
Linux Networking Documentation

The kernel development community

Jun 10, 2024

CONTENTS

1	netdev FAQ	3
1.1	Q: What is netdev?	3
1.2	Q: How do the changes posted to netdev make their way into Linux?	3
1.3	Q: How often do changes from these trees make it to the mainline Linus tree?	4
1.4	Q: How do I indicate which tree (net vs. net-next) my patch should be in?	5
1.5	Q: I sent a patch and I'm wondering what happened to it?	5
1.6	Q: The above only says "Under Review" . How can I find out more?	5
1.7	Q: I submitted multiple versions of the patch series	5
1.8	Q: I made changes to only a few patches in a patch series should I resend only those changed?	6
1.9	Q: I submitted multiple versions of a patch series and it looks like a version other than the last one has been accepted, what should I do?	6
1.10	Q: Are there special rules regarding stable submissions on netdev?	6
1.11	Q: Is the comment style convention different for the networking content?	6
1.12	Q: I am working in existing code that has the former comment style and not the latter.	7
1.13	Q: I found a bug that might have possible security implications or similar.	7
1.14	Q: What level of testing is expected before I submit my change?	7
1.15	Q: How do I post corresponding changes to user space components?	7
1.16	Q: Any other tips to help ensure my net/net-next patch gets OK'd?	8
2	AF_XDP	9
2.1	Overview	9
2.2	Concepts	10
2.3	Libbpf	12
2.4	XSKMAP / BPF_MAP_TYPE_XSKMAP	13
2.5	Configuration Flags and Socket Options	13
2.6	Usage	17
2.7	Sample application	19
2.8	FAQ	19
2.9	Credits	20
3	Bare UDP Tunnelling Module Documentation	21
3.1	Special Handling	21
3.2	Usage	21

4	batman-adv	23
4.1	Configuration	23
4.2	Usage	24
4.3	Logging/Debugging	25
4.4	batctl	25
4.5	Contact	26
5	SocketCAN - Controller Area Network	27
5.1	Overview / What is SocketCAN	27
5.2	Motivation / Why Using the Socket API	27
5.3	SocketCAN Concept	29
5.4	How to use SocketCAN	30
5.5	SocketCAN Core Module	44
5.6	CAN Network Drivers	45
5.7	SocketCAN Resources	52
5.8	Credits	52
6	The UCAN Protocol	53
6.1	USB Endpoints	53
6.2	CONTROL Messages	53
6.3	IN Message Format	56
6.4	OUT Message Format	58
6.5	CAN Error Handling	59
6.6	Example Conversation	59
7	Hardware Device Drivers	61
7.1	AppleTalk Device Drivers	61
7.2	Asynchronous Transfer Mode (ATM) Device Drivers	65
7.3	Cable Modem Device Drivers	72
7.4	Cellular Modem Device Drivers	76
7.5	Ethernet Device Drivers	78
7.6	Fiber Distributed Data Interface (FDDI) Device Drivers	312
7.7	Amateur Radio Device Drivers	318
7.8	Classic WAN Device Drivers	333
7.9	Wi-Fi Device Drivers	346
8	Distributed Switch Architecture	365
8.1	Architecture	365
8.2	Broadcom RoboSwitch Ethernet switch driver	376
8.3	Broadcom Starfighter 2 Ethernet switch driver	380
8.4	LAN9303 Ethernet switch driver	382
8.5	NXP SJA1105 switch driver	383
8.6	DSA switch configuration from userspace	394
9	Linux Devlink Documentation	401
9.1	Interface documentation	401
9.2	Driver-specific documentation	429
10	CAIF	443
10.1	Linux CAIF	443
10.2	Using Linux CAIF	447
11	Netlink interface for ethtool	451

11.1	Basic information	451
11.2	Conventions	451
11.3	Request header	452
11.4	Bit sets	452
11.5	List of message types	454
11.6	STRSET_GET	456
11.7	LINKINFO_GET	457
11.8	LINKINFO_SET	458
11.9	LINKMODES_GET	458
11.10	LINKMODES_SET	459
11.11	LINKSTATE_GET	460
11.12	DEBUG_GET	462
11.13	DEBUG_SET	462
11.14	WOL_GET	463
11.15	WOL_SET	463
11.16	FEATURES_GET	463
11.17	FEATURES_SET	464
11.18	PRIVFLAGS_GET	465
11.19	PRIVFLAGS_SET	465
11.20	RINGS_GET	465
11.21	RINGS_SET	466
11.22	CHANNELS_GET	466
11.23	CHANNELS_SET	467
11.24	COALESCE_GET	467
11.25	COALESCE_SET	469
11.26	PAUSE_GET	470
11.27	PAUSE_SET	471
11.28	EEE_GET	471
11.29	EEE_SET	471
11.30	TSINFO_GET	472
11.31	CABLE_TEST	472
11.32	CABLE_TEST TDR	473
11.33	TUNNEL_INFO	475
11.34	Request translation	476
12	IEEE 802.15.4 Developer' s Guide	479
12.1	Introduction	479
12.2	Socket API	479
12.3	6LoWPAN Linux implementation	480
12.4	Drivers	480
12.5	Device drivers API	481
13	J1939 Documentation	483
13.1	Overview / What Is J1939	483
13.2	Motivation	483
13.3	J1939 concepts	484
13.4	How to Use J1939	485
14	Linux Networking and Network Devices APIs	491
14.1	Linux Networking	491
14.2	Network device support	616

15 MSG_ZEROCOPY	765
15.1 Intro	765
15.2 Interface	766
15.3 Implementation	769
15.4 Testing	769
16 FAILOVER	771
16.1 Overview	771
17 Net DIM - Generic Network Dynamic Interrupt Moderation	773
17.1 Assumptions	773
17.2 Introduction	773
17.3 Net DIM Algorithm	774
17.4 Registering a Network Device to DIM	775
17.5 Example	775
17.6 Dynamic Interrupt Moderation (DIM) library API	776
18 NET_FAILOVER	785
18.1 Overview	785
18.2 virtio-net accelerated datapath: STANDBY mode	785
18.3 Live Migration of a VM with SR-IOV VF & virtio-net in STANDBY mode	786
19 Page Pool API	789
19.1 Architecture overview	789
19.2 API interface	790
19.3 Coding examples	791
20 PHY Abstraction Layer	793
20.1 Purpose	793
20.2 The MDIO bus	793
20.3 (RG)MII/electrical interface considerations	794
20.4 Connecting to a PHY	796
20.5 Letting the PHY Abstraction Layer do Everything	796
20.6 PHY interface modes	797
20.7 Pause frames / flow control	798
20.8 Keeping Close Tabs on the PAL	798
20.9 Doing it all yourself	799
20.10 PHY Device Drivers	800
20.11 Board Fixups	801
20.12 Standards	802
21 phylink	803
21.1 Overview	803
21.2 Modes of operation	803
21.3 Rough guide to converting a network driver to sfp/phylink	804
22 IP-Aliasing	809
22.1 Alias creation	809
22.2 Alias deletion	809
22.3 Alias (re-)configuring	809
22.4 Relationship with main device	810
23 Ethernet Bridging	811

24	SNMP counter	813
24.1	General IPv4 counters	813
24.2	ICMP counters	815
24.3	General TCP counters	816
24.4	TCP Fast Open	818
24.5	TCP Fast Path	819
24.6	TCP abort	820
24.7	TCP Hybrid Slow Start	821
24.8	TCP retransmission and congestion control	821
24.9	DSACK	823
24.10	Invalid SACK and DSACK	823
24.11	SACK shift	824
24.12	TCP out of order	824
24.13	TCP PAWS	825
24.14	TCP ACK skip	825
24.15	TCP receive window	826
24.16	Delayed ACK	826
24.17	Tail Loss Probe (TLP)	827
24.18	TCP Fast Open description	827
24.19	SYN cookies	828
24.20	Challenge ACK	828
24.21	prune	828
24.22	examples	829
25	Checksum Offloads	845
25.1	Introduction	845
25.2	TX Checksum Offload	845
25.3	LCO: Local Checksum Offload	846
25.4	RCO: Remote Checksum Offload	847
26	Segmentation Offloads	849
26.1	Introduction	849
26.2	TCP Segmentation Offload	849
26.3	UDP Fragmentation Offload	850
26.4	IPIP, SIT, GRE, UDP Tunnel, and Remote Checksum Offloads	850
26.5	Generic Segmentation Offload	851
26.6	Generic Receive Offload	851
26.7	Partial Generic Segmentation Offload	851
26.8	SCTP acceleration with GSO	852
27	Scaling in the Linux Networking Stack	853
27.1	Introduction	853
27.2	RSS: Receive Side Scaling	853
27.3	RPS: Receive Packet Steering	855
27.4	RFS: Receive Flow Steering	857
27.5	Accelerated RFS	859
27.6	XPS: Transmit Packet Steering	860
27.7	Per TX Queue rate limitation	862
27.8	Further Information	862
28	Kernel TLS	863
28.1	Overview	863

28.2 User interface	863
28.3 Statistics	867
29 Kernel TLS offload	869
29.1 Kernel TLS operation	869
29.2 Device configuration	870
29.3 Normal operation	871
29.4 Resync handling	873
29.5 Error handling	875
29.6 Performance metrics	876
29.7 Statistics	877
29.8 Notable corner cases, exceptions and additional requirements	878
30 Linux NFC subsystem	881
30.1 Architecture overview	881
30.2 Device Driver Interface	882
30.3 Userspace interface	882
31 Netdev private dataroom for 6lowpan interfaces	885
32 6pack Protocol	887
32.1 1. What is 6pack, and what are the advantages to KISS?	887
32.2 2. Who has developed the 6pack protocol?	888
32.3 3. Where can I get the latest version of 6pack for Linux?	888
32.4 4. Preparing the TNC for 6pack operation	888
32.5 5. Building and installing the 6pack driver	889
32.6 How to turn on 6pack support:	889
32.7 6. Known problems	890
33 ARCnet Hardware	891
33.1 Introduction to ARCnet	891
33.2 Cabling ARCnet Networks	892
33.3 Setting the Jumpers	896
33.4 Unclassified Stuff	899
33.5 Standard Microsystems Corp (SMC)	900
33.6 Possibly SMC	911
33.7 PureData Corp	913
33.8 CNet Technology Inc. (8-bit cards)	916
33.9 CNet Technology Inc. (16-bit cards)	920
33.10 Lantech	924
33.11 Acer	926
33.12 Datapoint?	929
33.13 Topware	932
33.14 Thomas-Conrad	935
33.15 Waterloo Microsystems Inc. ??	937
33.16 No Name	939
33.17 Tiara	953
33.18 Other Cards	955
34 ARCnet	957
34.1 Where do I discuss these drivers?	958
34.2 Other Drivers and Info	958

34.3	Installing the Driver	959
34.4	Loadable Module Support	960
34.5	Using the Driver	961
34.6	Multiple Cards in One Computer	961
34.7	How do I get it to work with...?	961
34.8	Using Multiprotocol ARCnet	963
34.9	It works: what now?	966
34.10	It doesn' t work: what now?	966
34.11	I want to send money: what now?	967
35	ATM	969
36	AX.25	971
37	Linux Ethernet Bonding Driver HOWTO	973
37.1	Introduction	973
37.2	1. Bonding Driver Installation	974
37.3	2. Bonding Driver Options	974
37.4	3. Configuring Bonding Devices	987
37.5	4 Querying Bonding Configuration	1001
37.6	5. Switch Configuration	1003
37.7	6. 802.1q VLAN Support	1003
37.8	7. Link Monitoring	1004
37.9	8. Potential Sources of Trouble	1005
37.10	9. SNMP agents	1007
37.11	10. Promiscuous mode	1008
37.12	11. Configuring Bonding for High Availability	1008
37.13	12. Configuring Bonding for Maximum Throughput	1010
37.14	13. Switch Behavior Issues	1015
37.15	14. Hardware Specific Considerations	1017
37.16	15. Frequently Asked Questions	1018
37.17	16. Resources and Links	1020
38	cdc_mbim - Driver for CDC MBIM Mobile Broadband modems	1021
38.1	Command Line Parameters	1021
38.2	Basic usage	1022
38.3	MBIM control channel userspace ABI	1022
38.4	MBIM data channel userspace ABI	1024
38.5	References	1027
39	DCCP protocol	1029
39.1	Introduction	1029
39.2	Missing features	1029
39.3	Socket options	1030
39.4	Sysctl variables	1031
39.5	IOCTLs	1032
39.6	Other tunables	1032
39.7	Notes	1033
40	DCTCP (DataCenter TCP)	1035
41	DNS Resolver Module	1037
41.1	Overview	1037

41.2	Compilation	1037
41.3	Setting up	1037
41.4	Usage	1038
41.5	Reading DNS Keys from Userspace	1039
41.6	Mechanism	1039
41.7	Debugging	1040
42	Softnet Driver Issues	1041
43	EQL Driver: Serial IP Load Balancing HOWTO	1043
43.1	1. Introduction	1043
43.2	2. Kernel Configuration	1044
43.3	3. Network Configuration	1044
43.4	4. About the Slave Scheduler Algorithm	1046
43.5	5. Testers' Reports	1047
44	LC-trie implementation notes	1051
44.1	Node types	1051
44.2	A few concepts explained	1051
44.3	Comments	1052
44.4	Locking	1053
44.5	Main lookup mechanism	1053
45	Linux Socket Filtering aka Berkeley Packet Filter (BPF)	1055
45.1	Introduction	1055
45.2	Structure	1056
45.3	Example	1056
45.4	BPF engine and instruction set	1058
45.5	JIT compiler	1064
45.6	BPF kernel internals	1066
45.7	eBPF opcode encoding	1071
45.8	eBPF verifier	1075
45.9	Register value tracking	1077
45.10	Direct packet access	1079
45.11	eBPF maps	1080
45.12	Pruning	1081
45.13	Understanding eBPF verifier messages	1081
45.14	Testing	1086
45.15	Misc	1086
45.16	Written by	1086
46	Frame Relay (FR)	1087
47	Generic HDLC layer	1089
47.1	Board-specific issues	1091
48	Generic Netlink	1093
49	Generic networking statistics for netlink users	1095
49.1	Collecting:	1095
49.2	Export to userspace (Dump):	1095
49.3	TCA_STATS/TCA_XSTATS backward compatibility:	1096
49.4	Locking:	1096

49.5 Rate Estimator:	1096
49.6 Authors:	1097
50 The Linux kernel GTP tunneling module	1099
50.1 What is GTP	1099
50.2 The Linux GTP tunnelling module	1100
50.3 Userspace Programs with Linux Kernel GTP-U support	1100
50.4 Userspace Library / Command Line Utilities	1100
50.5 Protocol Versions	1101
50.6 IPv6	1101
50.7 Mailing List	1101
50.8 Issue Tracker	1101
50.9 History / Acknowledgements	1101
50.10 Architectural Details	1102
50.11 APN vs. Network Device	1102
51 Identifier Locator Addressing (ILA)	1105
51.1 Introduction	1105
51.2 ILA terminology	1105
51.3 Operation	1106
51.4 Transport checksum handling	1107
51.5 Identifier types	1107
51.6 Identifier formats	1108
51.7 Configuration	1109
51.8 Some examples	1109
52 AppleTalk-IP Decapsulation and AppleTalk-IP Encapsulation	1111
52.1 Introduction	1111
52.2 Common Uses of ipddp.c	1112
52.3 Further Assistance	1112
53 IP dynamic address hack-port v0.03	1113
54 IPsec	1115
55 IP Sysctl	1117
55.1 /proc/sys/net/ipv4/* Variables	1117
55.2 INET peer storage	1120
55.3 TCP variables	1121
55.4 UDP variables	1132
55.5 RAW variables	1133
55.6 CIPSOv4 Variables	1133
55.7 IP Variables	1134
55.8 /proc/sys/net/ipv6/* Variables	1143
55.9 icmp/*:	1155
55.10 /proc/sys/net/bridge/* Variables:	1156
55.11 /proc/sys/net/sctp/* Variables:	1157
55.12 /proc/sys/net/core/*	1161
55.13 /proc/sys/net/unix/*	1161
56 IPv6	1163
57 IPVLAN Driver HOWTO	1165

57.1 1. Introduction:	1165
57.2 2. Building and Installation:	1165
57.3 3. Configuration:	1165
57.4 4. Operating modes:	1166
57.5 5. Mode flags:	1167
57.6 6. What to choose (macvlan vs. ipvlan)?	1167
57.7 6. Example configuration:	1168
58 IPvs-sysctl	1169
58.1 /proc/sys/net/ipv4/vs/* Variables:	1169
59 Kernel Connection Multiplexor	1175
59.1 KCM sockets	1175
59.2 Multiplexor	1176
59.3 TCP sockets & Psocks	1176
59.4 Connected mode semantics	1176
59.5 Socket types	1176
59.6 User interface	1177
59.7 Use in applications	1179
60 L2TP	1181
60.1 Overview	1181
60.2 L2TP APIs	1182
60.3 Internal Implementation	1190
60.4 Miscellaneous	1192
61 The Linux LAPB Module Interface	1195
61.1 Structures	1195
61.2 LAPB Initialisation Structure	1195
61.3 LAPB Parameter Structure	1196
61.4 Functions	1197
61.5 Callbacks	1199
62 How to use packet injection with mac80211	1201
63 MPLS Sysfs variables	1203
63.1 /proc/sys/net/mpls/* Variables:	1203
64 HOWTO for multiqueue network device support	1205
64.1 Section 1: Base driver requirements for implementing multiqueue support	1205
64.2 Section 2: Qdisc support for multiqueue devices	1205
64.3 Section 3: Brief howto using MULTIQ for multiqueue devices	1206
65 Netconsole	1207
65.1 Introduction:	1207
65.2 Sender and receiver configuration:	1207
65.3 Dynamic reconfiguration:	1208
65.4 Extended console:	1210
65.5 Miscellaneous notes:	1210
66 Netdev features mess and how to get out from it alive	1213
66.1 Part I: Feature sets	1213

66.2 Part II: Controlling enabled features	1214
66.3 Part III: Implementation hints	1214
66.4 Part IV: Features	1215
67 Network Devices, the Kernel, and You!	1217
67.1 Introduction	1217
67.2 struct net_device lifetime rules	1217
67.3 MTU	1220
67.4 struct net_device synchronization rules	1221
67.5 struct napi_struct synchronization rules	1221
68 Netfilter Sysfs variables	1223
68.1 /proc/sys/net/netfilter/* Variables:	1223
69 NETIF Msg Level	1225
69.1 History	1225
70 Netfilter Conntrack Sysfs variables	1227
70.1 /proc/sys/net/netfilter/nf_conntrack_* Variables:	1227
71 Netfilter's flowtable infrastructure	1231
71.1 Overview	1231
71.2 Example configuration	1233
71.3 More reading	1233
72 Open vSwitch datapath developer documentation	1235
72.1 Flow key compatibility	1235
72.2 Flow key format	1236
72.3 Wildcarded flow key format	1236
72.4 Unique flow identifiers	1237
72.5 Basic rule for evolving flow keys	1237
72.6 Handling malformed packets	1238
72.7 Other rules	1239
73 Operational States	1241
73.1 1. Introduction	1241
73.2 2. Querying from userspace	1241
73.3 3. Kernel driver API	1242
73.4 4. Setting from userspace	1243
74 Packet MMAP	1245
74.1 Abstract	1245
74.2 Why use PACKET_MMAP	1245
74.3 How to use mmap() to improve capture process	1246
74.4 How to use mmap() directly to improve capture process	1246
74.5 How to use mmap() directly to improve transmission process	1247
74.6 PACKET_MMAP settings	1249
74.7 PACKET_MMAP setting constraints	1250
74.8 PACKET_MMAP buffer size calculator	1251
74.9 What TPACKET versions are available and when to use them?	1255
74.10 AF_PACKET fanout mode	1257
74.11 AF_PACKET TPACKET_V3 example	1260
74.12 PACKET_QDISC_BYPASS	1265

74.13	PACKET_TIMESTAMP	1266
74.14	Miscellaneous bits	1267
74.15	THANKS	1267
75	Linux Phonet protocol family	1269
75.1	Introduction	1269
75.2	Packets format	1269
75.3	Link layer	1270
75.4	Network layer	1270
75.5	Low-level datagram protocol	1270
75.6	Resource subscription	1271
75.7	Phonet Pipe protocol	1271
75.8	Authors	1273
76	HOWTO for the linux packet generator	1275
76.1	Tuning NIC for max performance	1275
76.2	Kernel threads	1276
76.3	Viewing devices	1276
76.4	Configuring devices	1277
76.5	Sample scripts	1280
76.6	Interrupt affinity	1281
76.7	Enable IPsec	1281
76.8	Current commands and configuration options	1281
77	PLIP: The Parallel Line Internet Protocol Device	1285
77.1	PLIP Introduction	1285
77.2	PLIP driver details	1286
77.3	PLIP hardware interconnection	1287
78	PPP Generic Driver and Channel Interface	1291
78.1	PPP channel API	1292
78.2	Buffering and flow control	1293
78.3	SMP safety	1294
78.4	Interface to pppd	1295
79	The proc/net/tcp and proc/net/tcp6 variables	1299
80	How to use radiotap headers	1301
80.1	Pointer to the radiotap include file	1301
80.2	Structure of the header	1301
80.3	Requirements for arguments	1302
80.4	Example valid radiotap header	1302
80.5	Using the Radiotap Parser	1303
81	Overview	1305
82	RDS Architecture	1307
83	Socket Interface	1309
84	RDMA for RDS	1311
85	Congestion Notifications	1313

86 RDS Protocol	1315
87 RDS Transport Layer	1317
88 RDS Kernel Structures	1319
89 Connection management	1321
90 The send path	1323
91 The recv path	1325
92 Multipath RDS (mprds)	1327
93 Linux wireless regulatory documentation	1329
93.1 Keeping regulatory domains in userspace	1329
93.2 How to get regulatory domains to the kernel	1329
93.3 How to get regulatory domains to the kernel (old CRDA solution) . .	1329
93.4 Who asks for regulatory domains?	1330
93.5 Example code - drivers hinting an alpha2:	1331
93.6 Example code - drivers providing a built in regulatory domain: . . .	1332
93.7 Statically compiled regulatory database	1333
94 RxRPC Network Protocol	1335
94.1 Overview	1335
94.2 RxRPC Protocol Summary	1336
94.3 AF_RXRPC Driver Model	1338
94.4 Control Messages	1340
95 SOCKET OPTIONS	1345
96 SECURITY	1347
97 EXAMPLE CLIENT USAGE	1349
97.1 Example Server Usage	1351
97.2 AF_RXRPC Kernel Interface	1353
97.3 Configurable Parameters	1359
98 Linux Kernel SCTP	1361
98.1 Caveats	1361
99 LSM/SeLinux secid	1363
100 seg6 Sysfs variables	1365
100.1/proc/sys/net/conf/<iface>/seg6_* variables:	1365
101 Interface statistics	1367
101.1 Overview	1367
101.2 APIs	1368
101.3 struct rtnl_link_stats64	1369
101.4 Notes for driver authors	1373
102 Stream Parser (strparser)	1375
102.1 Introduction	1375

102.2	Interface	1375
102.3	Functions	1375
102.4	Callbacks	1377
102.5	Statistics	1378
102.6	Message assembly limits	1379
102.7	Author	1379
103	Ethernet switch device driver model (switchdev)	1381
103.1	Include Files	1382
103.2	Configuration	1382
103.3	Switch Ports	1382
103.4	L2 Forwarding Offload	1384
103.5	L3 Routing Offload	1387
104	sysfs tagging	1389
105	TC Actions - Environmental Rules	1391
106	Thin-streams and TCP	1393
106.1	References	1394
107	Team	1395
108	Timestamping	1397
108.1	Control Interfaces	1397
108.2	Data Interfaces	1403
108.3	Hardware Timestamping configuration: SIOCShwtstamp and SIOCghwtstamp	1405
109	Transparent proxy support	1411
109.1	Making non-local sockets work	1411
109.2	Redirecting traffic	1412
109.3	Iptables and nf_tables extensions	1412
109.4	Application support	1413
110	Universal TUN/TAP device driver	1415
110.1	Description	1415
110.2	Configuration	1416
110.3	Program interface	1416
110.4	Universal TUN/TAP device driver Frequently Asked Question	1419
111	The UDP-Lite protocol (RFC 3828)	1421
111.1	Applications	1421
111.2	Programming API	1422
111.3	Header Files	1423
111.4	Kernel Behaviour with Regards to the Various Socket Options	1423
111.5	UDP-Lite Runtime Statistics and their Meaning	1425
111.6	IPtables	1426
111.7	Maintainer Address	1427
112	Virtual Routing and Forwarding (VRF)	1429
112.1	The VRF Device	1429
112.2	Using iproute2 for VRFs	1431

113	Virtual eXtensible Local Area Networking documentation	1439
114	Packet Layer to Device Driver	1443
115	Device Driver to Packet Layer	1445
115.1	Possible Problems	1445
116	Linux X.25 Project	1447
117	XFRM device - offloading the IPsec computations	1449
117.1	Overview	1449
117.2	Callbacks to implement	1449
117.3	Flow	1450
118	XFRM proc - /proc/net/xfrm_* files	1453
118.1	Transformation Statistics	1453
119	XFRM	1455
119.1	1) Message Structure	1455
119.2	2) TLVS reflect the different parameters:	1456
119.3	3) Default configurations for the parameters:	1457
119.4	4) Message types	1457
119.5	5) Exceptions to threshold settings	1458
120	XFRM Syscall	1459
120.1	/proc/sys/net/core/xfrm_* Variables:	1459

Contents:

NETDEV FAQ

1.1 Q: What is netdev?

A: It is a mailing list for all network-related Linux stuff. This includes anything found under `net/` (i.e. core code like IPv6) and `drivers/net` (i.e. hardware specific drivers) in the Linux source tree.

Note that some subsystems (e.g. wireless drivers) which have a high volume of traffic have their own specific mailing lists.

The netdev list is managed (like many other Linux mailing lists) through VGER (<http://vger.kernel.org/>) and archives can be found below:

- <http://marc.info/?l=linux-netdev>
- <http://www.spinics.net/lists/netdev/>

Aside from subsystems like that mentioned above, all network-related Linux development (i.e. RFC, review, comments, etc.) takes place on netdev.

1.2 Q: How do the changes posted to netdev make their way into Linux?

A: There are always two trees (git repositories) in play. Both are driven by David Miller, the main network maintainer. There is the `net` tree, and the `net-next` tree. As you can probably guess from the names, the `net` tree is for fixes to existing code already in the mainline tree from Linus, and `net-next` is where the new code goes for the future release. You can find the trees here:

- <https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net.git>
- <https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git>

1.3 Q: How often do changes from these trees make it to the mainline Linus tree?

A: To understand this, you need to know a bit of background information on the cadence of Linux development. Each new release starts off with a two week “merge window” where the main maintainers feed their new stuff to Linus for merging into the mainline tree. After the two weeks, the merge window is closed, and it is called/tagged -rc1. No new features get mainlined after this - only fixes to the rc1 content are expected. After roughly a week of collecting fixes to the rc1 content, rc2 is released. This repeats on a roughly weekly basis until rc7 (typically; sometimes rc6 if things are quiet, or rc8 if things are in a state of churn), and a week after the last vX.Y-rcN was done, the official vX.Y is released.

Relating that to netdev: At the beginning of the 2-week merge window, the net-next tree will be closed - no new changes/features. The accumulated new content of the past ~10 weeks will be passed onto mainline/Linus via a pull request for vX.Y - at the same time, the net tree will start accumulating fixes for this pulled content relating to vX.Y

An announcement indicating when net-next has been closed is usually sent to netdev, but knowing the above, you can predict that in advance.

IMPORTANT: Do not send new net-next content to netdev during the period during which net-next tree is closed.

Shortly after the two weeks have passed (and vX.Y-rc1 is released), the tree for net-next reopens to collect content for the next (vX.Y+1) release.

If you aren't subscribed to netdev and/or are simply unsure if net-next has re-opened yet, simply check the net-next git repository link above for any new networking-related commits. You may also check the following website for the current status:

<http://vger.kernel.org/~davem/net-next.html>

The net tree continues to collect fixes for the vX.Y content, and is fed back to Linus at regular (~weekly) intervals. Meaning that the focus for net is on stabilization and bug fixes.

Finally, the vX.Y gets released, and the whole cycle starts over.

Q: So where are we now in this cycle?

Load the mainline (Linus) page here:

<https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git>

and note the top of the “tags” section. If it is rc1, it is early in the dev cycle. If it was tagged rc7 a week ago, then a release is probably imminent.

1.4 Q: How do I indicate which tree (net vs. net-next) my patch should be in?

A: Firstly, think whether you have a bug fix or new “next-like” content. Then once decided, assuming that you use git, use the prefix flag, i.e.

```
git format-patch --subject-prefix='PATCH net-next' start..finish
```

Use net instead of net-next (always lower case) in the above for bug-fix net content. If you don't use git, then note the only magic in the above is just the subject text of the outgoing e-mail, and you can manually change it yourself with whatever MUA you are comfortable with.

1.5 Q: I sent a patch and I'm wondering what happened to it?

Q: How can I tell whether it got merged? A: Start by looking at the main patchworks queue for netdev:

<https://patchwork.kernel.org/project/netdevbpf/list/>

The “State” field will tell you exactly where things are at with your patch.

1.6 Q: The above only says “Under Review” . How can I find out more?

A: Generally speaking, the patches get triaged quickly (in less than 48h). So be patient. Asking the maintainer for status updates on your patch is a good way to ensure your patch is ignored or pushed to the bottom of the priority list.

1.7 Q: I submitted multiple versions of the patch series

Q: should I directly update patchwork for the previous versions of these patch series? A: No, please don't interfere with the patch status on patchwork, leave it to the maintainer to figure out what is the most recent and current version that should be applied. If there is any doubt, the maintainer will reply and ask what should be done.

1.8 Q: I made changes to only a few patches in a patch series should I resend only those changed?

A: No, please resend the entire patch series and make sure you do number your patches such that it is clear this is the latest and greatest set of patches that can be applied.

1.9 Q: I submitted multiple versions of a patch series and it looks like a version other than the last one has been accepted, what should I do?

A: There is no revert possible, once it is pushed out, it stays like that. Please send incremental versions on top of what has been merged in order to fix the patches the way they would look like if your latest patch series was to be merged.

1.10 Q: Are there special rules regarding stable submissions on netdev?

While it used to be the case that netdev submissions were not supposed to carry explicit CC: `stable@vger.kernel.org` tags that is no longer the case today. Please follow the standard stable rules in `Documentation/process/stable-kernel-rules.rst`, and make sure you include appropriate Fixes tags!

1.11 Q: Is the comment style convention different for the networking content?

A: Yes, in a largely trivial way. Instead of this:

```
/*
 * foobar blah blah blah
 * another line of text
 */
```

it is requested that you make it look like this:

```
/* foobar blah blah blah
 * another line of text
 */
```


1.12 Q: I am working in existing code that has the former comment style and not the latter.

Q: Should I submit new code in the former style or the latter? A: Make it the latter style, so that eventually all code in the domain of netdev is of this format.

1.13 Q: I found a bug that might have possible security implications or similar.

Q: Should I mail the main netdev maintainer off-list? A: No. The current netdev maintainer has consistently requested that people use the mailing lists and not reach out directly. If you aren't OK with that, then perhaps consider mailing security@kernel.org or reading about <http://oss-security.openwall.org/wiki/ mailing-lists/distros> as possible alternative mechanisms.

1.14 Q: What level of testing is expected before I submit my change?

A: If your changes are against net-next, the expectation is that you have tested by layering your changes on top of net-next. Ideally you will have done run-time testing specific to your change, but at a minimum, your changes should survive an allyesconfig and an allmodconfig build without new warnings or failures.

1.15 Q: How do I post corresponding changes to user space components?

A: User space code exercising kernel features should be posted alongside kernel patches. This gives reviewers a chance to see how any new interface is used and how well it works.

When user space tools reside in the kernel repo itself all changes should generally come as one series. If series becomes too large or the user space project is not reviewed on netdev include a link to a public repo where user space patches can be seen.

In case user space tooling lives in a separate repository but is reviewed on netdev (e.g. patches to *iproute2* tools) kernel and user space patches should form separate series (threads) when posted to the mailing list, e.g.:

```
[PATCH net-next 0/3] net: some feature cover letter
└─ [PATCH net-next 1/3] net: some feature prep
└─ [PATCH net-next 2/3] net: some feature do it
└─ [PATCH net-next 3/3] selftest: net: some feature

[PATCH iproute2-next] ip: add support for some feature
```

Posting as one thread is discouraged because it confuses patchwork (as of patchwork 2.2.2).

1.16 Q: Any other tips to help ensure my net/net-next patch gets OK' d?

A: Attention to detail. Re-read your own work as if you were the reviewer. You can start with using `checkpatch.pl`, perhaps even with the `--strict` flag. But do not be mindlessly robotic in doing so. If your change is a bug fix, make sure your commit log indicates the end-user visible symptom, the underlying reason as to why it happens, and then if necessary, explain why the fix proposed is the best way to get things done. Don' t mangle whitespace, and as is common, don' t mis-indent function arguments that span multiple lines. If it is your first patch, mail it to yourself so you can test apply it to an unpatched tree to confirm infrastructure didn' t mangle it.

Finally, go back and read `Documentation/process/submitting-patches.rst` to be sure you are not repeating some common mistake documented there.

2.1 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`s `XPDP_RX_RING` and `XPDP_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` `XPDP_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.

2.2 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/>.

2.2.1 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` socket is 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.

2.2.2 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 mmapmed 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 $\log_2(2048)$ 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.

2.3 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.

2.4 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.

2.5 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.

2.5.1 XDP_COPY and XDP_ZERO_COPY 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_ZERO_COPY` flag will force the socket into zero-copy mode or fail.

2.5.2 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));
} xsk_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(&xsk_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.

2.5.3 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.

2.5.4 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.

2.5.5 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.

2.5.6 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.

2.5.7 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
↳*/
};
```

2.5.8 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.

2.6 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 {
```

(continues on next page)

(continued from previous page)

```
// __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.

2.7 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].

2.8 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 you 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-
```

(continues on next page)

(continued from previous page)

```
↪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.

2.9 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.

3.1 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.

3.2 Usage

1) Device creation & deletion

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

This creates a bareudp tunnel device which tunnels L3 traffic with ether-type 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.

- b) `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.

- a) `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.

- b) `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).

4.1 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.

4.2 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
```

4.3 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
```

4.4 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/>

4.5 Contact

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

IRC:

#batman on irc.freenode.org

Mailing-list:

b.a.t.m.a.n@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

5.1 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.

5.2 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.

5.3 SocketCAN Concept

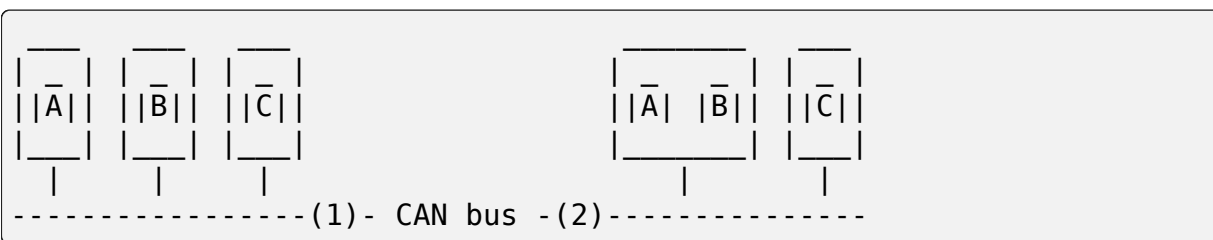
As described in [Motivation / Why Using the Socket API](#) the main goal of SocketCAN is to provide a socket interface to user space applications which builds upon the Linux network layer. In contrast to the commonly known TCP/IP and ethernet networking, the CAN bus is a broadcast-only(!) medium that 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.

5.3.1 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.

5.3.2 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

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

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` \(SOCK_RAW\)](#).

5.3.3 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” .

5.4 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 basic CAN frame structure and the sockaddr structure are defined in include/linux/can.h:

```
struct can_frame {
    canid_t can_id; /* 32 bit CAN_ID + EFF/RTR/ERR flags */
    __u8 can_dlc; /* frame payload length in byte (0 .. 8) */
    __u8 __pad; /* padding */
    __u8 __res0; /* reserved / padding */
    __u8 __res1; /* reserved / padding */
    __u8 data[8] __attribute__((aligned(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;

        /* 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;
```

(continues on next page)

(continued from previous page)

```

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 (legacy) 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.can_dlc` 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  => 'legacy'
↳ CAN frame
#define CANFD_MTU (sizeof(struct canfd_frame)) == 72  => CAN FD
↳ frame
```

5.4.1 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;
```

(continues on next page)

(continued from previous page)

```

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));
```

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, &cfd, 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 legacy 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 legacy 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 legacy 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 filter set 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.

5.4.2 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_COUNT EVT 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_COUNT EVT:

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]).

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 periode 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)
```

(continues on next page)

(continued from previous page)

```

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]) = 0xFF00000000000000ULL; /* MUX mask */
U64_DATA(&msg.frame[1]) = 0x01000000000000FFULL; /* data mask (MUX_
↪0x01) */
U64_DATA(&msg.frame[2]) = 0x0200FFFF000000FFULL; /* data mask (MUX_
↪0x02) */
U64_DATA(&msg.frame[3]) = 0x330000FFFFFFFF0003ULL; /* 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.

5.4.3 Connected Transport Protocols (SOCK_SEQPACKET)

(to be written)

5.4.4 Unconnected Transport Protocols (SOCK_DGRAM)

(to be written)

5.5 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](#)).

5.5.1 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)

5.5.2 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/rcvlist_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:

```
rcvlist_all - list for unfiltered entries (no filter operations)
rcvlist_eff - list for single extended frame (EFF) entries
rcvlist_err - list for error message frames masks
rcvlist_fil - list for mask/value filters
rcvlist_inv - list for mask/value filters (inverse semantic)
rcvlist_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 the SocketCAN core version and the ABI version
```

5.5.3 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 .

5.6 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:

5.6.1 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) -> legacy 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`.

5.6.2 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);
```

5.6.3 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 ...

5.6.4 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':
```

(continues on next page)

(continued from previous page)

```
$ ip link add dev vcan42 type vcan
```

- Remove a (virtual CAN) network interface 'vcan42':

```
$ ip link del vcan42
```

5.6.5 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 } ]

    [ restart-ms TIME-MS ]
    [ restart ]

Where: BITRATE      := { 1..1000000 }
```

(continues on next page)

(continued from previous page)

```

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
sjal000: 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](#)).

5.6.6 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.can_dlc` 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 legacy 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_dlc2len()` and `can_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)    => 'legacy' 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 legacy 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
```

5.6.7 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.

5.7 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].

5.8 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`.

6.1 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

6.2 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.

6.2.1 Setup Packet

bmRequestType	Direction Vendor (Interface or Device)
bRequest	Command Number
wValue	Subcommand Number (16 Bit) or 0 if not used
wIndex	USB Interface Index (0 for device commands)
wLength	<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.

6.2.2 Error Handling

The device indicates failed control commands by stalling the pipe.

6.2.3 Device Commands

UCAN_DEVICE_GET_FW_STRING

Dev2Host; optional

Request the device firmware string.

6.2.4 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.

6.2.5 Allowed interface commands

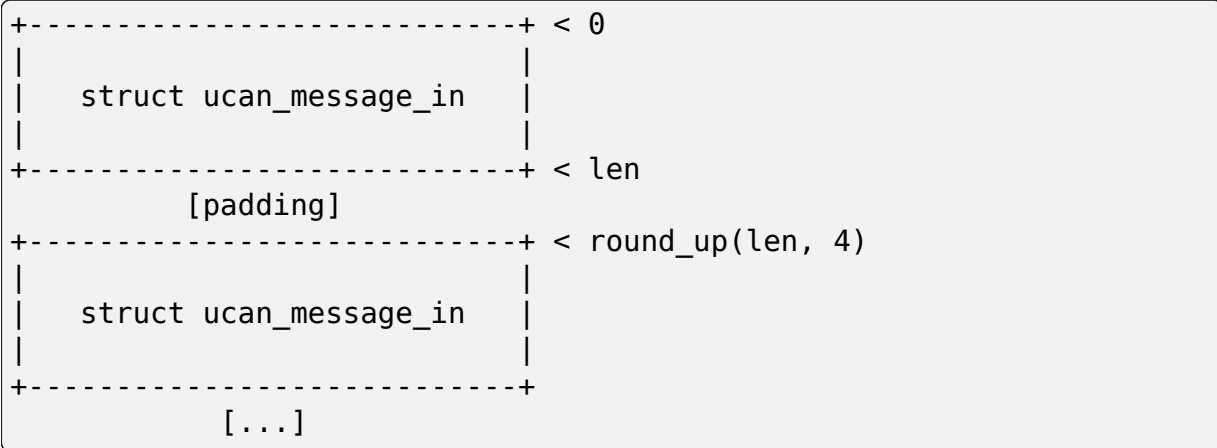
Legal Device State	Command	New Device State
stopped	SET_BITTIMING	stopped
stopped	START	started
started	STOP or RESET	stopped
stopped	STOP or RESET	stopped
started	RESTART	started
any	GET	<i>no change</i>

6.3 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).

6.3.1 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:



6.3.2 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>`.

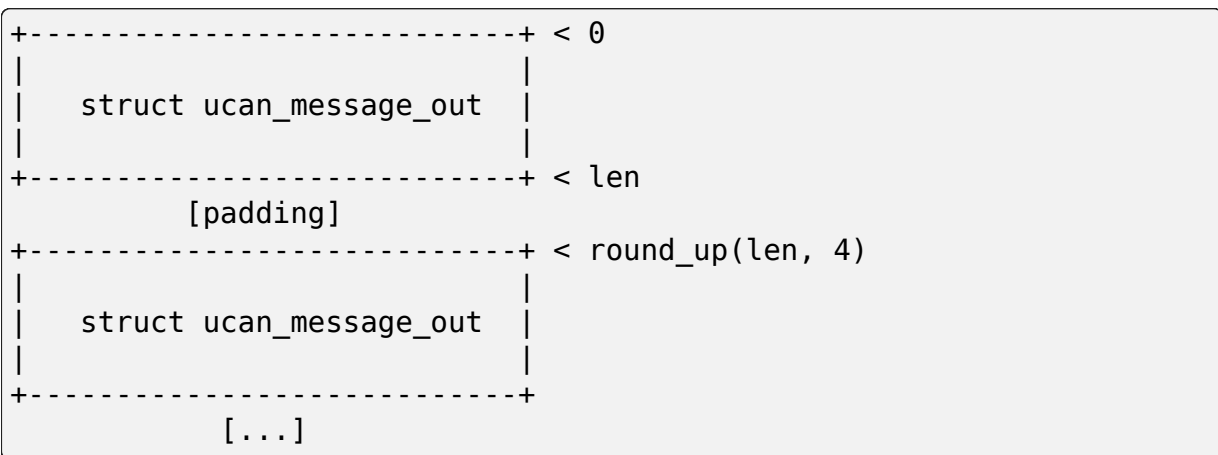
The echo-id identifies the frame from (echos the id from a previous `UCAN_OUT_TX` message). The flag indicates the result of the transmission. Whereas a set Bit 0 indicates success. All other bits are reserved and set to zero.

6.3.3 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](#))

6.4 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](#).



6.4.1 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)

6.4.2 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.

6.5 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.

6.5.1 Bus OFF

- The device does not recover from bus of 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.

6.6 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:

7.1 AppleTalk Device Drivers

Contents:

7.1.1 The COPS LocalTalk Linux driver (cops.c)

By Jay Schulist <jschlst@samba.org>

This driver has two modes and they are: Dayna mode and Tangent mode. Each mode corresponds with the type of card. It has been found that there are 2 main types of cards and all other cards are the same and just have different names or only have minor differences such as more IO ports. As this driver is tested it will become more clear exactly what cards are supported.

Right now these cards are known to work with the COPS driver. The LT-200 cards work in a somewhat more limited capacity than the DL200 cards, which work very well and are in use by many people.

TANGENT driver mode:

- Tangent ATB-II, Novell NL-1000, Daystar Digital LT-200

DAYNA driver mode:

- Dayna DL2000/DaynaTalk PC (Half Length), COPS LT-95,
- Farallon PhoneNET PC III, Farallon PhoneNET PC II

Other cards possibly supported mode unknown though:

- Dayna DL2000 (Full length)

The COPS driver defaults to using Dayna mode. To change the driver's mode if you built a driver with dual support use `board_type=1` or `board_type=2` for Dayna or Tangent with `insmod`.

Operation/loading of the driver

Use modprobe like this: `/sbin/modprobe cops.o (IO #) (IRQ #)` If you do not specify any options the driver will try and use the `IO = 0x240`, `IRQ = 5`. As of right now I would only use `IRQ 5` for the card, if autoprobng.

To load multiple COPS driver Localtalk cards you can do one of the following:

```
insmod cops io=0x240 irq=5
insmod -o cops2 cops io=0x260 irq=3
```

Or in `lilo.conf` put something like this:

```
append="ether=5,0x240,lt0 ether=3,0x260,lt1"
```

Then bring up the interface with `ifconfig`. It will look something like this:

```
lt0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-F7-00-00-
↳00-00-00-00-00-00
          inet addr:192.168.1.2  Bcast:192.168.1.255  Mask:255.255.
↳255.0
          UP BROADCAST RUNNING NOARP MULTICAST  MTU:600  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
↳coll:0
```

Netatalk Configuration

You will need to configure `atalkd` with something like the following to make it work with the `cops.c` driver.

- For single LTalk card use:

```
dummy -seed -phase 2 -net 2000 -addr 2000.10 -zone "1033"
lt0 -seed -phase 1 -net 1000 -addr 1000.50 -zone "1033"
```

- For multiple cards, Ethernet and LocalTalk:

```
eth0 -seed -phase 2 -net 3000 -addr 3000.20 -zone "1033"
lt0 -seed -phase 1 -net 1000 -addr 1000.50 -zone "1033"
```

- For multiple LocalTalk cards, and an Ethernet card.
- Order seems to matter here, Ethernet last:

```
lt0 -seed -phase 1 -net 1000 -addr 1000.10 -zone "LocalTalk1"
lt1 -seed -phase 1 -net 2000 -addr 2000.20 -zone "LocalTalk2"
eth0 -seed -phase 2 -net 3000 -addr 3000.30 -zone "EtherTalk"
```


7.1.2 LTPC Driver

This is the ALPHA version of the ltpc driver.

In order to use it, you will need at least version 1.3.3 of the netatalk package, and the Apple or Farallon LocalTalk PC card. There are a number of different LocalTalk cards for the PC; this driver applies only to the one with the 65c02 processor chip on it.

To include it in the kernel, select the CONFIG_LTPC switch in the configuration dialog. You can also compile it as a module.

While the driver will attempt to autoprobe the I/O port address, IRQ line, and DMA channel of the card, this does not always work. For this reason, you should be prepared to supply these parameters yourself. (see “Card Configuration” below for how to determine or change the settings on your card)

When the driver is compiled into the kernel, you can add a line such as the following to your /etc/lilo.conf:

```
append="ltpc=0x240,9,1"
```

where the parameters (in order) are the port address, IRQ, and DMA channel. The second and third values can be omitted, in which case the driver will try to determine them itself.

If you load the driver as a module, you can pass the parameters “io=”, “irq=”, and “dma=” on the command line with insmod or modprobe, or add them as options in a configuration file in /etc/modprobe.d/ directory:

```
alias lt0 ltpc # autoload the module when the interface is
↳ configured
options ltpc io=0x240 irq=9 dma=1
```

Before starting up the netatalk demons (perhaps in rc.local), you need to add a line such as:

```
/sbin/ifconfig lt0 127.0.0.42
```

The address is unimportant - however, the card needs to be configured with ifconfig so that Netatalk can find it.

The appropriate netatalk configuration depends on whether you are attached to a network that includes AppleTalk routers or not. If, like me, you are simply connecting to your home Macintoshes and printers, you need to set up netatalk to “seed”. The way I do this is to have the lines:

```
dummy -seed -phase 2 -net 2000 -addr 2000.26 -zone "1033"
lt0 -seed -phase 1 -net 1033 -addr 1033.27 -zone "1033"
```

in my atalkd.conf. What is going on here is that I need to fool netatalk into thinking that there are two AppleTalk interfaces present; otherwise, it refuses to seed. This is a hack, and a more permanent solution would be to alter the netatalk code. Also, make sure you have the correct name for the dummy interface - If it’s compiled as a module, you will need to refer to it as “dummy0” or some such.

If you are attached to an extended AppleTalk network, with routers on it, then you don't need to fool around with this - the appropriate line in `atalkd.conf` is:

```
lt0 -phase 1
```

Card Configuration

The interrupts and so forth are configured via the dipswitch on the board. Set the switches so as not to conflict with other hardware.

Interrupts - set at most one. If none are set, the driver uses polled mode. Because the card was developed in the XT era, the original documentation refers to IRQ2. Since you'll be running this on an AT (or later) class machine, that really means IRQ9.

SW1	IRQ 4
SW2	IRQ 3
SW3	IRQ 9 (2 in original card documentation only applies to XT)

DMA - choose DMA 1 or 3, and set both corresponding switches.

SW4	DMA 3
SW5	DMA 1
SW6	DMA 3
SW7	DMA 1

I/O address - choose one.

SW8	220 / 240
-----	-----------

IP

Yes, it is possible to do IP over LocalTalk. However, you can't just treat the LocalTalk device like an ordinary Ethernet device, even if that's what it looks like to Netatalk.

Instead, you follow the same procedure as for doing IP in EtherTalk. See [AppleTalk-IP Decapsulation and AppleTalk-IP Encapsulation](#) for more information about the kernel driver and userspace tools needed.

Bugs

IRQ autoprobining often doesn't work on a cold boot. To get around this, either compile the driver as a module, or pass the parameters for the card to the kernel as described above.

Also, as usual, autoprobining is not recommended when you use the driver as a module. (though it usually works at boot time, at least)

Polled mode is *really* slow sometimes, but this seems to depend on the configuration of the network.

It may theoretically be possible to use two LTSPC cards in the same machine, but this is unsupported, so if you really want to do this, you'll probably have to hack the initialization code a bit.

Thanks

Thanks to Alan Cox for helpful discussions early on in this work, and to Denis Hainsworth for doing the bleeding-edge testing.

Bradford Johnson <bradford@math.umn.edu>

Updated 11/09/1998 by David Huggins-Daines <dhd@debian.org>

7.2 Asynchronous Transfer Mode (ATM) Device Drivers

Contents:

7.2.1 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:

- adsl_controller_version
- adsl_headend
- adsl_headend_environment
 - Information about the remote headend.

- `adsl_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)
```

7.2.2 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>.

7.2.3 ATM (i)Chip IA Linux Driver Source

READ ME FISRT

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.iphase.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.

- a. Login as root.
- b. Shut down the system and power off the system.
- c. Install one or more ATM adapters in the system.

- d. 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.
 - e. Power on and boot the system.
2. [Removed]
3. Rebuild kernel with ABR support
 - [a. and b. removed]
 - c. Reconfigure the kernel, choose the Interphase ia driver through "make menuconfig" or "make xconfig" .
 - d. Rebuild the kernel, loadable modules and the atm tools.
 - e. 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 RAM size	Rx RAM size	Tx RAM size	Rx Buf size	Tx Buf size	Rx buf cnt	Tx buf 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 IADebugFlag in iphase.h. The debug trace can be turn on through the insmod command line option, for example, “insmod iphase.o IADebugFlag=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
```

7.3 Cable Modem Device Drivers

Contents:

7.3.1 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, `pnpdump`, 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 "nobroadcast" 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 SURF-board 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

7.4 Cellular Modem Device Drivers

Contents:

7.4.1 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 (data / control)

MAP header has the same endianness of the IP packet.

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. Control packet is used for transport level flow control. Data packets are standard IP packets.

Reserved bits are usually zeroed out and to be ignored by receiver.

Padding is number of bytes to be added for 4 byte alignment if required by hardware.

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 (command specific):

Bit	0	1	2-7	8 - 15	
↪ 16 - 31					
Function	Command	Reserved	Pad	Multiplexer ID	
↪ Payload length					
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

c. 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 padding
...

3. Userspace configuration

rmnet userspace configuration is done through netlink library librmnetctl and command line utility rmnetcli. Utility is hosted in codeaurora forum git. The driver uses rtnl_link_ops for communication.

<https://source.codeaurora.org/quic/la/platform/vendor/qcom-opensource/dataservices/tree/rmnetctl>

7.5 Ethernet Device Drivers

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

Contents:

7.5.1 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 provide 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

7.5.2 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	10Mbps 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 zerocopy 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 the `admin-guide/reporting-bugs.rst` file.

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 [Fast Etherlink] (rev 74)
      Subsystem: 3Com Corporation: Unknown device
  ↳9200
      Flags: bus master, medium devsel, latency 32,
  ↳IRQ 19
```

(continues on next page)

(continued from previous page)

```
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
20: 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 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.

7.5.3 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 extendable 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 extendable 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 and TCP transmit segmentation offload (TSO). 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.h	Management communication layer. This layer is responsible for the handling all the management (admin) communication between the device and the driver.
ena_eth_c	Tx/Rx data path.
ena_admin	Definition of ENA management interface.
ena_eth_i	Definition of ENA data path interface.
ena_comn	Common definitions for ena_com layer.
ena_regs	Definition of ENA PCI memory-mapped (MMIO) registers.
ena_netde	Main Linux kernel driver.
ena_syfsf	Sysfs files.
ena_eth_t	ethtool callbacks.
ena_pci_i	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 involved in 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 of the device's health. The driver maintains a watchdog (WD) handler which, if fired, logs the current

state and statistics then resets and restarts the ENA device and driver. A Keep-Alive event is delivered by the device every second. The driver re-arms the WD upon reception of a Keep-Alive event. A missed Keep-Alive event causes the WD handler to fire.

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 128 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.

The Rx SQs support only the regular mode.

Note: 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 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)` `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.

SKB

The driver-allocated SKB for frames received from Rx handling using NAPI context. The allocation method depends on the size of the packet. If the frame length is larger than `rx_copybreak`, `napi_get_frags()` is used, otherwise `netdev_alloc_skb_ip_align()` is used, the buffer content is copied (by CPU) to the SKB, and the buffer is recycled.

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:

- TSO over IPv4/IPv6
- TSO with ECN
- 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_INDIIRECTION_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

`end_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 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_rx_pkt()`, an ENA communication layer function, which returns the number of descriptors used for a new unhandled packet, and zero if no new packet is found.
- Then it calls the `ena_clean_rx_irq()` function.
- `ena_eth_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, then allocates the new SKB structure and hooks the Rx buffer to the SKB 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()`.

7.5.4 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 statistics 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_runts” 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).

7.5.5 Marvell(Aquantia) AQtion 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 enpls0
```

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 enpls0:
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
```

(continues on next page)

(continued from previous page)

```
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-eprom-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
```

(continues on next page)

(continued from previous page)

```
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> msglvl <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> msglvl 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.

7.5.6 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” an 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
```

2. 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.

3. 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.

7.5.7 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                      *
```

(continues on next page)

(continued from previous page)

```
* 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:

- If an EEPROM is present, any specified command-line parameter will override the corresponding configuration value stored in EEPROM.
- The "io" parameter must be specified on the command-line.
- 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:

```
LILO 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			

(continues on next page)

(continued from previous page)

→display adapter		9	EGA/VGA
→(PS/2)		12	Mouse
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=au1     or
cs89x0_media=bnc
```

7.5.8 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><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_fc</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_fc	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_fc	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.																										
vlan=n	By default, the NIC operates at autosense. 1000mbps_fd and 1000mbps_hd types are only available for fiber adapter. 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.																										

7.5. Ethernet Device Drivers

tx_coalesce=n	Number of tx frame handled each interrupt. Set n > 1 can reduce the interrupts congestion usually lower per-
---------------	--

Configuration Script Sample

Here is a sample of a simple configuration script:

```
DEVICE=eth0
USERCTL=no
ONBOOT=yes
P00TPROT0=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.

7.5.9 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-s3c2410/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.

7.5.10 DEC EtherWORKS Ethernet De4x5 cards

Originally, this driver was written for the Digital Equipment Corporation series of EtherWORKS Ethernet cards:

- DE425 TP/COAX EISA
- DE434 TP PCI
- DE435 TP/COAX/AUI PCI
- DE450 TP/COAX/AUI PCI
- DE500 10/100 PCI Fasternet

but it will now attempt to support all cards which conform to the Digital Semiconductor SROM Specification. The driver currently recognises the following chips:

- DC21040 (no SROM)
- DC21041[A]

- DC21140[A]
- DC21142
- DC21143

So far the driver is known to work with the following cards:

- KINGSTON
- Linksys
- ZNYX342
- SMC8432
- SMC9332 (w/new SROM)
- ZNYX31[45]
- ZNYX346 10/100 4 port (can act as a 10/100 bridge!)

The driver has been tested on a relatively busy network using the DE425, DE434, DE435 and DE500 cards and benchmarked with 'ttcp' : it transferred 16M of data to a DECstation 5000/200 as follows:

	TCP		UDP		
	TX	RX	TX	RX	
DE425	1030k	997k	1170k	1128k	
DE434	1063k	995k	1170k	1125k	
DE435	1063k	995k	1170k	1125k	
DE500	1063k	998k	1170k	1125k	in 10Mb/s mode

All values are typical (in kBytes/sec) from a sample of 4 for each measurement. Their error is +/-20k on a quiet (private) network and also depend on what load the CPU has.

The ability to load this driver as a loadable module has been included and used extensively during the driver development (to save those long reboot sequences). Loadable module support under PCI and EISA has been achieved by letting the driver autoprobe as if it were compiled into the kernel. Do make sure you're not sharing interrupts with anything that cannot accommodate interrupt sharing!

To utilise this ability, you have to do 8 things:

- 0) have a copy of the loadable modules code installed on your system.
- 1) copy de4x5.c from the /linux/drivers/net directory to your favourite temporary directory.
- 2) for fixed autoprobos (not recommended), edit the source code near line 5594 to reflect the I/O address you're using, or assign these when loading by:

```
insmod de4x5 io=0xghh
```

where g = bus number
hh = device number

Note: autoprobing for modules is now supported by default. You may just use:

```
insmod de4x5
```

to load all available boards. For a specific board, still use the 'io=?' above.

- 3) compile de4x5.c, but include -DMODULE in the command line to ensure that the correct bits are compiled (see end of source code).
- 4) if you are wanting to add a new card, goto 5. Otherwise, recompile a kernel with the de4x5 configuration turned off and reboot.
- 5) insmod de4x5 [io=0xgghh]
- 6) run the net startup bits for your new eth?? interface(s) manually (usually /etc/rc.inet[12] at boot time).
- 7) enjoy!

To unload a module, turn off the associated interface(s) 'ifconfig eth?? down' then 'rmmod de4x5' .

Automedia detection is included so that in principle you can disconnect from, e.g. TP, reconnect to BNC and things will still work (after a pause while the driver figures out where its media went). My tests using ping showed that it appears to work...

By default, the driver will now autodetect any DECchip based card. Should you have a need to restrict the driver to DIGITAL only cards, you can compile with a DEC_ONLY define, or if loading as a module, use the 'dec_only=1' parameter.

I've changed the timing routines to use the kernel timer and scheduling functions so that the hangs and other assorted problems that occurred while autosensing the media should be gone. A bonus for the DC21040 auto media sense algorithm is that it can now use one that is more in line with the rest (the DC21040 chip doesn't have a hardware timer). The downside is the 1 'jiffies' (10ms) resolution.

IEEE 802.3u MII interface code has been added in anticipation that some products may use it in the future.

The SMC9332 card has a non-compliant SROM which needs fixing - I have patched this driver to detect it because the SROM format used complies to a previous DEC-STD format.

I have removed the buffer copies needed for receive on Intels. I cannot remove them for Alphas since the Tulip hardware only does long-word aligned DMA transfers and the Alphas get alignment traps with non longword aligned data copies (which makes them really slow). No comment.

I have added SROM decoding routines to make this driver work with any card that supports the Digital Semiconductor SROM spec. This will help

all cards running the dc2114x series chips in particular. Cards using the dc2104x chips should run correctly with the basic driver. I'm in debt to <mjacob@feral.com> for the testing and feedback that helped get this feature working. So far we have tested KINGSTON, SMC8432, SMC9332 (with the latest SROM complying with the SROM spec V3: their first was broken), ZNYX342 and LinkSys. ZNYX314 (dual 21041 MAC) and ZNYX 315 (quad 21041 MAC) cards also appear to work despite their incorrectly wired IRQs.

I have added a temporary fix for interrupt problems when some SCSI cards share the same interrupt as the DECchip based cards. The problem occurs because the SCSI card wants to grab the interrupt as a fast interrupt (runs the service routine with interrupts turned off) vs. this card which really needs to run the service routine with interrupts turned on. This driver will now add the interrupt service routine as a fast interrupt if it is bounced from the slow interrupt. THIS IS NOT A RECOMMENDED WAY TO RUN THE DRIVER and has been done for a limited time until people sort out their compatibility issues and the kernel interrupt service code is fixed. YOU SHOULD SEPARATE OUT THE FAST INTERRUPT CARDS FROM THE SLOW INTERRUPT CARDS to ensure that they do not run on the same interrupt. PCMCIA/CardBus is another can of worms...

Finally, I think I have really fixed the module loading problem with more than one DECchip based card. As a side effect, I don't mess with the device structure any more which means that if more than 1 card in 2.0.x is installed (4 in 2.1.x), the user will have to edit `linux/drivers/net/Space.c` to make room for them. Hence, module loading is the preferred way to use this driver, since it doesn't have this limitation.

Where SROM media detection is used and full duplex is specified in the SROM, the feature is ignored unless `lp->params.fdx` is set at compile time OR during a module load (`insmod de4x5 args='eth??:fdx'` [see below]). This is because there is no way to automatically detect full duplex links except through autonegotiation. When I include the autonegotiation feature in the SROM autoconf code, this detection will occur automatically for that case.

Command line arguments are now allowed, similar to passing arguments through LILO. This will allow a per adapter board set up of full duplex and media. The only lexical constraints are: the board name (`dev>name`) appears in the list before its parameters. The list of parameters ends either at the end of the parameter list or with another board name. The following parameters are allowed:

<code>fdx</code>	for full duplex
<code>auto-</code>	to set the media/speed; with the following sub-
<code>to-</code>	parameters: TP, TP_NW, BNC, AUI, BNC_AUI, 100Mb,
<code>sen:</code>	10Mb, AUTO

Case sensitivity is important for the sub-parameters. They *must* be upper case. Examples:


```
insmod de4x5 args='eth1:fdx autosense=BNC
↳eth0:autosense=100Mb'.
```

For a compiled in driver, in linux/drivers/net/CONFIG, place e.g.:

```
DE4X5_OPTS = -DDE4X5_PARM='"eth0:fdx autosense=AUI
↳eth2:autosense=TP"'
```

Yes, I know full duplex isn't permissible on BNC or AUI; they're just examples. By default, full duplex is turned off and AUTO is the default autosense setting. In reality, I expect only the full duplex option to be used. Note the use of single quotes in the two examples above and the lack of commas to separate items.

7.5.11 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>

7.5.12 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

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 mqprio 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: \
    mqprio 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/dpaa2/ethernet.<id>/net/fm<nr>-mac<nr>/fqids
- the ID of the buffer pool in use /sys/devices/platform/soc/<addr>.fman/<addr>.ethernet/dpaa2/ethernet.<id>/net/fm<nr>-mac<nr>/bpids

7.5.13 DPAA2 Documentation

DPAA2 (Data Path Acceleration Architecture Gen2) Overview

Copyright

© 2015 Freescale Semiconductor Inc.

Copyright

© 2018 NXP

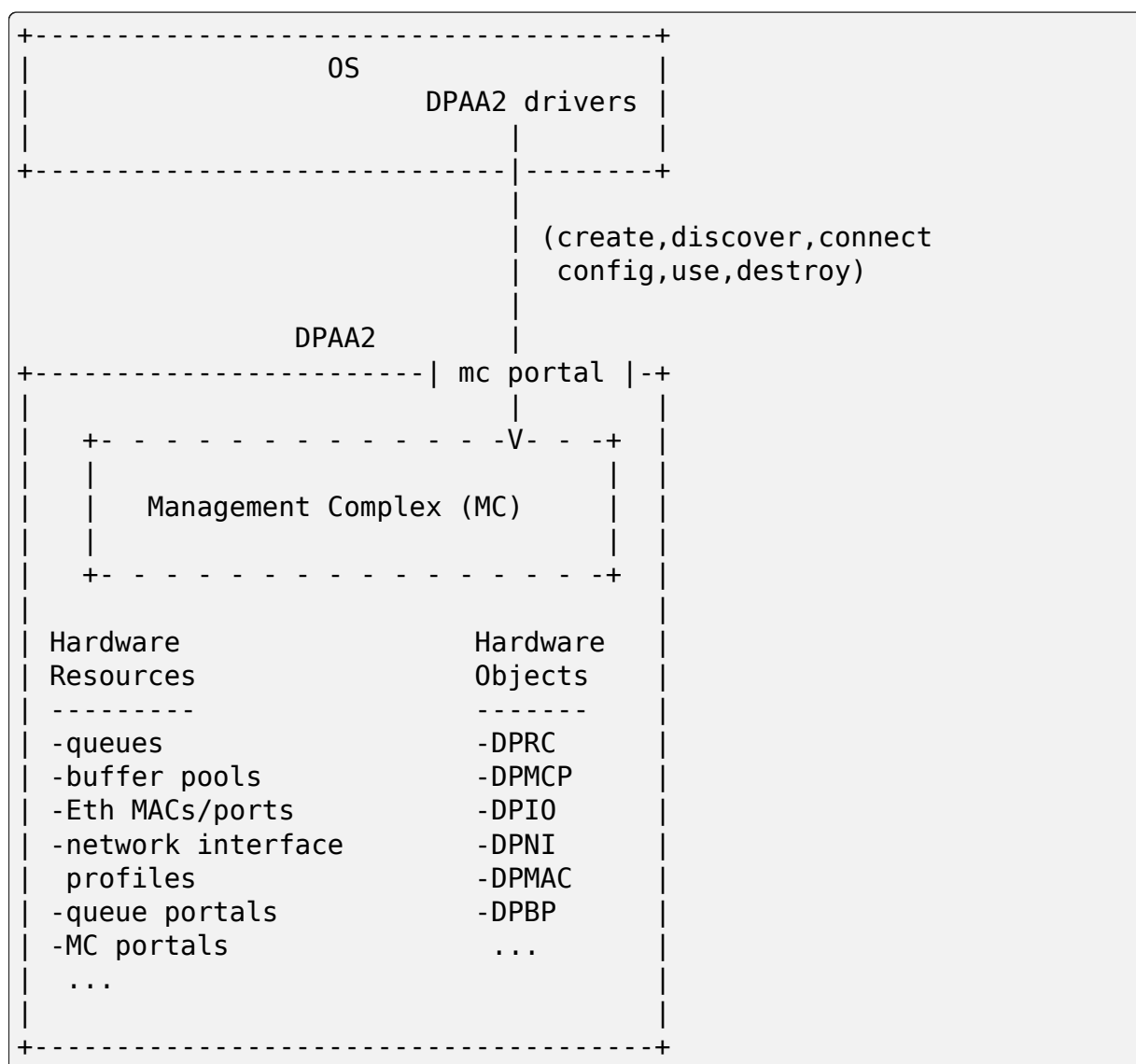
This document provides an overview of the Freescale DPAA2 architecture and how it is integrated into the Linux kernel.

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

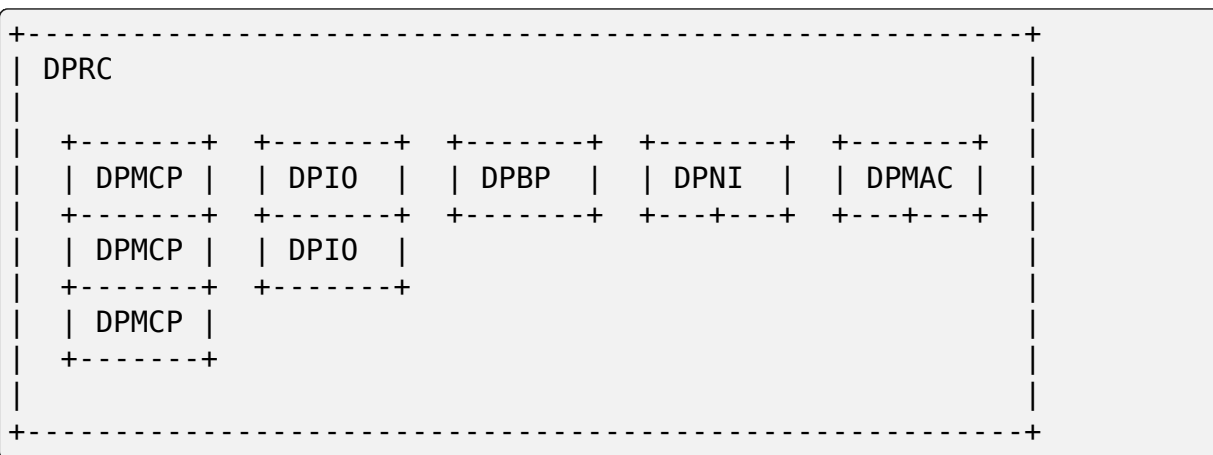
mediated by the MC and are done directly using memory mapped regions in DPIO objects.

Overview of DPAA2 Objects

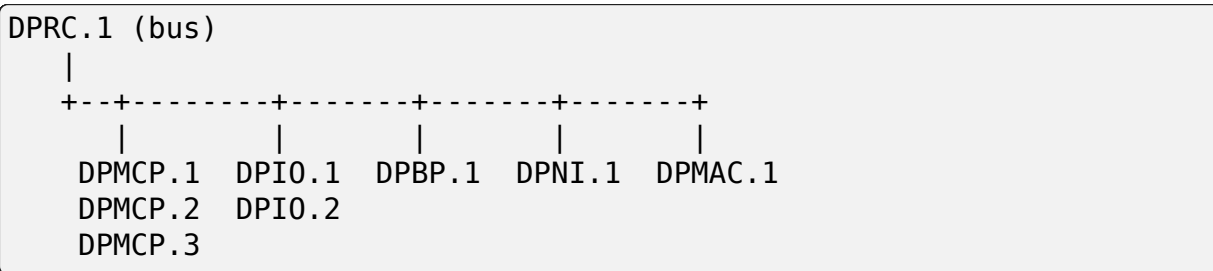
The section provides a brief overview of some key DPAA2 objects. A simple scenario is described illustrating the objects involved in creating a network interfaces.

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.

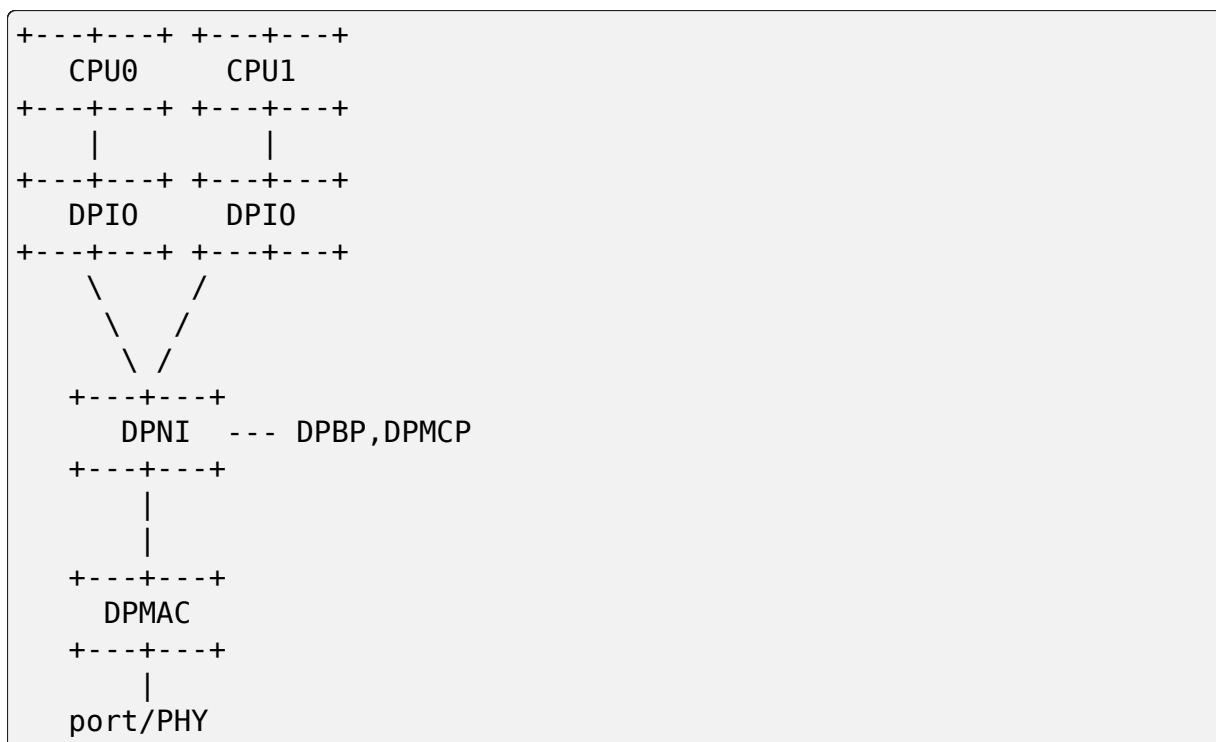
A DPRC has a mappable MMIO region (an MC portal) that can be used to send MC commands. It has an interrupt for status events (like hotplug). All objects in a container share the same hardware “isolation context”. This means that with respect to an IOMMU the isolation granularity is at the DPRC (container) 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  | | DPMAC  |
+---+---+ +---+---+
      |         |
      +=====+
```

- 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.

```
+-----+
| OS Network |
|   Stack   |
+-----+
```

(continues on next page)

```

+-----+
| Allocator | . . . . . | Ethernet |
| (DPMCP,DPBP) | | (DPNI) |
+-----+
.
.
.
+-----+
| DPRC driver | . . . . . | DPI0 driver |
| (DPRC) | | (DPIO) |
+-----+
| <dev add/remove> |
|
+-----+
| MC-bus driver |
| /bus/fsl-mc |
+-----+

===== HARDWARE
<-----|=====

DPIO
|
DPNI --- DPBP
|
DPMAC
|
PHY -----+
|
=====

```

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:

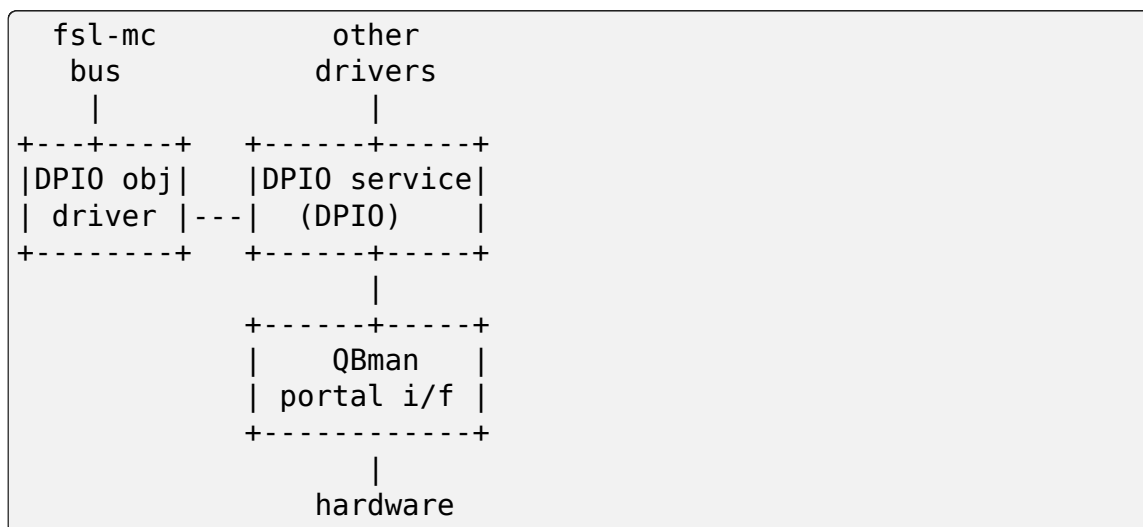
- A. allow other drivers, such as the Ethernet driver, to enqueue and dequeue frames for their respective objects
- B. allow drivers to register callbacks for data availability notifications when data becomes available on a queue or channel
- C. allow drivers to manage hardware buffer pools

The Linux DPIO driver consists of 3 primary components-

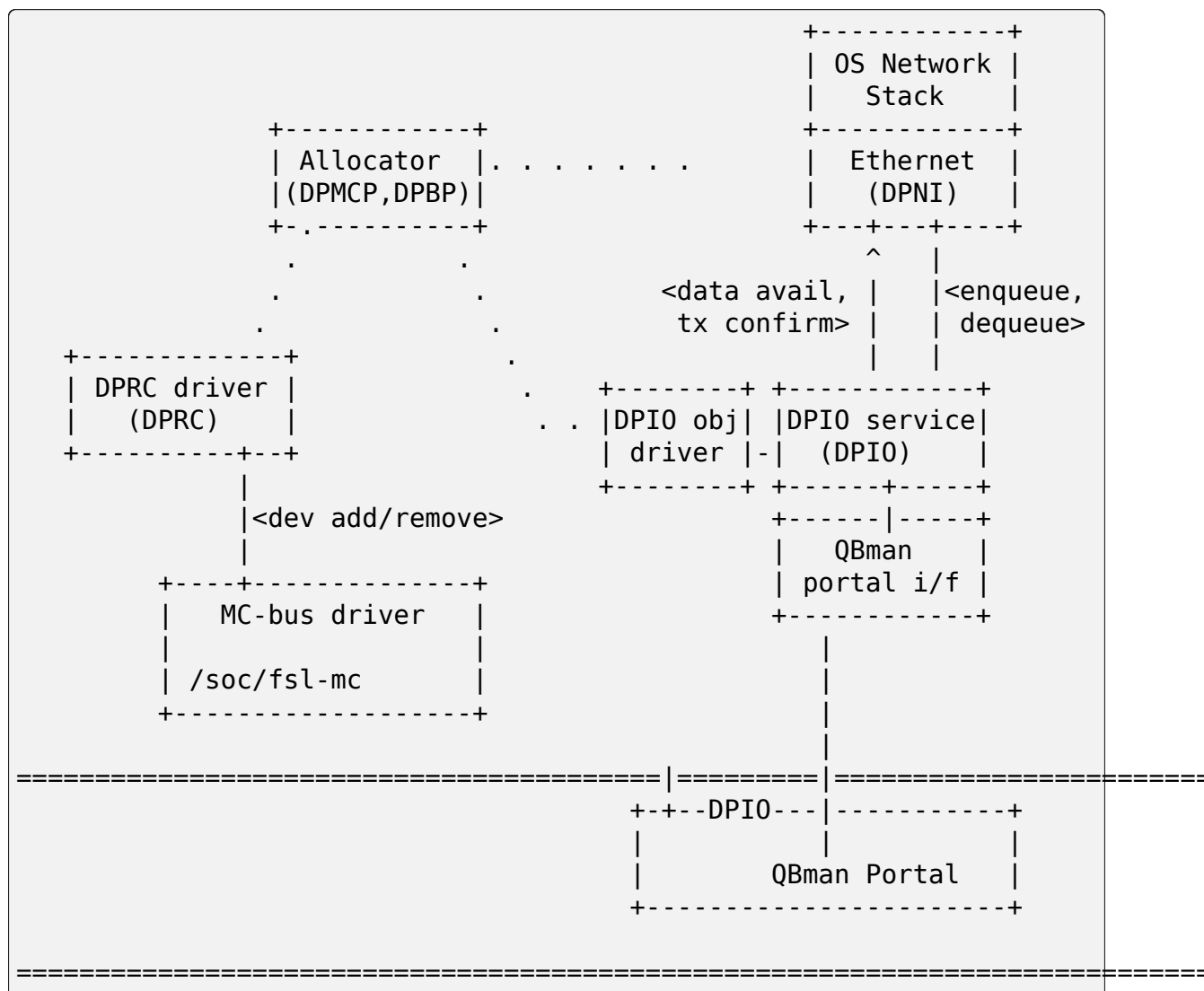
DPIO object driver- fsl-mc driver that manages the DPIO object

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

```
dpaa2_io_service_register()
dpaa2_io_service_deregister()
dpaa2_io_service_rearm()
```

Queuing

```
dpaa2_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.

Deque 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:

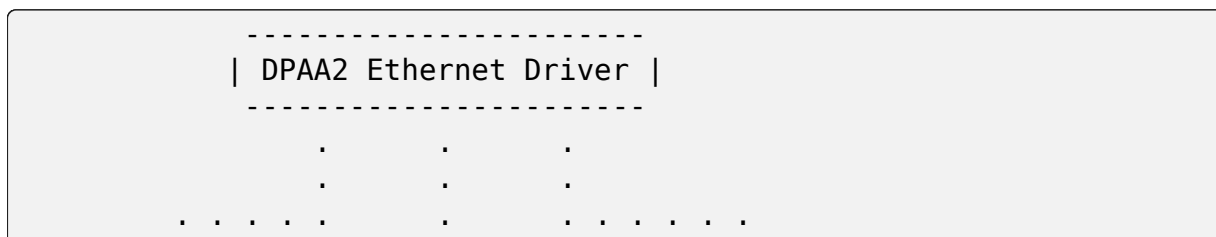
- network interfaces
- queues, channels
- buffer pools
- MAC/PHY

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](#).

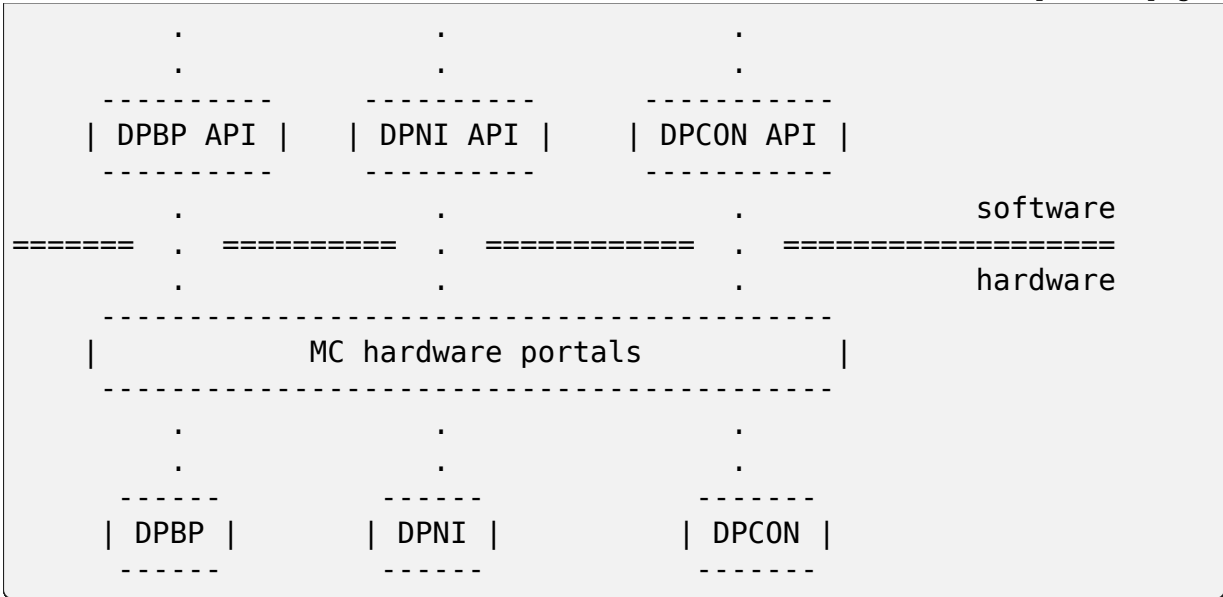
Each Linux net device is built on top of a Datapath Network Interface (DPNI) object and uses Buffer Pools (DPBPs), I/O Portals (DPIOs) and Concentrators (DPCONs).

Configuration interface:



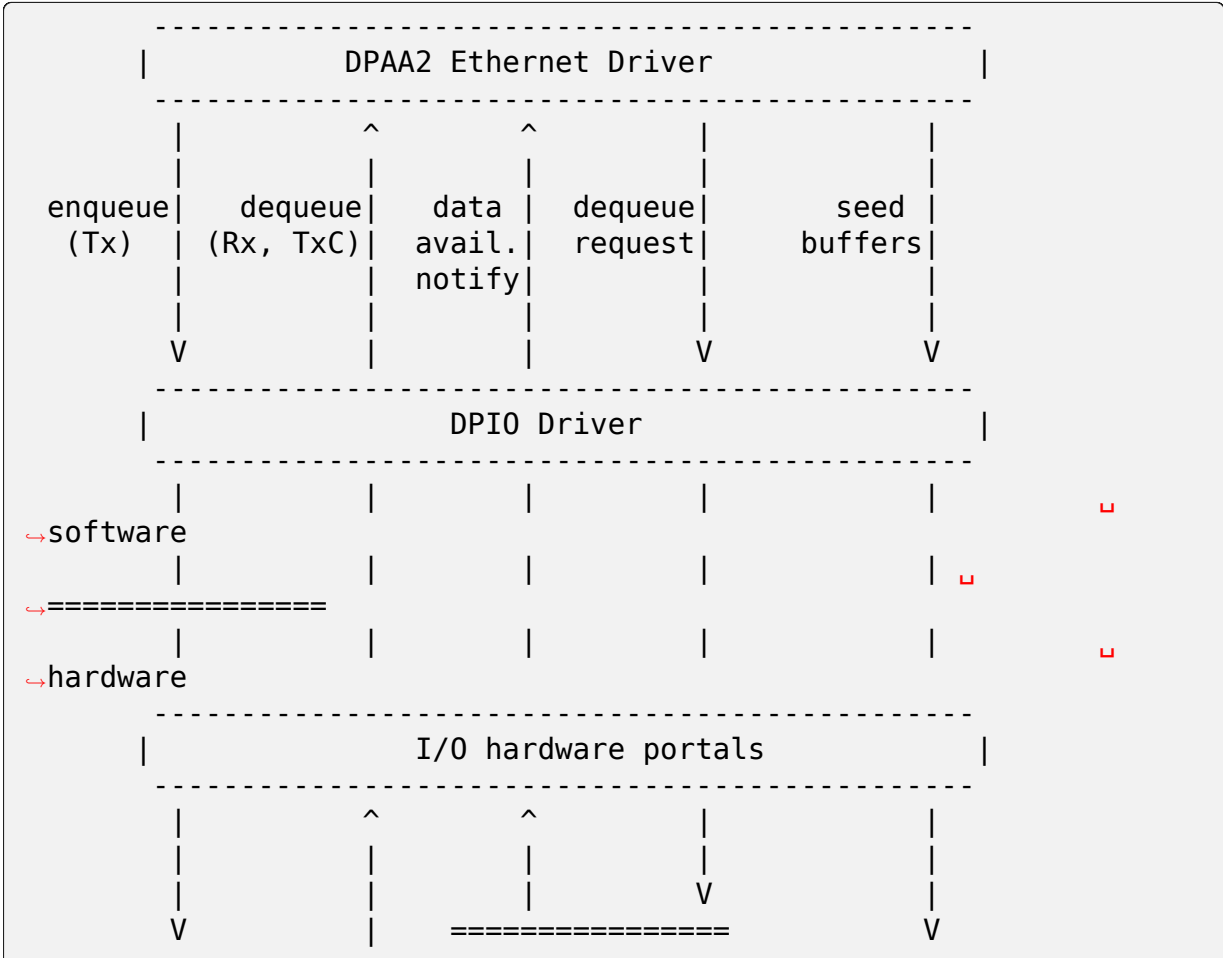
(continues on next page)

(continued from previous page)



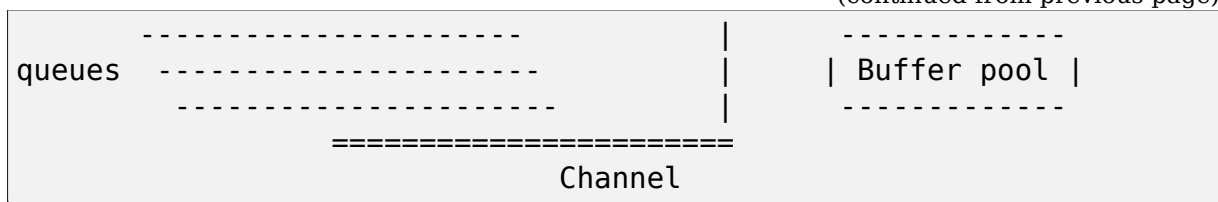
The DPNI is network interfaces without a direct one-on-one mapping to PHYs. DPBPs represent hardware buffer pools. Packet I/O is performed in the context of DPCON objects, using DPIO portals for managing and communicating with the hardware resources.

Datapath (I/O) interface:



(continues on next page)

(continued from previous page)



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.

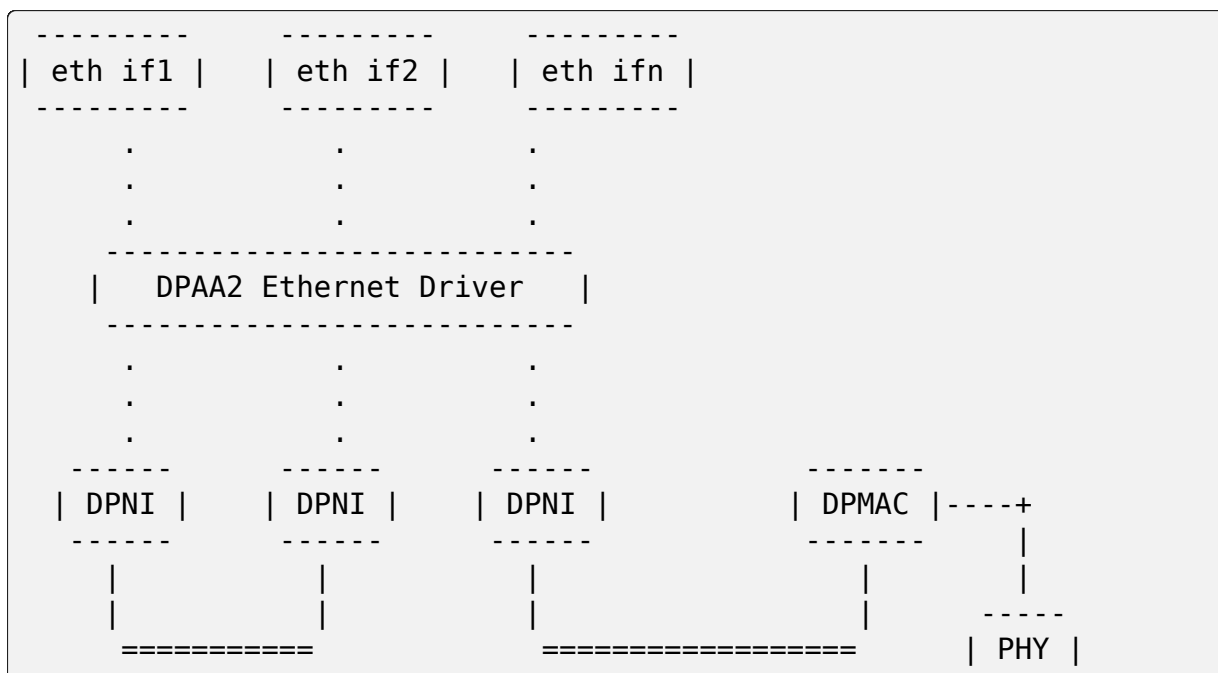
Frames are transmitted and received through hardware frame queues, which can be grouped in channels for the purpose of hardware scheduling. The Ethernet driver enqueues TX frames on egress queues and after transmission is complete a TX confirmation frame is sent back to the CPU.

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 DPPIOs.

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.



(continues on next page)

(continued from previous page)

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

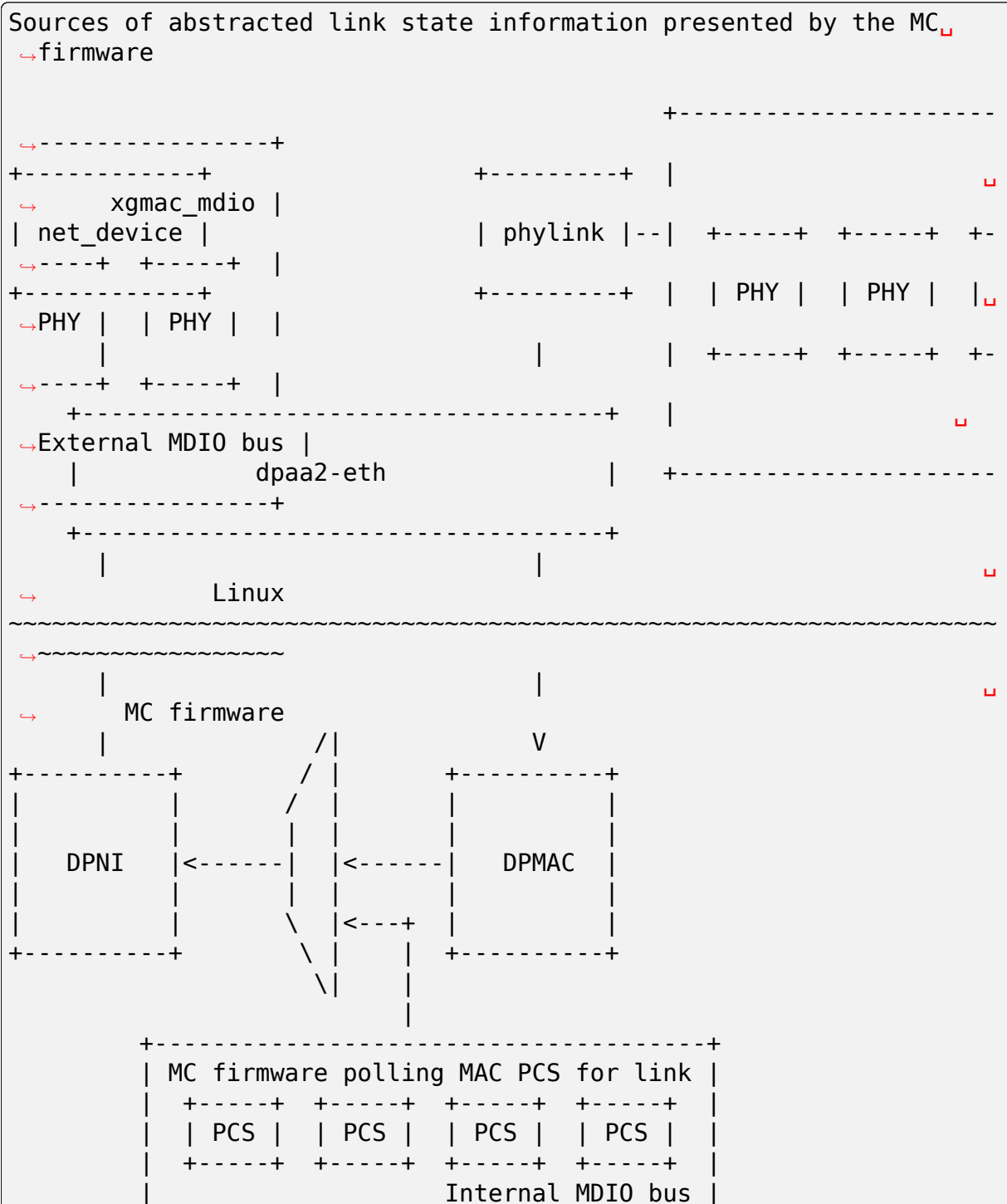
Overview

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 DPNIs. Depending on the connection type, the netif_carrier_[on/off] is handled directly by the dpaa2-eth driver or by phylink.



(continues on next page)

(continued from previous page)

+-----+

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:

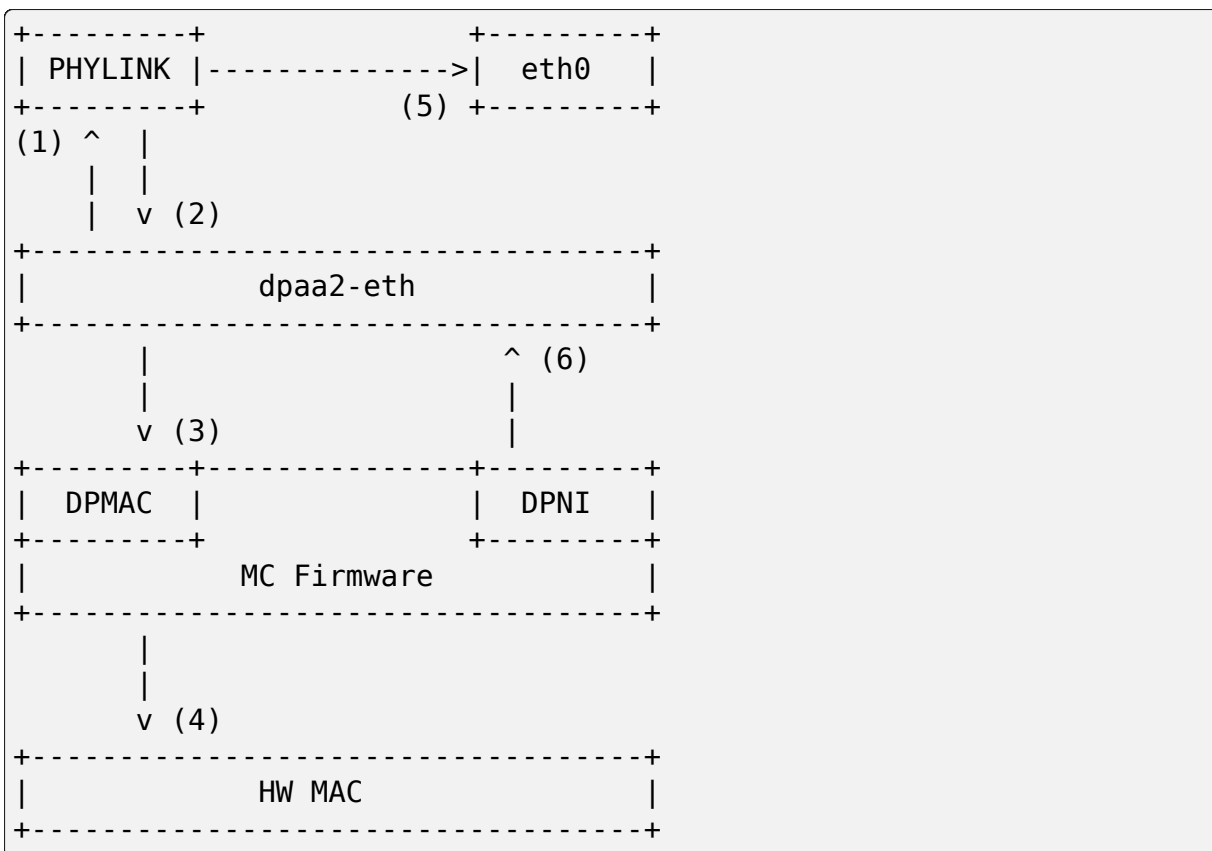
- `.validate()` will populate the supported linkmodes with the MAC capabilities only when the `phy_interface_t` is `RGMII_*` (at the moment, this is the only link type supported by the driver).
- `.mac_config()` will configure the MAC in the new configuration using the `dpmac_set_link_state()` MC firmware 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:

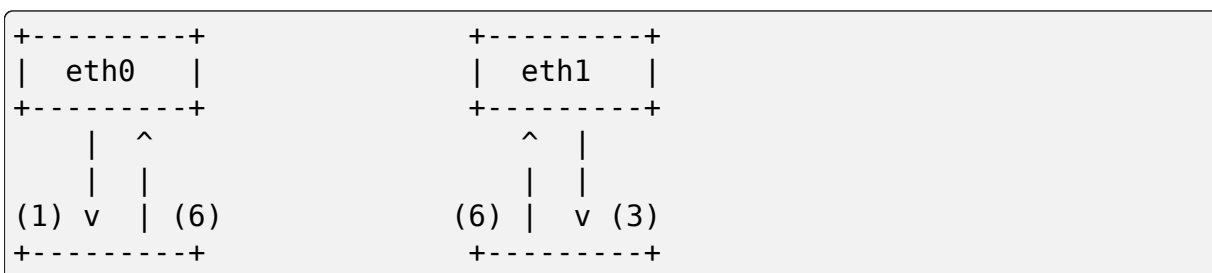
- (1) `phylink_start()` called from `.dev_open()`.
- (2) The `.mac_config()` and `.mac_link_up()` callbacks are called by PHYLINK.
- (3) In order to configure the HW MAC, the MC Firmware API `dpmac_set_link_state()` is called.
- (4) The firmware will eventually setup the HW MAC in the new configuration.

- (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()`.



(continues on next page)


```
|dpaa2-eth|                                     |dpaa2-eth|
+-----+                                     +-----+
      ^   |                                   ^   |
      |   |                                   |   |
(2) v   | (5)                               (5) | v (4)
+-----+                                     +-----+
| DPNI |                                     | DPNI |
+-----+                                     +-----+
| MC Firmware |                             | MC Firmware |
+-----+                                     +-----+
```

```
- int dpaa2_mac_connect(struct dpaa2_mac *mac);
- void dpaa2_mac_disconnect(struct dpaa2_mac *mac);
```

```
- bool dpaa2_mac_is_type_fixed(struct fsl_mc_device *dpmac_dev,  
↪ struct fsl_mc_io *mc_io);
```

Andy Fleming <afleming@freescale.com>

2005-07-28

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.

7.5.15 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

Registers

All registers are MMIO and big endian.

The registers are used for initializing and configuring the device as well as querying device status in response to management interrupts.

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.

Traffic Queues

gVNIC' s 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.

7.5.16 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.

7.5.17 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: [/Linux Ethernet Bonding Driver HOWTO](#)

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? msglvl 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 <https://wiki.linuxfoundation.org/networking/napi> for more information on NAPI.

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/>

or the Intel Wired Networking project hosted by Sourceforge at: <http://sourceforge.net/projects/e1000> 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 e1000-devel@lists.sf.net.

7.5.18 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/aplnots/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 run-time 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>

or the Intel Wired Networking project hosted by Sourceforge at:

<http://sourceforge.net/projects/e1000>

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 e1000-devel@lists.sf.net

7.5.19 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-xxxxxxx (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/>

or the Intel Wired Networking project hosted by Sourceforge at:

<https://sourceforge.net/projects/e1000>

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 e1000-devel@lists.sf.net.

7.5.20 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/>

or the Intel Wired Networking project hosted by Sourceforge at:

<https://sourceforge.net/projects/e1000>

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 e1000-devel@lists.sf.net.

7.5.21 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 multiqueue 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/>

or the Intel Wired Networking project hosted by Sourceforge at:

<https://sourceforge.net/projects/e1000>

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 e1000-devel@lists.sf.net.

7.5.22 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/>

or the Intel Wired Networking project hosted by Sourceforge at:

<https://sourceforge.net/projects/e1000>

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 e1000-devel@lists.sf.net.

7.5.23 Linux Base Driver for 10 Gigabit Intel(R) Ethernet Network Connection

October 1, 2018

Contents

- In This Release
- Identifying Your Adapter
- Command Line Parameters
- Improving Performance
- Additional Configurations
- Known Issues/Troubleshooting
- Support

In This Release

This file describes the ixgb Linux Base Driver for the 10 Gigabit Intel(R) Network Connection. This driver includes support for Itanium(R)2-based systems.

For questions related to hardware requirements, refer to the documentation supplied with your 10 Gigabit adapter. All hardware requirements listed apply to use with Linux.

The following features are available in this kernel:

- Native VLANs
- Channel Bonding (teaming)
- SNMP

Channel Bonding documentation can be found in the Linux kernel source: */Linux Ethernet Bonding Driver HOWTO*

The driver information previously displayed in the /proc filesystem is not supported in this release. Alternatively, you can use ethtool (version 1.6 or later), lspci, and iproute2 to obtain the same information.

Instructions on updating ethtool can be found in the section “Additional Configurations” later in this document.

Identifying Your Adapter

The following Intel network adapters are compatible with the drivers in this release:

Controller	Adapter Name	Physical Layer
82597EX	Intel(R) PRO/10GbE LR/SR/CX4 Adapters	Server <ul style="list-style-type: none">• 10G Base-LR (fiber)• 10G Base-SR (fiber)• 10G Base-CX4 (copper)

For more information on how to identify your adapter, go to the Adapter & Driver ID Guide at:

<https://support.intel.com>

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 ixgb [<option>=<VAL1>,<VAL2>,...]
```

For example, with two 10GbE PCI adapters, entering:

```
modprobe ixgb TxDescriptors=80,128
```

loads the `ixgb` driver with 80 TX resources for the first adapter and 128 TX resources for the second adapter.

The default value for each parameter is generally the recommended setting, unless otherwise noted.

Copybreak

Valid Range

0-XXXX

Default Value

256

This is the maximum size of packet that is copied to a new buffer on receive.

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.

FlowControl

Valid Range

0-3 (0=none, 1=Rx only, 2=Tx only, 3=Rx&Tx)

Default Value

1 if no EEPROM, otherwise read from EEPROM

This parameter controls the automatic generation(Tx) and response(Rx) to Ethernet PAUSE frames. There are hardware bugs associated with enabling Tx flow control so beware.

RxDescriptors

Valid Range

64-4096

Default Value

1024

This value is the number of receive descriptors allocated by the driver. Increasing this value allows the driver to buffer more incoming packets. Each descriptor is 16 bytes. A receive buffer is also allocated for each descriptor and can be either 2048, 4056, 8192, or 16384 bytes, depending on the MTU setting. When the MTU size is 1500 or less, the receive buffer size is 2048 bytes. When the MTU is greater than 1500 the receive buffer size will be either 4056, 8192, or 16384 bytes. The maximum MTU size is 16114.

TxDescriptors

Valid Range

64-4096

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.

RxIntDelay

Valid Range

0-65535 (0=off)

Default Value

72

This value delays the generation of receive interrupts in units of 0.8192 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.

TxIntDelay

Valid Range

0-65535 (0=off)

Default Value

32

This value delays the generation of transmit interrupts in units of 0.8192 microseconds. Transmit interrupt reduction can improve CPU efficiency if properly tuned for specific network traffic. Increasing this value adds extra latency to frame transmission and can end up decreasing the throughput of TCP traffic. If this value is set too high, it will cause the driver to run out of available transmit descriptors.

XsumRX

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.

RxFCHighThresh

Valid Range

1,536-262,136 (0x600 - 0x3FFF8, 8 byte granularity)

Default Value

196,608 (0x30000)

Receive Flow control high threshold (when we send a pause frame)

RxFCLowThresh

Valid Range

64-262,136 (0x40 - 0x3FFF8, 8 byte granularity)

Default Value

163,840 (0x28000)

Receive Flow control low threshold (when we send a resume frame)

FCReqTimeout

Valid Range

1-65535

Default Value

65535

Flow control request timeout (how long to pause the link partner's tx)

IntDelayEnable

Value Range

0,1

Default Value

1

Interrupt Delay, 0 disables transmit interrupt delay and 1 enables it.

Improving Performance

With the 10 Gigabit server adapters, the default Linux configuration will very likely limit the total available throughput artificially. There is a set of configuration changes that, when applied together, will increase the ability of Linux to transmit and receive data. The following enhancements were originally acquired from settings published at <https://www.spec.org/web99/> for various submitted results using Linux.

NOTE:

These changes are only suggestions, and serve as a starting point for tuning your network performance.

The changes are made in three major ways, listed in order of greatest effect:

- Use `ip link` to modify the `mtu` (maximum transmission unit) and the `txqueuelen` parameter.
- Use `sysctl` to modify `/proc` parameters (essentially kernel tuning)
- Use `setpci` to modify the `MMRBC` field in `PCI-X` configuration space to increase transmit burst lengths on the bus.

NOTE:

`setpci` modifies the adapter's configuration registers to allow it to read up to 4k bytes at a time (for transmits). However, for some systems the behavior after modifying this register may be undefined (possibly errors of some kind). A power-cycle, hard reset or explicitly setting the `e6` register back to 22 (`setpci -d 8086:1a48 e6.b=22`) may be required to get back to a stable configuration.

- COPY these lines and paste them into `ixgb_perf.sh`:


```
#!/bin/bash
echo "configuring network performance , edit this file to change
↳the interface
or device ID of 10GbE card"
# set mmrbc to 4k reads, modify only Intel 10GbE device IDs
# replace 1a48 with appropriate 10GbE device's ID installed on the
↳system,
# if needed.
setpci -d 8086:1a48 e6.b=2e
# set the MTU (max transmission unit) - it requires your switch and
↳clients
# to change as well.
# set the txqueuelen
# your ixgb adapter should be loaded as eth1 for this to work,
↳change if needed
ip li set dev eth1 mtu 9000 txqueuelen 1000 up
# call the sysctl utility to modify /proc/sys entries
sysctl -p ./sysctl_ixgb.conf
```

- COPY these lines and paste them into sysctl_ixgb.conf:

```
# some of the defaults may be different for your kernel
# call this file with sysctl -p <this file>
# these are just suggested values that worked well to increase
↳throughput in
# several network benchmark tests, your mileage may vary

### IPV4 specific settings
# turn TCP timestamp support off, default 1, reduces CPU use
net.ipv4.tcp_timestamps = 0
# turn SACK support off, default on
# on systems with a VERY fast bus -> memory interface this is the
↳big gainer
net.ipv4.tcp_sack = 0
# set min/default/max TCP read buffer, default 4096 87380 174760
net.ipv4.tcp_rmem = 10000000 10000000 10000000
# set min/pressure/max TCP write buffer, default 4096 16384 131072
net.ipv4.tcp_wmem = 10000000 10000000 10000000
# set min/pressure/max TCP buffer space, default 31744 32256 32768
net.ipv4.tcp_mem = 10000000 10000000 10000000

### CORE settings (mostly for socket and UDP effect)
# set maximum receive socket buffer size, default 131071
net.core.rmem_max = 524287
# set maximum send socket buffer size, default 131071
net.core.wmem_max = 524287
# set default receive socket buffer size, default 65535
net.core.rmem_default = 524287
# set default send socket buffer size, default 65535
net.core.wmem_default = 524287
```

(continues on next page)

(continued from previous page)

```
# set maximum amount of option memory buffers, default 10240
net.core.optmem_max = 524287
# set number of unprocessed input packets before kernel starts
↳dropping them; default 300
net.core.netdev_max_backlog = 300000
```

Edit the `ixgb_perf.sh` script if necessary to change `eth1` to whatever interface your `ixgb` driver is using and/or replace `'1a48'` with appropriate 10GbE device's ID installed on the system.

NOTE:

Unless these scripts are added to the boot process, these changes will only last only until the next system reboot.

Resolving Slow UDP Traffic

If your server does not seem to be able to receive UDP traffic as fast as it can receive TCP traffic, it could be because Linux, by default, does not set the network stack buffers as large as they need to be to support high UDP transfer rates. One way to alleviate this problem is to allow more memory to be used by the IP stack to store incoming data.

For instance, use the commands:

```
sysctl -w net.core.rmem_max=262143
```

and:

```
sysctl -w net.core.rmem_default=262143
```

to increase the read buffer memory max and default to 262143 (256k - 1) from defaults of max=131071 (128k - 1) and default=65535 (64k - 1). These variables will increase the amount of memory used by the network stack for receives, and can be increased significantly more if necessary for your application.

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.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 10GbE Family of Adapters is `ixgb`.

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 16114. Use the `ip` command to increase the MTU size. For example:

```
ip li set dev ethx mtu 9000
```

The maximum MTU setting for Jumbo Frames is 16114. This value coincides with the maximum Jumbo Frames size of 16128.

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/>

NOTE:

The `ethtool` version 1.6 only supports a limited set of `ethtool` options. Support for a more complete `ethtool` feature set can be enabled by upgrading to the latest version.

NAPI

NAPI (Rx polling mode) is supported in the `ixgb` driver.

See <https://wiki.linuxfoundation.org/networking/napi> for more information on NAPI.

Known Issues/Troubleshooting

NOTE:

After installing the driver, if your Intel Network Connection is not working, verify in the “In This Release” section of the readme that you have installed the correct driver.

Cable Interoperability Issue with Fujitsu XENPAK Module in SmartBits Chassis

Excessive CRC errors may be observed if the Intel(R) PRO/10GbE CX4 Server adapter is connected to a Fujitsu XENPAK CX4 module in a SmartBits chassis using 15 m/24AWG cable assemblies manufactured by Fujitsu or Leoni. The CRC errors may be received either by the Intel(R) PRO/10GbE CX4 Server adapter or the SmartBits. If this situation occurs using a different cable assembly may resolve the issue.

Cable Interoperability Issues with HP Procurve 3400cl Switch Port

Excessive CRC errors may be observed if the Intel(R) PRO/10GbE CX4 Server adapter is connected to an HP Procurve 3400cl switch port using short cables (1 m or shorter). If this situation occurs, using a longer cable may resolve the issue.

Excessive CRC errors may be observed using Fujitsu 24AWG cable assemblies that Are 10 m or longer or where using a Leoni 15 m/24AWG cable assembly. The CRC errors may be received either by the CX4 Server adapter or at the switch. If this situation occurs, using a different cable assembly may resolve the issue.

Jumbo Frames System Requirement

Memory allocation failures have been observed on Linux systems with 64 MB of RAM or less that are running Jumbo Frames. If you are using Jumbo Frames, your system may require more than the advertised minimum requirement of 64 MB of system memory.

Performance Degradation with Jumbo Frames

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.

Allocating Rx Buffers when Using Jumbo Frames

Allocating Rx buffers when using Jumbo Frames on 2.6.x kernels may fail if the available memory is heavily fragmented. This issue may be seen with PCI-X adapters or with packet split disabled. This can be reduced or eliminated by changing the amount of available memory for receive buffer allocation, by increasing `/proc/sys/vm/min_free_kbytes`.

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, do either of the following:

- Turn on ARP filtering by entering:

```
echo 1 > /proc/sys/net/ipv4/conf/all/arp_filter
```

- Install the interfaces in separate broadcast domains - either in different switches or in a switch partitioned to VLANs.

UDP Stress Test Dropped Packet Issue

Under small packets UDP stress test with 10GbE driver, the Linux system may drop UDP packets due to the fullness of socket buffers. You may want to change the driver's Flow Control variables to the minimum value for controlling packet reception.

Tx Hangs Possible Under Stress

Under stress conditions, if TX hangs occur, turning off TSO “`ethtool -K eth0 tso off`” may resolve the problem.

Support

For general information, go to the Intel support website at:

<https://www.intel.com/support/>

or the Intel Wired Networking project hosted by Sourceforge at:

<https://sourceforge.net/projects/e1000>

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 e1000-devel@lists.sf.net

7.5.24 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)	E10GQSFP5SR

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 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.

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-edc@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/>

or the Intel Wired Networking project hosted by Sourceforge at:

<https://sourceforge.net/projects/e1000>

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 e1000-devel@lists.sf.net.

7.5.25 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/>

or the Intel Wired Networking project hosted by Sourceforge at:

<https://sourceforge.net/projects/e1000>

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 e1000-devel@lists.sf.net.

7.5.26 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 0x4FFFF ...
```

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. For more information on NAPI, see <https://wiki.linuxfoundation.org/networking/napi>

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 802.10 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 new hardware offload mode in mqprio that makes full use of the mqprio 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 mqprio.
- 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 Side-band 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/>

or the Intel Wired Networking project hosted by Sourceforge at:

<https://sourceforge.net/projects/e1000>

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 e1000-devel@lists.sf.net.

7.5.27 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_mqprio`, `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 mqprio num_tc 2 map 0 0 0 0 1 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 new hardware offload mode in mqprio that makes full use of the mqprio 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 mqprio.

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 mqprio
- 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>

or the Intel Wired Networking project hosted by Sourceforge at:

<https://sourceforge.net/projects/e1000>

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 e1000-devel@lists.sf.net

7.5.28 Linux Base Driver for the Intel(R) Ethernet Connection E800 Series

Intel ice Linux driver. Copyright(c) 2018 Intel Corporation.

Contents

- Enabling the driver
- Support

The driver in this release supports Intel' s E800 Series of products. For more information, visit Intel' s support page at <https://support.intel.com>.

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**
 - > **Intel devices**
 - > Intel(R) Ethernet Connection E800 Series Support

Support

For general information, go to the Intel support website at:

<https://www.intel.com/support/>

or the Intel Wired Networking project hosted by Sourceforge at:

<https://sourceforge.net/projects/e1000>

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 e1000-devel@lists.sf.net.

7.5.29 Marvell OcteonTx2 RVU Kernel Drivers

Copyright (c) 2020 Marvell International Ltd.

Contents

- [Overview](#)
- [Drivers](#)
- [Basic packet flow](#)

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 netdev. 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 viceversa.
- 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.

7.5.30 Mellanox ConnectX(R) mlx5 core VPI Network Driver

Copyright (c) 2019, Mellanox Technologies LTD.

Contents

- *Enabling the driver and kconfig options*
- *Devlink info*
- *Devlink parameters*
- *Devlink health reporters*
- *mlx5 tracepoints*

Enabling the driver and kconfig options

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_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_EN_ARFS=(y/n)

Enables Hardware-accelerated receive flow steering (arfs) support, and ntuple filtering.

<https://community.mellanox.com/s/article/howto-configure-arfs-on-connectx-4>

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_CORE_EN_DCB=(y/n):

Enables [Data Center Bridging \(DCB\) 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_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_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_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_EN_IPSEC=(y/n)

Enables [IPSec XFRM cryptography-offload acceleration](#).

CONFIG_MLX5_EN_TLS=(y/n)

TLS cryptography-offload acceleration.

CONFIG_MLX5_INFINIBAND=(y/n/m) (module `mlx5_ib.ko`)

Provides low-level InfiniBand/RDMA and [RoCE](#) support.

External options (Choose if the corresponding `mlx5` feature is required)

- `CONFIG_PTP_1588_CLOCK`: When chosen, `mlx5` ptp support will be enabled
- `CONFIG_VXLAN`: When chosen, `mlx5` vxlan support will be enabled.
- `CONFIG_MLXFW`: When chosen, `mlx5` firmware flashing support will be enabled (via `devlink` and `ethtool`).

Devlink info

The `devlink` info reports the running and stored firmware versions on device. It also prints the device PSID which represents the HCA board type ID.

User command example:

```
$ devlink dev info pci/0000:00:06.0
pci/0000:00:06.0:
driver mlx5_core
versions:
  fixed:
    fw.psid MT_0000000009
  running:
    fw.version 16.26.0100
  stored:
    fw.version 16.26.0100
```

Devlink parameters

flow_steering_mode: Device flow steering mode

The flow steering mode parameter controls the flow steering mode of the driver. Two modes are supported: 1. ‘dmfs’ - Device managed flow steering. 2. ‘smfs’ - Software/Driver managed flow steering.

In DMFS mode, the HW steering entities are created and managed through the Firmware. In SMFS mode, the HW steering entities are created and managed though by the driver directly into Hardware without firmware intervention.

SMFS mode is faster and provides better rule insertion rate compared to default DMFS mode.

User command examples:

- Set SMFS flow steering mode:

```
$ devlink dev param set pci/0000:06:00.0 name flow_steering_  
↪mode value "smfs" cmode runtime
```

- Read device flow steering mode:

```
$ devlink dev param show pci/0000:06:00.0 name flow_steering_  
↪mode  
pci/0000:06:00.0:  
name flow_steering_mode type driver-specific  
values:  
    cmode runtime value smfs
```

enable_roce: RoCE enablement state

RoCE enablement state controls driver support for RoCE traffic. When RoCE is disabled, there is no gid table, only raw ethernet QPs are supported and traffic on the well known UDP RoCE port is handled as raw ethernet traffic.

To change RoCE enablement state a user must change the driverinit cmode value and run devlink reload.

User command examples:

- Disable RoCE:

```
$ devlink dev param set pci/0000:06:00.0 name enable_roce value_  
↪false cmode driverinit  
$ devlink dev reload pci/0000:06:00.0
```

- Read RoCE enablement state:

```
$ devlink dev param show pci/0000:06:00.0 name enable_roce  
pci/0000:06:00.0:  
name enable_roce type generic  
values:  
    cmode driverinit value true
```

Devlink health reporters

tx reporter

The tx reporter is responsible for reporting and recovering of the following two 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 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**
RX queues descriptors population 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 it could be recoverable 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 health 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.

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.

mlx5 tracepoints

mlx5 driver provides internal trace points for tracking and debugging using kernel tracepoints interfaces (refer to Documentation/trace/fttrace.rst).

For the list of support mlx5 events check `/sys/kernel/debug/tracing/events/mlx5/` tc and eswitch offloads tracepoints:

- `mlx5e_configure_flower`: trace flower filter actions and cookies offloaded to mlx5:

```
$ echo mlx5:mlx5e_configure_flow >> /sys/kernel/debug/tracing/
↪set_event
$ cat /sys/kernel/debug/tracing/trace
...
tc-6535 [019] ...1 2672.404466: mlx5e_configure_flow:
↪cookie=0000000067874a55 actions= REDIRECT
```

- `mlx5e_delete_flow`: trace flow filter actions and cookies deleted from `mlx5`:

```
$ echo mlx5:mlx5e_delete_flow >> /sys/kernel/debug/tracing/
↪set_event
$ cat /sys/kernel/debug/tracing/trace
...
tc-6569 [010] .N.1 2686.379075: mlx5e_delete_flow:
↪cookie=0000000067874a55 actions= NULL
```

- `mlx5e_stats_flow`: trace flow stats request:

```
$ echo mlx5:mlx5e_stats_flow >> /sys/kernel/debug/tracing/set_
↪event
$ cat /sys/kernel/debug/tracing/trace
...
tc-6546 [010] ...1 2679.704889: mlx5e_stats_flow:
↪cookie=0000000060eb3d6a bytes=0 packets=0 lastused=4295560217
```

- `mlx5e_tc_update_neigh_used_value`: trace tunnel rule neigh update value of-
loaded to `mlx5`:

```
$ echo mlx5:mlx5e_tc_update_neigh_used_value >> /sys/kernel/
↪debug/tracing/set_event
$ cat /sys/kernel/debug/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/debug/tracing/
↪set_event
$ cat /sys/kernel/debug/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
```

7.5.31 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 of 6144 bytes, each of section may contain one or more packets. The send buffer is an optimization, the driver will use slower method to handle very large packets or if the send buffer area is exhausted.

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.

7.5.32 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

- a. Insert the adapter(s) in your system.
- b. Build and load driver:

```
# insmod s2io.ko
```

- c. 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, ␣
```

(continues on next page)

(continued from previous page)

```
↪ 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 (IA64, Xeon) 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.)

7.5.33 Neterion’ s (Formerly S2io) X3100 Series 10GbE PCIe Server Adapter Linux driver

1. Introduction

This Linux driver supports all Neterion’ s X3100 series 10 GbE PCIe I/O Virtualized Server adapters.

The X3100 series supports four modes of operation, configurable via firmware:

- Single function mode
- Multi function mode
- SRIOV mode

- MRIOV mode

The functions share a 10GbE link and the pci-e bus, but hardly anything else inside the ASIC. Features like independent hw reset, statistics, bandwidth/ priority allocation and guarantees, GRO, TSO, interrupt moderation etc are supported independently on each function.

(See below for a complete list of features supported for both IPv4 and IPv6)

2. Features supported

- i) Single function mode (up to 17 queues)
- ii) Multi function mode (up to 17 functions)
- iii) PCI-SIG' s I/O Virtualization
 - Single Root mode: v1.0 (up to 17 functions)
 - Multi-Root mode: v1.0 (up to 17 functions)
- iv) Jumbo frames

X3100 Series supports MTU up to 9600 bytes, modifiable using ip command.
- v) Offloads supported: (Enabled by default)
 - Checksum offload (TCP/UDP/IP) on transmit and receive paths
 - TCP Segmentation Offload (TSO) on transmit path
 - Generic Receive Offload (GRO) on receive path
- vi) MSI-X: (Enabled by default)

Resulting in noticeable performance improvement (up to 7% on certain platforms).
- vii) NAPI: (Enabled by default)

For better Rx interrupt moderation.
- viii) RTH (Receive Traffic Hash): (Enabled by default)

Receive side steering for better scaling.
- ix) Statistics

Comprehensive MAC-level and software statistics displayed using “ethtool -S” option.
- x) Multiple hardware queues: (Enabled by default)

Up to 17 hardware based transmit and receive data channels, with multiple steering options (transmit multiqueue enabled by default).

3) Configurable driver parameters:

- i) **max_config_dev**
Specifies maximum device functions to be enabled.
Valid range: 1-8
- ii) **max_config_port**
Specifies number of ports to be enabled.
Valid range: 1,2
Default: 1
- iii) **max_config_vpath**
Specifies maximum VPATH(s) configured for each device function.
Valid range: 1-17
- iv) **vlan_tag_strip**
Enables/disables vlan tag stripping from all received tagged frames that are not replicated at the internal L2 switch.
Valid range: 0,1 (disabled, enabled respectively)
Default: 1
- v) **addr_learn_en**
Enable learning the mac address of the guest OS interface in virtualization environment.
Valid range: 0,1 (disabled, enabled respectively)
Default: 0

7.5.34 Netronome Flow Processor (NFP) Kernel Drivers

Copyright (c) 2019, Netronome Systems, Inc.

Contents

- *Overview*
- *Acquiring Firmware*

Overview

This driver supports Netronome's line of Flow Processor devices, including the NFP4000, NFP5000, and NFP6000 models, which are also incorporated in the company's family of Agilio SmartNICs. The SR-IOV physical and virtual functions for these devices are supported by the driver.

Acquiring Firmware

The 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.

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 specific firmware image for a specific card, you can use either the PCI bus address or serial number. 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_AMDAA0081-0001_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_AMDAA*` 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.

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.

Statistics

Following device statistics are available through the `ethtool -S` interface:

Table 1: 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.
7.5. Ethernet Device Drivers		269
dev_rx_bytes	3	Total number of bytes received.
dev_rx_uc_bytes	4	Unicast bytes received.

Note that statistics unknown to the driver will be displayed as `dev_unknown_stat$ID`, where `$ID` refers to the second column above.

7.5.35 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 ldd8:
b5:00.0 Ethernet controller: Device ldd8:1002
b6:00.0 Ethernet controller: Device ldd8:1002
```

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
...
```

(continues on next page)

(continued from previous page)

```
$ 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 maximim 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 mode 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         0
    TX: bytes  packets  errors  dropped  carrier  collsns
       1384        18         0         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
```

(continues on next page)

(continued from previous page)

```
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_pripe: 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
```

(continues on next page)

(continued from previous page)

```
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_pripe: 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_pripe_0_lus_count: 0
tx_pripe_1_lus_count: 0
tx_pripe_2_lus_count: 0
```

(continues on next page)

(continued from previous page)

```
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
```

7.5.36 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/smc9/smc9-12.tar.gz> <ftp://sfbox.vt.edu/filebox/F/fenris/smc9/smc9-12.tar.gz>

Contacting me:

erik@mail.vt.edu

7.5.37 Linux Driver for the Synopsys(R) Ethernet Controllers “stm-mac”

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 uper) 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 <= 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 `snps,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 Partner 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 incrx 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 net-dev@vger.kernel.org

7.5.38 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 then potential incoming rate, thus, rate of all incoming tx queues has to be a little less
- Real rates can differ, due to discreteness
- 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

```

+-----+
↪ +
↪ |
↪ |
↪ |
↪ |
+-----+ +---| t |
↪ |
|          | 1 | E | | | h | ./tsn_listener -d \
↪ |
| Target board: | 0 | t |---+ | 0 | 18:03:73:66:87:42 -i eth0 \
↪ |

```

(continues on next page)

(continued from previous page)

```

| AM572x EVM      | 0 | h |      | | -s 1500
↪|
|                | 0 | 0 |      | |--+
↪|
| Only 2 classes: |Mb +---|      +-----+
↪+
| class A, class B |      +---|      +-----+
↪+
|                | 1 | E |      |--+
↪|
|                | 0 | t |      |      Workstation1
↪|
|                | 0 | h |--+ |E| MAC 20:cf:30:85:7d:fd
↪|
|                |Mb | 1 | +---|t |
↪|
+-----+      |h |./tsn_listener -d \
↪|
|                |0 | 20:cf:30:85:7d:fd -i eth0 \
↪|
|                | | -s 1500
↪|
|                |--+
↪|
|                +-----+
↪+

```

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

```

+-----+
↪+ u
| +-----+ +-----+ +-----+ +-----+
↪| s
| |          | |          | |          | |          |
↪| e
| | App 1    | | App 2    | | Apps   | | Apps   |
↪| r

```

(continues on next page)

(continued from previous page)

[illegible]

(continues on next page)

7.5. Ethernet Device Drivers 293

(continued from previous page)

```
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
    ↪discreetness.
    // 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
```

(continues on next page)

(continued from previous page)

```
|
+---(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 ↵
↵discreetness
$ tc qdisc add dev eth0 parent 100:1 cbs locredit -1438 \
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:
// 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&
```

(continues on next page)

(continued from previous page)

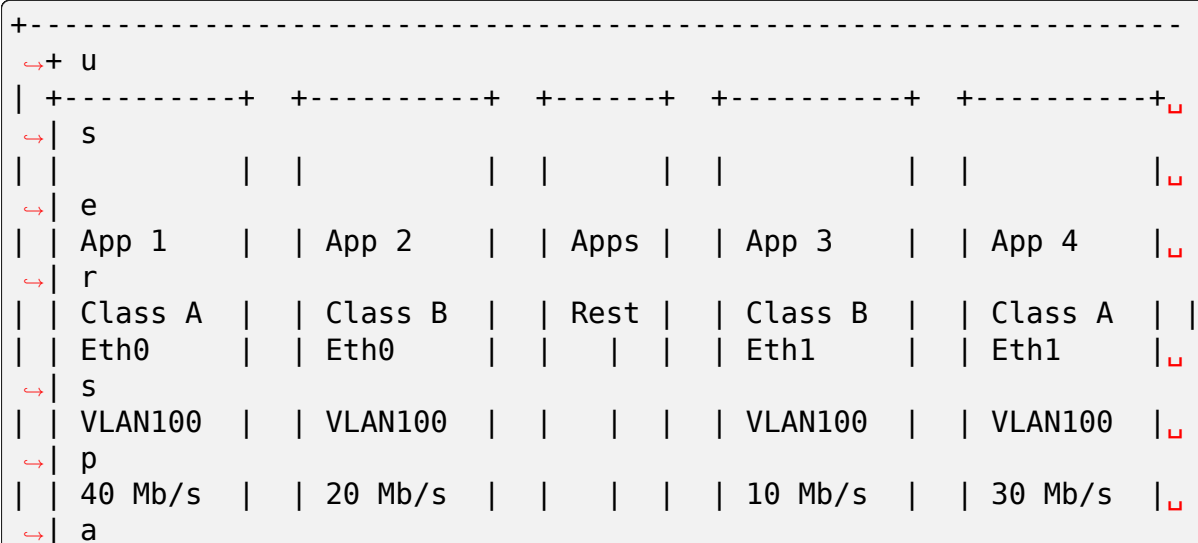
```
./tsn_talker -d 18:03:73:66:87:42 -i eth0.100 -p2 -s 1500&
```

```
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: 39012 kbps
Receiving data rate: 39012 kbps
Receiving data rate: 39012 kbps
Receiving data rate: 39012 kbps
Receiving data rate: 39012 kbps
Receiving data rate: 39012 kbps
Receiving data rate: 39012 kbps
Receiving data rate: 39012 kbps
Receiving data rate: 39012 kbps
Receiving data rate: 39012 kbps
Receiving data rate: 39012 kbps
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
```

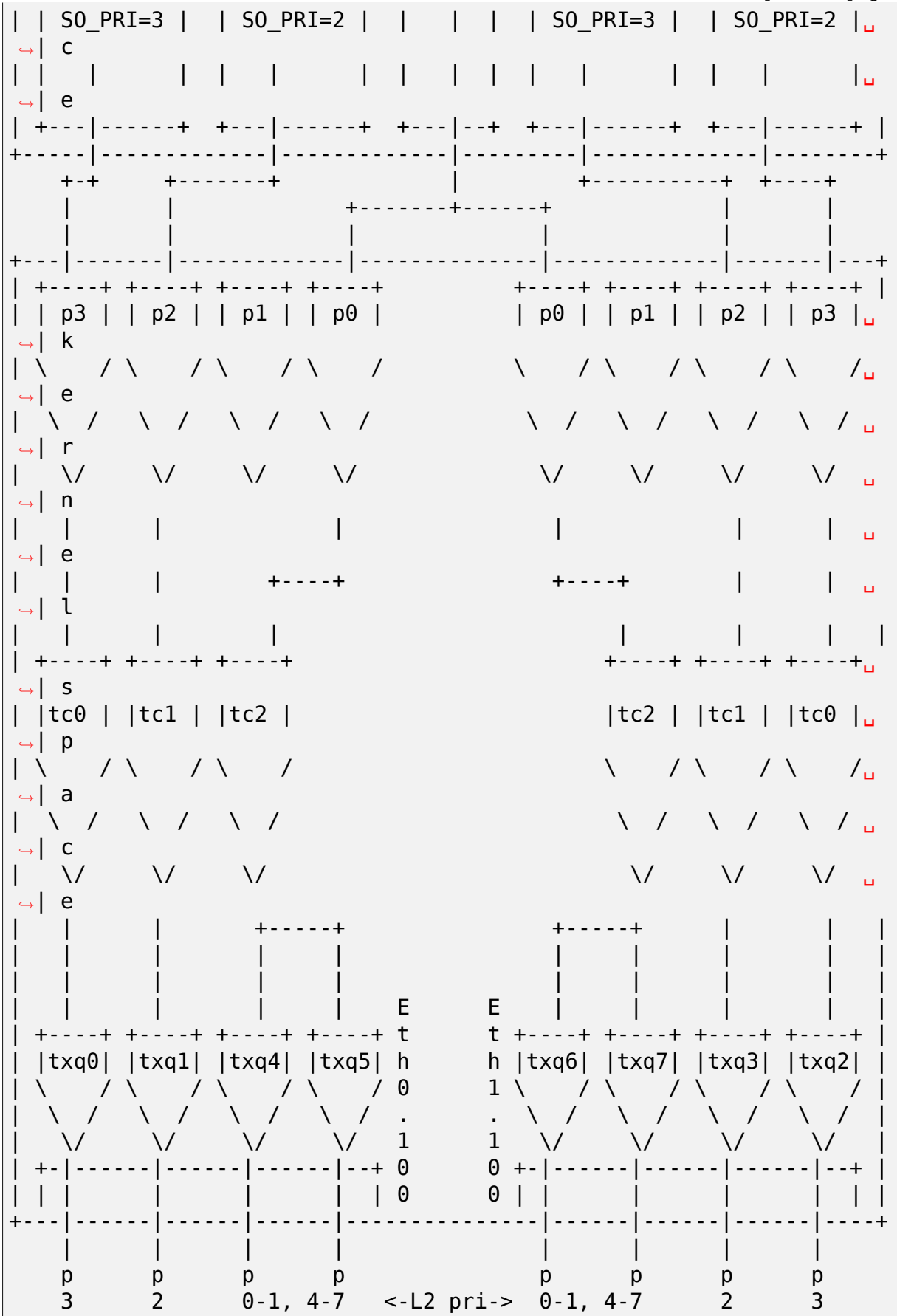
Example 2: Two port tx AVB configuration scheme for target board

(prints and scheme for AM572x evm, for dual emac boards only)



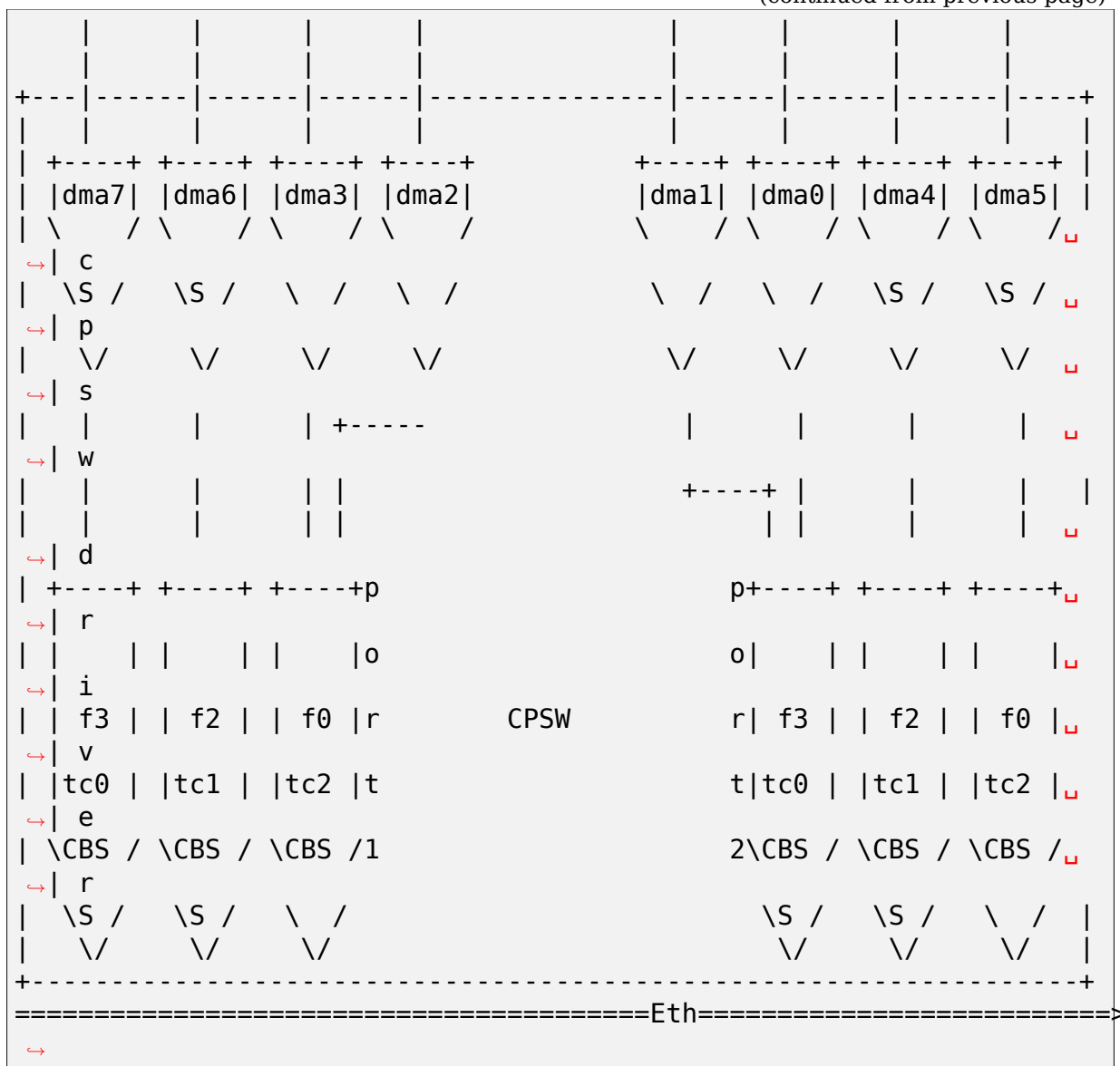
(continues on next page)

(continued from previous page)



(continues on next page)

(continued from previous page)



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

(continues on next page)

(continued from previous page)

```
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
    // for Eth0 and 30 and 10 Mb/s for Eth1.
    // Real speed can differ a bit due to discreteness
    // 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 discreteness
$ tc qdisc add dev eth0 parent 100:1 cbs locredit -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 locredit -1470 \
hicredit 65 sendslope -979000 idleslope 21000 offload 1
net eth0: set FIFO2 bw = 30
```
- 9)

```
// Create vlan 100 to map skb->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 mqprio 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) mqprio
|   +---(100:7) mqprio
|   +---(100:8) mqprio
|
+---(100:ffe1) mqprio
|   +---(100:4) mqprio
|
+---(100:ffe0) mqprio
|   +---(100:3) mqprio
```

- 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)
↪ html)
./tsn_listener -d 18:03:73:66:87:42 -i enp5s0 -s 1500
Receiving data rate: 39012 kbps
Receiving data rate: 39012 kbps
Receiving data rate: 39012 kbps
Receiving data rate: 39012 kbps
Receiving data rate: 39012 kbps
Receiving data rate: 39012 kbps
```

(continues on next page)

(continued from previous page)

```
Receiving data rate: 39012 kbps
Receiving data rate: 39012 kbps
Receiving data rate: 39012 kbps
Receiving data rate: 39012 kbps
Receiving data rate: 39012 kbps
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
```

7.5.39 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-ethernet 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 is

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 make 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
    udhchc -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/
```

(continues on next page)

(continued from previous page)

```

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

```

7.5.40 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 10Mbps operation, and speed=100 to force 100Mbps operation. (I'm not sure what will happen if a card which only supports 10Mbps is forced into 100Mbps 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	au
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>

7.5.41 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 `GRMFLINT` 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 descs. 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 descs 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 descs=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 descs with stat=xa0800000
net eth1: HW next desc (GDACNEXTDA) is at 1
net eth1: Have 127 descs with stat=x40800101
net eth1: Have 1 descs with stat=x40800001
net eth1: Have 126 descs with stat=x40800101
net eth1: Last 1 descs 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 1 KBytes, 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.

7.6 Fiber Distributed Data Interface (FDDI) Device Drivers

Contents:

7.6.1 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 preoccupied 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@linux-mips.org>

7.6.2 SysKonnnect driver - SKFP

© Copyright 1998-2000 SysKonnnect,

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: SysKonnnect 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: SysKonnnect 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 SysKonnnect web site:

www.syskonnnect.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@syskonnnect.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.

7.7 Amateur Radio Device Drivers

Contents:

7.7.1 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:

ser1	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.
pic-par	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).
epp1	This is a redesign of the EPP adaptor.
pga	

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
sethdslc -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
sethdslc -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 `sethdslc -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`).

`ser1` 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

7.7.2 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              #
mode nrzi         #      installed (1)
                  # 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:

```
dllbke-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	

↪ ---							
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 “`sccparam`” :

```
sccparam <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 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: jreuter@yaina.de
	WWW : http://yaina.de/jreuter

7.8 Classic WAN Device Drivers

Contents:

7.8.1 Z8530 Programming Guide

Author

Alan Cox

Introduction

The Z85x30 family synchronous/asynchronous controller chips are used on a large number of cheap network interface cards. The kernel provides a core interface layer that is designed to make it easy to provide WAN services using this chip.

The current driver only support synchronous operation. Merging the asynchronous driver support into this code to allow any Z85x30 device to be used as both a tty interface and as a synchronous controller is a project for Linux post the 2.4 release

Driver Modes

The Z85230 driver layer can drive Z8530, Z85C30 and Z85230 devices in three different modes. Each mode can be applied to an individual channel on the chip (each chip has two channels).

The PIO synchronous mode supports the most common Z8530 wiring. Here the chip is interface to the I/O and interrupt facilities of the host machine but not to the DMA subsystem. When running PIO the Z8530 has extremely tight timing requirements. Doing high speeds, even with a Z85230 will be tricky. Typically you should expect to achieve at best 9600 baud with a Z8C530 and 64Kbits with a Z85230.

The DMA mode supports the chip when it is configured to use dual DMA channels on an ISA bus. The better cards tend to support this mode of operation for a single channel. With DMA running the Z85230 tops out when it starts to hit ISA DMA constraints at about 512Kbits. It is worth noting here that many PC machines hang or crash when the chip is driven fast enough to hold the ISA bus solid.

Transmit DMA mode uses a single DMA channel. The DMA channel is used for transmission as the transmit FIFO is smaller than the receive FIFO. it gives better performance than pure PIO mode but is nowhere near as ideal as pure DMA mode.

Using the Z85230 driver

The Z85230 driver provides the back end interface to your board. To configure a Z8530 interface you need to detect the board and to identify its ports and interrupt resources. It is also your problem to verify the resources are available.

Having identified the chip you need to fill in a struct `z8530_dev`, which describes each chip. This object must exist until you finally shutdown the board. Firstly zero the active field. This ensures nothing goes off without you intending it. The `irq` field should be set to the interrupt number of the chip. (Each chip has a single interrupt source rather than each channel). You are responsible for allocating the interrupt line. The interrupt handler should be set to `z8530_interrupt()`. The device id should be set to the `z8530_dev` structure pointer. Whether the interrupt can be shared or not is board dependent, and up to you to initialise.

The structure holds two channel structures. Initialise `chanA.ctrlrio` and `chanA.dataio` with the address of the control and data ports. You can or this with `Z8530_PORT_SLEEP` to indicate your interface needs the 5uS delay for chip settling done in software. The `PORT_SLEEP` option is architecture specific. Other flags may become available on future platforms, eg for MMIO. Initialise the `chanA.irqs` to `&z8530_nop` to start the chip up as disabled and discarding interrupt events. This ensures that stray interrupts will be mopped up and not hang the bus. Set `chanA.dev` to point to the device structure itself. The private and name field you may use as you wish. The private field is unused by the Z85230 layer. The name is used for error reporting and it may thus make sense to make it match the network name.

Repeat the same operation with the B channel if your chip has both channels wired to something useful. This isn't always the case. If it is not wired then the I/O values do not matter, but you must initialise `chanB.dev`.

If your board has DMA facilities then initialise the `txdma` and `rxdma` fields for the relevant channels. You must also allocate the ISA DMA channels and do any necessary board level initialisation to configure them. The low level driver will do the Z8530 and DMA controller programming but not board specific magic.

Having initialised the device you can then call `z8530_init()`. This will probe the chip and reset it into a known state. An identification sequence is then run to identify the chip type. If the checks fail to pass the function returns a non zero error code. Typically this indicates that the port given is not valid. After this call the `type` field of the `z8530_dev` structure is initialised to either Z8530, Z85C30 or Z85230 according to the chip found.

Once you have called `z8530_init` you can also make use of the utility function `z8530_describe()`. This provides a consistent reporting format for the Z8530 devices, and allows all the drivers to provide consistent reporting.

Attaching Network Interfaces

If you wish to use the network interface facilities of the driver, then you need to attach a network device to each channel that is present and in use. In addition to use the generic HDLC you need to follow some additional plumbing rules. They may seem complex but a look at the example `hostess_sv11` driver should reassure you.

The network device used for each channel should be pointed to by the `netdevice` field of each channel. The `hdlc->priv` field of the network device points to your private data - you will need to be able to find your private data from this.

The way most drivers approach this particular problem is to create a structure holding the Z8530 device definition and put that into the private field of the network device. The network device fields of the channels then point back to the network devices.

If you wish to use the generic HDLC then you need to register the HDLC device.

Before you register your network device you will also need to provide suitable handlers for most of the network device callbacks. See the network device documentation for more details on this.

Configuring And Activating The Port

The Z85230 driver provides helper functions and tables to load the port registers on the Z8530 chips. When programming the register settings for a channel be aware that the documentation recommends initialisation orders. Strange things happen when these are not followed.

`z8530_channel_load()` takes an array of pairs of initialisation values in an array of `u8` type. The first value is the Z8530 register number. Add 16 to indicate the alternate register bank on the later chips. The array is terminated by a 255.

The driver provides a pair of public tables. The `z8530_hdlc_kilostream` table is for the UK 'Kilostream' service and also happens to cover most other end host configurations. The `z8530_hdlc_kilostream_85230` table is the same configuration using the enhancements of the 85230 chip. The configuration loaded is standard

NRZ encoded synchronous data with HDLC bitstuffing. All of the timing is taken from the other end of the link.

When writing your own tables be aware that the driver internally tracks register values. It may need to reload values. You should therefore be sure to set registers 1-7, 9-11, 14 and 15 in all configurations. Where the register settings depend on DMA selection the driver will update the bits itself when you open or close. Loading a new table with the interface open is not recommended.

There are three standard configurations supported by the core code. In PIO mode the interface is programmed up to use interrupt driven PIO. This places high demands on the host processor to avoid latency. The driver is written to take account of latency issues but it cannot avoid latencies caused by other drivers, notably IDE in PIO mode. Because the drivers allocate buffers you must also prevent MTU changes while the port is open.

Once the port is open it will call the `rx_function` of each channel whenever a completed packet arrived. This is invoked from interrupt context and passes you the channel and a network buffer (`struct sk_buff`) holding the data. The data includes the CRC bytes so most users will want to trim the last two bytes before processing the data. This function is very timing critical. When you wish to simply discard data the support code provides the function `z8530_null_rx()` to discard the data.

To active PIO mode sending and receiving the `z8530_sync_open` is called. This expects to be passed the network device and the channel. Typically this is called from your network device open callback. On a failure a non zero error status is returned. The `z8530_sync_close()` function shuts down a PIO channel. This must be done before the channel is opened again and before the driver shuts down and unloads.

The ideal mode of operation is dual channel DMA mode. Here the kernel driver will configure the board for DMA in both directions. The driver also handles ISA DMA issues such as controller programming and the memory range limit for you. This mode is activated by calling the `z8530_sync_dma_open()` function. On failure a non zero error value is returned. Once this mode is activated it can be shut down by calling the `z8530_sync_dma_close()`. You must call the close function matching the open mode you used.

The final supported mode uses a single DMA channel to drive the transmit side. As the Z85C30 has a larger FIFO on the receive channel this tends to increase the maximum speed a little. This is activated by calling the `z8530_sync_txdma_open`. This returns a non zero error code on failure. The `z8530_sync_txdma_close()` function closes down the Z8530 interface from this mode.

Network Layer Functions

The Z8530 layer provides functions to queue packets for transmission. The driver internally buffers the frame currently being transmitted and one further frame (in order to keep back to back transmission running). Any further buffering is up to the caller.

The function `z8530_queue_xmit()` takes a network buffer in `sk_buff` format and queues it for transmission. The caller must provide the entire packet with the exception of the bitstuffing and CRC. This is normally done by the caller via the generic HDLC interface layer. It returns 0 if the buffer has been queued and non zero values for queue full. If the function accepts the buffer it becomes property of the Z8530 layer and the caller should not free it.

The function `z8530_get_stats()` returns a pointer to an internally maintained per interface statistics block. This provides most of the interface code needed to implement the network layer `get_stats` callback.

Porting The Z8530 Driver

The Z8530 driver is written to be portable. In DMA mode it makes assumptions about the use of ISA DMA. These are probably warranted in most cases as the Z85230 in particular was designed to glue to PC type machines. The PIO mode makes no real assumptions.

Should you need to retarget the Z8530 driver to another architecture the only code that should need changing are the port I/O functions. At the moment these assume PC I/O port accesses. This may not be appropriate for all platforms. Replacing `z8530_read_port()` and `z8530_write_port` is intended to be all that is required to port this driver layer.

Known Bugs And Assumptions

Interrupt Locking

The locking in the driver is done via the global `cli/sti` lock. This makes for relatively poor SMP performance. Switching this to use a per device spin lock would probably materially improve performance.

Occasional Failures

We have reports of occasional failures when run for very long periods of time and the driver starts to receive junk frames. At the moment the cause of this is not clear.

Public Functions Provided

`irqreturn_t z8530_interrupt(int irq, void *dev_id)`

Handle an interrupt from a Z8530

Parameters

int irq

Interrupt number

void *dev_id

The Z8530 device that is interrupting.

A Z85[2]30 device has stuck its hand in the air for attention. We scan both the channels on the chip for events and then call the channel specific call backs for each channel that has events. We have to use callback functions because the two channels can be in different modes.

Locking is done for the handlers. Note that locking is done at the chip level (the 5uS delay issue is per chip not per channel). `c->lock` for both channels points to `dev->lock`

`int z8530_sync_open(struct net_device *dev, struct z8530_channel *c)`

Open a Z8530 channel for PIO

Parameters

struct net_device *dev

The network interface we are using

struct z8530_channel *c

The Z8530 channel to open in synchronous PIO mode

Switch a Z8530 into synchronous mode without DMA assist. We raise the RTS/DTR and commence network operation.

`int z8530_sync_close(struct net_device *dev, struct z8530_channel *c)`

Close a PIO Z8530 channel

Parameters

struct net_device *dev

Network device to close

struct z8530_channel *c

Z8530 channel to disassociate and move to idle

Close down a Z8530 interface and switch its interrupt handlers to discard future events.

`int z8530_sync_dma_open(struct net_device *dev, struct z8530_channel *c)`

Open a Z8530 for DMA I/O

Parameters

struct net_device *dev

The network device to attach

struct z8530_channel *c

The Z8530 channel to configure in sync DMA mode.

Set up a Z85x30 device for synchronous DMA in both directions. Two ISA DMA channels must be available for this to work. We assume ISA DMA driven I/O and PC limits on access.

int **z8530_sync_dma_close**(struct *net_device* *dev, struct z8530_channel *c)
Close down DMA I/O

Parameters

struct net_device *dev
Network device to detach

struct z8530_channel *c
Z8530 channel to move into discard mode

Shut down a DMA mode synchronous interface. Halt the DMA, and free the buffers.

int **z8530_sync_txdma_open**(struct *net_device* *dev, struct z8530_channel *c)
Open a Z8530 for TX driven DMA

Parameters

struct net_device *dev
The network device to attach

struct z8530_channel *c
The Z8530 channel to configure in sync DMA mode.

Set up a Z85x30 device for synchronous DMA transmission. One ISA DMA channel must be available for this to work. The receive side is run in PIO mode, but then it has the bigger FIFO.

int **z8530_sync_txdma_close**(struct *net_device* *dev, struct z8530_channel *c)
Close down a TX driven DMA channel

Parameters

struct net_device *dev
Network device to detach

struct z8530_channel *c
Z8530 channel to move into discard mode

Shut down a DMA/PIO split mode synchronous interface. Halt the DMA, and free the buffers.

void **z8530_describe**(struct z8530_dev *dev, char *mapping, unsigned long io)
Uniformly describe a Z8530 port

Parameters

struct z8530_dev *dev
Z8530 device to describe

char *mapping
string holding mapping type (eg "I/O" or "Mem")

unsigned long io
the port value in question

Describe a Z8530 in a standard format. We must pass the I/O as the port offset isn't predictable. The main reason for this function is to try and get a common format of report.

int **z8530_init**(struct z8530_dev *dev)

Initialise a Z8530 device

Parameters

struct z8530_dev *dev

Z8530 device to initialise.

Configure up a Z8530/Z85C30 or Z85230 chip. We check the device is present, identify the type and then program it to hopefully keep quite and behave. This matters a lot, a Z8530 in the wrong state will sometimes get into stupid modes generating 10Khz interrupt streams and the like.

We set the interrupt handler up to discard any events, in case we get them during reset or setp.

Return 0 for success, or a negative value indicating the problem in errno form.

int **z8530_shutdown**(struct z8530_dev *dev)

Shutdown a Z8530 device

Parameters

struct z8530_dev *dev

The Z8530 chip to shutdown

We set the interrupt handlers to silence any interrupts. We then reset the chip and wait 100uS to be sure the reset completed. Just in case the caller then tries to do stuff.

This is called without the lock held

int **z8530_channel_load**(struct z8530_channel *c, u8 *rtable)

Load channel data

Parameters

struct z8530_channel *c

Z8530 channel to configure

u8 *rtable

table of register, value pairs FIXME: ioctl to allow user uploaded tables

Load a Z8530 channel up from the system data. We use +16 to indicate the "prime" registers. The value 255 terminates the table.

void **z8530_null_rx**(struct z8530_channel *c, struct *sk_buff* *skb)

Discard a packet

Parameters

struct z8530_channel *c

The channel the packet arrived on

struct sk_buff *skb

The buffer

We point the receive handler at this function when idle. Instead of processing the frames we get to throw them away.

`netdev_tx_t z8530_queue_xmit(struct z8530_channel *c, struct sk_buff *skb)`

Queue a packet

Parameters

struct z8530_channel *c

The channel to use

struct sk_buff *skb

The packet to kick down the channel

Queue a packet for transmission. Because we have rather hard to hit interrupt latencies for the Z85230 per packet even in DMA mode we do the flip to DMA buffer if needed here not in the IRQ.

Called from the network code. The lock is not held at this point.

Internal Functions

`int z8530_read_port(unsigned long p)`

Architecture specific interface function

Parameters

unsigned long p

port to read

Provided port access methods. The Control SV11 requires no delays between accesses and uses PC I/O. Some drivers may need a 5uS delay

In the longer term this should become an architecture specific section so that this can become a generic driver interface for all platforms. For now we only handle PC I/O ports with or without the dread 5uS sanity delay.

The caller must hold sufficient locks to avoid violating the horrible 5uS delay rule.

`void z8530_write_port(unsigned long p, u8 d)`

Architecture specific interface function

Parameters

unsigned long p

port to write

u8 d

value to write

Write a value to a port with delays if need be. Note that the caller must hold locks to avoid read/writes from other contexts violating the 5uS rule

In the longer term this should become an architecture specific section so that this can become a generic driver interface for all platforms. For now we only handle PC I/O ports with or without the dread 5uS sanity delay.

u8 **read_zsreg**(struct z8530_channel *c, u8 reg)

Read a register from a Z85230

Parameters

struct z8530_channel *c

Z8530 channel to read from (2 per chip)

u8 reg

Register to read FIXME: Use a spinlock.

Most of the Z8530 registers are indexed off the control registers. A read is done by writing to the control register and reading the register back. The caller must hold the lock

u8 **read_zsdata**(struct z8530_channel *c)

Read the data port of a Z8530 channel

Parameters

struct z8530_channel *c

The Z8530 channel to read the data port from

The data port provides fast access to some things. We still have all the 5uS delays to worry about.

void **write_zsreg**(struct z8530_channel *c, u8 reg, u8 val)

Write to a Z8530 channel register

Parameters

struct z8530_channel *c

The Z8530 channel

u8 reg

Register number

u8 val

Value to write

Write a value to an indexed register. The caller must hold the lock to honour the irritating delay rules. We know about register 0 being fast to access.

Assumes c->lock is held.

void **write_zsctrl**(struct z8530_channel *c, u8 val)

Write to a Z8530 control register

Parameters

struct z8530_channel *c

The Z8530 channel

u8 val

Value to write

Write directly to the control register on the Z8530

void **write_zsdata**(struct z8530_channel *c, u8 val)

Write to a Z8530 control register

Parameters**struct z8530_channel *c**

The Z8530 channel

u8 val

Value to write

Write directly to the data register on the Z8530

void z8530_flush_fifo(struct z8530_channel *c)

Flush on chip RX FIFO

Parameters**struct z8530_channel *c**

Channel to flush

Flush the receive FIFO. There is no specific option for this, we blindly read bytes and discard them. Reading when there is no data is harmless. The 8530 has a 4 byte FIFO, the 85230 has 8 bytes.

All locking is handled for the caller. On return data may still be present if it arrived during the flush.

void z8530_rtsdtr(struct z8530_channel *c, int set)

Control the outgoing DTS/RTS line

Parameters**struct z8530_channel *c**

The Z8530 channel to control;

int set

1 to set, 0 to clear

Sets or clears DTR/RTS on the requested line. All locking is handled by the caller. For now we assume all boards use the actual RTS/DTR on the chip. Apparently one or two don't. We'll scream about them later.

void z8530_rx(struct z8530_channel *c)

Handle a PIO receive event

Parameters**struct z8530_channel *c**

Z8530 channel to process

Receive handler for receiving in PIO mode. This is much like the async one but not quite the same or as complex

Note**Its intended that this handler can easily be separated from**

the main code to run realtime. That'll be needed for some machines (eg to ever clock 64kbits on a sparc ;)).

The RT_LOCK macros don't do anything now. Keep the code covered by them as short as possible in all circumstances - clocks cost baud. The interrupt handler is assumed to be atomic w.r.t. to other code - this is true in the RT case too.

We only cover the sync cases for this. If you want 2Mbit async do it yourself but consider medical assistance first. This non DMA synchronous mode is portable code. The DMA mode assumes PCI like ISA DMA

Called with the device lock held

void **z8530_tx**(struct z8530_channel *c)

Handle a PIO transmit event

Parameters

struct z8530_channel *c

Z8530 channel to process

Z8530 transmit interrupt handler for the PIO mode. The basic idea is to attempt to keep the FIFO fed. We fill as many bytes in as possible, its quite possible that we won't keep up with the data rate otherwise.

void **z8530_status**(struct z8530_channel *chan)

Handle a PIO status exception

Parameters

struct z8530_channel *chan

Z8530 channel to process

A status event occurred in PIO synchronous mode. There are several reasons the chip will bother us here. A transmit underrun means we failed to feed the chip fast enough and just broke a packet. A DCD change is a line up or down.

void **z8530_dma_rx**(struct z8530_channel *chan)

Handle a DMA RX event

Parameters

struct z8530_channel *chan

Channel to handle

Non bus mastering DMA interfaces for the Z8x30 devices. This is really pretty PC specific. The DMA mode means that most receive events are handled by the DMA hardware. We get a kick here only if a frame ended.

void **z8530_dma_tx**(struct z8530_channel *chan)

Handle a DMA TX event

Parameters

struct z8530_channel *chan

The Z8530 channel to handle

We have received an interrupt while doing DMA transmissions. It shouldn't happen. Scream loudly if it does.

void **z8530_dma_status**(struct z8530_channel *chan)

Handle a DMA status exception

Parameters

struct z8530_channel *chan

Z8530 channel to process

A status event occurred on the Z8530. We receive these for two reasons when in DMA mode. Firstly if we finished a packet transfer we get one and kick the next packet out. Secondly we may see a DCD change.

void **z8530_rx_clear**(struct z8530_channel *c)

Handle RX events from a stopped chip

Parameters

struct z8530_channel *c

Z8530 channel to shut up

Receive interrupt vectors for a Z8530 that is in 'parked' mode. For machines with PCI Z85x30 cards, or level triggered interrupts (eg the MacII) we must clear the interrupt cause or die.

void **z8530_tx_clear**(struct z8530_channel *c)

Handle TX events from a stopped chip

Parameters

struct z8530_channel *c

Z8530 channel to shut up

Transmit interrupt vectors for a Z8530 that is in 'parked' mode. For machines with PCI Z85x30 cards, or level triggered interrupts (eg the MacII) we must clear the interrupt cause or die.

void **z8530_status_clear**(struct z8530_channel *chan)

Handle status events from a stopped chip

Parameters

struct z8530_channel *chan

Z8530 channel to shut up

Status interrupt vectors for a Z8530 that is in 'parked' mode. For machines with PCI Z85x30 cards, or level triggered interrupts (eg the MacII) we must clear the interrupt cause or die.

void **z8530_tx_begin**(struct z8530_channel *c)

Begin packet transmission

Parameters

struct z8530_channel *c

The Z8530 channel to kick

This is the speed sensitive side of transmission. If we are called and no buffer is being transmitted we commence the next buffer. If nothing is queued we idle the sync.

Note

We are handling this code path in the interrupt path, keep it fast or bad things will happen.

Called with the lock held.

void **z8530_tx_done**(struct z8530_channel *c)

TX complete callback

Parameters

struct z8530_channel *c

The channel that completed a transmit.

This is called when we complete a packet send. We wake the queue, start the next packet going and then free the buffer of the existing packet. This code is fairly timing sensitive.

Called with the register lock held.

void **z8530_rx_done**(struct z8530_channel *c)

Receive completion callback

Parameters

struct z8530_channel *c

The channel that completed a receive

A new packet is complete. Our goal here is to get back into receive mode as fast as possible. On the Z85230 we could change to using ESCC mode, but on the older chips we have no choice. We flip to the new buffer immediately in DMA mode so that the DMA of the next frame can occur while we are copying the previous buffer to an `sk_buff`

Called with the lock held

int **spans_boundary**(struct *sk_buff* *skb)

Check a packet can be ISA DMA' d

Parameters

struct sk_buff *skb

The buffer to check

Returns true if the buffer cross a DMA boundary on a PC. The poor thing can only DMA within a 64K block not across the edges of it.

7.9 Wi-Fi Device Drivers

Contents:

7.9.1 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

7.9.2 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 know. 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

7.9.3 Raylink wireless LAN card

September 21, 1999

Copyright © 1998 Corey Thomas (corey@world.std.com)

This file is the documentation for the Raylink Wireless LAN card driver for Linux. The Raylink wireless LAN card is a PCMCIA card which provides IEEE 802.11 compatible wireless network connectivity at 1 and 2 megabits/second. See <http://www.raytheon.com/micro/raylink/> for more information on the Raylink card. This driver is in early development and does have bugs. See the known bugs and limitations at the end of this document for more information. This driver also works with WebGear's Aviator 2.4 and Aviator Pro wireless LAN cards.

As of kernel 2.3.18, the ray_cs driver is part of the Linux kernel source. My web page for the development of ray_cs is at <http://web.ralinktech.com/ralink/Home/Support/Linux.html> and I can be emailed at corey@world.std.com

The kernel driver is based on ray_cs-1.62.tgz

The driver at my web page is intended to be used as an add on to David Hinds pcmcia package. All the command line parameters are available when compiled as a module. When built into the kernel, only the essid= string parameter is available via the kernel command line. This will change after the method of sorting out parameters for all the PCMCIA drivers is agreed upon. If you must have a built in driver with nondefault parameters, they can be edited in /usr/src/linux/drivers/net/pcmcia/ray_cs.c. Searching for module_param will find them all.

Information on card services is available at:

<http://pcmcia-cs.sourceforge.net/>

Card services user programs are still required for PCMCIA devices. pcmcia-cs-3.1.1 or greater is required for the kernel version of the driver.

Currently, ray_cs is not part of David Hinds card services package, so the following magic is required.

At the end of the /etc/pcmcia/config.opts file, add the line: source ./ray_cs.opts
This will make card services read the ray_cs.opts file when starting. Create the file /etc/pcmcia/ray_cs.opts containing the following:

```
#### start of /etc/pcmcia/ray_cs.opts #####
# Configuration options for Raylink Wireless LAN PCMCIA card
device "ray_cs"
    class "network" module "misc/ray_cs"

card "RayLink PC Card WLAN Adapter"
    manfid 0x01a6, 0x0000
    bind "ray_cs"

module "misc/ray_cs" opts ""
#### end of /etc/pcmcia/ray_cs.opts #####
```

To join an existing network with different parameters, contact the network administrator for the configuration information, and edit /etc/pcmcia/ray_cs.opts. Add the parameters below between the empty quotes.

Parameters for ray_cs driver which may be specified in ray_cs.opts:

bc	in- te- ger	0 = normal mode (802.11 timing), 1 = slow down inter frame timing to allow operation with older breezecom access points.
bea- con_pe	in- te- ger	beacon period in Kilo-microseconds, legal values = must be integer multiple of hop dwell default = 256
coun- try	in- te- ger	1 = USA (default), 2 = Europe, 3 = Japan, 4 = Korea, 5 = Spain, 6 = France, 7 = Israel, 8 = Australia
essid	strir	ESS ID - network name to join string with maximum length of 32 chars default value = "AD-HOC_ESSID"
hop_dw	in- te- ger	hop dwell time in Kilo-microseconds legal values = 16,32,64,128(default),256
irq_ma	in- te- ger	linux standard 16 bit value 1bit/IRQ lsb is IRQ 0, bit 1 is IRQ 1 etc. Used to restrict choice of IRQ's to use. Recommended method for controlling interrupts is in /etc/pcmcia/config.opts
net_ty	in- te- ger	0 (default) = adhoc network, 1 = infrastructure
phy_ad	strir	string containing new MAC address in hex, must start with x eg x00008f123456
psm	in- te- ger	0 = continuously active, 1 = power save mode (not useful yet)
pc_deb	in- te- ger	(0-5) larger values for more verbose logging. Replaces ray_debug.
ray_de	in- te- ger	Replaced with pc_debug
ray_mε	in- te- ger	defaults to 500
snif- fer	in- te- ger	0 = not sniffer (default), 1 = sniffer which can be used to record all network traffic using tcpdump or similar, but no normal network use is allowed.
trans- late	in- te- ger	0 = no translation (encapsulate frames), 1 = translation (RFC1042/802.1)

More on sniffer mode:

tcpdump does not understand 802.11 headers, so it can't interpret the contents, but it can record to a file. This is only useful for debugging 802.11 lowlevel protocols that are not visible to linux. If you want to watch ftp xfers, or do similar things, you don't need to use sniffer mode. Also, some packet types are never sent up by the card, so you will never see them (ack, rts, cts, probe etc.) There

is a simple program (showcap) included in the ray_cs package which parses the 802.11 headers.

Known Problems and missing features

Does not work with non x86

Does not work with SMP

Support for defragmenting frames is not yet debugged, and in fact is known to not work. I have never encountered a net set up to fragment, but still, it should be fixed.

The ioctl support is incomplete. The hardware address cannot be set using ifconfig yet. If a different hardware address is needed, it may be set using the phy_addr parameter in ray_cs.opts. This requires a card insertion to take effect.

DISTRIBUTED SWITCH ARCHITECTURE

8.1 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.

8.1.1 Design principles

The Distributed Switch Architecture is a subsystem which was primarily designed to support Marvell Ethernet switches (MV88E6xxx, a.k.a Linkstreet 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 is typically comprised of multiple front-panel ports, and one or more CPU or management port. 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-the rack switches. This host Ethernet controller will be later referred to as “master” 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 will create 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 “slave” 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 frames it received 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: master 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

Switch tagging protocols

DSA currently supports 5 different tagging protocols, and a tag-less mode as well. The different protocols are implemented in:

- `net/dsa/tag_trailer.c`: Marvell’ s 4 trailer tag mode (legacy)
- `net/dsa/tag_dsa.c`: Marvell’ s original DSA tag
- `net/dsa/tag_edsa.c`: Marvell’ s enhanced DSA tag
- `net/dsa/tag_brcm.c`: Broadcom’ s 4 bytes tag
- `net/dsa/tag_qca.c`: Qualcomm’ s 2 bytes tag

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

Master network devices

Master 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 master 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:

Master 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 slave network device
- invoked `netif_receive_skb()`

Past this point, the DSA slave network devices get delivered regular Ethernet frames that can be processed by the networking stack.

Slave network devices

Slave network devices created by DSA are stacked on top of their master network device, each of these network interfaces will be responsible for being a controlling and data-flowing end-point 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...
- external/internal PHY management: link, auto-negotiation etc.

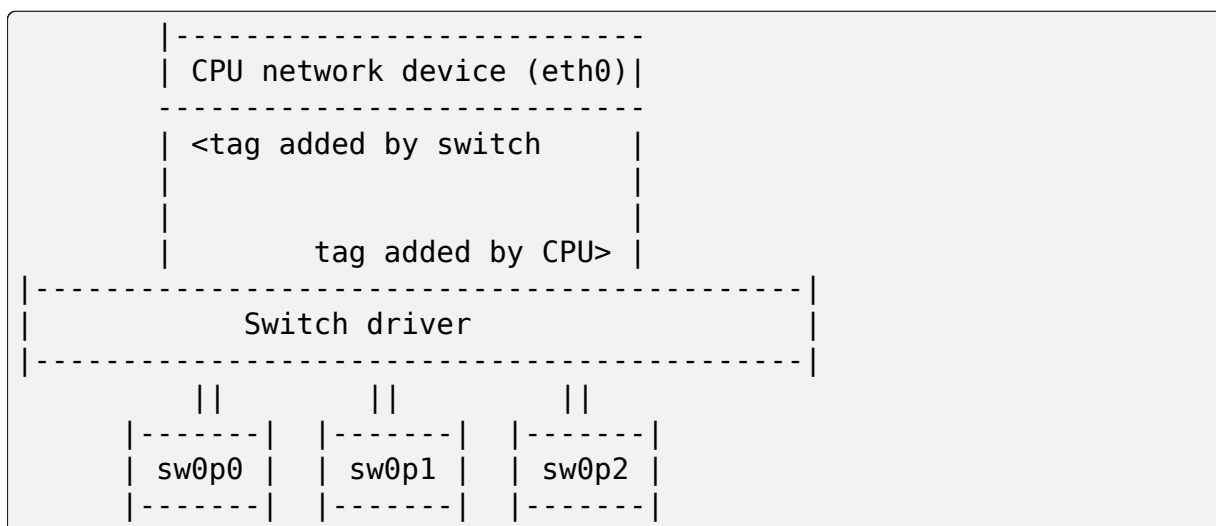
These slave network devices have custom `net_device_ops` and `ethtool_ops` function pointers which allow DSA to introduce a level of layering between the networking stack/ethtool, and the switch driver implementation.

Upon frame transmission from these slave 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 master 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 delivers these frames to the physical switch port.

Graphical representation

Summarized, this is basically how DSA looks like from a network device perspective:



Slave MDIO bus

In order to be able to read to/from a switch PHY built into it, DSA creates a slave 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 busses, the slave 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` structure if multiples switches are cascaded, the master network device this switch tree is attached to needs to be referenced
- `dsa_switch_tree`: structure assigned to the master 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, slave network devices, master network device, and a reference to the backing `dsa_switch_ops`
- `dsa_switch_ops`: structure referencing function pointers, see below for a full description.

8.1.2 Design limitations

Limits on the number of devices and ports

DSA currently limits the number of maximum switches within a tree to 4 (`DSA_MAX_SWITCHES`), and the number of ports per switch to 12 (`DSA_MAX_PORTS`). These limits could be extended to support larger configurations would this need arise.

Lack of CPU/DSA network devices

DSA does not currently create slave 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 master network device is configured to use DSA (`dev->dsa_ptr` becomes non-NULL), 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.

Slave network devices check that the master network device is UP before allowing you to administratively bring UP these slave network devices. A common configuration mistake is forgetting to bring UP the master network device first.

8.1.3 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

MDIO/PHY library

Slave 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_slave_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 slave 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 slave network devices. Since DSA primarily deals with MDIO-connected switches, although not exclusively, SWITCHDEV’s prepare/abort/commit phases are often simplified into a prepare phase which checks whether the operation is supported by the DSA switch driver, and a commit phase which applies the changes.

As of today, the only SWITCHDEV objects supported by DSA are the FDB and VLAN objects.

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..

8.1.4 Driver development

DSA switch drivers need to implement a `dsa_switch_ops` structure which will contain the various members described below.

`register_switch_driver()` registers this `dsa_switch_ops` in its internal list of drivers to probe for. `unregister_switch_driver()` does the exact opposite.

Unless requested differently by setting the `priv_size` member accordingly, DSA does not allocate any driver private context space.

Switch configuration

- `tag_protocol`: this is to indicate what kind of tagging protocol is supported, should be a valid value from the `dsa_tag_protocol` enum
- `probe`: probe routine which will be invoked by the DSA platform device upon registration to test for the presence/absence of a switch device. For MDIO devices, it is recommended to issue a read towards internal registers using the switch pseudo-PHY and return whether this is a supported device. For other buses, return a non-NULL string
- `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.

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-bits 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 slave 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 slave 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 slave 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 QS-GMII, 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 slave 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 master 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 slave network device `ndo_open` function when a port is administratively brought up, this function should be fully enabling 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 slave network device `ndo_close` function when a port is administratively brought down, this function should be fully disabling 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

Bridge layer

- **port_bridge_join**: bridge layer function invoked when a given switch port is added to a bridge, this function should be doing the necessary at the switch level to permit the joining port from being added to the relevant logical domain for it to ingress/egress traffic with other members of the bridge.
- **port_bridge_leave**: bridge layer function invoked when a given switch port is removed from a bridge, this function should be doing the necessary at the switch level to deny the leaving port from ingress/egress traffic from the remaining bridge members. When the port leaves the bridge, it should be aged out at the switch hardware for the switch to (re) learn MAC addresses behind this port.
- **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. The switch driver is responsible for computing a STP state change based on current and asked parameters and perform the relevant ageing based on the intersection results

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_prepare`: bridge layer function invoked when the bridge prepares the configuration of a VLAN on the given port. If the operation is not supported by the hardware, this function should return `-EOPNOTSUPP` to inform the bridge code to fallback to a software implementation. No hardware setup must be done in this function. See `port_vlan_add` for this and details.
- `port_vlan_add`: bridge layer function invoked when a VLAN is configured (tagged or untagged) for the given switch port
- `port_vlan_del`: bridge layer function invoked when a VLAN is removed from the given switch port
- `port_vlan_dump`: bridge layer function invoked with a `switchdev` callback function that the driver has to call for each VLAN the given port is a member of. A `switchdev` object is used to carry the VID and bridge flags.
- `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. If the operation is not supported, this function should return `-EOPNOTSUPP` to inform the bridge code to fallback to a software implementation.

Note: VLAN ID 0 corresponds to the port private database, which, in the context of DSA, would be its port-based VLAN, used by the associated bridge device.

- `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 specified MAC address from the specified VLAN ID if it was mapped into this port forwarding database
- `port_fdb_dump`: bridge layer function invoked with a `switchdev` callback function that the driver has to call for each MAC address known to be behind the given port. A `switchdev` object is used to carry the VID and FDB info.
- `port_mdb_prepare`: bridge layer function invoked when the bridge prepares the installation of a multicast database entry. If the operation is not supported, this function should return `-EOPNOTSUPP` to inform the bridge code to fallback to a software implementation. No hardware setup must be done in this function. See `port_fdb_add` for this and details.
- `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.

Note: VLAN ID 0 corresponds to the port private database, which, in the context of DSA, would be its port-based VLAN, used by the associated bridge device.

- `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.
- `port_mdb_dump`: bridge layer function invoked with a switchdev callback function that the driver has to call for each MAC address known to be behind the given port. A switchdev object is used to carry the VID and MDB info.

8.1.5 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.

Other hanging fruits

- making the number of ports fully dynamic and not dependent on `DSA_MAX_PORTS`
- allowing more than one CPU/management interface: <http://comments.gmane.org/gmane.linux.network/365657>
- porting more drivers from other vendors: <http://comments.gmane.org/gmane.linux.network/365510>

8.2 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.

8.2.1 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 slave 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 master interface needs to be brought up before the slave
↳ 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 slave interfaces
ip link set wan up
ip link set lan1 up
ip link set lan2 up
```

(continues on next page)

(continued from previous page)

```
# 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 master interface needs to be brought up before the slave
↳ ports.
ip link set eth0 up
ip link set eth0.1 up

# bring up the slave 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
```

(continues on next page)

(continued from previous page)

```
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 master interface needs to be brought up before the slave
↳ ports.
ip link set eth0 up
ip link set eth0.1 up
ip link set eth0.2 up

# bring up the slave 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
```

(continues on next page)

(continued from previous page)

```
ip link set br0 up
```

8.3 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/registers:

- 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

8.3.1 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 slave 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.

8.4 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 master network interface (e.g. fixed link).

8.4.1 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.

8.4.2 Driver limitations

- Support for VLAN filtering is not implemented
- The HW does not support VLAN-specific fdb entries

8.5 NXP SJA1105 switch driver

8.5.1 Overview

The NXP SJA1105 is a family of 6 devices:

- 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

These are SPI-managed automotive switches, with all ports being gigabit capable, and supporting MII/RMII/RGMII and optionally SGMII on one port.

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.

8.5.2 Traffic support

The switches do not have hardware support for DSA tags, except for “slow protocols” for switch control as STP and PTP. For these, the switches have two programmable filters for link-local destination MACs. These are used to trap BPDUs and PTP traffic to the master netdevice, and are further used to support STP and 1588 ordinary clock/boundary clock functionality. For frames trapped to the CPU, source port and switch ID information is encoded by the hardware into the frames.

But by leveraging `CONFIG_NET_DSA_TAG_8021Q` (a software-defined DSA tagging format based on VLANs), general-purpose traffic termination through the network stack can be supported under certain circumstances.

Depending on VLAN awareness state, the following operating modes are possible with the switch:

- Mode 1 (VLAN-unaware): a port is in this mode when it is used as a standalone net device, or when it is enslaved to a bridge with `vlan_filtering=0`.

- Mode 2 (fully VLAN-aware): a port is in this mode when it is enslaved to a bridge with `vlan_filtering=1`. Access to the entire VLAN range is given to the user through bridge `vlan` commands, but general-purpose (anything other than STP, PTP etc) traffic termination is not possible through the switch net devices. The other packets can be still by user space processed through the DSA master interface (similar to `DSA_TAG_PROTO_NONE`).
- Mode 3 (best-effort VLAN-aware): a port is in this mode when enslaved to a bridge with `vlan_filtering=1`, and the devlink property of its parent switch named `best_effort_vlan_filtering` is set to `true`. When configured like this, the range of usable VIDs is reduced (0 to 1023 and 3072 to 4094), so is the number of usable VIDs (maximum of 7 non-pvid VLANs per port*), and shared VLAN learning is performed (FDB lookup is done only by DMAC, not also by VID).

To summarize, in each mode, the following types of traffic are supported over the switch net devices:

	Mode 1	Mode 2	Mode 3
Regular traffic	Yes	No (use master)	Yes
Management traffic (BPDU, PTP)	Yes	Yes	Yes

To configure the switch to operate in Mode 3, the following steps can be followed:

```
ip link add dev br0 type bridge
# swp2 operates in Mode 1 now
ip link set dev swp2 master br0
# swp2 temporarily moves to Mode 2
ip link set dev br0 type bridge vlan_filtering 1
[ 61.204770] sja1105 spi0.1: Reset switch and programmed static
↪config. Reason: VLAN filtering
[ 61.239944] sja1105 spi0.1: Disabled switch tagging
# swp3 now operates in Mode 3
devlink dev param set spi/spi0.1 name best_effort_vlan_filtering
↪value true cmode runtime
[ 64.682927] sja1105 spi0.1: Reset switch and programmed static
↪config. Reason: VLAN filtering
[ 64.711925] sja1105 spi0.1: Enabled switch tagging
# Cannot use VLANs in range 1024-3071 while in Mode 3.
bridge vlan add dev swp2 vid 1025 untagged pvid
RTNETLINK answers: Operation not permitted
bridge vlan add dev swp2 vid 100
bridge vlan add dev swp2 vid 101 untagged
bridge vlan
port      vlan ids
swp5      1 PVID Egress Untagged

swp2      1 PVID Egress Untagged
          100
          101 Egress Untagged
```

(continues on next page)

(continued from previous page)

```

swp3      1 PVID Egress Untagged

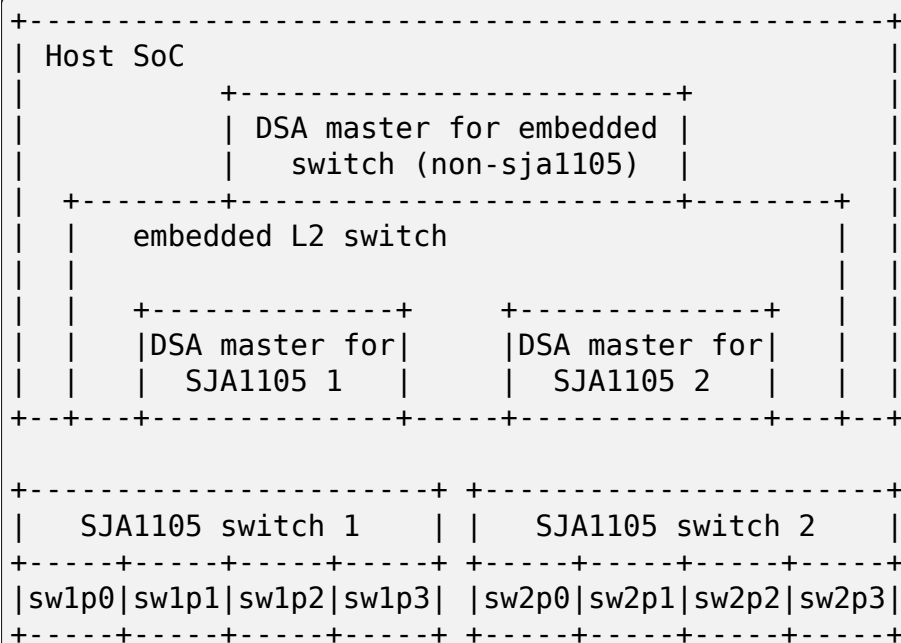
swp4      1 PVID Egress Untagged

br0       1 PVID Egress Untagged
bridge vlan add dev swp2 vid 102
bridge vlan add dev swp2 vid 103
bridge vlan add dev swp2 vid 104
bridge vlan add dev swp2 vid 105
bridge vlan add dev swp2 vid 106
bridge vlan add dev swp2 vid 107
# Cannot use mode than 7 VLANs per port while in Mode 3.
[ 3885.216832] sjal105 spi0.1: No more free subvlans

```

* “maximum of 7 non-pvid VLANs per port” : Decoding VLAN-tagged packets on the CPU in mode 3 is possible through VLAN retagging of packets that go from the switch to the CPU. In cross-chip topologies, the port that goes to the CPU might also go to other switches. In that case, those other switches will see only a retagged packet (which only has meaning for the CPU). So if they are interested in this VLAN, they need to apply retagging in the reverse direction, to recover the original value from it. This consumes extra hardware resources for this switch. There is a maximum of 32 entries in the Retagging Table of each switch device.

As an example, consider this cross-chip topology:



To reach the CPU, SJA1105 switch 1 (spi/spi2.1) uses the same port as is uses to reach SJA1105 switch 2 (spi/spi2.2), which would be port 4 (not drawn). Similarly for SJA1105 switch 2.

Also consider the following commands, that add VLAN 100 to every sjal105 user port:

```

devlink dev param set spi/spi2.1 name best_effort_vlan_filtering_
→value true cmode runtime
devlink dev param set spi/spi2.2 name best_effort_vlan_filtering_
→value true cmode runtime
ip link add dev br0 type bridge
for port in sw1p0 sw1p1 sw1p2 sw1p3 \
        sw2p0 sw2p1 sw2p2 sw2p3; do
    ip link set dev $port master br0
done
ip link set dev br0 type bridge vlan_filtering 1
for port in sw1p0 sw1p1 sw1p2 sw1p3 \
        sw2p0 sw2p1 sw2p2; do
    bridge vlan add dev $port vid 100
done
ip link add link br0 name br0.100 type vlan id 100 && ip link set_
→dev br0.100 up
ip addr add 192.168.100.3/24 dev br0.100
bridge vlan add dev br0 vid 100 self

bridge vlan
port      vlan ids
sw1p0     1 PVID Egress Untagged
          100

sw1p1     1 PVID Egress Untagged
          100

sw1p2     1 PVID Egress Untagged
          100

sw1p3     1 PVID Egress Untagged
          100

sw2p0     1 PVID Egress Untagged
          100

sw2p1     1 PVID Egress Untagged
          100

sw2p2     1 PVID Egress Untagged
          100

sw2p3     1 PVID Egress Untagged

br0       1 PVID Egress Untagged
          100

```

SJA1105 switch 1 consumes 1 retagging entry for each VLAN on each user port towards the CPU. It also consumes 1 retagging entry for each non-pvid VLAN that it is also interested in, which is configured on any port of any neighbor switch.

In this case, SJA1105 switch 1 consumes a total of 11 retagging entries, as follows:

- 8 retagging entries for VLANs 1 and 100 installed on its user ports (sw1p0 - sw1p3)
- 3 retagging entries for VLAN 100 installed on the user ports of SJA1105 switch 2 (sw2p0 - sw2p2), because it also has ports that are interested in it. The VLAN 1 is a pvid on SJA1105 switch 2 and does not need reverse retagging.

SJA1105 switch 2 also consumes 11 retagging entries, but organized as follows:

- 7 retagging entries for the bridge VLANs on its user ports (sw2p0 - sw2p3).
- 4 retagging entries for VLAN 100 installed on the user ports of SJA1105 switch 1 (sw1p0 - sw1p3).

8.5.3 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 are enslaved to 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). Also an inevitable limitation of the fact that VLAN awareness is global at the switch level is that once a bridge with `vlan_filtering` enslaves at least one switch port, the other un-bridged ports are no longer available for standalone traffic termination.

Topology and loop detection through STP is supported.

L2 FDB manipulation (add/delete/dump) is currently possible for the first generation devices. Aging time of FDB entries, as well as enabling fully static management (no address learning and no flooding of unknown traffic) is not yet configurable in the driver.

A special comment about bridging with other netdevices (illustrated with an example):

A board has eth0, eth1, `swp0@eth1`, `swp1@eth1`, `swp2@eth1`, `swp3@eth1`. The switch ports (swp0-3) are under br0. It is desired that eth0 is turned into another switched port that communicates with swp0-3.

If br0 has `vlan_filtering 0`, then eth0 can simply be added to br0 with the intended results. If br0 has `vlan_filtering 1`, then a new br1 interface needs to be created that enslaves eth0 and eth1 (the DSA master of the switch ports). This is because in this mode, the switch ports beneath br0 are not capable of regular traffic, and are only used as a conduit for switchdev operations.

8.5.4 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 master 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() {
```

(continues on next page)

(continued from previous page)

```
    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 $(cat /dev/urandom | fold -w 8 | tr -dc '0-7' | fold -w 1 | tr -d '\n' | paste -s, | sed 's/,/ /g') \
    sched-entry S $(cat /dev/urandom | fold -w 8 | tr -dc '0-7' | fold -w 1 | tr -d '\n' | paste -s, | sed 's/,/ /g') \
    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}"
```

(continues on next page)

(continued from previous page)

```
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.

8.5.5 Device Tree bindings and board design

This section references Documentation/devicetree/bindings/net/dsa/sja1105.txt 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 master'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.

8.6 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.

8.6.1 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 master interface. All other corresponding linux interfaces are called slave interfaces.

The slave interfaces depend on the master interface. They can only be brought up, when the master interface is up.

In this documentation the following Ethernet interfaces are used:

eth0

the master interface

lan1

a slave interface

lan2

another slave interface

lan3

a third slave interface

wan

A slave 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)

8.6.2 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

# The master interface needs to be brought up before the slave
↳ ports.
ip link set eth0 up

# bring up the slave interfaces
ip link set lan1 up
ip link set lan2 up
ip link set lan3 up
```

bridge

```
# The master interface needs to be brought up before the slave
↳ports.
ip link set eth0 up

# bring up the slave 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

```
# The master interface needs to be brought up before the slave
↳ports.
ip link set eth0 up

# bring up the slave 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
```

(continues on next page)

(continued from previous page)

```
# bring up the bridge
ip link set dev br0 up
```

8.6.3 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

# The master interface needs to be brought up before the slave
↳ 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 slave 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
```

(continues on next page)

(continued from previous page)

```
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 master interface needs to be brought up before the slave
# ports.
ip link set eth0 up
ip link set eth0.1 up

# bring up the slave 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

# The master interface needs to be brought up before the slave
↳ ports.
ip link set eth0 up
ip link set eth0.1 up
ip link set eth0.2 up

# bring up the slave 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 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
```


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.

9.1 Interface documentation

The following pages describe various interfaces available through devlink in general.

9.1.1 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 quiet 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,
```

(continues on next page)

(continued from previous page)

```
    meta.l3_forward: set }  
}
```

9.1.2 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 (eg 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
- OOB 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 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

User Interface

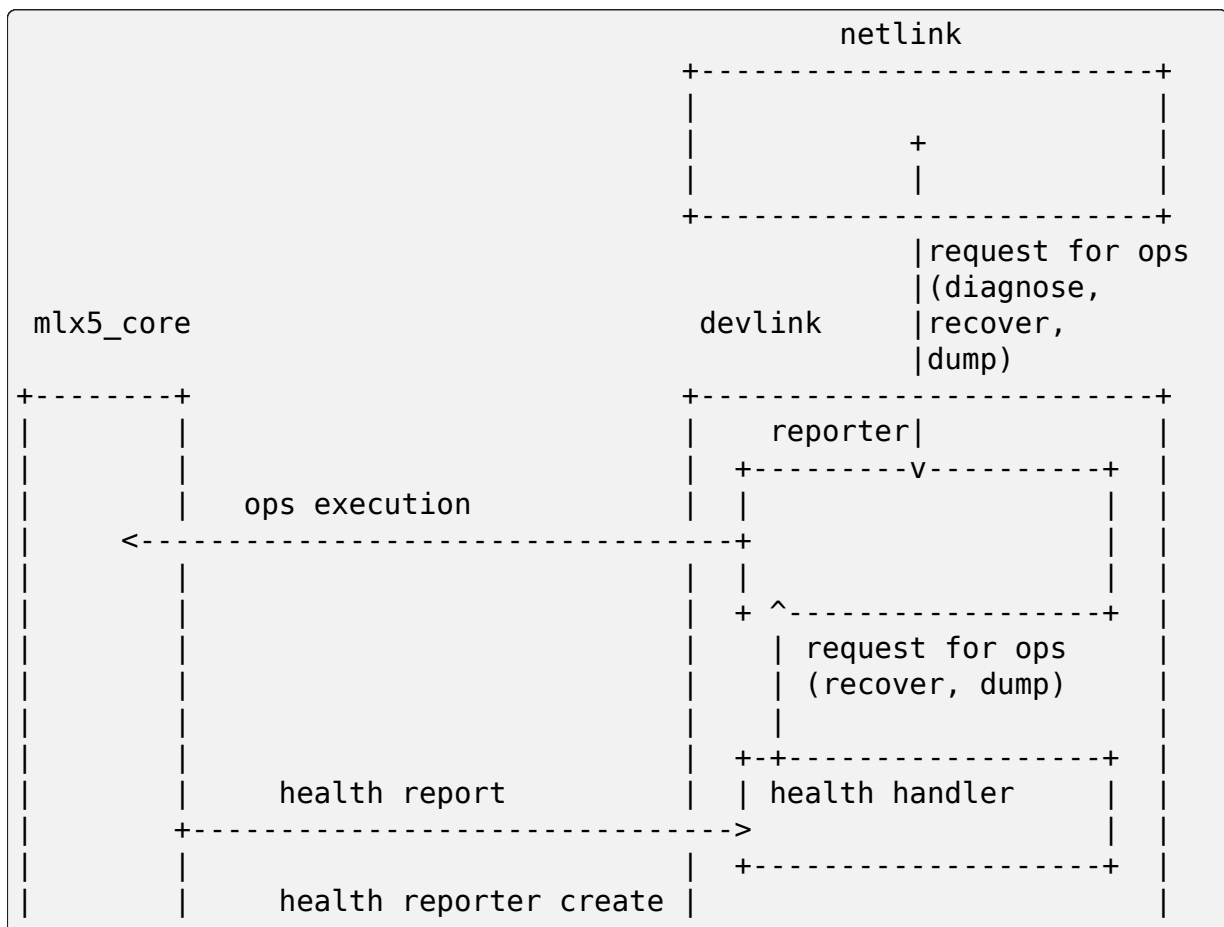
User can access/change each reporter' s parameters and driver specific callbacks via devlink, e.g per error type (per health reporter):

- 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
DEVLIN	Retrieves status and configuration info per DEV and reporter.
DEVLIN	Allows reporter-related configuration setting.
DEVLIN	Triggers a reporter' s recovery procedure.
DEVLIN	Retrieves diagnostics data from a reporter on a device.
DEVLIN	Retrieves the last stored dump. Devlink health saves a single dump. If an dump is not already stored by the devlink for this reporter, devlink generates a new dump. dump output is defined by the reporter.
DEVLIN	Clears the last saved dump file for the specified reporter.

The following diagram provides a general overview of devlink-health:



(continues on next page)

(continued from previous page)

	+	-----	>		
+	-----	+	+	-----	+

9.1.3 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 [Devlink Flash](#));
- 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

Na	Description
dri	Name of the currently used device driver, also available through sysfs.
sei	Serial number 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.
boæ	Board serial number of the device.
sei	This is usually the serial number of the board, often available in PCI <i>Vital Product Data</i> .
fi>	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 Devlink Flash to understand why.
rur	Group for information about currently running software/firmware. These versions often only update after a reboot, sometimes device reset.
stc	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 [Devlink Flash](#)). 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 Devlink Flash). <code>bundle_id</code> can appear in both running and stored versions, but it must not be reported if any of the components covered by the <code>bundle_id</code> 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.

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.

9.1.4 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

Na	Description
DE\	Indicates that the device should overwrite settings in the components being updated with the settings found in the provided image.
DE\	Indicates that the device should 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 ([Devlink Params](#)).

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
```

(continues on next page)

(continued from previous page)

```
if $want_fw_vers != devlink-dev-info['running']:
    devlink-reset()

# Reboot, if reset wasn't enough
if $want_fw_vers != 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.)

9.1.5 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

Na	Description
run	set while the driver is running, and takes effect immediately. No reset is required.
drv	applied while the driver initializes. Requires the user to restart the driver using the <code>devlink reload</code> command.
per	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

Na	Ty	Description
en	Bo	Enable Single Root I/O Virtualization (SRIOV) in the device.
igr	Bo	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.
msi	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.
msi	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	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.
res	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.
en	Bo	Enable handling of RoCE traffic in the device.
int	Bo	When enabled, the device driver will reset the device on internal errors.
max	u32	Specifies the maximum number of MAC addresses per ethernet port of this device.
req	Bo	Enable capture of devlink-region snapshots.
en	Bo	Enable 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.

9.1.6 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.

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]
pci/0000:00:05.0/fw-health: size 64 snapshot [1 2]

# 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
```

(continues on next page)

(continued from previous page)

```
000000000000000020 0016 0bb8 0016 1720 0000 0000 c00f 3ffc
000000000000000030 bada cce5 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
000000000000000000 0014 95dc 0014 9514 0035 1670 0034 db30
```

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.

9.1.7 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.

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.

9.1.8 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 7: Possible reload actions

Name	Description
dri	Devlink driver entities re-initialization, including applying new values to devlink entities which are used during driver load such as devlink-params in configuration mode driverinit or devlink-resources
fw_	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 downtime as needed to perform the actions.

However, some drivers support action limits, which limit the action implementation to specific constraints.

Table 8: Possible reload limits

Name	Description
no_	No 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
```

9.1.9 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.

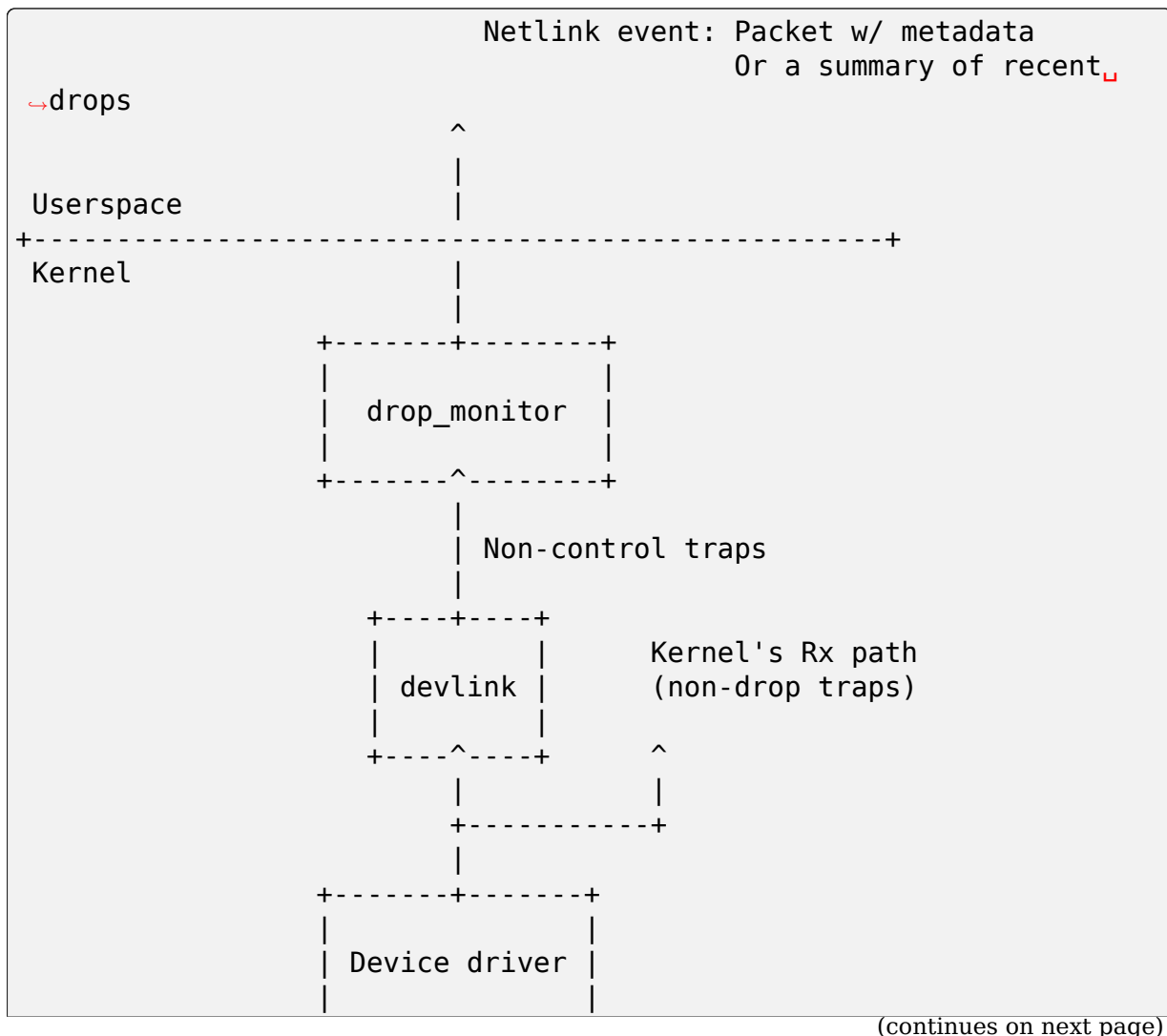
The fundamental ability of sending certain packets to the kernel for processing is called “packet trapping” .

Overview

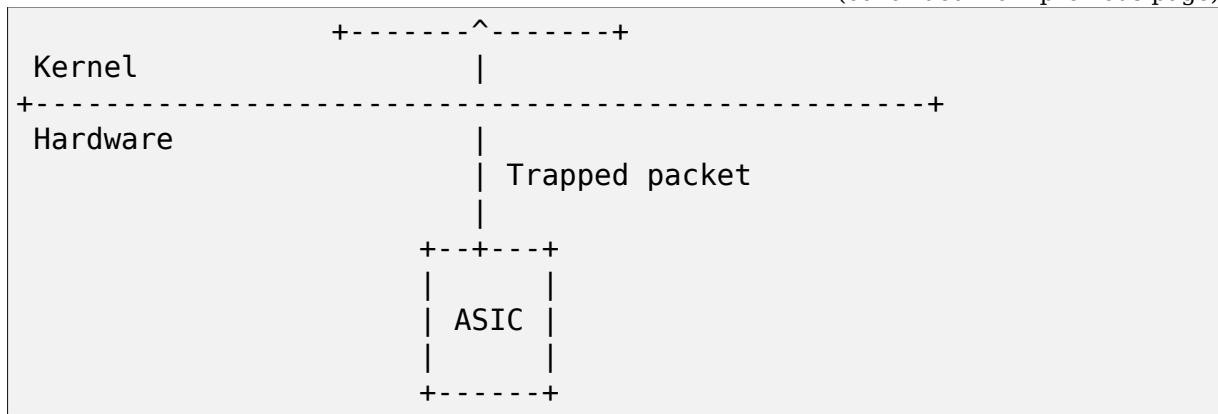
The `devlink-trap` mechanism allows capable device drivers to register their supported packet traps with devlink and report trapped packets to devlink for further analysis.

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`:



(continued from previous page)



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. 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 9: List of Generic Packet Traps

Na	Ty	Description
so	dro	Traps incoming packets that the device decided to drop because of a multicast source MAC
vl	dro	Traps incoming 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
in	dro	Traps incoming packets that the device decided to drop in case they are tagged with a VLAN that is not configured on the ingress bridge port
in	dro	Traps incoming packets that the device decided to drop in case the STP state of the ingress bridge port is not “forwarding”
po	dro	Traps 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
po	dro	Traps 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
bl	dro	Traps packets that the device decided to drop in case they hit a blackhole route
ttl	exc	Traps unicast packets that should be forwarded by the device whose TTL was decremented to 0 or less
ta	dro	Traps packets that the device decided to drop because they could not be enqueued to a transmission queue which is full
no	dro	Traps packets that the device decided to drop because they need to undergo a layer 3 lookup, but are not IP or MPLS packets
uc_	dro	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
di	dro	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)
si	dro	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)
si	dro	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_	dro	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)
ip\	dro	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)

continues on next page

Table 9 – continued from previous page

ip _v dro	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., ff _x 0::/16)
ip _v dro	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., ff _x 1::/16)
mtu exc	Traps packets that should have been routed by the device, but were bigger than the MTU of the egress interface
un _i exc	Traps packets that did not have a matching IP neighbour after routing
mc __ exc	Traps multicast IP packets that failed reverse-path forwarding (RPF) check during multicast routing
rej exc	Traps packets that hit reject routes (i.e., “unreachable” , “prohibit”)
ip _v exc	Traps unicast IPv4 packets that did not match any route
ip _v exc	Traps unicast IPv6 packets that did not match any route
nor dro	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
dec exc	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)
ove dro	Traps NVE packets that the device decided to drop because their overlay source MAC is multicast
ing dro	Traps packets dropped during processing of ingress flow action drop
eg _i dro	Traps packets dropped during processing of egress flow action drop
stp cor	Traps STP packets
lac cor	Traps LACP packets
lld cor	Traps LLDP packets
ign cor	Traps IGMP Membership Query packets
ign cor	Traps IGMP Version 1 Membership Report packets
ign cor	Traps IGMP Version 2 Membership Report packets
ign cor	Traps IGMP Version 3 Membership Report packets
ign cor	Traps IGMP Version 2 Leave Group packets
mlc cor	Traps MLD Multicast Listener Query packets
mlc cor	Traps MLD Version 1 Multicast Listener Report packets
mlc cor	Traps MLD Version 2 Multicast Listener Report packets
mlc cor	Traps MLD Version 1 Multicast Listener Done packets
ip _v cor	Traps IPv4 DHCP packets
ip _v cor	Traps IPv6 DHCP packets
arp cor	Traps ARP request packets
arp cor	Traps ARP response packets
arp cor	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
ip _v cor	Traps IPv6 Neighbour Solicitation packets
ip _v cor	Traps IPv6 Neighbour Advertisement packets
ip _v cor	Traps IPv4 BFD packets
ip _v cor	Traps IPv6 BFD packets
ip _v cor	Traps IPv4 OSPF packets

continues on next page

Table 9 – continued from previous page

ip\ cor	Traps IPv6 OSPF packets
ip\ cor	Traps IPv4 BGP packets
ip\ cor	Traps IPv6 BGP packets
ip\ cor	Traps IPv4 VRRP packets
ip\ cor	Traps IPv6 VRRP packets
ip\ cor	Traps IPv4 PIM packets
ip\ cor	Traps IPv6 PIM packets
uc_ cor	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
loc cor	Traps unicast packets that hit a local route and need to be locally delivered
ext cor	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
ip\ cor	Traps unicast 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
ip\ cor	Traps IPv6 packets that their destination IP address is the “All Nodes Address” (i.e., ff02::1)
ip\ cor	Traps IPv6 packets that their destination IP address is the “All Routers Address” (i.e., ff02::2)
ip\ cor	Traps IPv6 Router Solicitation packets
ip\ cor	Traps IPv6 Router Advertisement packets
ip\ cor	Traps IPv6 Redirect Message packets
ip\ cor	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
ip\ cor	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
pt\ cor	Traps PTP time-critical event messages (Sync, Delay_req, Pdelay_Req and Pdelay_Resp)
pt\ cor	Traps PTP general messages (Announce, Follow_Up, Delay_Resp, Pdelay_Resp_Follow_Up, management and signaling)
fl\ cor	Traps packets sampled during processing of flow action sample (e.g., via tc’s sample action)
fl\ cor	Traps packets logged during processing of flow action trap (e.g., via tc’s trap action)
ea\ dro	Traps packets dropped due to the RED (Random Early Detection) algorithm (i.e., early drops)
vx\ dro	Traps 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.
ll\ dro	Traps packets dropped due to an error in the LLC+SNAP header parsing
vl\ dro	Traps packets dropped due to an error in the VLAN header parsing. Could include unexpected packet truncation.

continues on next page

Table 9 – continued from previous page

ppp_drc	Traps 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
mpl_drc	Traps packets dropped due to an error in the MPLS header parsing which could include unexpected header truncation
arp_drc	Traps packets dropped due to an error in the ARP header parsing
ip_1_drc	Traps 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
ip_2_drc	Traps 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
gre_drc	Traps packets dropped due to an error in the GRE header parsing
udp_drc	Traps 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.
tcp_drc	Traps 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.
ipsec_drc	Traps packets dropped due to an error in the IPSEC header parsing
sctp_drc	Traps 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.
dccp_drc	Traps packets dropped due to an error in the DCCP header parsing
gtp_drc	Traps packets dropped due to an error in the GTP header parsing
esp_drc	Traps packets dropped due to an error in the ESP header parsing

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](#)

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 10: List of Generic Packet Trap Groups

Name	Description
l2_dro	Contains packet traps for packets that were dropped by the device during layer 2 forwarding (i.e., bridge)
l3_dro	Contains packet traps for packets that were dropped by the device during layer 3 forwarding
l3_exc	Contains packet traps for packets that hit an exception (e.g., TTL error) during layer 3 forwarding
buffer	Contains packet traps for packets that were dropped by the device due to an enqueue decision
tunnel	Contains packet traps for packets that were dropped by the device during tunnel encapsulation / decapsulation
acl_dr	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_sno	Contains packet traps for IGMP and MLD packets required for multicast snooping
dhcp	Contains packet traps for DHCP packets
neigh_	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_loo	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_	Contains packet traps for packets that should be locally delivered after routing, but do not match more specific packet traps (e.g., ipv4_bgp)
extern	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_ev	Contains packet traps for PTP time-critical event messages (Sync, Delay_req, Pdelay_Req and Pdelay_Resp)
ptp_ge	Contains packet traps for PTP general messages (Announce, Follow_Up, Delay_Resp, Pdelay_Resp_Follow_Up, management and signaling)
acl_sa	Contains packet traps for packets that were sampled by the device during ACL processing
acl_tr	Contains packet traps for packets that were trapped (logged) by the device during ACL processing
parser	Contains packet traps for packets that were marked by the device during parsing as erroneous

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.

9.2 Driver-specific documentation

Each driver that implements `devlink` is expected to document what parameters, info versions, and other features it supports.

9.2.1 bnxt devlink support

This document describes the `devlink` features implemented by the `bnxt` device driver.

Parameters

Table 11: Generic parameters implemented

Name	Mode
enable_sriov	Permanent
ignore_ari	Permanent
msix_vec_per_pf_max	Permanent
msix_vec_per_pf_min	Permanent

The `bnxt` driver also implements the following driver-specific parameters.

Table 12: Driver-specific parameters implemented

Name	Type	Mode	Description
gre博	Boolean	Permanent	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 13: 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

9.2.2 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 14: devlink info versions implemented

Name	Type	Description
fw	running	Version of firmware running on the device
asic	fixed	The ASIC type for this device
id		
asic	fixed	The revision of the ASIC for this device
rev		

9.2.3 ice devlink support

This document describes the devlink features implemented by the ice device driver.

Info versions

The ice driver reports the following versions

Table 15: devlink info versions implemented

Na	Ty	Ex	Description
		am ple	
bo: fix id	00	K6	The Product Board Assembly (PBA) identifier of the board.
fw mgr nir	2.1		3-digit version number of the management firmware that controls the PHY, link, etc.
fw mgr nir ap:	1.5		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 mgr nir bu:	0x		Unique identifier of the source for the management firmware.
fw unc nir	1.2		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 ps: nir ap:	0.8		Version defining the format of the flash contents.
fw bu: nir	0x		Unique identifier of the firmware image file that was loaded onto the device. Also referred to as the EETRACK identifier of the NVM.
fw ap: nir nar	IC		The name of the DDP package that is active in the device. The DDP OS package is loaded by the driver during initialization. Each variation of the DDP package has a unique name.
		fa Pa ag	
fw ap: nir	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 ap: nir bu:	0x		Unique 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 ne: nir	1.1 6.7		The 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 ne: nir bu:	0x		The first 4 bytes of the hash of the netlist module contents.

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 16: List of supported overwrite modes

Bit	Behavior
DE\	Do not preserve settings stored in the flash components being updated. This includes overwriting the port configuration that determines the number of physical functions the device will initialize with.
DE\	Do not preserve either settings or identifiers. Overwrite everything in the flash with the contents from the provided image, without performing any preservation. This includes overwriting device identifying fields 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_IDENTIFIERS 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.

Regions

The ice driver implements the following regions for accessing internal device data.

Table 17: regions implemented

Name	Description
nvm-flash	The contents of the entire flash chip, sometimes referred to as the device's Non Volatile Memory.
device-cap	The contents of the device firmware's capabilities buffer. Useful to determine the current state and configuration of the device.

Users can request an immediate capture of a snapshot via the DEVLINK_CMD_REGION_NEW

```
$ 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 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
```

(continues on next page)

(continued from previous page)

```

$ 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
0000000000000010 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
0000000000000030 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
0000000000000050 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
0000000000000070 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
0000000000000090 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
00000000000000b0 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
00000000000000d0 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
00000000000000f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000000000000100 14 00 01 00 01 00 00 00 00 00 00 00 00 00 00 00
0000000000000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000000000000120 15 00 01 00 01 00 00 00 00 00 00 00 00 00 00 00
0000000000000130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000000000000140 16 00 01 00 01 00 00 00 00 00 00 00 00 00 00 00
0000000000000150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000000000000160 17 00 01 00 06 00 00 00 00 00 00 00 00 00 00 00
0000000000000170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000000000000180 18 00 01 00 01 00 00 00 01 00 00 00 08 00 00 00
0000000000000190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000000001a0 22 00 01 00 01 00 00 00 00 00 00 00 00 00 00 00
00000000000001b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000000001c0 40 00 01 00 00 08 00 00 08 00 00 00 00 00 00 00
00000000000001d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000000001e0 41 00 01 00 00 08 00 00 00 00 00 00 00 00 00 00
00000000000001f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000000000000200 42 00 01 00 00 08 00 00 00 00 00 00 00 00 00 00
0000000000000210 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

```

9.2.4 mlx4 devlink support

This document describes the devlink features implemented by the mlx4 device driver.

Parameters

Table 18: Generic parameters implemented

Name	Mode
internal_err_reset	driverinit, runtime
max_macs	driverinit
region_snapshot_enable	driverinit, runtime

The mlx4 driver also implements the following driver-specific parameters.

Table 19: Driver-specific parameters implemented

Name	Type	Mode	Description
enable_64_byte_cqes_eqes	Boolean	driverinit	Enable 64 byte CQEs/EQEs, if the FW supports it.
enable_4k_uar	Boolean	driverinit	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.

9.2.5 mlx5 devlink support

This document describes the devlink features implemented by the mlx5 device driver.

Parameters

Table 20: Generic parameters implemented

Name	Mode
enable_roce	driverinit

The `mlx5` driver also implements the following driver-specific parameters.

Table 21: Driver-specific parameters implemented

Na	Ty	Mo	Description
flc	str	dur	Controls the flow steering mode of the driver
		tim	<ul style="list-style-type: none">• <code>dmfs</code> Device managed flow steering. In DMFS mode, the HW steering entities are created and managed through firmware.• <code>smfs</code> Software managed flow steering. In SMFS mode, the HW steering entities are created and manage through the driver without firmware intervention.
fdt	u32	dri	Control the number of large groups (size > 1) in the FDB table.
			<ul style="list-style-type: none">• The default value is 15, and the range is between 1 and 1024.

The `mlx5` driver supports reloading via `DEVLINK_CMD_RELOAD`

Info versions

The `mlx5` driver reports the following versions

Table 22: devlink info versions implemented

Na	Ty	Description
fw. fix	psi	Used to represent the board id of the device.
fw. sto	ver	Three digit major.minor.subminor firmware version number.
	dur	
	nin	

9.2.6 mlxsw devlink support

This document describes the devlink features implemented by the `mlxsw` device driver.

Parameters

Table 23: Generic parameters implemented

Name	Mode
fw_load_policy	driverinit

The mlxsw driver also implements the following driver-specific parameters.

Table 24: Driver-specific parameters implemented

Na	Ty	Mo	Description
acl_u32	ur	ur	Sets an interval for periodic ACL region rehashes. The value is specified in 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 25: devlink info versions implemented

Na	Ty	Description
hw_fix_rev	fix	The hardware revision for this board
fw_fix_psi	fix	Firmware PSID
fw_ver_nin	ur	Three digit firmware version

Driver-specific Traps

Table 26: List of Driver-specific Traps Registered by mlxsw

Na	Ty	Description
iri_drop	dro	Traps 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
eri_drop	dro	Traps 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

9.2.7 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 27: Driver-specific parameters implemented

Name	Type	Mode	Description
ATU	u8	runtime	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.

9.2.8 netdevsim devlink support

This document describes the devlink features supported by the netdevsim device driver.

Parameters

Table 28: Generic parameters implemented

Name	Mode
max_macs	driverinit

The netdevsim driver also implements the following driver-specific parameters.

Table 29: Driver-specific parameters implemented

Name	Type	Mode	Description
test	Bool	driver	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 and FIB rule entries 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 dev reload netdevsim/netdevsim0
```

Driver-specific Traps

Table 30: List of Driver-specific Traps Registered by netdevsim

Name	Type	Description
filter_exception	filter	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

9.2.9 nfp devlink support

This document describes the devlink features implemented by the nfp device driver.

Parameters

Table 31: 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 32: devlink info versions implemented

Na	Ty	Description
bo	fix	Part number identifying the board design
id		
bo	fix	Revision of the board design
rev		
bo	fix	Vendor of the board design
mar		
bo	fix	Model name of the board design
mod		
fw	sto	Firmware bundle id
bur	zur	
nin		
fw	sto	Version of the management firmware
mgn	zur	
nin		
fw	sto	The CPLD firmware component version
cpl	zur	
nin		
fw	sto	The APP firmware component version
ap	zur	
nin		
fw	sto	The UNDI firmware component version
unc	zur	
nin		
fw	sto	The NSCI firmware component version
nc	zur	
nin		
ch	sto	The CFGR firmware component version
inj	zur	
nin		

9.2.10 sja1105 devlink support

This document describes the devlink features implemented by the sja1105 device driver.

Parameters

Table 33: Driver-specific parameters implemented

Na	Ty	Mo	Description
be	Bo	run	Allow plain ETH_P_8021Q headers to be used as DSA tags.
	tim		Benefits: <ul style="list-style-type: none"> • Can terminate untagged traffic over switch net devices even when enslaved to a bridge with <code>vlan_filtering=1</code>. • Can terminate VLAN-tagged traffic over switch net devices even when enslaved to a bridge with <code>vlan_filtering=1</code>, with some constraints (no more than 7 non-pvid VLANs per user port). • Can do QoS based on VLAN PCP and VLAN membership admission control for autonomously forwarded frames (regardless of whether they can be terminated on the CPU or not). Drawbacks: <ul style="list-style-type: none"> • User cannot use VLANs in range 1024-3071. If the switch receives frames with such VIDs, it will misinterpret them as DSA tags. • Switch uses Shared VLAN Learning (FDB lookup uses only DMAC as key). • When VLANs span cross-chip topologies, the total number of permitted VLANs may be less than 7 per port, due to a maximum number of 32 VLAN retagging rules per switch.

9.2.11 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 34: Driver-specific parameters implemented

Na	Ty	Mo	Description
iw	Bo	run	Enable iWARP functionality for 100g devices. Note that this impacts L2 performance, and is therefore not enabled by default.

9.2.12 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 35: Driver-specific parameters implemented

Name	Type	Mode	Description
ale_bypass	Boolean	runtime	Enables ALE_CONTROL(4).BYPASS mode for debugging purposes. In this mode, all packets will be sent to the host port only.
switch_mode	Boolean	runtime	Enable switch mode

Contents:

10.1 Linux CAIF

Copyright © ST-Ericsson AB 2010

Author

Sjur Brendeland/ sjur.brandeland@stericsson.com

License terms

GNU General Public License (GPL) version 2

10.1.1 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.

10.1.2 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
!           +-----+   +-----!
```

(continues on next page)

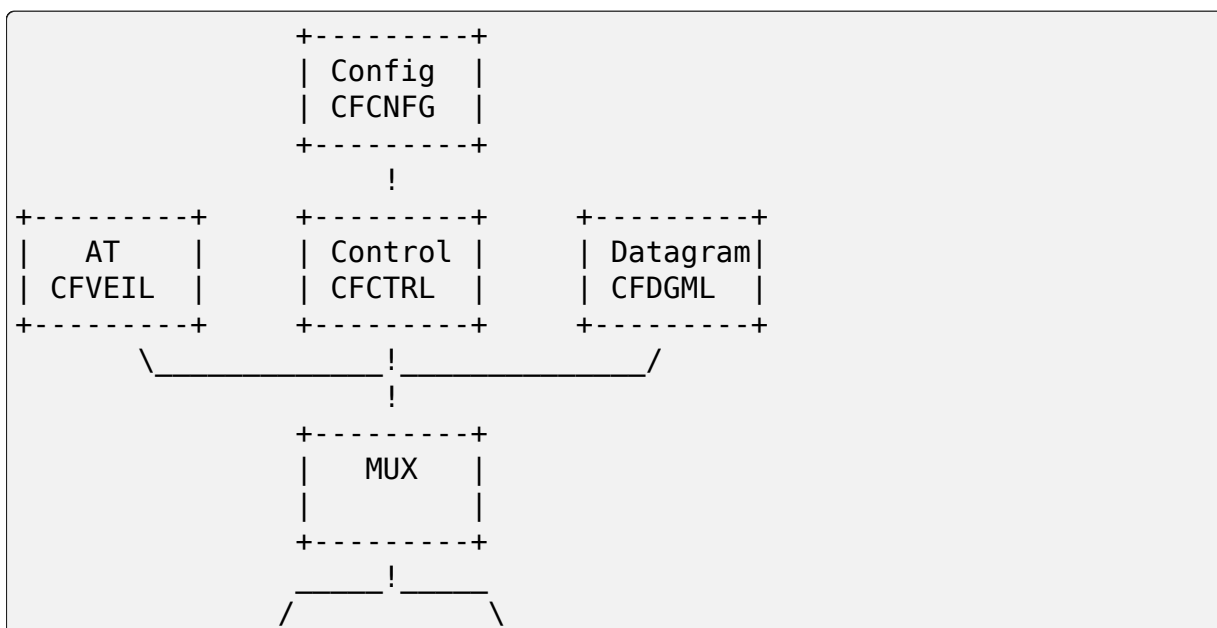
10.1.4 Layered Architecture

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.

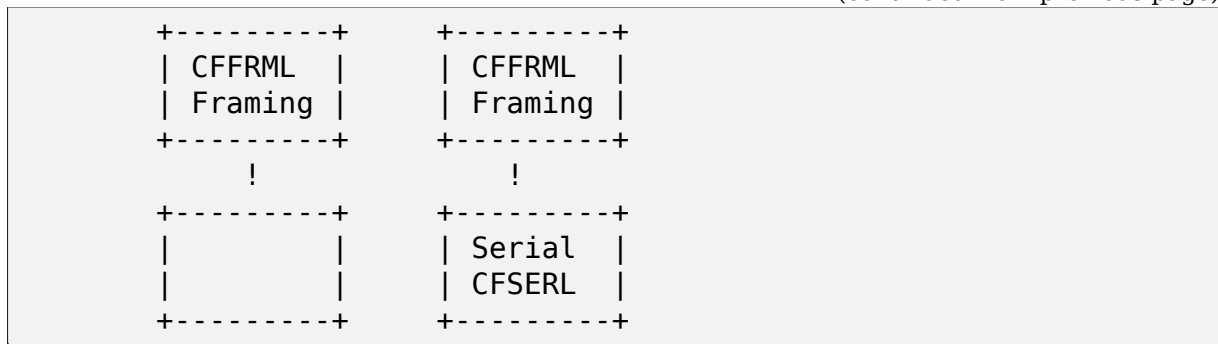
The CAIF Protocol implementation contains:

- 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.



(continues on next page)

(continued from previous page)



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);
```

10.1.5 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.

10.2 Using Linux CAIF

Copyright

© ST-Ericsson AB 2010

Author

Sjur Brendeland/ sjur.brandeland@stericsson.com

10.2.1 Start

If you have compiled CAIF for modules do:

```
$modprobe crc_ccitt
$modprobe caif
$modprobe caif_socket
$modprobe chnl_net
```

10.2.2 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!

10.2.3 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.
 - 0x02 - `tty->low_latency` is on.
 - 0x04 - `tty->packed` is on.
 - 0x08 - `tty->flow_stopped` is on.
 - 0x10 - `tty->hw_stopped` is on.
 - 0x20 - `tty->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_tx_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:

```
0000000 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:

```
0000000 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)
```

(continues on next page)

(continued from previous page)

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.

10.2.4 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

11.1 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.

11.2 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.

11.3 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 (<code>_SET</code> and <code>_ACT</code>)
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.

11.4 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:

ETHTOOL_A_BITSET_NOMASK	flag	no mask, only a list
ETHTOOL_A_BITSET_SIZE	u32	number of significant bits
ETHTOOL_A_BITSET_VALUE	binary	bitmap of bit values
ETHTOOL_A_BITSET_MASK	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:

ETHTOOL_A_BITSET_NOMASK	flag	no mask, only a list
ETHTOOL_A_BITSET_SIZE	u32	number of significant bits
ETHTOOL_A_BITSET_BITS	nested	array of bits
ETHTOOL_A_BITSET_BITS_BIT+	nested	one bit
ETHTOOL_A_BITSET_BIT_INDEX	u32	bit index (0 for LSB)
ETHTOOL_A_BITSET_BIT_NAME	string	bit name
ETHTOOL_A_BITSET_BIT_VALUE	flag	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 `ETHHTOOL_FLAG_COMPACT_BITSETS` flag in flags field of request header. Semantics of value and mask depends on the attribute.

11.5 List of message types

All constants identifying message types use `ETHHTOOL_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>ETHHTOOL_MSG_STRSET_GET</code>	get string set
<code>ETHHTOOL_MSG_LINKINFO_GET</code>	get link settings
<code>ETHHTOOL_MSG_LINKINFO_SET</code>	set link settings
<code>ETHHTOOL_MSG_LINKMODES_GET</code>	get link modes info
<code>ETHHTOOL_MSG_LINKMODES_SET</code>	set link modes info
<code>ETHHTOOL_MSG_LINKSTATE_GET</code>	get link state
<code>ETHHTOOL_MSG_DEBUG_GET</code>	get debugging settings
<code>ETHHTOOL_MSG_DEBUG_SET</code>	set debugging settings
<code>ETHHTOOL_MSG_WOL_GET</code>	get wake-on-lan settings
<code>ETHHTOOL_MSG_WOL_SET</code>	set wake-on-lan settings
<code>ETHHTOOL_MSG_FEATURES_GET</code>	get device features
<code>ETHHTOOL_MSG_FEATURES_SET</code>	set device features
<code>ETHHTOOL_MSG_PRIVFLAGS_GET</code>	get private flags
<code>ETHHTOOL_MSG_PRIVFLAGS_SET</code>	set private flags
<code>ETHHTOOL_MSG_RINGS_GET</code>	get ring sizes
<code>ETHHTOOL_MSG_RINGS_SET</code>	set ring sizes
<code>ETHHTOOL_MSG_CHANNELS_GET</code>	get channel counts
<code>ETHHTOOL_MSG_CHANNELS_SET</code>	set channel counts
<code>ETHHTOOL_MSG_COALESCE_GET</code>	get coalescing parameters
<code>ETHHTOOL_MSG_COALESCE_SET</code>	set coalescing parameters
<code>ETHHTOOL_MSG_PAUSE_GET</code>	get pause parameters
<code>ETHHTOOL_MSG_PAUSE_SET</code>	set pause parameters
<code>ETHHTOOL_MSG_EEE_GET</code>	get EEE settings
<code>ETHHTOOL_MSG_EEE_SET</code>	set EEE settings
<code>ETHHTOOL_MSG_TSINFO_GET</code>	get timestamping info
<code>ETHHTOOL_MSG_CABLE_TEST_ACT</code>	action start cable test
<code>ETHHTOOL_MSG_CABLE_TEST_TDR_ACT</code>	action start raw TDR cable test
<code>ETHHTOOL_MSG_TUNNEL_INFO_GET</code>	get tunnel offload info

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

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.

11.6 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:

<code>ETHTOOL_A_STRSET_HEADER</code>	nestec	request header
<code>ETHTOOL_A_STRSET_STRINGSETS</code>	nestec	string set to request
<code>ETHTOOL_A_STRINGSETS_STRINGSET+</code>	nestec	one string set
<code>ETHTOOL_A_STRINGS</code>	u32	set id

Kernel response contents:

ETHHTOOL_A_STRSET_HEADER	nested	reply header
ETHHTOOL_A_STRSET_STRINGSETS	nested	array of string sets
ETHHTOOL_A_STRINGSETS_STRINGSET+	nested	one string set
ETHHTOOL_A_STRINGSET_ID	u32	set id
ETHHTOOL_A_STRINGSET_COUNT	u32	number of strings
ETHHTOOL_A_STRINGSET_STRINGS	nested	array of strings
ETHHTOOL_A_STRINGS_STRING	nested	one string
ETHHTOOL_A_STRING_INDEX	u32	string index
ETHHTOOL_A_STRING_VALUE	string	string value
ETHHTOOL_A_STRSET_COUNTS_ONLY	flag	return only counts

Device identification in request header is optional. Depending on its presence 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 ETHHTOOL_A_STRSET_STRINGSETS array, all string sets of requested type are returned, otherwise only those specified in the request. Flag ETHHTOOL_A_STRSET_COUNTS_ONLY tells kernel to only return string counts of the sets, not the actual strings.

11.7 LINKINFO_GET

Requests link settings as provided by ETHHTOOL_GLINKSETTINGS except for link modes and autonegotiation related information. The request does not use any attributes.

Request contents:

ETHHTOOL_A_LINKINFO_HEADER	nested	request header
----------------------------	--------	----------------

Kernel response contents:

ETHHTOOL_A_LINKINFO_HEADER	nested	reply header
ETHHTOOL_A_LINKINFO_PORT	u8	physical port
ETHHTOOL_A_LINKINFO_PHYADDR	u8	phy MDIO address
ETHHTOOL_A_LINKINFO_TP_MDIX	u8	MDI(-X) status
ETHHTOOL_A_LINKINFO_TP_MDIX_CTRL	u8	MDI(-X) control
ETHHTOOL_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).

11.8 LINKINFO_SET

`LINKINFO_SET` request allows setting some of the attributes reported by `LINKINFO_GET`.

Request contents:

ETHHTOOL_A_LINKINFO_HEADER	nested	request header
ETHHTOOL_A_LINKINFO_PORT	u8	physical port
ETHHTOOL_A_LINKINFO_PHYADDR	u8	phy MDIO address
ETHHTOOL_A_LINKINFO_TP_MDIX_CTRL	u8	MDI(-X) control

MDI(-X) status and transceiver cannot be set, request with the corresponding attributes is rejected.

11.9 LINKMODES_GET

Requests link modes (supported, advertised and peer advertised) and related information (autonegotiation status, link speed and duplex) as provided by `ETHHTOOL_GLINKSETTINGS`. The request does not use any attributes.

Request contents:

ETHHTOOL_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_C	u8	Master/slave port mode	
ETHTOOL_A_LINKMODES_MASTER_SLAVE_S	u8	Master/slave port state	

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).

11.10 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_	u8	Master/slave port mode	

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 and duplex is specified, kernel adjusts advertised modes to all supported modes matching speed, duplex or both (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.

11.11 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:

<code>ETHTOOL_A_LINKSTATE_HEADER</code>	nested	request header
---	--------	----------------

Kernel response contents:

<code>ETHTOOL_A_LINKSTATE_HEADER</code>	nested	reply header
<code>ETHTOOL_A_LINKSTATE_LINK</code>	bool	link state (up/down)
<code>ETHTOOL_A_LINKSTATE_SQI</code>	u32	Current Signal Quality Index
<code>ETHTOOL_A_LINKSTATE_SQI_MAX</code>	u32	Max support SQI value
<code>ETHTOOL_A_LINKSTATE_EXT_STATE</code>	u8	link extended state
<code>ETHTOOL_A_LINKSTATE_EXT_SUBSTATE</code>	u8	link extended substate

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_</code>	States relating to the autonegotiation or issues therein
<code>ETHTOOL_LINK_EXT_STATE_</code>	Failure during link training
<code>ETHTOOL_LINK_EXT_STATE_</code>	Logical mismatch in physical coding sublayer or forward error correction sublayer
<code>ETHTOOL_LINK_EXT_STATE_</code>	Signal integrity issues
<code>ETHTOOL_LINK_EXT_STATE_</code>	No cable connected
<code>ETHTOOL_LINK_EXT_STATE_</code>	Failure is related to cable, e.g., unsupported cable
<code>ETHTOOL_LINK_EXT_STATE_</code>	Failure is related to EEPROM, e.g., failure during reading or parsing the data
<code>ETHTOOL_LINK_EXT_STATE_</code>	Failure during calibration algorithm
<code>ETHTOOL_LINK_EXT_STATE_</code>	The hardware is not able to provide the power required from cable or module
<code>ETHTOOL_LINK_EXT_STATE_</code>	The module is overheated

Link extended substates:

Autoneg substates:

ETHTOOL_LINK_EXT_SUBSTATE_AN_	Peer side is down
ETHTOOL_LINK_EXT_SUBSTATE_AN_	Ack not received from peer side
ETHTOOL_LINK_EXT_SUBSTATE_AN_	Next page exchange failed
ETHTOOL_LINK_EXT_SUBSTATE_AN_	Peer side is down during force mode or there is no agreement of speed
ETHTOOL_LINK_EXT_SUBSTATE_AN_	Forward error correction modes in both sides are mismatched
ETHTOOL_LINK_EXT_SUBSTATE_AN_	No Highest Common Denominator

Link training substates:

ETHTOOL_LINK_EXT_SUBSTATE_LT_KR_FR	Frames were not recognized, the lock failed
ETHTOOL_LINK_EXT_SUBSTATE_LT_KR_LIN	The lock did not occur before timeout
ETHTOOL_LINK_EXT_SUBSTATE_LT_KR_LIN	Peer side did not send RECEIVER_READY ready signal after training process
ETHTOOL_LINK_EXT_SUBSTATE_LT_REMOTE	Remote side is not ready yet

Link logical mismatch substates:

ETHTOOL_LINK_EXT_SUBSTATE_L	Physical coding sublayer was not locked in first phase - block lock
ETHTOOL_LINK_EXT_SUBSTATE_L	Physical coding sublayer was not locked in second phase - alignment markers lock
ETHTOOL_LINK_EXT_SUBSTATE_L	Physical coding sublayer did not get align status
ETHTOOL_LINK_EXT_SUBSTATE_L	FC forward error correction is not locked
ETHTOOL_LINK_EXT_SUBSTATE_L	RS forward error correction is not locked

Bad signal integrity substates:

ETHTOOL_LINK_EXT_SUBSTATE_L	Large number of physical errors
ETHTOOL_LINK_EXT_SUBSTATE_L	The system attempted to operate the cable at a rate that is not formally supported, which led to signal integrity issues

Cable issue substates:

ETHTOOL_LINK_EXT_SUBSTATE_CI_UNSUPPORTED_CA	Unsupported ca-
	ble
ETHTOOL_LINK_EXT_SUBSTATE_CI_CABLE_TEST_FAI	Cable test failure

11.12 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_GMSGLVL 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).

11.13 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.

11.14 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:

<code>ETHTOOL_A_WOL_HEADER</code>	nested	request header
-----------------------------------	--------	----------------

Kernel response contents:

<code>ETHTOOL_A_WOL_HEADER</code>	nested	reply header
<code>ETHTOOL_A_WOL_MODES</code>	bitset	mask of enabled WoL modes
<code>ETHTOOL_A_WOL_SOPASS</code>	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.

11.15 WOL_SET

Set or update wake-on-lan settings.

Request contents:

<code>ETHTOOL_A_WOL_HEADER</code>	nested	request header
<code>ETHTOOL_A_WOL_MODES</code>	bitset	enabled WoL modes
<code>ETHTOOL_A_WOL_SOPASS</code>	binary	SecureOn(tm) password

`ETHTOOL_A_WOL_SOPASS` is only allowed for devices supporting `WAKE_MAGICSECURE` mode.

11.16 FEATURES_GET

Gets netdev features like `ETHTOOL_GFEATURES` ioctl request.

Request contents:

<code>ETHTOOL_A_FEATURES_HEADER</code>	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. ETHA_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).

11.17 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()*.

11.18 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.

11.19 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.

11.20 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

11.21 RINGS_SET

Sets ring sizes like ETHTOOL_SRINGPARAM ioctl request.

Request contents:

ETHTOOL_A_RINGS_HEADER	nested	reply header
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

Kernel checks that requested ring sizes do not exceed limits reported by driver. Driver may impose additional constraints and may not support all attributes.

11.22 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

11.23 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.

11.24 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_I	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_I	u32	IRQ packets, Tx in IRQ
ETHTOOL_A_COALESCE_STATS_BLOCK_USE	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_L	u32	max packets, low Rx
ETHTOOL_A_COALESCE_TX_USECS_LOW	u32	delay (us), low Tx
ETHTOOL_A_COALESCE_TX_MAX_FRAMES_L	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_H	u32	max packets, high Rx
ETHTOOL_A_COALESCE_TX_USECS_HIGH	u32	delay (us), high Tx
ETHTOOL_A_COALESCE_TX_MAX_FRAMES_H	u32	max packets, high Tx
ETHTOOL_A_COALESCE_RATE_SAMPLE_INT	u32	rate sampling interval

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).

11.25 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_I	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_I	u32	IRQ packets, Tx in IRQ
ETHTOOL_A_COALESCE_STATS_BLOCK_USE	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_L	u32	max packets, low Rx
ETHTOOL_A_COALESCE_TX_USECS_LOW	u32	delay (us), low Tx
ETHTOOL_A_COALESCE_TX_MAX_FRAMES_L	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_H	u32	max packets, high Rx
ETHTOOL_A_COALESCE_TX_USECS_HIGH	u32	delay (us), high Tx
ETHTOOL_A_COALESCE_TX_MAX_FRAMES_H	u32	max packets, high Tx
ETHTOOL_A_COALESCE_RATE_SAMPLE_INT	u32	rate sampling interval

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.

11.26 PAUSE_GET

Gets channel counts like ETHTOOL_GPAUSE ioctl request.

Request contents:

ETHTOOL_A_PAUSE_HEADER	nested	request header
------------------------	--------	----------------

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 {  
    u64 tx_pause_frames;  
    u64 rx_pause_frames;  
};
```

Members

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.

11.27 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

11.28 EEE_GET

Gets channel counts 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.

11.29 EEE_SET

Sets pause parameters like ETHTOOL_GEEEPARAM ioctl request.

Request contents:

ETHTOOL_A_EEE_HEADER	nested	request header
ETHTOOL_A_EEE_MODES_OURS	bool	advertised modes
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)

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.

11.30 TSINFO_GET

Gets timestamping information like ETHTOOL_GET_TS_INFO ioctl request.

Request contents:

ETHTOOL_A_TSINFO_HEADER	nested	request header
-------------------------	--------	----------------

Kernel response contents:

ETHTOOL_A_TSINFO_HEADER	nested	request header
ETHTOOL_A_TSINFO_TIMESTAMPING	bitset	SO_TIMESTAMPING flags
ETHTOOL_A_TSINFO_TX_TYPES	bitset	supported Tx types
ETHTOOL_A_TSINFO_RX_FILTERS	bitset	supported Rx filters
ETHTOOL_A_TSINFO_PHC_INDEX	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).

11.31 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.

ETH00L_A_CABLE_TEST_HEADER	neste	reply header
ETH00L_A_CABLE_TEST_STATUS	u8	completed
ETH00L_A_CABLE_TEST_NTF_NEST	neste	all the results
ETH00L_A_CABLE_NEST_RESULT	neste	cable test result
ETH00L_A_CABLE_RESULT!	u8	pair number
ETH00L_A_CABLE_RESULT!	u8	result code
ETH00L_A_CABLE_NEST_RESULT	neste	cable test results
ETH00L_A_CABLE_RESULT!	u8	pair number
ETH00L_A_CABLE_RESULT!	u8	result code
ETH00L_A_CABLE_NEST_FAULT_LENGTH	neste	cable length
ETH00L_A_CABLE_FAULT_	u8	pair number
ETH00L_A_CABLE_FAULT_	u32	length in cm

11.32 CABLE_TEST TDR

Start a cable test and report raw TDR data

Request contents:

ETH00L_A_CABLE_TEST_TDR_HEADER	neste	reply header
ETH00L_A_CABLE_TEST_TDR_CFG	neste	test configuration
ETH00L_A_CABLE_STEP_FIRST_DI	u32	first data distance
ETH00L_A_CABLE_STEP_LAST_DIS	u32	last data distance
ETH00L_A_CABLE_STEP_STEP_DIS	u32	distance of each step
ETH00L_A_CABLE_TEST_TDR_CFG_	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 re- sults
ETHTOOL_A_CABLE_TDR_NEST_PULSE	nested	TX Pulse amplitude
ETHTOOL_A_CABLE_PULSE_	s16	Pulse ampli- tude
ETHTOOL_A_CABLE_NEST_STEP	nested	TDR step info
ETHTOOL_A_CABLE_STEP_F	u32	First data distance
ETHTOOL_A_CABLE_STEP_L	u32	Last data distance
ETHTOOL_A_CABLE_STEP_S	u32	distance of each step
ETHTOOL_A_CABLE_TDR_NEST_AMPLITUDE	nested	Reflection amplitude
ETHTOOL_A_CABLE_RESULT	u8	pair number
ETHTOOL_A_CABLE_AMPLIT	s16	Reflection amplitude
ETHTOOL_A_CABLE_TDR_NEST_AMPLITUDE	nested	Reflection amplitude
ETHTOOL_A_CABLE_RESULT	u8	pair number
ETHTOOL_A_CABLE_AMPLIT	s16	Reflection amplitude
ETHTOOL_A_CABLE_TDR_NEST_AMPLITUDE	nested	Reflection amplitude
ETHTOOL_A_CABLE_RESULT	u8	pair number
ETHTOOL_A_CABLE_AMPLIT	s16	Reflection amplitude

11.33 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	nest	reply header
ETHTOOL_A_TUNNEL_INFO_UDP_PORTS	nest	all UDP port tables
ETHTOOL_A_TUNNEL_UDP_TABLE	nest	one UDP port table
ETHTOOL_A_TUNNEL_UDP_TABLE_	u32	max size of the table
ETHTOOL_A_TUNNEL_UDP_TABLE_	bit-set	tunnel types which table can hold
ETHTOOL_A_TUNNEL_UDP_TABLE_	nest	offloaded UDP port
ETHTOOL_A_TUNNEL_	be16	UDP port
ETHTOOL_A_TUNNEL_	u32	tunnel type

For UDP tunnel table empty `ETHTOOL_A_TUNNEL_UDP_TABLE_TYPES` indicates that the table contains static entries, hard-coded by the NIC.

11.34 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
ETHTOOL_GRXCSUM	ETHTOOL_MSG_FEATURES_GET
ETHTOOL_SRXCSUM	ETHTOOL_MSG_FEATURES_SET
ETHTOOL_GTXCSUM	ETHTOOL_MSG_FEATURES_GET

continues on next page

Table 1 – continued from previous page

ioctl command	netlink command
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_GRXCLSRCNT	n/a
ETHTOOL_GRXCLSRULE	n/a
ETHTOOL_GRXCLRLALL	n/a
ETHTOOL_SRXCLRLDEL	n/a
ETHTOOL_SRXCLRLINS	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	n/a
ETHTOOL_GMODULEEEPROM	n/a
ETHTOOL_GEEE	ETHTOOL_MSG_EEE_GET
ETHTOOL_SEEE	ETHTOOL_MSG_EEE_SET
ETHTOOL_GRSSH	n/a

continues on next page

Table 1 – continued from previous page

ioctl command	netlink command
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	n/a
ETHTOOL_SFECPARAM	n/a
n/a	° ETHTOOL_MSG_CABLE_TEST_ACT''
n/a	° ETHTOOL_MSG_CABLE_TEST_TDR_ACT''
n/a	ETHTOOL_MSG_TUNNEL_INFO_GET

IEEE 802.15.4 DEVELOPER' S GUIDE

12.1 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

12.2 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>).

12.3 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

12.4 Drivers

Like with WiFi, there are several types of devices implementing IEEE 802.15.4. 1) 'HardMAC'. 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, automatic ACK handling, address matching, etc.

Those types of devices require different approach to be hooked into Linux kernel.

12.4.1 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.

12.4.2 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/`.

12.4.3 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.

12.5 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

13.1 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 $((2^{24} - 1) * 7 \text{ bytes}) = 111 \text{ MiB}$.

13.1.1 Specifications used

- SAE J1939-21 : data link layer
- SAE J1939-81 : network management
- ISO 11783-6 : Virtual Terminal (Extended Transport Protocol)

13.2 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.

13.3 J1939 concepts

13.3.1 PGN

The PGN (Parameter Group Number) is a number to identify a packet. The PGN is composed as follows: 1 bit : Reserved Bit 1 bit : Data Page 8 bits : PF (PDU Format) 8 bits : 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.

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.

13.3.2 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.

13.3.3 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.

13.4 How to Use J1939

13.4.1 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*()` 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 {
```

(continues on next page)

(continued from previous page)

```
    __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 (`!= 0`) the NAME is looked up by the kernel and the corresponding ADDR is used. If `can_addr.j1939.name` is not set (`== 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)`.

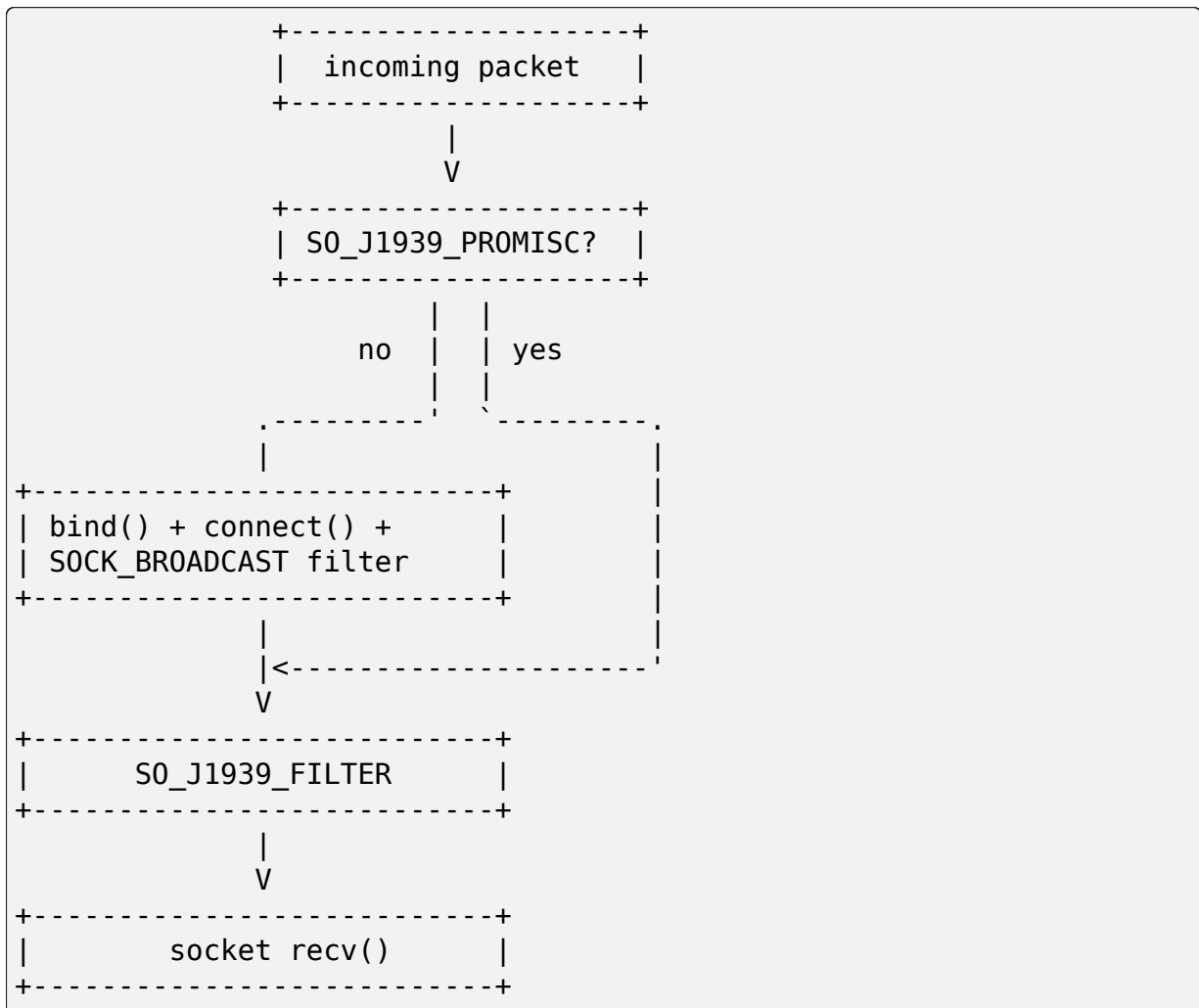
RX path related options:

- `S0_J1939_FILTER` - configure array of filters
- `S0_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 `S0_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_PRI0 - 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 `msg_hdr` 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, &msg)) {
    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;
    }
}
```

13.4.2 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"),
}
```

(continues on next page)

(continued from previous page)

```

};

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_COMMAND,
        .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 (Little_
↳ Endian). 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.

13.4.3 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

14.1 Linux Networking

14.1.1 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

14.1.2 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)
Decrements 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 {
    ktime_t hwtstamp;
};
```

Members

hwtstamp

hardware time stamp transformed into duration since arbitrary point in time

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          rbnode;
        struct list_head        list;
    };
    union {
        struct sock             *sk;
        int ip_defrag_offset;
    };
    union {
        ktime_t tstamp;
    };
};
```

(continues on next page)

(continued from previous page)

```

    u64 skb_mstamp_ns;
};
char cb[48] ;
union {
    struct {
        unsigned long    _skb_refdst;
        void (*destructor)(struct sk_buff *skb);
    };
    struct list_head      tcp_tsorted_anchor;
};
#ifdef 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;
#ifdef __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;
#ifdef CONFIG_SKB_EXTENSIONS;
    __u8 active_extensions;
#endif;
#ifdef __BIG_ENDIAN_BITFIELD;
#define PKT_TYPE_MAX      (7 << 5);
#else;
#define PKT_TYPE_MAX      7;
#endif;
#define PKT_TYPE_OFFSET()    offsetof(struct sk_buff, __pkt_type_
↳offset);
    __u8 pkt_type:3;
    __u8 ignore_df:1;
    __u8 nf_trace:1;
    __u8 ip_summed:2;
    __u8 ooo_okay:1;
    __u8 l4_hash:1;
    __u8 sw_hash:1;
    __u8 wifi_acked_valid:1;
    __u8 wifi_acked:1;
    __u8 no_fcs:1;
    __u8 encapsulation:1;
    __u8 encap_hdr_csum:1;
    __u8 csum_valid:1;
#ifdef __BIG_ENDIAN_BITFIELD;

```

(continues on next page)

(continued from previous page)

```

#define PKT_VLAN_PRESENT_BIT    7;
#else;
#define PKT_VLAN_PRESENT_BIT    0;
#endif;
#define PKT_VLAN_PRESENT_OFFSET()      offsetof(struct sk_buff, __
    ↪ pkt_vlan_present_offset);
    __u8 vlan_present:1;
    __u8 csum_complete_sw:1;
    __u8 csum_level:2;
    __u8 csum_not_inet:1;
    __u8 dst_pending_confirm:1;
#ifdef CONFIG_IPV6_NDISC_NODETYPE;
    __u8 ndisc_nodetype:2;
#endif;
    __u8 ipvs_property:1;
    __u8 inner_protocol_type:1;
    __u8 remcsum_offload:1;
#ifdef CONFIG_NET_SWITCHDEV;
    __u8 offload_fwd_mark:1;
    __u8 offload_l3_fwd_mark:1;
#endif;
#ifdef CONFIG_NET_CLS_ACT;
    __u8 tc_skip_classify:1;
    __u8 tc_at_ingress:1;
#endif;
#ifdef CONFIG_NET_REDIRECT;
    __u8 redirected:1;
    __u8 from_ingress:1;
#endif;
#ifdef CONFIG_TLS_DEVICE;
    __u8 decrypted:1;
#endif;
    __u8 scm_io_uring:1;
#ifdef CONFIG_NET_SCHED;
    __u16 tc_index;
#endif;
    union {
        __wsum csum;
        struct {
            __u16 csum_start;
            __u16 csum_offset;
        };
    };
    __u32 priority;
    int skb_iif;
    __u32 hash;
    __be16 vlan_proto;
    __u16 vlan_tci;
#ifdef CONFIG_NET_RX_BUSY_POLL || defined(CONFIG_XPS);

```

(continues on next page)

(continued from previous page)

```

union {
    unsigned int    napi_id;
    unsigned int    sender_cpu;
};
#endif;
#ifdef 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;
#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;
#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

{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)

_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

active_extensions
active extensions (skb_ext_id types)

pkt_type
Packet class

ignore_df
allow local fragmentation

nf_trace
netfilter packet trace flag

ip_summed
Driver fed us an IP checksum

ooo_okay
allow the mapping of a socket to a queue to be changed

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

vlan_present

VLAN tag is present

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)

csum_not_inet

use CRC32c to resolve CHECKSUM_PARTIAL

dst_pending_confirm

need to confirm neighbour

ndisc_nodetype

router type (from link layer)

ipvs_property

skbuff is owned by ipvs

inner_protocol_type

whether the inner protocol is ENCAP_TYPE_ETHER or ENCAP_TYPE_IPPROTO

remcsum_offload

remote checksum offload is enabled

offload_fwd_mark

Packet was L2-forwarded in hardware

offload_l3_fwd_mark

Packet was L3-forwarded in hardware

tc_skip_classify

do not classify packet. set by IFB device

tc_at_ingress

used within tc_classify to distinguish in/egress

redirected

packet was redirected by packet classifier

from_ingress

packet was redirected from the ingress path

decrypted

Decrypted SKB

scm_io_uring

SKB holds io_uring registered files

tc_index

Traffic control index

{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

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 == ENCAP_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

true_size

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.

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 this 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)
release reference to header

Parameters

struct sk_buff *skb
buffer to operate on

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 into 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* *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 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* *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 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_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.

```
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**(struct sk_buff_head *list)

empty a list

Parameters

struct sk_buff_head *list

list to empty

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.

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.

void **skb_propagate_pfmemalloc**(struct *page* *page, struct *sk_buff* *skb)
Propagate pfmemalloc if skb is allocated after RX page

Parameters

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)

release a reference on a paged fragment.

Parameters

skb_frag_t *frag

the paged fragment

Description

Releases a reference on the paged fragment **frag**.

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

void __skb_frag_set_page(skb_frag_t *frag, struct *page* *page)
sets the page contained in a paged fragment

Parameters

skb_frag_t *frag
the paged fragment

struct page *page
the page to set

Description

Sets the fragment **frag** to contain **page**.

void skb_frag_set_page(struct *sk_buff* *skb, int f, struct *page* *page)
sets the page contained in a paged fragment of an skb

Parameters

struct sk_buff *skb
the buffer

int f
the fragment offset

struct page *page
the page to set

Description

Sets the **f**'th fragment of **skb** to contain **page**.

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;
```

(continues on next page)

(continued from previous page)

```
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;
#ifdef 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;
#ifdef CONFIG_XPS;
    unsigned short      skc_rx_queue_mapping;
#endif
};

```

(continues on next page)

(continued from previous page)

```
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_KEEPAIVE, SO_OOBLINE 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;
#ifdef CONFIG_XPS;
#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;
```

(continues on next page)

(continued from previous page)

```

#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;
    socket_lock_t sk_lock;
    atomic_t sk_drops;
    int sk_rcvlowat;
    struct sk_buff_head         sk_error_queue;
    struct sk_buff              *sk_rx_skb_cache;
    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;
#ifdef CONFIG_NET_RX_BUSY_POLL;
    unsigned int                sk_ll_usec;
    unsigned int                sk_napi_id;
#endif;
    int sk_rcvbuf;
    int sk_wait_pending;
    struct sk_filter __rcu *sk_filter;
    union {
        struct socket_wq __rcu *sk_wq;
    };
#ifdef CONFIG_XFRM;
    struct xfrm_policy __rcu *sk_policy[2];
#endif;
    struct dst_entry __rcu *sk_rx_dst;
    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              *sk_tx_skb_cache;
    struct sk_buff_head         sk_write_queue;
    __s32 sk_peek_off;

```

(continues on next page)

(continued from previous page)

```
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;
netdev_features_t sk_route_nocaps;
netdev_features_t sk_route_forced_caps;
int sk_gso_type;
unsigned int           sk_gso_max_size;
gfp_t sk_allocation;
__u32 sk_txhash;
u8 sk_padding : 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;
spinlock_t sk_peer_lock;
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;
u16 sk_tsflags;
u8 sk_shutdown;
u32 sk_tskey;
atomic_t sk_zckey;
u8 sk_clockid;
u8 sk_txtime_deadline_mode : 1, sk_txtime_report_errors : 1, sk_
↪ txtime_unused : 6;
struct socket          *sk_socket;
void *sk_user_data;
#ifdef CONFIG_SECURITY;
    void *sk_security;
```

(continues on next page)

(continued from previous page)

```

#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);
#ifdef 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;
#ifdef CONFIG_BPF_SYSCALL;
struct bpf_local_storage __rcu  *sk_bpf_storage;
#endif;
struct rcu_head          sk_rcu;
};

```

Members**__sk_common**

shared layout with inet_timewait_sock

sk_lock

synchronizer

sk_drops

raw/udp drops counter

sk_rcvlowat

SO_RCVLOWAT setting

sk_error_queue

rarely used

sk_rx_skb_cache

cache copy of recently accessed RX skb

sk_receive_queue

incoming packets

sk_backlog

always used with the per-socket spinlock held

sk_forward_alloc

space allocated forward

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_wait_pending
number of threads blocked on this socket

sk_filter
socket filtering instructions

{unnamed_union}
anonymous

sk_wq
sock wait queue and async head

sk_policy
flow policy

sk_rx_dst
receive input route used by early demux

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_tx_skb_cache
cache copy of recently accessed TX skb

sk_write_queue
Packet sending queue

sk_peek_off
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
SO_SNDTIMEO setting

sk_timer
sock cleanup timer

sk_priority
SO_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 (SO_MAX_PACING_RATE)

sk_frag
cached page frag

sk_route_caps
route capabilities (e.g. NETIF_F_TS0)

sk_route_nocaps
forbidden route capabilities (e.g. NETIF_F_GSO_MASK)

sk_route_forced_caps
static, forced route capabilities (set in tcp_init_sock())

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_padding
unused element for alignment

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_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_tsflags

SO_TIMESTAMPING socket options

sk_shutdown

mask of SEND_SHUTDOWN and/or RCV_SHUTDOWN

sk_tskey
counter to disambiguate concurrent tstamp requests

sk_zckey
counter to order MSG_ZEROCOPY notifications

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_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

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 *__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.

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->rcv_nxt
...	...
tp->rcv_nxt check	sock_def_readable

(continues on next page)

(continued from previous page)

```
...      {
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->rcv_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` usage if the caller is potentially doing any of them. This assumes that page fault handlers use the GFP_NOFS flags.

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
timestamping 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_steal_sock**(struct *sk_buff* *skb, bool *refcounted)
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

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 the return is a ERR pointer (see linux/err.h). This function uses GFP_KERNEL internally.

struct *socket* ***sock_from_file**(struct *file* *file, int *err)

Return the *socket* bounded to **file**.

Parameters

struct *file* ***file**

file

int ***err**

pointer to an error code return

On failure returns NULL and assigns -ENOTSOCK to **err**.

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 functions 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 *sock* *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_recvmmsg**(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 0 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 0 or an error code.

int **kernel_sendpage**(struct *socket* *sock, struct *page* *page, int offset, size_t size, int flags)

send a *page* through a socket (kernel space)

Parameters

struct socket *sock

socket

struct page *page

page

int offset

page offset

size_t size

total size in bytes

int flags

flags (MSG_DONTWAIT, ...)

Returns the total amount sent in bytes or an error.

int **kernel_sendpage_locked**(struct *sock* *sk, struct *page* *page, int offset, size_t size, int flags)

send a *page* through the locked sock (kernel space)

Parameters

struct sock *sk

sock

struct page *page

page

int offset

page offset

size_t size

total size in bytes

int flags

flags (MSG_DONTWAIT, ...)

Returns the total amount sent in bytes or an error. Caller must hold **sk**.

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.

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* ***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, or 0 if head was kmalloced

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 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 **kfree_skb**(struct *sk_buff* *skb)
free an sk_buff

Parameters

struct sk_buff *skb
buffer to free

Drop a reference to the buffer and free it if the usage count has hit zero.

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 `SKBTX_DEV_ZEROCOPY` skb. 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)
reallocate 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_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_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**(struct sk_buff_head *list)

empty a list

Parameters

struct sk_buff_head *list

list to empty

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 zerocopy 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.

bool **skb_gso_validate_network_len**(const struct *sk_buff* *skb, unsigned int mtu)

Will a split GSO skb fit into a given MTU?

Parameters

const struct sk_buff *skb
GSO skb

unsigned int mtu
MTU to validate against

Description

skb_gso_validate_network_len validates if a given skb will fit a wanted MTU once split. It considers L3 headers, L4 headers, and the payload.

bool **skb_gso_validate_mac_len**(const struct *sk_buff* *skb, unsigned int len)
Will a split GSO skb fit in a given length?

Parameters

const struct sk_buff *skb
GSO skb

unsigned int len
length to validate against

Description

skb_gso_validate_mac_len validates if a given skb will fit a wanted length once split, including L2, L3 and L4 headers and the payload.

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 max_page_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 max_page_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_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.

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 *proto, 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_raise_allocated**(struct [sock](#) *sk, int size, int amt, int kind)

increase memory_allocated

Parameters

struct sock *sk

socket

int size

memory size to allocate

int amt

pages to allocate

int kind

allocation type

Similar to `__sk_mem_schedule()`, but does not update `sk_forward_alloc`

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_reduce_allocated(struct *sock* *sk, int amount)

reclaim `memory_allocated`

Parameters

struct sock *sk

socket

int amount

number of quanta

Similar to `__sk_mem_reclaim()`, but does not update `sk_forward_alloc`

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 `SK_MEM_QUANTUM` multiple)

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

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 wake-ups 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

14.1.3 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 **bpff_prog_create**(struct *bpff_prog* **pfp, struct *sock_fprog_kern* *fprog)
create an unattached filter

Parameters

struct bpff_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 **bpff_prog_create_from_user**(struct *bpff_prog* **pfp, struct *sock_fprog* *fprog, *bpff_aux_classic_check_t* trans, bool save_orig)
create an unattached filter from user buffer

Parameters

struct bpff_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.

14.1.4 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;
```

(continues on next page)

(continued from previous page)

```
__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 as a container for all other statistic TLVs.

The dumping handle is marked to be in backward compatibility mode telling all `gnets_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**(const seqcount_t *running, struct gnet_dump *d, struct gnet_stats_basic_cpu __percpu *cpu, struct gnet_stats_basic_packed *b)

copy basic statistics into statistic TLV

Parameters

const seqcount_t *running

seqcount_t pointer

struct gnet_dump *d

dumping handle

struct gnet_stats_basic_cpu __percpu *cpu

copy statistic per cpu

struct gnet_stats_basic_packed *b

basic statistics

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**(const seqcount_t *running, struct gnet_dump *d, struct gnet_stats_basic_cpu __percpu *cpu, struct gnet_stats_basic_packed *b)

copy basic hw statistics into statistic TLV

Parameters

const seqcount_t *running
seqcount_t pointer

struct gnet_dump *d
dumping handle

struct gnet_stats_basic_cpu __percpu *cpu
copy statistic per cpu

struct gnet_stats_basic_packed *b
basic statistics

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_packed *bstats, struct
                    gnet_stats_basic_cpu __percpu *cpu_bstats, struct
                    net_rate_estimator __rcu **rate_est, spinlock_t *lock,
                    seqcount_t *running, struct nlattr *opt)
    create a new rate estimator
```

Parameters

struct gnet_stats_basic_packed *bstats
basic statistics

struct gnet_stats_basic_cpu __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

seqcount_t *running
qdisc running seqcount

struct nlattr *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_packed *bstats, struct gnet_stats_basic_cpu __percpu *cpu_bstats, struct net_rate_estimator __rcu **rate_est, spinlock_t *lock, seqcount_t *running, struct nlattr *opt)
replace rate estimator configuration

Parameters

struct gnet_stats_basic_packed *bstats
basic statistics

struct gnet_stats_basic_cpu __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

seqcount_t *running
qdisc running seqcount (might be NULL)

struct nlattr *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.

14.1.5 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**(struct xdr_buf *buf, const u32 len)
 '0' -terminate a string residing in an xdr_buf

Parameters

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_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, struct kvec *vec, size_t
nbytes)

Reserves a large amount of buffer space for sending

Parameters

struct xdr_stream *xdr
pointer to xdr_stream

struct kvec *vec
pointer to a kvec array

size_t nbytes
number of bytes to reserve

Description

Reserves enough buffer space to encode ‘nbytes’ of data and stores the pointers in ‘vec’. 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.

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.

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->buflen 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

list of pages

unsigned int base

offset of first byte

unsigned int len

length of data in bytes

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_set_scratch_buffer(struct xdr_stream *xdr, void *buf, size_t buflen)

Attach a scratch buffer for decoding data.

Parameters

struct xdr_stream *xdr

pointer to xdr_stream struct

void *buf

pointer to an empty buffer

size_t buflen

size of 'buf'

Description

The scratch buffer is used when decoding from an array of pages. If an [*xdr_inline_decode\(\)*](#) call spans across page boundaries, then we copy the data into the scratch buffer in order to allow linear access.

__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)

Ensure page-based XDR data to decode is aligned at 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[].

Returns the number of XDR encoded bytes now contained in the pages

```
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(struct xdr_buf *buf, struct xdr_buf *subbuf, unsigned
                        int base, unsigned int len)
    set subbuf to a portion of buf
```

Parameters

```
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 of length are out of bounds.

```
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

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_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**.

```
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 xprt_class *transport
 transport to unregister

Return

0: transport successfully unregistered -ENOENT: transport never registered

int **xprt_load_transport**(const char *netid)
 load a transport implementation

Parameters

const char *netid
 transport to load

Return

0: transport successfully loaded -ENOENT: transport module not available

int **xprt_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 **xprt_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 **xprt_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 outstanding 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)

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.

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 *dummy)
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 *dummy
unused

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 *data)
```

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.

14.1.6 WiMAX

```
struct sk_buff *wimax_msg_alloc(struct wimax_dev *wimax_dev, const char
                                *pipe_name, const void *msg, size_t size, gfp_t
                                gfp_flags)
```

Create a new `skb` for sending a message to userspace

Parameters

struct wimax_dev *wimax_dev
WiMAX device descriptor

const char *pipe_name
“named pipe” the message will be sent to

const void *msg
pointer to the message data to send

size_t size
size of the message to send (in bytes), including the header.

gfp_t gfp_flags
flags for memory allocation.

Return

0 if ok, negative `errno` code on error

Description

Allocates an `skb` that will contain the message to send to user space over the messaging pipe and initializes it, copying the payload.

Once this call is done, you can deliver it with `wimax_msg_send()`.

IMPORTANT:

Don't use `skb_push()/skb_pull()/skb_reserve()` on the `skb`, as `wimax_msg_send()` depends on `skb->data` being placed at the beginning of the user message.

Unlike other WiMAX stack calls, this call can be used way early, even before `wimax_dev_add()` is called, as long as the `wimax_dev->net_dev` pointer is set to point to a proper `net_dev`. This is so that drivers can use it early in case they need to send stuff around or communicate with user space.

`const void *wimax_msg_data_len(struct sk_buff *msg, size_t *size)`

Return a pointer and size of a message's payload

Parameters

struct *sk_buff* *msg

Pointer to a message created with `wimax_msg_alloc()`

size_t *size

Pointer to where to store the message's size

Description

Returns the pointer to the message data.

`const void *wimax_msg_data(struct sk_buff *msg)`

Return a pointer to a message's payload

Parameters

struct *sk_buff* *msg

Pointer to a message created with `wimax_msg_alloc()`

`ssize_t wimax_msg_len(struct sk_buff *msg)`

Return a message's payload length

Parameters

struct *sk_buff* *msg

Pointer to a message created with `wimax_msg_alloc()`

`int wimax_msg_send(struct wimax_dev *wimax_dev, struct sk_buff *skb)`

Send a pre-allocated message to user space

Parameters

struct *wimax_dev* *wimax_dev

WiMAX device descriptor

struct *sk_buff* *skb

`struct sk_buff` returned by `wimax_msg_alloc()`. Note the ownership of `skb` is transferred to this function.

Return

0 if ok, < 0 errno code on error

Description

Sends a free-form message that was preallocated with `wimax_msg_alloc()` and filled up.

Assumes that once you pass an `skb` to this function for sending, it owns it and will release it when done (on success).

IMPORTANT:

Don't use `skb_push()/skb_pull()/skb_reserve()` on the `skb`, as `wimax_msg_send()` depends on `skb->data` being placed at the beginning of the user message.

Unlike other WiMAX stack calls, this call can be used way early, even before `wimax_dev_add()` is called, as long as the `wimax_dev->net_dev` pointer is set to point to a proper `net_dev`. This is so that drivers can use it early in case they need to send stuff around or communicate with user space.

```
int wimax_msg(struct wimax_dev *wimax_dev, const char *pipe_name, const void
               *buf, size_t size, gfp_t gfp_flags)
```

Send a message to user space

Parameters

struct wimax_dev *wimax_dev

WiMAX device descriptor (properly referenced)

const char *pipe_name

“named pipe” the message will be sent to

const void *buf

pointer to the message to send.

size_t size

size of the buffer pointed to by **buf** (in bytes).

gfp_t gfp_flags

flags for memory allocation.

Return

0 if ok, negative errno code on error.

Description

Sends a free-form message to user space on the device **wimax_dev**.

Once the **skb** is given to this function, who will own it and will release it when done (unless it returns error).

NOTES

```
int wimax_reset(struct wimax_dev *wimax_dev)
```

Reset a WiMAX device

Parameters

struct wimax_dev *wimax_dev

WiMAX device descriptor

Return**Description**

0 if ok and a warm reset was done (the device still exists in the system).

-ENODEV if a cold/bus reset had to be done (device has disconnected and reconnected, so current handle is not valid any more).

-EINVAL if the device is not even registered.

Any other negative error code shall be considered as non-recoverable.

Called when wanting to reset the device for any reason. Device is taken back to power on status.

This call blocks; on successful return, the device has completed the reset process and is ready to operate.

```
void wimax_report_rfkill_hw(struct wimax_dev *wimax_dev, enum
                           wimax_rf_state state)
```

Reports changes in the hardware RF switch

Parameters

```
struct wimax_dev *wimax_dev
```

WiMAX device descriptor

```
enum wimax_rf_state state
```

New state of the RF Kill switch. WIMAX_RF_ON radio on, WIMAX_RF_OFF radio off.

Description

When the device detects a change in the state of the hardware RF switch, it must call this function to let the WiMAX kernel stack know that the state has changed so it can be properly propagated.

The WiMAX stack caches the state (the driver doesn't need to). As well, as the change is propagated it will come back as a request to change the software state to mirror the hardware state.

If the device doesn't have a hardware kill switch, just report it on initialization as always on (WIMAX_RF_ON, radio on).

```
void wimax_report_rfkill_sw(struct wimax_dev *wimax_dev, enum
                           wimax_rf_state state)
```

Reports changes in the software RF switch

Parameters

```
struct wimax_dev *wimax_dev
```

WiMAX device descriptor

```
enum wimax_rf_state state
```

New state of the RF kill switch. WIMAX_RF_ON radio on, WIMAX_RF_OFF radio off.

Description

Reports changes in the software RF switch state to the WiMAX stack.

The main use is during initialization, so the driver can query the device for its current software radio kill switch state and feed it to the system.

On the side, the device does not change the software state by itself. In practice, this can happen, as the device might decide to switch (in software) the radio off for different reasons.

int **wimax_rfkill**(struct *wimax_dev* *wimax_dev, enum wimax_rf_state state)

Set the software RF switch state for a WiMAX device

Parameters

struct wimax_dev *wimax_dev

WiMAX device descriptor

enum wimax_rf_state state

New RF state.

Return

Description

≥ 0 toggle state if ok, < 0 errno code on error. The toggle state is returned as a bitmap, bit 0 being the hardware RF state, bit 1 the software RF state.

0 means disabled (WIMAX_RF_ON, radio on), 1 means enabled radio off (WIMAX_RF_OFF).

Called by the user when he wants to request the WiMAX radio to be switched on (WIMAX_RF_ON) or off (WIMAX_RF_OFF). With WIMAX_RF_QUERY, just the current state is returned.

This call will block until the operation is complete.

NOTE

void **wimax_state_change**(struct *wimax_dev* *wimax_dev, enum *wimax_st* new_state)

Set the current state of a WiMAX device

Parameters

struct wimax_dev *wimax_dev

WiMAX device descriptor (properly referenced)

enum wimax_st new_state

New state to switch to

Description

This implements the state changes for the wimax devices. It will

- verify that the state transition is legal (for now it'll just print a warning if not) according to the table in linux/wimax.h's documentation for '*enum wimax_st*'.
- perform the actions needed for leaving the current state and whichever are needed for entering the new state.
- issue a report to user space indicating the new state (and an optional payload with information about the new state).

NOTE

wimax_dev must be locked

enum *wimax_st* **wimax_state_get**(struct *wimax_dev* *wimax_dev)

Return the current state of a WiMAX device

Parameters

struct wimax_dev *wimax_dev

WiMAX device descriptor

Return

Current state of the device according to its driver.

void **wimax_dev_init**(struct *wimax_dev* *wimax_dev)

initialize a newly allocated instance

Parameters

struct wimax_dev *wimax_dev

WiMAX device descriptor to initialize.

Description

Initializes fields of a freshly allocated **wimax_dev** instance. This function assumes that after allocation, the memory occupied by **wimax_dev** was zeroed.

int **wimax_dev_add**(struct *wimax_dev* *wimax_dev, struct *net_device* *net_dev)

Register a new WiMAX device

Parameters

struct wimax_dev *wimax_dev

WiMAX device descriptor (as embedded in your **net_dev**'s priv data). You must have called *wimax_dev_init()* on it before.

struct net_device *net_dev

net device the **wimax_dev** is associated with. The function expects SET_NETDEV_DEV() and *register_netdev()* were already called on it.

Description

Registers the new WiMAX device, sets up the user-kernel control interface (generic netlink) and common WiMAX infrastructure.

Note that the parts that will allow interaction with user space are setup at the very end, when the rest is in place, as once that happens, the driver might get user space control requests via netlink or from debugfs that might translate into calls into *wimax_dev->op_**).

void **wimax_dev_rm**(struct *wimax_dev* *wimax_dev)

Unregister an existing WiMAX device

Parameters

struct wimax_dev *wimax_dev

WiMAX device descriptor

Description

Unregisters a WiMAX device previously registered for use with `wimax_add_rm()`.

IMPORTANT! Must call before calling `unregister_netdev()`.

After this function returns, you will not get any more user space control requests (via netlink or debugfs) and thus to `wimax_dev->ops`.

Reentrancy control is ensured by setting the state to `__WIMAX_ST QUIESCING`. rfkill operations coming through `wimax_*rfkill*()` will be stopped by the quiescing state; ops coming from the rfkill subsystem will be stopped by the support being removed by `wimax_rfkill_rm()`.

struct **wimax_dev**

Generic WiMAX device

Definition

```
struct wimax_dev {
    struct net_device *net_dev;
    struct list_head id_table_node;
    struct mutex mutex;
    struct mutex mutex_reset;
    enum wimax_st state;
    int (*op_msg_from_user)(struct wimax_dev *wimax_dev, const char *,
    ↪const void *, size_t, const struct genl_info *info);
    int (*op_rfkill_sw_toggle)(struct wimax_dev *wimax_dev, enum ↪
    ↪wimax_rf_state);
    int (*op_reset)(struct wimax_dev *wimax_dev);
    struct rfkill *rfkill;
    unsigned int rf_hw;
    unsigned int rf_sw;
    char name[32];
    struct dentry *debugfs_dentry;
};
```

Members

net_dev

[fill] Pointer to the `struct net_device` this WiMAX device implements.

id_table_node

[private] link to the list of wimax devices kept by id-table.c. Protected by it's own spinlock.

mutex

[private] Serializes all concurrent access and execution of operations.

mutex_reset

[private] Serializes reset operations. Needs to be a different mutex because as part of the reset operation, the driver has to call back into the stack to do things such as state change, that require `wimax_dev->mutex`.

state

[private] Current state of the WiMAX device.

op_msg_from_user

[fill] Driver-specific operation to handle a raw message from user space to the driver. The driver can send messages to user space using with `wimax_msg_to_user()`.

op_rfkill_sw_toggle

[fill] Driver-specific operation to act on userspace (or any other agent) requesting the WiMAX device to change the RF Kill software switch (`WIMAX_RF_ON` or `WIMAX_RF_OFF`). If such hardware support is not present, it is assumed the radio cannot be switched off and it is always on (and the stack will error out when trying to switch it off). In such case, this function pointer can be left as `NULL`.

op_reset

[fill] Driver specific operation to reset the device. This operation should always attempt first a warm reset that does not disconnect the device from the bus and return 0. If that fails, it should resort to some sort of cold or bus reset (even if it implies a bus disconnection and device disappearance). In that case, `-ENODEV` should be returned to indicate the device is gone. This operation has to be synchronous, and return only when the reset is complete. In case of having had to resort to bus/cold reset implying a device disconnection, the call is allowed to return immediately.

rfkill

[private] integration into the RF-Kill infrastructure.

rf_hw

[private] State of the hardware radio switch (OFF/ON)

rf_sw

[private] State of the software radio switch (OFF/ON)

name

[fill] A way to identify this device. We need to register a name with many subsystems (rfkill, workqueue creation, etc). We can't use the network device name as that might change and in some instances we don't know it yet (until we don't call `register_netdev()`). So we generate an unique one using the driver name and device bus id, place it here and use it across the board. Recommended naming: `DRIVERNAME-BUSNAME:BUSID` (`dev->bus->name`, `dev->bus_id`).

debugfs_dentry

[private] Used to hook up a debugfs entry. This shows up in the debugfs root as `wimax:DEVICENAME`.

NOTE**wimax_dev->mutex is NOT locked when this op is being**

called; however, `wimax_dev->mutex_reset` IS locked to ensure serialization of calls to `wimax_reset()`. See `wimax_reset()`'s documentation.

Description

This structure defines a common interface to access all WiMAX devices from different vendors and provides a common API as well as a free-form device-specific messaging channel.

Usage:

1. Embed a `struct wimax_dev` at the beginning the network device structure so that `netdev_priv()` points to it.
2. `memset()` it to zero
3. Initialize with `wimax_dev_init()`. This will leave the WiMAX device in the `__WIMAX_ST_NULL` state.
4. Fill all the fields marked with [fill]; once called `wimax_dev_add()`, those fields CANNOT be modified.
5. Call `wimax_dev_add()` after registering the network device. This will leave the WiMAX device in the `WIMAX_ST_DOWN` state. Protect the driver's `net_device->open()` against succeeding if the wimax device state is lower than `WIMAX_ST_DOWN`.
6. Select when the device is going to be turned on/initialized; for example, it could be initialized on 'ifconfig up' (when the netdev op 'open()' is called on the driver).

When the device is initialized (at *ifconfig up* time, or right after calling `wimax_dev_add()` from `_probe()`), make sure the following steps are taken

- a. Move the device to `WIMAX_ST_UNINITIALIZED`. This is needed so some API calls that shouldn't work until the device is ready can be blocked.
- b. Initialize the device. Make sure to turn the SW radio switch off and move the device to state `WIMAX_ST_RADIO_OFF` when done. When just initialized, a device should be left in `RADIO OFF` state until user space devices to turn it on.
- c. Query the device for the state of the hardware rfkill switch and call `wimax_rfkill_report_hw()` and `wimax_rfkill_report_sw()` as needed. See below.

`wimax_dev_rm()` undoes before unregistering the network device. Once `wimax_dev_add()` is called, the driver can get called on the `wimax_dev->op_*` function pointers

CONCURRENCY:

The stack provides a mutex for each device that will disallow API calls happening concurrently; thus, op calls into the driver through the `wimax_dev->op*()` function pointers will always be serialized and *never* concurrent.

For locking, take `wimax_dev->mutex` is taken; (most) operations in the API have to check for `wimax_dev_is_ready()` to return 0 before continuing (this is done internally).

REFERENCE COUNTING:

The WiMAX device is reference counted by the associated network device. The only operation that can be used to reference the device is `wimax_dev_get_by_genl_info()`, and the reference it acquires has to be released with `dev_put(wimax_dev->net_dev)`.

RFKILL:

At startup, both HW and SW radio switchess are assumed to be off.

At initialization time [after calling `wimax_dev_add()`], have the driver query the device for the status of the software and hardware RF kill switches and call `wimax_report_rfkill_hw()` and `wimax_rfkill_report_sw()` to indicate their state. If any is missing, just call it to indicate it is ON (radio always on).

Whenever the driver detects a change in the state of the RF kill switches, it should call `wimax_report_rfkill_hw()` or `wimax_report_rfkill_sw()` to report it to the stack.

enum **wimax_st**

The different states of a WiMAX device

Constants

__WIMAX_ST_NULL

The device structure has been allocated and zeroed, but still `wimax_dev_add()` hasn't been called. There is no state.

WIMAX_ST_DOWN

The device has been registered with the WiMAX and networking stacks, but it is not initialized (normally that is done with 'ifconfig DEV up' [or equivalent], which can upload firmware and enable communications with the device). In this state, the device is powered down and using as less power as possible. This state is the default after a call to `wimax_dev_add()`. It is ok to have drivers move directly to `WIMAX_ST_UNINITIALIZED` or `WIMAX_ST_RADIO_OFF` in `_probe()` after the call to `wimax_dev_add()`. It is recommended that the driver leaves this state when calling 'ifconfig DEV up' and enters it back on 'ifconfig DEV down' .

__WIMAX_ST QUIESCING

The device is being torn down, so no API operations are allowed to proceed except the ones needed to complete the device clean up process.

WIMAX_ST_UNINITIALIZED

[optional] Communication with the device is setup, but the device still requires some configuration before being operational. Some WiMAX API calls might work.

WIMAX_ST_RADIO_OFF

The device is fully up; radio is off (wether by hardware or software switches). It is recommended to always leave the device in this state after initialization.

WIMAX_ST_READY

The device is fully up and radio is on.

WIMAX_ST_SCANNING

[optional] The device has been instructed to scan. In this state, the device cannot be actively connected to a network.

WIMAX_ST_CONNECTING

The device is connecting to a network. This state exists because in some devices, the connect process can include a number of negotiations between user space, kernel space and the device. User space needs to know what the device is doing. If the connect sequence in a device is atomic and fast, the device can transition directly to `CONNECTED`

WIMAX_ST_CONNECTED

The device is connected to a network.

__WIMAX_ST_INVALID

This is an invalid state used to mark the maximum numeric value of states.

Description

Transitions from one state to another one are atomic and can only be caused in kernel space with `wimax_state_change()`. To read the state, use `wimax_state_get()`.

States starting with `__` are internal and shall not be used or referred to by drivers or userspace. They look ugly, but that's the point – if any use is made non-internal to the stack, it is easier to catch on review.

All API operations [with well defined exceptions] will take the device mutex before starting and then check the state. If the state is `__WIMAX_ST_NULL`, `WIMAX_ST_DOWN`, `WIMAX_ST_UNINITIALIZED` or `__WIMAX_ST_QUIESCING`, it will drop the lock and quit with `-EINVAL`, `-ENOMEDIUM`, `-ENOTCONN` or `-ESHUTDOWN`.

The order of the definitions is important, so we can do numerical comparisons (eg: `< WIMAX_ST_RADIO_OFF` means the device is not ready to operate).

14.2 Network device support

14.2.1 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.

void **dev_add_offload**(struct packet_offload *po)
register offload handlers

Parameters

struct packet_offload *po
protocol offload declaration

Add protocol offload handlers to the networking stack. The passed proto_offload 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 offload handlers (until the next received packet).

void **dev_remove_offload**(struct packet_offload *po)
remove packet offload handler

Parameters

struct packet_offload *po
packet offload declaration

Remove a packet offload handler that was previously added to the kernel offload handlers by [dev_add_offload\(\)](#). The passed offload_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 **netdev_boot_setup_check**(struct [net_device](#) *dev)
check boot time settings

Parameters

struct net_device *dev
the netdevice

Description

Check boot time settings for the device. The found settings are set for the device to be used later in the device probing. Returns 0 if no settings found, 1 if they are.

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* ***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. 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 *dev_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* ***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 returned has had a reference added and the pointer is safe until the user calls `dev_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**

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\(\)](#). 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 **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_get_num_default_rss_queues**(void)

default number of RSS queues

Parameters

void

no arguments

Description

This routine should set an upper limit on the number of RSS queues used by default by multiqueue devices.

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.

struct *sk_buff* ***skb_mac_gso_segment**(struct *sk_buff* *skb, netdev_features_t features)

mac layer segmentation handler.

Parameters

struct sk_buff *skb
buffer to segment

netdev_features_t features
features for the output path (see dev->features)

struct *sk_buff* ***__skb_gso_segment**(struct *sk_buff* *skb, netdev_features_t features, bool tx_path)

Perform segmentation on skb.

Parameters

struct sk_buff *skb
buffer to segment

netdev_features_t features
features for the output path (see dev->features)

bool tx_path
whether it is called in TX path

This function segments the given skb and returns a list of segments.

It may return NULL if the skb requires no segmentation. This is only possible when GSO is used for verifying header integrity.

Segmentation preserves SKB_GSO_CB_OFFSET bytes of previous skb cb.

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

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)
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. It always succeeds. The buffer may be dropped during processing for congestion control or by the protocol layers.

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 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_)pfnmemalloc`.

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.

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.

```
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.

void **dev_set_group**(struct *net_device* *dev, int new_group)

Change group this device belongs to

Parameters

struct net_device *dev
device

int new_group
group this device should belong to

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_change_carrier**(struct *net_device* *dev, bool new_carrier)

Change device carrier

Parameters

struct net_device *dev
device

bool new_carrier
new value

Change device carrier

int **dev_get_phys_port_id**(struct *net_device* *dev, struct netdev_phys_item_id *ppid)

Get device physical port ID

Parameters

struct net_device *dev
device

struct netdev_phys_item_id *ppid
port ID

Get device physical port ID

int **dev_get_phys_port_name**(struct *net_device* *dev, char *name, size_t len)
Get device physical port name

Parameters

struct net_device *dev
device

char *name
port name

size_t len
limit of bytes to copy to name

Get device physical port name

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' port parent identifier

bool **netdev_port_same_parent_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

int **dev_change_proto_down**(struct *net_device* *dev, bool proto_down)
update protocol port state information

Parameters

struct net_device *dev
device

bool proto_down
new value

This info can be used by switch drivers to set the phys state of the port.

int **dev_change_proto_down_generic**(struct *net_device* *dev, bool proto_down)
generic implementation for `ndo_change_proto_down` that sets carrier according to `proto_down`.

Parameters

struct net_device *dev
device

bool proto_down
new value

void **dev_change_proto_down_reason**(struct *net_device* *dev, unsigned long mask, u32 value)
proto down reason

Parameters

struct net_device *dev
device

unsigned long mask
proto down mask

u32 value
proto down value

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

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.

Callers must hold the rtnl semaphore. You may want `register_netdev()` instead of this.

BUGS: The locking appears insufficient to guarantee two parallel registers will not get the same name.

```
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**.

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)
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.

This function shuts down a device interface and moves it to a new network namespace. On success 0 is returned, on a failure a netagive 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.

u32 **eth_get_headlen**(const struct *net_device* *dev, void *data, unsigned int len)

determine the length of header for an ethernet frame

Parameters

const struct net_device *dev

pointer to network device

void *data

pointer to start of frame

unsigned int 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 **nvmem_get_mac_address**(struct device *dev, void *addrbuf)
address' associated with given device.

Parameters

struct device *dev
Device with which the mac-address cell is associated.

void *addrbuf
Buffer to which the MAC address will be copied on success.

Description

Returns 0 on success or a negative error number on failure.

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.

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.

```
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 is 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_skb_pkt_type(struct sk_buff *skb, const struct net_device *dev)
```

Assign packet type if destination address does not match

Parameters

```
struct sk_buff *skb
```

Assigned a packet type if address does not match **dev** address

```
const struct net_device *dev
```

Network device used to compare packet address against

Description

If the destination MAC address of the packet does not match the network device address, assign an appropriate packet type.

```
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.

```
void 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.

```
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**(struct napi_struct *n)
NAPI processing complete

Parameters

struct napi_struct *n
NAPI context

Description

Mark NAPI processing as complete. Consider using `napi_complete_done()` instead. 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_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`.

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_NOFCS

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_LIVE_RENAME_OK

rename is allowed while device is up and running

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 {
    char name[IFNAMSIZ];
    struct netdev_name_node *name_node;
```

(continues on next page)

(continued from previous page)

```
struct dev_ifalias      __rcu *ifalias;
unsigned long           mem_end;
unsigned long           mem_start;
unsigned long           base_addr;
int irq;
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 list_head        ptype_specific;
struct {
    struct list_head upper;
    struct list_head lower;
} adj_list;
netdev_features_t features;
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;
netdev_features_t gso_partial_features;
int ifindex;
int group;
struct net_device_stats stats;
atomic_long_t rx_dropped;
atomic_long_t tx_dropped;
atomic_long_t rx_nohandler;
atomic_t carrier_up_count;
atomic_t carrier_down_count;
#ifdef CONFIG_WIRELESS_EXT;
    const struct iw_handler_def *wireless_handlers;
    struct iw_public_data *wireless_data;
#endif;
const struct net_device_ops *netdev_ops;
const struct ethtool_ops *ethtool_ops;
#ifdef 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;
```

(continues on next page)

(continued from previous page)

```

const struct header_ops *header_ops;
unsigned int flags;
unsigned int priv_flags;
unsigned short gflags;
unsigned short padded;
unsigned char operstate;
unsigned char link_mode;
unsigned char if_port;
unsigned char dma;
unsigned int mtu;
unsigned int min_mtu;
unsigned int max_mtu;
unsigned short type;
unsigned short hard_header_len;
unsigned char min_header_len;
unsigned char name_assign_type;
unsigned short needed_headroom;
unsigned short needed_tailroom;
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;
spinlock_t addr_list_lock;
struct netdev_hw_addr_list uc;
struct netdev_hw_addr_list mc;
struct netdev_hw_addr_list dev_addrs;
#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;
#ifdef CONFIG_LOCKDEP;
unsigned char nested_level;
#endif;
#if IS_ENABLED(CONFIG_VLAN_8021Q);
struct vlan_info __rcu *vlan_info;
#endif;
#if IS_ENABLED(CONFIG_NET_DSA);
struct dsa_port *dsa_ptr;
#endif;
#if IS_ENABLED(CONFIG_TIPC);

```

(continues on next page)

(continued from previous page)

```

    struct tipc_bearer __rcu *tipc_ptr;
#endif;
#if IS_ENABLED(CONFIG_IRDA) || IS_ENABLED(CONFIG_ATALK);
    void *atalk_ptr;
#endif;
    struct in_device __rcu *ip_ptr;
    struct inet6_dev __rcu *ip6_ptr;
#if IS_ENABLED(CONFIG_AX25);
    void *ax25_ptr;
#endif;
    struct wireless_dev *ieee80211_ptr;
    struct wpan_dev *ieee802154_ptr;
#if IS_ENABLED(CONFIG_MPLS_ROUTING);
    struct mpls_dev __rcu *mpls_ptr;
#endif;
    unsigned char *dev_addr;
    struct netdev_rx_queue *_rx;
    unsigned int num_rx_queues;
    unsigned int real_num_rx_queues;
    struct bpf_prog __rcu *xdp_prog;
    unsigned long gro_flush_timeout;
    int napi_defer_hard_irqs;
    rx_handler_func_t __rcu *rx_handler;
    void __rcu *rx_handler_data;
#ifdef CONFIG_NET_CLS_ACT;
    struct mini_Qdisc __rcu *miniq_ingress;
#endif;
    struct netdev_queue __rcu *ingress_queue;
#ifdef CONFIG_NETFILTER_INGRESS;
    struct nf_hook_entries __rcu *nf_hooks_ingress;
#endif;
    unsigned char broadcast[MAX_ADDR_LEN];
#ifdef CONFIG_RFS_ACCEL;
    struct cpu_rmap *rx_cpu_rmap;
#endif;
    struct hlist_node index_hlist;
    struct netdev_queue *_tx;
    unsigned int num_tx_queues;
    unsigned int real_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;
#ifdef CONFIG_XPS;
    struct xps_dev_maps __rcu *xps_cpus_map;
    struct xps_dev_maps __rcu *xps_rxqs_map;
#endif;
#ifdef CONFIG_NET_CLS_ACT;
    struct mini_Qdisc __rcu *miniq_egress;

```

(continues on next page)

(continued from previous page)

```

#endif;
#ifdef 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;
    int __percpu           *pcpu_refcnt;
    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);
#ifdef CONFIG_NETPOLL;
    struct netpoll_info __rcu      *npinfo;
#endif;
    possible_net_t nd_net;
    void *ml_priv;
    enum netdev_ml_priv_type      ml_priv_type;
    union {
        struct pcpu_lstats __percpu      *lstats;
        struct pcpu_sw_netstats __percpu  *tstats;
        struct pcpu_dstats __percpu      *dstats;
    };
#ifdef IS_ENABLED(CONFIG_GARP);
    struct garp_port __rcu      *garp_port;
#endif;
#ifdef IS_ENABLED(CONFIG_MRP);
    struct mrp_port __rcu      *mrp_port;
#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_SIZE      65536;
    unsigned int      gso_max_size;
#define GSO_MAX_SEGS      65535;

```

(continues on next page)

(continued from previous page)

```
    u16 gso_max_segs;
#ifdef CONFIG_DCB;
    const struct dcbnl_rtnl_ops *dcbnl_ops;
#endif;
    s16 num_tc;
    struct netdev_tc_txq    tc_to_txq[TC_MAX_QUEUE];
    u8 prio_tc_map[TC_BITMASK + 1];
#ifdef IS_ENABLED(CONFIG_FCOE);
    unsigned int            fcoe_ddp_xid;
#endif;
#ifdef 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;
    struct lock_class_key    *qdisc_running_key;
    bool proto_down;
    unsigned wol_enabled:1;
    struct list_head         net_notifier_list;
#ifdef 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];
};
```

Members

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

irq

Device IRQ number

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

ptype_specific

Device-specific, protocol-specific packet handlers

adj_list

Directly linked devices, like slaves for bonding

features

Currently active device features

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

gso_partial_features

value(s) from `NETIF_F_GSO*`

ifindex

interface index

group

The group the device belongs to

stats

Statistics struct, which was left as a legacy, use `rtnl_link_stats64` instead

rx_dropped

Dropped packets by core network, do not use this in drivers

tx_dropped

Dropped packets by core network, do not use this in drivers

rx_nohandler

nohandler dropped packets by core network on inactive devices, 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

netdev_ops

Includes several pointers to callbacks, if one wants to override the ndo_*() functions

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

header_ops

Includes callbacks for creating,parsing,caching,etc of Layer 2 headers.

flags

Interface flags (a la BSD)

priv_flags

Like 'flags' but invisible to userspace, see if.h for the definitions

gflags

Global flags (kept as legacy)

padded

How much padding added by alloc_netdev()

operstate

RFC2863 operstate

link_mode

Mapping policy to operstate

if_port

Selectable AUI, TP, ...

dma
DMA channel

mtu
Interface MTU value

min_mtu
Interface Minimum MTU value

max_mtu
Interface Maximum MTU value

type
Interface hardware type

hard_header_len
Maximum hardware header length.

min_header_len
Minimum hardware header length

name_assign_type
network interface name assignment type

needed_headroom
Extra headroom the hardware may need, but not in all cases can this be guaranteed

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:

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

addr_list_lock
XXX: need comments on this one

uc

unicast mac addresses

mc

multicast mac addresses

dev_addr

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.

FIXME: cleanup *struct net_device* such that network protocol info moves out.

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`.

vlan_info

VLAN info

dsa_ptr

dsa specific data

tipc_ptr

TIPC specific data

atalk_ptr

AppleTalk link

ip_ptr

IPv4 specific data

ip6_ptr

IPv6 specific data

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

dev_addr
Hw address (before bcast, because most packets are unicast)

_rx
Array of RX queues

num_rx_queues
Number of RX queues allocated at *register_netdev()* time

real_num_rx_queues
Number of RX queues currently active in device

xdp_prog
XDP sockets filter program pointer

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.

rx_handler
handler for received packets

rx_handler_data
XXX: need comments on this one

miniq_ingress
ingress/clsact qdisc specific data for ingress processing

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

_tx
Array of TX queues

num_tx_queues
Number of TX queues allocated at `alloc_netdev_mq()` time

real_num_tx_queues
Number of TX queues currently active in device

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

xps_cpus_map

all CPUs map for XPS device

xps_rxqs_map

all RXQs map for XPS device

miniq_egress

clsact qdisc specific data for egress processing

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

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

npinfo

XXX: need comments on this one

nd_net
Network namespace this network device is inside

ml_priv
Mid-layer private

ml_priv_type
Mid-layer private type

{unnamed_union}
anonymous

lstats
Loopback statistics

tstats
Tunnel statistics

dstats
Dummy statistics

garp_port
GARP

mrp_port
MRP

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

gso_max_size
Maximum size of generic segmentation offload

gso_max_segs
Maximum number of segments that can be passed to the NIC for GSO

dcbnl_ops
Data Center Bridging netlink ops

num_tc
Number of traffic classes in the net device

tc_to_txq
XXX: need comments on this one

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

qdisc_running_key

lockdep class annotating Qdisc->running seqcount

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

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

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), int weight)

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

int weight
default weight

Description

[`netif_napi_add\(\)`](#) must be used to initialize a NAPI context prior to calling *any* of the other NAPI-related functions.

void **netif_tx_napi_add**(struct [`net_device`](#) *dev, struct napi_struct *napi, int (*poll)(struct napi_struct*, int), int weight)
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

int weight
default weight

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_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_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. **bytes** should be a good approximation and should exactly match *netdev_completed_queue()* **bytes**

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.

```
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 \geq 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_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.

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.

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**(struct *net_device* *dev)
is device available or removed

Parameters

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*))

Synchronize 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*))

Synchronize 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().

14.2.2 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_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.

```
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_do_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_do_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
```

```
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_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_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 reportes 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_clear_interrupt**(struct *phy_device* *phydev)
Ack the phy device' s interrupt

Parameters

struct phy_device *phydev
the phy_device struct

Description

If the **phydev** driver has an `ack_interrupt` function, call it to ack and clear the phy device' s interrupt.

Returns 0 on success or < 0 on error.

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.

```
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.

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.

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_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

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.

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.

```
void phy_error(struct phy_device *phydev)
    enter HALTED state for this PHY device
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

Description

Moves the PHY to the HALTED state in response to a read or write error, and tells the controller the link is down. Must not be called from interrupt context, or while the phydev->lock is held.

```
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
```

```
int phy_did_interrupt(struct phy_device *phydev)
    Checks if the PHY generated an interrupt
```

Parameters

```
struct phy_device *phydev
    target phy_device struct
```

```
irqreturn_t phy_handle_interrupt(struct phy_device *phydev)
    Handle PHY interrupt
```

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 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.

int **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_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_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 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_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_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 an 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.

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
Median-independent interface

PHY_INTERFACE_MODE_GMII
Gigabit median-independent interface

PHY_INTERFACE_MODE_SGMII
Serial gigabit media-independent interface

PHY_INTERFACE_MODE_TBI
Ten Bit Interface

PHY_INTERFACE_MODE_REVMII
Reverse Media Independent Interface

PHY_INTERFACE_MODE_RMII
Reduced Media Independent Interface

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_SMI

??? 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_QSGMII

Quad SGMII

PHY_INTERFACE_MODE_TRGMII

Turbo RGMII

PHY_INTERFACE_MODE_1000BASEX

1000 BaseX

PHY_INTERFACE_MODE_2500BASEX

2500 BaseX

PHY_INTERFACE_MODE_RXAUI

Reduced XAUI

PHY_INTERFACE_MODE_XAUI

10 Gigabit Attachment Unit Interface

PHY_INTERFACE_MODE_10GBASER

10G BaseR

PHY_INTERFACE_MODE_USXGMII

Universal Serial 10GE MII

PHY_INTERFACE_MODE_10GKR

10GBASE-KR - 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 {
    int addr;
    refcount_t refcnt;
    unsigned long flags;
    size_t priv_size;
    void *priv;
};
```

Members**addr**

Common PHY address used to combine PHYs in one package

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 (*reset)(struct mii_bus *bus);
    struct mdio_bus_stats stats[PHY_MAX_ADDR];
    struct mutex mdio_lock;
    struct device *parent;
    enum {
        MDIOBUS_ALLOCATED = 1,
        MDIOBUS_REGISTERED,
        MDIOBUS_UNREGISTERED,
        MDIOBUS_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;
    enum {
        MDIOBUS_NO_CAP = 0,
        MDIOBUS_C22,
        MDIOBUS_C45,
        MDIOBUS_C22_C45,
    } probe_capabilities;
    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

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

probe_capabilities
bus capabilities, used for probing

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. Or PHY is in an error state. - phy_start moves to **PHY_UP**

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 it 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;
    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 autoneg:1;
    unsigned link:1;
    unsigned autoneg_complete:1;
    unsigned interrupts:1;
    enum phy_state state;
    u32 dev_flags;
    phy_interface_t interface;
    int speed;
    int duplex;
    int port;
    int pause;
    int asym_pause;
    u8 master_slave_get;
    u8 master_slave_set;
```

(continues on next page)

(continued from previous page)

```
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)];
u32 eee_broken_modes;
#ifdef 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;
int irq;
void *priv;
struct phy_package_shared *shared;
struct sk_buff *skb;
void *ehdr;
struct nlattnr *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;
u8 mdix;
u8 mdix_ctrl;
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

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.

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

state

State of the PHY for management purposes

dev_flags

Device-specific flags used by the PHY driver.

interface

enum phy_interface_t value

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

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

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

mdix

Current crossover

mdix_ctrl

User setting of crossover

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_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 (*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 (*ack_interrupt)(struct phy_device *phydev);
    int (*config_intr)(struct phy_device *phydev);
    int (*did_interrupt)(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);
}
```

(continues on next page)

(continued from previous 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);
};

```

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.

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

ack_interrupt

Clears any pending interrupts

config_intr

Enables or disables interrupts

did_interrupt

Checks if the PHY generated an interrupt. For multi-PHY devices with shared PHY interrupt pin Set interrupt bits have to be cleared.

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

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_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_IGNORE_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_hwtstamp**(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_txtstamp**(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_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.

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 addr, size_t priv_size)
```

join a common PHY group

Parameters

struct phy_device *phydev

target phy_device struct

int addr

cookie and PHY address for 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 `addr` parameter serves as a cookie which has to have the same value for all members of one group and as a PHY address 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. If your PHY doesn't have global registers you can just pick any of the PHY addresses.

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
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 addr
cookie and PHY address for 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 is 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 is 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 adverting, 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.

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 (the driver's init function should set it to STARTING if needed).

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_deviced'ed 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_mdio_register(), the mii_bus cannot be found before it is registered with of_mdio_register().

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_mdio_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.

struct *phy_device* ***mdiobus_scan**(struct *mii_bus* *bus, int addr)

scan a bus for MDIO devices.

Parameters

struct *mii_bus* ***bus**

mii_bus to scan

int addr
address on bus to scan

Description

This function scans 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_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_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_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_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

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.

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.

14.2.3 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;
    unsigned int link:1;
    unsigned int an_enabled: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.

link

true if the link is up.

an_enabled

true if autonegotiation is enabled/desired.

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 pcs_poll;
    bool poll_fixed_state;
    void (*get_fixed_state)(struct phylink_config *config, struct
↪ phylink_link_state *state);
};

```

Members

dev

a pointer to a struct device associated with the MAC

type

operation type of PHYLINK instance

pcs_poll

MAC PCS cannot provide link change interrupt

poll_fixed_state

if true, starts link_poll, if MAC link is at MLO_AN_FIXED mode.

get_fixed_state

callback to execute to determine the fixed link state, if MAC link is at MLO_AN_FIXED mode.

struct phylink_mac_ops

MAC operations structure.

Definition

```

struct phylink_mac_ops {
    void (*validate)(struct phylink_config *config,unsigned long
↪ *supported, struct phylink_link_state *state);
    void (*mac_pcs_get_state)(struct phylink_config *config, struct
↪ phylink_link_state *state);
    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_an_restart)(struct phylink_config *config);
    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

validate

Validate and update the link configuration.

mac_pcs_get_state

Read the current link state from the hardware.

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_an_restart

restart 802.3z BaseX autonegotiation.

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.

void **validate**(struct *phylink_config* *config, unsigned long *supported, struct *phylink_link_state* *state)

Validate and update the link configuration

Parameters

struct phylink_config *config

a pointer to a *struct phylink_config*.

unsigned long *supported

ethtool bitmask for supported link modes.

struct phylink_link_state *state

a pointer to a *struct phylink_link_state*.

Description

Clear bits in the **supported** and **state->advertising** masks that are not supportable by the MAC.

Note that the PHY may be able to transform from one connection technology to another, so, eg, don't clear 1000BaseX just because the MAC is unable to BaseX mode. This is more about clearing unsupported speeds and duplex settings. The port modes should not be cleared; *phylink_set_port_modes()* will help with this.

If the **state->interface** mode is PHY_INTERFACE_MODE_1000BASEX or PHY_INTERFACE_MODE_2500BASEX, select the appropriate mode based on **state->advertising** and/or **state->speed** and update **state->interface** accordingly. See *phylink_helper_basex_speed()*.

When **state->interface** is PHY_INTERFACE_MODE_NA, phylink expects the MAC driver to return all supported link modes.

If the **state->interface** mode is not supported, then the **supported** mask must be cleared.


```
void mac_pcs_get_state(struct phylink_config *config, struct phylink_link_state
                      *state)
```

Read the current inband link state from the hardware

Parameters

struct *phylink_config* *config
a pointer to a *struct phylink_config*.

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 MLO_PAUSE_RX and MLO_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.

```
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 requires a rewrite - please refer to *mac_link_up()* for situations where the PCS and MAC are not tightly integrated.)

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 *mac_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_an_restart**(struct *phylink_config* *config)

restart 802.3z BaseX autonegotiation

Parameters

struct phylink_config *config

a pointer to a *struct phylink_config*.

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;  
    bool poll;  
};
```

Members

ops

a pointer to the *struct phylink_pcs_ops* structure

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.

struct **phylink_pcs_ops**

MAC PCS operations structure.

Definition

```

struct phylink_pcs_ops {
    void (*pcs_get_state)(struct phylink_pcs *pcs, struct phylink_
↪link_state *state);
    int (*pcs_config)(struct phylink_pcs *pcs, unsigned int 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 mode,
↪phy_interface_t interface, int speed, int duplex);
};

```

Members

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).

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 MLO_PAUSE_RX and MLO_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 *mac_pcs_get_state()* in *struct phylink_mac_ops*.

```
int pcs_config(struct phylink_pcs *pcs, unsigned int 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 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

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 MLO_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.

```
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 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 mode
link autonegotiation mode

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.

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.

struct phylink *phylink_create(struct *phylink_config* *config, 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*

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_set_pcs**(struct *phylink* *pl, struct *phylink_pcs* *pcs)

set the current PCS for phylink to use

Parameters

struct phylink *pl

a pointer to a *struct phylink* returned from *phylink_create()*

struct phylink_pcs *pcs

a pointer to the *struct phylink_pcs*

Description

Bind the MAC PCS to phylink. This may be called after *phylink_create()*, in *mac_prepare()* or *mac_config()* methods if it is desired to dynamically change the PCS.

Please note that there are behavioural changes with the *mac_config()* callback if a PCS is present (denoting a newer setup) so removing a PCS is not supported, and if a PCS is going to be used, it must be registered by calling *phylink_set_pcs()* at the latest in the first *mac_config()* call.

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.

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_phy()* will be performed.

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_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.

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.

SIOCGMIIPHY:

read register from the current PHY.

SIOCGMIIREG:

read register from the specified PHY.

SIOSMIIREG:

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_helper_basex_speed**(struct *phylink_link_state* *state)
1000BaseX/2500BaseX helper

Parameters

struct phylink_link_state *state

a pointer to a *struct phylink_link_state*

Description

Inspect the interface mode, advertising mask or forced speed and decide whether to run at 2.5Gbit or 1Gbit appropriately, switching the interface mode to suit. **state->interface** is appropriately updated, and the advertising mask has the “other” baseX_Full flag cleared.

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_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 `mac_pcs_get_state()` member of the `struct phylink_mac_ops` structure.

```
int phylink_mii_c22_pcs_set_advertisement(struct mdio_device *pcs,
                                         phy_interface_t interface, const
                                         unsigned long *advertising)
```

configure the clause 37 PCS advertisement

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

Description

Helper for MAC PCS supporting the 802.3 clause 22 register set for clause 37 negotiation and/or SGMII control.

Configure the clause 37 PCS advertisement as specified by **state**. This does not trigger a renegotiation; phylink will do that via the `mac_an_restart()` method of the `struct phylink_mac_ops` structure.

Returns negative error code on failure to configure the advertisement, zero if no change has been made, or one if the advertisement has changed.

```
int phylink_mii_c22_pcs_config(struct mdio_device *pcs, unsigned int mode,
                              phy_interface_t interface, const unsigned
                              long *advertising)
```

configure clause 22 PCS

Parameters

struct mdio_device *pcs
a pointer to a struct mdio_device.

unsigned int mode
link autonegotiation mode

phy_interface_t interface
the PHY interface mode being configured

const unsigned long *advertising
the ethtool advertisement mask

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 *mac_pcs_get_state()* member of the *struct phylink_mac_ops* structure.

14.2.4 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)

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

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.

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.

struct *sfp_bus* ***sfp_bus_find_fwnode**(struct fwnode_handle *fwnode)

parse and locate the SFP bus from fwnode

Parameters

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

15.1 Intro

The MSG_ZEROCOPY flag enables copy avoidance for socket send calls. The feature is currently implemented for TCP and UDP sockets.

15.1.1 Opportunity and Caveats

Copying large buffers between user process and kernel can be expensive. Linux supports various interfaces that eschew copying, such as sendpage 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.

15.1.2 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://>

lkml.kernel.org/netdev/20170803202945.70750-1-willemdebruijn.kernel@gmail.com

15.2 Interface

Passing the MSG_ZEROCOPY flag is the most obvious step to enable copy avoidance, but not the only one.

15.2.1 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");
```

15.2.2 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 zerocopy failure will return -1 with errno ENOBUFS. This happens if the socket option was not set, the socket exceeds its optmem limit or the user exceeds its 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.

15.2.3 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.

```
pfd.fd = fd;
pfd.events = 0;
if (poll(&pfd, 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 recvmmsg 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`.

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 zero-copy 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.

15.3 Implementation

15.3.1 Loopback

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.

15.4 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`.

FAILOVER**16.1 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*

17.1 Assumptions

This document assumes the reader has basic knowledge in network drivers and in general interrupt moderation.

17.2 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.

17.3 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.

17.4 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.

17.5 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;
```

(continues on next page)

(continued from previous page)

```

}

/* 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);
    ...
}

```

17.6 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

18.1 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.

18.2 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 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' />
  <address type='pci' domain='0x0000' bus='0x00' slot='0x0a'
  ↪function='0x0' />
</interface>
<interface type='hostdev' managed='yes'>
```

(continues on next page)

(continued from previous page)

```

<mac address='52:54:00:00:12:53' />
<source>
  <address type='pci' domain='0x0000' bus='0x42' slot='0x02'
↪function='0x5' />
</source>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x0b'
↪function='0x0' />
</interface>

```

Booting a VM with the above configuration will result in the following 3 netdevs 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,UP,LOWER_UP> mtu 1500 qdisc fq_
↪code1 master ens10 state UP 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

```

ens10 is the ‘failover’ master netdev, ens10nsby and ens11 are the slave ‘standby’ and ‘primary’ netdevs respectively.

18.3 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 on the source hypervisor.

```

# 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>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x0b'
↪

```

(continues on next page)

(continued from previous page)

```
↪function='0x0' />
</interface>

# Source Hypervisor
#!/bin/bash

DOMAIN=fedora27-tap01
PF=enp66s0f0
VF_NUM=5
TAP_IF=tap01
VF_XML=

MAC=52:54:00:00:12:53
ZERO_MAC=00:00:00:00:00:00

virsh domif-setlink $DOMAIN $TAP_IF up
bridge fdb del $MAC dev $PF master
virsh detach-device $DOMAIN $VF_XML
ip link set $PF vf $VF_NUM mac $ZERO_MAC

virsh migrate --live $DOMAIN qemu+ssh://$REMOTE_HOST/system

# Destination Hypervisor
#!/bin/bash

virsh attach-device $DOMAIN $VF_XML
virsh domif-setlink $DOMAIN $TAP_IF down
```


PAGE POOL API

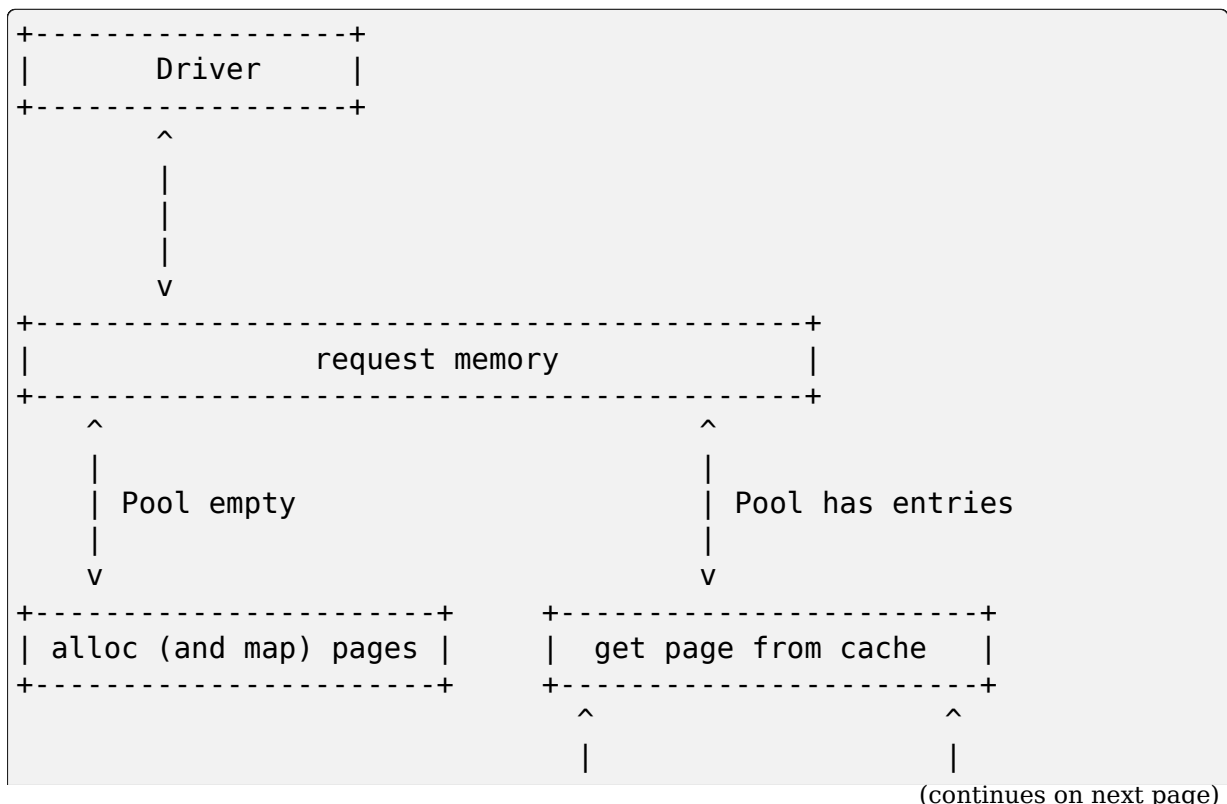
The `page_pool` allocator is optimized for the XDP mode that uses one frame per-page, but it can fallback on the regular page allocator APIs.

Basic use involves replacing `alloc_pages()` calls with the `page_pool_alloc_pages()` call. Drivers should use `page_pool_dev_alloc_pages()` replacing `dev_alloc_pages()`.

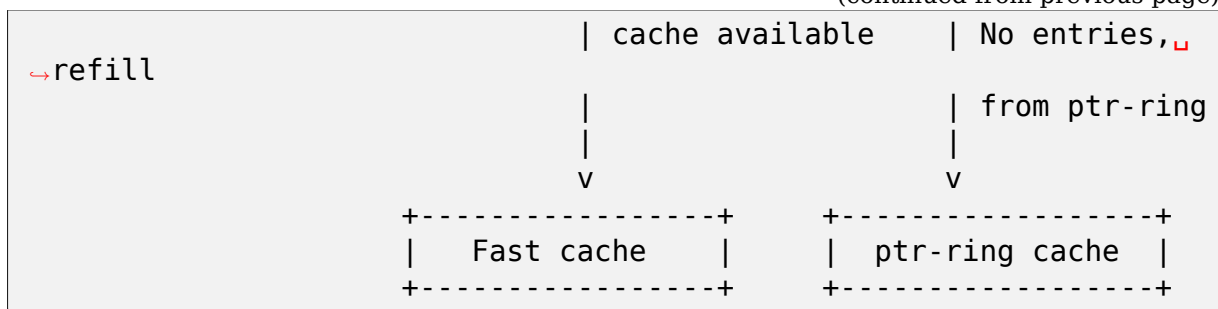
API keeps track of inflight pages, in order to let API user know when it is safe to free a `page_pool` object. Thus, API users must run `page_pool_release_page()` when a page is leaving the `page_pool` or call `page_pool_put_page()` where appropriate in order to maintain correct accounting.

API user must call `page_pool_put_page()` once on a page, as it will either recycle the page, or in case of `refcnt > 1`, it will release the DMA mapping and inflight state accounting.

19.1 Architecture overview



(continued from previous page)



19.2 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.

- **page_pool_create(): Create a pool.**

- flags: PP_FLAG_DMA_MAP, PP_FLAG_DMA_SYNC_DEV
- order: 2^{order} pages on allocation
- pool_size: size of the ptr_ring
- nid: preferred NUMA node for allocation
- dev: struct device. Used on DMA operations
- dma_dir: DMA direction
- max_len: max DMA sync memory size
- offset: DMA address offset

- page_pool_put_page(): 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().
- page_pool_put_full_page(): Similar to page_pool_put_page(), but will DMA sync for the entire memory area configured in area pool->max_len.
- page_pool_recycle_direct(): 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.
- page_pool_release_page(): Unmap the page (if mapped) and account for it on inflight counters.
- page_pool_dev_alloc_pages(): Get a page from the page allocator or page_pool caches.
- page_pool_get_dma_addr(): Retrieve the stored DMA address.

- `page_pool_get_dma_dir()`: Retrieve the stored DMA direction.

19.3 Coding examples

19.3.1 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.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;
```

19.3.2 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) {
        page_pool_release_page(page_pool, page);
        new_page = page_pool_dev_alloc_pages(page_pool);
    }
}
```

19.3.3 Driver unload

```
/* Driver unload */  
page_pool_put_full_page(page_pool, page, false);  
xdp_rxq_info_unreg(&xdp_rxq);
```

PHY ABSTRACTION LAYER

20.1 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.

20.2 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/”)

20.3 (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 operate 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.

20.3.1 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 10/100Mbps/sec makes the problem go away (since there is enough setup/hold time in that case)

20.4 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

20.5 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_IGNORE_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.

20.6 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_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_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.

20.7 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_ADVERTISE` 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.

20.8 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.

20.9 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.

20.10 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.

20.10.1 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.

20.10.2 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     = 0x0ffffff0,
    ...
}
```

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.

20.11 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);
```

20.12 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

21.1 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.

21.2 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.

21.3 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
<code>phy_start(phydev)</code>	<code>phylink_start(priv->phylink)</code>
<code>phy_stop(phydev)</code>	<code>phylink_stop(priv->phylink)</code>
<code>phy_mii_ioctl(phydev, ifr, cmd)</code>	<code>phylink_mii_ioctl(priv->phylink, ifr, cmd)</code>
<code>phy_ethtool_get_wol(phydev, wol)</code>	<code>phylink_ethtool_get_wol(priv->phylink, wol)</code>
<code>phy_ethtool_set_wol(phydev, wol)</code>	<code>phylink_ethtool_set_wol(priv->phylink, wol)</code>
<code>phy_disconnect(phydev)</code>	<code>phylink_disconnect_phy(priv->phylink)</code>

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 `of_phy_attach()` 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 `validate()` method should mask the supplied supported mask, and state->advertising with the supported ethtool link modes. These are the new ethtool link modes, so bitmask operations must be used. For an example, see `drivers/net/ethernet/marvell/mvneta.c`.

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/sfp, sfp.txt`

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.

22.1 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.

22.2 Alias deletion

The alias is removed by shutting the alias down:

```
# ifconfig eth0:0 down  
~~~~~ -> will delete alias
```

22.3 Alias (re-)configuring

Aliases are not real devices, but programs should be able to configure and refer to them as usual (`ifconfig`, `route`, etc).

22.4 Relationship with main device

If the base device is shut down the added aliases will be deleted too.

ETHERNET BRIDGING

In order to use the Ethernet bridging functionality, you'll need the userspace tools.

Documentation for Linux bridging is on:

<http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge>

The bridge-utilities are maintained at:

<git://git.kernel.org/pub/scm/linux/kernel/git/shemminger/bridge-utils.git>

Additionally, the iproute2 utilities can be used to configure bridge devices.

If you still have questions, don't hesitate to post to the mailing list (more info <https://lists.linux-foundation.org/mailman/listinfo/bridge>).

SNMP COUNTER

This document explains the meaning of SNMP counters.

24.1 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.

24.2 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](#)

`IcmpInSrcQuenchs`: [RFC1213 icmpInSrcQuenchs](#)

`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](#)

`IcmpOutSrcQuenchs`: [RFC1213 icmpOutSrcQuenchs](#)

`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.

24.2.1 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`.

24.3 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.

- `TcpExtTCPRcvCoalesce`

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 `TcpExtTCPRcvCoalesce`.

- `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.

24.4 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.

24.5 TCP Fast Path

When kernel receives a TCP packet, it has two paths to handler 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, `TcpExtTCPPureAcks` 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.

24.6 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 imme-

diately. 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.

24.7 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 `TcpExtTCPHystartTrainDetect` 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 `TcpExtTCPHystartDelayDetect` is the average CWND which detected by the packet delay.

24.8 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-retransmitted 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.

- `TcpExtTCPSynRetrans`

The TCP stack retransmits a SYN packet.

24.9 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.

24.10 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 the [Add counters for discarded SACK blocks](#) patch has additional explanation:

- `TcpExtTCPSACKDiscard`

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.

24.11 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' .

- `TcpExtTCPSackShifted`

A `skb` is shifted

- `TcpExtTCPSackMerged`

A `skb` is merged

- `TcpExtTCPSackShiftFallback`

A `skb` should be shifted or merged, but the TCP stack doesn't do it for some reasons.

24.12 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`.

24.13 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.

24.14 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](#)).

24.15 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.

24.16 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.

24.17 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.

24.18 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.

24.19 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.

24.20 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`.

24.21 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’ .

24.22 examples

24.22.1 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 $20+16+48=84$.

24.22.2 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.

24.22.3 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 exmaple):

```
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
TcpExtTCPPHPacks      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
TcpExtTCPPHPHits      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 'TcpExtTCPPHPacks'. The first server-side nstat and the second server-side nstat had a difference too: the second server-side nstat had a TcpExtTCPPHPHits, 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
```

(continues on next page)

(continued from previous page)

```
↪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 ‘TcpExtTCPPureAcks’ .

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’ .

24.22.4 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.

24.22.5 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 orphan 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
```

24.22.6 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
```

24.22.7 TcpExtTCPRcvCoalesce

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
TcpExtTCPRcvCoalesce   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 `TcpExtTCPRcvCoalesce` increased 1.

24.22.8 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.

24.22.9 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`.

24.22.10 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 -lkv 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 repeatly to nstat-b:

```
nstatuser@nstat-a:~$ for i in {1..3}; do sudo tcpreplay -i ens3 /
↳tmp/syn_fixcsum.pcap; done
```

Check snmp cunter on nstat-b:

```
nstatuser@nstat-b:~$ nstat | grep -i skip
TcpExtTCPACKSkippedSynRecv      1              0.0
```

As we expected, TcpExtTCPACKSkippedSynRecv is 1.

24.22.11 TcpExtTCPACKSkippedPAWS

To trigger PAWS, we could send an old SYN.

On nstat-b, let nc listen on port 9000:

```
nstatuser@nstat-b:~$ nc -lkv 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 TcpExtTCPACKSkippedPAWS.

24.22.12 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 -lkv 9000
Listening on [0.0.0.0] (family 0, port 9000)

nstatuser@nstat-b:~$ nc -lkv 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

25.1 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

25.2 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)`.

25.3 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).

25.4 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

26.1 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

26.2 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.

26.3 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.

26.4 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	<code>skb_mac_header</code>	
Network	<code>skb_network_header</code>	<code>skb_inner_network_header</code>
Transport	<code>skb_transport_header</code>	

UDP/GRE Tunnel:

	Outer	Inner
MAC	<code>skb_mac_header</code>	<code>skb_inner_mac_header</code>
Network	<code>skb_network_header</code>	<code>skb_inner_network_header</code>
Transport	<code>skb_transport_header</code>	<code>skb_inner_transport_header</code>

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.

26.5 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.

26.6 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.

26.7 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.

26.8 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

27.1 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

27.2 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 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`).

27.2.1 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 hyperthreading (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.

27.3 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.

27.3.1 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.

27.3.2 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.

27.4 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()`, `inet_sendpage()` 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 `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.

27.4.1 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.

27.5 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_steering` 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.

27.5.1 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.

27.6 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.

27.6.1 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.

27.7 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.

27.8 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 (willemb@google.com)

KERNEL TLS

28.1 Overview

Transport Layer Security (TLS) is a Upper Layer Protocol (ULP) that runs over TCP. TLS provides end-to-end data integrity and confidentiality.

28.2 User interface

28.2.1 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;
```

(continues on next page)

(continued from previous page)

```
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.

28.2.2 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 (2¹⁴).

```
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.

28.2.3 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.

28.2.4 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[MSG_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;
```

(continues on next page)

(continued from previous page)

```
msg.msg_iovlen = 1;

return sendmsg(sock, &msg, 0);
}
```

Control message data should be provided unencrypted, and will be encrypted by the kernel.

28.2.5 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[MSG_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.

28.2.6 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.

28.3 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

KERNEL TLS OFFLOAD

29.1 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 [Kernel TLS](#).

`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.

29.1.1 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.

29.1.2 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.

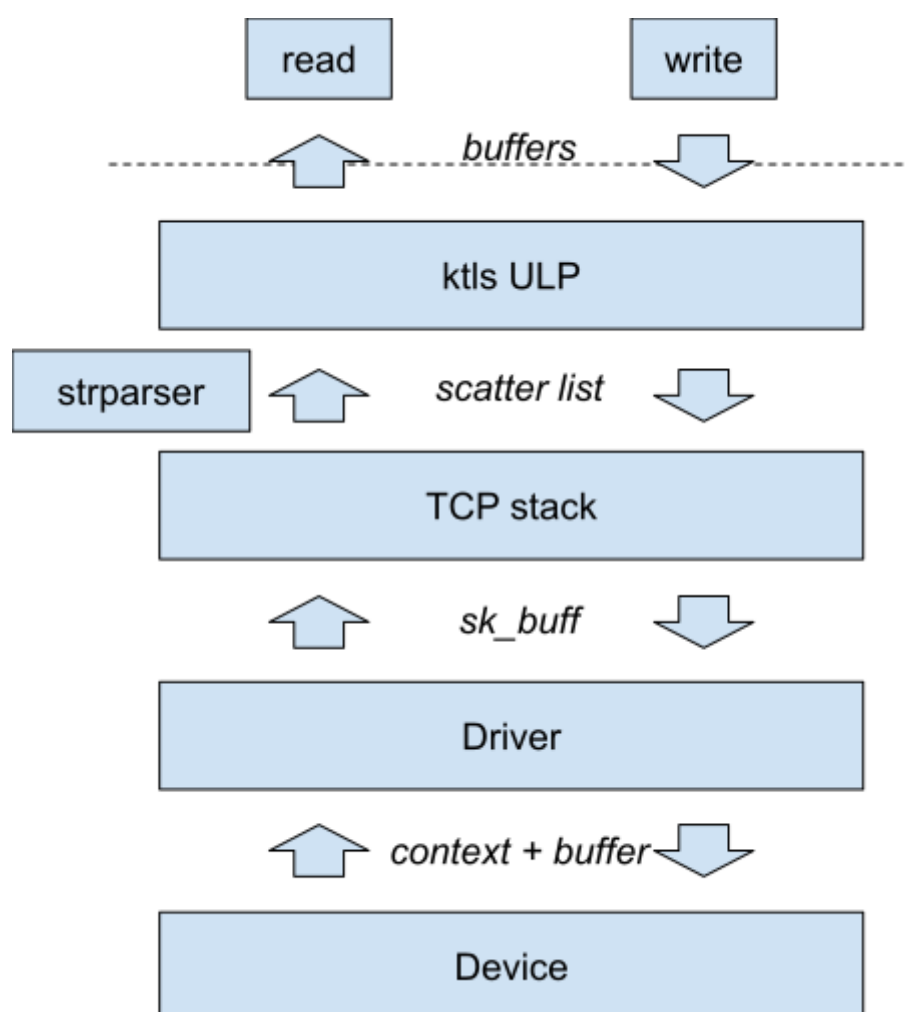


Fig. 1: Layers of Kernel TLS stack

29.2 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.

29.2.1 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.

29.2.2 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.

29.3 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.

29.3.1 TX

The kernel stack performs record framing reserving space for the authentication tag and populating all other TLS header and trailer 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.

29.3.2 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.

29.4 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.

29.4.1 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 fallback 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).

29.4.2 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).

29.5 Error handling

29.5.1 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.

29.5.2 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.

29.6 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.

29.6.1 Max connection count

The number of connections device can support can be exposed via devlink resource API.

29.6.2 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.

29.7 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.

29.8 Notable corner cases, exceptions and additional requirements

29.8.1 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).

29.8.2 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.

29.8.3 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.

29.8.4 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.

29.8.5 Transport layer transparency

The device should not modify any packet headers for the purpose of the simplifying TLS offload.

The device should not depend on any packet headers beyond what is strictly necessary for TLS offload.

29.8.6 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.

29.8.7 TLS device features

Drivers should ignore the changes to TLS the 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.

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

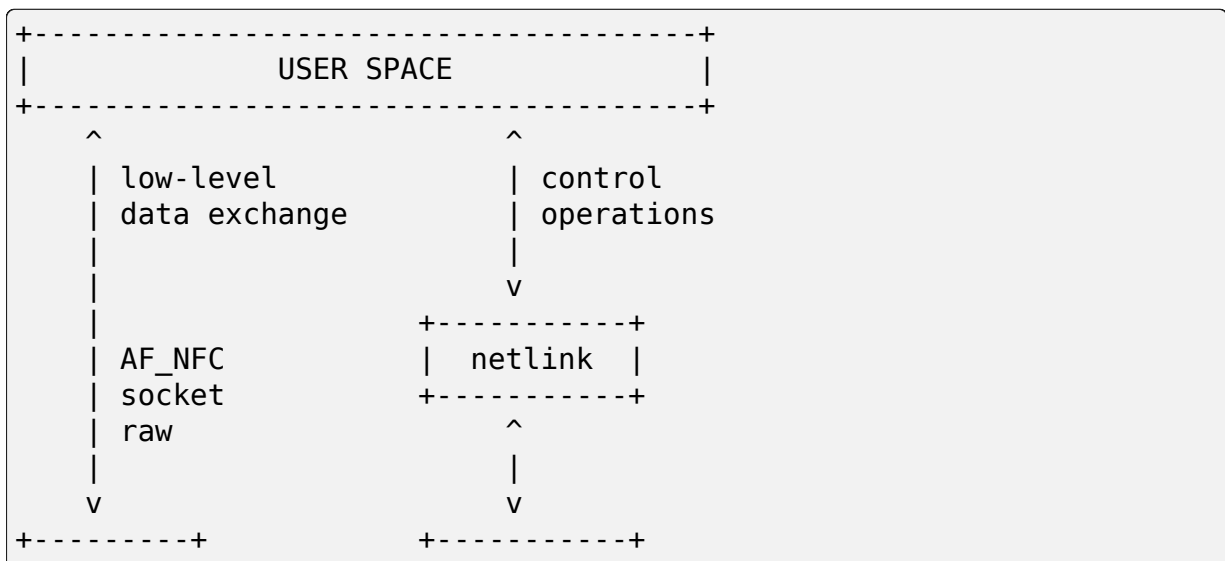
The NFC subsystem is responsible for:

- 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.

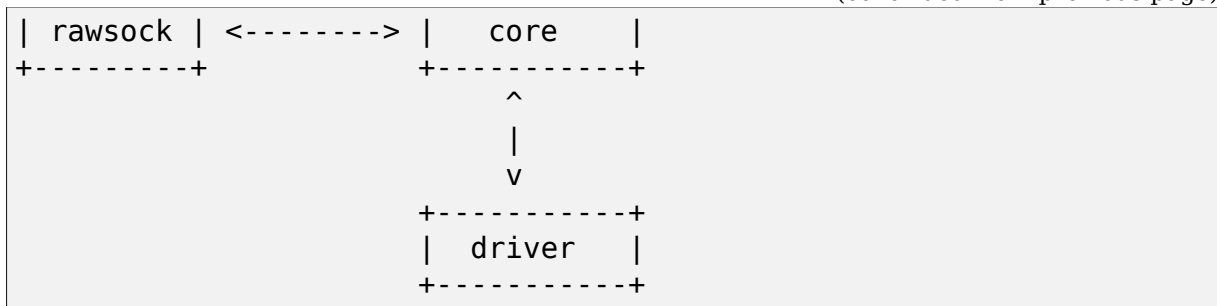
The control operations are available to userspace via generic netlink.

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.



(continues on next page)

(continued from previous page)



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_SOCKETPROTO_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_LLTTYPE_FOOBAR);
```

wheres LOWPAN_LLTTYPE_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_LLTTYPE_FOOBAR:
    /* do 802.15.4 6LoWPAN handling here */
    lowpan_foobar_priv(dev)->bar = foo;
    break;
...
    }
}
```

(continues on next page)

(continued from previous page)

```
        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.

6PACK PROTOCOL

This is the 6pack-mini-HOWTO, written by
Andreas Könsgen 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 be 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 insmodding the 6pack module and spattach-ing. 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.txt`. 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>

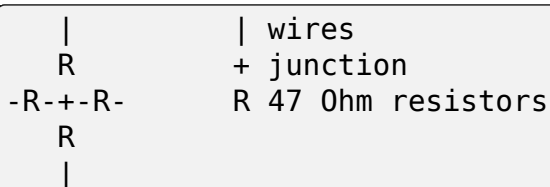
and Avery touched it up a bit, at Vojtech's request.

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:

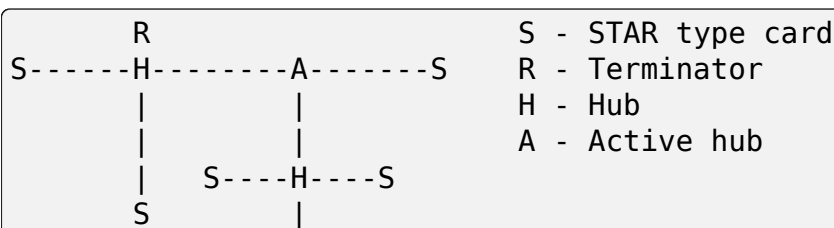
1. A card to a card. This is the simplest way of creating a 2-computer network.
2. A card to a passive hub. Remember that all unused connectors on the hub must be properly terminated with 93 Ohm (or something else if you don't have the right ones) terminators.

(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:



(continues on next page)

(continued from previous page)

```

      |
      S

```

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:

```

RT-----T-----T-----T-----T-----TR
 B       B       B       B       B       B

```

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:

```

      A-----T-----T-----TR
      |       B       B       B
S---H---S
      |
      S

```

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:

```

S-----T-----T-----S
      B       B

```

But, according to my own experiments, you can simply hang a BUS type card anywhere in middle of a cable in a STAR topology network. And more - you can use the bus card in place of any star card if you use a terminator. Then you can build very complicated networks fulfilling all your needs! An example:

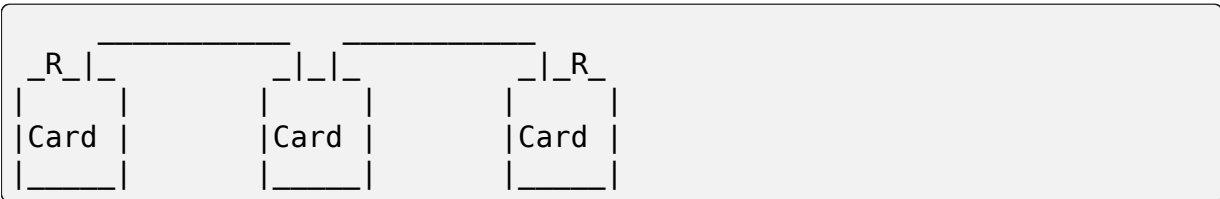
```

                        S
                        |
      RT-----T-----T-----H-----S
      B       B       B       |
S-----A-----T-----T-----A-----R-----TR
      |       B       B       |       |       B
      |       |       |       |       |
      |       S       |       BT      |
      |       |       |       |       |
S-----H-----A-----S       S-----A-----S
      |       |       |       |       |
      S       S       S       B       R       S       S

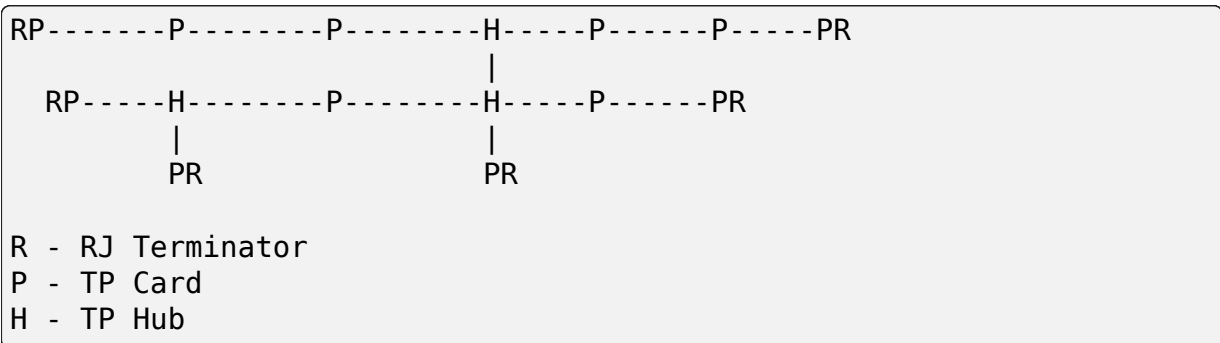
```

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:



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

(continues on next page)

(continued from previous page)

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)	[] : : : : IRQ2 IRQ3 IRQ4 IRQ5 IRQ7																																																																													
Put exactly one jumper on exactly one set of pins.																																																																														
S1 (I/O and Memory addresses)	<table border="0"> <tr> <td></td> <td>1</td> <td>2</td> <td>3</td> <td>4</td> <td>5</td> <td>6</td> <td>7</td> <td>8</td> <td>9</td> <td>10</td> </tr> <tr> <td></td> <td colspan="10">/-----\</td> </tr> <tr> <td></td> <td> </td> <td>1</td> <td>1</td> <td>*</td> <td>0</td> <td>0</td> <td>0</td> <td>*</td> <td>1</td> <td>1</td> </tr> <tr> <td></td> <td> </td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td></td> <td colspan="10">\-----/</td> </tr> <tr> <td></td> <td> </td> <td>--</td> <td> </td> <td colspan="4"> ----- </td> <td> </td> <td colspan="2"> ----- </td> </tr> <tr> <td></td> <td></td> <td>(a)</td> <td></td> <td>(b)</td> <td></td> <td></td> <td></td> <td></td> <td>(m)</td> <td></td> </tr> </table>		1	2	3	4	5	6	7	8	9	10		/-----\												1	1	*	0	0	0	*	1	1			0	0	0	0	0	0	0	0	0		\-----/												--		-----					-----				(a)		(b)					(m)	
	1	2	3	4	5	6	7	8	9	10																																																																				
	/-----\																																																																													
		1	1	*	0	0	0	*	1	1																																																																				
		0	0	0	0	0	0	0	0	0																																																																				
	\-----/																																																																													
		--		-----					-----																																																																					
		(a)		(b)					(m)																																																																					
↪which way	WARNING. It's very important when setting these.																																																																													
↪'1'!	you're holding the card, and which way you think is																																																																													
↪the	If you suspect that your settings are not being made correctly, try reversing the direction or inverting.																																																																													
	switch positions.																																																																													
	a: The first digit of the I/O address.																																																																													
	Setting Value																																																																													

(continues on next page)

(continued from previous page)

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

→ example, if

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!

S2
(Station Address)

	1	2	3	4	5	6	7	8
/	-----	-----	-----	-----	-----	-----	-----	-----
	1	1	0	0	0	0	0	0
\	-----	-----	-----	-----	-----	-----	-----	-----

Setting	Value
-----	-----
00000000	00
10000000	01
01000000	02
...	...
01111111	FE

(continues on next page)

(continued from previous page)

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>

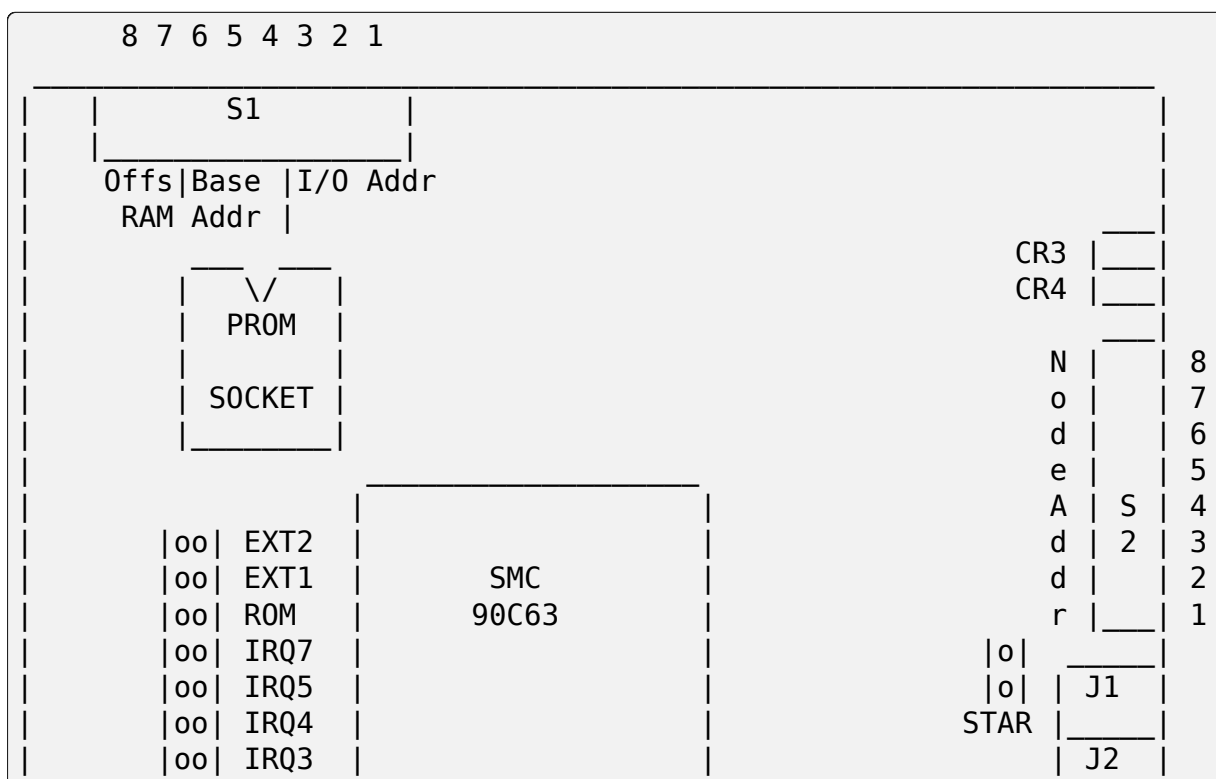
This description has been written by Juergen Seifert <seifert@htwm.de> using information from the following Original SMC Manual

“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

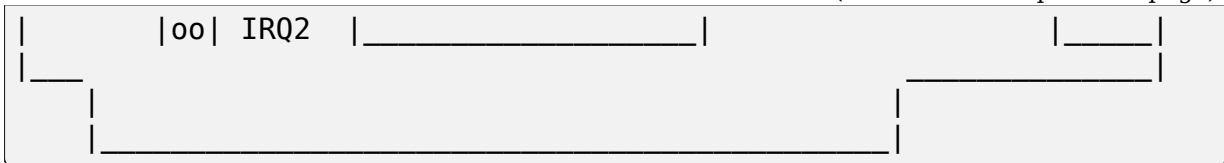
The PC130E is an enhanced version of the PC130 board, is equipped with a standard BNC female connector for connection to RG-62/U coax cable. Since this board is designed both for point-to-point connection in star networks and for connection to bus networks, it is downwardly compatible with all the other 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.



(continues on next page)

(continued from previous page)



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
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 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.

(continues on next page)

(continued from previous page)

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		incorrect memory or
	node ID is zero		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!]

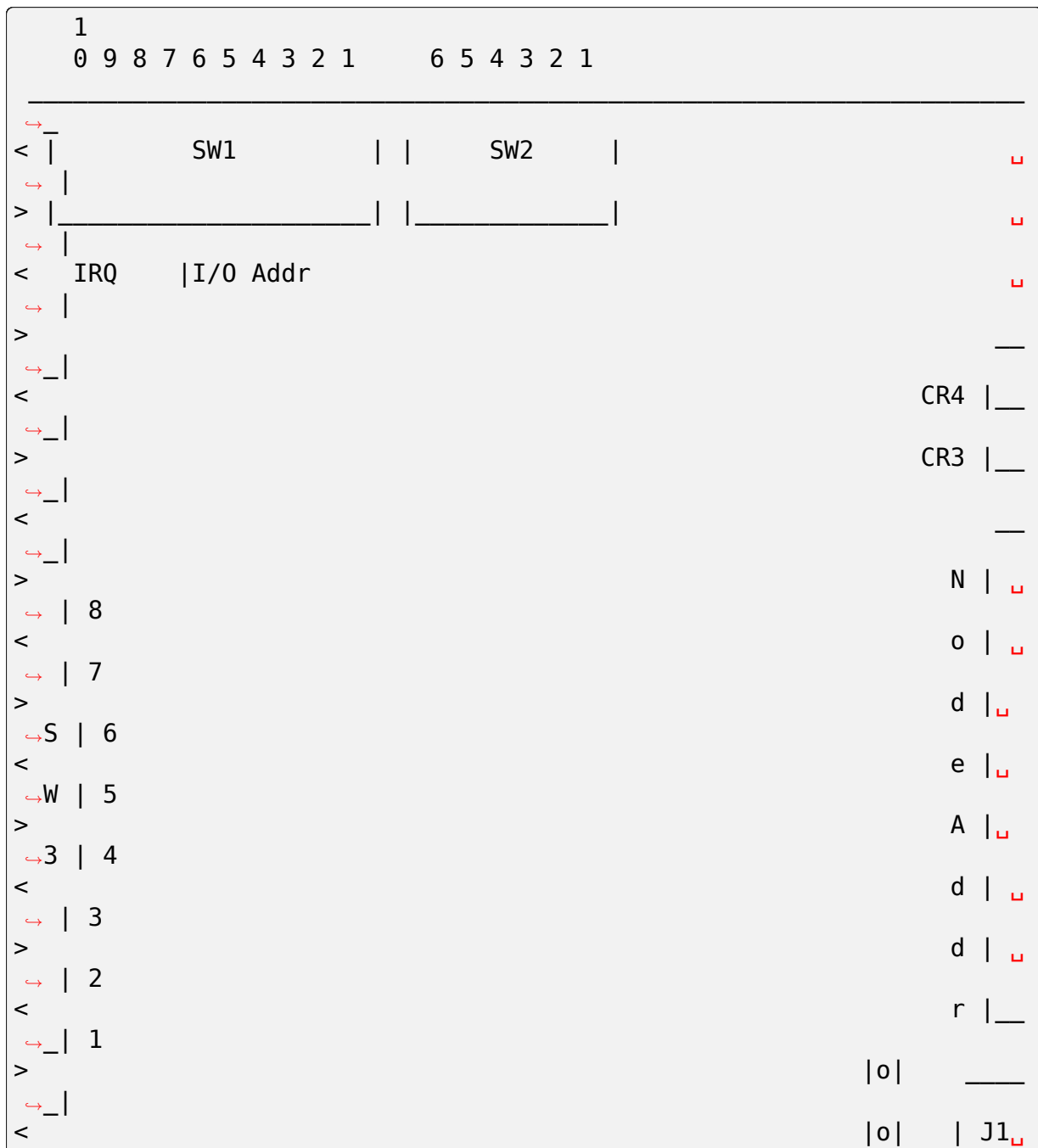
This description has been written by Juergen Seifert <seifert@htwm.de> using information from the following Original SMC Manual

“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.



(continues on next page)

(continued from previous page)



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								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
		.	.	.						
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						Hex I/O
6	5	4	3	2	1	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

(continues on next page)

(continued from previous page)

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:

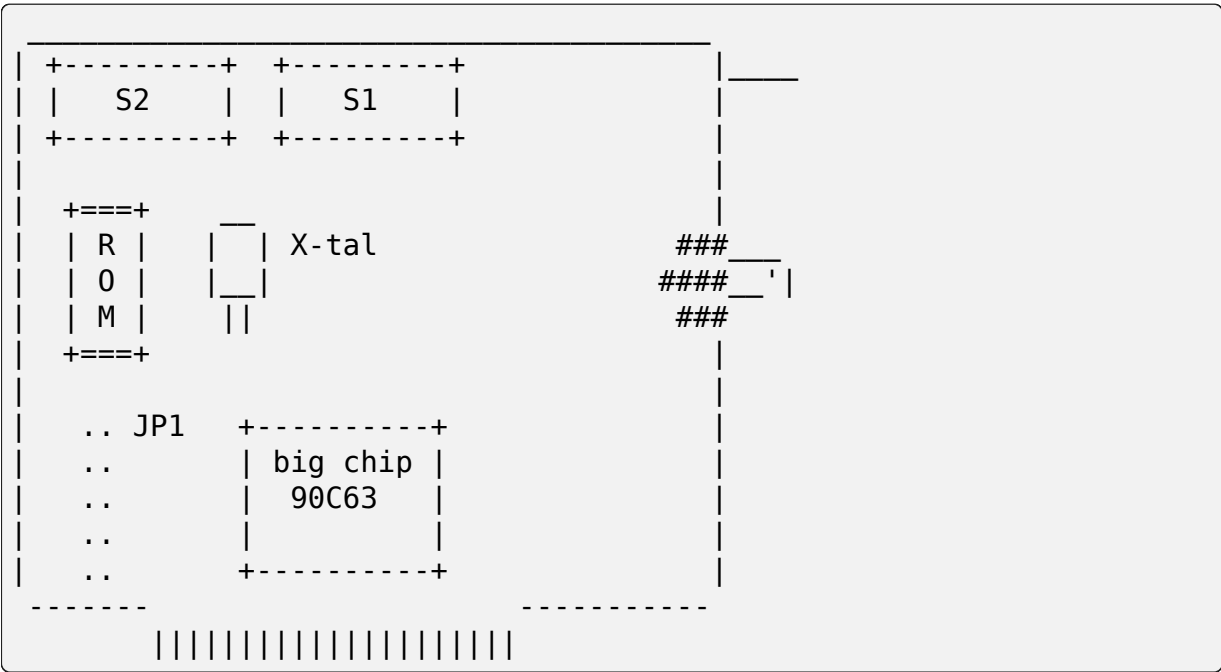
Green	Status	Red	Status
-----	-----	-----	-----
on	normal activity	flash/on	data transfer
blink	reconfiguration	off	no data transfer;
off	defective board or		incorrect memory or
	node ID is zero		I/O address

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.

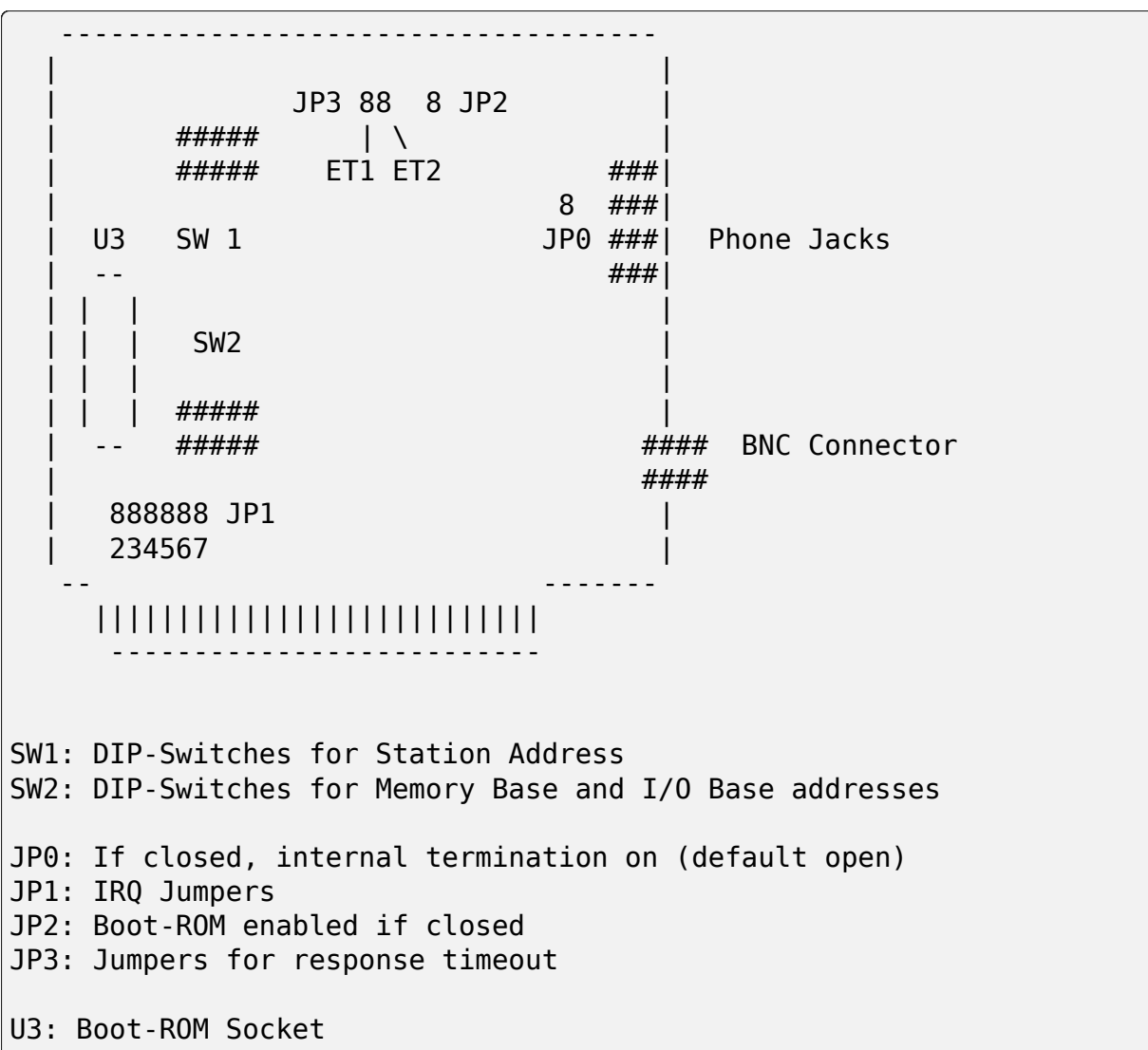
33.6 Possibly SMC

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:



(continues on next page)

(continued from previous page)

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 678	Base 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 12345	RAM Address Range	ROM 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	
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	

continues on next page

Table 1 - continued from previous page

Switches 12345	RAM Address Range	ROM Address Range
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:dfff	
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.

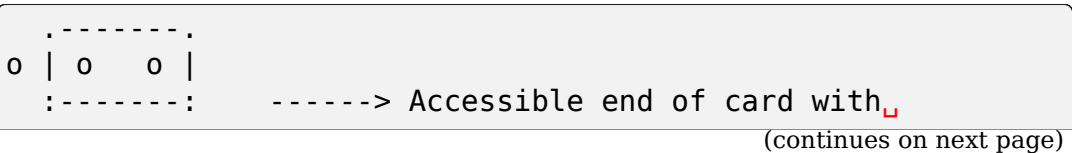
Jumpers:

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:



(continued from previous page)

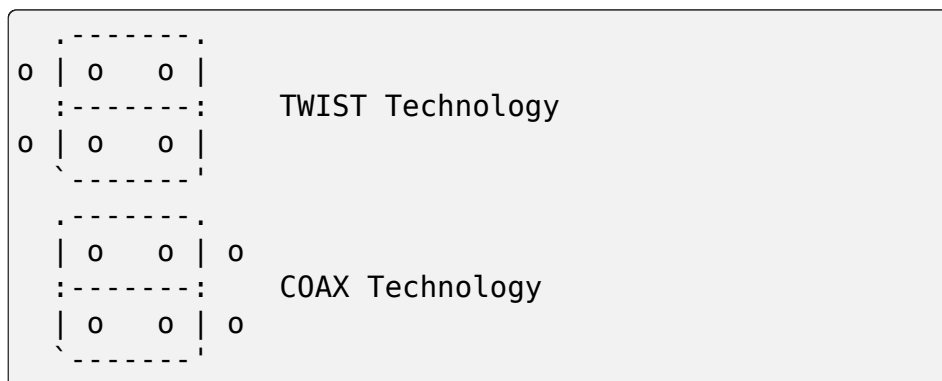
```

→ connectors
o | o  o |          in this direction ----->
  \-----/

```

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. 12345678	ARCnet address
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”

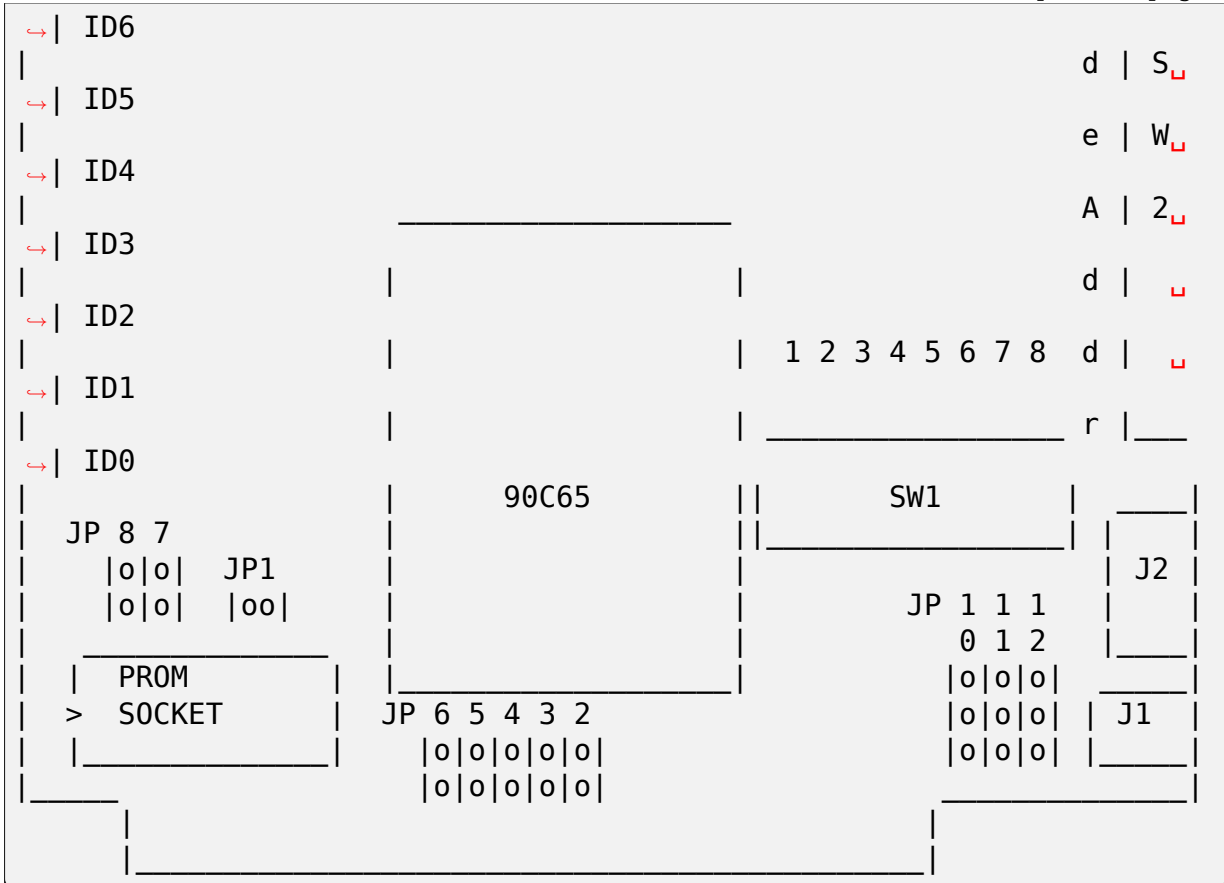
ARCNET is a registered trademark of the Datapoint Corporation

- 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



(continues on next page)

(continued from previous page)



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/ST/ST only)
JP12	Terminator Select (CN120AB/ST/ST only)
J1	BNC RG62/U Connector (all except CN120TP)
J2	Two 6-position Telephone Jack (CN120TP/ST/ST 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 a 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	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
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	Hex I/O
6 7 8	Address
-----	-----
ON ON ON	260
OFF ON ON	290
ON OFF ON	2E0 (Manufacturer's default)
OFF OFF ON	2F0
ON ON OFF	300

(continues on next page)

(continued from previous page)

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:

<div>0</div> <div>-----</div> <div> 0 </div> <div> 0 </div> <div> 0 </div> <div>-----</div> <div>Terminator disabled</div>	<div>ON</div> <div>OFF</div>	<div>-----</div> <div> 0 </div> <div> 0 </div> <div>-----</div> <div>0</div> <div>Terminator enabled</div>	<div>ON</div> <div>OFF</div>
--	------------------------------	--	------------------------------

Selecting the Connector Type on CN120ST/SBT

<div>JP10</div> <div>0</div> <div>-----</div> <div> 0 </div> <div> 0 </div> <div> 0 </div> <div>-----</div> <div>Coaxial Cable (Default)</div>	<div>JP11</div> <div>0</div> <div>-----</div> <div> 0 </div> <div> 0 </div> <div> 0 </div> <div>-----</div>	<div>JP10</div> <div>-----</div> <div> 0 </div> <div> 0 </div> <div>-----</div> <div>0</div> <div>Twisted Pair Cable</div>	<div>JP11</div> <div>-----</div> <div> 0 </div> <div> 0 </div> <div>-----</div> <div>0</div>
--	---	--	--

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>

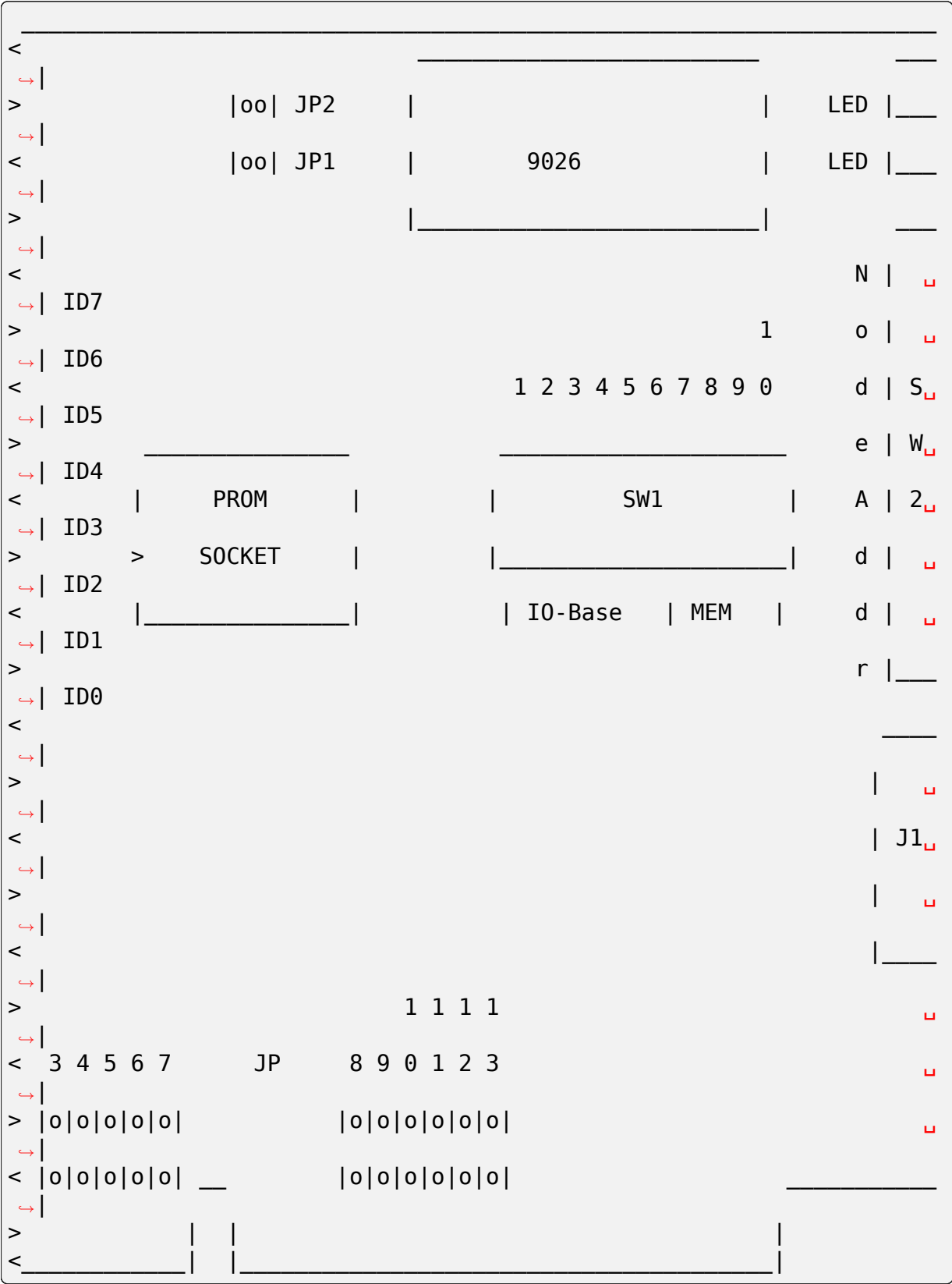
This description has been written by Juergen Seifert <seifert@htwm.de> using information from the following Original CNet Manual

“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

(continues on next page)

(continued from previous page)

```

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 a 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	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
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						Hex I/O
1	2	3	4	5	6	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				Hex RAM	Hex ROM
7	8	9	10	Address	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

(continues on next page)

(continued from previous page)

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

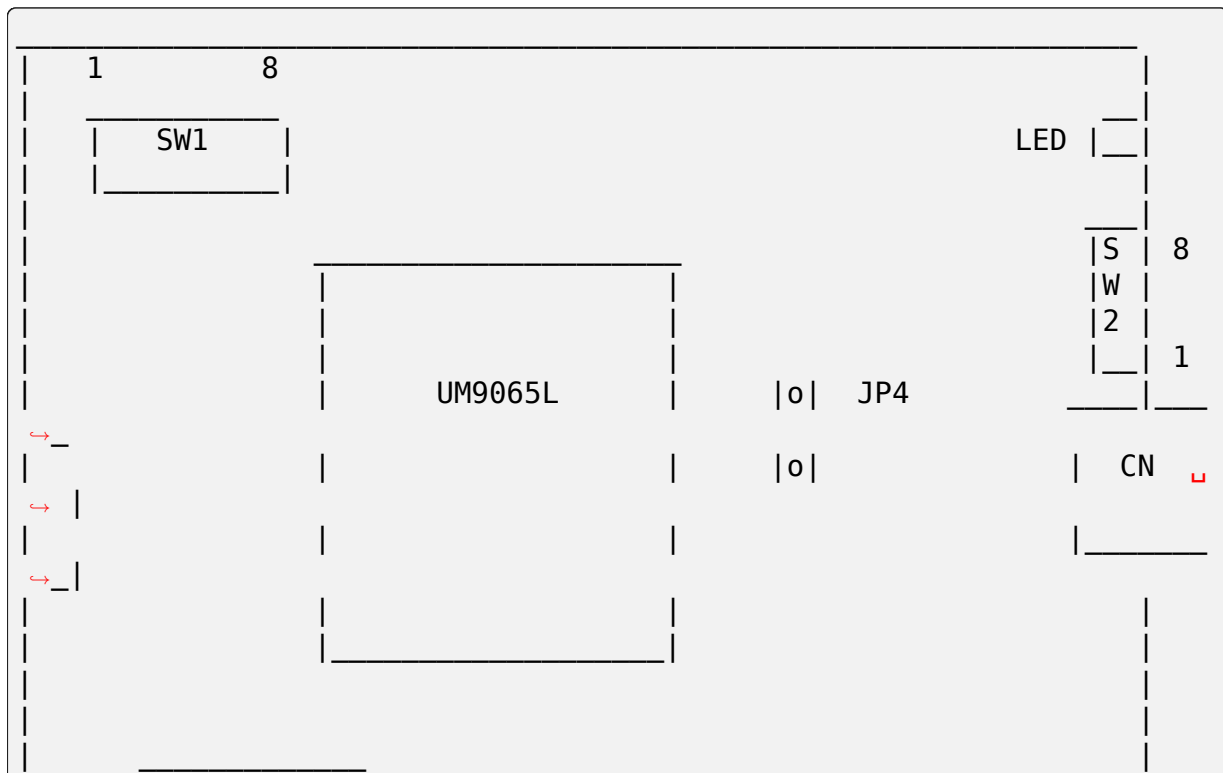
33.9.2 Setting the Timeout Parameters

The jumpers labeled JP1 and JP2 are used to determine the timeout parameters. These two jumpers are normally left open.

33.10 Lantech

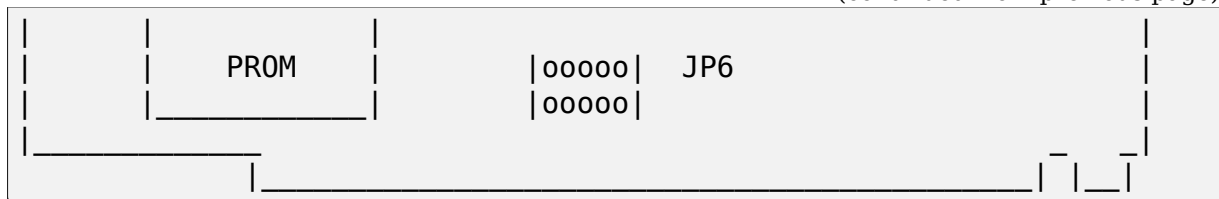
33.10.1 8-bit card, unknown model

- 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.



(continues on next page)

(continued from previous page)



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

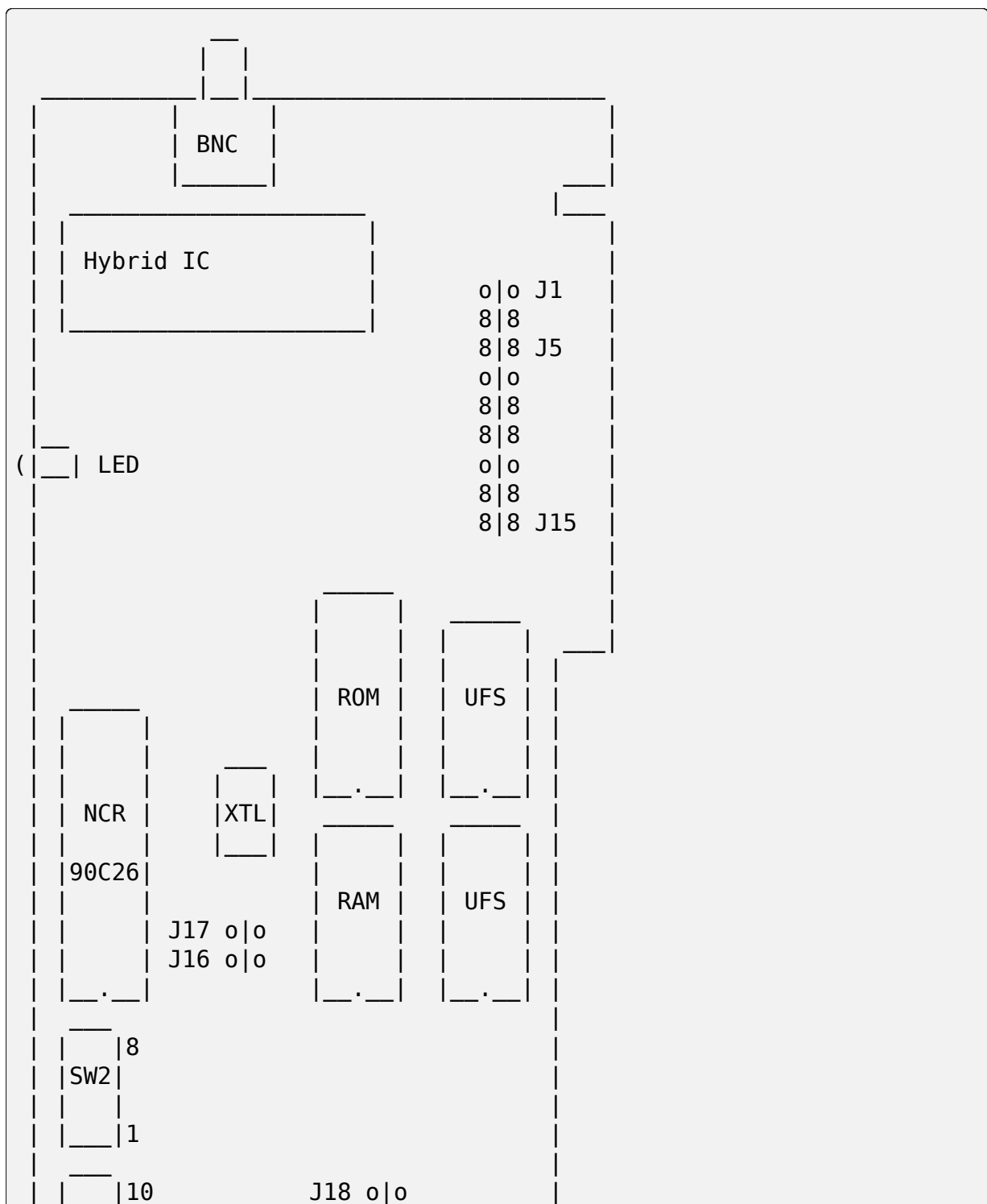
JP 6 : IRQ set (ONLY ONE jumper on 1-5 for IRQ 2-6)

33.11 Acer

33.11.1 8-bit card, Model 5210-003

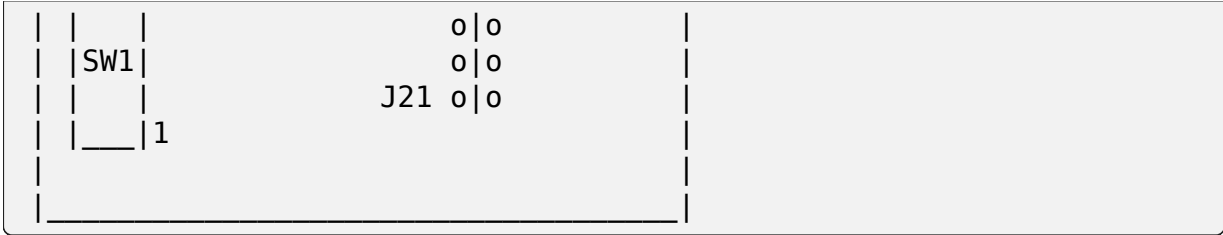
- from Vojtech Pavlik <vojtech@suse.cz> using portions of the existing arcnet-hardware file.

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.



(continues on next page)

(continued from previous page)



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 bellow 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	Hex RAM Address
7 8 9 10	-----
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

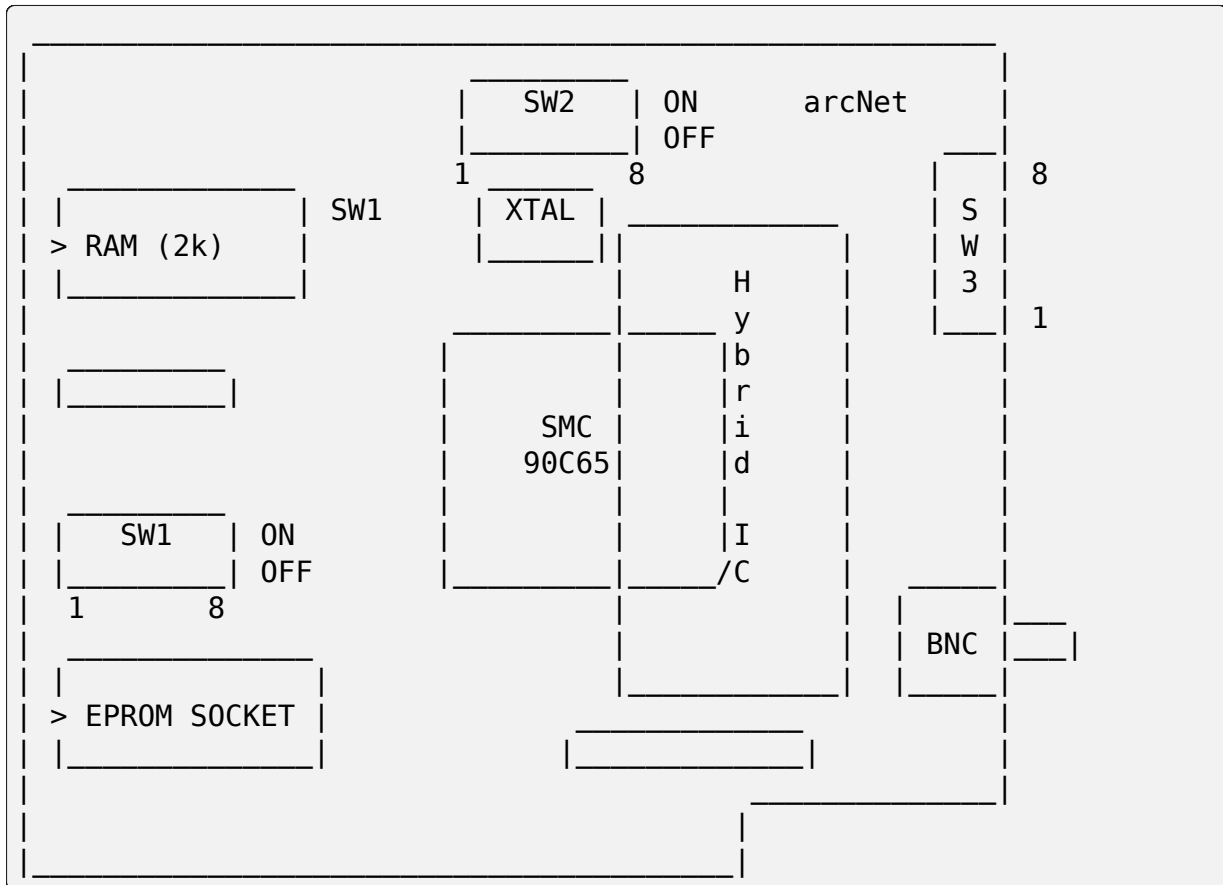
I know nothing about these. I just guess that J16&J17 are timeout jumpers and maybe one of J18-J21 selects ROM. Also J6-J10 and J11-J15 are connecting IRQ2-7 to some pins on the UFSs. I can't guess the purpose.

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

(continues on next page)

(continued from previous page)

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 a 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			Hex I/O
6	7	8	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 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

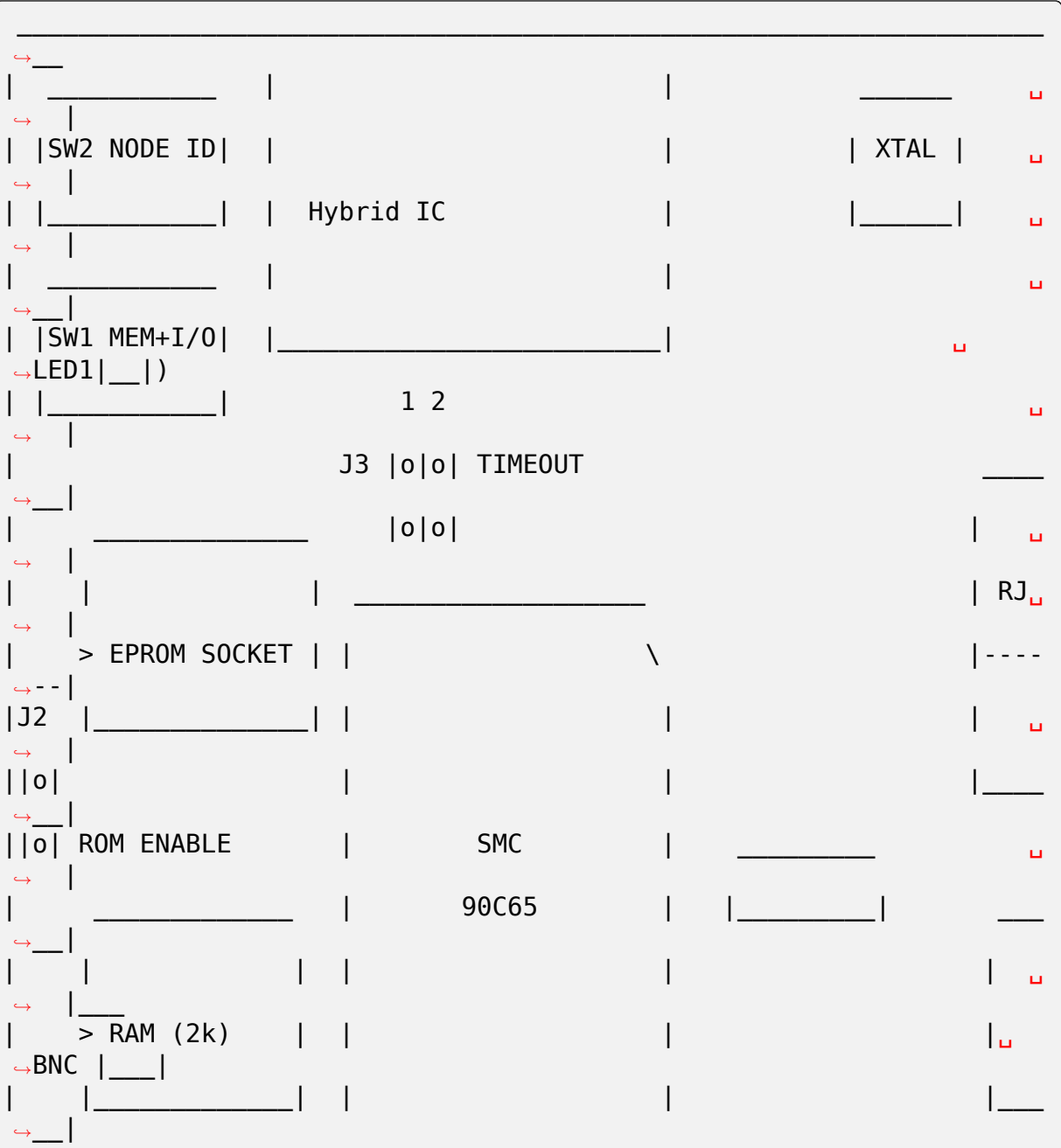
The switches 6-7 of the switch block SW3 are used to determine the timeout parameters. These two switches are normally left in the OFF position.

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.



(continues on next page)

(continued from previous page)



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			Hex I/O
6	7	8	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					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 (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	
-----					-----
ON	OFF	OFF	OFF	OFF	2
OFF	ON	OFF	OFF	OFF	3

(continues on next page)

(continued from previous page)

OFF	OFF	ON	OFF	OFF		4
OFF	OFF	OFF	ON	OFF		5
OFF	OFF	OFF	OFF	ON		7

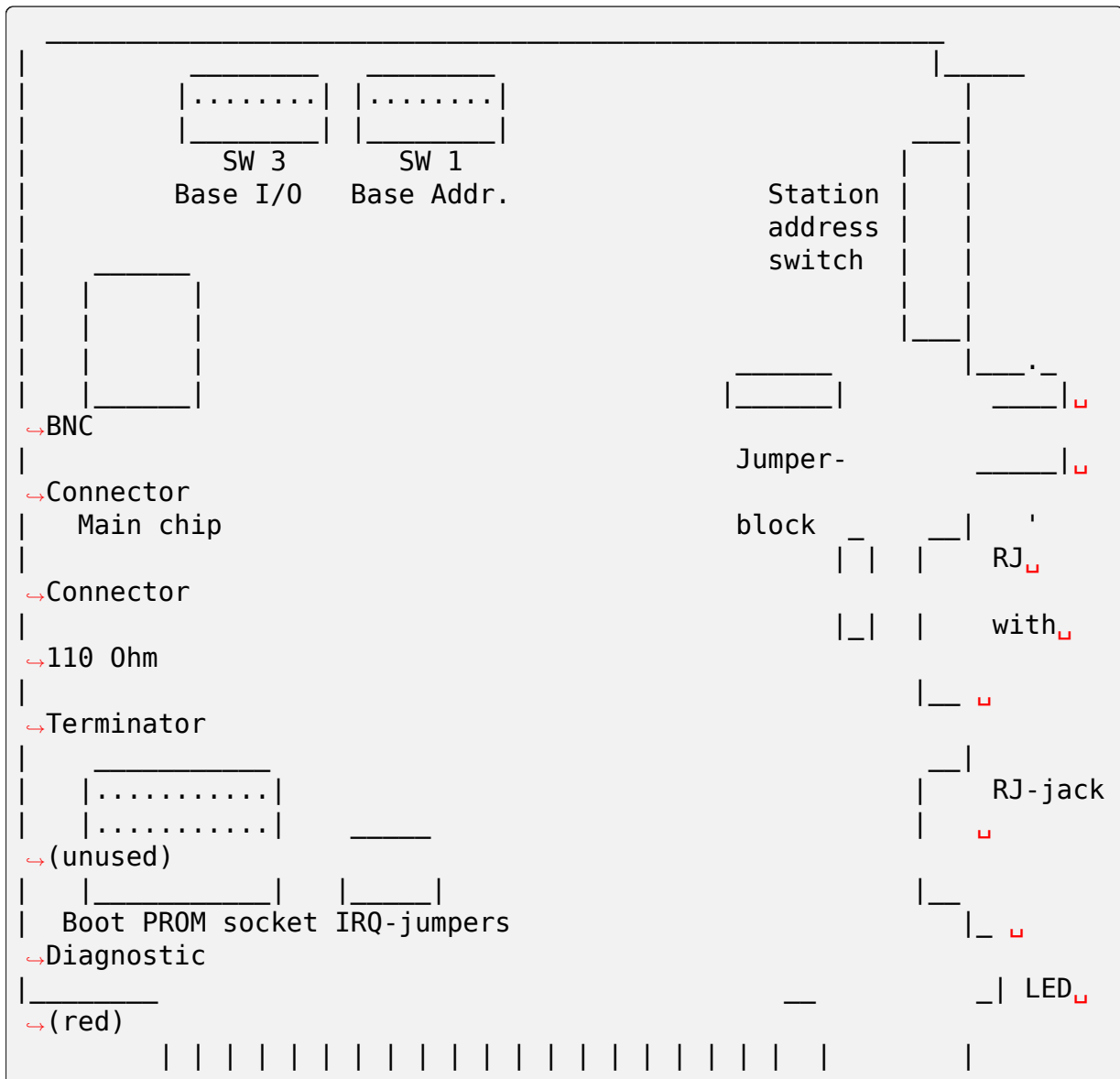
Setting the Timeout Parameters

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>



(continues on next page)

(continued from previous page)



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

```

```

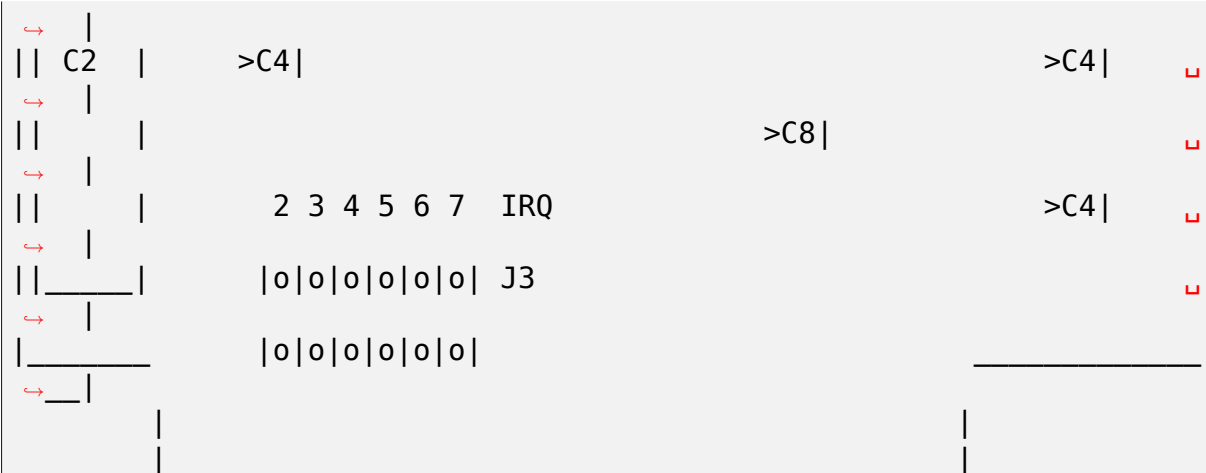
      IRQ
      3 4 5 7 2
      . . . . .
      . . . . .

```

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

(continued from previous page)



C1 -- "COM9026
SMC 8638"
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.

(continues on next page)

(continued from previous page)

```

C9 -- "PA116R4A-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

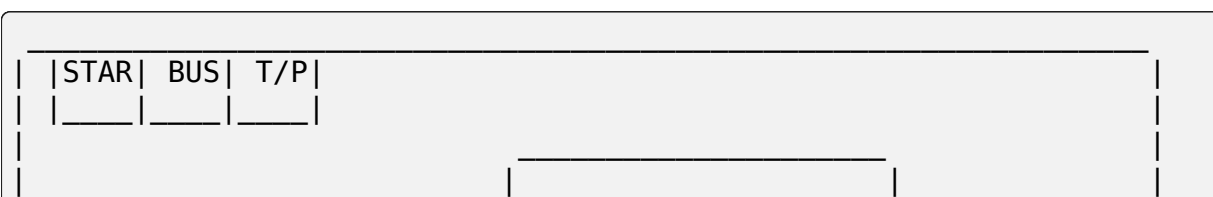
33.16.1 8-bit cards, 16-bit cards

- 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”

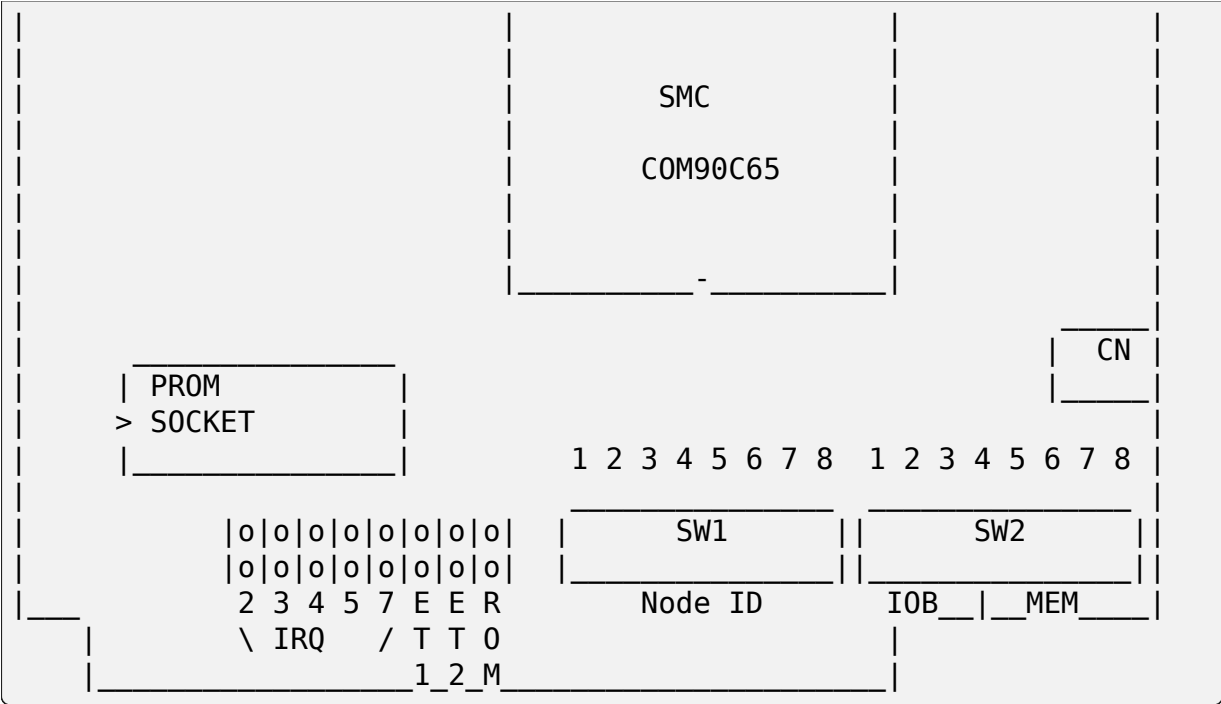
This description has been written by Juergen Seifert <seifert@htwm.de> using information from the Original

“ARCnet Installation Manual”



(continues on next page)

(continued from previous page)



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

(continues on next page)

(continued from previous page)

3		32
2		64
1		128

Some Examples:

Switch								Hex		Decimal	
1	2	3	4	5	6	7	8	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	
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			Hex I/O	
1	2	3	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					Hex RAM	Hex ROM	
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 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

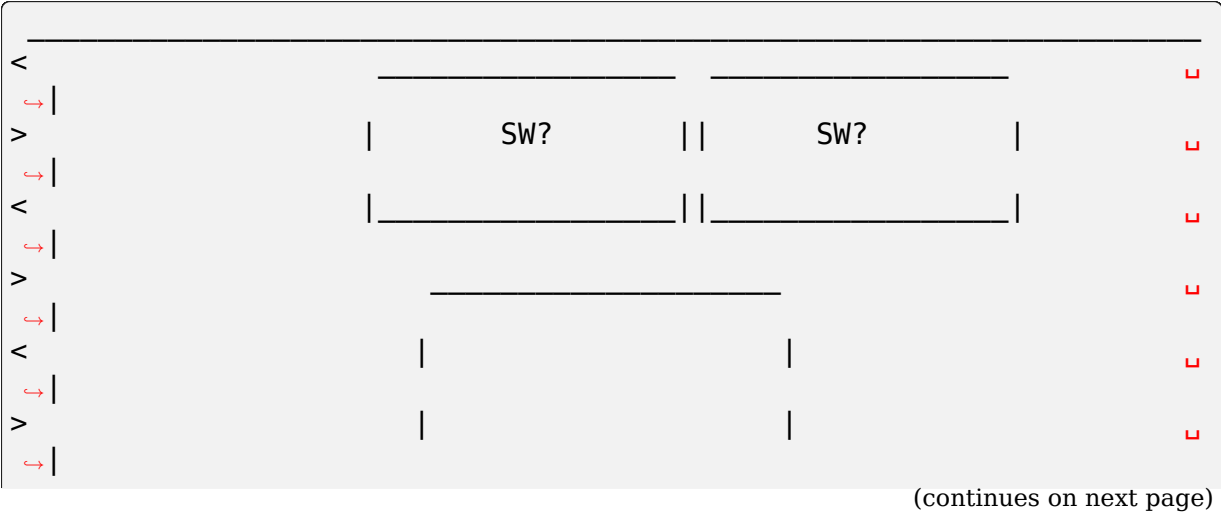
33.16.2 16-BIT ARCNET

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!

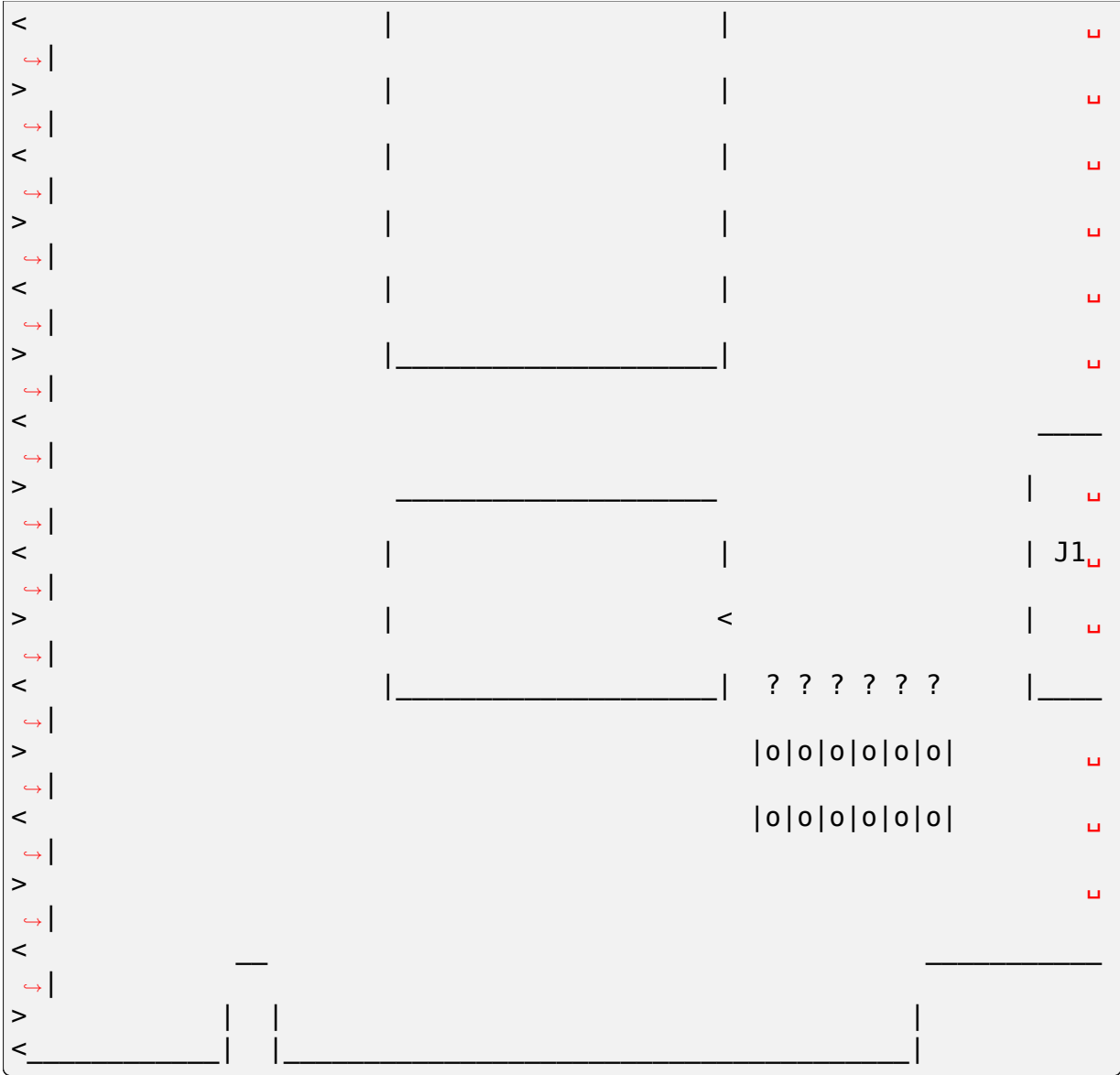
This description has been written by Juergen Seifert <seifert@htwm.de> using information from the Original

“ARCnet Installation Manual”



(continues on next page)

(continued from previous page)



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

(continues on next page)

(continued from previous page)

4		16
3		32
2		64
1		128

Some Examples:

Switch	Hex	Decimal
1 2 3 4 5 6 7 8	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
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	Hex I/O
3 2 1	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	

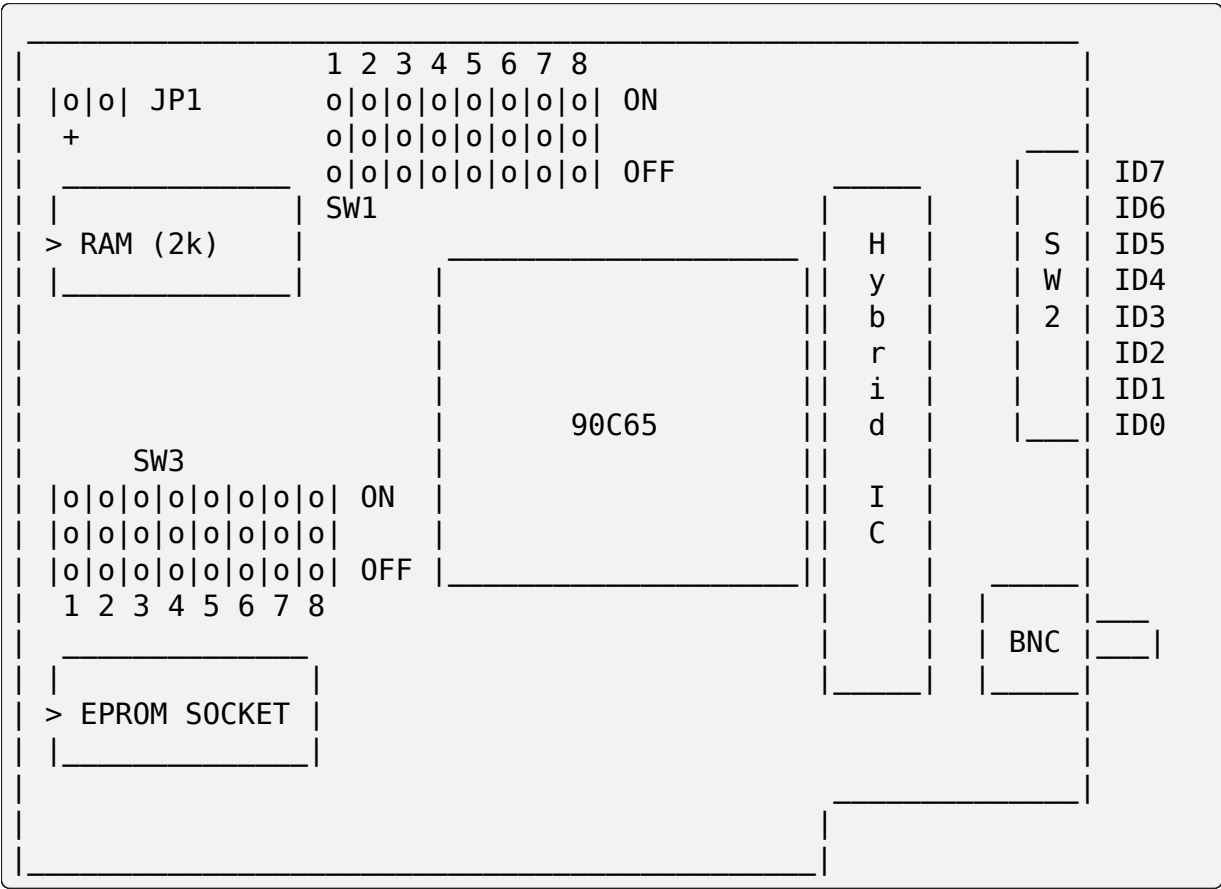
Setting Interrupt Request Lines (IRQ)

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								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
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 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	
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 1 2 3 4 5	IRQ
ON OFF OFF OFF OFF	2
OFF ON OFF OFF OFF	3

(continues on next page)

(continued from previous page)

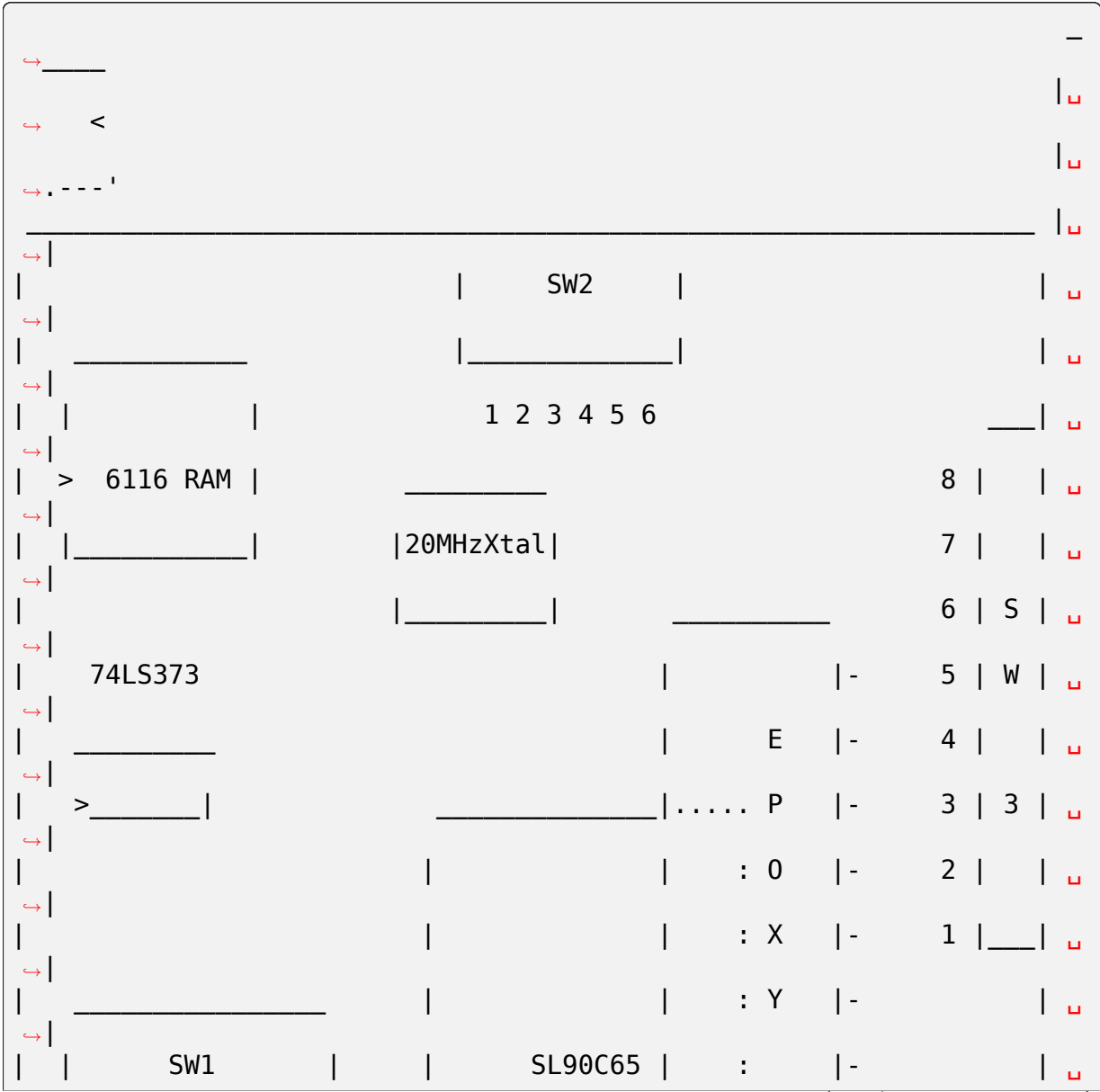
OFF	OFF	ON	OFF	OFF		4
OFF	OFF	OFF	ON	OFF		5
OFF	OFF	OFF	OFF	ON		7

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!)



(continues on next page)

[illegible]

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
4 5 6	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 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	Hex ROM
1 2 3	Address	Address
-----	-----	-----
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	
1 0 1 0 1 0 1 0	AA	170	+ Don't use 0 or 255!
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.

+-----+
(continues on next page)
```

(continued from previous page)

```
!           ! Transmitter Unit !           !
!           +-----+
!           MEM
!  ROM      7654321 <- I/O
!  :  :      +-----+
!  :  :      ! 90C66LJ!
!  :  :      !       !
!  :  :      !       !
!  :  :      +-----+
!
!           234567 <- IRQ
+-----+!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!+
!           !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

- 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)

123	Port
000	260
001	290
010	2E0
011	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.txt` 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 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: apenwarr@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 net-dev@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>.

Novell makes a networking stack for DOS which includes ARCnet drivers. Try FTPing to <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/pktdrvr. 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 following 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:

```
arccrimi=<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 compiled 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” :

- arcC 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 *only* this protocol. arcC 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.
- arcE “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. arcE also works slightly more slowly than arcC, for reasons yet to be determined. (Probably it’ s the smaller MTU that does it.)
- arcS 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 arcE. The arcS support was contributed by Tomasz Motylewski and modified somewhat by me. Bugs are probably my fault.

You can choose not to compile arcE and arcS 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 arcE and arcS devices are created automatically when you first ifconfig the arcC 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 arcC unless you have a good reason (like some other software, ie. WfWg, that only works with arcE).

If you need only arcC, 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):

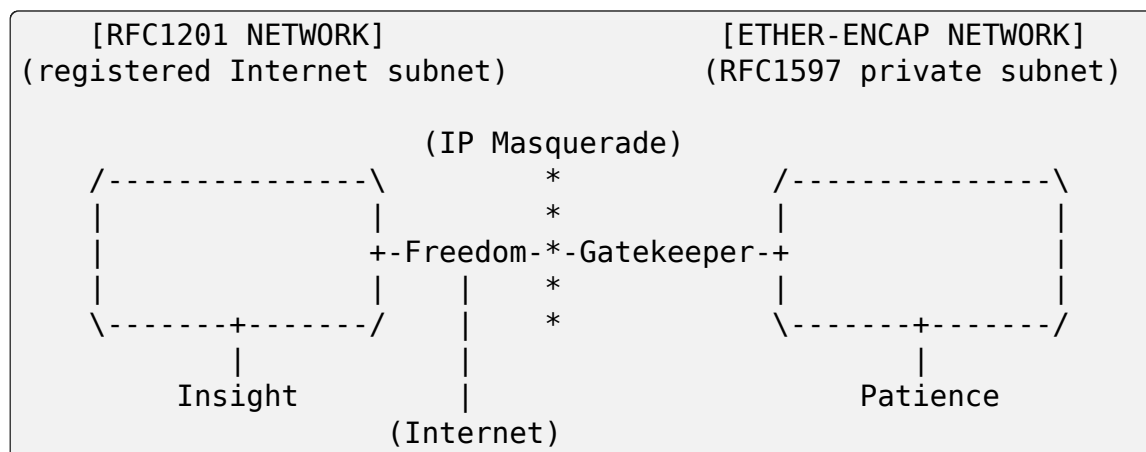
```
ifconfig arc0e gatekeeper
route add gatekeeper arc0e
route add patience arc0e
```

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 lxxx
/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.

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/>

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 <http://www.linux-ax25.org>.

There is an active mailing list for discussing Linux amateur radio matters called linux-hams@vger.kernel.org. To subscribe to it, send a message to major-domo@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 (LACPDUs). 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.

`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

`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 `downdelay` 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_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 LACPDU every 30 seconds

fast or 1

Request partner to transmit LACPDU 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 0.

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 mis-ordering 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 link monitor interval (`arp_interval` or `miimon`, whichever is active). The default value is 0 which means to match the value of the link monitor interval.

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 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

hash = source MAC XOR destination MAC XOR packet type ID
slave number = 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

hash = source MAC XOR destination MAC XOR packet type ID
hash = hash XOR source IP XOR destination IP
hash = hash XOR (hash RSHIFT 16)
hash = 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)
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 non-compliance.

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.

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. Distro's 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 `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'
```

(continues on next page)

(continued from previous page)

```
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:

on-	The device is started at boot time. If you're not sure, this is probably what you want.
man	The device is started only when ifup is called manually. Bonding
ual	devices may be configured this way if you do not wish them to
	start automatically at boot for some reason.
hot-	The device is started by a hotplug event. This is not a valid choice
plug	for a bonding device.
off	The device configuration is ignored.
or	
ig-	
nor€	

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_SLAVEn=` “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 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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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 desirable 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_u
↪ \
    dst 192.168.1.100 action skbedit 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 hosts 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 )) \
                        $(( RANDOM & 0xFF )) \
                        $(( RANDOM & 0xFF )) \
                        $(( RANDOM & 0xFF )) \
                        $(( 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
```

(continues on next page)

(continued from previous page)

```
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.

The ARP monitor relies on the device driver itself to verify that traffic is flowing. In particular, the driver must keep up to date the last receive time, `dev->last_rx`. Drivers that use `NETIF_F_LLTX` flag must also update `netdev_queue->trans_start`. If they do not, then the ARP monitor will immediately fail any slaves using that driver, and those slaves will stay down. If networking monitoring (tcpdump, etc) shows the ARP requests and replies on the network, then it may be that your device driver is not updating `last_rx` and `trans_start`.

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 `ETHOOL_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	
→ irrt 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					

(continues on next page)

(continued from previous page)

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      inter-
faces.ifTable.ifEntry.ifDescr.2          =      bond0      inter-
faces.ifTable.ifEntry.ifDescr.3          =      eth0      inter-
faces.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

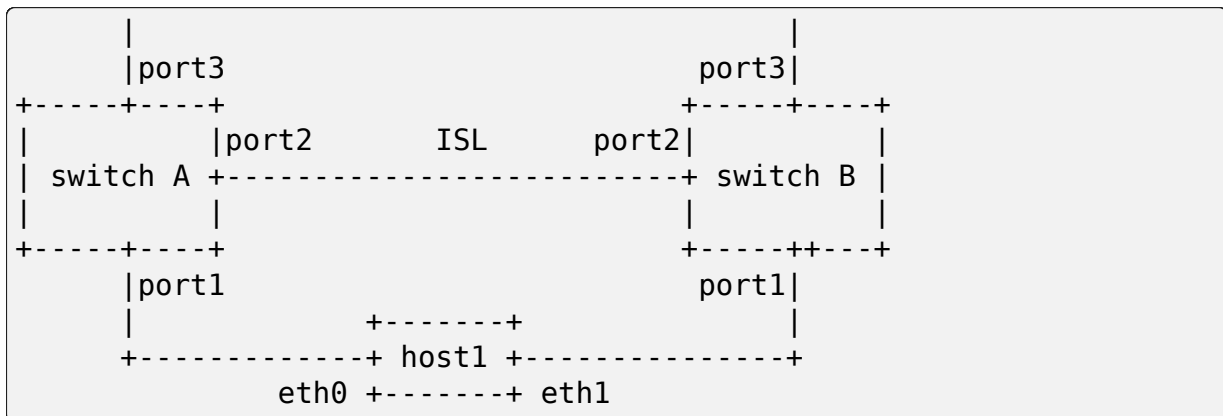
If two hosts (or a host and a single switch) are directly connected via multiple physical links, then there is no availability penalty to optimizing for maximum bandwidth. In this case, there is only one switch (or peer), so if it fails, there is no alternative access to fail over to. Additionally, the 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.

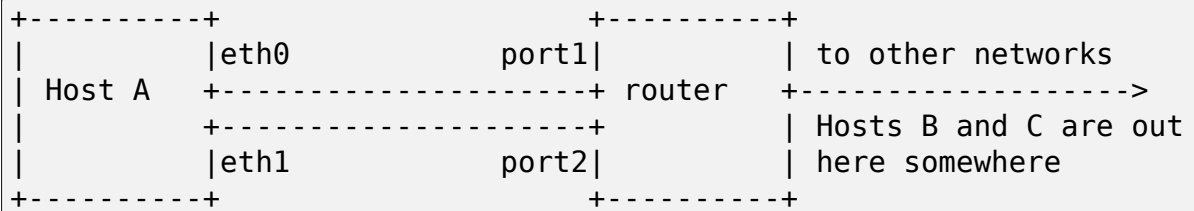
37.13 12. Configuring Bonding for Maximum Throughput

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 “gatewayed” 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:

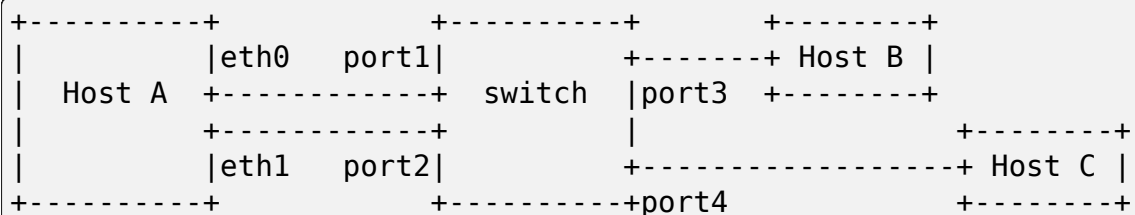


The router may be a dedicated router device, or another host acting as a gateway. For our discussion, the important point is that the majority of traffic from Host A will pass through the router to some other network before reaching its final destination.

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 “gatewayed” 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).

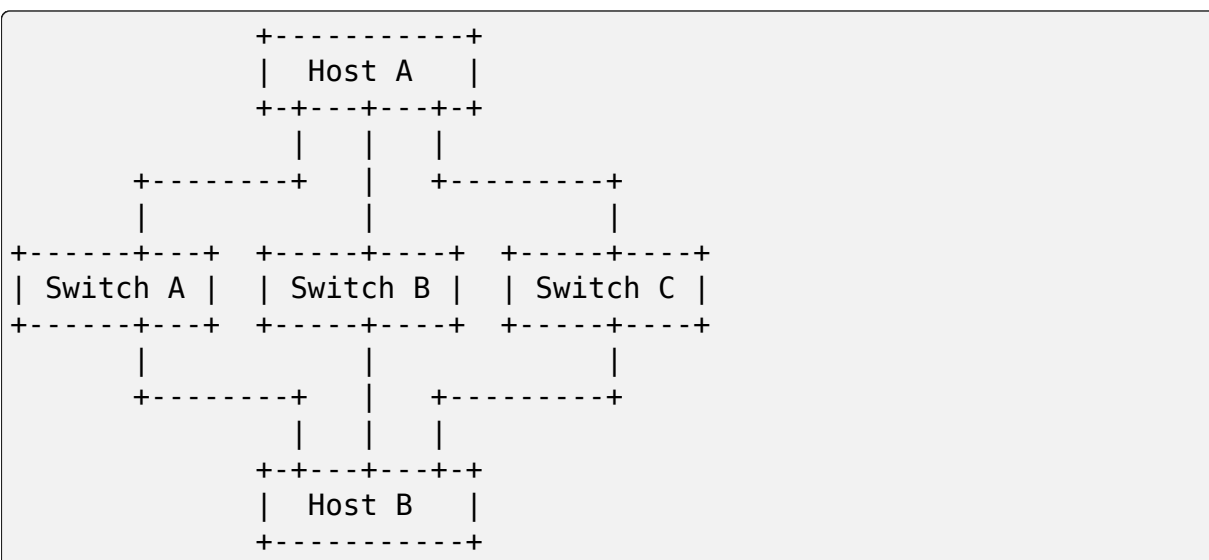
The only additional down side to this mode is that the network device driver must support changing the hardware address while the device is open.

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 requires 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 */

```

(continues on next page)

(continued from previous page)

```

/* 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>	
512 - 4093			b)
4094	IPS	0	c)

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](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](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   = MSG_LEN(sizeof(uint32_t)); /* or MSG_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 $W_{min} = 32$ (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	<code>dns_resolver</code>	*	*	<code>/usr/sbin/cifs.upcall</code>	<code>%k</code>			

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

(continues on next page)

(continued from previous page)

```
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/dnsresolver/parameters/debug
```

SOFTNET DRIVER ISSUES

Transmit path guidelines:

- 1) 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 it's transmit function will become busy.

Instead it must maintain the queue properly. For example, for a driver implementing scatter-gather this means:

```
static netdev_tx_t drv_hard_start_xmit(struct sk_buff *skb,
                                       struct net_device *dev)
{
    struct drv *dp = netdev_priv(dev);

    lock_tx(dp);
    ...
    /* This is a hard error log it. */
    if (TX_BUFFS_AVAIL(dp) <= (skb_shinfo(skb)->nr_frags +
↪1)) {
        netif_stop_queue(dev);
        unlock_tx(dp);
        printk(KERN_ERR PFX "%s: BUG! Tx Ring full when
↪queue awake!\n",
                dev->name);
        return NETDEV_TX_BUSY;
    }

    ... queue packet to card ...
    ... update tx consumer index ...

    if (TX_BUFFS_AVAIL(dp) <= (MAX_SKB_FRAGS + 1))
        netif_stop_queue(dev);

    ...
    unlock_tx(dp);
    ...
    return NETDEV_TX_OK;
}
```

And then at the end of your TX reclamation event handling:

```
if (netif_queue_stopped(dp->dev) &&
    TX_BUFFS_AVAIL(dp) > (MAX_SKB_FRAGS + 1))
    netif_wake_queue(dp->dev);
```

For a non-scatter-gather supporting card, the three tests simply become:

```
/* This is a hard error log it. */
if (TX_BUFFS_AVAIL(dp) <= 0)
```

and:

```
if (TX_BUFFS_AVAIL(dp) == 0)
```

and:

```
if (netif_queue_stopped(dp->dev) &&
    TX_BUFFS_AVAIL(dp) > 0)
    netif_wake_queue(dp->dev);
```

- 2) An `ndo_start_xmit` method must not modify the shared parts of a cloned SKB.
- 3) 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.

Probing guidelines:

- 1) 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()`

Close/stop guidelines:

- 1) 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.
- 2) The `ndo_stop` routine will be called by `unregister_netdevice` if device is still 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 1995 eql-1.1/N0-  
  ↳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/slipo/runslip.conf

Here is an example runslip.conf:

```
name          sl-line-1
enabled
baud          38400
mtu           576
ducmd         -e /etc/slipo/dialout/cua2-288.xp -t 9
command       eql_enlave eql $interface 28800
address       198.67.33.239
line          /dev/cua2

name          sl-line-2
enabled
baud          38400
mtu           576
ducmd         -e /etc/slipo/dialout/cua3-288.xp -t 9
command       eql_enlave 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

(continues on next page)

(continued from previous page)

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 kbits/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

continues on next page

Table 1 - continued from previous page

#lines	speed kbit/sec	mtu	seconds duration	theory speed	actual speed	%of max
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
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
 ↪tried
 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 subtries 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 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

¹ Documentation/userspace-api/seccomp_filter.rst

driver, PTP code, etc where BPF is being used.

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>]

45.2 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 SO_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)`.

45.3 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 },
```

(continues on next page)

(continued from previous page)

```

    { 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, 0x000000800 },
    { 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.

45.4 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[]
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	prandom_u32()

These extensions can also be prefixed with ‘#’. Examples for low-level BPF:

ARP packets:

```
ldh [12]
jne #0x806, drop
```

(continues on next page)

(continued from previous page)

```
ret #-1
drop: ret #0
```

IPv4 TCP packets:

```
ldh [12]
jne #0x800, drop
ldb [23]
jneq #6, drop
ret #-1
drop: ret #0
```

(Accelerated) VLAN w/ id 10:

```
ld vlan_tci
jneq #10, 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 */
```

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 -iem1 -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`.

45.5 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=fffffffffa0069c8f
[ 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)
fffffffa0069c8f + <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)
fffffffa0069c8f + <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
```

(continues on next page)

(continued from previous page)

```

25:    cmp    $0x800,%eax
      3d 00 08 00 00
2a:    jne    0x00000000000000042
      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    0x00000000000000042
      75 07
3b:    mov    $0xffff,%eax
      b8 ff ff 00 00
40:    jmp    0x00000000000000044
      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.

45.6 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' or 'internal BPF' interchangeably. (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.)

It is designed to be JITed with one to one mapping, which can also open up the possibility for GCC/LLVM compilers to generate optimized eBPF code through an eBPF backend that performs almost as fast as natively compiled code.

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 seccomp 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 ker-

nel 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 macro `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 perform JIT compilation from eBPF instruction set.

Some core changes of the new internal format:

- Number of registers increase from 2 to 10:

The old format had two registers A and X, and a hidden frame pointer. The new layout extends this to be 10 internal registers and a read-only frame pointer. Since 64-bit CPUs are passing arguments to functions via registers the number of args from eBPF program to in-kernel function is restricted to 5 and one register is used to accept return value from an in-kernel function. Natively, x86_64 passes first 6 arguments in registers, aarch64/ sparcv9/mips64 have 7 - 8 registers for arguments; x86_64 has 6 callee saved registers, and aarch64/sparcv9/mips64 have 11 or more callee saved registers.

Therefore, eBPF calling convention is defined as:

- R0 - return value from in-kernel function, and exit value for eBPF program
- R1 - R5 - arguments from eBPF program to in-kernel function
- R6 - R9 - callee saved registers that in-kernel function will preserve
- R10 - read-only frame pointer to access stack

Thus, all eBPF registers map one to one to HW registers on x86_64, aarch64, etc, and eBPF calling convention maps directly to ABIs used by the kernel on 64-bit architectures.

On 32-bit architectures JIT may map programs that use only 32-bit arithmetic and may let more complex programs to be interpreted.

R0 - R5 are scratch registers and eBPF program needs spill/fill them if necessary across calls. Note that there is only one eBPF program (== one eBPF main routine) and it cannot call other eBPF functions, it can only call predefined in-kernel functions, though.

- Register width increases from 32-bit to 64-bit:

Still, the semantics of the original 32-bit ALU operations are preserved via 32-bit subregisters. All eBPF registers are 64-bit with 32-bit lower subregisters that zero-extend into 64-bit if they are being written to. That behavior maps directly to x86_64 and arm64 subregister definition, but makes other JITs more difficult.

32-bit architectures run 64-bit internal BPF programs via interpreter. Their JITs may convert BPF programs that only use 32-bit subregisters into native

instruction set and let the rest being interpreted.

Operation is 64-bit, because on 64-bit architectures, pointers are also 64-bit wide, and we want to pass 64-bit values in/out of kernel functions, so 32-bit eBPF registers would otherwise require to define register-pair ABI, thus, there won't be able to use a direct eBPF register to HW register mapping and JIT would need to do combine/split/move operations for every register in and out of the function, which is complex, bug prone and slow. Another reason is the use of atomic 64-bit counters.

- Conditional jt/jf targets replaced with jt/fall-through:

While the original design has constructs such as `if (cond) jump_true; else jump_false;`, they are being replaced into alternative constructs like `if (cond) jump_true; /* else fall-through */`.

- Introduces `bpf_call` insn and register passing convention for zero overhead calls from/to other kernel functions:

Before an in-kernel function call, the internal BPF program needs to place function arguments into R1 to R5 registers to satisfy calling convention, then the interpreter will take them from registers and pass to in-kernel function. If R1 - R5 registers are mapped to CPU registers that are used for argument passing on given architecture, the JIT compiler doesn't need to emit extra moves. Function arguments will be in the correct registers and `BPF_CALL` instruction will be JITed as single 'call' HW instruction. This calling convention was picked to cover common call situations without performance penalty.

After an in-kernel function call, R1 - R5 are reset to unreadable and R0 has a return value of the function. Since R6 - R9 are callee saved, their state is preserved across the call.

For example, consider three C functions:

```
u64 f1() { return (*_f2)(1); }
u64 f2(u64 a) { return f3(a + 1, a); }
u64 f3(u64 a, u64 b) { return a - b; }
```

GCC can compile f1, f3 into x86_64:

```
f1:
    movl $1, %edi
    movq _f2(%rip), %rax
    jmp  *%rax
f3:
    movq %rdi, %rax
    subq %rsi, %rax
    ret
```

Function f2 in eBPF may look like:

```
f2:
    bpf_mov R2, R1
    bpf_add R1, 1
```

(continues on next page)

(continued from previous page)

```
bpf_call f3
bpf_exit
```

If f2 is JITed and the pointer stored to `_f2`. The calls `f1 -> f2 -> f3` and returns will be seamless. Without JIT, `__bpf_prog_run()` interpreter needs to be used to call into f2.

For practical reasons all eBPF programs have only one argument 'ctx' which is already placed into R1 (e.g. on `__bpf_prog_run()` startup) and the programs can call kernel functions with up to 5 arguments. Calls with 6 or more arguments are currently not supported, but these restrictions can be lifted if necessary in the future.

On 64-bit architectures all register map to HW registers one to one. For example, x86_64 JIT compiler can map them as ...

```
R0 - rax
R1 - rdi
R2 - rsi
R3 - rdx
R4 - rcx
R5 - r8
R6 - rbx
R7 - r13
R8 - r14
R9 - r15
R10 - rbp
```

...since x86_64 ABI mandates rdi, rsi, rdx, rcx, r8, r9 for argument passing and rbx, r12 - r15 are callee saved.

Then the following internal BPF pseudo-program:

```
bpf_mov R6, R1 /* save ctx */
bpf_mov R2, 2
bpf_mov R3, 3
bpf_mov R4, 4
bpf_mov R5, 5
bpf_call foo
bpf_mov R7, R0 /* save foo() return value */
bpf_mov R1, R6 /* restore ctx for next call */
bpf_mov R2, 6
bpf_mov R3, 7
bpf_mov R4, 8
bpf_mov R5, 9
bpf_call bar
bpf_add R0, R7
bpf_exit
```

After JIT to x86_64 may look like:

```
push %rbp
mov %rsp,%rbp
sub $0x228,%rsp
mov %rbx,-0x228(%rbp)
mov %r13,-0x220(%rbp)
mov %rdi,%rbx
mov $0x2,%esi
mov $0x3,%edx
mov $0x4,%ecx
mov $0x5,%r8d
callq foo
mov %rax,%r13
mov %rbx,%rdi
mov $0x6,%esi
mov $0x7,%edx
mov $0x8,%ecx
mov $0x9,%r8d
callq bar
add %r13,%rax
mov -0x228(%rbp),%rbx
mov -0x220(%rbp),%r13
leaveq
retq
```

Which is in this example equivalent in C to:

```
u64 bpf_filter(u64 ctx)
{
    return foo(ctx, 2, 3, 4, 5) + bar(ctx, 6, 7, 8, 9);
}
```

In-kernel functions `foo()` and `bar()` with prototype: `u64 (*)(u64 arg1, u64 arg2, u64 arg3, u64 arg4, u64 arg5)`; will receive arguments in proper registers and place their return value into `%rax` which is `R0` in eBPF. Prologue and epilogue are emitted by JIT and are implicit in the interpreter. `R0-R5` are scratch registers, so eBPF program needs to preserve them across the calls as defined by calling convention.

For example the following program is invalid:

```
bpf_mov R1, 1
bpf_call foo
bpf_mov R0, R1
bpf_exit
```

After the call the registers `R1-R5` contain junk values and cannot be read. An in-kernel eBPF verifier is used to validate internal BPF programs.

Also in the new design, eBPF is limited to 4096 insns, which means that any program will terminate quickly and will only call a fixed number of kernel functions. Original BPF and the new format are two operand instructions, which helps to do one-to-one mapping between eBPF insn and x86 insn during JIT.

The input context pointer for invoking the interpreter function is generic, its content is defined by a specific use case. For seccomp register R1 points to `seccomp_data`, for converted BPF filters R1 points to a `skb`.

A program, that is translated internally consists of the following elements:

```
op:16, jt:8, jf:8, k:32    ==>    op:8, dst_reg:4, src_reg:4,
↪off:16, imm:32
```

So far 87 internal BPF instructions were implemented. 8-bit ‘op’ opcode field has room for new instructions. Some of them may use 16/24/32 byte encoding. New instructions must be multiple of 8 bytes to preserve backward compatibility.

Internal BPF is a general purpose RISC instruction set. Not every register and every instruction are used during translation from original BPF to new format. For example, socket filters are not using exclusive add instruction, but tracing filters may do to maintain counters of events, for example. Register R9 is not used by socket filters either, but more complex filters may be running out of registers and would have to resort to spill/fill to stack.

Internal BPF can be used as a generic assembler for last step performance optimizations, socket filters and seccomp are using it as assembler. Tracing filters may use it as assembler to generate code from kernel. In kernel usage may not be bounded by security considerations, since generated internal BPF code may be optimizing internal code path and not being exposed to the user space. Safety of internal BPF can come from a verifier (TBD). In such use cases as described, it may be used as safe instruction set.

Just like the original BPF, the new format runs within a controlled environment, is deterministic and the kernel can easily prove that. The safety of the program can be determined in two steps: first step does depth-first-search to disallow loops and other CFG validation; second step starts from the first insn and descends all possible paths. It simulates execution of every insn and observes the state change of registers and stack.

45.7 eBPF opcode encoding

eBPF is reusing most of the opcode encoding from classic to simplify conversion of classic BPF to eBPF. For arithmetic and jump instructions the 8-bit ‘code’ field is divided into three parts:

+-----+-----+-----+-----+			
4 bits		1 bit	3 bits
operation code		source	instruction class
+-----+-----+-----+-----+			
(MSB)			(LSB)

Three LSB bits store instruction class which is one of:

Classic BPF classes	eBPF classes
BPF_LD 0x00	BPF_LD 0x00
BPF_LDX 0x01	BPF_LDX 0x01
BPF_ST 0x02	BPF_ST 0x02
BPF_STX 0x03	BPF_STX 0x03
BPF_ALU 0x04	BPF_ALU 0x04
BPF_JMP 0x05	BPF_JMP 0x05
BPF_RET 0x06	BPF_JMP32 0x06
BPF_MISC 0x07	BPF_ALU64 0x07

When `BPF_CLASS(code) == BPF_ALU` or `BPF_JMP`, 4th bit encodes source operand ...

BPF_K	0x00
BPF_X	0x08

- in classic BPF, this means:

<code>BPF_SRC(code) == BPF_X</code> - use register X as source ↳ operand
<code>BPF_SRC(code) == BPF_K</code> - use 32-bit immediate as source ↳ operand

- in eBPF, this means:

<code>BPF_SRC(code) == BPF_X</code> - use 'src_reg' register as ↳ source operand
<code>BPF_SRC(code) == BPF_K</code> - use 32-bit immediate as source ↳ operand

...and four MSB bits store operation code.

If `BPF_CLASS(code) == BPF_ALU` or `BPF_ALU64` [in eBPF], `BPF_OP(code)` is one of:

BPF_ADD	0x00	
BPF_SUB	0x10	
BPF_MUL	0x20	
BPF_DIV	0x30	
BPF_OR	0x40	
BPF_AND	0x50	
BPF_LSH	0x60	
BPF_RSH	0x70	
BPF_NEG	0x80	
BPF_MOD	0x90	
BPF_XOR	0xa0	
BPF_MOV	0xb0	/* eBPF only: mov reg to reg */
BPF_ARSH	0xc0	/* eBPF only: sign extending shift right */
BPF_END	0xd0	/* eBPF only: endianness conversion */

If `BPF_CLASS(code) == BPF_JMP` or `BPF_JMP32` [in eBPF], `BPF_OP(code)` is one of:

```

BPF_JA    0x00    /* BPF_JMP only */
BPF_JEQ   0x10
BPF_JGT   0x20
BPF_JGE   0x30
BPF_JSET   0x40
BPF_JNE   0x50    /* eBPF only: jump != */
BPF_JSGT  0x60    /* eBPF only: signed '>' */
BPF_JSGE  0x70    /* eBPF only: signed '>=' */
BPF_CALL  0x80    /* eBPF BPF_JMP only: function call */
BPF_EXIT  0x90    /* eBPF BPF_JMP only: function return */
BPF_JLT   0xa0    /* eBPF only: unsigned '<' */
BPF_JLE   0xb0    /* eBPF only: unsigned '<=' */
BPF_JSLT  0xc0    /* eBPF only: signed '<' */
BPF_JSLE  0xd0    /* eBPF only: signed '<=' */

```

So `BPF_ADD` | `BPF_X` | `BPF_ALU` means 32-bit addition in both classic BPF and eBPF. There are only two registers in classic BPF, so it means `A += X`. In eBPF it means `dst_reg = (u32) dst_reg + (u32) src_reg`; similarly, `BPF_XOR` | `BPF_K` | `BPF_ALU` means `A ^= imm32` in classic BPF and analogous `src_reg = (u32) src_reg ^ (u32) imm32` in eBPF.

Classic BPF is using `BPF_MISC` class to represent `A = X` and `X = A` moves. eBPF is using `BPF_MOV` | `BPF_X` | `BPF_ALU` code instead. Since there are no `BPF_MISC` operations in eBPF, the class 7 is used as `BPF_ALU64` to mean exactly the same operations as `BPF_ALU`, but with 64-bit wide operands instead. So `BPF_ADD` | `BPF_X` | `BPF_ALU64` means 64-bit addition, i.e.: `dst_reg = dst_reg + src_reg`

Classic BPF wastes the whole `BPF_RET` class to represent a single `ret` operation. Classic `BPF_RET` | `BPF_K` means copy `imm32` into return register and perform function exit. eBPF is modeled to match CPU, so `BPF_JMP` | `BPF_EXIT` in eBPF means function exit only. The eBPF program needs to store return value into register `R0` before doing a `BPF_EXIT`. Class 6 in eBPF is used as `BPF_JMP32` to mean exactly the same operations as `BPF_JMP`, but with 32-bit wide operands for the comparisons instead.

For load and store instructions the 8-bit ‘code’ field is divided as:

```

+-----+-----+-----+
| 3 bits | 2 bits | 3 bits |
| mode  | size  | instruction class |
+-----+-----+-----+
(MSB)                                     (LSB)

```

Size modifier is one of ...

```

BPF_W    0x00    /* word */
BPF_H    0x08    /* half word */
BPF_B    0x10    /* byte */
BPF_DW   0x18    /* eBPF only, double word */

```

...which encodes size of load/store operation:

```
B - 1 byte
H - 2 byte
W - 4 byte
DW - 8 byte (eBPF only)
```

Mode modifier is one of:

```
BPF_IMM 0x00 /* used for 32-bit mov in classic BPF and 64-bit in_
↳eBPF */
BPF_ABS 0x20
BPF_IND 0x40
BPF_MEM 0x60
BPF_LEN 0x80 /* classic BPF only, reserved in eBPF */
BPF_MSH 0xa0 /* classic BPF only, reserved in eBPF */
BPF_XADD 0xc0 /* eBPF only, exclusive add */
```

eBPF has two non-generic instructions: (BPF_ABS | <size> | BPF_LD) and (BPF_IND | <size> | BPF_LD) which are used to access packet data.

They had to be carried over from classic to have strong performance of socket filters running in eBPF interpreter. These instructions can only be used when interpreter context is a pointer to struct sk_buff and have seven implicit operands. Register R6 is an implicit input that must contain pointer to sk_buff. Register R0 is an implicit output which contains the data fetched from the packet. Registers R1-R5 are scratch registers and must not be used to store the data across BPF_ABS | BPF_LD or BPF_IND | BPF_LD instructions.

These instructions have implicit program exit condition as well. When eBPF program is trying to access the data beyond the packet boundary, the interpreter will abort the execution of the program. JIT compilers therefore must preserve this property. src_reg and imm32 fields are explicit inputs to these instructions.

For example:

BPF_IND | BPF_W | BPF_LD means:

```
R0 = ntohs(*(u32 *) (((struct sk_buff *) R6)->data + src_reg +_
↳imm32))
and R1 - R5 were scratched.
```

Unlike classic BPF instruction set, eBPF has generic load/store operations:

```
BPF_MEM | <size> | BPF_STX: *(size *) (dst_reg + off) = src_reg
BPF_MEM | <size> | BPF_ST:  *(size *) (dst_reg + off) = imm32
BPF_MEM | <size> | BPF_LDX: dst_reg = *(size *) (src_reg + off)
BPF_XADD | BPF_W | BPF_STX: lock xadd *(u32 *) (dst_reg + off16) +=_
↳src_reg
BPF_XADD | BPF_DW | BPF_STX: lock xadd *(u64 *) (dst_reg + off16) +=_
↳src_reg
```

Where size is one of: BPF_B or BPF_H or BPF_W or BPF_DW. Note that 1 and 2 byte atomic increments are not supported.

eBPF has one 16-byte instruction: `BPF_LD | BPF_DW | BPF_IMM` which consists of two consecutive `struct bpf_insn` 8-byte blocks and interpreted as single instruction that loads 64-bit immediate value into a `dst_reg`. Classic BPF has similar instruction: `BPF_LD | BPF_W | BPF_IMM` which loads 32-bit immediate value into a register.

45.8 eBPF verifier

The safety of the eBPF program is determined in two steps.

First step does DAG check to disallow loops and other CFG validation. In particular it will detect programs that have unreachable instructions. (though classic BPF checker allows them)

Second step starts from the first insn and descends all possible paths. It simulates execution of every insn and observes the state change of registers and stack.

At the start of the program the register `R1` contains a pointer to context and has type `PTR_TO_CTX`. If verifier sees an insn that does `R2=R1`, then `R2` has now type `PTR_TO_CTX` as well and can be used on the right hand side of expression. If `R1=PTR_TO_CTX` and insn is `R2=R1+R1`, then `R2=SCALAR_VALUE`, since addition of two valid pointers makes invalid pointer. (In ‘secure’ mode verifier will reject any type of pointer arithmetic to make sure that kernel addresses don’t leak to unprivileged users)

If register was never written to, it’s not readable:

```
bpf_mov R0 = R2
bpf_exit
```

will be rejected, since `R2` is unreadable at the start of the program.

After kernel function call, `R1-R5` are reset to unreadable and `R0` has a return type of the function.

Since `R6-R9` are callee saved, their state is preserved across the call.

```
bpf_mov R6 = 1
bpf_call foo
bpf_mov R0 = R6
bpf_exit
```

is a correct program. If there was `R1` instead of `R6`, it would have been rejected.

load/store instructions are allowed only with registers of valid types, which are `PTR_TO_CTX`, `PTR_TO_MAP`, `PTR_TO_STACK`. They are bounds and alignment checked. For example:

```
bpf_mov R1 = 1
bpf_mov R2 = 2
bpf_xadd *(u32 *) (R1 + 3) += R2
bpf_exit
```

will be rejected, since R1 doesn't have a valid pointer type at the time of execution of instruction `bpf_xadd`.

At the start R1 type is `PTR_TO_CTX` (a pointer to generic struct `bpf_context`) A callback is used to customize verifier to restrict eBPF program access to only certain fields within `ctx` structure with specified size and alignment.

For example, the following insn:

```
bpf_ld R0 = *(u32 *) (R6 + 8)
```

intends to load a word from address `R6 + 8` and store it into `R0` If `R6=PTR_TO_CTX`, via `is_valid_access()` callback the verifier will know that offset 8 of size 4 bytes can be accessed for reading, otherwise the verifier will reject the program. If `R6=PTR_TO_STACK`, then access should be aligned and be within stack bounds, which are `[-MAX_BPF_STACK, 0)`. In this example offset is 8, so it will fail verification, since it's out of bounds.

The verifier will allow eBPF program to read data from stack only after it wrote into it.

Classic BPF verifier does similar check with `M[0-15]` memory slots. For example:

```
bpf_ld R0 = *(u32 *) (R10 - 4)
bpf_exit
```

is invalid program. Though `R10` is correct read-only register and has type `PTR_TO_STACK` and `R10 - 4` is within stack bounds, there were no stores into that location.

Pointer register spill/fill is tracked as well, since four (`R6-R9`) callee saved registers may not be enough for some programs.

Allowed function calls are customized with `bpf_verifier_ops->get_func_proto()` The eBPF verifier will check that registers match argument constraints. After the call register `R0` will be set to return type of the function.

Function calls is a main mechanism to extend functionality of eBPF programs. Socket filters may let programs to call one set of functions, whereas tracing filters may allow completely different set.

If a function made accessible to eBPF program, it needs to be thought through from safety point of view. The verifier will guarantee that the function is called with valid arguments.

seccomp vs socket filters have different security restrictions for classic BPF. Seccomp solves this by two stage verifier: classic BPF verifier is followed by seccomp verifier. In case of eBPF one configurable verifier is shared for all use cases.

See details of eBPF verifier in `kernel/bpf/verifier.c`

45.9 Register value tracking

In order to determine the safety of an eBPF program, the verifier must track the range of possible values in each register and also in each stack slot. This is done with `struct bpf_reg_state`, defined in `include/linux/bpf_verifier.h`, which unifies tracking of scalar and pointer values. Each register state has a type, which is either `NOT_INIT` (the register has not been written to), `SCALAR_VALUE` (some value which is not usable as a pointer), or a pointer type. The types of pointers describe their base, as follows:

PTR_TO_CTX

Pointer to `bpf_context`.

CONST_PTR_TO_MAP

Pointer to `struct bpf_map`. “Const” because arithmetic on these pointers is forbidden.

PTR_TO_MAP_VALUE

Pointer to the value stored in a map element.

PTR_TO_MAP_VALUE_OR_NULL

Either a pointer to a map value, or `NULL`; map accesses (see section ‘eBPF maps’, below) return this type, which becomes a `PTR_TO_MAP_VALUE` when checked `!= NULL`. Arithmetic on these pointers is forbidden.

PTR_TO_STACK

Frame pointer.

PTR_TO_PACKET

`skb->data`.

PTR_TO_PACKET_END

`skb->data + headlen`; arithmetic forbidden.

PTR_TO_SOCKET

Pointer to `struct bpf_sock_ops`, implicitly refcounted.

PTR_TO_SOCKET_OR_NULL

Either a pointer to a socket, or `NULL`; socket lookup returns this type, which becomes a `PTR_TO_SOCKET` when checked `!= NULL`. `PTR_TO_SOCKET` is reference-counted, so programs must release the reference through the socket release function before the end of the program. Arithmetic on these pointers is forbidden.

However, a pointer may be offset from this base (as a result of pointer arithmetic), and this is tracked in two parts: the ‘fixed offset’ and ‘variable offset’. The former is used when an exactly-known value (e.g. an immediate operand) is added to a pointer, while the latter is used for values which are not exactly known. The variable offset is also used in `SCALAR_VALUES`, to track the range of possible values in the register.

The verifier’s knowledge about the variable offset consists of:

- minimum and maximum values as unsigned
- minimum and maximum values as signed

- knowledge of the values of individual bits, in the form of a ‘tnum’ : a u64 ‘mask’ and a u64 ‘value’ . 1s in the mask represent bits whose value is unknown; 1s in the value represent bits known to be 1. Bits known to be 0 have 0 in both mask and value; no bit should ever be 1 in both. For example, if a byte is read into a register from memory, the register’s top 56 bits are known zero, while the low 8 are unknown - which is represented as the tnum (0x0; 0xff). If we then OR this with 0x40, we get (0x40; 0xbf), then if we add 1 we get (0x0; 0x1ff), because of potential carries.

Besides arithmetic, the register state can also be updated by conditional branches. For instance, if a SCALAR_VALUE is compared > 8 , in the ‘true’ branch it will have a `umin_value` (unsigned minimum value) of 9, whereas in the ‘false’ branch it will have a `umax_value` of 8. A signed compare (with `BPF_JSGT` or `BPF_JSGE`) would instead update the signed minimum/maximum values. Information from the signed and unsigned bounds can be combined; for instance if a value is first tested < 8 and then tested $s > 4$, the verifier will conclude that the value is also > 4 and $s < 8$, since the bounds prevent crossing the sign boundary.

`PTR_TO_PACKETs` with a variable offset part have an ‘id’ , which is common to all pointers sharing that same variable offset. This is important for packet range checks: after adding a variable to a packet pointer register A, if you then copy it to another register B and then add a constant 4 to A, both registers will share the same ‘id’ but the A will have a fixed offset of +4. Then if A is bounds-checked and found to be less than a `PTR_TO_PACKET_END`, the register B is now known to have a safe range of at least 4 bytes. See ‘Direct packet access’ , below, for more on `PTR_TO_PACKET` ranges.

The ‘id’ field is also used on `PTR_TO_MAP_VALUE_OR_NULL`, common to all copies of the pointer returned from a map lookup. This means that when one copy is checked and found to be non-NULL, all copies can become `PTR_TO_MAP_VALUES`. As well as range-checking, the tracked information is also used for enforcing alignment of pointer accesses. For instance, on most systems the packet pointer is 2 bytes after a 4-byte alignment. If a program adds 14 bytes to that to jump over the Ethernet header, then reads IHL and adds $(IHL * 4)$, the resulting pointer will have a variable offset known to be $4n+2$ for some n , so adding the 2 bytes (`NET_IP_ALIGN`) gives a 4-byte alignment and so word-sized accesses through that pointer are safe. The ‘id’ field is also used on `PTR_TO_SOCKET` and `PTR_TO_SOCKET_OR_NULL`, common to all copies of the pointer returned from a socket lookup. This has similar behaviour to the handling for `PTR_TO_MAP_VALUE_OR_NULL`->`PTR_TO_MAP_VALUE`, but it also handles reference tracking for the pointer. `PTR_TO_SOCKET` implicitly represents a reference to the corresponding struct `sock`. To ensure that the reference is not leaked, it is imperative to NULL-check the reference and in the non-NULL case, and pass the valid reference to the socket release function.

45.10 Direct packet access

In `cls_bpf` and `act_bpf` programs the verifier allows direct access to the packet data via `skb->data` and `skb->data_end` pointers. Ex:

```
1:  r4 = *(u32 *)(r1 +80) /* load skb->data_end */
2:  r3 = *(u32 *)(r1 +76) /* load skb->data */
3:  r5 = r3
4:  r5 += 14
5:  if r5 > r4 goto pc+16
R1=ctx R3=pkt(id=0,off=0,r=14) R4=pkt_end R5=pkt(id=0,off=14,r=14)
↳R10=fp
6:  r0 = *(u16 *)(r3 +12) /* access 12 and 13 bytes of the packet */
```

this 2byte load from the packet is safe to do, since the program author did check `if (skb->data + 14 > skb->data_end) goto err` at insn #5 which means that in the fall-through case the register R3 (which points to `skb->data`) has at least 14 directly accessible bytes. The verifier marks it as `R3=pkt(id=0,off=0,r=14)`. `id=0` means that no additional variables were added to the register. `off=0` means that no additional constants were added. `r=14` is the range of safe access which means that bytes `[R3, R3 + 14)` are ok. Note that R5 is marked as `R5=pkt(id=0,off=14,r=14)`. It also points to the packet data, but constant 14 was added to the register, so it now points to `skb->data + 14` and accessible range is `[R5, R5 + 14 - 14)` which is zero bytes.

More complex packet access may look like:

```
R0=inv1 R1=ctx R3=pkt(id=0,off=0,r=14) R4=pkt_end R5=pkt(id=0,
↳off=14,r=14) R10=fp
6:  r0 = *(u8 *)(r3 +7) /* load 7th byte from the packet */
7:  r4 = *(u8 *)(r3 +12)
8:  r4 *= 14
9:  r3 = *(u32 *)(r1 +76) /* load skb->data */
10: r3 += r4
11: r2 = r1
12: r2 <= 48
13: r2 >= 48
14: r3 += r2
15: r2 = r3
16: r2 += 8
17: r1 = *(u32 *)(r1 +80) /* load skb->data_end */
18: if r2 > r1 goto pc+2
R0=inv(id=0,umax_value=255,var_off=(0x0; 0xff)) R1=pkt_end
↳R2=pkt(id=2,off=8,r=8) R3=pkt(id=2,off=0,r=8) R4=inv(id=0,umax_
↳value=3570,var_off=(0x0; 0xffff)) R5=pkt(id=0,off=14,r=14) R10=fp
19: r1 = *(u8 *)(r3 +4)
```

The state of the register R3 is `R3=pkt(id=2,off=0,r=8)` `id=2` means that two `r3 += rX` instructions were seen, so `r3` points to some offset within a packet and since the program author did `if (r3 + 8 > r1) goto err` at insn #18, the safe range is `[R3, R3 + 8)`. The verifier only allows ‘add’ / ‘sub’ operations on packet registers. Any other operation will set the register state to ‘SCALAR_VALUE’ and it won’t

be available for direct packet access.

Operation `r3 += rX` may overflow and become less than original `skb->data`, therefore the verifier has to prevent that. So when it sees `r3 += rX` instruction and `rX` is more than 16-bit value, any subsequent bounds-check of `r3` against `skb->data_end` will not give us ‘range’ information, so attempts to read through the pointer will give “invalid access to packet” error.

Ex. after insn `r4 = *(u8*)(r3+12)` (insn #7 above) the state of `r4` is `R4=inv(id=0,umax_value=255,var_off=(0x0; 0xff))` which means that upper 56 bits of the register are guaranteed to be zero, and nothing is known about the lower 8 bits. After insn `r4 *= 14` the state becomes `R4=inv(id=0,umax_value=3570,var_off=(0x0; 0xfffe))`, since multiplying an 8-bit value by constant 14 will keep upper 52 bits as zero, also the least significant bit will be zero as 14 is even. Similarly `r2 >>= 48` will make `R2=inv(id=0,umax_value=65535,var_off=(0x0; 0xffff))`, since the shift is not sign extending. This logic is implemented in `adjust_reg_min_max_vals()` function, which calls `adjust_ptr_min_max_vals()` for adding pointer to scalar (or vice versa) and `adjust_scalar_min_max_vals()` for operations on two scalars.

The end result is that bpf program author can access packet directly using normal C code as:

```
void *data = (void*)(long)skb->data;
void *data_end = (void*)(long)skb->data_end;
struct eth_hdr *eth = data;
struct iphdr *iph = data + sizeof(*eth);
struct udphdr *udp = data + sizeof(*eth) + sizeof(*iph);

if (data + sizeof(*eth) + sizeof(*iph) + sizeof(*udp) > data_end)
    return 0;
if (eth->h_proto != htons(ETH_P_IP))
    return 0;
if (iph->protocol != IPPROTO_UDP || iph->ihl != 5)
    return 0;
if (udp->dest == 53 || udp->source == 9)
    ...;
```

which makes such programs easier to write comparing to `LD_ABS` insn and significantly faster.

45.11 eBPF maps

‘maps’ is a generic storage of different types for sharing data between kernel and userspace.

The maps are accessed from user space via BPF syscall, which has commands:

- create a map with given type and attributes `map_fd = bpf(BPF_MAP_CREATE, union bpf_attr *attr, u32 size)` using `attr->map_type`, `attr->key_size`, `attr->value_size`, `attr->max_entries` returns process-local file descriptor or negative error

- lookup key in a given map `err = bpf(BPF_MAP_LOOKUP_ELEM, union bpf_attr *attr, u32 size)` using `attr->map_fd`, `attr->key`, `attr->value` returns zero and stores found elem into value or negative error
- create or update key/value pair in a given map `err = bpf(BPF_MAP_UPDATE_ELEM, union bpf_attr *attr, u32 size)` using `attr->map_fd`, `attr->key`, `attr->value` returns zero or negative error
- find and delete element by key in a given map `err = bpf(BPF_MAP_DELETE_ELEM, union bpf_attr *attr, u32 size)` using `attr->map_fd`, `attr->key`
- to delete map: `close(fd)` Exiting process will delete maps automatically

userspace programs use this syscall to create/access maps that eBPF programs are concurrently updating.

maps can have different types: hash, array, bloom filter, radix-tree, etc.

The map is defined by:

- type
- max number of elements
- key size in bytes
- value size in bytes

45.12 Pruning

The verifier does not actually walk all possible paths through the program. For each new branch to analyse, the verifier looks at all the states it's previously been in when at this instruction. If any of them contain the current state as a subset, the branch is 'pruned' - that is, the fact that the previous state was accepted implies the current state would be as well. For instance, if in the previous state, `r1` held a packet-pointer, and in the current state, `r1` holds a packet-pointer with a range as long or longer and at least as strict an alignment, then `r1` is safe. Similarly, if `r2` was `NOT_INIT` before then it can't have been used by any path from that point, so any value in `r2` (including another `NOT_INIT`) is safe. The implementation is in the function `regsafe()`. Pruning considers not only the registers but also the stack (and any spilled registers it may hold). They must all be safe for the branch to be pruned. This is implemented in `states_equal()`.

45.13 Understanding eBPF verifier messages

The following are few examples of invalid eBPF programs and verifier error messages as seen in the log:

Program with unreachable instructions:

```
static struct bpf_insn prog[] = {
    BPF_EXIT_INSN(),
```

(continues on next page)

(continued from previous page)

```
BPF_EXIT_INSN(),  
};
```

Error:

unreachable insn 1

Program that reads uninitialized register:

```
BPF_MOV64_REG(BPF_REG_0, BPF_REG_2),  
BPF_EXIT_INSN(),
```

Error:

```
0: (bf) r0 = r2  
R2 !read_ok
```

Program that doesn't initialize R0 before exiting:

```
BPF_MOV64_REG(BPF_REG_2, BPF_REG_1),  
BPF_EXIT_INSN(),
```

Error:

```
0: (bf) r2 = r1  
1: (95) exit  
R0 !read_ok
```

Program that accesses stack out of bounds:

```
BPF_ST_MEM(BPF_DW, BPF_REG_10, 8, 0),  
BPF_EXIT_INSN(),
```

Error:

```
0: (7a) *(u64 *) (r10 +8) = 0  
invalid stack off=8 size=8
```

Program that doesn't initialize stack before passing its address into function:

```
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),  
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),  
BPF_LD_MAP_FD(BPF_REG_1, 0),  
BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),  
BPF_EXIT_INSN(),
```

Error:

```
0: (bf) r2 = r10  
1: (07) r2 += -8  
2: (b7) r1 = 0x0  
3: (85) call 1  
invalid indirect read from stack off -8+0 size 8
```

Program that uses invalid map_fd=0 while calling to map_lookup_elem() function:

```
BPF_ST_MEM(BPF_DW, BPF_REG_10, -8, 0),
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_LD_MAP_FD(BPF_REG_1, 0),
BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
BPF_EXIT_INSN(),
```

Error:

```
0: (7a) *(u64 *) (r10 -8) = 0
1: (bf) r2 = r10
2: (07) r2 += -8
3: (b7) r1 = 0x0
4: (85) call 1
fd 0 is not pointing to valid bpf_map
```

Program that doesn't check return value of map_lookup_elem() before accessing map element:

```
BPF_ST_MEM(BPF_DW, BPF_REG_10, -8, 0),
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_LD_MAP_FD(BPF_REG_1, 0),
BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
BPF_ST_MEM(BPF_DW, BPF_REG_0, 0, 0),
BPF_EXIT_INSN(),
```

Error:

```
0: (7a) *(u64 *) (r10 -8) = 0
1: (bf) r2 = r10
2: (07) r2 += -8
3: (b7) r1 = 0x0
4: (85) call 1
5: (7a) *(u64 *) (r0 +0) = 0
R0 invalid mem access 'map_value_or_null'
```

Program that correctly checks map_lookup_elem() returned value for NULL, but accesses the memory with incorrect alignment:

```
BPF_ST_MEM(BPF_DW, BPF_REG_10, -8, 0),
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_LD_MAP_FD(BPF_REG_1, 0),
BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 1),
BPF_ST_MEM(BPF_DW, BPF_REG_0, 4, 0),
BPF_EXIT_INSN(),
```

Error:

```
0: (7a) *(u64 *)(r10 -8) = 0
1: (bf) r2 = r10
2: (07) r2 += -8
3: (b7) r1 = 1
4: (85) call 1
5: (15) if r0 == 0x0 goto pc+1
   R0=map_ptr R10=fp
6: (7a) *(u64 *)(r0 +4) = 0
   misaligned access off 4 size 8
```

Program that correctly checks `map_lookup_elem()` returned value for NULL and accesses memory with correct alignment in one side of ‘if’ branch, but fails to do so in the other side of ‘if’ branch:

```
BPF_ST_MEM(BPF_DW, BPF_REG_10, -8, 0),
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_LD_MAP_FD(BPF_REG_1, 0),
BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 2),
BPF_ST_MEM(BPF_DW, BPF_REG_0, 0, 0),
BPF_EXIT_INSN(),
BPF_ST_MEM(BPF_DW, BPF_REG_0, 0, 1),
BPF_EXIT_INSN(),
```

Error:

```
0: (7a) *(u64 *)(r10 -8) = 0
1: (bf) r2 = r10
2: (07) r2 += -8
3: (b7) r1 = 1
4: (85) call 1
5: (15) if r0 == 0x0 goto pc+2
   R0=map_ptr R10=fp
6: (7a) *(u64 *)(r0 +0) = 0
7: (95) exit

from 5 to 8: R0=imm0 R10=fp
8: (7a) *(u64 *)(r0 +0) = 1
R0 invalid mem access 'imm'
```

Program that performs a socket lookup then sets the pointer to NULL without checking it:

```
BPF_MOV64_IMM(BPF_REG_2, 0),
BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_2, -8),
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_MOV64_IMM(BPF_REG_3, 4),
BPF_MOV64_IMM(BPF_REG_4, 0),
BPF_MOV64_IMM(BPF_REG_5, 0),
```

(continues on next page)

(continued from previous page)

```
BPF_EMIT_CALL(BPF_FUNC_sk_lookup_tcp),
BPF_MOV64_IMM(BPF_REG_0, 0),
BPF_EXIT_INSN(),
```

Error:

```
0: (b7) r2 = 0
1: (63) *(u32 *) (r10 -8) = r2
2: (bf) r2 = r10
3: (07) r2 += -8
4: (b7) r3 = 4
5: (b7) r4 = 0
6: (b7) r5 = 0
7: (85) call bpf_sk_lookup_tcp#65
8: (b7) r0 = 0
9: (95) exit
Unreleased reference id=1, alloc_insn=7
```

Program that performs a socket lookup but does not NULL-check the returned value:

```
BPF_MOV64_IMM(BPF_REG_2, 0),
BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_2, -8),
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_MOV64_IMM(BPF_REG_3, 4),
BPF_MOV64_IMM(BPF_REG_4, 0),
BPF_MOV64_IMM(BPF_REG_5, 0),
BPF_EMIT_CALL(BPF_FUNC_sk_lookup_tcp),
BPF_EXIT_INSN(),
```

Error:

```
0: (b7) r2 = 0
1: (63) *(u32 *) (r10 -8) = r2
2: (bf) r2 = r10
3: (07) r2 += -8
4: (b7) r3 = 4
5: (b7) r4 = 0
6: (b7) r5 = 0
7: (85) call bpf_sk_lookup_tcp#65
8: (95) exit
Unreleased reference id=1, alloc_insn=7
```

45.14 Testing

Next to the BPF toolchain, the kernel also ships a test module that contains various test cases for classic and internal BPF 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`).

45.15 Misc

Also trinity, the Linux syscall fuzzer, has built-in support for BPF and SECCOMP-BPF kernel fuzzing.

45.16 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 Schulist <jschlst@samba.org>
- Daniel Borkmann <daniel@iogearbox.net>
- Alexei Starovoitov <ast@kernel.org>

FRAME RELAY (FR)

Frame Relay (FR) support for linux is built into a two tiered system of device drivers. The upper layer implements RFC1490 FR specification, and uses the Data Link Connection Identifier (DLCI) as its hardware address. Usually these are assigned by your network supplier, they give you the number/numbers of the Virtual Connections (VC) assigned to you.

Each DLCI is a point-to-point link between your machine and a remote one. As such, a separate device is needed to accommodate the routing. Within the net-tools archives is `'dlcicfg'`. This program will communicate with the base "DLCI" device, and create new net devices named `'dlci00'`, `'dlci01'` ... The configuration script will ask you how many DLCIs you need, as well as how many DLCIs you want to assign to each Frame Relay Access Device (FRAD).

The DLCI uses a number of function calls to communicate with the FRAD, all of which are stored in the FRAD's private data area. `assoc/deassoc`, `activate/deactivate` and `dlci_config`. The DLCI supplies a receive function to the FRAD to accept incoming packets.

With this initial offering, only 1 FRAD driver is available. With many thanks to Sangoma Technologies, David Mandelstam & Gene Kozin, the S502A, S502E & S508 are supported. This driver is currently set up for only FR, but as Sangoma makes more firmware modules available, it can be updated to provide them as well.

Configuration of the FRAD makes use of another net-tools program, `'fradcfg'`. This program makes use of a configuration file (which `dlcicfg` can also read) to specify the types of boards to be configured as FRADs, as well as perform any board specific configuration. The Sangoma module of `fradcfg` loads the FR firmware into the card, sets the irq/port/memory information, and provides an initial configuration.

Additional FRAD device drivers can be added as hardware is available.

At this time, the `dlcicfg` and `fradcfg` programs have not been incorporated into the net-tools distribution. They can be found at ftp.invlogic.com, in `/pub/linux`. Note that with OS/2 FTPD, you end up in `/pub` by default, so just use `'cd linux'`. v0.10 is for use on pre-2.0.3 and earlier, v0.15 is for pre-2.0.4 and later.

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 “`sethdlc`” utility, get it from:

<http://www.kernel.org/pub/linux/utils/net/hdlc/>

Compile `sethdlc.c` utility:

```
gcc -O2 -Wall -o sethdlc sethdlc.c
```

Make sure you’re using a correct version of `sethdlc` for your kernel.

Use `sethdlc` to set physical interface, clock rate, HDLC mode used, and add any required PVCs if using Frame Relay. Usually you want something like:

```
sethdlc hdlc0 clock int rate 128000
sethdlc hdlc0 cisco interval 10 timeout 25
```

or:

```
sethdlc hdlc0 rs232 clock ext
sethdlc hdlc0 fr lmi ansi
sethdlc 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.

47.1 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/hdlc/>.

GENERIC NETLINK

A wiki document on how to use Generic Netlink can be found here:

- http://www.linuxfoundation.org/collaborate/workgroups/networking/generic_netlink_howto

GENERIC NETWORKING STATISTICS FOR NETLINK USERS

Statistic counters are grouped into structs:

Struct	TLV type	Description
<code>gnet_stats_basic</code>	<code>TCA_STATS_BASIC</code>	Basic statistics
<code>gnet_stats_rate_est</code>	<code>TCA_STATS_RATE_EST</code>	Rate estimator
<code>gnet_stats_queue</code>	<code>TCA_STATS_QUEUE</code>	Queue statistics
<code>none</code>	<code>TCA_STATS_APP</code>	Application specific

49.1 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;
```

49.2 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;
```

(continues on next page)

(continued from previous page)

```
        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;

        ...
    }
```

49.3 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,
↪ dump,
                                   TCA_XSTATS, &mystruct->lock, &
                                   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.

49.4 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.

49.5 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.

49.6 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.

50.1 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

50.2 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. PP-PoE 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>

50.3 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/traveling/ergw>

50.4 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/>

50.5 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.

50.6 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!

50.7 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-gprs](mailto:osmocom-net-gprs@lists.osmocom.org) mailing list 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>

50.8 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>

50.9 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.

50.10 Architectural Details

50.10.1 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 therefor 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!

50.11 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)

51.1 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](#).

51.2 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.

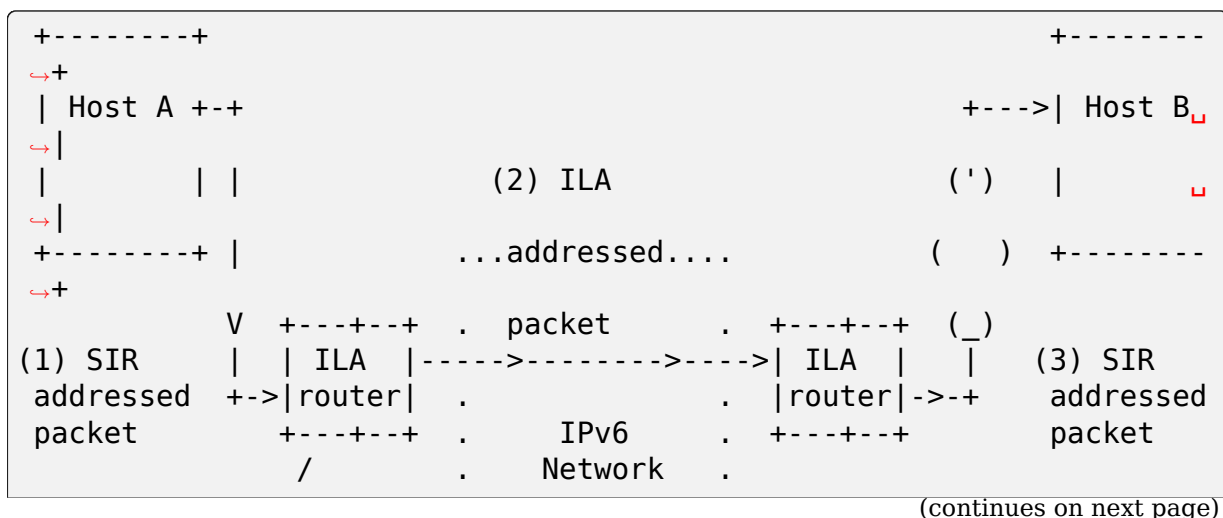
51.3 Operation

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:



(continued from previous page)



51.4 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.

51.5 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

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 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.

[illegible]

```

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                               Identifier                               |
|                               +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                               | Checksum-neutral adjustment |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

```

```

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|      |C|                               Identifier                    |
|      +-+                               +-+--+--+--+--+--+--+--+--+--+
|                                     |Checksum-neutral adjustment|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

```

Type	Identifier

Chapter 51. Identifier Locator Addressing (ILA)

```

+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Type|C|                                     Identifier                                     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     +-----+-----+-----+-----+-----+-----+
|                                     | Checksum-neutral adjustment |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

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 .

The usage of for ILA LWT routes is:

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.

```
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 translation. 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.

```
# 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 encap ila 2001:0:87:0 \
    csum-mode neutral-map ident-type use-format
```

1109

(continued from previous page)

```
# 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 encap 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
```


APPLETALK-IP DECAPSULATION AND APPLETALK-IP ENCAPSULATION

Documentation ipddp.c

This file is written by Jay Schulist <jschlst@samba.org>

52.1 Introduction

AppleTalk-IP (IPDDP) is the method computers connected to AppleTalk networks can use to communicate via IP. AppleTalk-IP is simply IP datagrams inside AppleTalk packets.

Through this driver you can either allow your Linux box to communicate IP over an AppleTalk network or you can provide IP gatewaying functions for your AppleTalk users.

You can currently encapsulate or decapsulate AppleTalk-IP on LocalTalk, EtherTalk and PPPTalk. The only limit on the protocol is that of what kernel AppleTalk layer and drivers are available.

Each mode requires its own user space software.

52.1.1 Compiling AppleTalk-IP Decapsulation/Encapsulation

AppleTalk-IP decapsulation needs to be compiled into your kernel. You will need to turn on AppleTalk-IP driver support. Then you will need to select ONE of the two options; IP to AppleTalk-IP encapsulation support or AppleTalk-IP to IP decapsulation support. If you compile the driver statically you will only be able to use the driver for the function you have enabled in the kernel. If you compile the driver as a module you can select what mode you want it to run in via a module loading param. `ipddp_mode=1` for AppleTalk-IP encapsulation and `ipddp_mode=2` for AppleTalk-IP to IP decapsulation.

52.1.2 Basic instructions for user space tools

I will briefly describe the operation of the tools, but you will need to consult the supporting documentation for each set of tools.

Decapsulation - You will need to download a software package called MacGate. In this distribution there will be a tool called MacRoute which enables you to add routes to the kernel for your Macs by hand. Also the tool MacRegGateWay is included to register the proper IP Gateway and IP addresses for your machine. Included in this distribution is a patch to netatalk-1.4b2+asun2.0a17.2 (available from <ftp.u.washington.edu/pub/user-supported/asun/>) this patch is optional but it allows automatic adding and deleting of routes for Macs. (Handy for locations with large Mac installations)

Encapsulation - You will need to download a software daemon called ipddpd. This software expects there to be an AppleTalk-IP gateway on the network. You will also need to add the proper routes to route your Linux box' s IP traffic out the ipddp interface.

52.2 Common Uses of ipddp.c

Of course AppleTalk-IP decapsulation and encapsulation, but specifically decapsulation is being used most for connecting LocalTalk networks to IP networks. Although it has been used on EtherTalk networks to allow Macs that are only able to tunnel IP over EtherTalk.

Encapsulation has been used to allow a Linux box stuck on a LocalTalk network to use IP. It should work equally well if you are stuck on an EtherTalk only network.

52.3 Further Assistance

You can contact me (Jay Schulist <jschlst@samba.org>) with any questions regarding decapsulation or encapsulation. Bradford W. Johnson <johns393@maroon.tc.umn.edu> originally wrote the ipddp.c driver for IP encapsulation in AppleTalk.

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
```


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.

55.1 /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 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 discovered Path MTU

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

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.

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 using 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

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.

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

55.2 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.

55.3 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

Count buffering overhead as $\text{bytes}/2^{\text{tcp_adv_win_scale}}$ (if $\text{tcp_adv_win_scale} > 0$) or $\text{bytes}-\text{bytes}/2^{-(\text{tcp_adv_win_scale})}$, 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 $\max(\text{window}/2^{\text{tcp_app_win}}, \text{mss})$ 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 reprobing 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 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 makes RACK' s reordering window static (<code>min_rtt/4</code>).

RACK disables RACK' s DUPACK threshold heuristic
--

Default: 0x1

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_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 syncookies 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.
0x20	(server)	accept data-in-SYN w/o any cookie option present.
0x40	(server)	enable all listeners to support Fast Open by default without explicit <code>TCP_FASTOPEN</code> 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-xxxxxxx-xxxxxxx-xxxxxxx. 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 63seconds 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 127seconds.

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_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_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_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 POLL-OUT 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) Default: 1000

tcp_rx_skb_cache - BOOLEAN

Controls a per TCP socket cache of one skb, that might help performance of some workloads. This might be dangerous on systems with a lot of TCP sockets, since it increases memory usage.

Default: 0 (disabled)

55.4 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: Below this number of pages UDP is not bothered about its memory appetite. When amount of memory allocated by UDP exceeds this number, UDP 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 UDP sockets.

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

Minimal size of send buffer used by UDP sockets in moderation. Each UDP socket is able to use the size for sending data, even if total pages of UDP sockets exceed `udp_mem` pressure. The unit is byte.

Default: 4K

55.5 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)

55.6 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 else where 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

55.7 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.

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 4294967295” would enable it for the world, “100 4294967295” 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_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 network many 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.

$M = 65536 - \text{sizeof}(\text{ip header}) / (\text{sizeof}(\text{Group record}))$

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).

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.

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 - BOOLEAN

Define behavior for gratuitous ARP frames who's IP is 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

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.

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)

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

55.8 /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 IPV6_AUTOFLOWLABEL socket option
2	automatic flow labels are allowed, they may be enabled on a per socket basis using the IPV6_AUTOFLOWLABEL 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. 0-0x7FFFF is reserved for the IPv6 flow manager facility, 0x80000-0xFFFFF 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

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)

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.

IPv6 Segment Routing:

seg6_flowlabel - INTEGER

Controls the behaviour of computing the flowlabel of outer IPv6 header in case of SR Tencaps

-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.

conf/default/*:

Change the interface-specific default settings.

conf/all/*:

Change all the interface-specific settings.

[XXX: Other special features than forwarding?]

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.

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.

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.

Default: 172800 (2 days)

temp_prefered_lft - INTEGER

Preferred lifetime (in seconds) for temporary addresses.

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 be 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_llao - 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)

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 `addrngenmode 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 <code>stable_secret</code> (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.

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

55.9 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

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 <yoshfuji@linux-ipv6.org>

55.10 /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

55.11 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

prsrctp_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_CRYPTOD_MD5 and CONFIG_CRYPTOD_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

55.12 /proc/sys/net/core/*

Please see: Documentation/admin-guide/sysctl/net.rst for descriptions of these entries.

55.13 /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>

57.1 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 it's slave devices. I have developed this driver in conjunction with network namespaces and not sure if there is use case outside of it.

57.2 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).

57.3 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 ipv10 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
```

57.4 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.

57.4.1 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.

57.4.2 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.

57.4.3 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.

57.5 5. Mode flags:

At this time following mode flags are available

57.5.1 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.

57.5.2 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.

57.5.3 5.3 vepa:

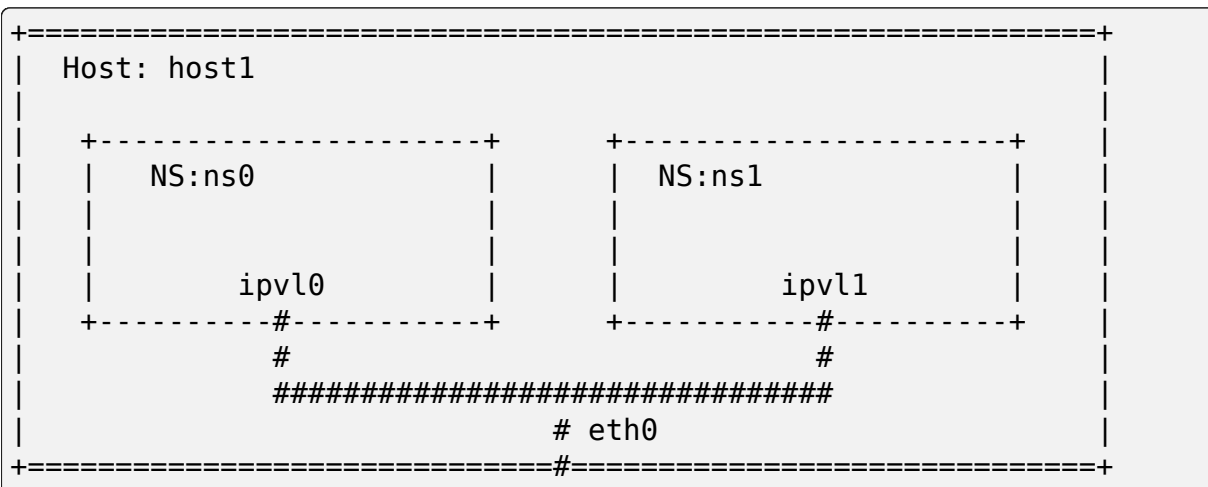
If this is added to the command-line, the port is set in VEPA mode. i.e. port will offload switching functionality to the external entity as described in 802.1Qbg Note: VEPA mode in IPvlan has limitations. IPvlan uses the mac-address of the master-device, so the packets which are emitted in this mode for the adjacent neighbor will have source and destination mac same. This will make the switch / router send the redirect message.

57.6 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 ipvlan:

- (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.

57.7 6. Example configuration:



- (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

58.1 /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.

conntrack - 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`.

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 real servers 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.

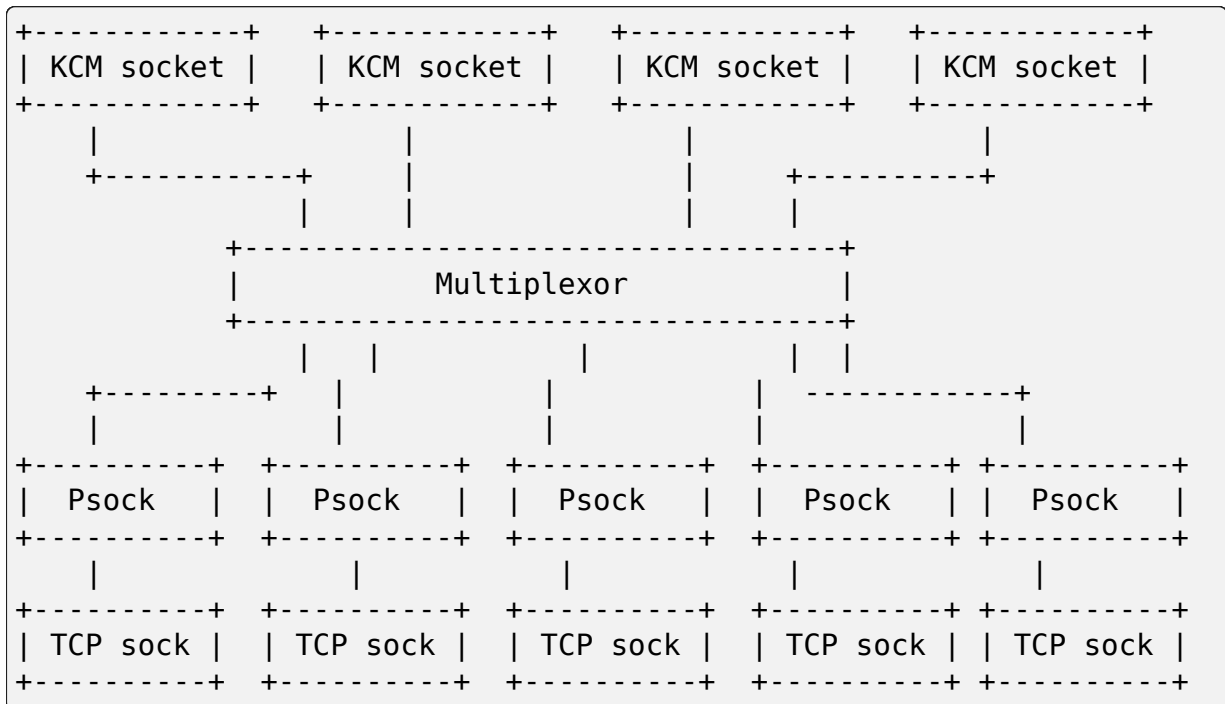
1 selects the current synchronisation protocol (version 1). This should be used where possible.

Kernels with this `sync_version` entry are able to receive messages of both version 1 and version 2 of the synchronisation protocol.

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:



59.1 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.

59.2 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.

59.3 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.

59.4 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 recvmmsg) can be used to send and receive messages from the KCM socket.

59.5 Socket types

KCM supports SOCK_DGRAM and SOCK_SEQPACKET socket types.

59.5.1 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.

59.5.2 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.

59.6 User interface

59.6.1 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`

59.6.2 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);
```

(continues on next page)

(continued from previous page)

```
if (!err)
    newkcmfd = info.fd;
```

59.6.3 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

59.6.4 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);
```

59.6.5 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))
```

59.6.6 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";
```

59.7 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.

59.7.1 Configurations

In an Nx1 configuration, KCM logically provides multiple socket handles to the same TCP connection. This allows parallelism between in 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.

59.7.2 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.

59.7.3 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).

59.7.4 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.

60.1 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.

60.2 L2TP APIs

This section documents each userspace API of the L2TP subsystem.

60.2.1 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.

60.2.2 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	Re-quired	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</code>	N	Zero IPv6 UDP checksum on transmit. Used only if FD is not set.
<code>UDP_ZERO_CSUM6</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	Re- quired	Use
CONN_ID	N	Identifies the tunnel id to be queried. Ignored in DUMP requests.

L2TP_CMD_SESSION_CREATE attributes:-

Attribute	Re- quired	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	Re- quire	Use
CONN	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	Require	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.
IF_NAME	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	Require	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.
IF_NAME	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;
```

(continues on next page)

(continued from previous page)

```
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, encap);
```

- 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;
```

(continues on next page)

(continued from previous page)

```
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 nlmsg_hdr *nlh;
struct genlmsg_hdr *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);
```

60.2.3 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 L2TP/IP 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_PROTO0_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 / L2TP/IP socket.
 */
fd = connect(session_fd, (struct sockaddr *)&sax, sizeof(sax));
if (fd < 0 ) {
```

(continues on next page)

(continued from previous page)

```
        return -errno;
    }
    return 0;
```

60.2.4 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.

60.2.5 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.

60.2.6 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.

60.3 Internal Implementation

This section is for kernel developers and maintainers.

60.3.1 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_rcv` 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`.

60.3.2 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 socket's `sk_put` is 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.

60.3.3 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.

60.3.4 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](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=3d609342cc04129ff7568e19316ce3d7451a27e8).

60.3.5 Ethernet

`net/l2tp/l2tp_eth.c` implements L2TPv3 Ethernet pseudowires. It manages a `net_dev` 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.

60.4 Miscellaneous

60.4.1 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

60.4.2 Implementations

A number of open source applications use the L2TP kernel subsystem:

<code>iproute2</code>	https://github.com/shemminger/iproute2
<code>go-l2tp</code>	https://github.com/katalix/go-l2tp
<code>tunneldigger</code>	https://github.com/wlanslovenija/tunneldigger
<code>xl2tpd</code>	https://github.com/xelerance/xl2tpd

60.4.3 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.

60.4.4 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.

61.1 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.

61.2 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.

61.3 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.

61.4 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.

61.5 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:

<code>LAPB_OK</code>	The LAPB link was terminated normally.
<code>LAPB_NOTCONNECTED</code>	The remote system was not connected.
<code>LAPB_TIMEDOUT</code>	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:

<code>LAPB_OK</code>	The LAPB link was terminated normally by the remote system.
<code>LAPB_REFUSED</code>	The remote system refused the connect request.
<code>LAPB_NOTCONNECTED</code>	The remote system was not connected.
<code>LAPB_TIMEDOUT</code>	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 *./How to use radiotap headers*.

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_FCS will be removed and recalculated
IEEE80211_RADIOTAP_FRAME will be encrypted if key available
IEEE80211_RADIOTAP_FRAME will be fragmented if longer than the current fragmentation threshold.

- IEEE80211_RADIOTAP_TX_FLAGS

IEEE80211_RADIOTAP_I 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>

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

HOWTO FOR MULTIQUEUE NETWORK DEVICE SUPPORT

64.1 Section 1: Base driver requirements for implementing multiqueue support

64.1.1 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.).

64.2 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.

64.3 Section 3: Brief howto using MULTIQ for multi-queue 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>

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

Please send bug reports to Matt Mackall <mpm@selenic.com> Satyam Sharma <satyam.sharma@gmail.com>, and Cong Wang <xiyou.wangcong@gmail.com>

65.1 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.

65.2 Sender and receiver configuration:

It takes a string configuration parameter “netconsole” in the following format:

```
netconsole=[+][src-port]@[src-ip]/[<dev>],[tgt-port]@<tgt-ip>/[tgt-  
↪macaddr]
```

where

+	if present, enable extended console support
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 -p <port>' / 'nc -u -l <port>
or::
netcat -u -l -p <port>' / 'netcat -u -l <port>
```

- 3) socat

```
socat udp-recv:<port> -
```

65.3 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. [Note that the parameters of netconsole targets that were specified/created from the boot/module option are not exposed via this interface, and hence cannot be modified dynamically.]

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` mount-point).

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)
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).

65.4 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>,<sequnum>,<timestamp>,<contflag>;<message text>
```

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.
```

65.5 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.qmqm.pl>

66.1 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.

66.2 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.

66.3 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.)

66.4 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.

NETWORK DEVICES, THE KERNEL, AND YOU!

67.1 Introduction

The following is a random collection of documentation regarding network devices.

67.2 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: `rmmod 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()`.

67.2.1 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(...);
```

(continues on next page)

(continued from previous page)

```
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.

67.2.2 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;
}
```

(continues on next page)

(continued from previous page)

```

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;

```

(continues on next page)

(continued from previous page)

```
}
```

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`.

67.3 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.

67.4 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

ndo_get_stats:

Synchronization: `dev_base_lock` rwlock. Context: nominally process, but don't sleep inside an rwlock

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

67.5 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

68.1 /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.

69.1 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

NETFILTER CONNTRACK SYSFS VARIABLES

70.1 /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 but the hash table will never have fewer than 32 and limited to 16384 buckets. For systems with more than 4GB of memory it will be 65536 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
- not 0 - enabled (default)

If this option is enabled, the connection tracking code will provide userspace with connection tracking events via ctnetlink.

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_helper` - BOOLEAN

- 0 - disabled (default)
- not 0 - enabled

Enable automatic conntrack helper assignment. If disabled it is required to set up iptables rules to assign helpers to connections. See the CT target description in the `iptables-extensions(8)` man page for further information.

`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

Size of connection tracking table. Default value is `nf_conntrack_buckets` value * 4.

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_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_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.

NETFILTER' S FLOWTABLE INFRASTRUCTURE

This documentation describes the software flowtable infrastructure available in Netfilter since Linux kernel 4.16.

71.1 Overview

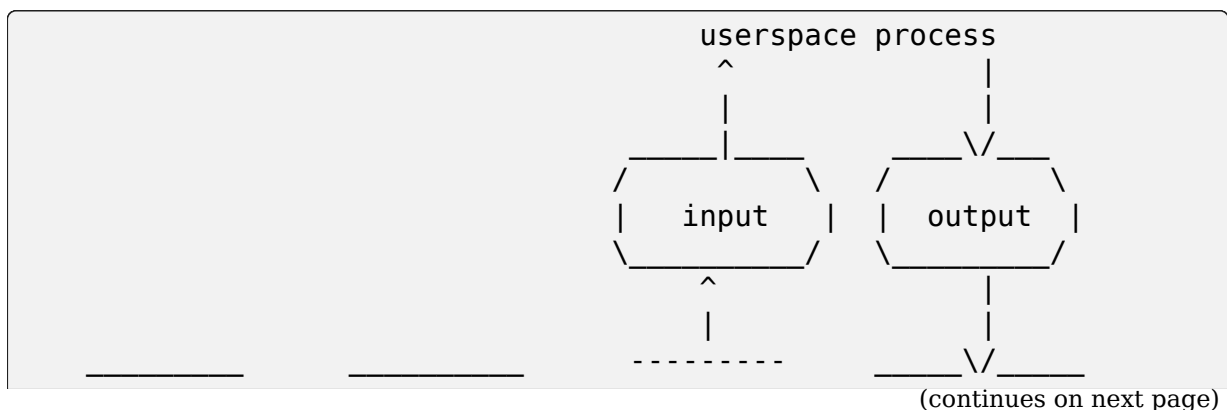
Initial packets follow the classic forwarding path, once the flow enters the established state according to the conntrack semantics (ie. we have seen traffic in both directions), then you can decide to offload the flow to the flowtable from the forward chain via the 'flow offload' action available in nftables.

Packets that find an entry in the flowtable (ie. flowtable hit) are sent to the output netdevice via `neigh_xmit()`, hence, they bypass the classic forwarding path (the visible effect is that you do not see these packets from any of the netfilter hooks coming after the ingress). In case of flowtable miss, the packet follows the classic forward path.

The flowtable uses a resizable hashtable, lookups are based on the following 7-tuple selectors: source, destination, layer 3 and layer 4 protocols, source and destination ports and the input interface (useful in case there are several conntrack zones in place).

Flowtables are populated via the 'flow offload' nftables action, so the user can selectively specify what flows are placed into the flow table. Hence, packets follow the classic forwarding path unless the user explicitly instruct packets to use this new alternative forwarding path via nftables policy.

This is represented in Fig.1, which describes the classic forwarding path including the Netfilter hooks and the flowtable fastpath bypass.



(continued from previous page)

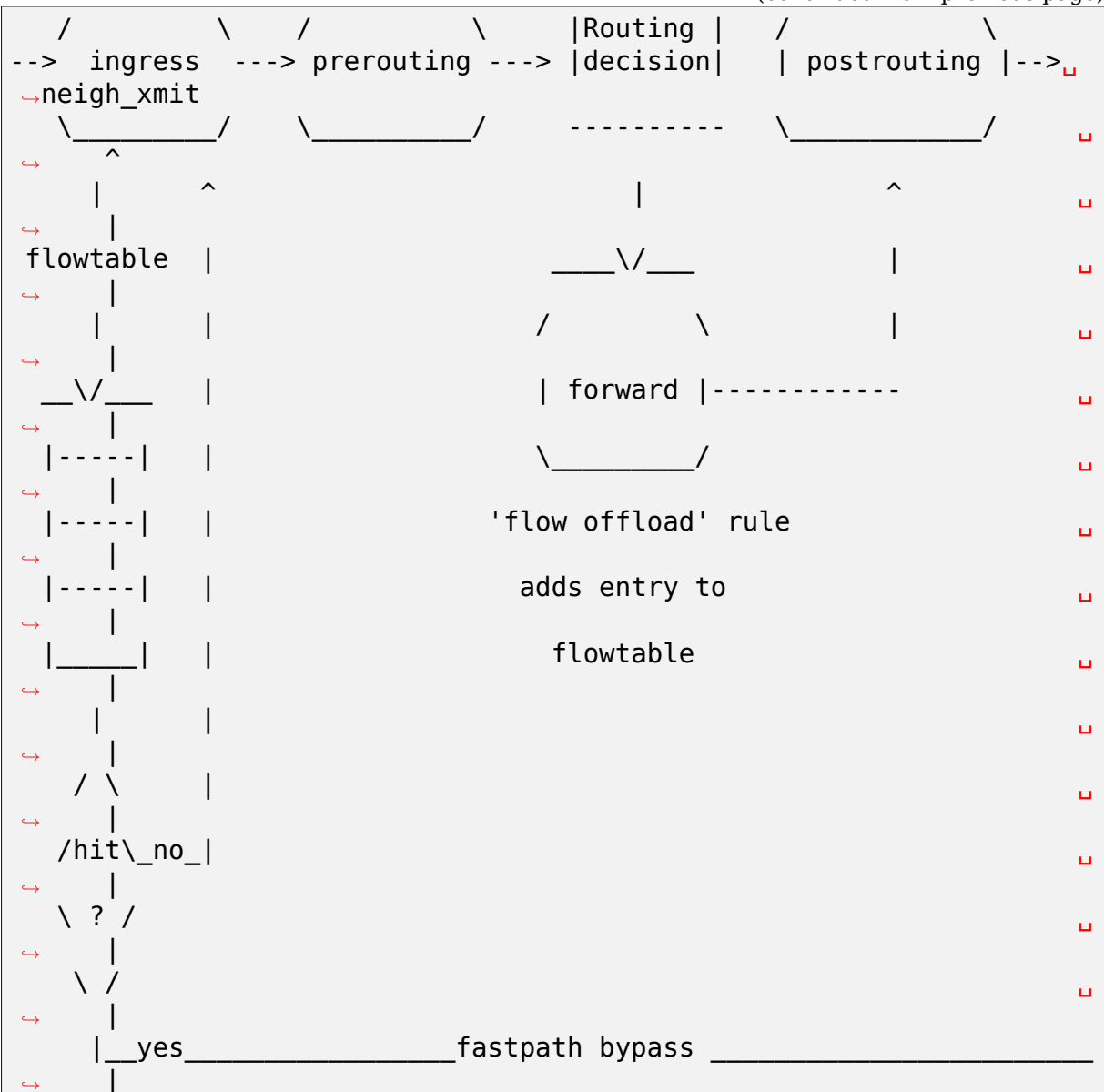


Fig.1 Netfilter hooks and flowtable interactions

The flowtable entry also stores the NAT configuration, so all packets are mangled according to the NAT policy that matches the initial packets that went through the classic forwarding path. The TTL is decremented before calling `neigh_xmit()`. Fragmented traffic is passed up to follow the classic forwarding path given that the transport selectors are missing, therefore flowtable lookup is not possible.

71.2 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 offload @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.

71.3 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).

72.1 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.

72.2 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(...)
```

72.3 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 wildcarded flows is undefined. It is the responsibility of the user space program to ensure that any incoming packet can match at most one flow, wildcarded or not. The current implementation performs best-effort detection of overlapping wildcarded flows and may reject some but not all of them. However, this behavior may change in future versions.

72.4 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.

72.5 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 im-

plement 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), encap(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.)

72.6 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.

72.7 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

73.1 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.

73.2 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()*

73.2.1 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):

Unused in current kernel.

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.

73.2.2 TLV IFLA_LINKMODE

contains link policy. This is needed for userspace interaction described below.

This TLV can also be queried via sysfs.

73.3 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 TVL 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_NOCARRIER` 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_LOWERLAYERDOWN 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.

73.4 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

74.1 Abstract

This file documents the `mmap()` facility available with the `PACKET` socket interface on 2.4/2.6/3.x kernels. 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

74.2 Why use `PACKET_MMAP`

In Linux 2.4/2.6/3.x if `PACKET_MMAP` is not enabled, the capture process 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).

In 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.

74.3 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

74.4 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.

74.5 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 interface
               mmap() -----> mapping of the allocated buffer_
↳ to the      user process

[transmission] poll() -----> wait for free packets (optional)
↳ ready in    send() -----> send all packets that are set as_
               the ring
↳ to return    The flag MSG_DONTWAIT can be used_
               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.
```

(continues on next page)

(continued from previous page)

```
.  
-----  
  
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 (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/↳packetmmap/>

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.

74.6 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 to 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:

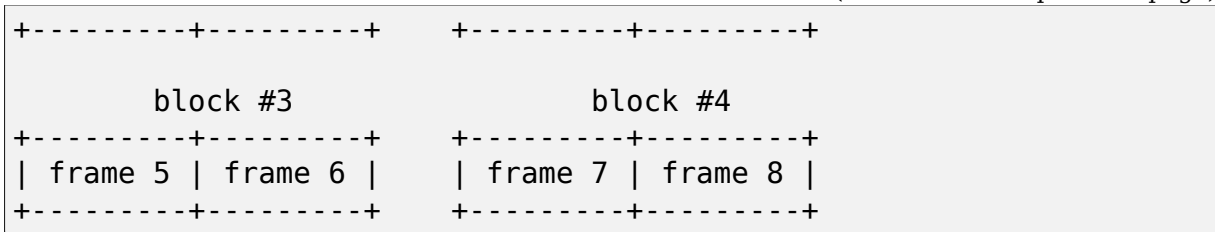
```
tp_block_size= 4096
tp_frame_size= 2048
tp_block_nr   = 4
tp_frame_nr   = 8
```

we will get the following buffer structure:

block #1		block #2	
+	+	+	+
frame 1	frame 2	frame 3	frame 4

(continues on next page)

(continued from previous page)



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)”

74.7 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. For information on these kernel versions see http://pusa.uv.es/~ulisses/packet_mmap/packet_mmap.pre-2.4.26_2.6.5.txt

74.7.1 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_ORDER` macro. More precisely the limit can be calculated as:

```
PAGE_SIZE << MAX_ORDER
```

```
In a i386 architecture PAGE_SIZE is 4096 bytes
```

```
In a 2.4/i386 kernel MAX_ORDER is 10
```

```
In a 2.6/i386 kernel MAX_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_ORDER` declarations.

The `pagesize` can also be determined dynamically with the `getpagesize(2)` system call.

74.7.2 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:

```
+---+---+---+---+
| x | x | x | x |
+---+---+---+---+
|   |   |   |   |
|   |   |   | v
|   |   | v  block #4
|   | v  block #3
| v  block #2
v  block #1
```

`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 pre-determined 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:

$$131072/4 = 32768 \text{ blocks}$$

74.8 PACKET_MMAP buffer size calculator

Definitions:

<size-max>	is the maximum size of allocable with <code>kmalloc</code> (see <code>/proc/slabinfo</code>)
<pointer size>	depends on the architecture - <code>sizeof(void *)</code>
<page size>	depends on the architecture - <code>PAGE_SIZE</code> or <code>getpagesize (2)</code>
<max-order>	is the value defined with <code>MAX_ORDER</code>
<frame size>	it's an upper bound of frame's capture size (more on this later)

from these definitions we will derive:

$$\begin{aligned} \text{<block number>} &= \text{<size-max>/<pointer size>} \\ \text{<block size>} &= \text{<pagesize> << <max-order>} \end{aligned}$$

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.

74.8.1 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.

74.8.2 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 discontinuous 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 spawn 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_ST 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 to 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_ST indicates there were packet drops from last time statistics where checked with `getsockopt()` and the `PACKET_STATISTICS` option.

TP_ST 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_ST 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);
```

74.9 What TPACKET versions are available and when to use them?

```
int val = tpacket_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 tpacket2_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 tpacket_hdr by struct tpacket2_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 tpacket_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_rhash
- RX Hash data available in user space
- TX_RING semantics are conceptually similar to TPACKET_V2; use tpacket3_hdr instead of tpacket2_hdr, and TPACKET3_HDRLEN instead of TPACKET2_HDRLEN. In the current implementation, the tp_next_offset field in the tpacket3_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.

74.10 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;

#ifdef 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;
```

(continues on next page)

(continued from previous page)

```
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) {
```

(continues on next page)

(continued from previous page)

```

        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);
        }
    }
}

```

(continues on next page)

(continued from previous page)

```
    for (i = 0; i < 4; i++) {
        int status;

        wait(&status);
    }

    return 0;
}
```

74.11 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>
```

(continues on next page)

(continued from previous page)

```

#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) {

```

(continues on next page)

(continued from previous page)

```

        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, 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_pkttype = 0;
    ll.sll_halen = 0;

    err = bind(fd, (struct sockaddr *) &ll, sizeof(ll));
    if (err < 0) {
        perror("bind");
        exit(1);
    }

```

(continues on next page)

(continued from previous page)

```

    }

    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 sbuff[NI_MAXHOST], dbuff[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),
↪sbuff, sizeof(sbuff), 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),
↪dbuff, sizeof(dbuff), NULL, 0, NI_
↪NUMERICHOST);

        printf("%s -> %s, ", sbuff, dbuff);
    }

    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 +

```

(continues on next page)

(continued from previous page)

```

        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;

```

(continues on next page)

(continued from previous page)

```

        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;
}

```

74.12 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.

74.13 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.

74.14 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)*

74.15 THANKS

Jesse Brandeburg, for fixing my grammatical/spelling errors

LINUX PHONET PROTOCOL FAMILY

75.1 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.

75.2 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.

75.3 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.

75.4 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()`.

75.5 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;
```

(continues on next page)

(continued from previous page)

```
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).

75.6 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.

75.7 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);

/* ... */
cfd = accept(lfd, NULL, NULL);
for (;;)
{
    char buf[...];
    ssize_t len = read(cfd, buf, sizeof(buf));

    /* ... */
}
```

(continues on next page)

(continued from previous page)

```
write(cfd, 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.

75.8 Authors

Linux Phonet was initially written by Sakari Ailus.

Other contributors include Mikä Liljeberg, Andras Domokos, Carlos Chinae 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
```

76.1 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 ($1/30 \cdot 10^6$):

```
# ethtool -C ethX rx-usecs 30
```

76.2 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: OK: 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.

76.3 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
```

(continues on next page)

(continued from previous page)

```
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
```

76.4 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
→stopped.	for continuous sends until explicitly
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

(continues on next page)

(continued from previous page)

```

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,
↳CONFIG_XFRM
                                IPSEC # IPsec encapsulation (needs
↳allocation
                                NODE_ALLOC # node specific memory
                                NO_TIMESTAMP # disable timestamping

pgset 'flag ![name]'    Clear a flag to determine behaviour.
↳quote in
                                Note that you might need to use single
↳'t expand
                                interactive mode, so that your shell wouldn
                                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.

```

(continues on next page)

(continued from previous page)

```

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"_

```

(continues on next page)

(continued from previous page)

```

→for AF11 no 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".

```

76.5 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 : ($DEST_IP)      destination IP
-m : ($DST_MAC)      destination MAC-addr
-t : ($THREADS)      threads to start
-c : ($SKB_CLONE)    SKB clones send before alloc new SKB
-b : ($BURST)        HW level bursting of SKBs
-v : ($VERBOSE)      verbose
-x : ($DEBUG)        debug

```

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.

The old scripts:

```

pktgen.conf-1-2          # 1 CPU 2 dev
pktgen.conf-1-1-rdos     # 1 CPU 1 dev w. route DoS
pktgen.conf-1-1-ipv6     # 1 CPU 1 dev ipv6
pktgen.conf-1-1-ipv6-rdos # 1 CPU 1 dev ipv6 w. route DoS
pktgen.conf-1-1-flows    # 1 CPU 1 dev multiple flows.

```

76.6 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()`).

76.7 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.

76.8 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
```

(continues on next page)

(continued from previous page)

```
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
UDPDEST_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

spi (ipsec)

dst_min
dst_max

src_min
src_max

dst_mac
src_mac
```

(continues on next page)

(continued from previous page)

```
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/>
- <tp://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)

77.1 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.

77.1.1 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.]

77.1.2 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.]

77.1.3 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).

77.1.4 Performance

PLIP easily outperforms Ethernet cards... (ups, I was dreaming, but it *is* getting late. EOB)

77.2 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.

77.2.1 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 $10^6/\text{HZ}$ 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 $10^6/\text{HZ}$ 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.

77.3 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.

77.3.1 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.

77.3.2 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.

77.3.3 PLIP Mode 0 transfer protocol

The PLIP driver is compatible with the “Crynwr” 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.

78.1 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.

78.2 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.

78.3 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.

78.4 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:

- PPPIOCCONNECT 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.
- 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:

SC_COMP_TCP	enable transmit TCP header compression
SC_NO_TCP_C	disable connection-id compression for TCP header compression
SC_REJ_COMP	disable receive TCP header decompression
SC_CCP_OPEN	Compression Control Protocol (CCP) is open, so inspect CCP packets
SC_CCP_UP	CCP is up, may (de)compress packets
SC_LOOP_TRA	send IP traffic to pppd
SC_MULTILINK	enable PPP multilink fragmentation on transmitted packets
SC_MP_SHORT	expect short multilink sequence numbers on received multilink fragments
SC_MP_XSHOF	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:

SC_COMP_RUN	CCP compressor is running
SC_DECOMP_RUN	CCP decompressor is running
SC_DC_ERROR	CCP decompressor detected non-fatal error
SC_DC_FERROR	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-ioct.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-ioct.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 `ioctl`s 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.
- `PPPIOCSMRRU` 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
```

(continues on next page)

(continued from previous page)

						(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

80.1 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.

80.2 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.

80.3 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 bitwise 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.

80.4 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
```


80.5 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) {
            /*
↪this_arg
↪ Use
↪dereference
↪arches.
            * You must take care when dereferencing iterator.
            * for multibyte types... the pointer is not aligned.
            * get_unaligned((type *)iterator.this_arg) to
↪
            * iterator.this_arg for type "type" safely on all
↪
            */
            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:
```

(continues on next page)

(continued from previous page)

```
                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>

== RDS ==

CHAPTER EIGHTYONE

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>

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 zerocopy 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 assignment 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`

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 `msg_hdr`. The format of the messages is described in man-pages.

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`.

RDMA FOR RDS

see rds-rdma(7) manpage (available in rds-tools)

CONGESTION NOTIFICATIONS

see rds(7) manpage

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 socket's `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.

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, loop-back, stats, usermem 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.

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 rbnod, waitq, etc.

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.

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

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_recvmmsg

- copy data into user iovec
- handle CMSGs
- return to application

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 private 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>

93.1 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.

93.2 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.

93.3 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.

93.4 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.

93.5 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);
```

93.6 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 `kmalloc()` 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.

Bellow 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],
```

(continues on next page)

(continued from previous page)

```
        sizeof(struct ieee80211_reg_rule));  
regulatory_struct_hint(rd);
```

93.7 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.

RXRPC NETWORK PROTOCOL

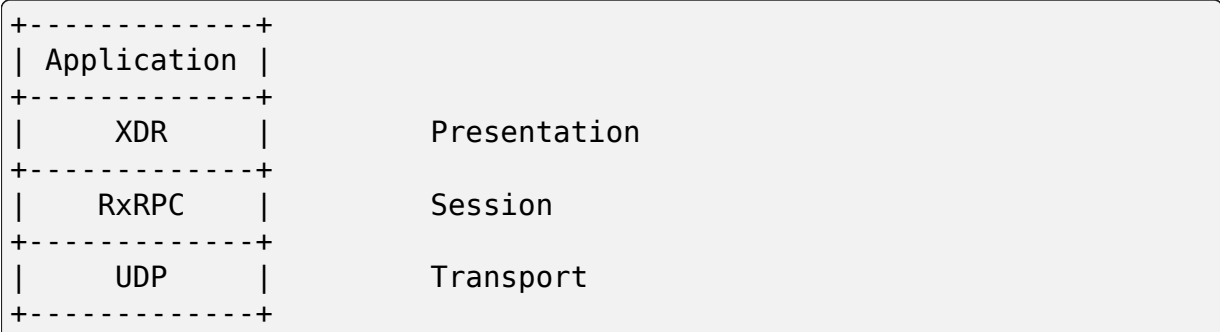
The RxRPC protocol driver provides a reliable two-phase transport on top of UDP that can be used to perform RxRPC remote operations. This is done over sockets of AF_RXRPC family, using `sendmsg()` and `recvmsg()` with control data to send and receive data, aborts and errors.

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.

94.1 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.

94.2 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	}	

(continues on next page)

(continued from previous page)

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.

94.3 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 bound 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 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 $2 \times \text{RTT}$ 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.

94.4 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_SESSION	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_RXXRPC sockets support a few socket options at the SOL_RXXRPC level:

(1) **RXXRPC_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) **RXXRPC_SECURITY_KEYRING**

Similar to above but specifies a keyring of server secret keys to use (key type "keyring"). See the "Security" section.

(3) **RXXRPC_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) **RXXRPC_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) **RXXRPC_SECURITY_PLAIN**

Encrypted checksum only.

(b) **RXXRPC_SECURITY_AUTH**

Encrypted checksum plus packet padded and first eight bytes of packet encrypted - which includes the actual packet length.

(c) **RXXRPC_SECURITY_ENCRYPT**

Encrypted checksum plus entire packet padded and encrypted, including actual packet length.

(5) **RXXRPC_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 kserver 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:

<code>RXRPC_USER_CALL_ID</code> 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).

97.1 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:

<code>RXRPC_USER_CALL_ID</code>	specifies the user ID for this call
<code>RXRPC_ABORT</code>	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.

97.2 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,
```

(continues on next page)

(continued from previous page)

```
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) End a client call:

```
void rxrpc_kernel_end_call(struct socket *sock,  
                          struct rxrpc_call *call);
```

This is used to end 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) Send data through a call:

```
typedef void (*rxrpc_notify_end_tx_t)(struct sock *sk,  
                                     unsigned long user_call_  
→ ID,  
                                     struct sk_buff *skb);
```

(continues on next page)

(continued from previous page)

```
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 the call-state spinlock held to prevent any reply or final ACK from being delivered first.

(4) 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

(continues on next page)

(continued from previous page)

↪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.

(5) 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.

(6) 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.

(7) 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.

(8) 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).

(9) Allocate a null key for doing anonymous security:

```
struct key *rxrpc_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.

- (10) Get the peer address of a call:

```
void rxrpc_kernel_get_peer(struct socket *sock, struct rxrpc_
    ↪call *call,
                          struct sockaddr_rxrpc *_srx);
```

This is used to find the remote peer address of a call.

- (11) Set the total transmit data size on a call:

```
void rxrpc_kernel_set_tx_length(struct socket *sock,
                               struct rxrpc_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.

- (12) Get call RTT:

```
u64 rxrpc_kernel_get_rtt(struct socket *sock, struct rxrpc_call
    ↪*call);
```

Get the RTT time to the peer in use by a call. The value returned is in nanoseconds.

- (13) Check call still alive:

```
bool rxrpc_kernel_check_life(struct socket *sock,
                             struct rxrpc_call *call,
                             u32 *_life);
void rxrpc_kernel_probe_life(struct socket *sock,
                             struct rxrpc_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.

- (14) Get reply timestamp:

```
bool rxrpc_kernel_get_reply_time(struct socket *sock,
                                struct rxrpc_call *call,
                                ktime_t *_ts)
```

This allows the timestamp on the first DATA packet of the reply of a client call to be queried, provided that it is still in the Rx ring. If successful, the timestamp will be stored into *_ts and true will be returned; false will be returned otherwise.

- (15) Get remote client epoch:

```
u32 rxrpc_kernel_get_epoch(struct socket *sock,
                           struct rxrpc_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.

97.3 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 evolvement 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.

98.1 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>](mailto: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

100.1 /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.

INTERFACE STATISTICS

101.1 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`.

101.1.1 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 ens4u1u1
6: ens4u1u1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_
→codel state UP mode DEFAULT group default qlen 1000
  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 enp58s0u1u1
```

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.

101.1.2 Protocol-specific statistics

Some of the interfaces used for configuring devices are also able to report related statistics. 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
```

101.1.3 Driver-defined statistics

Driver-defined `ethtool` statistics can be dumped using `ethtool -S $ifc`, e.g.:

```
$ ethtool -S ens4u1u1
NIC statistics:
  tx_single_collisions: 0
  tx_multi_collisions: 0
```

101.2 uAPIs

101.2.1 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.

101.2.2 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*.

101.2.3 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).

101.2.4 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 *ETHTOOL_GSTRINGS* with *string_set* set to *ETH_SS_STATS*, and values via *ETHTOOL_GSTATS*. User space should use *ETHTOOL_GDRVINFO* to retrieve the number of statistics (*.n_stats*).

101.2.5 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*

101.2.6 debugfs

Some drivers expose extra statistics via *debugfs*.

101.3 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;
```

(continues on next page)

(continued from previous page)

```
__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;  
};
```

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 should not include packets dropped by the device 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).

101.4 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.

101.4.1 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()*

STREAM PARSER (STRPARSER)

102.1 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.

102.2 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 rcv_msg function that is called when a full message has been completed.

102.3 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`.

102.4 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
-	current message should not be processed by the kernel, return
ESTR	control of the socket to userspace which can proceed to read
	the messages itself
other	Error in parsing, give control back to userspace assuming that
< 0	synchronization is lost and the stream is unrecoverable (appli-
	cation 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 *strp)
```

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 *strp)
```

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 (*rcv_msg)(struct strparser *strp, struct sk_buff ↵
↵ *skb);
```

rcv_msg is called when a full message has been received and is queued. The callee must consume the sk_buff; it can call strp_pause to prevent any further messages from being received in rcv_msg (see strp_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 *strp, 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 strp_init) a default function is used.

::

```
void (*abort_parser)(struct strparser *strp, 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-NULL in strp_init.

102.5 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.

102.6 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`.

102.7 Author

Tom Herbert (tom@quantonium.net)

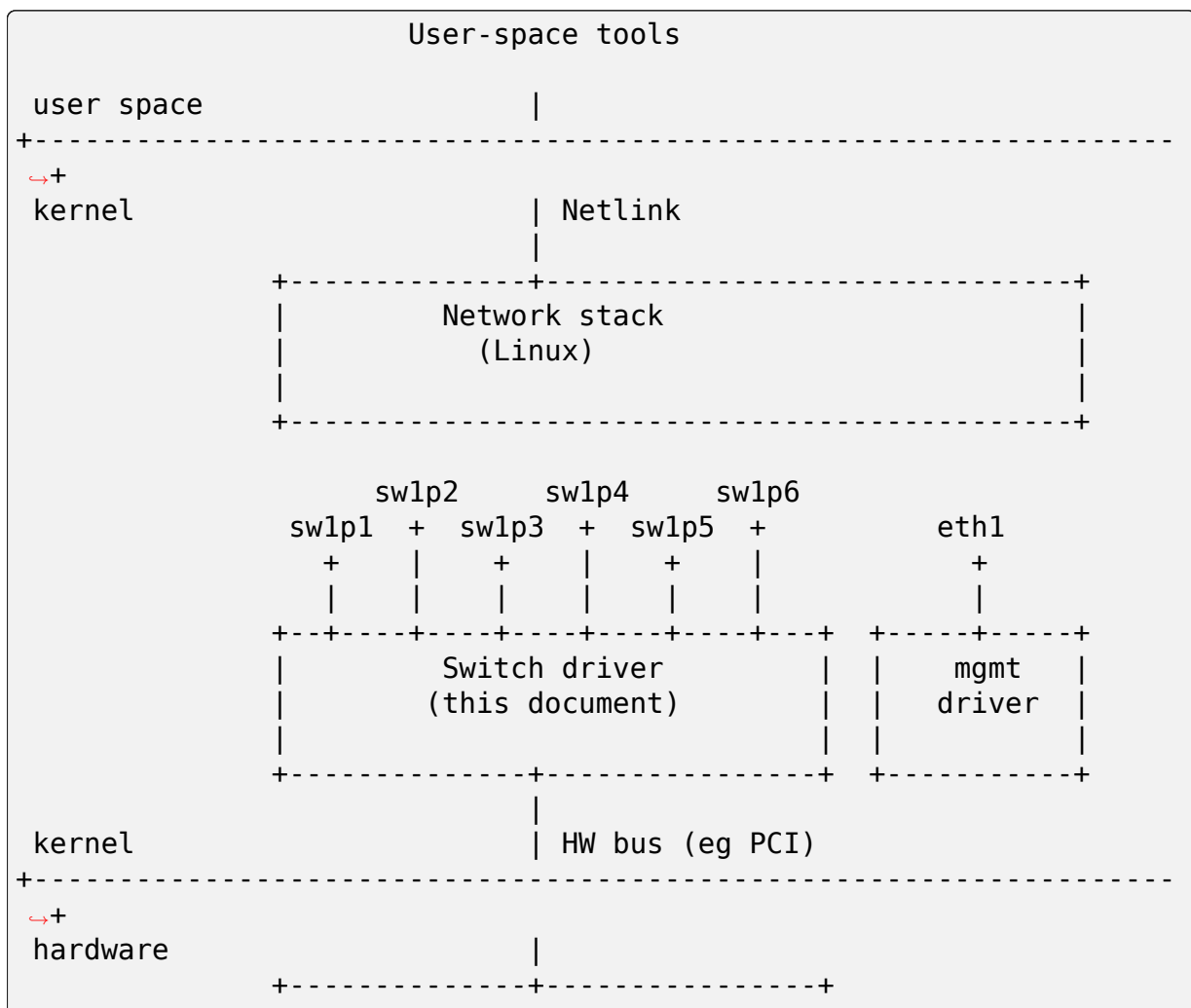
ETHERNET SWITCH DEVICE DRIVER MODEL (SWITCHDEV)

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.



(continues on next page)

(continued from previous page)

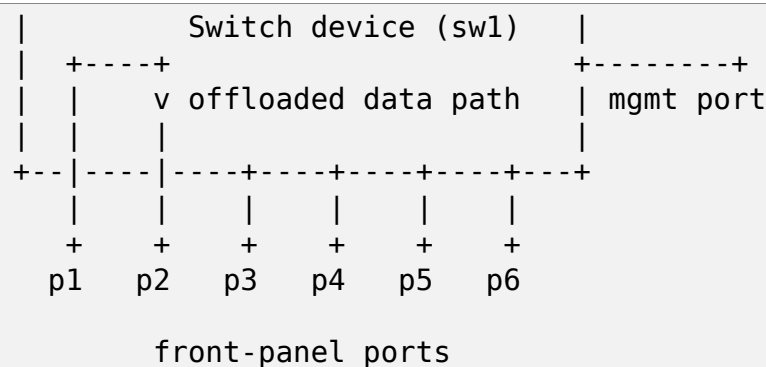


Fig 1.

103.1 Include Files

```
#include <linux/netdevice.h>
#include <net/switchdev.h>
```

103.2 Configuration

Use “depends NET_SWITCHDEV” in driver’s Kconfig to ensure switchdev model support is built for driver.

103.3 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.

103.3.1 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.

103.3.2 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.

103.3.3 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.

103.3.4 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.

103.4 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

103.4.1 Static FDB Entries

The switchdev driver should implement `ndo_fdb_add`, `ndo_fdb_del` and `ndo_fdb_dump` to support static FDB entries installed to the device. Static bridge FDB entries are installed, for example, using iproute2 bridge cmd:

```
bridge fdb add ADDR dev DEV [vlan VID] [self]
```

The driver should use the helper `switchdev_port_fdb_xxx ops` for `ndo_fdb_xxx ops`, and handle add/delete/dump of `SWITCHDEV_OBJ_ID_PORT_FDB` object using `switchdev_port_obj_xxx ops`.

XXX: what should be done if offloading this rule to hardware fails (for example, due to full capacity in hardware tables) ?

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
```

103.4.2 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 swlp1 master br0 permanent
00:02:00:00:02:00 dev swlp1 master br0 offload
00:02:00:00:02:00 dev swlp1 self
52:54:00:12:35:02 dev swlp2 master br0 permanent
00:02:00:00:03:00 dev swlp2 master br0 offload
00:02:00:00:03:00 dev swlp2 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`.

103.4.3 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).

103.4.4 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:xx link-local multicast packets can pass.

Note that STP BPDUs 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.

103.4.5 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 than 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.

103.4.6 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.

103.5 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 next hop(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_ENT</code>	used for both adding a new FIB entry to the device, or modifying an existing entry on the device.
<code>FIB_EVENT_ENT</code>	used for removing a FIB entry
<code>FIB_EVENT_RUL</code>	
<code>FIB_EVENT_RUL</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 next hops. `*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 swlp1 proto kernel scope link src 11.0.0.2
```

(continues on next page)

(continued from previous page)

```
↪offload
11.0.0.4/30 via 11.0.0.1 dev swlp1 proto zebra metric 20 offload
11.0.0.8/30 dev swlp2 proto kernel scope link src 11.0.0.10↪
↪offload
11.0.0.12/30 via 11.0.0.9 dev swlp2 proto zebra metric 20 offload
12.0.0.2 proto zebra metric 30 offload
    nexthop via 11.0.0.1 dev swlp1 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

103.5.1 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.

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_nstype` first mounts sysfs, a new superblock is created. It will be differentiated from other sysfs mounts by having its `s_fs_info->ns[kobj_nstype]` 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

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.

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` IOCTLS or the `tcp_thin_linear_timeouts` and `tcp_thin_dupack` sysctls. Both modifications are turned off by default.

106.1 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 SEVEN

TEAM

Team devices are driven from userspace via libteam library which is here:
<https://github.com/jpirko/libteam>

TIMESTAMPING

108.1 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.

108.1.1 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.

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.

108.1.2 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 acknowledgment. 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 re-ordered 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.

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_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.timestamp_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 nlattr) 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 incom-

ing 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));
```

108.1.3 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.

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.

108.2 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.

108.2.1 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_TSTAMP_*` to define the actual timestamp passed in `scm_timestamping`.

The `SCM_TSTAMP_*` types are 1:1 matches to the `SOF_TIMESTAMPING_*` control fields discussed previously, with one exception. For legacy reasons, `SCM_TSTAMP_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.

108.3 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_V2_EVENT, then it should generally always upscale HWTSTAMP_FILTER_V2_L2_SYNC_MESSAGE, and so forth, as HWTSTAMP_FILTER_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
     */
};
```


108.3.1 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_hwtstamp_tx()` with the original skb, the raw hardware time stamp. `skb_hwtstamp_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.

108.3.2 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_do_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 code, DSA provides for most of the infrastructure for timestamping already, in generic code: a BPF classifier (`ptp_classify_raw`) is used to identify PTP event messages (any other packets, including PTP general messages, are not timestamped), and provides two hooks to drivers:

- `.port_txtstamp()`: The driver is passed a clone of the timestampable skb to be transmitted, before actually transmitting it. Typically, a switch will have a PTP TX timestamp register (or sometimes a FIFO) where the timestamp becomes available. 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. Either way, in the `.port_txtstamp()` method, the driver only needs to save the clone for later use (when the timestamp becomes available). Each skb is annotated with a pointer to its clone, in `DSA_SKB_CB(skb) -> clone`, to ease the driver's job of keeping track of which clone belongs to which skb.

- `.port_rxtstamp()`: 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_ni()` 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_ts` 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_do_ioctl`, whether `phy_has_hwtstamp(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 skbs 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_do_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.

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`.

109.1 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));
```

(continues on next page)

(continued from previous page)

```
/* - 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

109.2 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.

109.3 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 following modules for nf_tables:

- NFT_SOCKET
- NFT_TPROXY

109.4 4. Application support

109.4.1 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>

110.1 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> :))

110.2 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.

110.3 3. Program interface

110.3.1 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 )
```

(continues on next page)

(continued from previous page)

```

    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(ifr.ifr_name, dev, IFNAMSIZ);

    if( (err = ioctl(fd, TUNSETIFF, (void *) &ifr)) < 0 ){
        close(fd);
        return err;
    }
    strcpy(dev, ifr.ifr_name);
    return fd;
}

```

110.3.2 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.

```

110.3.3 3.3 Multiqueue tuntap interface

From version 3.8, Linux supports multiqueue tuntap 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;

```

(continues on next page)

(continued from previous page)

```
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
 *        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));
```

(continues on next page)

(continued from previous page)

```
if (enable)
    ifr.ifr_flags = IFF_ATTACH_QUEUE;
else
    ifr.ifr_flags = IFF_DETACH_QUEUE;

return ioctl(fd, TUNSETQUEUE, (void *)&ifr);
}
```

110.4 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>

111.1 1. Applications

Several applications have been ported successfully to UDP-Lite. Ethereal (now called Wireshark) has UDP-Lite v4/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.

111.2 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.

111.3 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:

```
#define IPPROTO_UDPLITE      136
#define SOL_UDPLITE          136
#define UDPLITE_SEND_CSCOV   10
#define UDPLITE_RECV_CSCOV   11
```

Ready-made header files for various distros are in the UDP-Lite tarball.

111.4 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 `buffer_size` 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 </pre>			

(continues on next page)

(continued from previous page)

```
*****checksum coverage*****^
```

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.

111.5 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 InErrors).
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)• <code>xfrm4_policy_check()</code> 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).

111.6 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.

111.7 7. Maintainer Address

The UDP-Lite patch was developed at

University of Aberdeen Electronics Research Group Depart-
ment 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,
<william@erg.abdn.ac.uk>.

VIRTUAL ROUTING AND FORWARDING (VRF)

112.1 The VRF Device

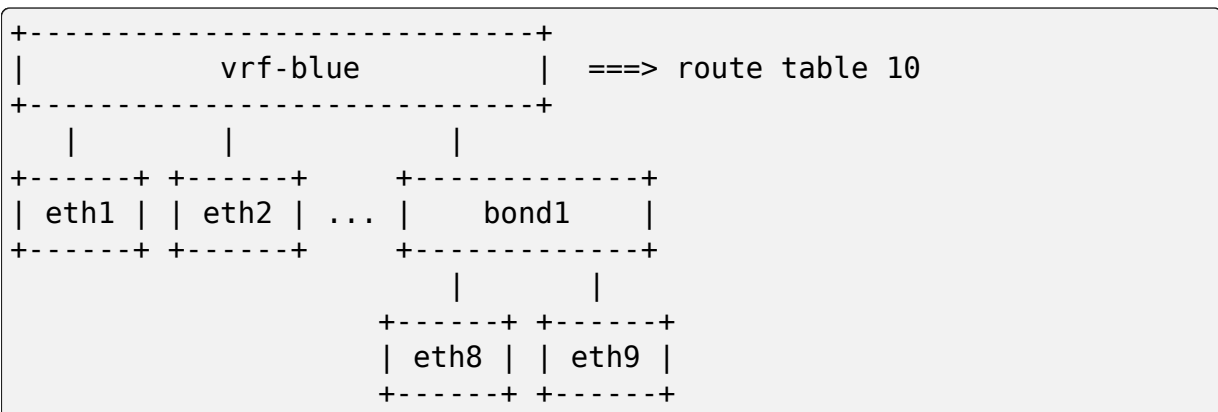
The VRF device combined with ip rules provides the ability to create virtual routing and forwarding domains (aka VRFs, VRF-lite to be specific) in the Linux network stack. One use case is the multi-tenancy problem where each tenant has their own unique routing tables and in the very least need different 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.

112.1.1 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.

112.1.2 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:

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.


```
ip route add table 10 ...
```

112.1.3 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.

112.2 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 addrngenmode 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 addrngenmode 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 addrngenmode 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 addrngenmode 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

```

(continues on next page)

(continued from previous page)

```
↪ 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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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 pre-dates 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
```

```
===== 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.

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.

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.

115.1 Possible Problems

(Henner Eisen, 2000-10-28)

The X.25 packet layer protocol depends on a reliable datalink service. The LAPB protocol provides such reliable service. But this reliability is not preserved by the Linux network device driver interface:

- With Linux 2.4.x (and above) SMP kernels, packet ordering is not preserved. Even if a device driver calls `netif_rx(skb1)` and later `netif_rx(skb2)`, `skb2` might be delivered to the network layer earlier than `skb1`.
- Data passed upstream by means of `netif_rx()` might be dropped by the kernel if the backlog queue is congested.

The X.25 packet layer protocol will detect this and reset the virtual call in question. But many upper layer protocols are not designed to handle such N-Reset events gracefully. And frequent N-Reset events will always degrade performance.

Thus, driver authors should make `netif_rx()` as reliable as possible:

SMP re-ordering will not occur if the driver's interrupt handler is always executed on the same CPU. Thus,

- Driver authors should use irq affinity for the interrupt handler.

The probability of packet loss due to backlog congestion can be reduced by the following measures or a combination thereof:

- (1) Drivers for kernel versions 2.4.x and above should always check the return value of `netif_rx()`. If it returns `NET_RX_DROP`, the driver's LAPB protocol must not confirm reception of the frame to the peer. This will reliably suppress packet loss. The LAPB protocol will automatically cause the peer to re-transmit the dropped packet later. The lapb module interface was modified to support this. Its `data_indication()` method should now transparently pass the `netif_rx()` return value to the (lapb module) caller.
- (2) Drivers for kernel versions 2.2.x should always check the global variable `netdev_dropping` when a new frame is received. The driver should only call `netif_rx()` if `netdev_dropping` is zero. Otherwise the driver should not confirm delivery of the frame and drop it. Alternatively, the driver can queue the frame internally and call `netif_rx()` later when `netif_dropping` is 0 again. In that case, delivery confirmation should also be deferred such that the internal queue cannot grow to much. This will not reliably avoid packet loss, but the probability of packet loss in `netif_rx()` path will be significantly reduced.
- (3) Additionally, driver authors might consider to support `CONFIG_NET_HW_FLOWCONTROL`. This allows the driver to be woken up when a previously congested backlog queue becomes empty again. The driver could use this for flow-controlling the peer by means of the LAPB protocol's flow-control service.

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 check-summing. 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 for Linux is being written which will allow X.25 to be run over an Ethernet (or Token Ring) and conform with the JNT “Pink Book”, this will have a different interface to the Packet Layer but there will be no confusion since the class of device being served by the LLC will be completely separate from LAPB. The LLC implementation is being done as part of another protocol project (SNA) and by a different author.

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

XFRM DEVICE - OFFLOADING THE IPSEC COMPUTATIONS

Shannon Nelson <shannon.nelson@oracle.com>

117.1 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.

Userland access to the offload is typically through a system such as libreswan or KAME/racoon, but the iproute2 ‘ip xfrm’ command set can be handy when experimenting. An example command might look something like this:

```
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
```

Yes, that’ s ugly, but that’ s what shell scripts and/or libreswan are for.

117.2 Callbacks to implement

```
/* from include/linux/netdevice.h */
struct xfrmdev_ops {
    int      (*xdo_dev_state_add) (struct xfrm_state *x);
    void     (*xdo_dev_state_delete) (struct xfrm_state *x);
    void     (*xdo_dev_state_free) (struct xfrm_state *x);
    bool     (*xdo_dev_offload_ok) (struct sk_buff *skb,
                                   struct xfrm_state *x);
    void     (*xdo_dev_state_advance_esn) (struct xfrm_state *x);
};
```

The NIC driver offering ipsec offload will need to implement these callbacks 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.

117.3 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
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.

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.

When the SA is removed by the user, the driver's `xdo_dev_state_delete()` is asked to disable the offload. Later, `xdo_dev_state_free()` is called from a garbage collection routine after all reference counts to the state 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()` and `xdo_dev_state_free()` on any remaining offloaded states.

XFRM PROC - /PROC/NET/XFRM_* FILES

Masahide NAKAMURA <nakam@linux-ipv6.org>

118.1 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.

118.1.1 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

XfrmInStateModeError:

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

118.1.2 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

XFRM

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

119.1 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;
```

(continues on next page)

(continued from previous page)

```
        __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.

119.2 2) TLVS reflect the different parameters:

a) byte value (XFRMA_LTIME_VAL)

This TLV carries the running/current counter for byte lifetime since last event.

b) replay value (XFRMA_REPLAY_VAL)

This TLV carries the running/current counter for replay sequence since last event.

c) replay threshold (XFRMA_REPLAY_THRESH)

This TLV carries the threshold being used by the kernel to trigger events when the replay sequence is exceeded.

d) expiry timer (XFRMA_ETIME_THRESH)

This is a timer value in milliseconds which is used as the nagle value to rate limit the events.

119.3 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_REPLAY_THRESH parameter in incremental packet count. The default is two packets.

119.4 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_REPLAY_VAL TLVs. * if XFRM_AE_RTHR flag is set, then XFRMA_REPLAY_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_REPLAY_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_REPLAY_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 (time-out 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_REPLAY_VAL TLVs.

119.5 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

120.1 /proc/sys/net/core/xfrm_* Variables:

xfrm_acq_expires - INTEGER

default 30 - hard timeout in seconds for acquire requests

INDEX

- \spxentry__alloc_skb\spxextraC function, 547
- \spxentry__dev_alloc_page\spxextraC function, 514
- \spxentry__dev_alloc_pages\spxextraC function, 514
- \spxentry__dev_get_by_flags\spxextraC function, 620
- \spxentry__dev_get_by_index\spxextraC function, 619
- \spxentry__dev_get_by_name\spxextraC function, 618
- \spxentry__dev_mc_sync\spxextraC function, 677
- \spxentry__dev_mc_unsync\spxextraC function, 678
- \spxentry__dev_remove_pack\spxextraC function, 616
- \spxentry__dev_uc_sync\spxextraC function, 677
- \spxentry__dev_uc_unsync\spxextraC function, 677
- \spxentry__genphy_config_aneg\spxextraC function, 727
- \spxentry__kfree_skb\spxextraC function, 549
- \spxentry__mdiobus_modify_changed\spxextraC function, 736
- \spxentry__mdiobus_read\spxextraC function, 736
- \spxentry__mdiobus_register\spxextraC function, 735
- \spxentry__mdiobus_write\spxextraC function, 736
- \spxentry__napi_alloc_skb\spxextraC function, 548
- \spxentry__napi_schedule\spxextraC function, 628
- \spxentry__napi_schedule_irqoff\spxextraC function, 629
- \spxentry__netdev_alloc_skb\spxextraC function, 548
- \spxentry__netif_napi_del\spxextraC function, 669
- \spxentry__netif_subqueue_stopped\spxextraC function, 672
- \spxentry__phy_clear_bits\spxextraC function, 716
- \spxentry__phy_clear_bits_mmd\spxextraC function, 717
- \spxentry__phy_modify\spxextraC function, 691
- \spxentry__phy_modify_changed\spxextraC function, 715
- \spxentry__phy_modify_mmd\spxextraC function, 693
- \spxentry__phy_modify_mmd_changed\spxextraC function, 692
- \spxentry__phy_read\spxextraC function, 714
- \spxentry__phy_read_mmd\spxextraC function, 689
- \spxentry__phy_set_bits\spxextraC function, 716
- \spxentry__phy_set_bits_mmd\spxextraC function, 717
- \spxentry__phy_write\spxextraC function, 715
- \spxentry__phy_write_mmd\spxextraC function, 690
- \spxentry__pskb_copy_fclone\spxextraC function, 551
- \spxentry__pskb_pull_tail\spxextraC function, 554
- \spxentry__rcu_dereference_sk_user_data_with_flags function, 536
- \spxentry__sk_mem_raise_allocated\spxextraC function, 567
- \spxentry__sk_mem_reclaim\spxextraC function, 568
- \spxentry__sk_mem_reduce_allocated\spxextraC function, 568

<code>\spxentry_sk_mem_schedule\spxextraC</code> function, 568	<code>\spxentryalloc_skb_fclone\spxextraC</code> function, 504
<code>\spxentry_sk_dequeue\spxextraC</code> function, 511	<code>\spxentryalloc_skb_for_msg\spxextraC</code> function, 549
<code>\spxentry_sk_dequeue_tail\spxextraC</code> function, 511	<code>\spxentryalloc_skb_with_frags\spxextraC</code> function, 564
<code>\spxentry_sk_fill_page_desc\spxextraC</code> function, 511	<code>\spxentrybpf_prog_create\spxextraC</code> function, 573
<code>\spxentry_sk_frag_ref\spxextraC</code> function, 516	<code>\spxentrybpf_prog_create_from_user\spxextraC</code> function, 573
<code>\spxentry_sk_frag_set_page\spxextraC</code> function, 517	<code>\spxentrybuild_skb_around\spxextraC</code> function, 547
<code>\spxentry_sk_frag_unref\spxextraC</code> function, 516	<code>\spxentrycall_netdevice_notifiers\spxextraC</code> function, 624
<code>\spxentry_sk_gso_segment\spxextraC</code> function, 626	<code>\spxentrycompare_ether_header\spxextraC</code> function, 651
<code>\spxentry_sk_header_release\spxextraC</code> function, 507	<code>\spxentryconsume_skb\spxextraC</code> function, 549
<code>\spxentry_sk_pad\spxextraC</code> function, 552	<code>\spxentrycsum_partial_copy_to_xdr\spxextraC</code> function, 597
<code>\spxentry_sk_peek\spxextraC</code> function, 508	<code>\spxentrydatagram_poll\spxextraC</code> function, 572
<code>\spxentry_sk_put_padto\spxextraC</code> function, 519	<code>\spxentrydev_add_offload\spxextraC</code> function, 617
<code>\spxentry_sk_queue_after\spxextraC</code> function, 510	<code>\spxentrydev_add_pack\spxextraC</code> function, 616
<code>\spxentry_sk_queue_head\spxextraC</code> function, 510	<code>\spxentrydev_alloc_name\spxextraC</code> function, 621
<code>\spxentry_sk_queue_head_init\spxextraC</code> function, 509	<code>\spxentrydev_change_carrier\spxextraC</code> function, 635
<code>\spxentry_sk_queue_purge\spxextraC</code> function, 514	<code>\spxentrydev_change_flags\spxextraC</code> function, 634
<code>\spxentry_sk_queue_tail\spxextraC</code> function, 510	<code>\spxentrydev_change_net_namespace\spxextraC</code> function, 641
<code>\spxentry_sk_try_rcv_datagram\spxextraC</code> function, 569	<code>\spxentrydev_change_proto_down\spxextraC</code> function, 637
<code>\spxentry_sock_create\spxextraC</code> function, 543	<code>\spxentrydev_change_proto_down_generic\spxextraC</code> function, 637
<code>\spxentry_copy_from_pages\spxextraC</code> function, 582	<code>\spxentrydev_change_proto_down_reason\spxextraC</code> function, 637
<code>\spxentry_phy_start_aneg\spxextraC</code> function, 686	<code>\spxentrydev_close\spxextraC</code> function, 622
<code>\spxentry_sock_tx_timestamp\spxextraC</code> function, 538	<code>\spxentrydev_disable_lro\spxextraC</code> function, 622
<code>\spxentryalloc_etherdev_mqs\spxextraC</code> function, 645	<code>\spxentrydev_fetch_sw_netstats\spxextraC</code> function, 639
<code>\spxentryalloc_netdev_mqs\spxextraC</code> function, 639	<code>\spxentrydev_fill_metadata_dst\spxextraC</code> function, 618
<code>\spxentryalloc_skb\spxextraC</code> function, 504	<code>\spxentrydev_forward_skb\spxextraC</code> function, 624

<code>\spxentrydev_get_by_index\spxextraC</code>	function, 620
<code>function, 619</code>	<code>\spxentrydevm_phy_package_join\spxextraC</code>
<code>\spxentrydev_get_by_index_rcu\spxextraC</code>	function, 725
<code>function, 619</code>	<code>\spxentrydim\spxextraC</code> struct, 778
<code>\spxentrydev_get_by_name\spxextraC</code>	<code>\spxentrydim_calc_stats\spxextraC</code>
<code>function, 619</code>	function, 781
<code>\spxentrydev_get_by_name_rcu\spxextraC</code>	<code>\spxentrydim_cq_moder\spxextraC</code>
<code>function, 618</code>	struct, 776
<code>\spxentrydev_get_by_napi_id\spxextraC</code>	<code>\spxentrydim_cq_period_mode\spxextraC</code>
<code>function, 620</code>	enum, 779
<code>\spxentrydev_get_flags\spxextraC</code> function, 634	<code>\spxentrydim_on_top\spxextraC</code> function, 780
<code>\spxentrydev_get_iflink\spxextraC</code> function, 618	<code>\spxentrydim_park_on_top\spxextraC</code> function, 781
<code>\spxentrydev_get_phys_port_id\spxextraC</code> function, 636	<code>\spxentrydim_park_tired\spxextraC</code> function, 781
<code>\spxentrydev_get_phys_port_name\spxextraC</code> function, 636	<code>\spxentrydim_sample\spxextraC</code> struct, 777
<code>\spxentrydev_get_port_parent_id\spxextraC</code> function, 636	<code>\spxentrydim_state\spxextraC</code> enum, 779
<code>\spxentrydev_get_stats\spxextraC</code> function, 639	<code>\spxentrydim_stats\spxextraC</code> struct, 777
<code>\spxentrydev_getbyhwaddr_rcu\spxextraC</code> function, 620	<code>\spxentrydim_stats_state\spxextraC</code> enum, 780
<code>\spxentrydev_hold\spxextraC</code> function, 674	<code>\spxentrydim_step_result\spxextraC</code> enum, 780
<code>\spxentrydev_loopback_xmit\spxextraC</code> function, 626	<code>\spxentrydim_tune_state\spxextraC</code> enum, 779
<code>\spxentrydev_nit_active\spxextraC</code> function, 624	<code>\spxentrydim_turn\spxextraC</code> function, 780
<code>\spxentrydev_open\spxextraC</code> function, 622	<code>\spxentrydim_update_sample\spxextraC</code> function, 781
<code>\spxentrydev_pre_changeaddr_notify\spxextraC</code> function, 635	<code>\spxentrydim_update_sample_with_comps\spxextraC</code> function, 782
<code>\spxentrydev_put\spxextraC</code> function, 674	<code>\spxentrydev\spxextraC</code> function, 482
<code>\spxentrydev_remove_offload\spxextraC</code> function, 617	<code>\spxentryeth_addr_dec\spxextraC</code> function, 651
<code>\spxentrydev_remove_pack\spxextraC</code> function, 617	<code>\spxentryeth_addr_inc\spxextraC</code> function, 651
<code>\spxentrydev_set_alias\spxextraC</code> function, 621	<code>\spxentryeth_broadcast_addr\spxextraC</code> function, 648
<code>\spxentrydev_set_allmulti\spxextraC</code> function, 634	<code>\spxentryeth_commit_mac_addr_change\spxextraC</code> function, 644
<code>\spxentrydev_set_group\spxextraC</code> function, 635	<code>\spxentryeth_get_headlen\spxextraC</code> function, 642
<code>\spxentrydev_set_mac_address\spxextraC</code> function, 635	<code>\spxentryeth_header\spxextraC</code> function, 642
<code>\spxentrydev_set_promiscuity\spxextraC</code> function, 634	<code>\spxentryeth_header_cache\spxextraC</code> function, 643
<code>\spxentrydev_valid_name\spxextraC</code>	<code>\spxentryeth_header_cache_update\spxextraC</code> function, 643

<code>\spxentryeth_header_parse\spxextraC</code> function, 643	<code>\spxentrygen_new_estimator\spxextraC</code> function, 579
<code>\spxentryeth_header_parse_protocol\spxextraC</code> function, 644	<code>\spxentrygen_replace_estimator\spxextraC</code> function, 580
<code>\spxentryeth_hw_addr_crc\spxextraC</code> function, 648	<code>\spxentrygenphy_aneg_done\spxextraC</code> function, 727
<code>\spxentryeth_hw_addr_inherit\spxextraC</code> function, 649	<code>\spxentrygenphy_c37_config_advert\spxextraC</code> function, 734
<code>\spxentryeth_hw_addr_random\spxextraC</code> function, 648	<code>\spxentrygenphy_c37_config_aneg\spxextraC</code> function, 727
<code>\spxentryeth_hw_addr_set\spxextraC</code> function, 649	<code>\spxentrygenphy_c37_read_status\spxextraC</code> function, 728
<code>\spxentryeth_mac_addr\spxextraC</code> func- tion, 644	<code>\spxentrygenphy_c45_an_config_aneg\spxextraC</code> function, 696
<code>\spxentryeth_prepare_mac_addr_change\spxextraC</code> function, 644	<code>\spxentrygenphy_c45_an_disable_aneg\spxextraC</code> function, 696
<code>\spxentryeth_proto_is_802_3\spxextraC</code> function, 647	<code>\spxentrygenphy_c45_aneg_done\spxextraC</code> function, 697
<code>\spxentryeth_random_addr\spxextraC</code> function, 647	<code>\spxentrygenphy_c45_check_and_restart_aneg\spxextraC</code> function, 697
<code>\spxentryeth_skb_pad\spxextraC</code> func- tion, 652	<code>\spxentrygenphy_c45_config_aneg\spxextraC</code> function, 699
<code>\spxentryeth_skb_pkt_type\spxextraC</code> function, 652	<code>\spxentrygenphy_c45_pma_read_abilities\spxextraC</code> function, 698
<code>\spxentryeth_type_trans\spxextraC</code> function, 643	<code>\spxentrygenphy_c45_pma_setup_forced\spxextraC</code> function, 696
<code>\spxentryeth_zero_addr\spxextraC</code> func- tion, 648	<code>\spxentrygenphy_c45_read_link\spxextraC</code> function, 697
<code>\spxentryether_addr_copy\spxextraC</code> function, 648	<code>\spxentrygenphy_c45_read_lpa\spxextraC</code> function, 698
<code>\spxentryether_addr_equal\spxextraC</code> function, 649	<code>\spxentrygenphy_c45_read_mdix\spxextraC</code> function, 698
<code>\spxentryether_addr_equal_64bits\spxextraC</code> function, 649	<code>\spxentrygenphy_c45_read_pma\spxextraC</code> function, 698
<code>\spxentryether_addr_equal_masked\spxextraC</code> function, 650	<code>\spxentrygenphy_c45_read_status\spxextraC</code> function, 698
<code>\spxentryether_addr_equal_unaligned\spxextraC</code> function, 650	<code>\spxentrygenphy_c45_restart_aneg\spxextraC</code> function, 697
<code>\spxentryether_addr_to_u64\spxextraC</code> function, 650	<code>\spxentrygenphy_check_and_restart_aneg\spxextraC</code> function, 727
<code>\spxentryether_setup\spxextraC</code> func- tion, 644	<code>\spxentrygenphy_config_advert\spxextraC</code> function, 733
<code>\spxentryethtool_pause_stats\spxextraC</code> struct, 470	<code>\spxentrygenphy_config_eee_advert\spxextraC</code> function, 726
<code>\spxentryfree_netdev\spxextraC</code> func- tion, 640	<code>\spxentrygenphy_read_abilities\spxextraC</code> function, 729
<code>\spxentrygen_estimator_active\spxextraC</code> function, 581	<code>\spxentrygenphy_read_status\spxextraC</code> function, 728
<code>\spxentrygen_kill_estimator\spxextraC</code> function, 580	<code>\spxentrygenphy_read_status_fixed\spxextraC</code> function, 728
	<code>\spxentrygenphy_restart_aneg\spxextraC</code>

function, [726](#)
\spxentrygenphy_setup_forced\spxextraC function, [726](#)
\spxentrygenphy_soft_reset\spxextraC function, [728](#)
\spxentrygenphy_update_link\spxextraC function, [728](#)
\spxentryget_phy_c22_id\spxextraC function, [732](#)
\spxentryget_phy_c45_ids\spxextraC function, [732](#)
\spxentryget_phy_device\spxextraC function, [721](#)
\spxentrygnet_estimator\spxextraC struct, [576](#)
\spxentrygnet_stats_basic\spxextraC struct, [574](#)
\spxentrygnet_stats_copy_app\spxextraC function, [579](#)
\spxentrygnet_stats_copy_basic\spxextraC function, [577](#)
\spxentrygnet_stats_copy_basic_hw\spxextraC function, [577](#)
\spxentrygnet_stats_copy_queue\spxextraC function, [578](#)
\spxentrygnet_stats_copy_rate_est\spxextraC function, [578](#)
\spxentrygnet_stats_finish_copy\spxextraC function, [579](#)
\spxentrygnet_stats_queue\spxextraC struct, [575](#)
\spxentrygnet_stats_rate_est\spxextraC struct, [574](#)
\spxentrygnet_stats_rate_est64\spxextraC struct, [575](#)
\spxentrygnet_stats_start_copy\spxextraC function, [576](#)
\spxentrygnet_stats_start_copy_compat\spxextraC function, [576](#)
\spxentryieee802154_alloc_device\spxextraC function, [481](#)
\spxentryieee802154_free_device\spxextraC function, [481](#)
\spxentryieee802154_register_device\spxextraC function, [481](#)
\spxentryieee802154_rx_irqsafe\spxextraC function, [481](#)
\spxentryieee802154_unregister_device\spxextraC function, [481](#)
\spxentryieee802154_xmit_complete\spxextraC function, [481](#)
\spxentryinit_dummy_netdev\spxextraC function, [638](#)
\spxentryis_broadcast_ether_addr\spxextraC function, [646](#)
\spxentryis_etherdev_addr\spxextraC function, [651](#)
\spxentryis_link_local_ether_addr\spxextraC function, [646](#)
\spxentryis_local_ether_addr\spxextraC function, [646](#)
\spxentryis_multicast_ether_addr\spxextraC function, [646](#)
\spxentryis_unicast_ether_addr\spxextraC function, [647](#)
\spxentryis_valid_ether_addr\spxextraC function, [647](#)
\spxentryis_zero_ether_addr\spxextraC function, [646](#)
\spxentrykernel_accept\spxextraC function, [545](#)
\spxentrykernel_bind\spxextraC function, [544](#)
\spxentrykernel_connect\spxextraC function, [545](#)
\spxentrykernel_getpeername\spxextraC function, [546](#)
\spxentrykernel_getsockname\spxextraC function, [545](#)
\spxentrykernel_listen\spxextraC function, [544](#)
\spxentrykernel_recvmsg\spxextraC function, [542](#)
\spxentrykernel_sendmsg\spxextraC function, [541](#)
\spxentrykernel_sendmsg_locked\spxextraC function, [541](#)
\spxentrykernel_sendpage\spxextraC function, [546](#)
\spxentrykernel_sendpage_locked\spxextraC function, [546](#)
\spxentrykernel_sock_ip_overhead\spxextraC function, [547](#)
\spxentrykernel_sock_shutdown\spxextraC function, [547](#)
\spxentrykfree_skb\spxextraC function, [549](#)
\spxentrylock_sock_fast\spxextraC function, [568](#)
\spxentrymac_an_restart\spxextraC function, [745](#)

<code>\spxentrymac_config\spxextraC</code> function, 743	<code>\spxentrynapi_schedule_irqoff\spxextraC</code> function, 652
<code>\spxentrymac_finish\spxextraC</code> function, 745	<code>\spxentrynapi_schedule_prep\spxextraC</code> function, 629
<code>\spxentrymac_link_down\spxextraC</code> function, 745	<code>\spxentrynapi_synchronize\spxextraC</code> function, 653
<code>\spxentrymac_link_up\spxextraC</code> function, 745	<code>\spxentrynet_device\spxextraC</code> struct, 655
<code>\spxentrymac_pcs_get_state\spxextraC</code> function, 742	<code>\spxentrynet_dim\spxextraC</code> function, 783
<code>\spxentrymac_prepare\spxextraC</code> function, 743	<code>\spxentrynet_dim_get_def_rx_moderation\spxextraC</code> function, 782
<code>\spxentrymdio_bus_match\spxextraC</code> function, 739	<code>\spxentrynet_dim_get_def_tx_moderation\spxextraC</code> function, 782
<code>\spxentrymdio_bus_stats\spxextraC</code> struct, 700	<code>\spxentrynet_dim_get_rx_moderation\spxextraC</code> function, 782
<code>\spxentrymdio_find_bus\spxextraC</code> function, 734	<code>\spxentrynet_dim_get_tx_moderation\spxextraC</code> function, 782
<code>\spxentrymdiobus_alloc\spxextraC</code> function, 704	<code>\spxentrynetdev_alloc_frag\spxextraC</code> function, 548
<code>\spxentrymdiobus_alloc_size\spxextraC</code> function, 734	<code>\spxentrynetdev_alloc_skb\spxextraC</code> function, 514
<code>\spxentrymdiobus_create_device\spxextraC</code> function, 739	<code>\spxentrynetdev_bonding_info_change\spxextraC</code> function, 633
<code>\spxentrymdiobus_free\spxextraC</code> function, 735	<code>\spxentrynetdev_boot_setup_check\spxextraC</code> function, 617
<code>\spxentrymdiobus_modify\spxextraC</code> function, 738	<code>\spxentrynetdev_cap_txqueue\spxextraC</code> function, 671
<code>\spxentrymdiobus_read\spxextraC</code> function, 737	<code>\spxentrynetdev_change_features\spxextraC</code> function, 638
<code>\spxentrymdiobus_read_nested\spxextraC</code> function, 737	<code>\spxentrynetdev_completed_queue\spxextraC</code> function, 671
<code>\spxentrymdiobus_release\spxextraC</code> function, 739	<code>\spxentrynetdev_features_change\spxextraC</code> function, 621
<code>\spxentrymdiobus_scan\spxextraC</code> function, 735	<code>\spxentrynetdev_get_xmit_slave\spxextraC</code> function, 633
<code>\spxentrymdiobus_write\spxextraC</code> function, 738	<code>\spxentrynetdev_has_any_upper_dev\spxextraC</code> function, 630
<code>\spxentrymdiobus_write_nested\spxextraC</code> function, 738	<code>\spxentrynetdev_has_upper_dev\spxextraC</code> function, 629
<code>\spxentrymii_bus\spxextraC</code> struct, 702	<code>\spxentrynetdev_has_upper_dev_all_rcu\spxextraC</code> function, 629
<code>\spxentrynapi_complete\spxextraC</code> function, 652	<code>\spxentrynetdev_increment_features\spxextraC</code> function, 641
<code>\spxentrynapi_disable\spxextraC</code> function, 653	<code>\spxentrynetdev_is_rx_handler_busy\spxextraC</code> function, 627
<code>\spxentrynapi_enable\spxextraC</code> function, 653	<code>\spxentrynetdev_lower_get_first_private_rcu\spxextraC</code> function, 631
<code>\spxentrynapi_if_scheduled_mark_missed\spxextraC</code> function, 653	<code>\spxentrynetdev_lower_get_next\spxextraC</code> function, 631
<code>\spxentrynapi_schedule\spxextraC</code> function, 652	<code>\spxentrynetdev_lower_get_next_private\spxextraC</code>

function, 630	\spxentrynetif_carrier_ok\spxextraC
\spxentrynetdev_lower_get_next_private_rcu\spxextraC	function, 675
function, 631	\spxentrynetif_carrier_on\spxextraC
\spxentrynetdev_lower_state_changed\spxextraC	function, 645
function, 633	\spxentrynetif_device_attach\spxextraC
\spxentrynetdev_master_upper_dev_get\spxextraC	function, 625
function, 630	\spxentrynetif_device_detach\spxextraC
\spxentrynetdev_master_upper_dev_get_rcu\spxextraC	function, 625
function, 631	\spxentrynetif_device_present\spxextraC
\spxentrynetdev_master_upper_dev_link\spxextraC	function, 676
function, 632	\spxentrynetif_dormant\spxextraC
\spxentrynetdev_notify_peers\spxextraC	function, 675
function, 622	\spxentrynetif_dormant_off\spxextraC
\spxentrynetdev_port_same_parent_id\spxextraC	function, 675
function, 636	\spxentrynetif_dormant_on\spxextraC
\spxentrynetdev_priv\spxextraC	function, 675
function, 668	\spxentrynetif_get_num_default_rss_queues\spxextraC
\spxentrynetdev_priv_flags\spxextraC	function, 625
enum, 654	\spxentrynetif_is_multiqueue\spxextraC
\spxentrynetdev_reset_queue\spxextraC	function, 674
function, 671	\spxentrynetif_napi_add\spxextraC
\spxentrynetdev_rx_handler_register\spxextraC	function, 668
function, 627	\spxentrynetif_napi_del\spxextraC
\spxentrynetdev_rx_handler_unregister\spxextraC	function, 669
function, 627	\spxentrynetif_oper_up\spxextraC
\spxentrynetdev_sent_queue\spxextraC	function, 676
function, 671	\spxentrynetif_queue_stopped\spxextraC
\spxentrynetdev_state_change\spxextraC	function, 670
function, 621	\spxentrynetif_receive_skb\spxextraC
\spxentrynetdev_txq_bql_complete_prefetchw\spxextraC	function, 628
function, 670	\spxentrynetif_receive_skb_core\spxextraC
\spxentrynetdev_txq_bql_enqueue_prefetchw\spxextraC	function, 628
function, 670	\spxentrynetif_receive_skb_list\spxextraC
\spxentrynetdev_update_features\spxextraC	function, 628
function, 637	\spxentrynetif_running\spxextraC
\spxentrynetdev_upper_dev_link\spxextraC	function, 672
function, 632	\spxentrynetif_rx\spxextraC
\spxentrynetdev_upper_dev_unlink\spxextraC	function, 627
function, 633	\spxentrynetif_set_real_num_rx_queues\spxextraC
\spxentrynetdev_upper_get_next_dev_rcu\spxextraC	function, 625
function, 630	\spxentrynetif_stacked_transfer_operstate\spxextraC
\spxentrynetif_attr_test_mask\spxextraC	function, 638
function, 673	\spxentrynetif_start_queue\spxextraC
\spxentrynetif_attr_test_online\spxextraC	function, 670
function, 673	\spxentrynetif_start_subqueue\spxextraC
\spxentrynetif_attrmask_next\spxextraC	function, 672
function, 673	\spxentrynetif_stop_queue\spxextraC
\spxentrynetif_attrmask_next_and\spxextraC	function, 670
function, 674	\spxentrynetif_stop_subqueue\spxextraC
\spxentrynetif_carrier_off\spxextraC	function, 672
function, 645	\spxentrynetif_testing\spxextraC

- tion, [676](#)
- \spxentrynetif_testing_off\spxextraC function, [676](#)
- \spxentrynetif_testing_on\spxextraC function, [675](#)
- \spxentrynetif_tx_lock\spxextraC function, [676](#)
- \spxentrynetif_tx_napi_add\spxextraC function, [669](#)
- \spxentrynetif_wake_queue\spxextraC function, [670](#)
- \spxentrynetif_wake_subqueue\spxextraC function, [673](#)
- \spxentrynvmem_get_mac_address\spxextraC function, [645](#)
- \spxentryof_mdio_find_bus\spxextraC function, [734](#)
- \spxentrypcs_an_restart\spxextraC function, [748](#)
- \spxentrypcs_config\spxextraC function, [748](#)
- \spxentrypcs_get_state\spxextraC function, [747](#)
- \spxentrypcs_link_up\spxextraC function, [748](#)
- \spxentryphy_advertise_supported\spxextraC function, [729](#)
- \spxentryphy_aneg_done\spxextraC function, [678](#)
- \spxentryphy_attach\spxextraC function, [724](#)
- \spxentryphy_attach_direct\spxextraC function, [723](#)
- \spxentryphy_c45_device_ids\spxextraC struct, [704](#)
- \spxentryphy_check_downshift\spxextraC function, [689](#)
- \spxentryphy_check_link_status\spxextraC function, [686](#)
- \spxentryphy_check_valid\spxextraC function, [685](#)
- \spxentryphy_clear_bits\spxextraC function, [717](#)
- \spxentryphy_clear_bits_mmd\spxextraC function, [718](#)
- \spxentryphy_clear_interrupt\spxextraC function, [684](#)
- \spxentryphy_config_interrupt\spxextraC function, [684](#)
- \spxentryphy_connect\spxextraC function, [722](#)
- \spxentryphy_connect_direct\spxextraC function, [722](#)
- \spxentryphy_detach\spxextraC function, [725](#)
- \spxentryphy_device\spxextraC struct, [705](#)
- \spxentryphy_device_register\spxextraC function, [721](#)
- \spxentryphy_device_remove\spxextraC function, [721](#)
- \spxentryphy_did_interrupt\spxextraC function, [687](#)
- \spxentryphy_disable_interrupts\spxextraC function, [687](#)
- \spxentryphy_disconnect\spxextraC function, [723](#)
- \spxentryphy_do_ioctl\spxextraC function, [679](#)
- \spxentryphy_do_ioctl_running\spxextraC function, [679](#)
- \spxentryphy_driver\spxextraC struct, [710](#)
- \spxentryphy_driver_register\spxextraC function, [732](#)
- \spxentryphy_duplex_to_str\spxextraC function, [688](#)
- \spxentryphy_enable_interrupts\spxextraC function, [687](#)
- \spxentryphy_error\spxextraC function, [687](#)
- \spxentryphy_ethtool_get_eee\spxextraC function, [683](#)
- \spxentryphy_ethtool_get_sset_count\spxextraC function, [680](#)
- \spxentryphy_ethtool_get_stats\spxextraC function, [680](#)
- \spxentryphy_ethtool_get_strings\spxextraC function, [679](#)
- \spxentryphy_ethtool_get_wol\spxextraC function, [684](#)
- \spxentryphy_ethtool_nway_reset\spxextraC function, [684](#)
- \spxentryphy_ethtool_set_eee\spxextraC function, [683](#)
- \spxentryphy_ethtool_set_wol\spxextraC function, [683](#)
- \spxentryphy_find_first\spxextraC function, [722](#)
- \spxentryphy_find_valid\spxextraC func-

- tion, 684
- \spxentryphy_free_interrupt\spxextraC function, 682
- \spxentryphy_get_eee_err\spxextraC function, 683
- \spxentryphy_get_internal_delay\spxextraC function, 731
- \spxentryphy_get_pause\spxextraC function, 731
- \spxentryphy_handle_interrupt\spxextraC function, 687
- \spxentryphy_has_hwtstamp\spxextraC function, 719
- \spxentryphy_has_rxtstamp\spxextraC function, 719
- \spxentryphy_has_tsinfo\spxextraC function, 719
- \spxentryphy_has_txtstamp\spxextraC function, 719
- \spxentryphy_init_eee\spxextraC function, 683
- \spxentryphy_interface_is_rgmii\spxextraC function, 720
- \spxentryphy_interface_mode_is_8023z\spxextraC function, 719
- \spxentryphy_interface_mode_is_rgmii\spxextraC function, 719
- \spxentryphy_interface_t\spxextraC enum, 699
- \spxentryphy_interrupt\spxextraC function, 687
- \spxentryphy_interrupt_is_valid\spxextraC function, 718
- \spxentryphy_is_internal\spxextraC function, 719
- \spxentryphy_is_pseudo_fixed_link\spxextraC function, 720
- \spxentryphy_is_started\spxextraC function, 714
- \spxentryphy_lookup_setting\spxextraC function, 688
- \spxentryphy_mac_interrupt\spxextraC function, 682
- \spxentryphy_mii_ioctl\spxextraC function, 678
- \spxentryphy_modes\spxextraC function, 700
- \spxentryphy_modify\spxextraC function, 691
- \spxentryphy_modify_changed\spxextraC function, 691
- \spxentryphy_modify_mmd\spxextraC function, 693
- \spxentryphy_modify_mmd_changed\spxextraC function, 692
- \spxentryphy_modify_paged\spxextraC function, 696
- \spxentryphy_modify_paged_changed\spxextraC function, 695
- \spxentryphy_module_driver\spxextraC macro, 720
- \spxentryphy_package_join\spxextraC function, 724
- \spxentryphy_package_leave\spxextraC function, 725
- \spxentryphy_package_shared\spxextraC struct, 701
- \spxentryphy_poll_reset\spxextraC function, 733
- \spxentryphy_polling_mode\spxextraC function, 718
- \spxentryphy_prepare_link\spxextraC function, 733
- \spxentryphy_print_status\spxextraC function, 678
- \spxentryphy_probe\spxextraC function, 734
- \spxentryphy_queue_state_machine\spxextraC function, 679
- \spxentryphy_read\spxextraC function, 714
- \spxentryphy_read_mmd\spxextraC function, 690
- \spxentryphy_read_mmd_poll_timeout\spxextraC macro, 715
- \spxentryphy_read_paged\spxextraC function, 694
- \spxentryphy_register_fixup\spxextraC function, 720
- \spxentryphy_remove_link_mode\spxextraC function, 729
- \spxentryphy_request_interrupt\spxextraC function, 681
- \spxentryphy_reset_after_clk_enable\spxextraC function, 726
- \spxentryphy_resolve_aneg_linkmode\spxextraC function, 689
- \spxentryphy_resolve_aneg_pause\spxextraC function, 689
- \spxentryphy_restart_aneg\spxextraC function, 678
- \spxentryphy_restore_page\spxextraC

- function, 694
- \spxentryphy_sanitize_settings\spxextraC function, 685
- \spxentryphy_save_page\spxextraC function, 694
- \spxentryphy_select_page\spxextraC function, 694
- \spxentryphy_set_asym_pause\spxextraC function, 730
- \spxentryphy_set_bits\spxextraC function, 716
- \spxentryphy_set_bits_mmd\spxextraC function, 718
- \spxentryphy_set_max_speed\spxextraC function, 688
- \spxentryphy_set_sym_pause\spxextraC function, 730
- \spxentryphy_sfp_attach\spxextraC function, 723
- \spxentryphy_sfp_detach\spxextraC function, 723
- \spxentryphy_sfp_probe\spxextraC function, 723
- \spxentryphy_speed_down\spxextraC function, 681
- \spxentryphy_speed_to_str\spxextraC function, 688
- \spxentryphy_speed_up\spxextraC function, 681
- \spxentryphy_start\spxextraC function, 682
- \spxentryphy_start_aneg\spxextraC function, 680
- \spxentryphy_start_cable_test\spxextraC function, 680
- \spxentryphy_start_cable_test_tdr\spxextraC function, 680
- \spxentryphy_start_machine\spxextraC function, 681
- \spxentryphy_state\spxextraC enum, 704
- \spxentryphy_state_machine\spxextraC function, 688
- \spxentryphy_stop\spxextraC function, 682
- \spxentryphy_stop_machine\spxextraC function, 686
- \spxentryphy_support_asym_pause\spxextraC function, 730
- \spxentryphy_support_sym_pause\spxextraC function, 729
- \spxentryphy_supported_speeds\spxextraC function, 685
- \spxentryphy_tdr_config\spxextraC struct, 709
- \spxentryphy_trigger_machine\spxextraC function, 686
- \spxentryphy_unregister_fixup\spxextraC function, 721
- \spxentryphy_validate_pause\spxextraC function, 731
- \spxentryphy_write\spxextraC function, 714
- \spxentryphy_write_mmd\spxextraC function, 690
- \spxentryphy_write_paged\spxextraC function, 695
- \spxentryphylink\spxextraC struct, 749
- \spxentryphylink_config\spxextraC struct, 740
- \spxentryphylink_connect_phy\spxextraC function, 750
- \spxentryphylink_create\spxextraC function, 749
- \spxentryphylink_decode_usxgmii_word\spxextraC function, 756
- \spxentryphylink_destroy\spxextraC function, 750
- \spxentryphylink_disconnect_phy\spxextraC function, 751
- \spxentryphylink_ethtool_get_eee\spxextraC function, 754
- \spxentryphylink_ethtool_get_pauseparam\spxextraC function, 753
- \spxentryphylink_ethtool_get_wol\spxextraC function, 752
- \spxentryphylink_ethtool_ksettings_get\spxextraC function, 753
- \spxentryphylink_ethtool_ksettings_set\spxextraC function, 753
- \spxentryphylink_ethtool_nway_reset\spxextraC function, 753
- \spxentryphylink_ethtool_set_eee\spxextraC function, 754
- \spxentryphylink_ethtool_set_pauseparam\spxextraC function, 754
- \spxentryphylink_ethtool_set_wol\spxextraC function, 752
- \spxentryphylink_get_eee_err\spxextraC function, 754
- \spxentryphylink_helper_basex_speed\spxextraC function, 756

<code>\spxentryphylink_init_eee\spxextraC</code> function, 754	<code>\spxentryregister_netdev\spxextraC</code> function, 639
<code>\spxentryphylink_link_state\spxextraC</code> struct, 740	<code>\spxentryregister_netdevice\spxextraC</code> function, 638
<code>\spxentryphylink_mac_change\spxextraC</code> function, 751	<code>\spxentryregister_netdevice_notifier\spxextraC</code> function, 622
<code>\spxentryphylink_mac_ops\spxextraC</code> struct, 741	<code>\spxentryregister_netdevice_notifier_net\spxextraC</code> function, 623
<code>\spxentryphylink_mii_c22_pcs_an_restart\spxextraC</code> function, 757	<code>\spxentryrpc_add_pipe_dir_object\spxextraC</code> function, 599
<code>\spxentryphylink_mii_c22_pcs_config\spxextraC</code> function, 757	<code>\spxentryrpc_alloc_iostats\spxextraC</code> function, 597
<code>\spxentryphylink_mii_c22_pcs_get_state\spxextraC</code> function, 756	<code>\spxentryrpc_bind_new_program\spxextraC</code> function, 602
<code>\spxentryphylink_mii_c22_pcs_set_advertis\spxextraC</code> function, 757	<code>\spxentryrpc_bind_new_program_async\spxextraC</code> function, 602
<code>\spxentryphylink_mii_ioctl\spxextraC</code> function, 755	<code>\spxentryrpc_call_sync\spxextraC</code> func- tion, 602
<code>\spxentryphylink_of_phy_connect\spxextraC</code> function, 751	<code>\spxentryrpc_clnt_add_xprt\spxextraC</code> function, 606
<code>\spxentryphylink_pcs\spxextraC</code> struct, 746	<code>\spxentryrpc_clnt_iterate_for_each_xprt\spxextraC</code> function, 601
<code>\spxentryphylink_pcs_ops\spxextraC</code> struct, 747	<code>\spxentryrpc_clnt_setup_test_and_add_xprt\spxextraC</code> function, 605
<code>\spxentryphylink_set_pcs\spxextraC</code> function, 750	<code>\spxentryrpc_clnt_test_and_add_xprt\spxextraC</code> function, 605
<code>\spxentryphylink_set_port_modes\spxextraC</code> function, 749	<code>\spxentryrpc_clone_client\spxextraC</code> function, 600
<code>\spxentryphylink_speed_down\spxextraC</code> function, 755	<code>\spxentryrpc_clone_client_set_auth\spxextraC</code> function, 601
<code>\spxentryphylink_speed_up\spxextraC</code> function, 755	<code>\spxentryrpc_count_iostats\spxextraC</code> function, 597
<code>\spxentryphylink_start\spxextraC</code> func- tion, 752	<code>\spxentryrpc_count_iostats_metrics\spxextraC</code> function, 597
<code>\spxentryphylink_stop\spxextraC</code> func- tion, 752	<code>\spxentryrpc_create\spxextraC</code> func- tion, 600
<code>\spxentrypskb_expand_head\spxextraC</code> function, 551	<code>\spxentryrpc_find_or_alloc_pipe_dir_object\spxextraC</code> function, 600
<code>\spxentrypskb_put\spxextraC</code> function, 552	<code>\spxentryrpc_force_rebind\spxextraC</code> function, 605
<code>\spxentrypskb_trim_rcsum\spxextraC</code> function, 521	<code>\spxentryrpc_free\spxextraC</code> function, 596
<code>\spxentrypskb_trim_unique\spxextraC</code> function, 513	<code>\spxentryrpc_free_iostats\spxextraC</code> function, 597
<code>\spxentryrdma_dim\spxextraC</code> function, 783	<code>\spxentryrpc_init_pipe_dir_head\spxextraC</code> function, 599
<code>\spxentryread_zsdata\spxextraC</code> func- tion, 342	<code>\spxentryrpc_init_pipe_dir_object\spxextraC</code> function, 599
<code>\spxentryread_zsreg\spxextraC</code> func- tion, 341	<code>\spxentryrpc_localaddr\spxextraC</code> func- tion, 604
	<code>\spxentryrpc_malloc\spxextraC</code> func-

- tion, 596
- \spxentryrpc_max_bc_payload\spxextraC function, 604
- \spxentryrpc_max_payload\spxextraC function, 604
- \spxentryrpc_mkpipe_dentry\spxextraC function, 598
- \spxentryrpc_net_ns\spxextraC function, 604
- \spxentryrpc_peeraddr\spxextraC function, 603
- \spxentryrpc_peeraddr2str\spxextraC function, 603
- \spxentryrpc_prepare_reply_pages\spxextraC function, 603
- \spxentryrpc_queue_upcall\spxextraC function, 598
- \spxentryrpc_remove_pipe_dir_object\spxextraC function, 599
- \spxentryrpc_run_task\spxextraC function, 602
- \spxentryrpc_switch_client_transport\spxextraC function, 601
- \spxentryrpc_unlink\spxextraC function, 598
- \spxentryrpc_wake_up\spxextraC function, 596
- \spxentryrpc_wake_up_status\spxextraC function, 596
- \spxentryrpcb_getport_async\spxextraC function, 600
- \spxentryrps_may_expire_flow\spxextraC function, 626
- \spxentryrtnl_link_stats64\spxextraC struct, 1369
- \spxentryset_channel\spxextraC function, 482
- \spxentrysfp_bus\spxextraC struct, 758
- \spxentrysfp_bus_add_upstream\spxextraC function, 762
- \spxentrysfp_bus_del_upstream\spxextraC function, 763
- \spxentrysfp_bus_find_fwnode\spxextraC function, 762
- \spxentrysfp_bus_put\spxextraC function, 760
- \spxentrysfp_eeprom_id\spxextraC struct, 758
- \spxentrysfp_get_module_eeprom\spxextraC function, 761
- \spxentrysfp_get_module_info\spxextraC function, 761
- \spxentrysfp_may_have_phy\spxextraC function, 760
- \spxentrysfp_parse_port\spxextraC function, 759
- \spxentrysfp_parse_support\spxextraC function, 760
- \spxentrysfp_select_interface\spxextraC function, 760
- \spxentrysfp_upstream_ops\spxextraC struct, 758
- \spxentrysfp_upstream_start\spxextraC function, 761
- \spxentrysfp_upstream_stop\spxextraC function, 762
- \spxentrysk_alloc\spxextraC function, 566
- \spxentrysk_attach_filter\spxextraC function, 574
- \spxentrysk_buff\spxextraC struct, 494
- \spxentrysk_capable\spxextraC function, 565
- \spxentrysk_clone_lock\spxextraC function, 567
- \spxentrysk_eat_skb\spxextraC function, 539
- \spxentrysk_filter_trim_cap\spxextraC function, 573
- \spxentrysk_for_each_entry_offset_rcu\spxextraC macro, 536
- \spxentrysk_has_allocations\spxextraC function, 537
- \spxentrysk_net_capable\spxextraC function, 566
- \spxentrysk_ns_capable\spxextraC function, 565
- \spxentrysk_page_frag\spxextraC function, 538
- \spxentrysk_rmem_alloc_get\spxextraC function, 537
- \spxentrysk_set_memalloc\spxextraC function, 566
- \spxentrysk_stream_wait_connect\spxextraC function, 572
- \spxentrysk_stream_wait_memory\spxextraC function, 572
- \spxentrysk_user_data_is_nocopy\spxextraC function, 536
- \spxentrysk_wait_data\spxextraC function, 567

`\spxentrysk_wmem_alloc_get\spxextraC` function, 536

`\spxentryskb_abort_seq_read\spxextraC` function, 558

`\spxentryskb_append\spxextraC` function, 557

`\spxentryskb_availroom\spxextraC` function, 512

`\spxentryskb_checksum_complete\spxextraC` function, 523

`\spxentryskb_checksum_none_assert\spxextraC` function, 524

`\spxentryskb_checksum_setup\spxextraC` function, 561

`\spxentryskb_checksum_trimmed\spxextraC` function, 561

`\spxentryskb_clone\spxextraC` function, 550

`\spxentryskb_clone_sk\spxextraC` function, 560

`\spxentryskb_clone_writable\spxextraC` function, 518

`\spxentryskb_cloned\spxextraC` function, 506

`\spxentryskb_complete_tx_timestamp\spxextraC` function, 522

`\spxentryskb_complete_wifi_ack\spxextraC` function, 523

`\spxentryskb_copy\spxextraC` function, 550

`\spxentryskb_copy_and_csum_datagram_iter\spxextraC` function, 571

`\spxentryskb_copy_and_hash_datagram_iter\spxextraC` function, 570

`\spxentryskb_copy_bits\spxextraC` function, 554

`\spxentryskb_copy_datagram_from_iter\spxextraC` function, 571

`\spxentryskb_copy_datagram_iter\spxextraC` function, 570

`\spxentryskb_copy_expand\spxextraC` function, 552

`\spxentryskb_copy_ubufs\spxextraC` function, 550

`\spxentryskb_cow\spxextraC` function, 518

`\spxentryskb_cow_data\spxextraC` function, 560

`\spxentryskb_cow_head\spxextraC` function, 519

`\spxentryskb_dequeue\spxextraC` function, 555

`\spxentryskb_dequeue_tail\spxextraC` function, 555

`\spxentryskb_dst\spxextraC` function, 502

`\spxentryskb_dst_is_noref\spxextraC` function, 503

`\spxentryskb_dst_set\spxextraC` function, 503

`\spxentryskb_dst_set_noref\spxextraC` function, 503

`\spxentryskb_eth_pop\spxextraC` function, 562

`\spxentryskb_eth_push\spxextraC` function, 563

`\spxentryskb_ext\spxextraC` struct, 523

`\spxentryskb_ext_add\spxextraC` function, 565

`\spxentryskb_fclone_busy\spxextraC` function, 504

`\spxentryskb_fill_page_desc\spxextraC` function, 512

`\spxentryskb_find_text\spxextraC` function, 558

`\spxentryskb_frag_address\spxextraC` function, 517

`\spxentryskb_frag_address_safe\spxextraC` function, 517

`\spxentryskb_frag_dma_map\spxextraC` function, 518

`\spxentryskb_frag_foreach_page\spxextraC` macro, 493

`\spxentryskb_frag_iter\spxextraC` macro, 493

`\spxentryskb_frag_loop\spxextraC` function, 493

`\spxentryskb_frag_off\spxextraC` function, 515

`\spxentryskb_frag_off_add\spxextraC` function, 515

`\spxentryskb_frag_off_copy\spxextraC` function, 515

`\spxentryskb_frag_off_set\spxextraC` function, 515

`\spxentryskb_frag_page\spxextraC` function, 516

`\spxentryskb_frag_page_copy\spxextraC` function, 517

`\spxentryskb_frag_ref\spxextraC` function, 516

`\spxentryskb_frag_set_page\spxextraC` function, 517

`\spxentryskb_frag_size\spxextraC` function, 555

- tion, 492
- \spxentryskb_frag_size_add\spxextraC function, 493
- \spxentryskb_frag_size_set\spxextraC function, 492
- \spxentryskb_frag_size_sub\spxextraC function, 493
- \spxentryskb_frag_unref\spxextraC function, 516
- \spxentryskb_get\spxextraC function, 506
- \spxentryskb_get_timestamp\spxextraC function, 522
- \spxentryskb_gso_validate_mac_len\spxextraC function, 562
- \spxentryskb_gso_validate_network_len\spxextraC function, 562
- \spxentryskb_has_shared_frag\spxextraC function, 520
- \spxentryskb_head_is_locked\spxextraC function, 524
- \spxentryskb_header_cloned\spxextraC function, 506
- \spxentryskb_headroom\spxextraC function, 512
- \spxentryskb_kill_datagram\spxextraC function, 570
- \spxentryskb_linearize\spxextraC function, 520
- \spxentryskb_linearize_cow\spxextraC function, 520
- \spxentryskb_mac_gso_segment\spxextraC function, 625
- \spxentryskb_morph\spxextraC function, 550
- \spxentryskb_mpls_dec_ttl\spxextraC function, 564
- \spxentryskb_mpls_pop\spxextraC function, 563
- \spxentryskb_mpls_push\spxextraC function, 563
- \spxentryskb_mpls_update_lse\spxextraC function, 564
- \spxentryskb_napi_id\spxextraC function, 503
- \spxentryskb_needs_linearize\spxextraC function, 521
- \spxentryskb_orphan\spxextraC function, 513
- \spxentryskb_orphan_fragments\spxextraC function, 513
- \spxentryskb_pad\spxextraC function, 504
- \spxentryskb_padto\spxextraC function, 519
- \spxentryskb_page_frag_refill\spxextraC function, 567
- \spxentryskb_partial_csum_set\spxextraC function, 560
- \spxentryskb_peek\spxextraC function, 507
- \spxentryskb_peek_next\spxextraC function, 508
- \spxentryskb_peek_tail\spxextraC function, 508
- \spxentryskb_pfmemalloc\spxextraC function, 502
- \spxentryskb_postpull_rcsum\spxextraC function, 520
- \spxentryskb_postpush_rcsum\spxextraC function, 521
- \spxentryskb_prepare_seq_read\spxextraC function, 557
- \spxentryskb_propagate_pfmemalloc\spxextraC function, 515
- \spxentryskb_pull\spxextraC function, 553
- \spxentryskb_pull_rcsum\spxextraC function, 559
- \spxentryskb_push\spxextraC function, 553
- \spxentryskb_push_rcsum\spxextraC function, 521
- \spxentryskb_put\spxextraC function, 553
- \spxentryskb_put_padto\spxextraC function, 519
- \spxentryskb_queue_empty\spxextraC function, 505
- \spxentryskb_queue_empty_lockless\spxextraC function, 505
- \spxentryskb_queue_head\spxextraC function, 556
- \spxentryskb_queue_is_first\spxextraC function, 505
- \spxentryskb_queue_is_last\spxextraC function, 505
- \spxentryskb_queue_len\spxextraC function, 508
- \spxentryskb_queue_len_lockless\spxextraC function, 509
- \spxentryskb_queue_next\spxextraC

function, 505
\spxentryskb_queue_prev\spxextraC function, 506
\spxentryskb_queue_purge\spxextraC function, 556
\spxentryskb_queue_splice\spxextraC function, 509
\spxentryskb_queue_splice_init\spxextraC function, 509
\spxentryskb_queue_splice_tail\spxextraC function, 509
\spxentryskb_queue_splice_tail_init\spxextraC function, 510
\spxentryskb_queue_tail\spxextraC function, 556
\spxentryskb_reserve\spxextraC function, 512
\spxentryskb_rtable\spxextraC function, 503
\spxentryskb_scrub_packet\spxextraC function, 562
\spxentryskb_segment\spxextraC function, 559
\spxentryskb_seq_read\spxextraC function, 558
\spxentryskb_share_check\spxextraC function, 507
\spxentryskb_shared\spxextraC function, 507
\spxentryskb_shared_hwtstamps\spxextraC struct, 494
\spxentryskb_split\spxextraC function, 557
\spxentryskb_steal_sock\spxextraC function, 539
\spxentryskb_store_bits\spxextraC function, 554
\spxentryskb_tailroom\spxextraC function, 512
\spxentryskb_tailroom_reserve\spxextraC function, 513
\spxentryskb_to_sgvec\spxextraC function, 559
\spxentryskb_trim\spxextraC function, 553
\spxentryskb_try_coalesce\spxextraC function, 561
\spxentryskb_tstamp_tx\spxextraC function, 522
\spxentryskb_tx_error\spxextraC function, 549
\spxentryskb_tx_timestamp\spxextraC function, 523
\spxentryskb_unlink\spxextraC function, 556
\spxentryskb_unref\spxextraC function, 503
\spxentryskb_unshare\spxextraC function, 507
\spxentryskb_zerocopy\spxextraC function, 555
\spxentryskwq_has_sleeper\spxextraC function, 537
\spxentrysock\spxextraC struct, 528
\spxentrysock_alloc\spxextraC function, 540
\spxentrysock_alloc_file\spxextraC function, 539
\spxentrysock_common\spxextraC struct, 524
\spxentrysock_create\spxextraC function, 543
\spxentrysock_create_kern\spxextraC function, 543
\spxentrysock_create_lite\spxextraC function, 542
\spxentrysock_from_file\spxextraC function, 539
\spxentrysock_poll_wait\spxextraC function, 538
\spxentrysock_recvmsg\spxextraC function, 541
\spxentrysock_register\spxextraC function, 544
\spxentrysock_release\spxextraC function, 540
\spxentrysock_sendmsg\spxextraC function, 540
\spxentrysock_shutdown_cmd\spxextraC enum, 491
\spxentrysock_type\spxextraC enum, 491
\spxentrysock_unregister\spxextraC function, 544
\spxentrysocket\spxextraC struct, 492
\spxentrysockfd_lookup\spxextraC function, 540
\spxentryspans_boundary\spxextraC function, 346
\spxentrystart\spxextraC function, 482
\spxentrystop\spxextraC function, 482
\spxentrysvc_find_xprt\spxextraC func-

- tion, [590](#)
- \spxentrysvc_print_addr\spxextraC function, [589](#)
- \spxentrysvc_reserve\spxextraC function, [589](#)
- \spxentrysvc_xprt_names\spxextraC function, [590](#)
- \spxentrysynchronize_net\spxextraC function, [640](#)
- \spxentryu64_to_ether_addr\spxextraC function, [651](#)
- \spxentryunlock_sock_fast\spxextraC function, [536](#)
- \spxentryunregister_netdev\spxextraC function, [641](#)
- \spxentryunregister_netdevice_many\spxextraC function, [641](#)
- \spxentryunregister_netdevice_notifier\spxextraC function, [623](#)
- \spxentryunregister_netdevice_notifier_net\spxextraC function, [623](#)
- \spxentryunregister_netdevice_queue\spxextraC function, [640](#)
- \spxentryvalidate\spxextraC function, [742](#)
- \spxentrywimax_dev\spxextraC struct, [612](#)
- \spxentrywimax_dev_add\spxextraC function, [611](#)
- \spxentrywimax_dev_init\spxextraC function, [611](#)
- \spxentrywimax_dev_rm\spxextraC function, [611](#)
- \spxentrywimax_msg\spxextraC function, [608](#)
- \spxentrywimax_msg_alloc\spxextraC function, [606](#)
- \spxentrywimax_msg_data\spxextraC function, [607](#)
- \spxentrywimax_msg_data_len\spxextraC function, [607](#)
- \spxentrywimax_msg_len\spxextraC function, [607](#)
- \spxentrywimax_msg_send\spxextraC function, [607](#)
- \spxentrywimax_report_rfkill_hw\spxextraC function, [609](#)
- \spxentrywimax_report_rfkill_sw\spxextraC function, [609](#)
- \spxentrywimax_reset\spxextraC function, [608](#)
- \spxentrywimax_rfkill\spxextraC function, [610](#)
- \spxentrywimax_st\spxextraC enum, [615](#)
- \spxentrywimax_state_change\spxextraC function, [610](#)
- \spxentrywimax_state_get\spxextraC function, [611](#)
- \spxentrywrite_zsctrl\spxextraC function, [342](#)
- \spxentrywrite_zsdata\spxextraC function, [342](#)
- \spxentrywrite_zsreg\spxextraC function, [342](#)
- \spxentryxdr_buf_subsegment\spxextraC function, [587](#)
- \spxentryxdr_buf_trim\spxextraC function, [587](#)
- \spxentryxdr_commit_encode\spxextraC function, [583](#)
- \spxentryxdr_encode_opaque\spxextraC function, [581](#)
- \spxentryxdr_encode_opaque_fixed\spxextraC function, [581](#)
- \spxentryxdr_enter_page\spxextraC function, [587](#)
- \spxentryxdr_init_decode\spxextraC function, [585](#)
- \spxentryxdr_init_decode_pages\spxextraC function, [585](#)
- \spxentryxdr_init_encode\spxextraC function, [583](#)
- \spxentryxdr_inline_decode\spxextraC function, [586](#)
- \spxentryxdr_inline_pages\spxextraC function, [582](#)
- \spxentryxdr_page_pos\spxextraC function, [583](#)
- \spxentryxdr_read_pages\spxextraC function, [586](#)
- \spxentryxdr_reserve_space\spxextraC function, [583](#)
- \spxentryxdr_reserve_space_vec\spxextraC function, [584](#)
- \spxentryxdr_restrict_buflen\spxextraC function, [584](#)
- \spxentryxdr_set_scratch_buffer\spxextraC function, [586](#)

`\spxentryxdr_stream_decode_opaque\spxextraC` function, 591
function, 588 `\spxentryxprt_unpin_rqst\spxextraC`
`\spxentryxdr_stream_decode_opaque_dup\spxextraC` function, 594
function, 588 `\spxentryxprt_unregister_transport\spxextraC`
`\spxentryxdr_stream_decode_string\spxextraC` function, 591
function, 588 `\spxentryxprt_update_rtt\spxextraC`
`\spxentryxdr_stream_decode_string_dup\spxextraC` function, 595
function, 589 `\spxentryxprt_wait_for_buffer_space\spxextraC`
`\spxentryxdr_stream_pos\spxextraC` function, 593
function, 582 `\spxentryxprt_wait_for_reply_request_def\spxextraC`
`\spxentryxdr_terminate_string\spxextraC` function, 595
function, 582 `\spxentryxprt_wait_for_reply_request_rtt\spxextraC`
`\spxentryxdr_truncate_encode\spxextraC` function, 595
function, 584 `\spxentryxprt_wake_pending_tasks\spxextraC`
`\spxentryxdr_write_pages\spxextraC` function, 593
function, 585 `\spxentryxprt_write_space\spxextraC`
`\spxentryxmit_async\spxextraC` function, 593
function, 482
`\spxentryxprt_adjust_cwnd\spxextraC` `\spxentryz8530_channel_load\spxextraC`
function, 592 function, 340
`\spxentryxprt_complete_rqst\spxextraC` `\spxentryz8530_describe\spxextraC`
function, 595 function, 339
`\spxentryxprt_disconnect_done\spxextraC` `\spxentryz8530_dma_rx\spxextraC`
function, 593 function, 344
`\spxentryxprt_force_disconnect\spxextraC` `\spxentryz8530_dma_status\spxextraC`
function, 593 function, 344
`\spxentryxprt_get\spxextraC` function, `\spxentryz8530_dma_tx\spxextraC` function,
595 function, 344
`\spxentryxprt_load_transport\spxextraC` `\spxentryz8530_flush_fifo\spxextraC`
function, 591 function, 343
`\spxentryxprt_lookup_rqst\spxextraC` `\spxentryz8530_init\spxextraC` function,
function, 594 340
`\spxentryxprt_pin_rqst\spxextraC` function, `\spxentryz8530_interrupt\spxextraC`
function, 594 function, 338
`\spxentryxprt_put\spxextraC` function, `\spxentryz8530_null_rx\spxextraC` function,
596 function, 340
`\spxentryxprt_reconnect_backoff\spxextraC` `\spxentryz8530_queue_xmit\spxextraC`
function, 594 function, 341
`\spxentryxprt_reconnect_delay\spxextraC` `\spxentryz8530_read_port\spxextraC`
function, 594 function, 341
`\spxentryxprt_register_transport\spxextraC` `\spxentryz8530_rtsdtr\spxextraC` function,
function, 590 343
`\spxentryxprt_release_rqst_cong\spxextraC` `\spxentryz8530_rx\spxextraC` function,
function, 592 343
`\spxentryxprt_release_xprt\spxextraC` `\spxentryz8530_rx_clear\spxextraC`
function, 591 function, 345
`\spxentryxprt_release_xprt_cong\spxextraC` `\spxentryz8530_rx_done\spxextraC`
function, 592 function, 346
`\spxentryxprt_request_get_cong\spxextraC` `\spxentryz8530_shutdown\spxextraC`
function, 592 function, 340
`\spxentryxprt_reserve_xprt\spxextraC` `\spxentryz8530_status\spxextraC` function,
344

`\spxentryz8530_status_clear\spxextraC`
function, [345](#)

`\spxentryz8530_sync_close\spxextraC`
function, [338](#)

`\spxentryz8530_sync_dma_close\spxextraC`
function, [339](#)

`\spxentryz8530_sync_dma_open\spxextraC`
function, [338](#)

`\spxentryz8530_sync_open\spxextraC`
function, [338](#)

`\spxentryz8530_sync_txdma_close\spxextraC`
function, [339](#)

`\spxentryz8530_sync_txdma_open\spxextraC`
function, [339](#)

`\spxentryz8530_tx\spxextraC` function,
[344](#)

`\spxentryz8530_tx_begin\spxextraC`
function, [345](#)

`\spxentryz8530_tx_clear\spxextraC`
function, [345](#)

`\spxentryz8530_tx_done\spxextraC`
function, [345](#)

`\spxentryz8530_write_port\spxextraC`
function, [341](#)

`\spxentryzerocopy_sg_from_iter\spxextraC`
function, [571](#)