Metadata](#) on how to access similar metadata on the receive side.

120.1 General Design

The headroom for the metadata is reserved via `tx_metadata_len` in `struct xdp_umem_reg`. The metadata length is therefore the same for every socket that shares the same umem. The metadata layout is a fixed UAPI, refer to `union xsk_tx_metadata` in `include/uapi/linux/if_xdp.h`. Thus, generally, the `tx_metadata_len` field above should contain `sizeof(union xsk_tx_metadata)`.

The headroom and the metadata itself should be located right before `xdp_desc->addr` in the umem frame. Within a frame, the metadata layout is as follows:



An AF_XDP application can request headrooms larger than `sizeof(struct xsk_tx_metadata)`. The kernel will ignore the padding (and will still use `xdp_desc->addr - tx_metadata_len` to locate the `xsk_tx_metadata`). For the frames that shouldn't carry any metadata (i.e., the ones that don't have `XDP_TX_METADATA` option), the metadata area is ignored by the kernel as well.

The flags field enables the particular offload:

- `XDP_TXMD_FLAGS_TIMESTAMP`: requests the device to put transmission timestamp into `tx_timestamp` field of `union xsk_tx_metadata`.
- `XDP_TXMD_FLAGS_CHECKSUM`: requests the device to calculate L4 checksum. `csum_start` specifies byte offset of where the checksumming should start and `csum_offset` specifies byte offset where the device should store the computed checksum.

Besides the flags above, in order to trigger the offloads, the first packet's `struct xdp_desc` descriptor should set `XDP_TX_METADATA` bit in the `options` field. Also note that in a multi-buffer packet only the first chunk should carry the metadata.

120.2 Software TX Checksum

For development and testing purposes its possible to pass `XDP_UMEM_TX_SW_CSUM` flag to `XDP_UMEM_REG` UMEM registration call. In this case, when running in `XDK_COPY` mode, the TX checksum is calculated on the CPU. Do not enable this option in production because it will negatively affect performance.

120.3 Querying Device Capabilities

Every devices exports its offloads capabilities via netlink netdev family. Refer to `xsk-flags` features bitmask in Documentation/netlink/specs/netdev.yaml.

- `tx-timestamp`: device supports `XDP_TXMD_FLAGS_TIMESTAMP`
- `tx-checksum`: device supports `XDP_TXMD_FLAGS_CHECKSUM`

See tools/net/ynl/samples/netdev.c on how to query this information.

120.4 Example

See tools/testing/selftests/bpf/xdp_hw_metadata.c for an example program that handles TX metadata. Also see <https://github.com/fomichev/xskgen> for a more bare-bones example.

INDEX

Symbols

`_alloc_skb (C function), 636`
`_dev_alloc_page (C function), 604`
`_dev_alloc_pages (C function), 603`
`_dev_change_net_namespace (C function), 724`
`_dev_get_by_flags (C function), 702`
`_dev_get_by_index (C function), 701`
`_dev_get_by_name (C function), 700`
`_dev_mc_sync (C function), 764`
`_dev_mc_unsync (C function), 764`
`_dev_queue_xmit (C function), 709`
`_dev_remove_pack (C function), 699`
`_dev_uc_sync (C function), 764`
`_dev_uc_unsync (C function), 764`
`_genphy_config_aneg (C function), 824`
`_kfree_skb (C function), 637`
`_locked_read_sk_user_data_with_flags (C function), 625`
`_mdiobus_c45_modify_changed (C function), 841`
`_mdiobus_c45_read (C function), 835`
`_mdiobus_c45_write (C function), 836`
`_mdiobus_modify_changed (C function), 835`
`_mdiobus_read (C function), 834`
`_mdiobus_register (C function), 833`
`_mdiobus_write (C function), 834`
`_napi_alloc_skb (C function), 637`
`_napi_schedule (C function), 712`
`_napi_schedule_irqoff (C function), 712`
`_netdev_alloc_skb (C function), 637`
`_netdev_notify_peers (C function), 704`
`_netif_napi_del (C function), 755`
`_netif_rx (C function), 710`
`_netif_subqueue_stopped (C function), 759`
`_phy_clear_bits (C function), 814`
`_phy_clear_bits_mmd (C function), 815`
`_phy_hwtstamp_get (C function), 773`
`_phy_hwtstamp_set (C function), 773`
`_phy_modify (C function), 781`
`_phy_modify_changed (C function), 813`

`_phy_modify_mmd (C function), 783`
`_phy_modify_mmd_changed (C function), 782`
`_phy_package_read_mmd (C function), 779`
`_phy_package_write_mmd (C function), 780`
`_phy_read (C function), 812`
`_phy_read_mmd (C function), 778`
`_phy_set_bits (C function), 814`
`_phy_set_bits_mmd (C function), 815`
`_phy_write (C function), 812`
`_phy_write_mmd (C function), 778`
`_pskb_copy_fclone (C function), 640`
`_pskb_pull_tail (C function), 643`
`_rcu_dereference_sk_user_data_with_flags (C function), 625`
`_sk_mem_reclaim (C function), 657`
`_sk_mem_schedule (C function), 656`
`_skb_dequeue (C function), 599`
`_skb_dequeue_tail (C function), 599`
`_skb_fill_page_desc (C function), 600`
`_skb_frag_ref (C function), 605`
`_skb_frag_unref (C function), 606`
`_skb_header_release (C function), 595`
`_skb_pad (C function), 641`
`_skb_peek (C function), 596`
`_skb_put_padto (C function), 608`
`_skb_queue_after (C function), 598`
`_skb_queue_head (C function), 599`
`_skb_queue_head_init (C function), 597`
`_skb_queue_purge_reason (C function), 603`
`_skb_queue_tail (C function), 599`
`_skb_try_recv_datagram (C function), 657`
`_sock_create (C function), 632`
`_tipc_bind (C function), 1608`
`_tipc_node_link_down (C function), 1602`
`_tipc_node_link_up (C function), 1601`
`_xdr_commit_encode (C function), 671`
`_copy_from_pages (C function), 670`
`_phy_start_aneg (C function), 768`
`_sock_tx_timestamp (C function), 628`

A

`alloc_etherdev_mqs` (*C function*), 727
`alloc_netdev_mqs` (*C function*), 722
`alloc_skb` (*C function*), 592
`alloc_skb_fclone` (*C function*), 593
`alloc_skb_for_msg` (*C function*), 638
`alloc_skb_with_frags` (*C function*), 653

B

`bearer_disable` (*C function*), 1562
`bearer_name_validate` (*C function*), 1561
`bpf_prog_create` (*C function*), 661
`bpf_prog_create_from_user` (*C function*), 661
`bpf_xdp_metadata_rx_hash` (*C function*), 1669
`bpf_xdp_metadata_rx_timestamp` (*C function*), 1669
`bpf_xdp_metadata_rx_vlan_tag` (*C function*), 1670
`build_skb_around` (*C function*), 636

C

`call_netdevice_notifiers` (*C function*), 706
`compare_ether_header` (*C function*), 735
`consume_skb` (*C function*), 638
`csum_partial_copy_to_xdr` (*C function*), 689
`ctucan_chip_start` (*C function*), 85
`ctucan_chip_stop` (*C function*), 89
`ctucan_close` (*C function*), 89
`ctucan_do_set_mode` (*C function*), 85
`ctucan_err_interrupt` (*C function*), 88
`ctucan_get_berr_counter` (*C function*), 89
`ctucan_get_rec_tec` (*C function*), 87
`ctucan_get_tx_status` (*C function*), 85
`ctucan_give_txb_cmd` (*C function*), 86
`ctucan_insert_frame` (*C function*), 86
`ctucan_interrupt` (*C function*), 88
`ctucan_is_txt_buf_writable` (*C function*), 85
`ctucan_open` (*C function*), 89
`ctucan_pci_probe` (*C function*), 90
`ctucan_pci_remove` (*C function*), 90
`ctucan_platform_probe` (*C function*), 90
`ctucan_platform_remove` (*C function*), 91
`ctucan_probe_common` (*C function*), 83
`ctucan_read_fault_state` (*C function*), 87
`ctucan_read_rx_frame` (*C function*), 87
`ctucan_reset` (*C function*), 83
`ctucan_rotate_txb_prio` (*C function*), 88
`ctucan_rx` (*C function*), 87

`ctucan_rx_poll` (*C function*), 88
`ctucan_set_bittiming` (*C function*), 84
`ctucan_set_btr` (*C function*), 83
`ctucan_set_data_bittiming` (*C function*), 84
`ctucan_set_mode` (*C function*), 84
`ctucan_set_secondary_sample_point` (*C function*), 84

`ctucan_start_xmit` (*C function*), 86
`ctucan_state_to_str` (*C function*), 83
`ctucan_tx_interrupt` (*C function*), 88

D

`datagram_poll` (*C function*), 660
`dev_add_pack` (*C function*), 699
`dev_alloc_name` (*C function*), 703
`dev_change_flags` (*C function*), 718
`dev_close` (*C function*), 704
`dev_disable_lro` (*C function*), 705
`dev_fetch_sw_netstats` (*C function*), 721
`dev_fill_metadata_dst` (*C function*), 700
`dev_forward_skb` (*C function*), 706
`dev_get_by_index_rcu` (*C function*), 701
`dev_get_by_name_rcu` (*C function*), 700
`dev_get_by_napi_id` (*C function*), 702
`dev_get_flags` (*C function*), 718
`dev_get_iflink` (*C function*), 699
`dev_get_port_parent_id` (*C function*), 719
`dev_get_stats` (*C function*), 721
`dev_get_tstats64` (*C function*), 722
`dev_getbyhwaddr_rcu` (*C function*), 702
`dev_hold` (*C function*), 761
`dev_loopback_xmit` (*C function*), 709
`dev_nit_active` (*C function*), 707
`dev_open` (*C function*), 704
`dev_page_is_reusable` (*C function*), 604
`dev_pre_changeaddr_notify` (*C function*), 719
`dev_put` (*C function*), 761
`dev_remove_pack` (*C function*), 699
`dev_set_alias` (*C function*), 703
`dev_set_allmulti` (*C function*), 718
`dev_set_mac_address` (*C function*), 719
`dev_set_promiscuity` (*C function*), 718
`dev_valid_name` (*C function*), 703
`device_get_ethdev_address` (*C function*), 728
`device_get_mac_address` (*C function*), 728
`device_phy_find_device` (*C function*), 830
`devm_phy_package_join` (*C function*), 823
`dim` (*C struct*), 886
`dim_calc_stats` (*C function*), 889

dim_cq_moder (*C struct*), 884
dim_cq_period_mode (*C enum*), 887
dim_on_top (*C function*), 888
dim_park_on_top (*C function*), 888
dim_park_tired (*C function*), 889
dim_sample (*C struct*), 884
dim_state (*C enum*), 887
dim_stats (*C struct*), 885
dim_stats_state (*C enum*), 887
dim_step_result (*C enum*), 888
dim_tune_state (*C enum*), 887
dim_turn (*C function*), 888
dim_update_sample (*C function*), 889
dim_update_sample_with_comps (*C function*), 889
disc_dupl_alert (*C function*), 1578
distr_item (*C struct*), 1558
drop_reasons_register_subsys (*C function*), 635
drop_reasons_unregister_subsys (*C function*), 636

E

ed (*C function*), 570
eth_addr_add (*C function*), 735
eth_addr_dec (*C function*), 734
eth_addr_inc (*C function*), 734
eth_broadcast_addr (*C function*), 731
eth_commit_mac_addr_change (*C function*), 726
eth_get_headlen (*C function*), 725
eth_header (*C function*), 724
eth_header_cache (*C function*), 726
eth_header_cache_update (*C function*), 726
eth_header_parse (*C function*), 725
eth_header_parse_protocol (*C function*), 726
eth_hw_addr_crc (*C function*), 732
eth_hw_addr_gen (*C function*), 735
eth_hw_addr_inherit (*C function*), 733
eth_hw_addr_random (*C function*), 732
eth_hw_addr_set (*C function*), 732
eth_mac_addr (*C function*), 727
eth_prepare_mac_addr_change (*C function*), 726
eth_proto_is_802_3 (*C function*), 731
eth_random_addr (*C function*), 731
eth_skb_pad (*C function*), 736
eth_type_trans (*C function*), 725
eth_zero_addr (*C function*), 731
ether_addr_copy (*C function*), 732

ether_addr_equal (*C function*), 733
ether_addr_equal_64bits (*C function*), 733
ether_addr_equal_masked (*C function*), 734
ether_addr_equal_unaligned (*C function*), 733
ether_addr_to_u64 (*C function*), 734
ether_setup (*C function*), 727
ethtool_fec_stats (*C struct*), 551
ethtool_mac_stats_src (*C enum*), 544
ethtool_mm_cfg (*C struct*), 563
ethtool_mm_state (*C struct*), 561
ethtool_mm_stats (*C struct*), 562
ethtool_mm_verify_status (*C enum*), 562
ethtool_module_power_mode (*C enum*), 555
ethtool_module_power_mode_policy (*C enum*), 555
ethtool_pause_stats (*C struct*), 544
ethtool_powl_pse_admin_state (*C enum*), 556
ethtool_powl_pse_pw_d_status (*C enum*), 556

F

free_netdev (*C function*), 722
fwnode_get_mac_address (*C function*), 728
fwnode_get_phys_node (*C function*), 830
fwnode_mdio_find_device (*C function*), 829
fwnode_phy_find_device (*C function*), 829

G

gen_estimator_active (*C function*), 668
gen_kill_estimator (*C function*), 668
gen_new_estimator (*C function*), 667
gen_replace_estimator (*C function*), 668
genphy_aneg_done (*C function*), 825
genphy_c37_config_advert (*C function*), 832
genphy_c37_config_aneg (*C function*), 825
genphy_c37_read_status (*C function*), 826
genphy_c45_an_config_aneg (*C function*), 787
genphy_c45_an_disable_aneg (*C function*), 787
genphy_c45_aneg_done (*C function*), 788
genphy_c45_check_and_restart_aneg (*C function*), 787
genphy_c45_config_aneg (*C function*), 790
genphy_c45_eee_is_active (*C function*), 791
genphy_c45_ethtool_get_eee (*C function*), 792
genphy_c45_ethtool_set_eee (*C function*), 792
genphy_c45_fast_retrain (*C function*), 790

genphy_c45_plca_get_cfg (*C function*), 790
 genphy_c45_plca_get_status (*C function*), 791
 genphy_c45_plca_set_cfg (*C function*), 791
 genphy_c45_pma_baset1_read_abilities (*C function*), 789
 genphy_c45_pma_baset1_read_master_slave (*C function*), 788
 genphy_c45_pma_baset1_setup_master_slave (*C function*), 786
 genphy_c45_pma_read_abilities (*C function*), 789
 genphy_c45_pma_read_ext_abilities (*C function*), 789
 genphy_c45_pma_resume (*C function*), 786
 genphy_c45_pma_setup_forced (*C function*), 787
 genphy_c45_pma_suspend (*C function*), 786
 genphy_c45_read_eee_abilities (*C function*), 789
 genphy_c45_read_link (*C function*), 788
 genphy_c45_read_lpa (*C function*), 788
 genphy_c45_read_mdix (*C function*), 789
 genphy_c45_read_pma (*C function*), 789
 genphy_c45_read_status (*C function*), 790
 genphy_c45_restart_aneg (*C function*), 787
 genphy_check_and_restart_aneg (*C function*), 824
 genphy_config_advert (*C function*), 832
 genphy_config_eee_advert (*C function*), 824
 genphy_read_abilities (*C function*), 826
 genphy_read_status (*C function*), 826
 genphy_read_status_fixed (*C function*), 825
 genphy_restart_aneg (*C function*), 824
 genphy_setup_forced (*C function*), 824
 genphy_soft_reset (*C function*), 826
 genphy_update_link (*C function*), 825
 get_phy_c22_id (*C function*), 831
 get_phy_c45_ids (*C function*), 830
 get_phy_device (*C function*), 819
 gnet_estimator (*C struct*), 663
 gnet_stats_basic (*C struct*), 662
 gnet_stats_copy_app (*C function*), 666
 gnet_stats_copy_basic (*C function*), 665
 gnet_stats_copy_basic_hw (*C function*), 665
 gnet_stats_copy_queue (*C function*), 666
 gnet_stats_copy_rate_est (*C function*), 666
 gnet_stats_finish_copy (*C function*), 667
 gnet_stats_queue (*C struct*), 663
 gnet_stats_rate_est (*C struct*), 662
 gnet_stats_rate_est64 (*C struct*), 663
 gnet_stats_start_copy (*C function*), 664
 gnet_stats_start_copy_compat (*C function*), 664
 |
 ieee802154_alloc_device (*C function*), 569
 ieee802154_free_device (*C function*), 569
 ieee802154_register_device (*C function*), 569
 ieee802154_rx_irqsafe (*C function*), 569
 ieee802154_unregister_device (*C function*), 569
 ieee802154_xmit_complete (*C function*), 569
 init_dummy_netdev (*C function*), 721
 is_broadcast_ether_addr (*C function*), 730
 is_etherdev_addr (*C function*), 735
 is_link_local_ether_addr (*C function*), 729
 is_local_ether_addr (*C function*), 730
 is_multicast_ether_addr (*C function*), 730
 is_unicast_ether_addr (*C function*), 730
 is_valid_ether_addr (*C function*), 731
 is_zero_ether_addr (*C function*), 729
 K
 kernel_accept (*C function*), 634
 kernel_bind (*C function*), 633
 kernel_connect (*C function*), 634
 kernel_getpeername (*C function*), 635
 kernel_getsockname (*C function*), 635
 kernel_listen (*C function*), 634
 kernel_recvmsg (*C function*), 631
 kernel_sendmsg (*C function*), 630
 kernel_sendmsg_locked (*C function*), 630
 kernel_sock_ip_overhead (*C function*), 635
 kernel_sock_shutdown (*C function*), 635
 kfree_skb (*C function*), 592
 kfree_skb_reason (*C function*), 638
 L
 link_prepare_wakeup (*C function*), 1586
 link_retransmit_failure (*C function*), 1586
 link_schedule_user (*C function*), 1585
 linkwatch_sync_dev (*C function*), 761
 lock_sock_fast (*C function*), 625
 M
 mac_config (*C function*), 846
 mac_finish (*C function*), 847
 mac_get_caps (*C function*), 845
 mac_link_down (*C function*), 847
 mac_link_up (*C function*), 848
 mac_prepare (*C function*), 845

mac_select_pcs (*C function*), 845
mdio_bus_match (*C function*), 842
mdio_bus_stats (*C struct*), 794
mdio_find_bus (*C function*), 833
mdiobus_alloc (*C function*), 797
mdiobus_alloc_size (*C function*), 832
mdiobus_c45_modify (*C function*), 839
mdiobus_c45_modify_changed (*C function*), 840
mdiobus_c45_read (*C function*), 837
mdiobus_c45_read_nested (*C function*), 837
mdiobus_c45_write (*C function*), 838
mdiobus_c45_write_nested (*C function*), 839
mdiobus_create_device (*C function*), 841
mdiobus_free (*C function*), 834
mdiobus_modify (*C function*), 839
mdiobus_modify_changed (*C function*), 840
mdiobus_read (*C function*), 836
mdiobus_read_nested (*C function*), 836
mdiobus_release (*C function*), 841
mdiobus_scan_c22 (*C function*), 833
mdiobus_scan_c45 (*C function*), 841
mdiobus_write (*C function*), 838
mdiobus_write_nested (*C function*), 837
media_find_id (*C function*), 1560
mii_bus (*C struct*), 795

N

name_table (*C struct*), 1558
named_distribute (*C function*), 1596
named_prepare_buf (*C function*), 1596
napi_build_skb (*C function*), 636
napi_complete_done (*C function*), 737
napi_disable (*C function*), 737
napi_enable (*C function*), 713
napi_if_scheduled_mark_missed (*C function*), 737
napi_is_scheduled (*C function*), 736
napi_schedule (*C function*), 736
napi_schedule_irqoff (*C function*), 736
napi_schedule_prep (*C function*), 712
napi_synchronize (*C function*), 737
net_bridge_vlan (*C struct*), 927
net_device (*C struct*), 739
net_dim (*C function*), 890
net_dim_get_def_rx_moderation (*C function*), 890
net_dim_get_def_tx_moderation (*C function*), 890
net_dim_get_rx_moderation (*C function*), 890

net_dim_get_tx_moderation (*C function*), 890
netdev_alloc_frag (*C function*), 603
netdev_alloc_skb (*C function*), 603
netdev_bonding_info_change (*C function*), 717
netdev_cap_txqueue (*C function*), 758
netdev_change_features (*C function*), 720
netdev_completed_queue (*C function*), 757
netdev_features_change (*C function*), 703
netdev_get_by_index (*C function*), 701
netdev_get_by_name (*C function*), 700
netdev_get_xmit_slave (*C function*), 717
netdev_has_any_upper_dev (*C function*), 713
netdev_has_upper_dev (*C function*), 713
netdev_has_upper_dev_all_rcu (*C function*), 713
netdev_increment_features (*C function*), 724
netdev_is_rx_handler_busy (*C function*), 710
netdev_lower_get_first_private_rcu (*C function*), 715
netdev_lower_get_next (*C function*), 715
netdev_lower_get_next_private (*C function*), 714
netdev_lower_get_next_private_rcu (*C function*), 714
netdev_lower_state_changed (*C function*), 717
netdev_master_upper_dev_get (*C function*), 714
netdev_master_upper_dev_get_rcu (*C function*), 715
netdev_master_upper_dev_link (*C function*), 716
netdev_notify_peers (*C function*), 704
netdev_port_same_parent_id (*C function*), 719
netdev_priv (*C function*), 754
netdev_priv_flags (*C enum*), 738
netdev_queue_set_dql_min_limit (*C function*), 756
netdev_reset_queue (*C function*), 758
netdev_rx_handler_register (*C function*), 710
netdev_rx_handler_unregister (*C function*), 711
netdev_sent_queue (*C function*), 757
netdev_sk_get_lowest_dev (*C function*), 717
netdev_state_change (*C function*), 703

netdev_sw_irq_coalesce_default_on (*C function*), 722
netdev_tx_completed_queue (*C function*), 757
netdev_tx_sent_queue (*C function*), 757
netdev_txq_bql_complete_prefetchw (*C function*), 756
netdev_txq_bql_enqueue_prefetchw (*C function*), 756
netdev_update_features (*C function*), 720
netdev_upper_dev_link (*C function*), 715
netdev_upper_dev_unlink (*C function*), 716
netdev_upper_get_next_dev_rcu (*C function*), 714
netif_attr_test_mask (*C function*), 760
netif_attr_test_online (*C function*), 760
netif_attrmask_next (*C function*), 760
netif_attrmask_next_and (*C function*), 760
netif_carrier_event (*C function*), 729
netif_carrier_off (*C function*), 729
netif_carrier_ok (*C function*), 762
netif_carrier_on (*C function*), 729
netif_device_attach (*C function*), 708
netif_device_detach (*C function*), 708
netif_device_present (*C function*), 763
netif_dormant (*C function*), 762
netif_dormant_off (*C function*), 762
netif_dormant_on (*C function*), 762
netif_get_num_default_rss_queues (*C function*), 708
netif_inherit_tso_max (*C function*), 708
netif_is_multiqueue (*C function*), 761
netif_napi_add (*C function*), 754
netif_napi_add_tx (*C function*), 755
netif_napi_del (*C function*), 755
netif_oper_up (*C function*), 763
netif_queue_set_napi (*C function*), 712
netif_queue_stopped (*C function*), 756
netif_receive_skb (*C function*), 711
netif_receive_skb_core (*C function*), 711
netif_receive_skb_list (*C function*), 711
netif_running (*C function*), 758
netif_rx (*C function*), 710
netif_set_real_num_queues (*C function*), 707
netif_set_real_num_rx_queues (*C function*), 707
netif_set_tso_max_segs (*C function*), 707
netif_set_tso_max_size (*C function*), 707
netif_stacked_transfer_operstate (*C function*), 720

netif_start_queue (*C function*), 755
netif_start_subqueue (*C function*), 758
netif_stop_queue (*C function*), 756
netif_stop_subqueue (*C function*), 759
netif_subqueue_stopped (*C function*), 759
netif_testing (*C function*), 763
netif_testing_off (*C function*), 763
netif_testing_on (*C function*), 762
netif_tx_lock (*C function*), 763
netif_wake_queue (*C function*), 755
netif_wake_subqueue (*C function*), 759

O

of_mdio_find_bus (*C function*), 833

P

page_pool_alloc_stats (*C struct*), 904
page_pool_create (*C function*), 898
page_pool_dev_alloc (*C function*), 900
page_pool_dev_alloc_frag (*C function*), 900
page_pool_dev_alloc_pages (*C function*), 899
page_pool_dev_alloc_va (*C function*), 900
page_pool_free_va (*C function*), 902
page_pool_get_dma_addr (*C function*), 902
page_pool_get_dma_dir (*C function*), 901
page_pool_get_stats (*C function*), 902
page_pool_params (*C struct*), 899
page_pool_put_full_page (*C function*), 901
page_pool_put_page (*C function*), 901
page_pool_put_page_bulk (*C function*), 903
page_pool_recycle_direct (*C function*), 902
page_pool_recycle_stats (*C struct*), 904
page_pool_stats (*C struct*), 905
pcs_an_restart (*C function*), 852
pcs_config (*C function*), 851
pcs_disable (*C function*), 851
pcs_enable (*C function*), 850
pcs_get_state (*C function*), 851
pcs_link_up (*C function*), 852
pcs_validate (*C function*), 850
phy_advertise_supported (*C function*), 827
phy_aneg_done (*C function*), 765
phy_attach (*C function*), 822
phy_attach_direct (*C function*), 821
phy_c45_device_ids (*C struct*), 798
phy_check_downshift (*C function*), 778
phy_check_link_status (*C function*), 775
phy_check_valid (*C function*), 766
phy_clear_bits (*C function*), 815
phy_clear_bits_mmd (*C function*), 816
phy_config_interrupt (*C function*), 772

phy_connect (C function), 820
phy_connect_direct (C function), 820
phy_detach (C function), 823
phy_device (C struct), 799
phy_device_register (C function), 819
phy_device_remove (C function), 819
phy_disable_interrupts (C function), 775
phy_disconnect (C function), 820
phy_do_ioctl (C function), 766
phy_do_ioctl_running (C function), 767
phy_driver (C struct), 806
phy_driver_register (C function), 830
phy_duplex_to_str (C function), 776
phy_enable_interrupts (C function), 776
phy_error (C function), 769
phy_ethtool_get_eee (C function), 771
phy_ethtool_get_plca_cfg (C function), 774
phy_ethtool_get_plca_status (C function),
775
phy_ethtool_get_sset_count (C function),
767
phy_ethtool_get_stats (C function), 767
phy_ethtool_get_strings (C function), 767
phy_ethtool_get_wol (C function), 772
phy_ethtool_nway_reset (C function), 772
phy_ethtool_set_eee (C function), 771
phy_ethtool_set_plca_cfg (C function), 774
phy_ethtool_set_wol (C function), 772
phy_find_first (C function), 820
phy_find_valid (C function), 772
phy_free_interrupt (C function), 770
phy_get_c45_ids (C function), 819
phy_get_eee_err (C function), 771
phy_get_internal_delay (C function), 829
phy_get_pause (C function), 828
phy_get_rate_matching (C function), 765
phy_has_hwtstamp (C function), 816
phy_has_rxtstamp (C function), 816
phy_has_tsinfo (C function), 817
phy_has_txstamp (C function), 817
phy_id_compare (C function), 811
phy_init_eee (C function), 771
phy_interface_is_rgmii (C function), 817
phy_interface_mode_is_8023z (C function),
817
phy_interface_mode_is_rgmii (C function),
817
phy_interface_num_ports (C function), 776
phy_interface_t (C enum), 792
phy_interrupt (C function), 775
phy_interrupt_is_valid (C function), 816
phy_is_internal (C function), 817
phy_is_pseudo_fixed_link (C function), 818
phy_is_started (C function), 812
phy_led (C struct), 806
phy_lookup_setting (C function), 776
phy_mac_interrupt (C function), 770
phy_mii_ioctl (C function), 766
phy_modes (C function), 794
phy_modify (C function), 782
phy_modify_changed (C function), 781
phy_modify_mmd (C function), 783
phy_modify_mmd_changed (C function), 783
phy_modify_paged (C function), 786
phy_modify_paged_changed (C function), 785
phy_module_driver (C macro), 818
phy_on_sfp (C function), 817
phy_package_join (C function), 822
phy_package_leave (C function), 823
phy_package_read_mmd (C function), 779
phy_package_shared (C struct), 795
phy_package_write_mmd (C function), 780
phy_plca_cfg (C struct), 805
phy_plca_status (C struct), 806
phy_poll_reset (C function), 831
phy_polling_mode (C function), 816
phy_prepare_link (C function), 831
phy_print_status (C function), 765
phy_probe (C function), 832
phy_queue_state_machine (C function), 767
phy_rate_matching_to_str (C function), 776
phy_read (C function), 812
phy_read_mmd (C function), 778
phy_read_mmd_poll_timeout (C macro), 813
phy_read_paged (C function), 785
phy_register_fixup (C function), 818
phy_remove_link_mode (C function), 827
phy_request_interrupt (C function), 770
phy_reset_after_clk_enable (C function),
823
phy_resolve_aneg_linkmode (C function),
777
phy_resolve_aneg_pause (C function), 777
phy_restart_aneg (C function), 765
phy_restore_page (C function), 784
phy_SANITIZE_SETTINGS (C function), 773
phy_save_page (C function), 784
phy_select_page (C function), 784
phy_set_asym_pause (C function), 828
phy_set_bits (C function), 814
phy_set_bits_mmd (C function), 815
phy_set_max_speed (C function), 777

phy_set_sym_pause (C function), 827
phy_sfp_attach (C function), 821
phy_sfp_detach (C function), 821
phy_sfp_probe (C function), 821
phy_speed_down (C function), 769
phy_speed_to_str (C function), 776
phy_speed_up (C function), 769
phy_start (C function), 770
phy_start_aneg (C function), 768
phy_start_cable_test (C function), 768
phy_start_cable_test_tdr (C function), 768
phy_start_machine (C function), 769
phy_state (C enum), 797
phy_state_machine (C function), 776
phy_stop (C function), 770
phy_stop_machine (C function), 775
phy_support_asym_pause (C function), 827
phy_support_sym_pause (C function), 827
phy_supported_speeds (C function), 773
phy_tdr_config (C struct), 804
phy_trigger_machine (C function), 767
phy_unregister_fixup (C function), 818
phy_validate_pause (C function), 828
phy_write (C function), 812
phy_write_mmd (C function), 779
phy_write_paged (C function), 785
phydev_id_compare (C function), 811
phylink (C struct), 853
phylink_cap_from_speed_duplex (C function), 854
phylink_caps_to_linkmodes (C function), 853
phylink_config (C struct), 843
phylink_connect_phy (C function), 856
phylink_create (C function), 855
phylink_decode_usgmii_word (C function), 863
phylink_decode_usxgmii_word (C function), 863
phylink_destroy (C function), 856
phylink_disconnect_phy (C function), 857
phylink_ethtool_get_eee (C function), 861
phylink_ethtool_get_pauseparam (C function), 861
phylink_ethtool_get_wol (C function), 859
phylink_ethtool_ksettings_get (C function), 860
phylink_ethtool_ksettings_set (C function), 860
phylink_ethtool_nway_reset (C function), 860
phylink_ethtool_set_eee (C function), 862
phylink_ethtool_set_pauseparam (C function), 861
phylink_ethtool_set_wol (C function), 859
phylink_expects_phy (C function), 856
phylink_fwnode_phy_connect (C function), 857
phylink_get_capabilities (C function), 854
phylink_get_eee_err (C function), 861
phylink_get_link_timer_ns (C function), 852
phylink_init_eee (C function), 861
phylink_interface_max_speed (C function), 853
phylink_limit_mac_speed (C function), 854
phylink_link_state (C struct), 842
phylink_mac_change (C function), 858
phylink_mac_ops (C struct), 844
phylink_mii_c22_pcs_an_restart (C function), 865
phylink_mii_c22_pcs_config (C function), 864
phylink_mii_c22_pcs_decode_state (C function), 863
phylink_mii_c22_pcs_encode_advertisement (C function), 864
phylink_mii_c22_pcs_get_state (C function), 864
phylink_mii_ioctl (C function), 862
phylink_of_phy_connect (C function), 857
phylink_pcs (C struct), 849
phylink_pcs_change (C function), 858
phylink_pcs_neg_mode (C function), 855
phylink_pcs_ops (C struct), 849
phylink_resume (C function), 859
phylink_set_port_modes (C function), 853
phylink_speed_down (C function), 862
phylink_speed_up (C function), 863
phylink_start (C function), 858
phylink_stop (C function), 858
phylink_suspend (C function), 859
phylink_validate_mask_caps (C function), 854
platform_get_ethdev_address (C function), 727
plca_check_valid (C function), 774
pskb_expand_head (C function), 640
pskb_put (C function), 641
pskb_trim_rcsum (C function), 610
pskb_trim_unique (C function), 602
publ_to_item (C function), 1596

publication (*C struct*), 1557

R

`rcvbuf_limit` (*C function*), 1615

`rdma_dim` (*C function*), 891

`register_netdev` (*C function*), 721

`register_netdevice` (*C function*), 720

`register_netdevice_notifier` (*C function*), 705

`register_netdevice_notifier_net` (*C function*), 705

`rpc_add_pipe_dir_object` (*C function*), 692

`rpc_alloc_iostats` (*C function*), 690

`rpc_bind_new_program` (*C function*), 695

`rpc_call_async` (*C function*), 695

`rpc_call_sync` (*C function*), 695

`rpc_cancel_tasks` (*C function*), 694

`rpc_clnt_add_xprt` (*C function*), 698

`rpc_clnt_iterate_for_each_xprt` (*C function*), 694

`rpc_clnt_setup_test_and_add_xprt` (*C function*), 698

`rpc_clnt_test_and_add_xprt` (*C function*), 697

`rpc_clone_client` (*C function*), 693

`rpc_clone_client_set_auth` (*C function*), 693

`rpc_count_iostats` (*C function*), 690

`rpc_count_iostats_metrics` (*C function*), 690

`rpc_create` (*C function*), 693

`rpc_find_or_alloc_pipe_dir_object` (*C function*), 692

`rpc_force_rebind` (*C function*), 697

`rpc_free` (*C function*), 689

`rpc_free_iostats` (*C function*), 690

`rpc_init_pipe_dir_head` (*C function*), 691

`rpc_init_pipe_dir_object` (*C function*), 691

`rpc_localaddr` (*C function*), 696

`rpc_malloc` (*C function*), 689

`rpc_max_bc_payload` (*C function*), 697

`rpc_max_payload` (*C function*), 697

`rpc_mkpipe_dentry` (*C function*), 690

`rpc_net_ns` (*C function*), 697

`rpc_peeraddr` (*C function*), 696

`rpc_peeraddr2str` (*C function*), 696

`rpc_prepare_reply_pages` (*C function*), 696

`rpc_queue_upcall` (*C function*), 690

`rpc_remove_pipe_dir_object` (*C function*), 692

`rpc_run_task` (*C function*), 695

`rpc_switch_client_transport` (*C function*), 693

`rpc_unlink` (*C function*), 691

`rpc_wake_up` (*C function*), 688

`rpc_wake_up_status` (*C function*), 689

`rpcb_getport_async` (*C function*), 693

`rps_may_expire_flow` (*C function*), 709

`rtnl_link_stats64` (*C struct*), 1500

S

`service_range` (*C struct*), 1591

`service_range_foreach_match` (*C macro*), 1592

`service_range_match_first` (*C function*), 1593

`service_range_match_next` (*C function*), 1593

`set_channel` (*C function*), 570

`sfp_bus` (*C struct*), 865

`sfp_bus_add_upstream` (*C function*), 870

`sfp_bus_del_upstream` (*C function*), 871

`sfp_bus_find_fwnode` (*C function*), 870

`sfp_bus_put` (*C function*), 868

`sfp_eeprom_id` (*C struct*), 865

`sfp_get_module_eeprom` (*C function*), 868

`sfp_get_module_eeprom_by_page` (*C function*), 869

`sfp_get_module_info` (*C function*), 868

`sfp_may_have_phy` (*C function*), 867

`sfp_parse_port` (*C function*), 867

`sfp_parse_support` (*C function*), 867

`sfp_select_interface` (*C function*), 868

`sfp_upstream_ops` (*C struct*), 866

`sfp_upstream_set_signal_rate` (*C function*), 870

`sfp_upstream_start` (*C function*), 869

`sfp_upstream_stop` (*C function*), 869

`sk_alloc` (*C function*), 655

`sk_attach_filter` (*C function*), 662

`sk_buff` (*C struct*), 582

`sk_capable` (*C function*), 654

`sk_clone_lock` (*C function*), 656

`sk_eat_skb` (*C function*), 628

`sk_filter_trim_cap` (*C function*), 661

`sk_for_each_entry_offset_rcu` (*C macro*), 625

`sk_has_allocations` (*C function*), 626

`sk_net_capable` (*C function*), 655

`sk_ns_capable` (*C function*), 654

`sk_page_frag` (*C function*), 627

`sk_rmem_alloc_get` (*C function*), 626

skb_set_memalloc (*C function*), 655
 skb_stream_wait_connect (*C function*), 660
 skb_stream_wait_memory (*C function*), 660
 skb_user_data_is_nocopy (*C function*), 624
 skb_wait_data (*C function*), 656
 skb_wmem_alloc_get (*C function*), 626
 skb_abort_seq_read (*C function*), 647
 skb_append (*C function*), 646
 skb_availroom (*C function*), 601
 skb_checksum_complete (*C function*), 612
 skb_checksum_none_assert (*C function*), 612
 skb_checksum_setup (*C function*), 650
 skb_checksum_trimmed (*C function*), 650
 skb_clone (*C function*), 639
 skb_clone_sk (*C function*), 649
 skb_clone_writable (*C function*), 607
 skb_cloned (*C function*), 595
 skb_complete_tx_timestamp (*C function*), 611
 skb_complete_wifi_ack (*C function*), 611
 skb_condense (*C function*), 653
 skb_copy (*C function*), 639
 skb_copy_and_csum_datagram_msg (*C function*), 659
 skb_copy_and_hash_datagram_iter (*C function*), 658
 skb_copy_bits (*C function*), 643
 skb_copy_datagram_from_iter (*C function*), 659
 skb_copy_datagram_iter (*C function*), 658
 skb_copy_expand (*C function*), 641
 skb_copy_ubufs (*C function*), 639
 skb_cow (*C function*), 607
 skb_cow_data (*C function*), 649
 skb_cow_head (*C function*), 607
 skb_dequeue (*C function*), 644
 skb_dequeue_tail (*C function*), 645
 skb_dst (*C function*), 591
 skb_dst_is_noref (*C function*), 592
 skb_dst_set (*C function*), 591
 skb_dst_set_noref (*C function*), 591
 skb_eth_pop (*C function*), 651
 skb_eth_push (*C function*), 651
 skb_expand_head (*C function*), 640
 skb_ext (*C struct*), 612
 skb_ext_add (*C function*), 653
 skb_fclone_busy (*C function*), 593
 skb_fill_page_desc (*C function*), 600
 skb_fill_page_desc_noacc (*C function*), 601
 skb_find_text (*C function*), 647
 skb_frag_address (*C function*), 606
 skb_frag_address_safe (*C function*), 606
 skb_frag_dma_map (*C function*), 607
 skb_frag.foreach.page (*C macro*), 581
 skb_frag.must.loop (*C function*), 581
 skb_frag.off (*C function*), 604
 skb_frag.off.add (*C function*), 604
 skb_frag.off.copy (*C function*), 605
 skb_frag.off.set (*C function*), 605
 skb_frag.page (*C function*), 605
 skb_frag.page.copy (*C function*), 606
 skb_frag.ref (*C function*), 605
 skb_frag.size (*C function*), 580
 skb_frag.size.add (*C function*), 581
 skb_frag.size.set (*C function*), 580
 skb_frag.size.sub (*C function*), 581
 skb_frag.unref (*C function*), 606
 skb_get (*C function*), 595
 skb_get_timestamp (*C function*), 610
 skb_has_shared_frag (*C function*), 609
 skb_head_is_locked (*C function*), 613
 skb_header_cloned (*C function*), 595
 skb_headroom (*C function*), 601
 skb_kill_datagram (*C function*), 658
 skb_len_add (*C function*), 599
 skb_linearize (*C function*), 609
 skb_linearize_cow (*C function*), 609
 skb_morph (*C function*), 638
 skb_mpls_dec_ttl (*C function*), 652
 skb_mpls_pop (*C function*), 652
 skb_mpls_push (*C function*), 651
 skb_mpls_update_lse (*C function*), 652
 skb_napi_id (*C function*), 592
 skb_needs_linearize (*C function*), 610
 skb_orphan (*C function*), 602
 skb_orphan frags (*C function*), 602
 skb_pad (*C function*), 593
 skb_padto (*C function*), 608
 skb_page_frag_refill (*C function*), 656
 skb_partial_csum_set (*C function*), 649
 skb_peek (*C function*), 596
 skb_peek_next (*C function*), 596
 skb_peek_tail (*C function*), 597
 skb_pfmemalloc (*C function*), 591
 skb_postpull_rcsum (*C function*), 609
 skb_postpush_rcsum (*C function*), 609
 skb_prepare_seq_read (*C function*), 646
 skb_propagate_pfmemalloc (*C function*), 604
 skb_pull (*C function*), 642
 skb_pull_data (*C function*), 642
 skb_pull_rcsum (*C function*), 648
 skb_push (*C function*), 642

skb_push_rcsum (*C function*), 610
 skb_put (*C function*), 642
 skb_put_padto (*C function*), 608
 skb_queue_empty (*C function*), 593
 skb_queue_empty_lockless (*C function*), 594
 skb_queue_head (*C function*), 645
 skb_queue_is_first (*C function*), 594
 skb_queue_is_last (*C function*), 594
 skb_queue_len (*C function*), 597
 skb_queue_len_lockless (*C function*), 597
 skb_queue_next (*C function*), 594
 skb_queue_prev (*C function*), 594
 skb_queue_purge_reason (*C function*), 645
 skb_queue_splice (*C function*), 597
 skb_queue_splice_init (*C function*), 598
 skb_queue_splice_tail (*C function*), 598
 skb_queue_splice_tail_init (*C function*), 598
 skb_queue_tail (*C function*), 645
 skb_reserve (*C function*), 601
 skb_rtable (*C function*), 592
 skb_scrub_packet (*C function*), 650
 skb_segment (*C function*), 648
 skb_seq_read (*C function*), 647
 skb_share_check (*C function*), 596
 skb_shared (*C function*), 595
 skb_shared_hwtstamps (*C struct*), 582
 skb_splice_from_iter (*C function*), 654
 skb_split (*C function*), 646
 skb_steal_sock (*C function*), 628
 skb_store_bits (*C function*), 644
 skb_tailroom (*C function*), 601
 skb_tailroom_reserve (*C function*), 602
 skb_to_sgvec (*C function*), 648
 skb_trim (*C function*), 643
 skb_try_coalesce (*C function*), 650
 skb_tstamp_tx (*C function*), 611
 skb_tx_error (*C function*), 638
 skb_tx_timestamp (*C function*), 611
 skb_unlink (*C function*), 646
 skb_unref (*C function*), 592
 skb_unshare (*C function*), 596
 skb_zerocopy (*C function*), 644
 skwq_has_sleeper (*C function*), 626
 sock (*C struct*), 616
 sock_alloc (*C function*), 629
 sock_alloc_file (*C function*), 629
 sock_common (*C struct*), 613
 sock_create (*C function*), 632
 sock_create_kern (*C function*), 633
 sock_create_lite (*C function*), 632

sock_from_file (*C function*), 629
 sock_poll_wait (*C function*), 627
 sock_recvmsg (*C function*), 631
 sock_register (*C function*), 633
 sock_release (*C function*), 630
 sock_sendmsg (*C function*), 630
 sock_shutdown_cmd (*C enum*), 579
 sock_type (*C enum*), 579
 sock_unregister (*C function*), 633
 socket (*C struct*), 580
 sockfd_lookup (*C function*), 629
 start (*C function*), 570
 stop (*C function*), 570
 svc_find_xprt (*C function*), 682
 svc_print_addr (*C function*), 681
 svc_recv (*C function*), 682
 svc_reg_xprt_class (*C function*), 679
 svc_reserve (*C function*), 681
 svc_unreg_xprt_class (*C function*), 680
 svc_wake_up (*C function*), 681
 svc_xprt_close (*C function*), 682
 svc_xprt_create (*C function*), 680
 svc_xprt_deferred_close (*C function*), 680
 svc_xprt_destroy_all (*C function*), 682
 svc_xprt_enqueue (*C function*), 681
 svc_xprt_names (*C function*), 683
 svc_xprt_received (*C function*), 680
 synchronize_net (*C function*), 723

T

tipc_accept (*C function*), 1617
 tipc_aead (*C struct*), 1565
 tipc_aead_clone (*C function*), 1569
 tipc_aead_decrypt (*C function*), 1571
 tipc_aead_encrypt (*C function*), 1570
 tipc_aead_free (*C function*), 1569
 tipc_aead_init (*C function*), 1569
 tipc_aead_key_generate (*C function*), 1568
 tipc_aead_key_validate (*C function*), 1568
 tipc_aead_mem_alloc (*C function*), 1570
 tipc_aead_tfm_next (*C function*), 1569
 tipc_bc_base (*C struct*), 1559
 tipc_bearer (*C struct*), 1555
 tipc_bearer_find (*C function*), 1561
 tipc_buf_acquire (*C function*), 1588
 tipc_conn (*C struct*), 1622
 tipc_connect (*C function*), 1617
 tipc_crypto (*C struct*), 1566
 tipc_crypto_key_attach (*C function*), 1572
 tipc_crypto_key_distr (*C function*), 1575
 tipc_crypto_key_init (*C function*), 1572

tipc_crypto_key_pick_tx (*C function*), 1573
tipc_crypto_key_rcv (*C function*), 1576
tipc_crypto_key_synch (*C function*), 1573
tipc_crypto_key_try_align (*C function*), 1572
tipc_crypto_key_xmit (*C function*), 1575
tipc_crypto_msg_rcv (*C function*), 1575
tipc_crypto_rcv (*C function*), 1574
tipc_crypto_rekeying_sched (*C function*), 1576
tipc_crypto_stats (*C struct*), 1566
tipc_crypto_work_rx (*C function*), 1576
tipc_crypto_work_tx (*C function*), 1577
tipc_crypto_xmit (*C function*), 1574
tipc_data_ready (*C function*), 1615
tipc_disc_create (*C function*), 1579
tipc_disc_delete (*C function*), 1579
tipc_disc_init_msg (*C function*), 1578
tipc_disc_rcv (*C function*), 1578
tipc_disc_reset (*C function*), 1579
tipc_discoverer (*C struct*), 1577
tipc_ehdr_build (*C function*), 1571
tipc_ehdr_validate (*C function*), 1571
tipc_enable_bearer (*C function*), 1561
tipc_get_gap_ack_blk (*C function*), 1587
tipc_getname (*C function*), 1608
tipc_getsockopt (*C function*), 1619
tipc_l2_device_event (*C function*), 1563
tipc_l2_rcv_msg (*C function*), 1562
tipc_l2_send_msg (*C function*), 1562
tipc_link (*C struct*), 1579
tipc_link_bc_create (*C function*), 1584
tipc_link_create (*C function*), 1583
tipc_link_dump (*C function*), 1587
tipc_link_failover_prepare (*C function*), 1587
tipc_link_fsm_evt (*C function*), 1585
tipc_link_reset_stats (*C function*), 1587
tipc_link_set_skb_retransmit_time (*C function*), 1586
tipc_link_too_silent (*C function*), 1585
tipc_link_xmit (*C function*), 1586
tipc_list_dump (*C function*), 1623
tipc_listen (*C function*), 1617
tipc_media (*C struct*), 1554
tipc_media_addr (*C struct*), 1553
tipc_media_addr_printf (*C function*), 1560
tipc_media_find (*C function*), 1560
tipc_msg_append (*C function*), 1588
tipc_msg_build (*C function*), 1589
tipc_msg_bundle (*C function*), 1589
tipc_msg_extract (*C function*), 1590
tipc_msg_fragment (*C function*), 1589
tipc_msg_lookup_dest (*C function*), 1591
tipc_msg_reverse (*C function*), 1591
tipc_msg_try_bundle (*C function*), 1590
tipc_named_node_up (*C function*), 1597
tipc_named_publish (*C function*), 1596
tipc_named_rcv (*C function*), 1598
tipc_named_reinit (*C function*), 1598
tipc_named_withdraw (*C function*), 1596
tipc_nametbl_lookup_anycast (*C function*), 1594
tipc_nametbl_subscribe (*C function*), 1595
tipc_nametbl_unsubscribe (*C function*), 1595
tipc_nametbl_withdraw (*C function*), 1595
tipc_node (*C struct*), 1598
tipc_node_bc_rcv (*C function*), 1603
tipc_node_check_state (*C function*), 1603
tipc_node_crypto_rx (*C function*), 1601
tipc_node_dump (*C function*), 1604
tipc_node_get_linkname (*C function*), 1602
tipc_node_link_failover (*C function*), 1601
tipc_node_link_up (*C function*), 1601
tipc_node_xmit (*C function*), 1603
tipc_parse_udp_addr (*C function*), 1564
tipc_poll (*C function*), 1609
tipc_publ_create (*C function*), 1593
tipc_publ_purge (*C function*), 1597
tipc_rcv (*C function*), 1604
tipc_recvmsg (*C function*), 1614
tipc_recvstream (*C function*), 1614
tipc_release (*C function*), 1607
tipc_reset_bearer (*C function*), 1561
tipc_send_group_anycast (*C function*), 1610
tipc_send_group_bcast (*C function*), 1611
tipc_send_group_mcast (*C function*), 1611
tipc_send_group_msg (*C function*), 1610
tipc_send_group_unicast (*C function*), 1610
tipc_send_packet (*C function*), 1613
tipc_sendmcast (*C function*), 1609
tipc_sendmsg (*C function*), 1612
tipc_sendstream (*C function*), 1613
tipc_service (*C struct*), 1592
tipc_service_create (*C function*), 1594
tipc_service_delete (*C function*), 1595
tipc_service_remove_publ (*C function*), 1594
tipc_service_subscribe (*C function*), 1594
tipc_setsockopt (*C function*), 1618
tipc_shutdown (*C function*), 1618

tipc_sk_anc_data_recv (*C function*), 1614
 tipc_sk_backlog_rcv (*C function*), 1616
 tipc_sk_conn_proto_rcv (*C function*), 1612
 tipc_sk_create (*C function*), 1607
 tipc_sk_dump (*C function*), 1620
 tipc_sk_enqueue (*C function*), 1616
 tipc_sk_filter_connect (*C function*), 1615
 tipc_sk_filter_rcv (*C function*), 1616
 tipc_sk_filtering (*C function*), 1619
 tipc_sk_mcast_rcv (*C function*), 1612
 tipc_sk_overlimit1 (*C function*), 1620
 tipc_sk_overlimit2 (*C function*), 1620
 tipc_sk_rcv (*C function*), 1617
 tipc_sk_set_orig_addr (*C function*), 1613
 tipc_skb_dump (*C function*), 1623
 tipc_sock (*C struct*), 1604
 tipc_socket_init (*C function*), 1619
 tipc_socket_stop (*C function*), 1619
 tipc_sub_check_overlap (*C function*), 1621
 tipc_subscription (*C struct*), 1553
 tipc_tfm (*C struct*), 1565
 tipc_topsrv (*C struct*), 1621
 tipc_udp_enable (*C function*), 1564
 tipc_update_nametbl (*C function*), 1597
 tipc_write_space (*C function*), 1615
 tsk_advance_rx_queue (*C function*), 1607
 tsk_rej_rx_queue (*C function*), 1607

U

u64_to_ether_addr (*C function*), 734
 udp_bearer (*C struct*), 1563
 udp_media_addr (*C struct*), 1563
 unlock_sock_fast (*C function*), 626
 unregister_netdev (*C function*), 723
 unregister_netdevice_many (*C function*), 723
 unregister_netdevice_notifier (*C function*), 705
 unregister_netdevice_notifier_net (*C function*), 706
 unregister_netdevice_queue (*C function*), 723

X

xdr_buf_subsegment (*C function*), 675
 xdr_buf_trim (*C function*), 677
 xdr_encode_opaque (*C function*), 669
 xdr_encode_opaque_fixed (*C function*), 669
 xdr_enter_page (*C function*), 675
 xdr_finish_decode (*C function*), 674
 xdr_init_decode (*C function*), 673
 xdr_init_decode_pages (*C function*), 674

xdr_init_encode (*C function*), 671
 xdr_init_encode_pages (*C function*), 671
 xdr_inline_decode (*C function*), 674
 xdr_inline_pages (*C function*), 670
 xdr_page_pos (*C function*), 670
 xdr_read_pages (*C function*), 674
 xdr_reserve_space (*C function*), 672
 xdr_reserve_space_vec (*C function*), 672
 xdr_restrict buflen (*C function*), 673
 xdr_set_pagelen (*C function*), 675
 xdr_stream_decode_opaque (*C function*), 677
 xdr_stream_decode_opaque_auth (*C function*), 679
 xdr_stream_decode_opaque_dup (*C function*), 677
 xdr_stream_decode_string (*C function*), 678
 xdr_stream_decode_string_dup (*C function*), 678
 xdr_stream_encode_opaque_auth (*C function*), 679
 xdr_stream_move_subsegment (*C function*), 676
 xdr_stream_pos (*C function*), 670
 xdr_stream_subsegment (*C function*), 676
 xdr_stream_zero (*C function*), 677
 xdr_terminate_string (*C function*), 669
 xdr_truncate_decode (*C function*), 673
 xdr_truncate_encode (*C function*), 672
 xdr_write_pages (*C function*), 673
 xmit_async (*C function*), 570
 xpert_adjust_cwnd (*C function*), 685
 xpert_complete_rqst (*C function*), 687
 xpert_disconnect_done (*C function*), 686
 xpert_find_transport_ident (*C function*), 684
 xpert_force_disconnect (*C function*), 686
 xpert_get (*C function*), 688
 xpert_lookup_rqst (*C function*), 687
 xpert_pin_rqst (*C function*), 687
 xpert_put (*C function*), 688
 xpert_reconnect_backoff (*C function*), 686
 xpert_reconnect_delay (*C function*), 686
 xpert_register_transport (*C function*), 683
 xpert_release_rqst_cong (*C function*), 685
 xpert_release_xprt (*C function*), 684
 xpert_release_xprt_cong (*C function*), 684
 xpert_request_get_cong (*C function*), 685
 xpert_reserve_xprt (*C function*), 684
 xpert_unpin_rqst (*C function*), 687
 xpert_unregister_transport (*C function*), 683

`xprt_update_rtt` (*C function*), 687
`xprt_wait_for_buffer_space` (*C function*),
686
`xprt_wait_for_reply_request_def` (*C function*), 688
`xprt_wait_for_reply_request_rtt` (*C function*), 688
`xprt_wake_pending_tasks` (*C function*), 685
`xprt_write_space` (*C function*), 686

Z

`zerocopy_sg_from_iter` (*C function*), 659