µC/TCP-IP™

Protocol Stack

User's Manual

Micriµm

For the way Engineers work

Disclaimer

Specifications written in this manual are believed to be accurate, but are not guaranteed to be entirely free of error. Specifications in this manual may be changed for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, Micrium assumes no responsibility for any errors or omissions and makes no warranties. Micrium specifically disclaims any implied warranty of fitness for a particular purpose.

Registration

Please register the software via email. This way we can make sure you will receive updates or notifications of updates as soon as they become available. For registration please provide the following information:

- Your full name and the name of your supervisor
- Your company name
- Your job title
- Your email address and telephone number
- Company name and address
- Your company's main phone number
- Your company's web site address
- Name and version of the product

Please send this information to: **licensing@micrium.com**

Contact address

Micriµm 949 Crestview Circle, Weston, FL 33327-1848, U.S.A.
+1 954 217 2036 | FAX: +1 954 217 2037
www.micrium.com | support@micrium.com

Manual versions

If you find any errors in this document, please inform us and we will make the appropriate corrections for future releases.

Manual Version	Date	By	Description
V1.54 Rev. A	2004/12/09	JJL	Combined chapters in single document.
V1.54 Rev. B	2004/12/14	JJL	Added Configuration constants in Chapter 17
V1.56	2005/01/23	JJL	Added information about buffers and timers
V1.61	2005/02/10	JJL	Updated ASCII APIs because they were changed
V1.70	2005/04/24	JJL	device chapter revision
V1.71	2005/05/12	ITJ	Updated Configuration Chapter; Updated Socket error codes; Added Debug Management Chapter
V1.72	2005/07/11	ITJ	Updated device Receive/Transmit Sections; Updated Configuration Sections
V1.73	2005/08/04	ITJ	Corrected Socket Receive/Send Functional Descriptions
V1.80	2005/10/18	JJL	Minor changes to the manual
V1.81	2005/10/21	JJL	Added Delayed Acknowledge
V1.82	2005/12/13	ITJ	Updated Debug Management Chapter
V1.83	2005/12/21	ITJ	
V1.84	2006/04/14	ITJ	
V1.84	2006/04/26	JDH	Added information about example file usage
V1.85	2006/05/08	ITJ	Updated Debug Status Functions
V1.86	2006/08/08	ITJ	Corrected Socket API Descriptions
V1.87	2006/09/20	JJL	Updated Manual
V1.88	2006/11/27	ITJ	Updated Manual
V1.89	2007/03/08	ITJ	Corrected Socket API & Configuration Descriptions
V1.90	2007/05/24	ITJ	Updated Manual
V1.91	2007/10/24	ITJ	Updated Manual
V1.92	2008/07/15	ITJ	Updated Manual
V2.01	2008/12/01	ITJ	Updated Manual
V2.02	2009/01/06	EHS	Added Interface and Run-Time Configuration Chapters. All chapters reviewed and updated.
V2.02	2009/01/29	ITJ	Updated Manual
V2.02	2009/01/30	EHS	Added Section Describing Socket Error Codes.
V2.02	2009/02/01	ITJ	Added Appendices A, B, & C
V2.05	2009/06/05	SR	Updated Manual

Table of Contents

	Introduction	1
I.1	Portable	2
I.2	Scalable	2
I.3	Coding Standards	2
I.4	MISRA C	2
I.5	Safety Critical Certification	3
I.6	RTOS	3
I.7	Network Devices	4
I.8	**µC/TCP-IP** Protocols	4
I.9	Application Protocols	4
I.10	**µC/TCP-IP** Limitations	5

1	µC/TCP-IP Architecture	7
1.01	**µC/TCP-IP** Module Relationships	9
1.01.01	Your Application	9
1.01.02	**µC/LIB** Libraries	9
1.01.03	BSD Socket API Layer	10
1.01.04	TCP/IP Layer	10
1.01.05	Network Interface (IF) Layer	11
1.01.06	Network Device Drivers Layer	11
1.01.07	Network Physical (Phy) Layer	12
1.01.08	CPU Layer	12
1.01.09	Real-Time Operating System (RTOS) Layer	12
1.02	Directories	13
1.02.01	**µC/TCP-IP** Directories	13
1.02.02	Support Directories	15
1.02.03	Test Code Directories	16
1.03	Block Diagram	18
1.04	Task model	19
1.04.01	**µC/TCP-IP** Tasks and Priorities	19
1.04.02	Receiving a Packet	21
1.04.03	Transmitting a Packet	24

2	Getting Started with µC/TCP-IP	27
2.01	Installing **µC/TCP-IP**	27

2.02 .... **μC/TCP-IP** Example Project 28
2.02.01.......... BSP 30
2.02.02.......... Example OS-TCPIP-V2-APPS.......... 31

3 **μC/TCP-IP Configuration** **41**

3.01 .... Network Configuration 42
3.01.01.......... Network Configuration, NET_CFG_INIT_CFG_VALS 42
3.01.02.......... Network Configuration, NET_CFG_OPTIMIZE 44
3.01.03.......... Network Configuration, NET_CFG_OPTIMIZE_ASM_EN.......... 44
3.02 .... Debug Configuration.......... 44
3.02.01.......... Debug Configuration, NET_DBG_CFG_INFO_EN 44
3.02.02.......... Debug Configuration, NET_DBG_CFG_STATUS_EN 45
3.02.03.......... Debug Configuration, NET_DBG_CFG_MEM_CLR_EN 45
3.02.04.......... Debug Configuration, NET_DBG_CFG_TEST_EN 46
3.03 .... Argument Checking Configuration.......... 46
3.03.01.......... Argument Checking Configuration, NET_ERR_CFG_ARG_CHK_EXT_EN 46
3.03.02.......... Argument Checking Configuration, NET_ERR_CFG_ARG_CHK_DBG_EN.......... 47
3.04 .... Network Counter Configuration.......... 47
3.04.01.......... Network Counter Configuration, NET_CTR_CFG_STAT_EN 47
3.04.02.......... Network Counter Configuration, NET_CTR_CFG_ERR_EN 47
3.05 .... Network Timer Configuration.......... 48
3.05.01.......... Network Timer Configuration, NET_TMR_CFG_NBR_TMR.......... 48
3.05.02.......... Network Timer Configuration, NET_TMR_CFG_TASK_FREQ 48
3.06 .... Network Buffer Configuration 49
3.07 .... Network Interface Layer Configuration 49
3.07.01.......... Network Interface Layer Configuration, NET_IF_CFG_MAX_NBR_IF.......... 49
3.07.02.......... Network Interface Layer Configuration, NET_IF_CFG_ADDR_FLTR_EN 49
3.07.03.......... Network Interface Layer Configuration,
NET_IF_CFG_TX_SUSPEND_TIMEOUT_MS.......... 49
3.07.04.01...... Network Interface Layer Configuration, NET_IF_CFG_LOOPBACK_EN.......... 50
3.07.04.02...... Network Interface Layer Configuration, NET_IF_CFG_ETHER_EN.......... 50
3.08 .... ARP (Address Resolution Protocol) Configuration 51
3.08.01.......... ARP Configuration, NET_ARP_CFG_HW_TYPE.......... 51
3.08.02.......... ARP Configuration, NET_ARP_CFG_PROTOCOL_TYPE.......... 51
3.08.03.......... ARP Configuration, NET_ARP_CFG_NBR_CACHE 51
3.08.04.......... ARP Configuration, NET_ARP_CFG_ADDR_FLTR_EN.......... 52
3.09 .... IP (Internet Protocol) Configuration 52

3.09.01 .... IP Configuration, NET_IP_CFG_IF_MAX_NBR_ADDR 52
3.09.02 .... IP Configuration, NET_IP_CFG_MULTICAST_SEL 52
3.10 .... ICMP (Internet Control Message Protocol) Configuration 53
3.10.01 .... ICMP Configuration, NET_ICMP_CFG_TX_SRC_QUENCH_EN 53
3.10.02 .... ICMP Configuration, NET_ICMP_CFG_TX_SRC_QUENCH_NBR 53
3.11 .... IGMP (Internet Group Management Protocol) Configuration 54
3.11.01 .... IGMP Configuration, NET_IGMP_CFG_MAX_NBR_HOST_GRP 54
3.12 .... Transport Layer Configuration,NET_CFG_TRANSPORT_LAYER_SEL 54
3.13 .... UDP (User Datagram Protocol) Configuration 55
3.13.01 .... UDP Configuration, NET_UDP_CFG_APP_API_SEL 55
3.13.02 .... UDP Configuration, NET_UDP_CFG_RX_CHK_SUM_DISCARD_EN 56
3.13.03 .... UDP Configuration, NET_UDP_CFG_TX_CHK_SUM_EN 56
3.14 .... TCP (Transport Control Protocol) Configuration 57
3.14.01 .... TCP Configuration, NET_TCP_CFG_NBR_CONN 57
3.14.02 .... TCP Configuration, NET_TCP_CFG_RX_WIN_SIZE_OCTET 57
3.14.03 .... TCP Configuration, NET_TCP_CFG_TX_WIN_SIZE_OCTET 57
3.14.04 .... TCP Configuration, NET_TCP_CFG_TIMEOUT_CONN_MAX_SEG_SEC 58
3.14.05 .... TCP Configuration, NET_TCP_CFG_TIMEOUT_CONN_ACK_DLY_MS 58
3.14.06 .... TCP Configuration, NET_TCP_CFG_TIMEOUT_CONN_RX_Q_MS 58
3.14.07 .... TCP Configuration, NET_TCP_CFG_TIMEOUT_CONN_TX_Q_MS 59
3.15 .... BSD v4 Sockets Configuration 59
3.15.01 .... BSD v4 Sockets Configuration, NET_SOCK_CFG_FAMILY 59
3.15.02 .... BSD v4 Sockets Configuration, NET_SOCK_CFG_NBR_SOCK 59
3.15.03 .... BSD v4 Sockets Configuration, NET_SOCK_CFG_BLOCK_SEL 59
3.15.04 .... BSD v4 API Configuration, NET_SOCK_CFG_SEL_EN 60
3.15.05 .... BSD v4 Sockets Configuration,
NET_SOCK_CFG_SEL_NBR_EVENTS_MAX 60
3.15.06 .... BSD v4 Sockets Configuration, NET_SOCK_CFG_CONN_ACCEPT_Q_SIZE_MAX ... 61
3.15.07 .... BSD v4 Sockets Configuration, NET_SOCK_CFG_PORT_NBR_RANDOM_BASE 61
3.15.08 .... BSD v4 Sockets Configuration, NET_SOCK_CFG_TIMEOUT_RX_Q_MS 61
3.15.09 .... BSD v4 Sockets Configuration, NET_SOCK_CFG_TIMEOUT_CONN_REQ_MS 61
3.15.10 .... BSD v4 Sockets Configuration,
NET_SOCK_CFG_TIMEOUT_CONN_ACCEPT_MS 62
3.15.11 .... BSD v4 Sockets Configuration, NET_SOCK_CFG_TIMEOUT_CONN_CLOSE_MS 62
3.15.12 .... BSD v4 Sockets Configuration, NET_BSD_CFG_API_EN 62
3.16 .... Network Connection Manager Configuration 62
3.16.01 .... Network Connection Manager Configuration,
NET_CONN_CFG_FAMILY 62
3.16.02 .... Network Connection Manager Configuration,
NET_CONN_CFG_NBR_CONN 63

3.17 .... Application-Specific Configuration, app_cfg.h. 63
3.17.01......... Application-Specific Configuration, Operating System Configuration.......... 64
3.17.02......... Application-Specific Configuration, **µC/TCP-IP** Configuration 65

4 Network Interface Layer 67

4.01 .... Network Interface Configuration 68
4.01.01......... Interface / Device / Phy Configuration 68
4.01.01.01...... Loopback Configuration 68
4.01.01.02.01 ... Ethernet Device MAC Configuration 71
4.01.01.02.02 ... Ethernet Phy Configuration 75
4.01.02......... Adding Network Interfaces 77
4.01.03......... Configuring an Internet Protocol Address 79
4.02 .... Starting & Stopping Network Interfaces 81
4.02.01......... Starting Network Interfaces 81
4.02.02......... Stopping Network Interfaces 82
4.03 .... Network Interfaces' MTU. 82
4.03.01......... Getting Network Interface MTU 82
4.03.02......... Setting Network Interface MTU 83
4.04 .... Network Interfaces' Hardware Addresses 84
4.04.01......... Getting Network Interface Hardware Address. 84
4.04.02......... Setting Network Interface Hardware Address 85
4.05 .... Obtaining Link State 86

5 Network Device Drivers 89

6 Network Socket Interface 91

6.01 .... Network Socket Data Structures. 92
6.02 .... Example Socket Applications 95
6.02.01......... UDP Socket Calls 95
6.02.02......... TCP Socket Calls. 96
6.03 .... **µC/TCP-IP** Socket API List 97
6.04 .... **µC/TCP-IP** Socket Error Codes 97
6.04.01......... Fatal Socket Error Codes 97
6.04.02......... Socket Error Code List. 97

7 Buffer Management 99

7.01 .... Network Buffer Architecture 100

8 Timer Management 103

9 Statistics and Error Counters 105

9.01 .... Statistics 105
9.02 .... Error Counters 107

10 Debug Management 109

10.01 .... Network Debug Information Constants 109
10.02 .... Network Debug Monitor Task 110

Appendix A µC/TCP-IP Application Programming Interface (API) Functions & Macro's 111

A.01 .... General Network Functions 112
A.01.01 .... Net_Init() 112
A.01.02 .... Net_InitDflt() 113
A.01.03 .... Net_VersionGet() 114
A.02 .... Network-to-Application Functions 115
A.03 .... ARP Functions 116
A.03.01 .... NetARP_CacheCalcStat() 116
A.03.02 .... NetARP_CacheGetAddrHW() 117
A.03.03 .... NetARP_CachePoolStatGet() 119
A.03.04 .... NetARP_CachePoolStatResetMaxUsed() 120
A.03.05 .... NetARP_CfgCacheAccessedTh() 121
A.03.06 .... NetARP_CfgCacheTimeout() 122
A.03.07 .... NetARP_CfgReqMaxRetries() 123
A.03.08 .... NetARP_CfgReqTimeout() 124
A.03.09 .... NetARP_IsAddrProtocolConflict() 124
A.03.10 .... NetARP_ProbeAddrOnNet() 126
A.04 .... Network ASCII Functions 128
A.04.01 .... NetASCII_IP_to_Str() 128
A.04.02 .... NetASCII_MAC_to_Str() 130
A.04.03 .... NetASCII_Str_to_IP() 132

A.04.04 .... NetASCII_Str_to_MAC() 134
A.05 .... Network Buffer Functions 136
A.05.01 .... NetBuf_PoolStatGet() 136
A.05.02 .... NetBuf_PoolStatResetMaxUsed() 137
A.05.03 .... NetBuf_RxLargePoolStatGet() 138
A.05.04 .... NetBuf_RxLargePoolStatResetMaxUsed() 139
A.05.05 .... NetBuf_TxLargePoolStatGet() 140
A.05.06 .... NetBuf_TxLargePoolStatResetMaxUsed() 141
A.05.07 .... NetBuf_TxSmallPoolStatGet() 142
A.05.08 .... NetBuf_TxSmallPoolStatResetMaxUsed() 143
A.06 .... Network Connection Functions 144
A.06.01 .... NetConn_CfgAccessedTh() 144
A.06.02 .... NetConn_PoolStatGet() 145
A.06.03 .... NetConn_PoolStatResetMaxUsed() 146
A.07 .... Network Debug Functions 147
A.07.01 .... NetDbg_CfgMonTaskTime() 147
A.07.02 .... NetDbg_CfgRsrcARP_CacheThLo() 148
A.07.03 .... NetDbg_CfgRsrcBufThLo() 149
A.07.04 .... NetDbg_CfgRsrcBufRxLargeThLo() 150
A.07.05 .... NetDbg_CfgRsrcBufTxLargeThLo() 151
A.07.06 .... NetDbg_CfgRsrcBufTxSmallThLo() 152
A.07.07 .... NetDbg_CfgRsrcConnThLo() 153
A.07.08 .... NetDbg_CfgRsrcSockThLo() 154
A.07.09 .... NetDbg_CfgRsrcTCP_ConnThLo() 155
A.07.10 .... NetDbg_CfgRsrcTmrThLo() 156
A.07.11 .... NetDbg_ChkStatus() 157
A.07.12 .... NetDbg_ChkStatusBufs() 158
A.07.13 .... NetDbg_ChkStatusConns() 159
A.07.14 .... NetDbg_ChkStatusRsrcLost() / NetDbg_MonTaskStatusGetRsrcLost() 161
A.07.15 .... NetDbg_ChkStatusRsrcLo() / NetDbg_MonTaskStatusGetRsrcLo() 162
A.07.16 .... NetDbg_ChkStatusTCP() 163
A.07.17 .... NetDbg_ChkStatusTmrs() 165
A.07.18 .... NetDbg_MonTaskStatusGetRsrcLost() 167
A.07.19 .... NetDbg_MonTaskStatusGetRsrcLo() 167
A.08 .... ICMP Functions 168
A.08.01 .... NetICMP_CfgTxSrcQuenchTh() 168
A.09 .... Network Interface Functions 169
A.09.01 .... NetIF_Add() 169
A.09.02 .... NetIF_AddrHW_Get() 170

A.10 .... Network Interface Functions 171
A.10.01 .... NetIF_Add() 171
A.10.02 .... NetIF_AddrHW_Get() 173
A.10.03 .... NetIF_AddrHW_IsValid() 174
A.10.04 .... NetIF_AddrHW_Set() 175
A.10.05 .... NetIF_CfgPerfMonPeriod() 177
A.10.06 .... NetIF_CfgPhyLinkPeriod() 178
A.10.07 .... NetIF_IO_Ctrl() 179
A.10.08 .... NetIF_IsEn() 180
A.10.09 .... NetIF_IsEnCfgd() 181
A.10.10 .... NetIF_IsValid() 182
A.10.11 .... NetIF_IsValidCfgd() 183
A.10.12 .... NetIF_LinkStateGet() 184
A.10.13 .... NetIF_MTU_Get() 185
A.10.14 .... NetIF_MTU_Set() 186
A.10.15 .... NetIF_Start() 187
A.10.16 .... NetIF_Stop() 188
A.11 .... IP Functions 189
A.11.01 .... NetIP_CfgAddrAdd() 189
A.11.02 .... NetIP_CfgAddrAddDynamic() 191
A.11.03 .... NetIP_CfgAddrAddDynamicStart() 193
A.11.04 .... NetIP_CfgAddrAddDynamicStop() 194
A.11.05 .... NetIP_CfgAddrRemove() 195
A.11.06 .... NetIP_CfgAddrRemoveAll() 196
A.11.07 .... NetIP_CfgFragReasmTimeout() 197
A.11.08 .... NetIP_GetAddrDfltGateway() 198
A.11.09 .... NetIP_GetAddrHost() 199
A.11.10 .... NetIP_GetAddrSubnetMask() 201
A.11.11 .... NetIP_IsAddrBroadcast() 202
A.11.12 .... NetIP_IsAddrClassA() 203
A.11.13 .... NetIP_IsAddrClassB() 204
A.11.14 .... NetIP_IsAddrClassC() 205
A.11.15 .... NetIP_IsAddrHost() 206
A.11.16 .... NetIP_IsAddrHostCfgd() 207
A.11.17 .... NetIP_IsAddrLocalHost() 208
A.11.18 .... NetIP_IsAddrLocalLink() 209
A.11.19 .... NetIP_IsAddrsCfgdOnIF() 210
A.11.20 .... NetIP_IsAddrThisHost() 211
A.11.21 .... NetIP_IsValidAddrHost() 212

A.11.22 .... NetIP_IsValidAddrHostCfgd() 213
A.11.23 .... NetIP_IsValidAddrSubnetMask() 214
A.12 .... Network Socket Functions 215
A.12.01 .... NetSock_Accept() / accept() TCP 215
A.12.02 .... NetSock_Bind() / bind() TCP/UDP 217
A.12.03 .... NetSock_CfgTimeoutConnAcceptDflt() TCP 220
A.12.04 .... NetSock_CfgTimeoutConnAcceptGet_ms() TCP 221
A.12.05 .... NetSock_CfgTimeoutConnAcceptSet() TCP 222
A.12.06 .... NetSock_CfgTimeoutConnCloseDflt() TCP 223
A.12.07 .... NetSock_CfgTimeoutConnCloseGet_ms() TCP 224
A.12.08 .... NetSock_CfgTimeoutConnCloseSet() TCP 225
A.12.09 .... NetSock_CfgTimeoutConnReqDflt() TCP 226
A.12.10 .... NetSock_CfgTimeoutConnReqGet_ms() TCP 227
A.12.11 .... NetSock_CfgTimeoutConnReqSet() TCP 228
A.12.12 .... NetSock_CfgTimeoutRxQ_Dflt() TCP/UDP 229
A.12.13 .... NetSock_CfgTimeoutRxQ_Get_ms() TCP/UDP 230
A.12.14 .... NetSock_CfgTimeoutRxQ_Set() TCP/UDP 231
A.12.15 .... NetSock_CfgTimeoutTxQ_Dflt() TCP 233
A.12.16 .... NetSock_CfgTimeoutTxQ_Get_ms() TCP 234
A.12.17 .... NetSock_CfgTimeoutTxQ_Set() TCP 235
A.12.18 .... NetSock_Close() / close() TCP/UDP 238
A.12.19 .... NetSock_Conn() / connect() TCP/UDP 239
A.12.20 .... NET_SOCK_DESC_CLR() / FD_CLR() TCP/UDP 242
A.12.21 .... NET_SOCK_DESC_COPY() TCP/UDP 243
A.12.22 .... NET_SOCK_DESC_ INIT() / FD_ZERO() TCP/UDP 244
A.12.23 .... NET_SOCK_DESC_IS_SET() / FD_IS_SET() TCP/UDP 245
A.12.24 .... NET_SOCK_DESC_SET() / FD_SET() TCP/UDP 246
A.12.25 .... NetSock_GetConnTransportID() 247
A.12.26 .... NetSock_IsConn() 248
A.12.27 .... NetSock_Listen() / listen() TCP 249
A.12.28 .... NetSock_Open() / socket() TCP/UDP 251
A.12.29 .... NetSock_PoolStatGet() 254
A.12.30 .... NetSock_PoolStatResetMaxUsed() 255
A.12.31 .... NetSock_RxData() / recv()TCP/UDP NetSock_RxDataFrom() / recvfrom() TCP/UDP .. 256
A.12.32 .... NetSock_Sel() / select() TCP/UDP 260
A.12.33 .... NetSock_TxData() / send() TCP/UDP NetSock_TxDataTo() / sendto() TCP/UDP 263
A.13 .... TCP Functions 267
A.13.01 .... NetTCP_ConnCfgMaxSegSizeLocal() 267
A.13.02 .... NetTCP_ConnCfgReTxMaxTh() 269

A.13.03 .... NetTCP_ConnCfgReTxMaxTimeout() 270
A.13.04 .... NetTCP_ConnCfgRxWinSize() 272
A.13.05 .... NetTCP_ConnCfgTxAckImmedRxdPushEn() 273
A.13.06 .... NetTCP_ConnPoolStatGet() 275
A.13.07 .... NetTCP_ConnPoolStatResetMaxUsed() 276
A.13.08 .... NetTCP_InitTxSeqNbr() 277
A.14 .... Network Timer Functions 278
A.14.01 .... NetTmr_PoolStatGet() 278
A.14.02 .... NetTmr_PoolStatResetMaxUsed() 279
A.15 .... UDP Functions 280
A.15.01 .... NetUDP_RxAppData() 280
A.15.02 .... NetUDP_RxAppDataHandler() 282
A.15.03 .... NetUDP_TxAppData() 284
A.16 .... General Network Utility Functions 287
A.16.01 .... NET_UTIL_HOST_TO_NET_16() 287
A.16.02 .... NET_UTIL_ HOST_TO_NET_32() 288
A.16.03 .... NET_UTIL_NET_TO_HOST_16() 289
A.16.04 .... NET_UTIL_NET_TO_HOST_32() 290
A.16.05 .... NetUtil_TS_Get() 291
A.16.06 .... NetUtil_TS_Get_ms() 292
A.17 .... BSD Functions 293
A.17.01 .... accept() TCP 293
A.17.02 .... bind() TCP/UDP 294
A.17.03 .... close() TCP/UDP 295
A.17.04 .... connect() TCP/UDP 296
A.17.05 .... FD_CLR() TCP/UDP 297
A.17.06 .... FD_ISSET() TCP/UDP 297
A.17.07 .... FD_SET() TCP/UDP 298
A.17.08 .... FD_ZERO() TCP/UDP 298
A.17.09 .... htonl() 299
A.17.10 .... htons() 299
A.17.11 .... inet_addr() IPv4 300
A.17.12 .... inet_ntoa() IPv4 302
A.17.13 .... listen() TCP 304
A.17.14 .... ntohl() 305
A.17.15 .... ntohs() 305
A.17.16 .... recv() / recvfrom() TCP/UDP 306
A.17.17 .... select() TCP/UDP 307
A.17.18 .... send() / sendto() TCP/UDP 308
A.17.19 .... socket() TCP/UDP 309

Appendix B µC/TCP-IP Error Codes 311

B.01 .... Network Error Codes 312
B.02 .... ARP Error Codes 312
B.03 .... Network ASCII Error Codes 312
B.04 .... Network Buffer Error Codes 313
B.05 .... ICMP Error Codes 313
B.06 .... Network Interface Error Codes 313
B.07 .... IP Error Codes 314
B.08 .... IGMP Error Codes 314
B.09 .... OS Error Codes 315
B.10 .... Network Socket Error Codes 315
B.11 .... UDP Error Codes 316

Appendix C µC/TCP-IP Frequently Asked Questions (FAQ) 317

C.01 .... **µC/TCP-IP** Licensing? 317
C.01.01 .... How do I obtain **µC/TCP-IP** source code? 317
C.01.02 .... How do I license **µC/TCP-IP**? 317
C.01.03 .... How do I obtain **µC/TCP-IP** source code updates? 317
C.01.04 .... How do I obtain support for **µC/TCP-IP**? 318
C.01.05 .... How do I renew maintenance for **µC/TCP-IP**? 318
C.02 .... **µC/TCP-IP** Configuration and Initialization 319
C.02.01 .... How do I configure the **µC/TCP-IP** stack? 319
C.02.02 .... How do I know how large to configure the **µC/LIB** memory heap? 319
C.02.03 .... How do I know how large to make the **µC/TCP-IP** task stacks? 322
C.02.04 .... How do I configure **µC/TCP-IP** task priorities? 322
C.02.05 .... How do I configure **µC/TCP-IP** queue sizes? 322
C.02.06 .... How do I initialize **µC/TCP-IP**? 323
C.03 .... Network Interfaces, Devices, and Buffers 327
C.03.01 .... Network Interface Configuration 327
C.03.01.01 ..... How do I add an interface? 327
C.03.01.02 ..... How do I start an interface? 327
C.03.01.03 ..... How do I stop an interface? 327
C.03.01.04 ..... How do I check if an interface is enabled? 327
C.03.02 .... Network and Device Buffer Configuration 328
C.03.03.01 ..... Why are large transmit buffers 1594 (or 1614) bytes? 328
C.03.03.02 ..... How do I determine how many Rx or 328
C.03.02.03 ..... How do I determine how many 329

C.03.02.04 .....How do I write or obtain additional device drivers? 330
C.03.03 ....Ethernet MAC Address 330
C.03.03.01 .....How do I obtain the MAC address of an interface? 330
C.03.03.02 .....How do I change the MAC address of an interface? 330
C.03.03.03 .....How do I obtain the MAC address 331
C.03.04 ....Ethernet Phy Link State..333
C.03.04.01 .....How do I increase the rate of link state polling? 333
C.03.04.02 .....How do I obtain the current link state for an interface? 333
C.03.04.03 .....How do I force an Ethernet Phy to a specific link state? 335
C.04 ....IP Address Configuration 336
C.04.01 ....How do I convert IP addresses to and from their dotted decimal representation? ... 336
C.04.02 ....How do I statically assign one or more IP addresses to an interface? 336
C.04.03 ....How do I remove one or more statically assigned IP addresses from an interface?.... 337
C.04.04 ....How do I get a dynamical IP address? 337
C.04.05 ....How do I obtain all the IP addresses configured on a specific interface? 337
C.05 ....Socket Programming... 338
C.05.01 ....How do I use the Micrium socket to develop an application?................. 338
C.05.02 ....How do I join and leave an IGMP host group? 338
C.05.03 ....How do I transmit to a multicast IP group address?......................... 339
C.05.04 ....How do I receive from a multicast IP group?............................... 339
C.05.05 ....Why does my application receive socket errors immediately after reboot? 340
C.05.06 ....How do I reduce the number of transitory errors (NET_ERR_TX)? 340
C.05.07 ....How do I control socket blocking options?................................ 341
C.05.08 ....How do I tell if a socket is still connected to a peer? 341
C.05.09 ....Why do I receive -1 instead of 0 when calling recv() for a closed socket?........ 342
C.05.10 ....How do I determine which interface a UDP datagram was received on? 342
C.06 ....**µC/TCP-IP** Statistics and Debug 343
C.06.01 ....How do I obtain performance statistics during run-time? 343
C.06.02 ....How do I view error and statistics counters?............................... 344
C.06.03 ....How do I use network debug functions to check network status conditions?.... 344
C.07 ....**µC/TCP-IP** Optimization 345
C.07.01 ....How do I optimize _C/TCP-IP for additional performance? 345
C.08 ....Miscellaneous... 346
C.08.01 ....How do I send and receive ICMP Echo Requests from the target? 346
C.08.02 ....How do I enable TCP Keep-Alives?....................................... 346
C.08.03 ....Can I use _C/TCP-IP for inter-process communication? 346

Appendix D µC/TCP-IP Licensing Policy 345

Introduction

µC/TCP-IP is a compact, reliable, high performance TCP/IP protocol stack. Built from the ground up with Micrium's renowned quality, scalability and reliability, **µC/TCP-IP** enables the rapid configuration of required network options to minimize your time to market. **µC/TCP-IP** is the result of many man-years of development.

The source code for **µC/TCP-IP** contains over 100,000 lines of the cleanest, most consistent ANSI C source code you will ever find in a TCP/IP stack implementation. C was chosen because C is still the most predominant language in the embedded industry. Over 50% of the code actually consists of comments. Most global variables and all functions are described. References to RFC (Request For Comments) are referenced in the code when applicable.

I.1 Portable

µC/TCP-IP was designed for resource constrained embedded applications. The code is designed to be used with just about any CPU, RTOS and network devices. Although **µC/TCP-IP** can work on some 8 and 16-bit processors, **µC/TCP-IP** will work best with 32 or 64-bit CPUs.

I.2 Scalable

The memory footprint of **µC/TCP-IP** can be adjusted at compile time based on the features you need and the desired level of run-time argument checking. Also, throughout **µC/TCP-IP**, we keep track of statistics many of which may be disabled in order to further reduce the footprint.

I.3 Coding Standards

Coding standards have been established early in the design of **µC/TCP-IP** and include the following:

- C coding style
- Naming convention for #define constants, macros, variables and functions
- Commenting
- Directory structure

These conventions make **µC/TCP-IP** the cleanest TCP/IP stack implementation in the industry and makes it easier to attain third party certification as outlined in the next section.

I.4 MISRA C

The source code for **µC/TCP-IP** follows the Motor Industry Software Reliability Association (MISRA) C Coding Standards. These standards were created by MISRA to improve the reliability and predictability of C programs in critical automotive systems. Members of the MISRA consortium include Delco Electronics, Ford Motor Company, Jaguar Cars Ltd., Lotus Engineering, Lucas

Electronics, Rolls-Royce, Rover Group Ltd., and other firms and universities dedicated to improving safety and reliability in automotive electronics. Full details of this standard can be obtained directly from the MISRA web site, www.misra.org.uk.

I.5 Safety Critical Certification

µC/TCP-IP has been designed from the ground up to be certifiable for use in avionics as well as medical and other safety critical products. A company called Validated Software is preparing a Validation Suite(tm) for **µC/TCP-IP** to provide all of the documentation necessary to deliver **µC/TCP-IP** as a pre-certifiable software component for safety critical systems, including avionics RTCA DO-178B and EUROCAE ED-12B, medical FDA 510(k), IEC 61508 industrial control systems, and EN-50128 rail transportation and nuclear systems. The very affordable Validation Suite(tm), is available through Validated Software. It will be immediately certifiable for the highest criticality systems, including DO-178B Level A, Class III medical devices, and SIL3/SIL4 IEC-certified systems. For more information, check out the **µC/TCP-IP** page on the Validated Software web site www.ValidatedSoftware.com.

If your product is NOT safety critical, you should view the certification as proof that **µC/TCP-IP** is a very robust and highly reliable TCP/IP stack.

I.6 RTOS

µC/TCP-IP assumes the presence of an RTOS. However, we do not make any assumptions about which RTOS you will be using with **µC/TCP-IP**. The only requirements from the RTOS are:

1) The RTOS must be able to support multiple tasks
2) The RTOS must provide binary and counting semaphore management services
3) The RTOS must provide message queue services

µC/TCP-IP contains an encapsulation layer that allows you to use just about any commercial or open source RTOS. In other words, details about the RTOS you use are hidden from **µC/TCP-IP**. **µC/TCP-IP** includes the encapsulation layer for **µC/OS-II**, The Real-Time Kernel.

I.7 Network Devices

µC/TCP-IP may be configured with multiple network devices and network (IP) addresses. Any device may be used provided a driver with the appropriate API and BSP software is provided. The API for a specific device (i.e. chip) is encapsulated in a couple of files and it is quite easy to adapt different devices to **µC/TCP-IP** (see Chapter 4).

Currently, only Ethernet devices are supported. However, we are currently working on adding PPP (Point-to-Point Protocol) support to **µC/TCP-IP**.

I.8 µC/TCP-IP Protocols

µC/TCP-IP consists of the following protocols:

- Device drivers
- Network Interfaces (e.g. Ethernet, PPP (TBA), etc.)
- ARP (Address Resolution Protocol)
- IP (Internet Protocol)
- ICMP (Internet Control Message Protocol)
- IGMP (Internet Group Management Protocol)
- UDP (User Datagram Protocol)
- TCP (Transport Control Protocol)
- Sockets (Micrium and BSD v4)

I.9 Application Protocols

µC/TCP-IP can work with well known application layer protocols such as DHCP, DNS, TFTP, HTTP servers (web server), FTP servers, SNTP clients, SNMP clients and more.

I.10 µC/TCP-IP Limitations

By design, we have limited some of the features of **µC/TCP-IP**. Table I-1 describes those limitations.

No IP forwarding / routing
No IP transmit fragmentation
No IP Security options & RFC #1108
No ICMP Address Mask Agent / Server & RFC #1122, Section 3.2.2.9
No TCP Urgent Data
No TCP Security & Precedence

Table I-1, *µ*C/TCP-IP limitations for current software version

Chapter

1

μC/TCP-IP Architecture

μC/TCP-IP was written from the ground up to be modular and easy to adapt to different CPUs (Central Processing Units), RTOSs (Real-Time Operating Systems), network devices and compilers. Figure 1-1 shows a simplified block diagram of the various different **μC/TCP-IP** modules and their relationships.

Notice that all of the **μC/TCP-IP** files start with '`net_`'. This convention allows you to quickly identify which files belong to **μC/TCP-IP**. Also note that all functions and global variables start with '`Net`', and all macros and `#defines` start with '`net_`'.

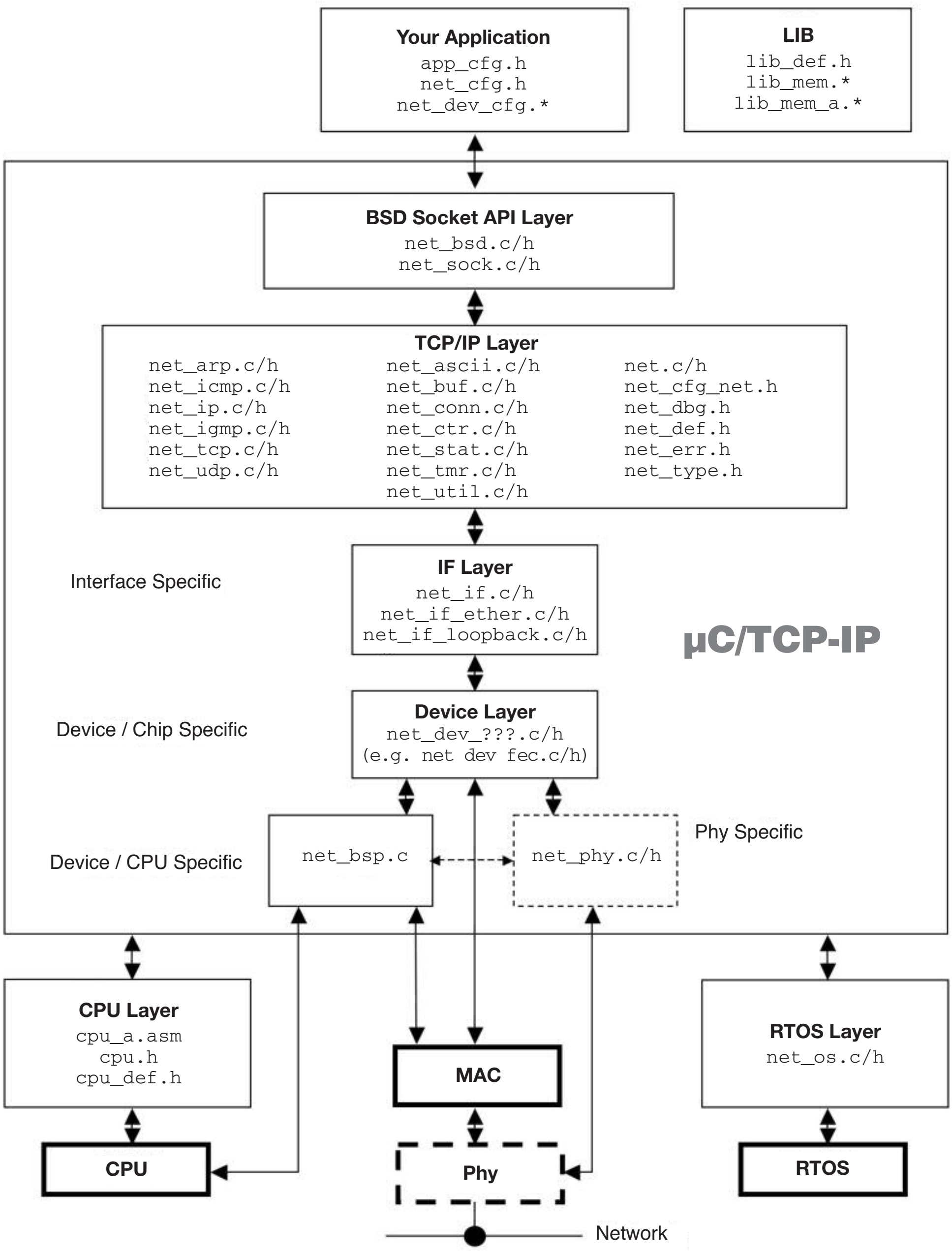

Figure 1-1, Module Relationships

1.01 µC/TCP-IP Module Relationships

1.01.01 Your Application

Your application needs to provide configuration information to **µC/TCP-IP** in the form of four C files named `app_cfg.h`, `net_cfg.h`, `net_dev_cfg.c` and `net_dev_cfg.h`.

`app_cfg.h` is an application specific configuration file that MUST be present in your application. `app_cfg.h` contains `#defines` to specify the task priorities of each of the tasks in your application (including those of **µC/TCP-IP**) as well as the stack size for those tasks. The reason all task priorities are placed in one file is to make it easier to 'see' the task priorities for your entire application in one place.

Some of the configuration data in `net_cfg.h` consist of specifying the number of timers to allocate to the stack, whether statistic counters will be maintained or not, the number of ARP cache entries, whether UDP checksums are computed or not and more. In all, there are about 50 `#define` to set. However, most of the `#define` constants can be set to their recommended default value.

Lastly, `net_dev_cfg.c` consists of device specific configuration such as the number of buffers allocated to a device, the MAC address for that device and any required physical layer device configuration including physical layer device bus address and link characteristics. Each **µC/TCP-IP** compatible device requires its configuration to be specified within `net_dev_cfg.c`.

1.01.02 µC/LIB Libraries

Because **µC/TCP-IP** is designed to be used in safety critical applications, all 'standard' library functions like `strcpy()`, `memset()`, etc. have been re-written to follow the same quality as the rest as the protocol stack.

1.01.03 BSD Socket API Layer

Your application interfaces to **µC/TCP-IP** using the well known BSD socket API (Application Programming Interface). You can either write your own TCP/IP applications using the BSD socket API or, you can purchase a number of off-the-shelf TCP/IP components (Telnet, Web server, FTP server, etc.) which all interface to the BSD socket interface. Note that the BSD socket layer is shown as a separate module but is actually part of **µC/TCP-IP**.

Alternatively, you can use **µC/TCP-IP**'s own socket interface functions (`net_sock.*`). Basically, `net_bsd.*` is a layer of software converting BSD socket calls to **µC/TCP-IP** socket calls. Of course, you would have a slight performance gain by interfacing directly to `net_sock.*` functions. Micrium network products use the **µC/TCP-IP** socket interface functions.

1.01.04 TCP/IP Layer

This layer contains most of the CPU, RTOS and compiler independent code for **µC/TCP-IP**. There are three categories of files in this section:

1) TCP/IP protocol specific files
   ARP (`net_arp.*`),
   ICMP (`net_icmp.*`),
   IGMP (`net_igmp.*`),
   IP (`net_ip.*`),
   TCP (`net_tcp.*`),
   UDP (`net_udp.*`)

2) Support files
   ASCII conversions (`net_ascii.*`),
   Buffer management (`net_buf.*`),
   TCP/UDP connection management (`net_conn.*`),
   Counter management (`net_ctr.*`),
   Statistics (`net_stat.*`),
   Timer Management (`net_tmr.*`),
   other utilities (`net_util.*`).

3) Miscellaneous header files
   Master **µC/TCP-IP** header file (`net.h`)

File containing error codes (`net_err.h`)
Miscellaneous **μC/TCP-IP** data types (`net_type.h`)
Miscellaneous definitions (`net_def.h`)
Debug (`net_dbg.h`)
Configuration definitions (`net_cfg_net.h`)

1.01.05 Network Interface (IF) Layer

The IF Layer understands about types of network interfaces (Ethernet, Token Ring, etc.). However, the current version of **μC/TCP-IP** only supports Ethernet interfaces. The IF layer is actually split into two sub-layers.

`net_if.*` is the interface between higher Network Protocol Suite layers and the link layer protocols. This layer also provides network device management routines to the application.

`net_if_*.*` contains the link layer protocol specifics independent of the actual device (i.e. hardware). In other words, for Ethernet, `net_if_ether.*` understands about Ethernet frames, MAC addresses, frame de-multiplexing, and so on but, assumes nothing about actual Ethernet hardware.

1.01.06 Network Device Drivers Layer

μC/TCP-IP can work with just about any network device. This layer knows about the specifics of the hardware, e.g. how to initialize the device, how to enable and disable interrupts from the device, how to find the size of a received packet, how to read a packet out of the frame buffer, how to write a packet to the device, and more.

In order for device drivers to have independent configuration for clock gating, interrupt controller, and general purpose IO, we add an additionnal file, `net_bsp.c`, to encapsulate such details.

`net_bsp.c` contains code for configuring clock gating to the device, configuring an internal or external interrupt controller, configuring necessary IO pins, time delays, obtaining a time stamp from the environment and so on. This file is assumed to reside in your application.

1.01.07 Network Physical (Phy) Layer

Some devices interface to external physical layer devices which may need to be initialized and controlled. This layer is shown in Figure 1-1 as 'dotted' indicating that it is not present with all devices. In fact, some devices have Phy control built-in. Micrium provides a generic Phy driver which can sufficiently control most external (R)MII compliant Ethernet physical layer devices.

1.01.08 CPU Layer

µC/TCP-IP can work with either an 8, 16, 32 or even 64-bit CPU but, needs to have information about the CPU you are using. The CPU layer defines such things as the C data type corresponding to 16-bit and 32-bit variables, whether the CPU is little or big endian and, how interrupts are disabled and enabled on the CPU, etc.

CPU specific files are found in the **…\uC-CPU** directory and, in order to adapt **µC/TCP-IP** to a different CPU, you would need to either modify the `cpu*.*` files or, create new ones based on the ones supplied in the **uC-CPU** directory. In general, it is much easier to modify existing files because you have a better chance of not forgetting anything.

1.01.09 Real-Time Operating System (RTOS) Layer

µC/TCP-IP assumes the presence of an RTOS but, the RTOS layer allows **µC/TCP-IP** to be independent of any specific RTOS. **µC/TCP-IP** consists of three tasks. One task is responsible for handling packet reception, another task is responsible for asynchronous transmit buffer de-allocation, and the last task is responsible for managing timers. Depending on the configuration, a fourth task may be present for handling loopback operation.

As a minimum, the RTOS you use needs to provide the following services:

1) You need to be able to create at least three tasks (a Receive Task, a Transmit De-allocation Task, and a Timer Task)
2) Semaphore management (or the equivalent) and **µC/TCP-IP** need to be able to create at least 2 semaphores for each socket and an additional 4 semaphores for internal use.

3) Provide timer management services
4) If BSD socket `select()` is required, than the OS port must also include support for pending on multiple OS objects.

µC/TCP-IP is provided with a **µC/OS-II** interface. If you use a different RTOS, you can use the `net_os.*` for **µC/OS-II** as a template to interface to the RTOS of your choice.

1.02 Directories

The files shown in Figure 1-1 and 1-2 are placed in a directory structure such as to group common elements. The **µC/TCP-IP** distribution contains the following directories:

1.02.01 µC/TCP-IP Directories

→ **\Micrium\Software\uC-TCPIP-V2**
This is the main directory for **µC/TCP-IP**.

→ **\Micrium\Software\uC-TCPIP-V2\IF**
This directory contains interface specific files. Currently, **µC/TCP-IP** only supports two type of interfaces, Ethernet and Loopback. The Ethernet interface specific files are found in the following directories:

\Micrium\Software\uC-TCPIP-V2\IF`net_if.*`
`net_if.*` presents a programming interface between higher **µC/TCP-IP** layers and the link layer protocols. These files also provide interface management routines to the application.

\Micrium\Software\uC-TCPIP-V2\IF`net_if_ether.*`
`net_if_ether.*` contains the Ethernet interface specifics. This file should not need to be modified.

\Micrium\Software\uC-TCPIP-V2\IF`net_if_loopback.*`
`net_if_loopback.*` contains loopback interface specifics. This file should not need to be modified.

→ **\Micrium\Software\uC-TCPIP-V2\Dev**
This directory contains device drivers for different interfaces. Currently, **µC/TCP-IP** only supports one type of interface, Ethernet. We tested **µC/TCP-IP** with many types of Ethernet devices. For example, a SMC LAN91C115 and an Atmel AT91RM9200 CPU containing an on-chip Ethernet MAC. The Dev specific code is thus found in the following directories:

\Micrium\Software\uC-TCPIP-V2\Dev\Ether\LAN91C115net_dev_lan91c115.*

\Micrium\Software\uC-TCPIP-V2\Dev\Ether\AT91RM9200net_dev_at91rm9200.*

Other Ethernet controller drivers will be placed under the `Ether` sub-directory. Note that other device drivers must also be called `net_dev_*.*`.

Ethernet Phy support may be found in the directory below.

\Micrium\Software\uC-TCPIP-V2\Dev\Ether\Phy

Micrium provides a generic Ethernet Phy driver which will provide sufficient support for most (R)MII compliant Ethernet physical layer devices. Specific Phy driver may be authored in order to provide extended functionality such as link state interrupt support.

\Micrium\Software\uC-TCPIP-V2\Dev\Ether\Phy\Generic

µC/TCP-IP supports multiple instances or combinations of devices and associated drivers.

→ **\Micrium\Software\uC-TCPIP-V2\OS**
This directory contains the RTOS abstraction layer which allows you to use **µC/TCP-IP** with just about any commercial or in-house RTOS. You would place the abstraction layer for your own RTOS in a sub-directory under `OS` as follows:

\Micrium\Software\uC-TCPIP-V2\OS\rtos_namenet_os.*

Note that you must always name the files for your RTOS abstraction layer `net_os.*`.

µC/TCP-IP has been tested with **µC/OS-II** and the RTOS layer files for this RTOS are found in the following directory:

\Micrium\Software\uC-TCPIP-V2\OS\uCOS-II`net_os.*`

→ **\Micrium\Software\uC-TCPIP-V2\Source**
This directory contains all the CPU and RTOS independent files as shown in Figure 1-1 and 1-2. You should not have to change anything in this directory in order to use **µC/TCP-IP**.

1.02.02 Support Directories

µC/TCP-IP assumes the presence of a number of support files as described in this section.

→ **\Micrium\Software\CPU**
This directory is optional and may contain processor register definition files that are generally supplied by a CPU or compiler manufacturer. The names of these files are determined by the manufacturer and are generally referenced from the application file `includes.h`. Alternatively, they may be replaced by the application BSP directory.

→ **\Micrium\Software\uC-CPU**
This directory contains files that are used to adapt to different CPUs/compilers. Micrium provides these files for all internally supported architectures.

`cpu_def.h`
This file contains definitions used by the other CPU specific files. In fact, `cpu_def.h` should be independent of the actual CPU used.

Within the `uC-CPU` directory, we define processor specific files.

\Micrium\Software\uC-CPU\cpu_type\compiler

`cpu.h`
This file contains definitions of data word sizes. Specifically, you define here what C data types are associated with 8, 16 and 32 bit integers and, 32 and 64 bit floating point numbers. `cpu.h` also defines endianess, critical section macros `CPU_CRITICAL_ENTER()` and `CPU_CRITICAL_EXIT()`, and more.

`cpu_a.s`
This file contains functions to implement critical section protection for the specific CPU type used.

→ **`\Micrium\Software\uC-LIB`**
This directory contains library support files that are independent of the CPU, **µC/TCP-IP** and compiler. Micriµm does not use standard C library functions in order to simplify third-party certification.

`lib_def.h`
This file contains definitions that can be used by other applications. All `#defines` in this file are prefixed by `DEF_` and contains definitions for `DEF_TRUE` and `DEF_FALSE`, `DEF_YES` and `DEF_NO`, `DEF_ON` and `DEF_OFF` and bit masks.

`lib_mem.*`
These files contain functions to copy memory, clear memory, etc. All functions in this file start with `Mem_`. Library memory management functions are required for **µC/TCP-IP** and above.

1.02.03 Test Code Directories

→ **`\Micrium\Software\EvalBoards`**
Although not actually part of **µC/TCP-IP**, we created a standard directory structure where application examples are placed. Specifically, sample code is placed under an `'EvalBoards'` sub-directory. Each different evaluation board is categorized by its *manufacturer*, the actual *board name* and the *tools* that were used to test the sample code. Specifically, sample code is placed using the following structure:

`\Micrium\Software\EvalBoard\manufacturer\board\tools`

We tested **µC/TCP-IP** using the Cogent CSB637 (ARM9) processor using the IAR tool chain. Sample code is thus placed under the following directory:

`\Micrium\Software\EvalBoard\Cogent\CSB637\IAR\Project_Name`

Each different project / example has its own directory under the IAR directory. The example directories contain the following **µC/TCP-IP** related files:

```
app.c
app_cfg.h
includes.h
net_cfg.h
net_dev_cfg.c
net_dev_cfg.h
os_cfg.h
```

Of course, other source files, compiler build files, linker command files and so on are also found in the example directories or their respective BSP directories in order to create the specific example.

Code that is common to all of the examples and related to control of I/Os for a given evaluation board are placed in a BSP sub-directory as shown below. BSP stands for 'Board Support Package'

\Micrium\Software\EvalBoard\Cogent\CSB637\IAR\BSP

The BSP directory can contain the following files:

```
bsp.c
bsp.h
```

\Micrium\Software\EvalBoard\Cogent\CSB637\IAR\BSP\TCPIP-V2

```
net_bsp.c
net_bsp.h
```

You will notice the presence of a `net_bsp.c` files. This file is placed in the BSP directory because it is dependent of the CPU and network device used (interrupt structure, I/O ports, etc.) but, has a '`net_`' prefix indicating that it is a network specific file.

1.03 Block Diagram

Figure 1-2 shows a block diagram of the modules found in **µC/TCP-IP** and their relationship. Also included are the names of the files that are related to **µC/TCP-IP**.

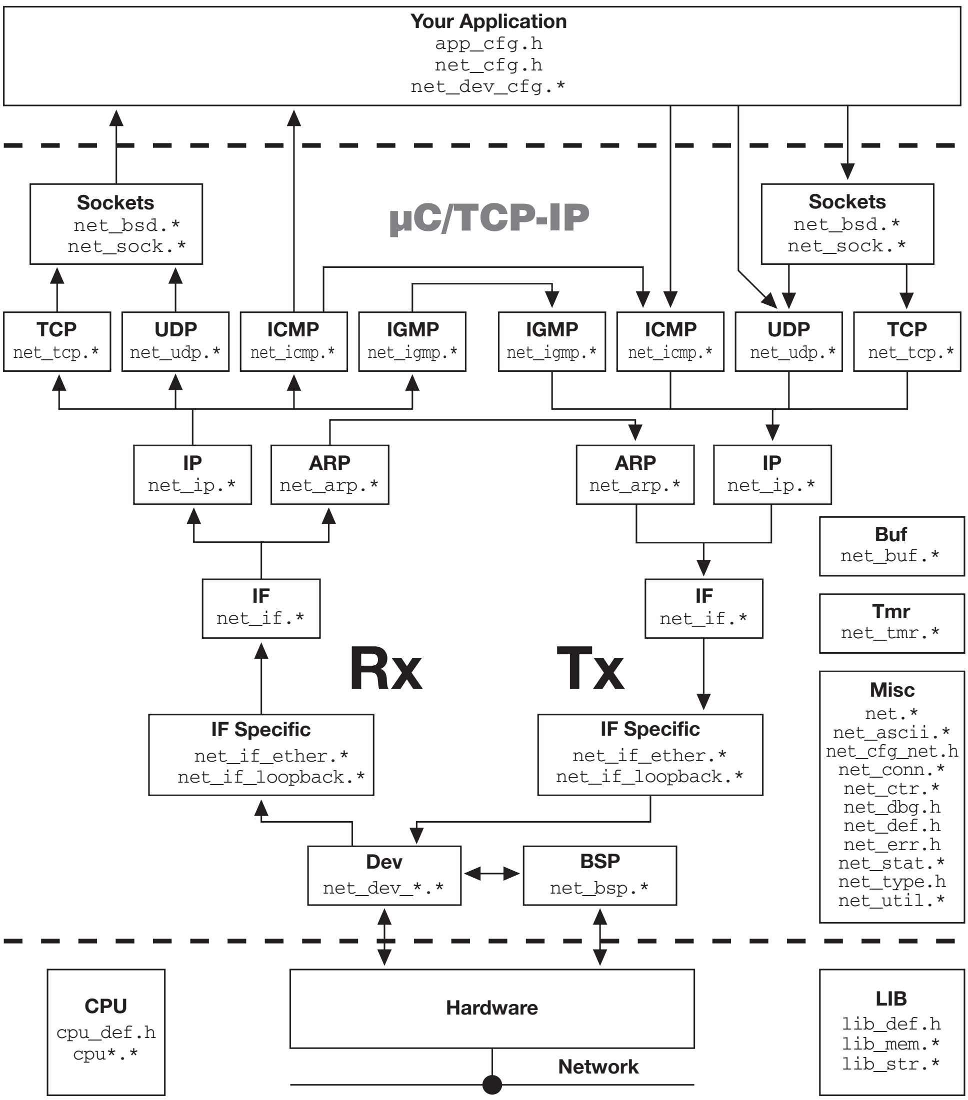

Figure 1-2, µC/TCP-IP Block Diagram

1.04 Task model

Your application interfaces to **µC/TCP-IP** via a well known API (Application Programming Interface) called BSD sockets (or **µC/TCP-IP**'s internal socket interface). Basically, you application can send and receive data to/from other hosts on the network via this interface.

The BSD sockets API interfaces to internal structures and variables (i.e. data) that are maintained by **µC/TCP-IP**. A binary semaphore (the global lock in Figure 1-3) is used to guard the access to this data to ensure exclusive access. In other words, to read or write to this data, a task needs to acquire the binary semaphore before it can access the data and release it when done. Of course, your application tasks do not have to know anything about this semaphore nor the data since its use is encapsulated by functions within **µC/TCP-IP**.

Figure 1-3 shows a simplified task model of **µC/TCP-IP** along with application tasks.

1.04.01 µC/TCP-IP Tasks and Priorities

µC/TCP-IP basically defines three internal tasks: a Receive Task, a Transmit De-allocation Task, and a Timer Task. The Receive Task is responsible for processing received packets from all devices. The Transmit De-allocation Task frees transmit buffer resources when they are no longer required. The Timer Task is responsible for handling all timeouts related to the TCP/IP protocols and network interface management.

When setting up task priorities, it is recommended that you set the priority number of tasks that will use **µC/TCP-IP**'s services below (higher priority) that of the of **µC/TCP-IP** internal tasks. This is to reduce starvation issues when an application task needs to send a lot of data. However, if your application task that uses **µC/TCP-IP** needs to have a higher priority then you should voluntarily relinquish the CPU on a regular basis. For example, you can suspend the task for a number of OS clock ticks.

We recommend configuring the Transmit De-allocation Task with the highest **µC/TCP-IP** internal task priority. The next highest priority task should be the Receive Task or the Loopback Task and lastly the Timer Task.

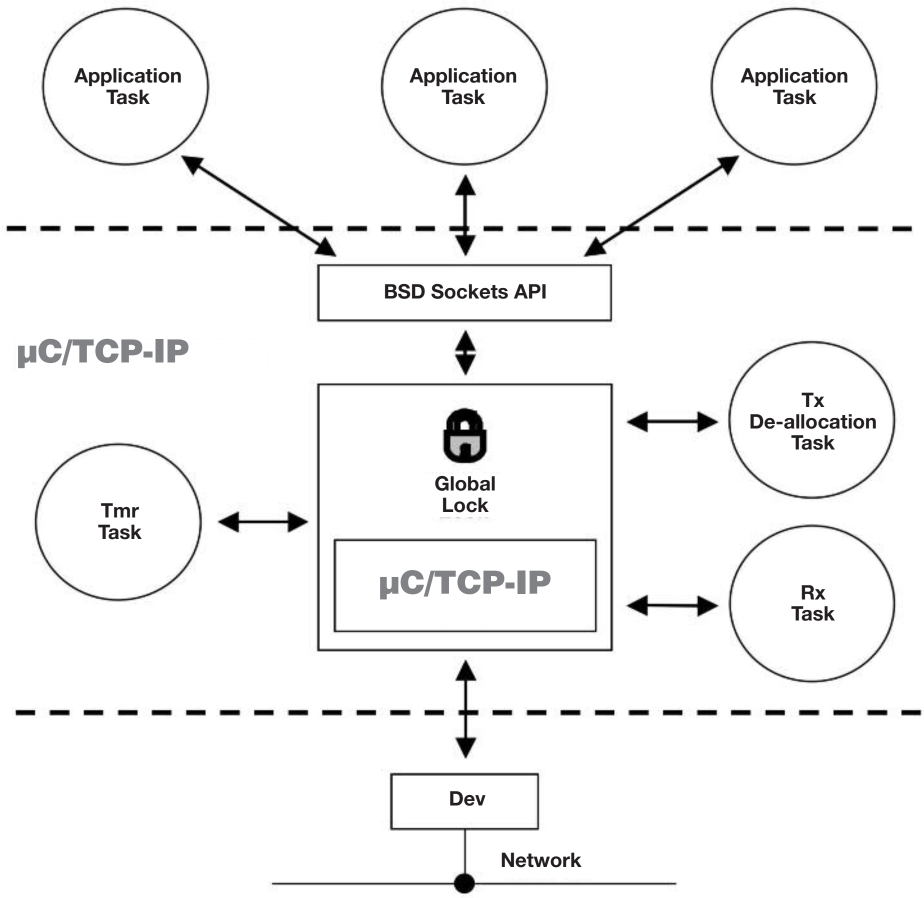

Figure 1-3, µC/TCP-IP Task Model

1.04.02 Receiving a Packet

Figure 1-4 shows a simplified task model of **µC/TCP-IP** when packets are received from the device. In this example, a TCP packet is being received.

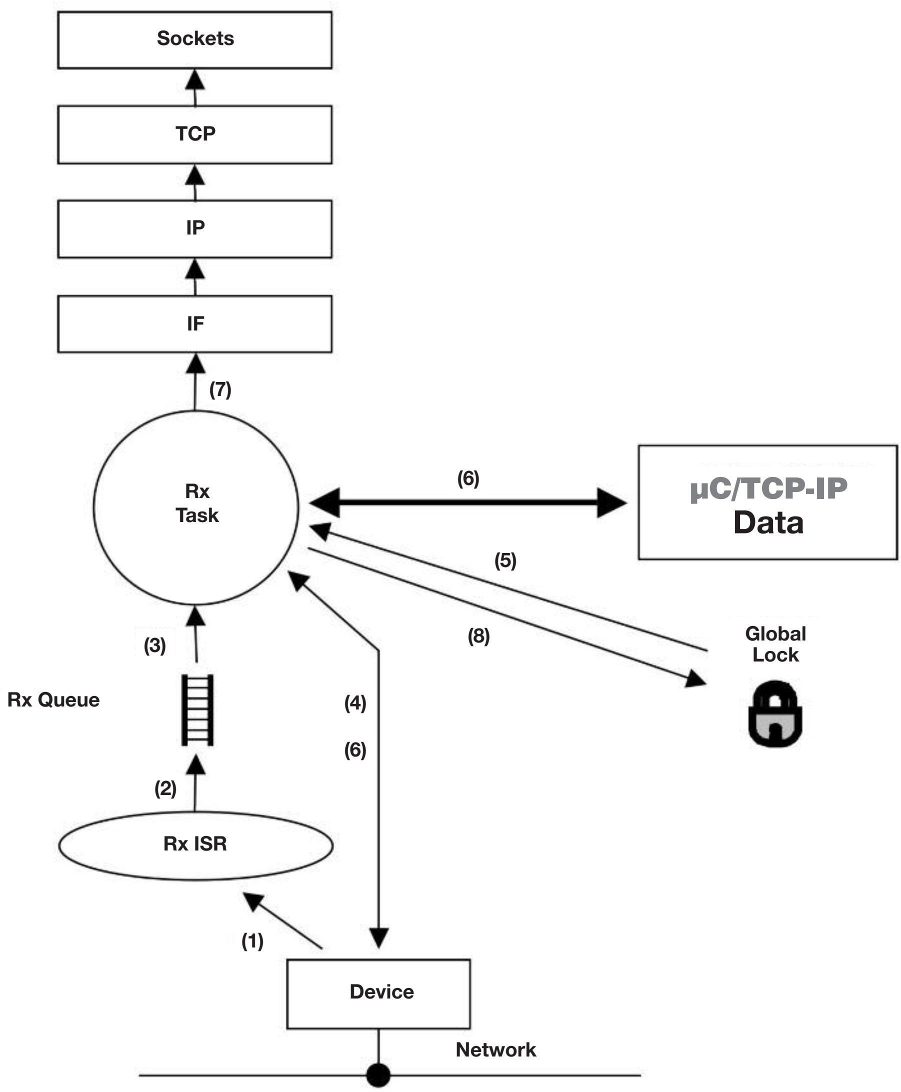

Figure 1-4, µC/TCP-IP Receiving a Packet

F1-4(1) A packet is sent on the network and the device recognizes its address as the destination for the packet. The device then generates an interrupt and the BSP global ISR handler is called for non vectored interrupt controllers. Either the global ISR handler or the vectored interrupt controller calls the Net BSP device specific ISR handler which in turn indirectly calls the device ISR handler using a predefined Net IF function call. The device ISR handler determines that the interrupt comes from a packet reception (as opposed to the completion of a transmission).

F1-4(2) Instead of processing the received packet directly from the ISR, we decided to pass the responsibility to a task. The Rx ISR thus simply ‘signals’ the Receive Task by posting the interface number to the Receive Task queue. Note that further Rx interrupts are generally disabled while processing the interrupt within the device ISR handler.

F1-4(3) The Receive Task does nothing until a signal is received from the *Rx ISR*.

F1-4(4) When a signal is received from an Ethernet device, the Receive Task wakes up and extracts the packet from the hardware and places it in a receive buffer. For DMA based devices, the receive descriptor buffer pointer is updated to point to a new data area and the pointer to the receive packet is passed to higher layers for processing.

µC/TCP-IP maintains three types of device buffers: small transmit, large transmit, and large receive. For a common Ethernet configuration, a small transmit buffer can typically hold up to 256 bytes of data, a large transmit buffer can typically hold up to 1500 bytes of data, and a large receive buffer typically holds 1518 bytes of data. Note that the large transmit buffer size is generally specified within the device configuration as 1594 or 1614 bytes. The additional space is used to hold additional protocol header data. These sizes as well as the quantity of these buffers are configurable for each interface during either compile time or run time.

F1-4(5) Buffers are shared resource and any access to those or any other **µC/TCP-IP** data structures is guarded by the binary semaphore guarding the data. This means that the Receive Task will need to acquire the semaphore before it can get a buffer from the pool.

F1-4(6) The Receive Task obtains a buffer from the buffer pool. The packet is removed from the device and placed in the buffer for further processing. For DMA, the acquired buffer pointer replaces the descriptor buffer pointer which received the current frame. The pointer to the received frame is passed to higher layers for further processing.

F1-4(7) The Receive Task examines received data via the appropriate link layer protocol and determines whether the packet is destined for the ARP or IP layer and passes the buffer to the appropriate layer for further processing. Note that the Receive Task brings the data all the way up to the application layer and thus the appropriate **µC/TCP-IP** functions operate within the context of the Receive Task.

F1-4(8) When the packet is processed, the lock is released and the Receive Task waits for the next packet to be received.

1.04.03 Transmitting a Packet

Figure 1-5 shows a simplified task model of **µC/TCP-IP** when packets are transmitted through the device. In this example, a TCP packet is being transmitted.

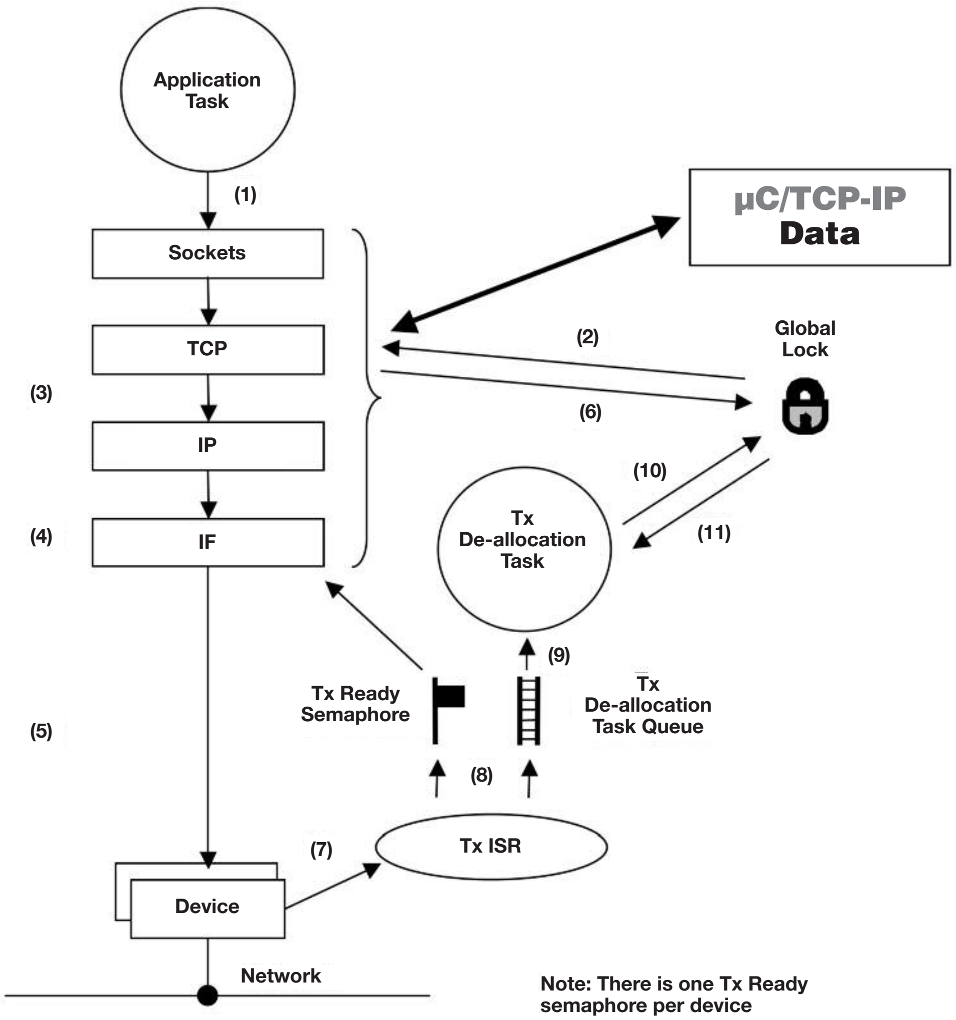

Figure 1-5, µC/TCP-IP Sending a Packet

F1-5(1) A task (assuming an application task) that wants to send data interfaces to **µC/TCP-IP** through the BSD socket API.

F1-5(2) A function within **µC/TCP-IP** acquires the binary semaphore (i.e. the global lock) in order to place the data to send into **µC/TCP-IP**'s data structures.

F1-5(3) The appropriate **µC/TCP-IP** layer processes the data, preparing if for transmission.

F1-5(4) The task (via the IF layer) then waits on a counting semaphore which is used to indicate that the transmitter in the device is available to send a packet. If the device is not able to send the packet, the task blocks until the semaphore is signaled by the device. Note that during device initialization, the semaphore is initialized with a value corresponding to the number of packets that can be sent at one time through the device. In other words, if the device has sufficient buffer space to be able to queue up four packets, then the counting semaphore is initialized with a count of 4. For DMA based devices, the value of the semaphore is initialized to the number of available transmit descriptors.

F1-5(5) When the device becomes ready, the driver either copies the data to the device internal memory space or configures the DMA transmit descriptor. When the device is fully configured, the device driver issues a transmit command.

F1-5(6) After placing the packet into the device, the task releases the global data lock and continues execution.

F1-5(7) When the device completes sending the data, the device generates an interrupt.

F1-5(8) The Tx ISR signals the Tx Available semaphore indicating that the device is able to send another packet. Additionally, the Tx ISR handler passes the address of the buffer that completed transmission to the Transmit De-allocation Task via a queue which is encapsulated by an OS port function call.

F1-5(9) The Transmit De-allocation Task wakes up when a device driver posts a transmit buffer address to its queue.

F1-5(10) The global data lock is acquired. If the global data lock is held by another task, the Transmit De-allocation Task must wait to acquire the global data lock. Since the Transmit De-allocation Task is recommended to be configured as the highest priority **µC/TCP-IP** task, it will run following the release of the global data lock assuming the queue has at least one entry present.

F1-5(11) The lock is released when transmit buffer de-allocation completes. Further transmission and reception of additional data by application and **µC/TCP-IP** tasks may resume.

Chapter

2

Getting Started with µC/TCP-IP

This chapter provides examples on how to use **µC/TCP-IP**. We decided to include this chapter early in the **µC/TCP-IP** manual so you could start using **µC/TCP-IP** as soon as possible. In fact, we assume you know little about **µC/TCP-IP** and networking. Concepts will be introduced as needed.

The example project was compiled using the IAR Embedded Workbench for the ARM processor. Tests were done using a Cogent CSB637 (ARM9 CPU) board which has an AT91RM9200 EMAC Ethernet Controller. We selected an ARM processor because of its popularity in the networking world.

2.01 Installing µC/TCP-IP

Distribution of **µC/TCP-IP** is performed through release files named `uC-TCPIP-Vxyy.zip`. The release archive files contain all the source code and documentation for the **µC/TCP-IP**. Additional support files such as those located within the **CPU** directory may or may not be required depending on your target hardware and development tools. Examples startup code, if available, may be delivered upon request. Example code is located in the **Evalboards** directory when applicable.

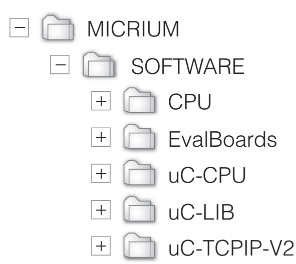

Figure 2-1, Directory tree for µC/TCP-IP

2.02 µC/TCP-IP Example Project

The following example project is used to show you the basic architecture of **µC/TCP-IP** and build an empty application. The application also uses **µC/OS-II** as the RTOS. Figure 2-2 shows the example project test setup. A Windows-based PC and the target system were connected to a 100 Mbps Ethernet router/switch. The PC's IP address is set to `10.10.10.111` and one of the target's addresses is configured to `10.10.10.64`.

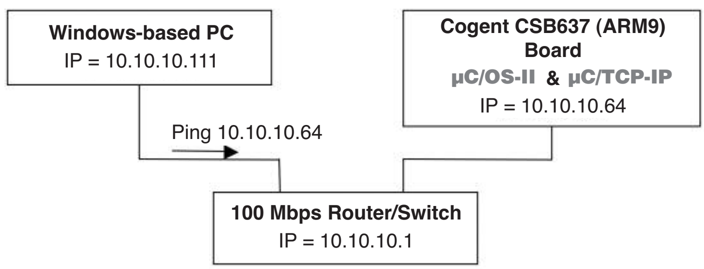

Figure 2-2, Test setup

This example contains enough code to be able to 'ping' the board. 'ping' is a utility found on most computer (Windows-based PCs, Linux, Unix, etc.) and allows you to see if a particular network enabled device is connected to your network. The IP address of the board is forced to `10.10.10.64` so, if you were to have a similar setup, you would be able to issue the following command from a command-prompt:

ping `10.10.10.64`

Ping (on the PC) should reply back with the ping time to the target. Most **µC/TCP-IP** target projects achieve ping times of less than 2 milliseconds.

Figure 2-3 shows the directory tree of the different components needed to build a **µC/TCP-IP** example project.

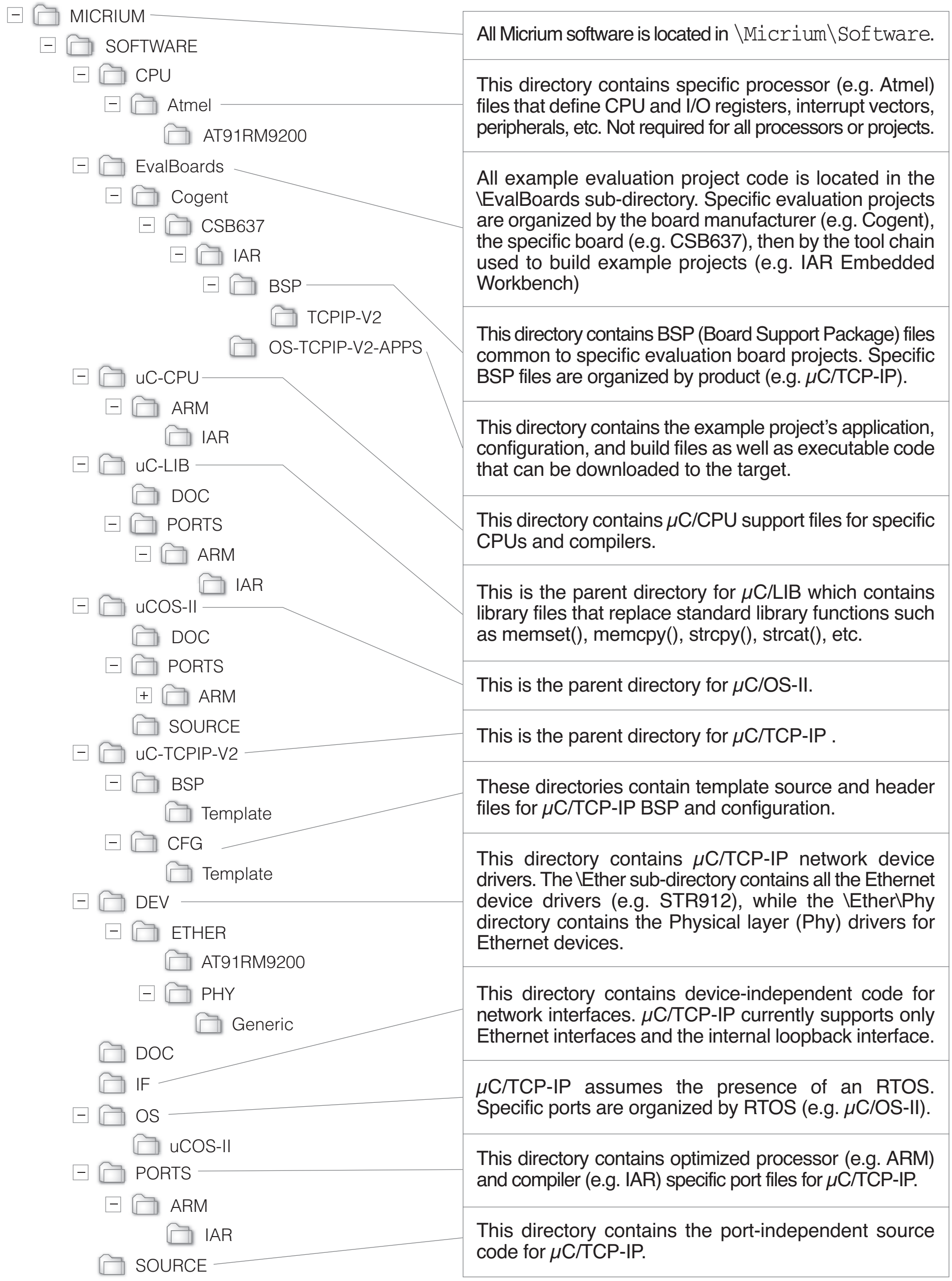

Figure 2-3, Directory tree for µC/TCP-IP based project

2.02.01 BSP

As described in Figure 2-3, the BSP (Board Support Package) directory contains common code that can be used in more than one example. Specifically, the BSP directories contain the following files:

```
bsp.c
bsp.h
net_bsp.c
net_bsp.h
```

BSP stands for Board Support Package and we place 'services' that the board provides in such a file. In our case, `bsp.c` contains I/O, timer initialization code, LED control code, and more. The I/Os that we use on the board are initialized when `BSP_Init()` is called.

The concept of a BSP is to hide the hardware details from the application code. It is important that functions in a BSP reflect the function and do not make references to any CPU specifics. For example, the code to turn on an LED is called `LED_On()` and not `csb337_led()`. If you use `LED_On()` in your code, you can easily port your code to another processor (or board) simply by rewriting `LED_On()` to control the LEDs on a different board. The same is true for other services. You will also notice that BSP functions are prefixed with the function's group. LED services start with `LED_`, timer services start with `Tmr_`, etc. In other words, BSP functions do not need to be prefixed by `BSP_`.

The `net_bsp.*` files contain hardware-dependent code specific to the network device(s) and other **µC/TCP-IP** functions. Specifically, these files may contain code to read data from and write data to network devices, handle hardware-level device interrupts, provide delay functions, get time stamps, etc.

2.02.02 Example OS-TCPIP-V2-APPS

The example directory contains the project's application and configuration code:

```
OS-TCPIP-V2-APPS.*
CSB_ARM_ROM_LNK.xcl
CSB_ARM_RAM_LNK.xcl
includes.h
app_cfg.h
net_cfg.h
net_dev_cfg.c
net_dev_cfg.h
os_cfg.h
app.c
```

The `OS-TCPIP-V2-APPS.*` files are IAR Embedded Workbench project files. Please consult IAR documentation for further information how to configure and build IAR projects.

The `*.xcl`, OR `*.icf` files depending on which version of EWARM you are using, are IAR Embedded Workbench linker command files that specify where code and data is located in memory. For example applications, code is located in RAM to ease debugging. When an application is ready to be deployed, a `csb63x_lnk_flash.xcl` linker file could be used to locate code in FLASH instead of RAM. Please consult IAR documentation for further information how to configure and use IAR linker command files.

The file `includes.h` is the application-specific master include header file. Almost all Micrium products require this file.

The file `net_cfg.h` is configuration file used to configure **µC/TCP-IP** parameters like the number of network timers, sockets, and connections created; default timeout values, and more. You MUST include `net_cfg.h` in your application as **µC/TCP-IP** requires this file. See Chapter 4 for more information.

The files `net_dev_cfg.c` and `net_dev_cfg.h` are configuration files used to configure **µC/TCP-IP** interface parameters such as the number of transmit and receive buffers and so forth. See the chapter on interface configuration for more details.

The file `os_cfg.h` is a configuration file used to configure **µC/OS-II** parameters like the maximum number of tasks, events, and objects; which **µC/OS-II** services are enabled (semaphores, mailboxes, queues); and more. You MUST include `os_cfg.h` in any **µC/OS-II** application since the OS requires this file. See **µC/OS-II** documentation and books for further information.

Lastly, the file `app.c` contains the application code for the CSB637 example project. As with most C programs, code execution start at `main()` which is shown in Listing 2-1. The application code starts **µC/TCP-IP** and a generic set of TCP-IP application services. These services (like web servers and FTP clients) are sold separately from **µC/TCP-IP**.

Listing 2-1

```
int main (void)
{
#if (OS_TASK_NAME_SIZE >= 6)
    INT8U  os_err;
#endif

    BSP_Init();                                  (1)
    CPU_Init();                                  (2)

    OSInit();                                    (4)

                                                 (5)
    os_err = OSTaskCreateExt(
        (void (*)(void *)) App_TaskStart,
        (void          * ) 0,
        (OS_STK        * )&AppStartTaskStk[APP_OS_CFG_START_TASK_STK_SIZE-1],
        (INT8U           ) APP_OS_CFG_START_TASK_PRIO,
        (INT16U          ) APP_OS_CFG_START_TASK_PRIO,
        (OS_STK        * )&AppStartTaskStk[0],
        (INT32U          ) APP_OS_CFG_START_TASK_STK_SIZE,
        (void          * ) 0,
        (INT16U          )(OS_TASK_OPT_STK_CLR | OS_TASK_OPT_STK_CHK));

                                                 (6)
#if (OS_TASK_NAME_SIZE >= 6)
    OSTaskNameSet(APP_OS_CFG_START_TASK_PRIO, "Start", &os_err);
#endif

    OSStart();                                   (7)
}
```

L2-1(1) We start by initializing all BSP (Board Support Package) I/O's on this board, as well as the **μC/OS-II** tick interrupt and the AT91RM9200's interrupt controller.

L2-1(2) Initialize **μC/CPU** module for CSB637 AT91RM9200 processor (configure CPU name).

L2-1(3) The example code assumes **µC/OS-II** is available and calls `OSInit()` to initialize the OS.

L2-1(4) **µC/OS-II** requires that we create at least one application task. This is done by calling `OSTaskCreateExt()` and specifying the task start address, the top-of-stack to use for this task, the priority of the task and a few other arguments.

L2-1(5) **µC/OS-II** allows you to assign names to tasks that have been created. We thus assign a name to the application task. These names are used mostly during debug. In fact, task names are displayed during debug when using IAR's C-Spy debugger (or other **µC/OS-II** aware debuggers).

L2-1(6) In order to start **µC/OS-II** multitasking, your application must call OSStart(), which schedules the highest-priority task to run. In this case, **µC/OS-II** will schedule and start `AppTaskStart()`.

The only example project application task created, `AppTaskStart()`, is shown in Listing 2-2:

Listing 2-2

```
static  void  AppTaskStart (void  *p_arg)
{
    NET_IF_NBR        if_nbr;
    INT8U             dly_sec;
    CPU_CHAR         *paddr;
    CPU_CHAR         *paddr_mask;
    CPU_CHAR         *paddr_dflt_gateway;
    NET_IP_ADDR       addr_ip;
    NET_IP_ADDR       addr_mask;
    NET_IP_ADDR       addr_dflt_gateway;
    NET_ERR           err;

    (void)&p_arg;

    Mem_Init();                                   (1)

    FS_Init();                                    (2)
```

```
err = Net_Init();                        (3)
                                                             (4)
if_nbr = NetIF_Add((void    *)&NetIF_API_Ether,               (a)
                   (void    *)&NetDev_API_AT91RM9200,         (b)
                   (void    *)&NetDev_Cfg_AT91RM9200_0,       (c)
                   (void    *)&NetPhy_API_Generic,            (d)
                   (void    *)&NetPhy_Cfg_Generic_0,          (e)
                   (NET_ERR *)&err);

APP_TEST_FAULT(err, NET_IF_ERR_NONE);
APP_TRACE_INFO(("  IF #%2d   added\n\r",   if_nbr));

NetIF_Start(if_nbr, &err);               (5)
APP_TEST_FAULT(err, NET_IF_ERR_NONE);
APP_TRACE_INFO(("  IF #%2d   started\n\r", if_nbr));

dly_sec = 4;                             (6)
APP_TRACE_INFO(("  IF #%2d   PHY delay %d seconds\n\r", if_nbr, dly_sec));
OSTimeDlyHMSM(0, 0, dly_sec, 0);

                                         (7)
paddr              = (CPU_CHAR *)"10.10.10.64";
paddr_mask         = (CPU_CHAR *)"255.255.255.0";
paddr_dflt_gateway = (CPU_CHAR *)"10.10.10.1";

addr_ip            = NetASCII_Str_to_IP((CPU_CHAR *) paddr,
                                        (NET_ERR  *)&err);

addr_mask          = NetASCII_Str_to_IP((CPU_CHAR *) paddr_mask,
                                        (NET_ERR  *)&err);

addr_dflt_gateway  = NetASCII_Str_to_IP((CPU_CHAR *) paddr_dflt_gateway,
                                        (NET_ERR  *)&err);

NetIP_CfgAddrAdd(if_nbr, addr_ip, addr_mask, addr_dflt_gateway, &err);
APP_TEST_FAULT(err, NET_IP_ERR_NONE);
```

```
                                                    (8)
    DNSc_Init();
    FTPs_Init();
    HTTPs_Init();
    TELNETs_Init();



    (void)OSTaskDel(OS_PRIO_SELF);          (9)
}
```

L2-2(1) Initialize **µC/LIB** memory manager used by **µC/TCP-IP**. Memory management is used to manage pools of memory as well as allocate memory from a global heap.

L2-2(2) Initialize (optional) File System used by some **µC/TCP-IP** services—**µC/FTPs**, **µC/TFTPs**, **µC/HTTPs**, etc.

L2-2(3) Initialize **µC/TCP-IP** by calling `NetInit()` which initializes all network modules, data structures, and creates three network tasks to handle network interfaces & devices and one task to manage network timers.

IMPORTANT

`Net_Init()` returns an error code indicating whether **µC/TCP-IP** successfully initialized. If `Net_Init()` returns `NET_ERR_NONE` then **µC/TCP-IP** was successfully initialized.

If `Net_Init()` does return any other error code, the application MUST NOT call `Net_Init()` a second time. Instead, you should examine the error code to determine the source of the error. Since `Net_Init()` aborts initialization as soon as it finds the first initialization error, you can search for the returned numerical error code in `net_err.h` to find the corresponding `NET_ERR_` error code. Then you can search the **µC/TCP-IP** source code for the instance(s) where the `NET_ERR_` error code is returned. This should help determine why **µC/TCP-IP** initialization failed.

NOTE: If your application does not examine the return error code, **µC/TCP-IP** initialization may have partially or completely failed and the results are unpredictable.

L2-2(4) `NetIF_Add()` adds **µC/TCP-IP** network interface(s)/device(s). The desired network interfaces and network devices are added by passing pointers to the appropriate APIs and configuration structures. For the example project, the CSB637's AT91RM9200 Ethernet controller is added as a network interface by passing pointers to the Ethernet API (4a), AT91RM9200 device driver API (4b) and configuration (4c), and the Ethernet Physical Layer API (4d) and configuration (4e). Interface and device API structures are defined in their respective network interface & device drivers (e.g. `NetIF_API_Ether` is defined in **\uC-TCPIP-V2\IF**`net_if_ether.c`, `NetDev_API_AT91RM9200` is defined in **\uC-TCPIP-V2\Dev\Ether**`net_dev_at91rm9200.c`, and `NetPhy_API_Generic` is defined in **\uC-TCPIP-V2\Dev\Ether\Phy\Generic**`net_phy.c`). You must define all interface and device configuration structures in your application's `net_dev_cfg.c`, but you may use the templates provided in **\uC-TCPIP-V2\Cfg\Templates**`net_dev_cfg.c`.

`NetIF_Add()` returns either a valid network interface number, if the network interface was successfully added, or `NET_IF_NBR_NONE` otherwise. The return network interface number should be saved—at least temporarily—in order to perform further operations like starting or stopping an interface, adding network addresses to an interface, configuring network interface parameters, etc.

Additional network interfaces may be added on boards with more than one network device. The maximum number of network interfaces that can be added at run-time by the application is configured by `NET_IF_CFG_MAX_NBR_IF`.

Note that the first network interface added and started will be the default interface used for all default communication.

L2-2(5) After adding network interface(s)/device(s), `NetIF_Start()` is called to enable the interfaces for receive and transmit. Network interfaces/devices may be started **before OR after** network addresses are added but can typically only communicate via the interface's link layer protocol while no network addresses are configured on the interface. (See L2-2(7) below for more information.)

L2-2(6) Many Ethernet devices and/or physical layers require several seconds after initialization before they can communicate on the network.

L2-2(7) Each 'host' on a TCP/IP network requires a unique network address to communicate on the network. On IP version 4 networks (IPv4), this is called the *IP address* and consists of a 32-bit value (IPv6 addresses are 128-bit). An IPv4 address is generally represented in Dotted Decimal Notation as four decimal numbers separated by a dot. Each of the four decimal numbers correspond to an 8-bit value in the 32-bit address, where the first 8-bit number is the most-significant octet (byte) of the address and the last 8-bit number is the least-significant octet. For example, `192.168.1.64` corresponds to an IP address of 0xC0A80140.

IP addresses are added to a network interface at run-time either statically or dynamically. Applications may configure IP addresses statically by calling `NetIP_CfgAddrAdd()` and passing the desired IP address, IP address subnet mask, & gateway's IP address (most Ethernet networks have routers for the gateway). The maximum number of statically-configured IP addresses that can be added at run-time to a network interface is configured by `NET_IP_CFG_IF_MAX_NBR_ADDR`.

Applications may also configure each network interface with a single dynamic address. For IPv4 Ethernet networks, this is typically performed through the use of a service called DHCP (Dynamic Host Configuration Protocol). In order to dynamically configure addresses via DHCP, there must be a DHCP server or service available on the interface's Ethernet network. If there is no DHCP server available, then the interface's address(s) MUST BE configured statically.

The optional **µC/DHCPc** module provides applications with the capability to dynamically configure IP addresses on Ethernet interfaces. (See **µC/DHCPc** documentation for further information.)

L2-2(8) Initialize optional clients and server modules (e.g. **µC/DNSc**, **µC/FTPs**, **µC/FTPc**, **µC/HTTPs**, **µC/POP3c**, **µC/SMTPc**, etc.). Please consult their respective documentation for further information.

µC/DNSc client module: DNS (Domain Name Service) protocol resolves fully qualified domain names (FQDN) to IP addresses. For example, the domain name www.micrium.com currently resolves to `64.15.155.21`. This enables your application to use domain names instead of the IP addresses, which are less descriptive and subject to change over time.

µC/FTPc client module: FTP (File Transfer Protocol) client allows the target to connect and transfer files to or from FTP servers. Target files may be stored directly in RAM or to other storage media via a file system (e.g. **µC/FS**).

µC/FTPs server module: Allows FTP clients to connect and transfer files to or from the target.

µC/HTTPs server module: HTTP (Hypertext Transfer Protocol) allows web browsers to connect to the target to download HTML web pages and other files.

µC/POP3c client module: POP3 (Post Office Protocol 3) enables the reception of email messages from a mail server. The email messages are stored in a user's mailbox in the server and *pulled* by the user. If you have to receive email messages on your target from other internet hosts, you need **µC/POP3c**. You may also need **µC/DNSc** since POP3 servers are usually addressed by their DNS names.

µC/SMTPc client module: SMTP (Simple Mail Transfer Protocol) enables the transfer of email messages over internets. SMTP is a push protocol. The sender of the message *pushes* the message to the recipient's mailbox. If you have to send email message from your target to other internet hosts, you need **µC/SMTPc**. You may also need **µC/DNSc** since SMTP servers are usually addressed by their DNS names.

µC/SNTPc client module: SNTP (Simple Network Time Protocol) synchronizes a target's real time clock over the Internet. All time information is in number of milliseconds since January 1, 1900 at midnight Greenwich Mean Time (GMT). **µC/SNTPc** transfers the current time over the Internet but may require the **µC/CLK** module to store and update the time.

L2-2(9) After all tasks have been started, `OSTaskDel()` deletes the start application task. However, it is not absolutely necessary to delete the task if it will be used for other purposes.

Once the source code is built and loaded into the target, the target will respond to ICMP Echo (ping) requests.

Chapter

3

μC/TCP-IP Configuration

μC/TCP-IP is configurable at compile time via approximately 60 `#defines` in an application's `net_cfg.h` and `app_cfg.h` files. **μC/TCP-IP** uses `#defines` because they allow code and data sizes to be scaled at compile time based on enabled features and the configured number of network objects. In other words, this allows the ROM and RAM footprints of **μC/TCP-IP** to be adjusted based on your requirements.

Most of the `#defines` should be configured with the default configuration values. Another small handful of values may likely never change because there is currently only one configuration choice available. This leaves about a dozen or so values that should be configured with values that may deviate from the default configuration.

It is recommended that you start your configuration process with the recommended or default configuration values which are shown in **`bold`**.

3.01 Network Configuration

3.01.01 Network Configuration, NET_CFG_INIT_CFG_VALS

This configuration constant is used to determine whether internal TCP/IP parameters are set to default values or are set by the user. `NET_CFG_INIT_CFG_VALS` can take one of two values:

`NET_INIT_CFG_VALS_DFLT`
Configure **µC/TCP-IP**'s network parameters with default values. The application need only call `Net_Init()` to initialize both **µC/TCP-IP** and its configurable parameters. This configuration is highly recommended since configuring network parameters requires in-depth knowledge of the protocol stack. In fact, most books recommend many of the default values we have selected.

Parameter	Units	Min.	Max.	Default	Configuration Function
Interface's Network Buffer Low Threshold	% of the Total Number of an Interface's Network Buffers	5%	50%	5%	`NetDbg_CfgRsrcBufThLo()`
Interface's Network Buffer Low Threshold Hysteresis	% of the Total Number of an Interface's Network Buffers	0%	15%	3%	`NetDbg_CfgRsrcBufThLo()`
Interface's Large Receive Buffer Low Threshold	% of the Total Number of an Interface's Large Receive Buffers	5%	50%	5%	`NetDbg_CfgRsrcBufRxLargeThLo()`
Interface's Large Receive Buffer Low Threshold Hysteresis	% of the Total Number of an Interface's Large Receive Buffers	0%	15%	3%	`NetDbg_CfgRsrcBufRxLargeThLo()`
Interface's Small Transmit Buffer Low Threshold	% of the Total Number of an Interface's Small Transmit Buffers	5%	50%	5%	`NetDbg_CfgRsrcBufTxSmallThLo()`
Interface's Small Transmit Buffer Low Threshold Hysteresis	% of the Total Number of an Interface's Small Transmit Buffers	0%	15%	3%	`NetDbg_CfgRsrcBufTxSmallThLo()`
Interface's Large Transmit Buffer Low Threshold	% of the Total Number of an Interface's Large Transmit Buffers	5%	50%	5%	`NetDbg_CfgRsrcBufTxLargeThLo()`
Interface's Large Transmit Buffer Low Threshold Hysteresis	% of the Total Number of an Interface's Large Transmit Buffers	0%	15%	3%	`NetDbg_CfgRsrcBufTxLargeThLo()`
Network Timer Low Threshold	% of the Total Number of Network Timers	5%	50%	5%	`NetDbg_CfgRsrcTmrLoTh()`
Network Timer Low Threshold Hysteresis	% of the Total Number of Network Timers	0%	15%	3%	`NetDbg_CfgRsrcTmrLoTh()`
Network Connection Low Threshold	% of the Total Number of Network Connections	5%	50%	5%	`NetDbg_CfgRsrcConnLoTh()`
Network Connection Low Threshold Hysteresis	% of the Total Number of Network Connections	0%	15%	3%	`NetDbg_CfgRsrcConnLoTh()`
ARP Cache Low Threshold	% of the Total Number of ARP Caches	5%	50%	5%	`NetDbg_CfgRsrcARP_CacheLoTh()`

ARP Cache Low Threshold Hysteresis	% of the Total Number of ARP Caches	0%	15%	3%	NetDbg_CfgRsrcARP_CacheLoTh()
TCP Connection Low Threshold	% of the Total Number of TCP Connections	5%	50%	5%	NetDbg_CfgRsrcTCP_ConnLoTh()
TCP Connection Low Threshold Hysteresis	% of the Total Number of TCP Connections	0%	15%	3%	NetDbg_CfgRsrcTCP_ConnLoTh()
Socket Low Threshold	% of the Total Number of Sockets	5%	50%	5%	NetDbg_CfgRsrcSockLoTh()
Socket Low Threshold Hysteresis	% of the Total Number of Sockets	0%	15%	3%	NetDbg_CfgRsrcSockLoTh()
Resource Monitor Task Time	Seconds	1	600	60	NetDbg_CfgMonTaskTime()
Network Connection Accessed Threshold	Number of Network Connections	10	65000	100	NetConn_CfgAccessTh()
Network Interface Physical Link Monitor Period	Milliseconds	50	60000	250	NetIF_CfgPhyLinkPeriod()
Network Interface Performance Monitor Period	Milliseconds	50	60000	250	NetIF_CfgPerfMonPeriod()
ARP Cache Timeout	Seconds	60	600	600	NetARP_CfgCacheTimeout()
ARP Cache Accessed Threshold	Number of ARP Caches	100	65000	100	NetARP_CfgCacheAccessedTh()
ARP Request Timeout	Seconds	1	10	5	NetARP_CfgReqTimeout()
ARP Request Maximum Number of Retries	Maximum Number of Transmitted ARP Request Retries	0	5	3	NetARP_CfgReqMaxRetries()
IP Receive Fragments Reassembly Timeout	Seconds	1	15	5	NetIP_CfgFragReasmTimeout()
ICMP Transmit Source Quench Threshold	Number of Transmitted ICMP Source Quenches	1	100	5	NetICMP_CfgTxSrcQuenchTh()

Table 3-1, *µC/TCP-IP* Internal Configuration Parameters

NET_INIT_CFG_VALS_APP_INIT

It is possible to change the parameters listed in Table 3-1 by calling the above configuration functions that will do the work for you based on the parameter values you desire. These values could be stored in non-volatile memory and recalled at power-up (e.g. using EEPROM or battery-backed RAM) by your application. Or these values could be hard-coded directly in your application. Regardless of how the application configures these values, if you select this option, your application will need to initialize ALL of the above configuration parameters using the configuration functions listed above.

Alternatively, your application could call `Net_InitDflt()` to initialize all the internal configuration parameters to their default values and then your application could call the configuration functions for only the values you desire.

3.01.02 Network Configuration, NET_CFG_OPTIMIZE

Select portions of **µC/TCP-IP** code may be optimized for better performance or for smallest code size by configuring `NET_CFG_OPTIMIZE`:

`NET_OPTIMIZE_SPD` . . . . Optimizes **µC/TCP-IP** for best speed performance

`NET_OPTIMIZE_SIZE` . . . . Optimizes **µC/TCP-IP** for best binary image size

3.01.03 Network Configuration, NET_CFG_OPTIMIZE_ASM_EN

Select portions of **µC/TCP-IP** code may even call optimized assembly functions by configuring `NET_CFG_OPTIMIZE_ASM_EN`:

`DEF_DISABLED` . . . . No optimized assembly files/functions are included in the **µC/TCP-IP** build

or

`DEF_ENABLED` . . . . Optimized assembly files/functions are included in the **µC/TCP-IP** build

3.02 Debug Configuration

A fair amount of code in **µC/TCP-IP** has been included to simplify debugging. There are several configuration constants used to aid debugging.

3.02.01 Debug Configuration, NET_DBG_CFG_INFO_EN

`NET_DBG_CFG_INFO_EN` is used to enable/disable **µC/TCP-IP** debug information:

- Internal constants assigned to global variables
- Internal variable data sizes calculated & assigned to global variables

The value can either be:

DEF_DISABLED
or
DEF_ENABLED

3.02.02 Debug Configuration, NET_DBG_CFG_STATUS_EN

NET_DBG_CFG_STATUS_EN is used to enable/disable **µC/TCP-IP** debug status information:

Run-time debug functions that provide information on :

- Internal resource usage – low or lost resources
- Internal faults or errors

The value can either be:

DEF_DISABLED
or
DEF_ENABLED

3.02.03 Debug Configuration, NET_DBG_CFG_MEM_CLR_EN

NET_DBG_CFG_MEM_CLR_EN is used to clear internal network data structures when allocated or de-allocated. By clearing we mean setting all bytes in internal data structures to '0' or to default initialization values. You can set this configuration constant to either:

DEF_DISABLED
or
DEF_ENABLED

You would typically set it to DEF_DISABLED unless you are debugging and you want to examine the contents of the buffer. Having the buffer cleared generally helps you differentiate between 'proper' data and 'pollution'.

3.02.04 Debug Configuration, NET_DBG_CFG_TEST_EN

NET_DBG_CFG_TEST_EN is used internally for testing/debugging purposes and can be set to either:

DEF_DISABLED
or
DEF_ENABLED

3.03 Argument Checking Configuration

Most functions in **µC/TCP-IP** include code to validate arguments that are passed to it. Specifically, **µC/TCP-IP** checks to see if passed pointers are NULL, if arguments are within valid ranges, etc. The following constants configure additional argument checking.

3.03.01 Argument Checking Configuration, NET_ERR_CFG_ARG_CHK_EXT_EN

NET_ERR_CFG_ARG_CHK_EXT_EN allows code to be generated to check arguments for functions that can be called by the user and, for functions which are internal but receives arguments from an API that the user can call. Also, enabling this check verifies whether **µC/TCP-IP** is initialized before API tasks and functions perform the desired function.

You can set this configuration constant to either:

DEF_DISABLED
or
DEF_ENABLED

3.03.02 Argument Checking Configuration, NET_ERR_CFG_ARG_CHK_DBG_EN

NET_ERR_CFG_ARG_CHK_DBG_EN allows code to be generated which checks to make sure that pointers passed to functions are not NULL, that arguments are within range, etc. You can set this configuration constant to either:

DEF_DISABLED
or
DEF_ENABLED

3.04 Network Counter Configuration

µC/TCP-IP contains code that increments counters to keep track of statistics such as the number of packets received, the number of packets transmitted, etc. Also, **µC/TCP-IP** contains counters that are incremented when error conditions are detected. The following constants enable or disable network counters.

3.04.01 Network Counter Configuration, NET_CTR_CFG_STAT_EN

NET_CTR_CFG_STAT_EN determines whether the code and data space used to keep track of statistics will be included. You can set this configuration constant to either:

DEF_DISABLED
or
DEF_ENABLED

3.04.02 Network Counter Configuration, NET_CTR_CFG_ERR_EN

NET_CTR_CFG_ERR_EN determines whether the code and data space used to keep track of errors will be included. You can set this configuration constant to either:

DEF_DISABLED
or
DEF_ENABLED

3.05 Network Timer Configuration

µC/TCP-IP manages software timers used to keep track of timeouts and execute callback functions when needed.

3.05.01 Network Timer Configuration, NET_TMR_CFG_NBR_TMR

`NET_TMR_CFG_NBR_TMR` determines the number of timers that **µC/TCP-IP** will be managing. Of course, the number of timers affect the amount of RAM required by **µC/TCP-IP**. Each timer requires 12 bytes plus 4 pointers. Timers are required for:

- The Network Debug Monitor Task . . . . 1 total
- The Network Performance Monitor . . . . 1 total
- The Network Link State Handler . . . . 1 total
- Each ARP cache entry . . . . 1 per ARP cache
- Each IP fragment reassembly . . . . 1 per IP fragment chain
- Each TCP connection . . . . 7 per TCP connection

We recommend starting with at least `12`, but a better starting point may be to allocate the maximum number of timers for all resources. For instance, if the Network Debug Monitor Task is enabled (see Section 10.02), 20 ARP caches are configured (`NET_ARP_CFG_NBR_CACHE = 20`), & 10 TCP connections are configured (`NET_TCP_CFG_NBR_CONN = 10`); the maximum number of timers for these resources is 1 for the Network Debug Monitor Task, 1 for the Network Performance Monitor, `1` for the Link State Handler, `(20 * 1)` for the ARP caches and, `(10 * 7)` for the TCP connections:

`# Timers =  1 + 1 + 1 +  (20 * 1)  +  (10 * 7)  =  93`

3.05.02 Network Timer Configuration, NET_TMR_CFG_TASK_FREQ

`NET_TMR_CFG_TASK_FREQ` determines how often (in Hz) the network timers are to be updated. This value MUST NOT be configured as a floating-point number. You would typically set this value to **`10 Hz`**.

3.06 Network Buffer Configuration

µC/TCP-IP manages Network Buffers to read data to and from network applications and network devices. Network Buffers are specially configured with network devices as described in the *µC/TCP-IP Driver Architecture* document.

3.07 Network Interface Layer Configuration

3.07.01 Network Interface Layer Configuration, NET_IF_CFG_MAX_NBR_IF

`NET_IF_CFG_MAX_NBR_IF` determines the maximum number of network interfaces that **µC/TCP-IP** may create at run-time. The default value of **`1`** is for a single network interface.

3.07.02 Network Interface Layer Configuration, NET_IF_CFG_ADDR_FLTR_EN

`NET_IF_CFG_ADDR_FLTR_EN` determines whether address filtering is enabled or not. This configuration value can either be set to:

`DEF_DISABLED` . . . . Addresses are NOT filtered
or
`DEF_ENABLED` . . . . Addresses are filtered

3.07.03 Network Interface Layer Configuration, NET_IF_CFG_TX_SUSPEND_TIMEOUT_MS

`NET_IF_CFG_TX_SUSPEND_TIMEOUT_MS` configures the network interface transmit suspend timeout value. The value is specified in integer milliseconds. We recommend starting with a value of **`1 millisecond`**.

3.07.04.01 Network Interface Layer Configuration, NET_IF_CFG_LOOPBACK_EN

NET_IF_CFG_LOOPBACK_EN determines whether the code and data space used to support the loopback interface for internal-only communication only will be included. You can set this configuration constant to either:

DEF_DISABLED
or
DEF_ENABLED

3.07.04.02 Network Interface Layer Configuration, NET_IF_CFG_ETHER_EN

NET_IF_CFG_ETHER_EN determines whether the code and data space used to support Ethernet interfaces and devices will be included. You can set this configuration constant to either:

DEF_DISABLED
or
DEF_ENABLED

This parameter should be enabled if your target expects to communicate over Ethernet networks.

3.08 ARP (Address Resolution Protocol) Configuration

ARP is only required for some network interfaces like Ethernet.

3.08.01 ARP Configuration, NET_ARP_CFG_HW_TYPE

The current version of **µC/TCP-IP** only supports Ethernet-type networks, and thus `NET_ARP_CFG_HW_TYPE` should ALWAYS be set to **`NET_ARP_HW_TYPE_ETHER`**.

3.08.02 ARP Configuration, NET_ARP_CFG_PROTOCOL_TYPE

The current version of **µC/TCP-IP** only supports IPv4, and thus `NET_ARP_CFG_PROTOCOL_TYPE` should ALWAYS be set to **`NET_ARP_PROTOCOL_TYPE_IP_V4`**.

3.08.03 ARP Configuration, NET_ARP_CFG_NBR_CACHE

ARP caches the mapping of IP addresses to physical (i.e. MAC) addresses. `NET_ARP_CFG_NBR_CACHE` configures the number of ARP cache entries. Each cache entry requires about 18 bytes of RAM plus 5 pointers plus a hardware address and protocol address (10 bytes assuming Ethernet interfaces and IPv4 addresses).

The number of ARP caches required by your application depends on how many different hosts you expect your product to communicate with. If your application ONLY communicates with hosts on remote networks via the local network's default gateway (i.e. router), then only a single ARP cache need be configured.

We tested **µC/TCP-IP** with a fairly small network and set the default number of ARP caches to `3`.

3.08.04 ARP Configuration, NET_ARP_CFG_ADDR_FLTR_EN

`NET_ARP_CFG_ADDR_FLTR_EN` determines whether address filtering is enabled or not. This configuration value can either be set to:

`DEF_DISABLED` . . . . Addresses are NOT filtered
or
`DEF_ENABLED` . . . . Addresses are filtered

3.09 IP (Internet Protocol) Configuration

3.09.01 IP Configuration, NET_IP_CFG_IF_MAX_NBR_ADDR

`NET_IP_CFG_IF_MAX_NBR_ADDR` determines the maximum number of IP addresses that may be configured per network interface at run-time. The default value of 1 is for a single IP address per network interface.

3.09.02 IP Configuration, NET_IP_CFG_MULTICAST_SEL

`NET_IP_CFG_MULTICAST_SEL` is used to determine the IP multicast support level. The allowable values for this parameter are :

`NET_IP_MULTICAST_SEL_NONE` . . . . .NO multicasting
`NET_IP_MULTICAST_SEL_TX` . . . . . . .Transmit multicasting only
`NET_IP_MULTICAST_SEL_TX_RX` . . . .Transmit and receive multicasting

3.10 ICMP (Internet Control Message Protocol) Configuration

3.10.01 ICMP Configuration, NET_ICMP_CFG_TX_SRC_QUENCH_EN

ICMP transmits ICMP source quench messages to other hosts when the Network Resources are low (see Section 10.02). `NET_ICMP_CFG_TX_SRC_QUENCH_EN` can be configured to either:

`DEF_DISABLED` . . . . ICMP Transmit Source Quenches disabled
or
`DEF_ENABLED` . . . . ICMP Transmit Source Quenches enabled

3.10.02 ICMP Configuration, NET_ICMP_CFG_TX_SRC_QUENCH_NBR

`NET_ICMP_CFG_TX_SRC_QUENCH_NBR` configures the number of ICMP transmit source quench entries. Each source quench entry requires about 12 bytes of RAM plus 2 pointers.

The number of entries depends on how many number different hosts you expect your product to communicate with. We recommend starting with a value of **`5`**.

3.11 IGMP (Internet Group Management Protocol) Configuration

3.11.01 IGMP Configuration, NET_IGMP_CFG_MAX_NBR_HOST_GRP

`NET_IGMP_CFG_MAX_NBR_HOST_GRP` configures the maximum number of IGMP host groups that may be joined at any one time. Each group entry requires about 12 bytes of RAM plus 3 pointers plus a protocol address (4 bytes assuming IPv4 address).

The number of IGMP host groups required by your application depends on how many host groups you expect to join at a given time. Since each configured multicast address requires its own IGMP host group, we recommend configuring at least 1 host group per multicast address used by your application, plus one additional host group. Thus for a single multicast address, we recommend starting with a value of **2**.

3.12 Transport Layer Configuration, NET_CFG_TRANSPORT_LAYER_SEL

µC/TCP-IP allows you to either select to include code for UDP or for both UDP and TCP. Most application software requires TCP. However, enabling only UDP would reduce both the code and data size required by **µC/TCP-IP**. `NET_CFG_TRANSPORT_LAYER_SEL` can be configured to either:

NET_TRANSPORT_LAYER_SEL_UDP_TCP
or
NET_TRANSPORT_LAYER_SEL_UDP

3.13 UDP (User Datagram Protocol) Configuration

3.13.01 UDP Configuration, NET_UDP_CFG_APP_API_SEL

NET_UDP_CFG_APP_API_SEL is used to determine where to send the de-multiplexed UDP datagram. Specifically, the datagram may be sent to the socket layer, only to a function you would write in your application or both. The allowable values for this parameter are:

NET_UDP_APP_API_SEL_SOCK . . . . De-multiplex receive datagrams to socket layer only

NET_UDP_APP_API_SEL_APP . . . . De-multiplex receive datagrams to your application only

NET_UDP_APP_API_SEL_SOCK_APP . . . De-multiplex receive datagrams to socket layer first, then to your application

Your application must define the following function to handle de-multiplexed receive datagrams:

```
void NetUDP_RxAppDataHandler (NET_BUF          *pbuf,
                              NET_IP_ADDR       src_addr,
                              NET_UDP_PORT_NBR  src_port,
                              NET_IP_ADDR       dest_addr,
                              NET_UDP_PORT_NBR  dest_port,
                              NET_ERR          *perr);
```

3.13.02 UDP Configuration, NET_UDP_CFG_RX_CHK_SUM_DISCARD_EN

NET_UDP_CFG_RX_CHK_SUM_DISCARD_EN is used to determine whether received UDP packets without a valid checksum are discarded or are handled & processed. Before a UDP Datagram Check-Sum is validated, it is necessary to check whether the UDP datagram was transmitted with or without a computed Check-Sum (see RFC #768, Section 'Fields : Checksum').

This configuration value can either be set to:

DEF_DISABLED .... ALL UDP datagrams received without a check-sum are flagged so that "an application may optionally discard datagrams without checksums" (see RFC #1122, Section 4.1.3.4).

or

DEF_ENABLED .... ALL UDP datagrams received without a checksum are discarded.

3.13.03 UDP Configuration, NET_UDP_CFG_TX_CHK_SUM_EN

NET_UDP_CFG_TX_CHK_SUM_EN is used to determine whether UDP checksums are computed for transmission to other hosts. An application MAY optionally be able to control whether a UDP checksum will be generated (see RFC #1122, Section 4.1.3.4).
This configuration value can either be set to:

DEF_DISABLED .... ALL UDP datagrams are transmitted without a computed checksum

or

DEF_ENABLED .... ALL UDP datagrams are transmitted with a computed checksum

3.14 TCP (Transport Control Protocol) Configuration

3.14.01 TCP Configuration, NET_TCP_CFG_NBR_CONN

`NET_TCP_CFG_NBR_CONN` configures the maximum number of TCP connections that **µC/TCP-IP** can handle concurrently. This number depends entirely on how many simultaneous TCP connections your product's applications will require. Each TCP connection requires about 220 bytes of RAM plus 16 pointers. We recommend starting with at least **`10`**.

3.14.02 TCP Configuration, NET_TCP_CFG_RX_WIN_SIZE_OCTET

`NET_TCP_CFG_RX_WIN_SIZE_OCTET` configures each TCP connections' receive window size. We recommend setting TCP window sizes to integer multiples of each TCP connection's maximum segment size (MSS). For example, systems with an Ethernet MSS of 1460, a value 5840 (4 * 1460) is probably a better configuration than the default window size of **`4096`** (4K).

3.14.03 TCP Configuration, NET_TCP_CFG_TX_WIN_SIZE_OCTET

`NET_TCP_CFG_TX_WIN_SIZE_OCTET` configures each TCP connections' transmit window size. We recommend setting TCP window sizes to integer multiples of each TCP connection's maximum segment size (MSS). For example, systems with an Ethernet MSS of 1460, a value 5840 (4 * 1460) is probably a better configuration than the default window size of **`4096`** (4K).

3.14.04 TCP Configuration, NET_TCP_CFG_TIMEOUT_CONN_MAX_SEG_SEC

NET_TCP_CFG_TIMEOUT_CONN_MAX_SEG_SEC configures TCP connections' default maximum segment lifetime timeout (MSL) value, specified in integer seconds. We recommend starting with a value of **3 seconds**.

If TCP connections are established and closed rapidly, it is possible that this timeout may further delay new TCP connections from becoming available. Thus, an even lower timeout value may be desirable to free TCP connections and make them available for new connections as fast as possible. However, a **0 second** timeout prevents **µC/TCP-IP** from performing the complete TCP connection close sequence and will instead send TCP reset (RST) segments.

3.14.05 TCP Configuration, NET_TCP_CFG_TIMEOUT_CONN_ACK_DLY_MS

NET_TCP_CFG_TIMEOUT_CONN_ACK_DLY_MS configures the TCP acknowledgement delay in integer milliseconds. We recommend configuring the default value of **500 milliseconds** since RFC #2581, Section 4.2 states that "an ACK MUST be generated within 500 ms of the arrival of the first unacknowledged packet".

3.14.06 TCP Configuration, NET_TCP_CFG_TIMEOUT_CONN_RX_Q_MS

NET_TCP_CFG_TIMEOUT_CONN_RX_Q_MS configures each TCP connections' receive timeout (in milliseconds OR no timeout if configured with NET_TMR_TIME_INFINITE). We recommend starting with a value of **3000** milliseconds OR the no-timeout value of **NET_TMR_TIME_INFINITE**.

3.14.07 TCP Configuration, NET_TCP_CFG_TIMEOUT_CONN_TX_Q_MS

NET_TCP_CFG_TIMEOUT_CONN_TX_Q_MS configures each TCP connections' transmit timeout (in milliseconds OR no timeout if configured with NET_TMR_TIME_INFINITE). We recommend starting with a value of **3000** milliseconds OR the no-timeout value of **NET_TMR_TIME_INFINITE.**

3.15 BSD v4 Sockets Configuration

μC/TCP-IP supports BSD 4.x sockets and basic socket API for the TCP/UDP/IP protocols.

3.15.01 BSD v4 Sockets Configuration, NET_SOCK_CFG_FAMILY

The current version of **μC/TCP-IP** only supports IPv4 BSD sockets, and thus NET_SOCK_CFG_FAMILY should ALWAYS be set to **NET_SOCK_FAMILY_IP_V4**.

3.15.02 BSD v4 Sockets Configuration, NET_SOCK_CFG_NBR_SOCK

NET_SOCK_CFG_NBR_SOCK configures the maximum number of sockets that **μC/TCP-IP** can handle concurrently. This number depends entirely on how many simultaneous socket connections your product's applications will require. Each socket requires about 28 bytes of RAM plus 3 pointers. We recommend starting with at least **10**.

3.15.03 BSD v4 Sockets Configuration, NET_SOCK_CFG_BLOCK_SEL

NET_SOCK_CFG_BLOCK_SEL determines the default blocking (or non-blocking) behavior for sockets. This parameter can take the following values:

NET_SOCK_BLOCK_SEL_DFLT	Sockets will be blocking by default, but may be individually configured in a future release
NET_SOCK_BLOCK_SEL_BLOCK	Sockets will be blocking by default

`NET_SOCK_BLOCK_SEL_NO_BLOCK` .... Sockets will be non-blocking by default

If you select blocking mode, you can block with a timeout. The amount of time for the timeout is determined by various timeout functions implemented in `net_sock.c`:

`NetSock_CfgTimeoutRxQ_Set()` .... Configure datagram socket receive timeout
`NetSock_CfgTimeoutConnReqSet()` ..... Configure socket connection timeout
`NetSock_CfgTimeoutConnAcceptSet()` ... Configure socket accept timeout
`NetSock_CfgTimeoutConnCloseSet()` .... Configure socket close timeout

3.15.04 BSD v4 API Configuration, NET_SOCK_CFG_SEL_EN

`NET_SOCK_CFG_SEL_EN` determines whether the code and data space used to support socket `select()` functionality is enabled or not. This configuration value can either be set to:

`DEF_DISABLED` .... BSD `select()` API disabled
or
`DEF_ENABLED` .... BSD `select()` API enabled

3.15.05 BSD v4 Sockets Configuration, NET_SOCK_CFG_SEL_NBR_EVENTS_MAX

`NET_SOCK_CFG_SEL_NBR_EVENTS_MAX` is used to configure the maximum number of socket events/operations that the socket select() functionality can wait on. We recommend starting with a value no more than **`10`**.

3.15.06 BSD v4 Sockets Configuration, NET_SOCK_CFG_CONN_ACCEPT_Q_SIZE_MAX

`NET_SOCK_CFG_CONN_ACCEPT_Q_SIZE_MAX` is used to configure the absolute maximum queue size of `accept()` connections for stream-type sockets. We recommend starting with a value of **5**.

3.15.07 BSD v4 Sockets Configuration, NET_SOCK_CFG_PORT_NBR_RANDOM_BASE

`NET_SOCK_CFG_PORT_NBR_RANDOM_BASE` is used to configure the starting base socket number for 'ephemeral' or 'random' port numbers. Since two times the number of random ports are required for each socket, the base value for the random port number must be :

```
Random Port Number Base  <=  65535 - (2 * NET_SOCK_CFG_NBR_SOCK)
```

We recommend the arbitrary default value of **65000** as a good starting point.

3.15.08 BSD v4 Sockets Configuration, NET_SOCK_CFG_TIMEOUT_RX_Q_MS

`NET_SOCK_CFG_TIMEOUT_RX_Q_MS` configures socket timeout value (in milliseconds OR no timeout if configured with `NET_TMR_TIME_INFINITE`) for datagram socket `recv()` operations. We recommend starting with a value of **3000** milliseconds OR the no-timeout value of **NET_TMR_TIME_INFINITE**.

3.15.09 BSD v4 Sockets Configuration, NET_SOCK_CFG_TIMEOUT_CONN_REQ_MS

`NET_SOCK_CFG_TIMEOUT_CONN_REQ_MS` configures socket timeout value (in milliseconds OR no timeout if configured with `NET_TMR_TIME_INFINITE`) for stream socket `connect()` operations. We recommend starting with a value of **3000** milliseconds OR the no-timeout value of **NET_TMR_TIME_INFINITE**.

3.15.10 BSD v4 Sockets Configuration, NET_SOCK_CFG_TIMEOUT_CONN_ACCEPT_MS

NET_SOCK_CFG_TIMEOUT_CONN_ACCEPT_MS configures socket timeout value (in milliseconds OR no timeout if configured with NET_TMR_TIME_INFINITE) for socket accept() operations. We recommend starting with a value of **3000** milliseconds OR the no-timeout value of **NET_TMR_TIME_INFINITE**.

3.15.11 BSD v4 Sockets Configuration, NET_SOCK_CFG_TIMEOUT_CONN_CLOSE_MS

NET_SOCK_CFG_TIMEOUT_CONN_CLOSE_MS configures socket timeout value (in milliseconds OR no timeout if configured with NET_TMR_TIME_INFINITE) for socket close() operations. We recommend starting with a value of **3000** milliseconds OR the no-timeout value of **NET_TMR_TIME_INFINITE**.

3.15.12 BSD v4 Sockets Configuration, NET_BSD_CFG_API_EN

NET_BSD_CFG_API_EN determines whether the standard BSD 4.x socket API is enabled or not. This configuration value can either be set to:

DEF_DISABLED . . . . BSD 4.x layer API disabled
or
DEF_ENABLED . . . . BSD 4.x layer API enabled

3.16 Network Connection Manager Configuration

3.16.01 Network Connection Manager Configuration, NET_CONN_CFG_FAMILY

The current version of **µC/TCP-IP** only supports IPv4 connections, and thus NET_CONN_CFG_FAMILY should ALWAYS be set to **NET_CONN_FAMILY_IP_V4_SOCK**.

3.16.02 Network Connection Manager Configuration, NET_CONN_CFG_NBR_CONN

NET_CONN_CFG_NBR_CONN configures the maximum number of connections that **µC/TCP-IP** can handle concurrently. This number depends entirely on how many simultaneous connections your product's applications will require and MUST be at least greater than the configured number of application (socket) connections and transport layer (TCP) connections. Each connection requires about 28 bytes of RAM plus 5 pointers plus two protocol addresses (8 bytes assuming IPv4 addresses). We recommend starting with at least **20**.

3.17 Application-Specific Configuration, app_cfg.h

This section defines the configuration constants that are related to **µC/TCP-IP** but are application-specific. Most of these configuration constants relate to the various ports for **µC/TCP-IP** such as the CPU, OS, device, or network interface ports. Some of the configuration constants relate to the compiler and standard library ports.

These configuration constants should be defined in an application's **app_cfg.h** file.

3.17.01 Application-Specific Configuration, Operating System Configuration

The following configuration constants relate to the **µC/TCP-IP** OS port. For many OSs, the **µC/TCP-IP** task priorities, stack sizes, and other options will need to be explicitly configured for the particular OS (consult the specific OS's documentation for more information).

The priority of **µC/TCP-IP** tasks is dependent on the network communication requirements of the application. For most applications, the priority for **µC/TCP-IP** tasks is typically lower than the priority for other application tasks.

For **µC/OS-II**, the following macros must be configured within `app_cfg.h`:

`NET_OS_CFG_IF_TX_DEALLOC_PRIO` .... `50`
`NET_OS_CFG_IF_RX_TASK_PRIO` .... `51`
`NET_OS_CFG_IF_LOOPBACK_TASK_PRIO` .... `52`
`NET_OS_CFG_TMR_TASK_PRIO` .... `53`

The arbitrary task priorities of **`50`** through **`53`** are a good starting point for most applications, where the Network Interface Transmit De-allocation Task is assigned the highest priority of all network tasks followed by the Network Interface Receive Task, the Network Interface Loopback Task, and lastly, the Network Timer Task.

`NET_OS_CFG_IF_TX_DEALLOC_TASK_STK_SIZE` .... `1000`
`NET_OS_CFG_IF_RX_TASK_STK_SIZE` .... `1000`
`NET_OS_CFG_IF_LOOPBACK_TASK_STK_SIZE` .... `1000`
`NET_OS_CFG_TMR_TASK_STK_SIZE` .... `1000`

The arbitrary stack size of **`1000`** is a good starting point for most applications.

3.17.02 Application-Specific Configuration, μC/TCP-IP Configuration

The following configuration constants relate to the **μC/TCP-IP** OS port. For many OSs, the **μC/TCP-IP** maximum queue sizes may need to be explicitly configured for the particular OS (consult the specific OS's documentation for more information).

For **μC/OS-II**, the following macros must be configured within `app_cfg.h`:

NET_OS_CFG_IF_RX_Q_SIZE
NET_OS_CFG_IF_TX_DEALLOC_Q_SIZE
NET_OS_CFG_IF_LOOPBACK_Q_SIZE

The values configured for these macros depend on additional application dependent information such as the number of transmit or receive buffers configured for the total number of interfaces.

We recommend the following configuration for the above macros as follows:

`NET_OS_CFG_IF_RX_Q_SIZE` should be configured such that it reflects the total number of DMA receive descriptors on all physical interfaces. If DMA is not available, or a combination of DMA and I/O based interfaces are configured then this number reflects the maximum number of packets than can be acknowledged and signaled for during a single receive interrupt event for all interfaces.

For example, if one interface has 10 receive descriptors and another interface is I/O based but is capable of receiving 4 frames within its internal memory and issuing a single interrupt request, then the `NET_OS_CFG_IF_RX_Q_SIZE` macro should be configured to 14. Defining a number in excess of the maximum number of receivable frames per interrupt across all interfaces would not hurt anything, but the additional queue space will not be utilized.

`NET_OS_CFG_IF_TX_DEALLOC_Q_SIZE` should be defined to be the total number of small and large transmit buffers declared for all interfaces excluding the loopback interface.

`NET_OS_CFG_IF_LOOPBACK_Q_SIZE` should be defined to be the total number of large receive buffers declared for the loopback interface.

Chapter

4

Network Interface Layer

The following sections describe how **µC/TCP-IP** manages the interaction between devices, device drivers, and network interfaces.

4.01 Network Interface Configuration

4.01.01 Interface / Device / Phy Configuration

All network devices, including secondary devices such as Ethernet Phy's, require that the application developer provide a configuration structure during compile time for each device.

All device configuration structures and declarations must be placed within application provided files named `net_dev_cfg.c` and `net_dev_cfg.h`.

4.01.01.01 Loopback Configuration

Listing 4-1 Sample Loopback Interface Configuration

```
NET_IF_CFG_LOOPBACK  NetIF_Cfg_Loopback = {
    NET_IF_MEM_TYPE_MAIN,                              (1)
          1518,                                        (2)
            10,                                        (3)
             4,                                        (4)

    NET_IF_MEM_TYPE_MAIN,                              (5)
          1594,                                        (6)
             5,                                        (7)
           256,                                        (8)
             5,                                        (9)

             4,                                       (10)

    0x00000000,                                       (11)
             0,                                       (12)
};
```

L4-1(1) Receive buffer pool type. This configuration setting controls the memory placement of the receive data buffers. Buffers may either be placed in main memory or in a dedicated, perhaps higher speed, memory region (see #11). This field should be set to one of the two macros:

```
NET_IF_MEM_TYPE_MAIN
NET_IF_MEM_TYPE_DEDICATED
```

L4-1(2) Receive buffer size. This field sets the size of the largest receivable packet and may be set to match the applications requirements.

NOTE: If packets are sent from a socket bound to a non local-host address, to the local host address, e.g. `127.0.0.1`, then the receive buffer size must be configured to match the maximum transmit buffer size, or maximum expected data size, that could be generated from a socket bound to any other interface.

L4-1(3) Number of receive buffers. This setting controls the number of receive buffers that will be allocated to the loopback interface. This value MUST be set greater than or equal to one buffer if loopback is receiving ONLY UDP. If TCP data is expected to be transferred across the loopback interface, then there MUST be a minimum of four receive buffers.

L4-1(4) Receive buffer alignment. This setting controls the alignment of the receive buffers in bytes. Some processor architectures do not allow multi-byte reads and writes across word boundaries and therefore may require buffer alignment. In general, it is probably best practice to align buffers to the data bus width of the processor which may improve performance. For example, a 32-bit processor may benefit from having buffers aligned on a 4 byte boundary.

L4-1(5) Transmit buffer pool type. This configuration setting controls the memory placement of the transmit data buffers for the loopback interface. Buffers may either be placed in main memory or in a dedicated, perhaps higher speed, memory region (see #11). This field should be set to one of the two macros:

```
NET_IF_MEM_TYPE_MAIN
NET_IF_MEM_TYPE_DEDICATED
```

L4-1(6) Large transmit buffer size. At the time of this writing, transmit fragmentation is NOT supported; therefore this field sets the size of the largest transmittable buffer for the loopback interface when the application sends from a socket that is bound to the local-host address.

L4-1(7) Number of large transmit buffers. This field controls the number of large transmit buffers allocated to the loopback interface. The developer may set this field to zero to make room for additional large transmit buffers, however, the number of large plus the number of small transmit buffers MUST be greater than or equal to one for UDP traffic and three for TCP traffic.

L4-1(8) Small transmit buffer size. For devices with a minimal amount of RAM, it is possible to allocate small transmit buffers as well as large transmit buffers. In general, Micrium recommends 256 byte small transmit buffers, however, the developer may adjust this value according to the application requirements. This field has no effect if the number of small transmit buffers is configured to zero.

L4-1(9) Number of small transmit buffers. This field controls the number of small transmit buffers allocated to the device. The developer may set this field to zero to make room for additional large transmit buffers, however, the number of large plus the number of small transmit buffers MUST be greater than or equal to one for UDP traffic and three for TCP traffic.

L4-1(10) Transmit buffer alignment. This setting controls the alignment of the receive buffers in bytes. Some processor architectures do not allow multi-byte reads and writes across word boundaries and therefore may require buffer alignment. In general, it is probably best practice to align buffers to the data bus width of the processor which may improve performance. For example, a 32-bit processor may benefit from having buffers aligned on a 4 byte boundary.

L4-1(11) Memory Address. By default, this field is configured to `0x00000000`. A value of `0` tells the Network Protocol Suite to allocate buffers for the loopback interface from the **µC/LIB** Memory Manager default heap. If a faster, more specialized memory is available, the loopback interface buffers may be allocated into an alternate region if desired.

L4-1(12) Memory Size. By default, this field is configured to `0`. A value of `0` tells the Network Protocol Suite to allocate as much memory as required from the **µC/LIB** Memory Manager default heap. If an alternate memory region is specified in the 'Memory Address' field above, then the maximum size of the specified memory segment must be specified.

4.01.01.02.01 Ethernet Device MAC Configuration

Listing 4-2 shows a sample configuration structure for a single Freescale FEC MAC on the MCF5275.

Listing 4-2 Sample Freescale FEC Configuration

```
NET_DEV_CFG_ETHER  NetDev_Cfg_FEC_0 = {
    NET_IF_MEM_TYPE_MAIN,                              (1)
          1518,                                        (2)
            10,                                        (3)
            16,                                        (4)

    NET_IF_MEM_TYPE_MAIN,                              (5)
          1594,                                        (6)
             5,                                        (7)
           256,                                        (8)
             5,                                        (9)

            16,                                       (10)

    0x00000000,                                       (11)
             0,                                       (12)

             5,                                       (13)
            10,                                       (14)

    0x40001000,                                       (15)

             0,                                       (16)
        DEF_NO,                                       (17)

   "00:50:C2:25:60:02"                                (18)
};
```

L4-2(1) Receive buffer pool type. This configuration setting controls the memory placement of the receive data buffers. Buffers may either be placed in main memory or in a dedicated memory region. Non DMA based Ethernet controllers should be configured to use the main memory pool. DMA based Ethernet controllers may or may not require dedicated memory (see #14). This depends on the type of controller being configured. This field should be set to one of the two macros:

```
NET_IF_MEM_TYPE_MAIN
NET_IF_MEM_TYPE_DEDICATED
```

L4-2(2) Receive buffer size. For Ethernet, this setting is generally configured to 1518 bytes which represents the MTU of an Ethernet network. For DMA based Ethernet controllers, the developer MUST configure the receive data buffer size greater or equal to the size of the largest receivable frame.

L4-2(3) Number of receive buffers. This setting controls the number of receive buffers that will be allocated to the device. For DMA devices, this number MUST be greater than `1`. If the size of the total buffer allocation is greater than the amount of available memory in the chosen memory region, a run-time error will be generated when the device is initialized.

L4-2(4) Receive buffer alignment. This setting controls the alignment of the receive buffers in bytes. Some devices, such as the Freescale FEC require that the receive buffers be aligned to a specific byte boundary. Additionally, some processor architectures do not allow multi-byte reads and writes across word boundaries and therefore may require buffer alignment. In general, it is probably best practice to align buffers to the data bus width of the processor which may improve performance. For example, a 32-bit processor may benefit from having buffers aligned on a 4 byte boundary.

L4-2(5) Transmit buffer pool type. This configuration setting controls the memory placement of the transmit data buffers. Buffers may either be placed in main memory or in a dedicated memory region. Non DMA based Ethernet controllers should be configured to use the main memory pool. DMA based Ethernet controllers may or may not require dedicated memory (see #14). In some cases, DMA based Ethernet controllers with dedicated memory may be able to transmit from main memory. If so, then the remaining dedicated memory may be allocated to additional receive buffers. This field should be set to one of the two macros:

```
NET_IF_MEM_TYPE_MAIN
NET_IF_MEM_TYPE_DEDICATED
```

L4-2(6) Large transmit buffer size. This field controls the size of the large transmit buffers allocated to the device in bytes. This field has no effect if the number of large transmit buffers is configured to zero. Setting the size of the large transmit buffers below 1594, bytes may hinder the stacks ability to transmit full sized IP datagrams since IP transmit fragmentation is not yet supported. Micrium recommends setting this field to 1594 bytes in order to accommodate the Network Protocol Suites internal packet building mechanisms.

L4-2(7) Number of large transmit buffers. This field controls the number of large transmit buffers allocated to the device. The developer may set this field to zero to make room for additional large transmit buffers, however, the size of the maximum transmittable UDP packet will depend on the size of the small transmit buffers, see #8.

L4-2(8) Small transmit buffer size. For devices with a minimal amount of RAM, it is possible to allocate small transmit buffers as well as large transmit buffers. In general, Micrium recommends 256 byte small transmit buffers, however, the developer may adjust this value according to the application requirements. This field has no effect if the number of small transmit buffers is configured to zero.

L4-2(9) Number of small transmit buffers. This field controls the number of small transmit buffers allocated to the device. The developer may set this field to zero to make room for additional large transmit buffers if required.

L4-2(10) Transmit buffer alignment. This setting controls the alignment of the transmit buffers in bytes. Some devices, such as the Freescale FEC require that the transmit buffers be aligned to a specific byte boundary. Additionally, some processor architectures do not allow multi-byte reads and writes across word boundaries and therefore may require buffer alignment. In general, it is probably best practice to align buffers to the data bus width of the processor which may improve performance. For example, a 32-bit processor may benefit from having buffers aligned on a 4 byte boundary.

L4-2(11) Memory Address. For devices with dedicated memory, this field represents the starting address of the dedicated memory region. Non dedicated memory based devices may initialize this field to `0`.

L4-2(12) Memory Size. For devices with dedicated memory, this field represents the size of the dedicated memory region in bytes. Non dedicated memory based devices may initialize this field to `0`.

L4-2(13) Number of receive descriptors. For DMA based devices, this value is utilized by the device driver during initialization in order to allocate a fixed size pool of receive descriptors to be used by the device. The number of descriptors MUST be less than the number of configured receive buffers. Micrium recommends setting this value to approximately one half of the number of receive buffers. Non DMA based devices may configure this value to zero.

L4-2(14) Number of transmit descriptors. For DMA based devices, this value is utilized by the device driver during initialization in order to allocate a fixed size pool of transmit descriptors to be used by the device. For best performance, the number of transmit descriptors should be equal to the number of small, plus the number of large transmit buffers configured for the device. Non DMA based devices may configure this value to zero.

L4-2(15) Device base address. This field represents the base register address of the device. This field is used by the device driver to determine the address of device registers given this base address and pre-defined register offset within the driver.

L4-2(16) Device data bus size configured in number of bits. For devices with configurable data bus sizes, it may be desirable to specify the width of the data bus in order for well written device drivers to correctly configure the device during initialization. For devices that abstract the data bus from the developer, such as MCU integrated MAC's, this value should be specified as zero and ignored by the driver; for all external devices, this value should be defined to `8`, `16`, `32`, or `64`.

L4-2(17) Configure to either `DEF_YES or DEF_NO` (default) to swap data octets; i.e. swap data words' high-order octet(s) with data words' low-order octet(s), and vice-versa if required by device-to-CPU data bus wiring and or CPU endian word order.

L4-2(18) Hardware address. For Ethernet devices, the hardware address field provides a location to hard code the device's MAC address is string format. This must be configured in one of the following three ways:

a. `"aa:bb:cc:dd:ee:ff"` where aa, bb, cc, dd, ee, and ff represent the desired static MAC address octets which are to be configured to the device during initialization.

b. `"00:00:00:00:00:00"` where a zero value MAC address string disables static configuration of the device MAC address. If this mechanism is used, then the driver must check if the user has configured a MAC address during run-time, or if the MAC address is automatically loaded from a non-volatile memory source. Micrium recommends this method of configuring the MAC address when the MAC address is to be determined during run-time via hard coding in software, or loading from an external memory device.

c. `""` where an empty MAC address disables static configuration of the device MAC address. If this mechanism is used, then the driver must check if the user has configured a MAC address during run-time, or if the MAC address is automatically loaded from a non-volatile memory source.

4.01.01.02.02 Ethernet Phy Configuration

Listing 4-3 shows a typical Ethernet Phy configuration structure.

Listing 4-3 Sample Ethernet Phy Configuration

```
NET_PHY_CFG_ETHER  NetPhy_Cfg_FEC_0 = {
    1,                                          (1)
    NET_PHY_BUS_MODE_MII,                       (2)
    NET_PHY_TYPE_EXT                            (3)
    NET_PHY_SPD_AUTO,                           (4)
    NET_PHY_DUPLEX_AUTO,                        (5)
};
```

L4-3(1) Phy Address. This field represents the address of the Phy on the (R)MII bus.

The value configured depends on the Phy and the state of the Phy pins during power-up. Developers may need to consult the schematics for their board in order to determine the configured Phy address. Alternatively, the Phy address may be detected automatically by specifying `NET_PHY_ADDR_AUTO`; however, this will increase the initialization latency of the Network Protocol Suite and possibly the rest of the application depending on where the application places the call to `NetIF_Start()`.

L4-3(2) Phy bus mode. This value should be set to one of the following values depending on the hardware capabilities, and schematics of the development board. The NET BSP will be written to match the configured value.

```
NET_PHY_BUS_MODE_MII
NET_PHY_BUS_MODE_RMII
NET_PHY_BUS_MODE_SMII
```

L4-3(3) Phy bus type. This field represents the type of electrical attachment of the Phy to the Ethernet controller. In some cases, the Phy may be internal to the network controller, while in other cases, it may be attached via an external MII or RMII bus. It is desirable to specify which attachment method is in use so that a device driver can initialize additional hardware resources should an external Phy be attached to a device that also has an internal Phy. Available settings for this field are:

```
NET_PHY_TYPE_INT
NET_PHY_TYPE_EXT
```

L4-3(4) Initial Phy link speed. This configuration setting will force the Phy to link to the specified link speed. Optionally, auto-negotiation may be enabled. This field must be set to one of the following values:

```
NET_PHY_SPD_AUTO
NET_PHY_SPD_10
NET_PHY_SPD_100
NET_PHY_SPD_1000
```

L4-3(5) Initial Phy link duplex. This configuration setting will force the Phy to link using the specified duplex. This setting must be set to one of the following values:

```
NET_PHY_DUPLEX_AUTO
NET_PHY_DUPLEX_HALF
NET_PHY_DUPLEX_FULL
```

4.01.02 Adding Network Interfaces

In **µC/TCP-IP**, the term Network Interfaces is used to represent an abstract view of the device hardware and data path that connects the hardware to the higher layers of the Network Protocol Stack. In order to communicate with hosts outside the localhost, the application developer must add at least one Network Interface to the system. Note that the first interface added and started will be the default interface used for all default communication.

A typical call to `NetIF_Add()` is shown below. See Section A.10.01 for more details.

Listing 4-4 Calling NetIF_Add()

```
if_nbr = NetIF_Add((void    *)&NetIF_API_Ether,        (1)
                   (void    *)&NetDev_API_STR912,      (2)
                   (void    *)&NetDev_Cfg_STR912_0,    (3)
                   (void    *)&NetPHY_API_Generic,     (4)
                   (void    *)&NetPhy_Cfg_STR912_0,    (5)
                   (NET_ERR *)&err);                   (6)
```

L4-4(1) The first argument specifies the link layer API that will receive data from the hardware device. For an Ethernet interface, this value will always be defined as `NetIF_API_Ether`. This symbol is defined by **µC/TCP-IP** and can be used to add as many Ethernet Network Interface's as necessary.

L4-4(2) The second argument represents the hardware device driver API which is defined as a fixed structure of function pointers of the type specified by Micrium for use with **µC/TCP-IP**. If Micrium supplies the device driver, then the symbol name of the device API will be defined within the device driver documentation and at the top of the device driver source code file. Otherwise, the device driver developer is responsible for creating the device driver and the device driver API structure. For more information pertaining to the device driver API, see the *µC/TCP-IP Driver Architecture* document.

L4-4(3) The third argument specifies the device driver configuration structure that will be used to configure the device hardware for the interface being added. The device configuration structure format has been specified by Micrium and must be provided by the application developer since it is specific to the selection of device hardware and design of the evaluation board. Micrium may be able to supply example device configuration structures for some evaluation boards. For more information about declaring a device configuration structure, see Section 4.01.01.02.01.

L4-4(4) The fourth argument represents the physical layer hardware device API. In most cases, when Ethernet is the link layer API specified as argument number 1, the physical layer API may be defined as `NetPHY_API_Generic`. This symbol has been defined by the generic Ethernet physical layer device driver which can be supplied by Micrium. If a custom physical layer device driver is required, then the device driver developer would be responsible for creating the API structure. Some Ethernet devices have built in Physical layer devices which on occasion, are NOT (R)MII compliant. In this circumstance, the Physical layer device driver API field may be left `NULL` and the Ethernet device driver may implement routines for the built in Phy. For more information about the physical layer hardware device API, see the *µC/TCP-IP Driver Architecture* document.

L4-4(5) The fifth argument represents the physical layer hardware device configuration structure. This structure is specified by the application developer and contains information such as the physical device connection type, address, and desired link state upon initialization. For devices with built in non (R)MII compliant Physical layer devices, this field may be left `NULL`. However, it may be convenient to declare a Physical layer device configuration structure and use some of the members for Physical layer device initialization from within the Ethernet device driver. For more information about declaring a physical layer hardware configuration structure, see Section 4.01.01.02.02.

L4-4(6) The last argument is a pointer to a `NET_ERR` variable that will contain the return error code for `NetIF_Add()`. This variable SHOULD be checked by the application to ensure that no errors have occurred during network interface addition. Upon success, the return error code will be `NET_IF_ERR_NONE`.

NOTE: If an error occurs during the call to `NetIF_Add()`, the application **MUST NOT** attempt to call `NetIF_Add()` a second time for the same interface. Instead, the application developer should observe the error code, determine and resolve the cause of the error, rebuild the application and try again. If a hardware failure has occurred, the interface will not operate and should be left in the default disabled state. Additional interfaces may be added and may continue to operate as expected. One common problem to watch for is a

µC/LIB Memory Manager heap under-run condition. This may occur when adding network interfaces when there is not enough memory is available to complete the operation. If this error occurs, try increasing the configured size of the **µC/LIB** heap configured within `app_cfg.h`.

Once an interface has been added successfully, the next step is to configure the interface with one or more network layer protocol addresses.

4.01.03 Configuring an Internet Protocol Address

Each Network Interface must be configured with at least one Internet Protocol address. This may be performed using **µC/DHCPc** or manually during run-time. If run-time configuration is chosen, then the following functions may be utilized in order to set the IP, Net Mask, and Gateway address for a specific interface. More than one set of addresses may be configured for a specific network interface by calling the functions below. Note that on the default interface, the first IP address added will be the default address used for all default communication.

```
NetASCII_Str_to_IP()
NetIP_CfgAddrAdd()
```

The first function aids the developer by converting a string format IP address such as "192.168.1.2" to its hexadecimal equivalent. The second function is used to configure an interface with the specified IP, Net Mask and Gateway addresses. An example of each function call is shown below.

Listing 4-5 Calling NetASCII_Str_to_IP()

```
ip     = NetASCII_Str_to_IP((CPU_CHAR*)"192.168.1.2",   &err); (1)
msk    = NetASCII_Str_to_IP((CPU_CHAR*)"255.255.255.0", &err);
gateway= NetASCII_Str_to_IP((CPU_CHAR*)"192.168.1.1",   &err);
```

L4-5(1) `NetASCII_Str_to_IP()` requires two arguments. The first function argument is a string representing a valid IP address that you would like to use, and the second argument is a pointer to a `NET_ERR` to contain the return error code. Upon successfully conversion, the return error will contain the value `NET_ASCII_ERR_NONE` and the function will return a variable of type `NET_IP_ADDR` containing the hexadecimal equivalent of the specified address.

Listing 4-6 Calling NetIP_CfgAddrAdd ()

```
cfg_success = NetIP_CfgAddrAdd(if_nbr,              (1)
                                ip,                 (2)
                                msk,                (3)
                                gateway,            (4)
                               &err);               (5)
```

L4-6(1) The first argument is the number representing the Network Interface that is to be configured. This value is obtained as the result of a *successful* call to `NetIF_Add()`.

L4-6(2) The second argument is the `NET_IP_ADDR` value representing the IP address that is to be configured.

L4-6(3) The third argument is the `NET_IP_ADDR` value representing the Subnet Mask address that is to be configured.

L4-6(4) The fourth argument is the `NET_IP_ADDR` value representing the Default Gateway IP address that is to be configured.

L4-6(5) The fifth argument is a pointer to a `NET_ERR` variable containing the return error code for the function. If the interface address information is configured successfully, then the return error code will contain the value `NET_IP_ERR_NONE`. Additionally, function returns a Boolean value of `DEF_OK` or `DEF_FAIL` depending on the result. Either the return value or the `NET_ERR` variable may be checked for return status; however, the `NET_ERR` contains more detailed information and should therefore be the preferred check.

NOTE: The application may configure a Network Interface with more than one set of IP addresses. This may be desirable when a Network Interface and its paired device are connected to a switch or HUB with more than one network present. Additionally, an application may choose to NOT configure any interface addresses, and thus may ONLY receive packets and should not attempt to transmit.

Additionally, addresses may be removed from an interface by calling `NetIP_CfgAddrRemove()` [see Sections A.11.05 & A.11.06].

Once a Network Interface has been successfully configured with Internet Protocol address information, the next step is to start the interface.

4.02 Starting & Stopping Network Interfaces

4.02.01 Starting Network Interfaces

When a Network Interface is 'Started', it becomes an active interface that is capable of transmitting and receiving data assuming an operational link to the network medium. A Network Interface may be started any time after the Network Interface has been successfully 'Added' to the system. A successful call to `NetIF_Start()` marks the end of the initialization sequence of **µC/TCP-IP** for a specific Network Interface. Recall that the first interface added and started will be the default interface used for all default communication.

The application developer may start a Network Interface by calling the `NetIF_Start()` API function with the necessary parameters. A call to `NetIF_Start()` is shown below.

Listing 4-7 Calling NetIF_Start()

```
NetIF_Start(if_nbr, &err);     (1)
```

L4-7(1) `NetIF_Start()` requires two arguments. The first function argument is the interface number that the application wants to start, and the second argument is a pointer to a `NET_ERR` to contain the return error code. The interface number is acquired upon successful addition of the interface and upon the successful start of the interface; the return error variable will contain the value `NET_IF_ERR_NONE`.

There are very few things that could cause a Network Interface to not start properly. The application developer should always inspect the return error code and take the appropriate action if an error occurs. Once the error is resolved, the application may attempt to call `NetIF_Start()` again.

4.02.02 Stopping Network Interfaces

Under some circumstances, it may be desirable to stop a network interface. A Network Interface may be stopped any time after it has been successfully 'Added' to the system. Stopping an interface may be performed by calling `NetIF_Stop()` with the appropriate arguments as shown below.

Listing 4-8 Calling NetIF_Stop()

```
NetIF_Stop(if_nbr, &err);      (1)
```

L4-8(1) `NetIF_Stop()` requires two arguments. The first function argument is the interface number that the application wants to stop, and the second argument is a pointer to a `NET_ERR` to contain the return error code. The interface number is acquired upon the successful addition of the interface and upon the successful stop of the interface; the return error variable will contain the value `NET_IF_ERR_NONE`.

There are very few things that could cause a Network Interface to not stop properly. The application developer should always inspect the return error code and take the appropriate action if an error occurs. Once the error is resolved, the application may attempt to call `NetIF_Stop()` again.

4.03 Network Interfaces' MTU

4.03.01 Getting Network Interface MTU

On occasion, it may be desirable to have the application aware of an interface's Maximum Transmission Unit. The MTU for a particular interface may be acquired by calling `NetIF_MTU_Get()` with the appropriate arguments.

Listing 4-9 Calling NetIF_MTU_Get()

```
mtu = NetIF_MTU_Get(if_nbr, &err);      (1)
```

L4-9(1) `NetIF_MTU_Get()` requires two arguments. The first function argument is the interface number to obtain the current configured MTU, and the second argument is a pointer to a `NET_ERR` to contain the return error code. The interface number is acquired upon the successful addition of the interface, and upon the successful return of the function, the return error variable will contain the value `NET_IF_ERR_NONE`. The result is returned into a local variable of type `NET_MTU`.

4.03.02 Setting Network Interface MTU

Some networks prefer to operate with a non standard MTU. Should this be the case, the application may specify the MTU for a particular interface by calling `NetIF_MTU_Set()` with the appropriate arguments.

Listing 4-10 Calling NetIF_MTU_Set()

```
NetIF_MTU_Set(if_nbr, mtu, &err);      (1)
```

L4-10(1) `NetIF_MTU_Set()` requires three arguments. The first function argument is the interface number of the interface to set the specified MTU. The second argument is the desired MTU to set, and the third argument is a pointer to a `NET_ERR` variable that will contain the return error code. The interface number is acquired upon the successful addition of the interface, and upon the successful return of the function, the return error variable will contain the value `NET_IF_ERR_NONE` and the specified MTU will be set.

NOTE: The configured MTU cannot be greater than the largest configured transmit buffer size associated with the specified interfaces' device minus some overhead. The transmit buffer sizes are specified in the device configuration structure for the specified interface. For more information about configuring device buffer sizes, please see the *µC/TCP-IP Driver Architecture* document.

4.04 Network Interfaces' Hardware Addresses

4.04.01 Getting Network Interface Hardware Address

Many types of Network Interface hardware require the use of a link layer protocol address. In the case for Ethernet, this address is sometimes known as the Hardware Address or MAC Address. In some applications, it may be desirable to obtain the current configured hardware address for a specific interface. This may be performed by calling `NetIF_AddrHW_Get()` with the appropriate arguments.

Listing 4-11 Calling NetIF_AddrHW_Get()

```
NetIF_AddrHW_Get((NET_IF_NBR  ) if_nbr,              (1)
                 (CPU_INT08U *)&addr_hw_sender[0],  (2)
                 (CPU_INT08U *)&addr_hw_len,        (3)
                 (NET_ERR    *) perr);              (4)
```

L4-11(1) The first argument specifies the interface number of which to obtain the hardware address. The interface number is acquired upon the successful addition of the interface.

L4-11(2) The second argument is a pointer to a `CPU_INT08U` array used to provide storage for the returned hardware address. This array MUST be sized large enough to hold the returned number of bytes for the given interfaces' hardware address. The lowest index number in the hardware address array represents the most significant byte of the hardware address.

L4-11(3) The third function is a pointer to a `CPU_INT08U` variable that the function returns the length of the specified interfaces' hardware address.

L4-11(4) The fourth argument is a pointer to a `NET_ERR` variable containing the return error code for the function. If the hardware address is successfully obtained, then the return error code will contain the value `NET_IF_ERR_NONE`.

4.04.02 Setting Network Interface Hardware Address

Some applications prefer to configure the hardware device's hardware address via software during run-time as opposed to a run-time auto-loading EEPROM as is common for many Ethernet hardware devices. If the application would like to set or change the hardware address during run-time, this may be performed by calling `NetIF_AddrHW_Set()` with the appropriate arguments. Alternatively, the hardware address may be statically configured via the device configuration structure and later changed during run-time if desired.

Listing 4-12 Calling NetIF_AddrHW_Set()

```
NetIF_AddrHW_Set((NET_IF_NBR  ) if_nbr,             (1)
                 (CPU_INT08U *)&addr_hw[0],         (2)
                 (CPU_INT08U *)&addr_hw_len,        (3)
                 (NET_ERR    *) perr);              (4)
```

L4-12(1) The first argument specifies the interface number of which to set the hardware address. The interface number is acquired upon the successful addition of the interface.

L4-12(2) The second argument is a pointer to a `CPU_INT08U` array which contains the desired hardware address to set. The lowest index number in the hardware address array represents the most significant byte of the hardware address.

L4-12(3) The third function is a pointer to a `CPU_INT08U` variable that specifies the length of the hardware address being set. In most cases, this can be specified as `size of(addr_hw)` assuming `addr_hw` is declared as an array of `CPU_INT08U`.

L4-12(4) The fourth argument is a pointer to a `NET_ERR` variable containing the return error code for the function. If the hardware address is successfully obtained, then the return error code will contain the value `NET_IF_ERR_NONE`.

NOTE:
In order to set the hardware address for a particular interface, it MUST first be stopped. The hardware address may then be set, and the interface re-started.

4.05 Obtaining Link State

Some applications may wish to obtain the physical link state for a specific interface. Link state information may be obtained by calling `NetIF_IO_Ctrl()` or **`NetIF_LinkStateGet()`** with the appropriate arguments.

Calling `NetIF_IO_Ctrl()` will poll the hardware for the current link state. Alternatively, `NetIF_LinkStateGet()` obtains approximate link state by reading the interface link state flag. Polling the Ethernet hardware for link state takes significantly longer due to the speed and latency of the MII bus. Consequently, it may not be desirable to poll the hardware in a tight loop. Reading the interface flag is fast, but the flag is only periodically updated by the Net IF every 250mS (default) when using the generic Ethernet Phy driver. Phy drivers that implement link state change interrupts may change the value of the interface flag immediately upon link state change detection. In this scenario, calling `NetIF_LinkStateGet()` becomes ideal for those interfaces.

Listing 4-13 Calling NetIF_IO_Ctrl()

```
NetIF_IO_Ctrl((NET_IF_NBR) if_nbr,                               (1)
              (CPU_INT08U) NET_IF_IO_CTRL_LINK_STATE_GET_INFO,(2)
              (void     *)&link_state,                           (3)
              (NET_ERR  *)&err);
(4)
```

L4-13(1) The first argument specifies the interface number of which to obtain the physical link state.

L4-13(2) The second argument specifies the desired function that `NetIF_IO_Ctrl()` will perform. In order to obtain the current interfaces' link state, the application should specify this argument as either:

```
NET_IF_IO_CTRL_LINK_STATE_GET
NET_IF_IO_CTRL_LINK_STATE_GET_INFO
```

L4-13(3) The third argument is a pointer to a link state variable that must be declared by the application and passed to `NetIF_IO_Ctrl()`.

Specifying NET_IF_IO_CTRL_LINK_STATE_GET will require link_state to be declared as a CPU_BOOLEAN which will take on the values NET_IF_LINK_UP or NET_IF_LINK_DOWN.

```
CPU_BOOLEAN             link_state;
```

Alternatively, specifying NET_IF_IO_CTRL_LINK_STATE_GET_INFO will require link_state to be declared as type NET_DEV_LINK_ETHER which is a structure that contains specific link information such as speed and duplex.

```
NET_DEV_LINK_ETHER      link_state;
```

L4-13(4) The fourth argument is a pointer to a NET_ERR variable containing the return error code for the function. If the link state is successfully obtained, then the return error code will contain the value NET_IF_ERR_NONE.

Listing 4-14 Calling NetIF_LinkStateGet()

```
CPU_BOOLEAN    link_state;

link_state = NetIF_LinkStateGet((NET_IF_NBR) if_nbr,(1)
                                (NET_ERR  *)&err);  (2)
```

L4-14(1) The first argument specifies the interface number of which to obtain the physical link state.

L4-14(2) The second argument is a pointer to a NET_ERR variable containing the return error code for the function. If the link state is successfully obtained, then the return error code will contain the value NET_IF_ERR_NONE.

Chapter

5

Network Device Drivers

µC/TCP-IP is able to operate with a variety of network devices. Currently, **µC/TCP-IP** only supports Ethernet type interface controllers but may support serial, PPP, USB, and other popular interfaces in future releases.

There are many Ethernet controllers available on the market and each one requires its own driver to work with **µC/TCP-IP**. The amount of code needed to port a specific device to **µC/TCP-IP** greatly depends on the complexity of the device.

If we do not have a driver for the specific network device you are planning on using, you can write your own driver as described in the *µC/TCP-IP Driver Architecture* document. However, we recommend that you modify an already existing device driver with your device's specific code but following our coding convention for consistency. It's also possible to adapt drivers written for other TCP/IP stacks especially if the drivers short and simply copy data to and from the device.

Chapter

6

Network Socket Interface

Your application interfaces to **µC/TCP-IP** using one of two network socket interfaces. This chapter describes both the **µC/TCP-IP** socket interface defined in the `net_sock.*` and `net_ascii.*` files, as well as the BSD socket interface defined in the `net_bsd.*` files. Even though both socket interfaces are available, the BSD socket interface function calls are converted to their equivalent **µC/TCP-IP** socket interface function calls as shown in Figure 6-1. In addition, the **µC/TCP-IP** socket interface functions are more versatile than their BSD equivalents because they return error codes directly (and re-entrantly) to calling application functions instead of just `0` or `-1`. Fatal socket error codes are described in Section 6.04.01 and a list of all **µC/TCP-IP** socket error code explanations may be found in Appendix B, Section B.07.

Micrium layer 7 application typically use the **µC/TCP-IP** socket interface functions instead of the equivalent BSD functions. Your applications may use either the **µC/TCP-IP** or BSD socket interface functions. However, if you purchase off-the-shelf TCP/IP components (Telnet, Web server, FTP server, etc.) which call BSD socket interface functions, you will need to enable the BSD sockets API (Application Programming Interface) via `NET_BSD_CFG_API_EN` in `net_cfg.h` (see also Section 3.15.12).

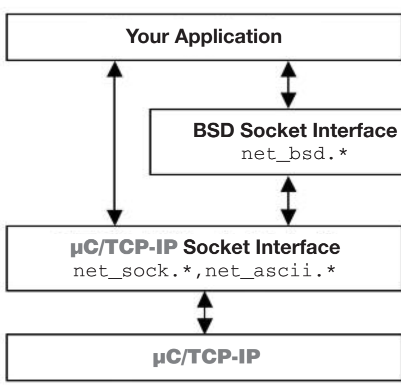

Figure 6-1, Application relationship to µC/TCP-IP Network Socket Interface

6.01 Network Socket Data Structures

Communication using network sockets requires configuring or reading network addresses from network socket address structures. The BSD socket API defines both a generic socket address structure as a blank template with no address-specific configuration … :

Listing 6-1 **Generic (non-address-specific) address structures**

```
struct  sockaddr {                   /* Generic BSD      socket address structure    */
    CPU_INT16U  sa_family;           /*   Socket address family                      */
    CPU_CHAR    sa_data[14];         /*   Protocol-specific address information      */
};

typedef  struct  net_sock_addr {      /* Generic µC/TCP-IP socket address structure   */
    NET_SOCK_ADDR_FAMILY  AddrFamily;
    CPU_INT08U            Addr[NET_SOCK_BSD_ADDR_LEN_MAX = 14];
} NET_SOCK_ADDR;
```

… as well as specific socket address structures to configure each specific protocol address family's network address configuration (e.g. IPv4 socket addresses):

Listing 6-2 Internet (IPv4) address structures

```
struct  in_addr {
    NET_IP_ADDR  s_addr;              /* IPv4 address (32 bits)                        */
};

struct  sockaddr_in {                 /* BSD       IPv4 socket address structure */
        CPU_INT16U  sin_family;       /*   Internet address family (e.g. AF_INET)*/
        CPU_INT16U  sin_port;         /*   Socket   address port number (16 bits)*/
struct  in_addr     sin_addr;         /*   IPv4     address               (32 bits)*/
        CPU_CHAR    sin_zero[8];      /*   Not used (all zeroes)                    */
};

typedef  struct  net_sock_addr_ip {    /* µC/TCP-IP IPv4 socket address structure   */
    NET_SOCK_ADDR_FAMILY  AddrFamily;
    NET_PORT_NBR          Port;
    NET_IP_ADDR           Addr;
    CPU_INT08U            Unused[NET_SOCK_ADDR_IP_NBR_OCTETS_UNUSED = 8];
} NET_SOCK_ADDR_IP;
```

A socket address structure's `AddrFamily/sa_family/sin_family` value **MUST** be read/written in host CPU byte order while all `Addr/sa_data` values **MUST** be read/written in network byte order (big endian).

Even though network socket functions—both **µC/TCP-IP** and BSD—pass pointers to the generic socket address structure, applications **MUST** declare and pass an instance of the specific protocol's socket address structure (e.g. an IPv4 address structure). For microprocessors that require data access to be aligned to appropriate word boundaries, this forces compilers to declare an appropriately-aligned socket address structure so that all socket address members are correctly aligned to their appropriate word boundaries.

CAUTION: Applications should avoid, or be cautious when, declaring and configuring a generic byte array as a socket address structure since the compiler may not correctly align the array to or the socket address structure's members to appropriate word boundaries.

Figure 6-2 shows an example IPv4 instance of the **µC/TCP-IP** `NET_SOCK_ADDR_IP` (`sockaddr_in`) structure overlaid on the `NET_SOCK_ADDR` (`sockaddr`) structure:

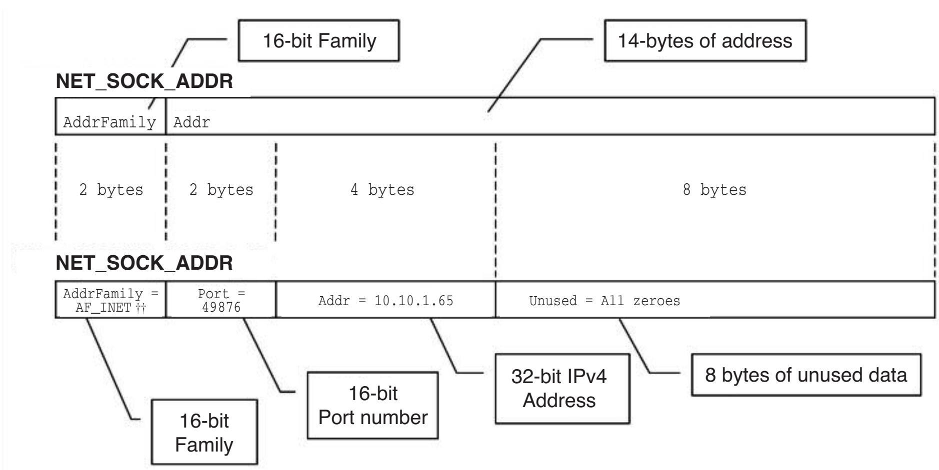

Figure 6-2, NET_SOCK_ADDR_IP is the IPv4 specific instance of the generic NET_SOCK_ADDR data structure

A socket could configure the example socket address structure to bind on IP address `10.10.1.65` and port number `49876` with the following code:

Listing 6-3

```
NET_SOCK_ADDR_IP   addr_local;
NET_IP_ADDR        addr_ip;
NET_PORT_NBR       addr_port;
NET_SOCK_RTN_CODE  rtn_code;
NET_ERR            err;

addr_ip   = NetASCII_Str_to_IP("10.10.1.65", &err);
addr_port = 49876;

Mem_Clr((void     *)&addr_local,
        (CPU_SIZE_T) sizeof(addr_local));
addr_local.AddrFamily = NET_SOCK_ADDR_FAMILY_IP_V4;       /* = AF_INET†† (Fig. 10-2) */
addr_local.Addr       = NET_UTIL_HOST_TO_NET_32(addr_ip);
addr_local.Port       = NET_UTIL_HOST_TO_NET_16(addr_port);

rtn_code = NetSock_Bind((NET_SOCK_ID      ) sock_id,
                        (NET_SOCK_ADDR   *)&addr_local,   /* Cast to generic addr†   */
                        (NET_SOCK_ADDR_LEN) sizeof(addr_local),
                        (NET_ERR         *)&err);
```

† Note the address of the specific IPv4 socket address structure is cast to a pointer to the generic socket address structure.

6.02 Example Socket Applications

6.02.01 Example UDP Socket Application

Figure 6-3 shows a typical UDP client-server application and the typical socket functions used. UDP clients do not establish (dedicated) connections with UDP servers. Instead, UDP clients send request datagrams to UDP servers by specifying the socket address of the servers. A UDP server waits until data arrives from a client, upon which the server processes the client's request and replies back to the client (if necessary). The UDP server then waits for new client requests. Since UDP clients/servers do not establish dedicated connections, each request from each UDP client to the same UDP server are handled independently since there is no state or connection information perserved between UDP client-server requests.

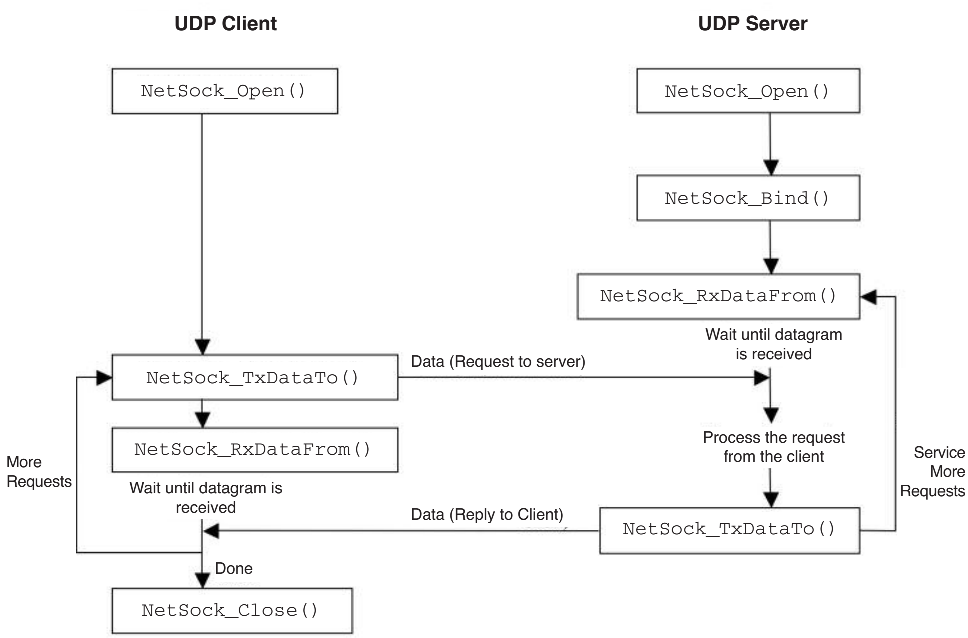

Figure 6-3, µC/TCP-IP Socket calls used in a typical UDP client-server application

6.02.02 Example TCP Socket Application

Figure 6-4 shows a typical TCP client-server application and the typical socket functions used. Typically, after a TCP server starts, TCP clients can connect and send requests to the server. A TCP server waits until client connections arrive and then creates a dedicated TCP socket connection to process the client's requests and reply back to the client (if necessary). This continues until either the client or the server closes the dedicated server-client connection. Also while handling multiple, simultaneous server-client connections, the TCP server can also wait for new server-client connections.

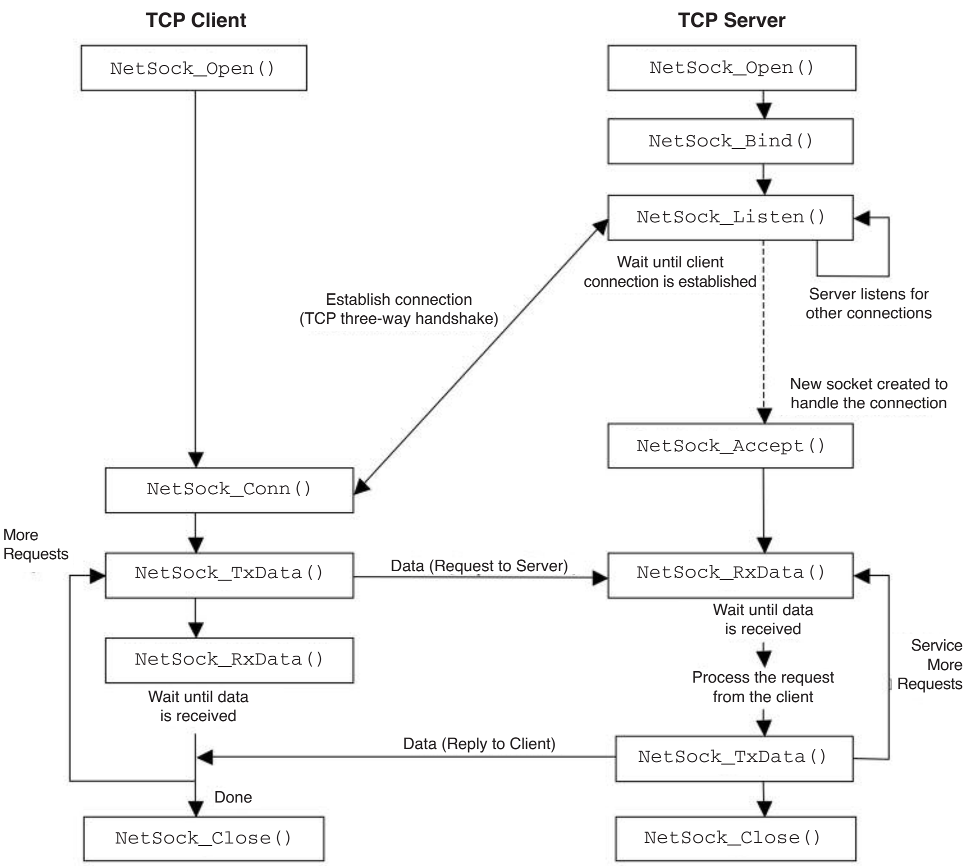

Figure 6-4, μC/TCP-IP Socket calls used in a typical TCP client-server application

6.03 µC/TCP-IP Socket API List

See Section A.12 for a list of all **µC/TCP-IP** socket API functions.

6.04 µC/TCP-IP Socket Error Codes

When socket functions return error codes, the error codes **SHOULD** be inspected to determine if the error is a temporary, non-fault condition (like no data to receive) or fatal (like the socket has been closed).

6.04.01 Fatal Socket Error Codes

Whenever any of the following fatal error codes are returned by any **µC/TCP-IP** socket function, that socket **MUST** be immediately `closed()`'d without further access by any other socket functions:

```
NET_SOCK_ERR_INVALID_FAMILY
NET_SOCK_ERR_INVALID_PROTOCOL
NET_SOCK_ERR_INVALID_TYPE
NET_SOCK_ERR_INVALID_STATE
NET_SOCK_ERR_FAULT
```

Whenever any of the following fatal error codes are returned by any **µC/TCP-IP** socket function, that socket **MUST NOT** be accessed by any other socket functions but must also **NOT** be `closed()`'d:

```
NET_SOCK_ERR_NOT_USED
```

6.04.02 Socket Error Code List

See Section B.07 for a brief explanation of all **µC/TCP-IP** socket error codes.

Chapter

7

Buffer Management

The following sections describe how **µC/TCP-IP** uses buffers to receive and transmit application data as well as network protocol control information.

7.01 Network Buffer Architecture

µC/TCP-IP stores transmitted and received data in data structures known as Network Buffer's. Each Network Buffer consists of two parts: the Network Buffer Header and the Network Buffer Data Area pointer. Network Buffer headers contain information about the data being pointed to via the data area pointer. Data to be received or transmitted is stored in the Network Buffer Data Area. Depending on the configuration, up to eight pools of Network Buffer related objects may be created per network interface. Only four pools are shown below and the remaining pools are used for keeping Network Buffer usage statistics for each of the pools shown.

1) Network Buffers
2) Small Transmit Buffers
3) Large Transmit Buffers
4) Large Receive Buffers

One Network Buffer is allocated for each small transmit, large transmit and large receive buffer. Currently, Network Buffer consume approximately 200 bytes each. We use the term 'data areas' to generically refer to transmit and receive buffers. The Network Buffer is connected to the data area via the Network Buffer Data Area pointer and both move through the Network Protocol Stack layers as a single entity. When the data area is no longer required, both the Network Buffer and the data area are freed. Figure 7-1 depicts the Network Buffer and data area objects.

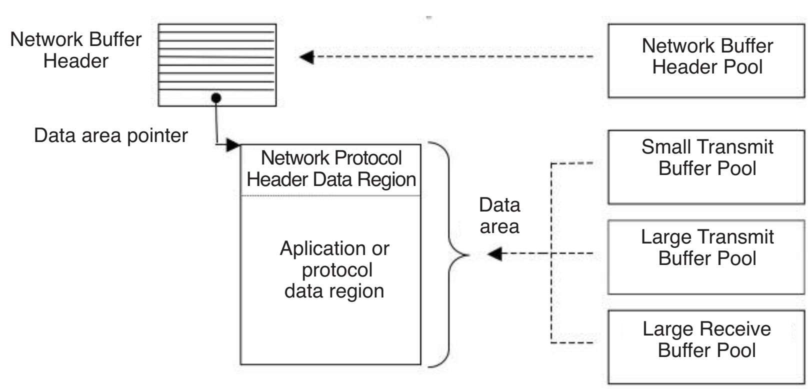

Figure 7-1, Network Buffer Architecture

All transmit data areas contain a small region of reserved space located at the top of the data area address space. The reserved space is used for network protocol header data and is currently fixed to 134 bytes in length. In general, not all of this space is required, however, the network protocol header region has been sized according to the worst case supported network protocol header configuration.

µC/TCP-IP copies application specified data from the application buffer in to the application data region before writing network protocol header data to the protocol header region. Once the application data has been transferred into the Network Buffer Data Area by the highest required **µC/TCP-IP** layer, the Network Buffer descends through the remaining layers where additional protocol headers are added to the network protocol header data region.

Assuming an Ethernet interface (with non jumbo or VLAN tagged frames), the maximum transmit unit is 1500 bytes. When the TCP and IP protocol header sizes are subtracted from this total, only 1460 bytes of application data may be sent in a full sized Ethernet frame. If an Ethernet frame is created such that the frame length is less than 60 bytes, frame padding must be added to the application data area in order to meet the minimum length.

The ARP protocol typically creates frames of 42 bytes and therefore up to 18 bytes of padding must be added. The additional padding must fit within the Network Buffer Data Area.

Therefore, transmit data areas must be configured such that their size is defined as:

```
Minimum (ARP) :   18 + 134 =  152 bytes
Maximum (TCP) : 1460 + 134 = 1594 bytes
Maximum (ICMP): 1480 + 134 = 1614 bytes
```

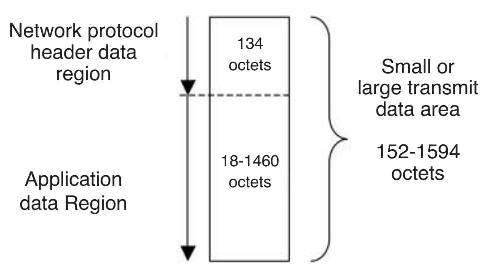

Figure 7-2, Transmit Buffer Data Area

If both small and large transmit data areas are configured, and no small data areas are available, a data area from the large transmit data area pool is allocated.

Receive data areas do not require transmit protocol header reserved data space and therefore may be sized according to the hardware specification or the maximum amount of expected data to be received per frame.

Data area sizes are configured on a per interface basis and must be specified within the device configuration structure specified within the application defined `net_dev_cfg.c` file.

Chapter

8

Timer Management

µC/TCP-IP manages software timers used to keep track of various network-related timeouts. Timer management functions are found in `net_tmr.*`. Timers are required for:

- Device driver and link-layer timeouts
- ARP cache management
- IP fragment reassembly
- Various TCP connection timeouts
- Debug monitor task

See Section 3.05.01 for more information on timer usage and configuration.

Chapter

9

Statistics and Error Counters

µC/TCP-IP maintains counters and statistics for a variety of expected, unexpected, and/or error conditions. Some of these statistics are optional since they require additional code and memory and are enabled only if `NET_CTR_CFG_STAT_EN or NET_CTR_CFG_ERR_EN` is enabled (see Section 3.04).

9.01 Statistics

µC/TCP-IP maintains run-time statistics on interfaces and most **µC/TCP-IP** object pools. If desired, your application can thus query **µC/TCP-IP** to find out how many frames have been processed on a particular interface, transmit and receive performance metrics, buffer utilization and more. Your application can also reset the statistic pools back to their initialization values (`see net_stat.h`).

Applications may choose to monitor statistics for various reasons. For example, examining buffer statistics allows you to better manage your memory usage. Typically, you would allocate more buffers than you 'think' you need and then, by examining buffer usage statistics, you can make adjustments. For example, if you allocate 100 small transmit buffers for a particular interface but your application never uses more than 25, than you might want to consider reducing the number of small transmit buffers to perhaps 30.

Network protocol and interface statistics are kept in a instance of a data structure named `Net_StatCtrs`. This variable may be viewed within a debugger or referenced externally by the application for run-time analysis.

Unlike network protocol statistics, object pool statistics have functions to obtain a copy the specified statistic pool and functions for resetting the pools to their default values. These statistics are kept in a data structure called NET_STAT_POOL which can be declared by the application and used as a return variable from the statistics API functions.

The data structure is shown below:

```
typedef  struct  net_stat_pool {
    NET_TYPE              Type;

    NET_STAT_POOL_QTY     EntriesInit;
    NET_STAT_POOL_QTY     EntriesTotal;
    NET_STAT_POOL_QTY     EntriesAvail;
    NET_STAT_POOL_QTY     EntriesUsed;
    NET_STAT_POOL_QTY     EntriesUsedMax;
    NET_STAT_POOL_QTY     EntriesLostCur;
    NET_STAT_POOL_QTY     EntriesLostTotal;

    CPU_INT32U            EntriesAllocatedCtr;
    CPU_INT32U            EntriesDeallocatedCtr;
} NET_STAT_POOL;
```

NET_STAT_POOL_QTY is a data type currently set to CPU_INT16U and thus can contains a maximum count of 65535.

Access to buffer statistics is obtained via interface functions that your application can call (described in the next sections). Most likely, you will only need to examine the following variables in NET_STAT_POOL:

.EntriesAvail
This variable indicates how many buffers are available in the pool.

.EntriesUsed
This variable indicates how many buffers are currently used by the TCP/IP stack.

.EntriesUsedMax
This variable indicates the maximum number of buffers used since it was last reset.

.EntriesAllocatedCtr
This variable indicates the total number of times buffers were allocated (i.e. used by the TCP/IP stack).

.EntriesDeallocatedCtr
This variable indicates the total number of times buffers were returned back to the buffer pool.

In order to enable run-time statistics, the macro `NET_CTR_CFG_STAT_EN` located within `net_cfg.h` must be defined to `DEF_ENABLED`.

9.02 Error Counters

µC/TCP-IP maintains run-time counters for tracking error conditions within the Network Protocol Stack. If desired, your application may view the error counters in order to debug run-time problems such as low memory conditions, slow performance, packet loss and so forth.

Network protocol error counters are kept in an instance of a data structure named `Net_ErrCtrs`. This variable may be viewed within a debugger or referenced externally by the application for run-time analysis (`see net_stat.h`).

In order to enable run-time error counters, the macro `NET_CTR_CFG_ERR_EN` located within `net_cfg.h` must be defined to `DEF_ENABLED`.

Chapter

10

Debug Management

µC/TCP-IP contains debug constants & functions that may be used by applications to determine network RAM usage, check run-time network resource usage, or check network error or fault conditions. These constants & functions are found in `net_dbg.*`. Most of these debug features must be enabled by appropriate configuration constants (see Chapter 3).

10.01 Network Debug Information Constants

Network debug information constants provide the developer with run-time statistics on **µC/TCP-IP** configuration, data type & structure sizes, & data RAM usage. The list of debug information constants can be found in `net_dbg.c` Sections `GLOBAL NETWORK MODULE DEBUG INFORMATION CONSTANTS` & `GLOBAL NETWORK MODULE DATA SIZE CONSTANTS`. These debug constants are enabled by configuring `NET_DBG_CFG_DBG_INFO_EN` to `DEF_ENABLED`.

For example, you could use these constants as follows:

```
CPU_INT16U  net_version;
CPU_INT32U  net_data_size;
CPU_INT32U  net_data_nbr_if;

net_version     = Net_Version;
net_data_size   = Net_DataSize;
net_data_nbr_if = NetIF_CfgMaxNbrIF;

printf("µC/TCP-IP Version         : %05d\n", net_version);
printf("Total  Network RAM Used   : %05d\n", net_data_size);
printf("Number Network Interfaces : %05d\n", net_data_nbr_if);
```

10.02 Network Debug Monitor Task

The Network Debug Monitor Task periodically checks the current run-time status of certain **µC/TCP-IP** conditions & saves that status to global variables which may be queried by other network modules.

Currently, the Network Debug Monitor Task is only enabled when ICMP Transmit Source Quenches are enabled (see section 3.10.01) because this is the only network functionality that requires a periodic update of certain network status conditions. Applications do not need the Debug Monitor Task functionality since applications have access to the same debug status functions that the Monitor Task calls and may call them asynchronously.

Appendix

A

µC/TCP-IP Application Programming Interface (API) Functions & Macro's

Your application interfaces to **µC/TCP-IP** using any of the functions or macro's described in this appendix.

A.01 General Network Functions

A.01.01 Net_Init()

Initializes **µC/TCP-IP** and MUST be called prior to calling any other **µC/TCP-IP** API functions.

Files

`net.h/net.c`

Prototype

```
NET_ERR  Net_Init(void);
```

Arguments

None.

Returned Value

`NET_ERR_NONE`, if successful;
Specific initialization error code, otherwise.

The return SHOULD be inspected to determine if **µC/TCP-IP** successfully initialized or not. If **µC/TCP-IP** did NOT successfully initialize, search for the returned error code in `net_err.h` and source files to locate where **µC/TCP-IP** initialization failed.

Required Configuration

None.

Notes / Warnings

µC/LIB memory management function `Mem_Init()` MUST be called prior to calling this function.

A.01.02 Net_InitDflt()

Initializes default values for all **µC/TCP-IP** configurable parameters.

Files

net.h/net.c

Prototype

```
void  Net_InitDflt(void);
```

Arguments

None.

Returned Value

None.

Required Configuration

None.

Notes / Warnings

Some default parameters are specified in `net_cfg.h` (see Chapter 3).

A.01.03 Net_VersionGet()

Get the **µC/TCP-IP** software version.

Files

net.h/net.c

Prototype

```
CPU_INT16U  Net_VersionGet(void);
```

Arguments

None.

Returned Value

µC/TCP-IP software version.

Required Configuration

None.

Notes / Warnings

The value returned is multiplied by 100. For example, version 2.01 would be returned as 201.

A.02 Network-to-Application Functions

Not yet supported.

A.03 ARP Functions

A.03.01 NetARP_CacheCalcStat()

Calculate ARP cache found percentage statistics.

Files

net_arp.h/net_arp.c

Prototype

```
CPU_INT08U  NetARP_CacheCalcStat(void);
```

Arguments

None.

Returned Value

ARP cache found percentage, if NO errors;
NULL cache found percentage, otherwise.

Required Configuration

Available only if an appropriate network interface layer is present (e.g. Ethernet; see Section 3.07.04.02).

Notes / Warnings

None.

A.03.02 **NetARP_CacheGetAddrHW()**

Gets the hardware address corresponding to a specific ARP cache's protocol address.

Files

net_arp.h/net_arp.c

Prototype

```
NET_ARP_ADDR_LEN  NetARP_CacheGetAddrHW   (CPU_INT08U        *paddr_hw
                                           NET_ARP_ADDR_LEN   addr_hw_len_buf,
                                           CPU_INT08U        *paddr_protocol,
                                           NET_ARP_ADDR_LEN   addr_protocol_len,
                                           NET_ERR           *perr);
```

Arguments

paddr_hw — Pointer to a memory buffer that will receive the hardware address:

Hardware address that corresponds to the desired protocol address, if NO errors;
Hardware address cleared to all zeros, otherwise.

addr_hw_len_buf — Size of hardware address memory buffer (in bytes).

paddr_protocol — Pointer to the specific protocol address.

addr_protocol_len — Length of protocol address (in bytes).

perr — Pointer to variable that will receive the return error code from this function:

NET_ARP_ERR_NONE
NET_ARP_ERR_NULL_PTR
NET_ARP_ERR_INVALID_HW_ADDR_LEN
NET_ARP_ERR_INVALID_PROTOCOL_ADDR_LEN
NET_ARP_ERR_CACHE_NOT_FOUND
NET_ARP_ERR_CACHE_PEND

Returned Value

Length of returned hardware address, if available;
0, otherwise.

Required Configuration

Available only if an appropriate network interface layer is present (e.g. Ethernet; see Section 3.07.04.02).

Notes / Warnings

`NetARP_CacheGetAddrHW()` may be used in conjunction with `NetARP_ProbeAddrOnNet()` to determine if a specific protocol address is available on the local network.

A.03.03 NetARP_CachePoolStatGet()

Get ARP caches' statistics pool.

Files

net_arp.h/net_arp.c

Prototype

```
NET_STAT_POOL  NetARP_CachePoolStatGet(void);
```

Arguments

None.

Returned Value

ARP caches' statistics pool, if NO errors;
NULL statistics pool, otherwise.

Required Configuration

Available only if an appropriate network interface layer is present (e.g. Ethernet; see Section 3.07.04.02).

Notes / Warnings

None.

A.03.04 NetARP_CachePoolStatResetMaxUsed()

Reset ARP caches' statistics pool's maximum number of entries used.

Files

net_arp.h/net_arp.c

Prototype

```
void  NetARP_CachePoolStatResetMaxUsed(void);
```

Arguments

None.

Returned Value

None.

Required Configuration

Available only if an appropriate network interface layer is present (e.g. Ethernet; see Section 3.07.04.02).

Notes / Warnings

None.

A.03.05 NetARP_CfgCacheAccessedTh()

Configure ARP cache access promotion threshold.

Files

net_arp.h/net_arp.c

Prototype

```
CPU_BOOLEAN  NetARP_CfgCacheAccessedTh(CPU_INT16U  nbr_access);
```

Arguments

nbr_access	Desired number of ARP cache accesses before ARP cache entry is promoted.

Returned Value

DEF_OK,	ARP cache access promotion threshold successfully configured;
DEF_FAIL,	otherwise.

Required Configuration

Available only if an appropriate network interface layer is present (e.g. Ethernet; see Section 3.07.04.02).

Notes / Warnings

None.

A.03.06 NetARP_CfgCacheTimeout()

Configure ARP cache timeout for ARP Cache List. ARP cache entries will be retired if they are not used within the specified timeout.

Files

net_arp.h/net_arp.c

Prototype

```
CPU_BOOLEAN  NetARP_CfgCacheTimeout(CPU_INT16U  timeout_sec);
```

Arguments

timeout_sec	Desired value for ARP cache timeout (in seconds)

Returned Value

DEF_OK,	ARP cache timeout successfully configured;
DEF_FAIL,	otherwise.

Required Configuration

Available only if an appropriate network interface layer is present (e.g. Ethernet; see Section 3.07.04.02).

Notes / Warnings

None.

A.03.07 NetARP_CfgReqMaxRetries()

Configure maximum number of ARP Request retries.

Files

net_arp.h/net_arp.c

Prototype

```
CPU_BOOLEAN  NetARP_CfgReqMaxRetries(CPU_INT08U  max_nbr_retries);
```

Arguments

max_nbr_retries	Desired maximum number of ARP Request retries.

Returned Value

DEF_OK,	maximum number of ARP Request retries configured.
DEF_FAIL,	otherwise.

Required Configuration

Available only if an appropriate network interface layer is present (e.g. Ethernet; see Section 3.07.04.02).

Notes / Warnings

None.

A.03.08 NetARP_CfgReqTimeout()

Configure timeout between ARP request timeouts.

Files

net_arp.h/net_arp.c

Prototype

```
CPU_BOOLEAN  NetARP_CfgReqTimeout(CPU_INT08U  timeout_sec);
```

Arguments

timeout_sec Desired value for ARP request pending ARP Reply timeout (in seconds).

Returned Value

DEF_OK, ARP Request timeout successfully configured,
DEF_FAIL, otherwise.

Required Configuration

Available only if an appropriate network interface layer is present (e.g. Ethernet; see Section 3.07.04.02).

Notes / Warnings

None.

A.03.09 **NetARP_IsAddrProtocolConflict()**

Check interface's protocol address conflict status between this interface's ARP host protocol address(s) and any other host(s) on the local network.

Files

net_arp.h/net_arp.c

Prototype

```
CPU_BOOLEAN  NetARP_IsAddrProtocolConflict (NET_IF_NBR if_nbr,
                                             NET_ERR    *perr);
```

Arguments

`if_nbr` Interface number to get protocol address conflict status.

`perr` Pointer to variable that will receive the return error code from this function:
NET_ARP_ERR_NONE
NET_IF_ERR_INVALID_IF
NET_OS_ERR_LOCK

Returned Value

DEF_YES, if address conflict detected;
DEF_NO, otherwise.

Required Configuration

Available only if an appropriate network interface layer is present (e.g. Ethernet; see Section 3.07.04.02).

Notes / Warnings

None.

A.03.10 NetARP_ProbeAddrOnNet()

Transmit an ARP Request to probe the local network for a specific protocol address.

Files

net_arp.h/net_arp.c

Prototype

```
void  NetARP_ProbeAddrOnNet(NET_PROTOCOL_TYPE   protocol_type,
                            CPU_INT08U         *paddr_protocol_sender,
                            CPU_INT08U         *paddr_protocol_target
                            NET_ARP_ADDR_LEN    addr_protocol_len,
                            NET_ERR            *perr);
```

Arguments

protocol_type — Address protocol type.

paddr_protocol_sender — Pointer to protocol address to send probe from.

paddr_protocol_target — Pointer to protocol address to probe local network.

addr_protocol_len — Length of protocol address (in bytes).

perr — Pointer to variable that will receive the return error code from this function:

NET_ARP_ERR_NONE
NET_ARP_ERR_NULL_PTR
NET_ARP_ERR_INVALID_PROTOCOL_ADDR_LEN
NET_ARP_ERR_CACHE_INVALID_TYPE
NET_ARP_ERR_CACHE_NONE_AVAIL
NET_MGR_ERR_INVALID_PROTOCOL
NET_MGR_ERR_INVALID_PROTOCOL_ADDR
NET_MGR_ERR_INVALID_PROTOCOL_ADDR_LEN
NET_TMR_ERR_NULL_OBJ
NET_TMR_ERR_NULL_FNCT
NET_TMR_ERR_NONE_AVAIL
NET_TMR_ERR_INVALID_TYPE
NET_OS_ERR_LOCK

Returned Value

None.

Required Configuration

Available only if an appropriate network interface layer is present (e.g. Ethernet; see Section 3.07.04.02).

Notes / Warnings

`NetARP_ProbeAddrOnNet()` may be used in conjunction with `NetARP_CacheGetAddrHW()` to determine if a specific protocol address is available on the local network.

A.04 Network ASCII Functions

A.04.01 NetASCII_IP_to_Str()

Converts an IPv4 address in host-order into an IPv4 dotted-decimal notation ASCII string.

Files

net_ascii.h/net_ascii.c

Prototype

```
void   NetASCII_IP_to_Str(NET_IP_ADDR      addr_ip,
                          CPU_CHAR        *paddr_ip_ascii,
                          CPU_BOOLEAN      lead_zeros,
                          NET_ERR         *perr);
```

Arguments

`addr_ip` IPv4 address (in host-order).

`paddr_ip_ascii` Pointer to a memory buffer of size greater than or equal to `NET_ASCII_LEN_MAX_ADDR_IP` bytes to receive the IPv4 address string. Note that the first ASCII character in the string is the most significant nibble of the IP address's most significant byte and that the last character in the string is the least significant nibble of the IP address's least significant byte

Example: `"10.10.1.65" = 0x0A0A0141`

`lead_zeros` Selects formatting the IPv4 address string with leading zeros ('0') prior to the first non-zero digit in each IP address byte. The number of leading zeros added is such that each byte's total number of decimal digits is equal to the maximum number of digits for each byte (i.e. 3).

`DEF_NO` Do **NOT** pre-pend leading zeros to each IP address byte
`DEF_YES` Pre-pend leading zeros to each IP address byte

`perr` Pointer to variable that will receive the return error code from this function:
`NET_ASCII_ERR_NONE`
`NET_ASCII_ERR_NULL_PTR`
`NET_ASCII_ERR_INVALID_CHAR_LEN`

Returned Value

None.

Required Configuration

None.

Notes / Warnings

RFC 1983 states that "dotted decimal notation ... refers [to] IP addresses of the form A.B.C.D; where each letter represents, in decimal, one byte of a four byte IP address". In other words, the dotted-decimal notation separates four decimal byte values by the dot, or period, character ('.'). Each decimal value represents one byte of the IP address starting with the most significant byte in network order.

IPV4 ADDRESS EXAMPLES :

DOTTED DECIMAL NOTATION	HEXADECIMAL EQUIVALENT
`127.0.0.1`	`0x7F000001`
`192.168.1.64`	`0xC0A80140`
`255.255.255.0`	`0xFFFFFF00`
MSB .... LSB	MSB .... LSB

MSB Most Significant Byte in Dotted-Decimal IP Address
LSB Least Significant Byte in Dotted-Decimal IP Address

A.04.02 NetASCII_MAC_to_Str()

Converts a Media Access Control (MAC) address into a hexadecimal address string.

Files

net_ascii.h/net_ascii.c

Prototype

```
void  NetASCII_MAC_to_Str(CPU_INT08U      *paddr_mac,
                          CPU_CHAR        *paddr_mac_ascii,
                          CPU_BOOLEAN      hex_lower_case,
                          CPU_BOOLEAN      hex_colon_sep,
                          NET_ERR         *perr);
```

Arguments

paddr_mac — Pointer to a memory buffer of size NET_ASCII_NBR_OCTET_ADDR_MAC bytes that contains the MAC address.

paddr_mac_ascii — Pointer to a memory buffer of size greater than or equal to NET_ASCII_LEN_MAX_ADDR_MAC bytes to receive the MAC address string. Note that the first ASCII character in the string is the most significant nibble of the MAC address's most significant byte and that the last character in the string is the least significant nibble of the MAC address's least significant address byte.

Example: "00:1A:07:AC:22:09" = 0x001A07AC2209

hex_lower_case — Selects formatting the MAC address string with upper- or lower-case ASCII characters:

DEF_NO	Format MAC address string with upper-case characters
DEF_YES	Format MAC address string with lower-case characters

hex_colon_sep — Selects formatting the MAC address string with colon (':') or dash ('-') characters to separate the MAC address hexadecimal bytes:

DEF_NO	Separate MAC address bytes with hyphen characters
DEF_YES	Separate MAC address bytes with colon characters

`perr`	Pointer to variable that will receive the return error code from this function:
	`NET_ASCII_ERR_NONE`
	`NET_ASCII_ERR_NULL_PTR`

Returned Value

None.

Required Configuration

None.

Notes / Warnings

None.

A.04.03 NetASCII_Str_to_IP()

Converts a string of an IPv4 address in dotted-decimal notation to an IPv4 address in host-order.

Files

net_ascii.h/net_ascii.c

Prototype

```
NET_IP_ADDR  NetASCII_Str_to_IP(CPU_CHAR     *paddr_ip_ascii,
                                NET_ERR      *perr);
```

Arguments

paddr_ip_ascii	Pointer to an ASCII string that contains a dotted-decimal IPv4 address. Each decimal byte of the IPv4 address string must be separated by the dot, or period, character ('.'). Note that the first ASCII character in the string is the most significant nibble of the IP address's most significant byte and that the last character in the string is the least significant nibble of the IP address's least significant byte. Example: "10.10.1.65" = 0x0A0A0141
perr	Pointer to variable that will receive the return error code from this function: NET_ASCII_ERR_NONE NET_ASCII_ERR_NULL_PTR NET_ASCII_ERR_INVALID_STR_LEN NET_ASCII_ERR_INVALID_CHAR NET_ASCII_ERR_INVALID_CHAR_LEN NET_ASCII_ERR_INVALID_CHAR_VAL NET_ASCII_ERR_INVALID_CHAR_SEQ

Returned Value

Returns the IPv4 address, represented by the IPv3 address string, in host-order, if **NO** errors; NET_IP_ADDR_NONE, otherwise.

Required Configuration

None.

Notes / Warnings

RFC 1983 states that "dotted decimal notation ... refers [to] IP addresses of the form A.B.C.D; where each letter represents, in decimal, one byte of a four byte IP address". In other words, the dotted-decimal notation separates four decimal byte values by the dot, or period, character ('.'). Each decimal value represents one byte of the IP address starting with the most significant byte in network order.

IPv4 Address Examples :

DOTTED DECIMAL NOTATION	HEXADECIMAL EQUIVALENT
127.0.0.1	0x7F000001
192.168.1.64	0xC0A80140
255.255.255.0	0xFFFFFF00
MSB .... LSB	MSB .... LSB

MSB Most Significant Byte in Dotted-Decimal IP Address
LSB Least Significant Byte in Dotted-Decimal IP Address

The IPv4 dotted-decimal ASCII string **MUST** include **ONLY** decimal values and the dot, or period, character ('.'); **ALL** other characters are trapped as invalid, including any leading or trailing characters. The ASCII string **MUST** include exactly four decimal values separated by exactly three dot characters. Each decimal value **MUST NOT** exceed the maximum byte value (i.e. 255), or exceed the maximum number of digits for each byte (i.e. 3) including any leading zeros.

A.04.04 NetASCII_Str_to_MAC()

Converts a hexadecimal address string to a Media Access Control (MAC) address.

Files

net_ascii.h/net_ascii.c

Prototype

```
void  NetASCII_Str_to_MAC(CPU_CHAR     *paddr_mac_ascii,
                          CPU_INT08U   *paddr_mac,
                          NET_ERR      *perr);
```

Arguments

paddr_mac_ascii Pointer to an ASCII string that contains hexadecimal bytes separated by colons or dashes that represents the MAC address. Each hexadecimal byte of the MAC address string must be separated by either the colon (‘:’) or dash (‘-’) characters. Note that the first ASCII character in the string is the most significant nibble of the MAC address’s most significant byte and that the last character in the string is the least significant nibble of the MAC address’s least significant address byte.

Example: "00:1A:07:AC:22:09" = 0x001A07AC2209

paddr_mac Pointer to a memory buffer of size greater than or equal to NET_ASCII_NBR_OCTET_ADDR_MAC bytes to receive the MAC address.

perr Pointer to variable that will receive the return error code from this function:

NET_ASCII_ERR_NONE
NET_ASCII_ERR_NULL_PTR
NET_ASCII_ERR_INVALID_STR_LEN
NET_ASCII_ERR_INVALID_CHAR
NET_ASCII_ERR_INVALID_CHAR_LEN
NET_ASCII_ERR_INVALID_CHAR_SEQ

Returned Value

None.

Required Configuration

None.

Notes / Warnings

None.

A.05 Network Buffer Functions

A.05.01 NetBuf_PoolStatGet()

Get an interface's Network Buffers' statistics pool.

Files

net_buf.h/net_buf.c

Prototype

```
NET_STAT_POOL  NetBuf_PoolStatGet(NET_IF_NBR  if_nbr);
```

Arguments

`if_nbr` Interface number to get Network Buffer statistics.

Returned Value

Network Buffers' statistics pool, if NO errors;
`NULL` statistics pool, otherwise.

Required Configuration

None.

Notes / Warnings

None.

A.05.02 NetBuf_PoolStatResetMaxUsed()

Reset an interface's Network Buffers' statistics pool's maximum number of entries used.

Files

net_buf.h/net_buf.c

Prototype

```
void  NetBuf_PoolStatResetMaxUsed(NET_IF_NBR  if_nbr);
```

Arguments

if_nbr	Interface number to reset Network Buffer statistics.

Returned Value

None.

Required Configuration

None.

Notes / Warnings

None.

A.05.03 NetBuf_RxLargePoolStatGet()

Get an interface's large receive buffers' statistics pool.

Files

net_buf.h/net_buf.c

Prototype

```
NET_STAT_POOL  NetBuf_RxLargePoolStatGet(NET_IF_NBR  if_nbr);
```

Arguments

`if_nbr` Interface number to get Network Buffer statistics.

Returned Value

Large receive buffers' statistics pool, if NO errors;
`NULL` statistics pool, otherwise.

Required Configuration

None.

Notes / Warnings

None.

A.05.04 NetBuf_RxLargePoolStatResetMaxUsed()

Reset an interface's large receive buffers' statistics pool's maximum number of entries used.

Files

net_buf.h/net_buf.c

Prototype

```
void  NetBuf_RxLargePoolStatResetMaxUsed(NET_IF_NBR  if_nbr);
```

Arguments

if_nbr	Interface number to reset Network Buffer statistics.

Returned Value

None.

Required Configuration

None.

Notes / Warnings

None.

A.05.05 NetBuf_TxLargePoolStatGet()

Get an interface's large transmit buffers' statistics pool.

Files

net_buf.h/net_buf.c

Prototype

```
NET_STAT_POOL  NetBuf_TxLargePoolStatGet(NET_IF_NBR  if_nbr);
```

Arguments

if_nbr	Interface number to get Network Buffer statistics.

Returned Value

Large transmit buffers' statistics pool, if NO errors;
NULL statistics pool, otherwise.

Required Configuration

None.

Notes / Warnings

None.

A.05.06 NetBuf_TxLargePoolStatResetMaxUsed()

Reset an interface's large transmit buffers' statistics pool's maximum number of entries used.

Files

`net_buf.h/net_buf.c`

Prototype

```
void  NetBuf_TxLargePoolStatResetMaxUsed(NET_IF_NBR  if_nbr);
```

Arguments

`if_nbr` Interface number to reset Network Buffer statistics.

Returned Value

None.

Required Configuration

None.

Notes / Warnings

None.

A.05.07 NetBuf_TxSmallPoolStatGet()

Get an interface's small transmit buffers' statistics pool.

Files

net_buf.h/net_buf.c

Prototype

```
NET_STAT_POOL  NetBuf_TxSmallPoolStatGet(NET_IF_NBR  if_nbr);
```

Arguments

`if_nbr` Interface number to get Network Buffer statistics.

Returned Value

Small transmit buffers' statistics pool, if NO errors;
`NULL` statistics pool, otherwise.

Required Configuration

None.

Notes / Warnings

None.

A.05.08 NetBuf_TxSmallPoolStatResetMaxUsed()

Reset an interface's small transmit buffers' statistics pool's maximum number of entries used.

Files

net_buf.h/net_buf.c

Prototype

```
void  NetBuf_TxSmallPoolStatResetMaxUsed(NET_IF_NBR  if_nbr);
```

Arguments

`if_nbr` Interface number to reset Network Buffer statistics.

Returned Value

None.

Required Configuration

None.

Notes / Warnings

None.

A.06 Network Connection Functions

A.06.01 NetConn_CfgAccessedTh()

Configure network connection access promotion threshold.

Files

net_conn.h/net_conn.c

Prototype

```
CPU_BOOLEAN  NetConn_CfgAccessedTh(CPU_INT16U  nbr_access);
```

Arguments

nbr_access	Desired number of accesses before network connection is promoted.

Returned Value

DEF_OK,	network connection access promotion threshold configured.
DEF_FAIL,	otherwise.

Required Configuration

Available only if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01).

Notes / Warnings

None.

A.06.02 NetConn_PoolStatGet()

Get Network Connections' statistics pool.

Files

net_conn.h/net_conn.c

Prototype

```
NET_STAT_POOL  NetConn_PoolStatGet(void);
```

Arguments

None.

Returned Value

Network Connections' statistics pool, if NO errors;
NULL statistics pool, otherwise.

Required Configuration

Available only if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01).

Notes / Warnings

None.

A.06.03 NetConn_PoolStatResetMaxUsed()

Reset Network Connections' statistics pool's maximum number of entries used.

Files

net_conn.h/net_conn.c

Prototype

```
void  NetConn_PoolStatResetMaxUsed(void);
```

Arguments

None.

Returned Value

None.

Required Configuration

Available only if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01).

Notes / Warnings

None.

A.07 Network Debug Functions

A.07.01 NetDbg_CfgMonTaskTime()

Configure Network Debug Monitor time.

Files

net_dbg.h/net_dbg.c

Prototype

```
CPU_BOOLEAN  NetDbg_CfgMonTaskTime(CPU_INT16U  time_sec);
```

Arguments

time_sec	Desired value for Network Debug Monitor Task time (in seconds).

Returned Value

DEF_OK,	Network Debug Monitor Task time successfully configured.
DEF_FAIL,	otherwise.

Required Configuration

Available only if the Network Debug Monitor Task is enabled (see Section 10.02).

Notes / Warnings

None.

A.07.02 NetDbg_CfgRsrcARP_CacheThLo()

Configure ARP caches' low resource threshold.

Files

net_dbg.h/net_dbg.c

Prototype

```
CPU_BOOLEAN  NetDbg_CfgRsrcARP_CacheThLo(CPU_INT08U  th_pct,
                                         CPU_INT08U  hyst_pct);
```

Arguments

th_pct	Desired percentage of ARP caches available to trip low resources.
hyst_pct	Desired percentage of ARP caches freed to clear low resources.

Returned Value

DEF_OK	ARP caches' low resource threshold successfully configured.
DEF_FAIL,	otherwise.

Required Configuration

Available only if NET_DBG_CFG_DBG_STATUS_EN is enabled (see Section 3.02.02) and/or if the Network Debug Monitor Task is enabled (see Section 10.02) AND if an appropriate network interface layer is present (e.g. Ethernet; see Section 3.07.04.02).

Notes / Warnings

None.

A.07.03 **NetDbg_CfgRsrcBufThLo()**

Configure an interface's network buffers' low resource threshold.

Files

net_dbg.h/net_dbg.c

Prototype

```
CPU_BOOLEAN  NetDbg_CfgRsrcBufThLo(NET_IF_NBR   if_nbr,
                                   CPU_INT08U   th_pct,
                                   CPU_INT08U   hyst_pct);
```

Arguments

if_nbr	Interface number to configure low threshold & hysteresis.
th_pct	Desired percentage of network buffers available to trip low resources.
hyst_pct	Desired percentage of network buffers freed to clear low resources.

Returned Value

DEF_OK,	Network buffers' low resource threshold successfully configured.
DEF_FAIL,	otherwise.

Required Configuration

Available only if NET_DBG_CFG_DBG_STATUS_EN is enabled (see Section 3.02.02) and/or if the Network Debug Monitor Task is enabled (see Section 10.02).

Notes / Warnings

None.

A.07.04 **NetDbg_CfgRsrcBufRxLargeThLo()**

Configure an interface's large receive buffers' low resource threshold.

Files

net_dbg.h/net_dbg.c

Prototype

```
CPU_BOOLEAN  NetDbg_CfgRsrcBufRxLargeThLo(NET_IF_NBR  if_nbr,
                                          CPU_INT08U  th_pct,
                                          CPU_INT08U  hyst_pct);
```

Arguments

if_nbr	Interface number to configure low threshold & hysteresis.
th_pct	Desired percentage of large receive buffers available to trip low resources.
hyst_pct	Desired percentage of large receive buffers freed to clear low resources.

Returned Value

DEF_OK,	Large receive buffers' low resource threshold successfully configured.
DEF_FAIL,	otherwise.

Required Configuration

Available only if NET_DBG_CFG_DBG_STATUS_EN is enabled (see Section 3.02.02) and/or if the Network Debug Monitor Task is enabled (see Section 10.02).

Notes / Warnings

None.

A.07.05 NetDbg_CfgRsrcBufTxLargeThLo()

Configure an interface's large transmit buffers' low resource threshold.

Files

net_dbg.h/net_dbg.c

Prototype

```
CPU_BOOLEAN  NetDbg_CfgRsrcBufTxLargeThLo(NET_IF_NBR  if_nbr,
                                          CPU_INT08U  th_pct,
                                          CPU_INT08U  hyst_pct);
```

Arguments

if_nbr	Interface number to configure low threshold & hysteresis.
th_pct	Desired percentage of large transmit buffers available to trip low resources.
hyst_pct	Desired percentage of large transmit buffers freed to clear low resources.

Returned Value

DEF_OK,	Large transmit buffers' low resource threshold successfully configured.
DEF_FAIL,	otherwise.

Required Configuration

Available only if NET_DBG_CFG_DBG_STATUS_EN is enabled (see Section 3.02.02) and/or if the Network Debug Monitor Task is enabled (see Section 10.02).

Notes / Warnings

None.

A.07.06 NetDbg_CfgRsrcBufTxSmallThLo()

Configure an interface's small transmit buffers' low resource threshold.

Files

net_dbg.h/net_dbg.c

Prototype

```
CPU_BOOLEAN  NetDbg_CfgRsrcBufTxSmallThLo(NET_IF_NBR  if_nbr,
                                          CPU_INT08U  th_pct,
                                          CPU_INT08U  hyst_pct);
```

Arguments

if_nbr	Interface number to configure low threshold & hysteresis.
th_pct	Desired percentage of small transmit buffers available to trip low resources.
hyst_pct	Desired percentage of small transmit buffers freed to clear low resources.

Returned Value

DEF_OK,	Small transmit buffers' low resource threshold successfully configured.
DEF_FAIL,	otherwise.

Required Configuration

Available only if NET_DBG_CFG_DBG_STATUS_EN is enabled (see Section 3.02.02) and/or if the Network Debug Monitor Task is enabled (see Section 10.02).

Notes / Warnings

None.

A.07.07 NetDbg_CfgRsrcConnThLo()

Configure network connections' low resource threshold.

Files

net_dbg.h/net_dbg.c

Prototype

```
CPU_BOOLEAN  NetDbg_CfgRsrcConnThLo(CPU_INT08U  th_pct,
                                    CPU_INT08U  hyst_pct);
```

Arguments

th_pct	Desired percentage of network connections available to trip low resources.
hyst_pct	Desired percentage of network connections freed to clear low resources.

Returned Value

DEF_OK,	Network connections' low resource threshold successfully configured.
DEF_FAIL,	otherwise.

Required Configuration

Available only if NET_DBG_CFG_DBG_STATUS_EN is enabled (see Section 3.02.02) and/or if the Network Debug Monitor Task is enabled (see Section 10.02) AND if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01).

Notes / Warnings

None.

A.07.08 NetDbg_CfgRsrcSockThLo()

Configure network sockets' low resource threshold.

Files

net_dbg.h/net_dbg.c

Prototype

```
CPU_BOOLEAN  NetDbg_CfgRsrcSockThLo(CPU_INT08U  th_pct,
                                    CPU_INT08U  hyst_pct);
```

Arguments

th_pct	Desired percentage of network sockets available to trip low resources.
hyst_pct	Desired percentage of network sockets freed to clear low resources.

Returned Value

DEF_OK,	Network sockets' low resource threshold successfully configured.
DEF_FAIL,	otherwise.

Required Configuration

Available only if NET_DBG_CFG_DBG_STATUS_EN is enabled (see Section 3.02.02) and/or if the Network Debug Monitor Task is enabled (see Section 10.02) AND if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01).

Notes / Warnings

None.

A.07.09 NetDbg_CfgRsrcTCP_ConnThLo()

Configure TCP connections' low resource threshold.

Files

net_dbg.h/net_dbg.c

Prototype

```
CPU_BOOLEAN  NetDbg_CfgRsrcTCP_ConnThLo(CPU_INT08U  th_pct,
                                        CPU_INT08U  hyst_pct);
```

Arguments

th_pct	Desired percentage of TCP connections available to trip low resources.
hyst_pct	Desired percentage of TCP connections freed to clear low resources.

Returned Value

DEF_OK,	TCP connections' low resource threshold successfully configured.
DEF_FAIL,	otherwise.

Required Configuration

Available only if NET_DBG_CFG_DBG_STATUS_EN is enabled (see Section 3.02.02) and/or if the Network Debug Monitor Task is enabled (see Section 10.02) AND if NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12).

Notes / Warnings

None.

A.07.10 NetDbg_CfgRsrcTmrThLo()

Configure network timers' low resource threshold.

Files

net_dbg.h/net_dbg.c

Prototype

```
CPU_BOOLEAN  NetDbg_CfgRsrcTmrThLo(CPU_INT08U  th_pct,
                                   CPU_INT08U  hyst_pct);
```

Arguments

th_pct	Desired percentage of network timers available to trip low resources.
hyst_pct	Desired percentage of network timers freed to clear low resources.

Returned Value

DEF_OK	Network timers' low resource threshold successfully configured.
DEF_FAIL,	otherwise.

Required Configuration

Available only if NET_DBG_CFG_DBG_STATUS_EN is enabled (see Section 3.02.02) and/or if the Network Debug Monitor Task is enabled (see Section 10.02).

Notes / Warnings

None.

A.07.11 NetDbg_ChkStatus()

Return the current run-time status of certain **µC/TCP-IP** conditions.

Files

net_dbg.h/net_dbg.c

Prototype

```
NET_DBG_STATUS  NetDbg_ChkStatus(void);
```

Arguments

None.

Returned Value

NET_DBG_STATUS_OK, if all network conditions are OK (i.e. no warnings, faults, or errors currently exist); Otherwise, returns the following status condition codes logically OR'd:

NET_DBG_STATUS_FAULT .... Some network status fault(s)
NET_DBG_STATUS_RSRC_LOST .... Some network resources lost.
NET_DBG_STATUS_RSRC_LO .... Some network resources low.
NET_DBG_STATUS_FAULT_BUF .... Some network buffer management fault(s).
NET_DBG_STATUS_FAULT_TMR .... Some network timer management fault(s).
NET_DBG_STATUS_FAULT_CONN .... Some network connection management fault(s).
NET_DBG_STATUS_FAULT_TCP .... Some TCP layer fault(s).

Required Configuration

Available only if NET_DBG_CFG_DBG_STATUS_EN is enabled (see Section 3.02.02).

Notes / Warnings

None.

A.07.12 NetDbg_ChkStatusBufs()

Return the current run-time status of **µC/TCP-IP** network buffers.

Files

`net_dbg.h/net_dbg.c`

Prototype

```
NET_DBG_STATUS  NetDbg_ChkStatusBufs(void);
```

Arguments

None.

Returned Value

`NET_DBG_STATUS_OK`, if all network buffer conditions are OK (i.e. no warnings, faults, or errors currently exist); Otherwise, returns the following status condition codes logically OR'd:

`NET_DBG_SF_BUF` .... Some Network Buffer management fault(s).

Required Configuration

Available only if `NET_DBG_CFG_DBG_STATUS_EN` is enabled (see Section 3.02.02).

Notes / Warnings

Debug status information for network buffers has been deprecated in **µC/TCP-IP**.

A.07.13 NetDbg_ChkStatusConns()

Return the current run-time status of **µC/TCP-IP** network connections.

Files

`net_dbg.h/net_dbg.c`

Prototype

```
NET_DBG_STATUS  NetDbg_ChkStatusConns(void);
```

Arguments

None.

Returned Value

`NET_DBG_STATUS_OK`, if all network connection conditions are OK (i.e. no warnings, faults, or errors currently exist); Otherwise, returns the following status condition codes logically OR'd:

`NET_DBG_SF_CONN`	Some network connection management fault(s).
`NET_DBG_SF_CONN_TYPE`	Network connection invalid type.
`NET_DBG_SF_CONN_FAMILY`	Network connection invalid family.
`NET_DBG_SF_CONN_PROTOCOL_IX_NBR_MAX`	Network connection invalid protocol list index number.
`NET_DBG_SF_CONN_ID`	Network connection invalid ID.
`NET_DBG_SF_CONN_ID_NONE`	Network connection with NO connection IDs.
`NET_DBG_SF_CONN_ID_UNUSED`	Network connection linked to unused connection.
`NET_DBG_SF_CONN_LINK_TYPE`	Network connection invalid link type.
`NET_DBG_SF_CONN_LINK_UNUSED`	Network connection link unused.
`NET_DBG_SF_CONN_LINK_BACK_TO_CONN`	Network connection invalid link back to same connection.
`NET_DBG_SF_CONN_LINK_NOT_TO_CONN`	Network connection invalid link NOT back to same connection.
`NET_DBG_SF_CONN_LINK_NOT_IN_LIST`	Network connection NOT in appropriate connection list.
`NET_DBG_SF_CONN_POOL_TYPE`	Network connection invalid pool type.
`NET_DBG_SF_CONN_POOL_ID`	Network connection invalid pool id.
`NET_DBG_SF_CONN_POOL_DUP`	Network connection pool contains duplicate connection(s).

NET_DBG_SF_CONN_POOL_NBR_MAX	Network connection pool number of connections greater than maximum number of connections.
NET_DBG_SF_CONN_LIST_NBR_NOT_SOLITARY	Network connection lists number of connections NOT equal to solitary connection.
NET_DBG_SF_CONN_USED_IN_POOL	Network connection used but in pool.
NET_DBG_SF_CONN_USED_NOT_IN_LIST	Network connection used but NOT in list.
NET_DBG_SF_CONN_UNUSED_IN_LIST	Network connection unused but in list.
NET_DBG_SF_CONN_UNUSED_NOT_IN_POOL	Network connection unused but NOT in pool.
NET_DBG_SF_CONN_IN_LIST_IN_POOL	Network connection in list & in pool.
NET_DBG_SF_CONN_NOT_IN_LIST_NOT_IN_POOL	Network connection NOT in list & NOT in pool.

Required Configuration

Available only if NET_DBG_CFG_DBG_STATUS_EN is enabled (see Section 3.02.02) AND if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01).

Notes / Warnings

None.

A.07.14 NetDbg_ChkStatusRsrcLost() / NetDbg_MonTaskStatusGetRsrcLost()

Return whether any **µC/TCP-IP** resources are currently lost.

Files

net_dbg.h/net_dbg.c

Prototypes

```
NET_DBG_STATUS  NetDbg_ChkStatusRsrcLost(void);
NET_DBG_STATUS  NetDbg_MonTaskStatusGetRsrcLost(void);
```

Arguments

None.

Returned Value

NET_DBG_STATUS_OK, if NO network resources are lost; Otherwise, returns the following status condition codes logically OR'd:

NET_DBG_SF_RSRC_LOST	Some network	resources lost.
NET_DBG_SF_RSRC_LOST_BUF_SMALL	Some network SMALL buffer	resources lost.
NET_DBG_SF_RSRC_LOST_BUF_LARGE	Some network LARGE buffer	resources lost.
NET_DBG_SF_RSRC_LOST_TMR	Some network timer	resources lost.
NET_DBG_SF_RSRC_LOST_CONN	Some network connection	resources lost.
NET_DBG_SF_RSRC_LOST_ARP_CACHE	Some network ARP cache	resources lost.
NET_DBG_SF_RSRC_LOST_TCP_CONN	Some network TCP connection	resources lost.
NET_DBG_SF_RSRC_LOST_SOCK	Some network socket	resourceslost.

Required Configuration

NetDbg_ChkStatusRsrcLost() available only if NET_DBG_CFG_DBG_STATUS_EN is enabled (see Section 3.02.02). NetDbg_MonTaskStatusGetRsrcLost() available only if the Network Debug Monitor Task is enabled (see Section 10.02).

Notes / Warnings

NetDbg_ChkStatusRsrcLost() checks network conditions lost status inline, whereas NetDbg_MonTaskStatusGetRsrcLost() checks the Network Debug Monitor Task's last known lost status.

A.07.15 NetDbg_ChkStatusRsrcLo() / NetDbg_MonTaskStatusGetRsrcLo()

Return whether any **µC/TCP-IP** resources are currently low.

Files

net_dbg.h/net_dbg.c

Prototypes

```
NET_DBG_STATUS  NetDbg_ChkStatusRsrcLo(void);
NET_DBG_STATUS  NetDbg_MonTaskStatusGetRsrcLo(void);
```

Arguments

None.

Returned Value

NET_DBG_STATUS_OK, if NO network resources are low;
Otherwise, returns the following status condition codes logically OR'd:

NET_DBG_SF_RSRC_LO	Some network	resources low.
NET_DBG_SF_RSRC_LO_BUF_SMALL	Network SMALL buffer	resources low
NET_DBG_SF_RSRC_LO_BUF_LARGE	Network LARGE buffer	resources low.
NET_DBG_SF_RSRC_LO_TMR	Network timer	resources low.
NET_DBG_SF_RSRC_LO_CONN	Network connection	resources low.
NET_DBG_SF_RSRC_LO_ARP_CACHE	Network ARP cache	resources low.
NET_DBG_SF_RSRC_LO_TCP_CONN	Network TCP connection	resources low.
NET_DBG_SF_RSRC_LO_SOCK	Network socket	resources low.

Required Configuration

NetDbg_ChkStatusRsrcLo() available only if NET_DBG_CFG_DBG_STATUS_EN is enabled (see Section 3.02.02). NetDbg_MonTaskStatusGetRsrcLo() available only if the Network Debug Monitor Task is enabled (see Section 10.02).

Notes / Warnings

NetDbg_ChkStatusRsrcLo() checks network conditions low status inline, whereas NetDbg_MonTaskStatusGetRsrcLo() checks the Network Debug Monitor Task's last known low status.

A.07.16 NetDbg_ChkStatusTCP()

Return the current run-time status of **µC/TCP-IP** TCP connections.

Files

net_dbg.h/net_dbg.c

Prototype

```
NET_DBG_STATUS  NetDbg_ChkStatusTCP(void);
```

Arguments

None.

Returned Value

NET_DBG_STATUS_OK, if all TCP layer conditions are OK (i.e. no warnings, faults, or errors currently exist); Otherwise, returns the following status condition codes logically OR'd:

NET_DBG_SF_TCP	Some TCP layer fault(s).
NET_DBG_SF_TCP_CONN_TYPE	TCP connection invalid type.
NET_DBG_SF_TCP_CONN_ID	TCP connection invalid id.
NET_DBG_SF_TCP_CONN_LINK_TYPE	TCP connection invalid link type.
NET_DBG_SF_TCP_CONN_LINK_UNUSED	TCP connection link unused.
NET_DBG_SF_TCP_CONN_POOL_TYPE	TCP connection invalid pool type.
NET_DBG_SF_TCP_CONN_POOL_ID	TCP connection invalid pool id.
NET_DBG_SF_TCP_CONN_POOL_DUP	TCP connection pool contains duplicate connection(s).
NET_DBG_SF_TCP_CONN_POOL_NBR_MAX	TCP connection pool number of connections greater than maximum number of connections.
NET_DBG_SF_TCP_CONN_USED_IN_POOL	TCP connection used in pool.
NET_DBG_SF_TCP_CONN_UNUSED_NOT_IN_POOL	TCP connection unused NOT in pool.
NET_DBG_SF_TCP_CONN_Q	Some TCP connection queue fault(s).
NET_DBG_SF_TCP_CONN_Q_BUF_TYPE	TCP connection queue buffer invalid type.
NET_DBG_SF_TCP_CONN_Q_BUF_UNUSED	TCP connection queue buffer unused.
NET_DBG_SF_TCP_CONN_Q_LINK_TYPE	TCP connection queue buffer invalid link type.
NET_DBG_SF_TCP_CONN_Q_LINK_UNUSED	TCP connection queue buffer link unused.
NET_DBG_SF_TCP_CONN_Q_BUF_DUP	TCP connection queue contains duplicate buffer(s).

Required Configuration

Available only if `NET_DBG_CFG_DBG_STATUS_EN` is enabled (see Section 3.02.02) AND if `NET_CFG_TRANSPORT_LAYER_SEL` is configured for TCP (see Section 3.12).

Notes / Warnings

None.

A.07.17 NetDbg_ChkStatusTmrs()

Return the current run-time status of **µC/TCP-IP** network timers.

Files

`net_dbg.h/net_dbg.c`

Prototype

```
NET_DBG_STATUS  NetDbg_ChkStatusTmrs(void);
```

Arguments

None.

Returned Value

`NET_DBG_STATUS_OK`, if all network timer conditions are OK (i.e. no warnings, faults, or errors currently exist); Otherwise, returns the following status condition codes logically OR'd:

`NET_DBG_SF_TMR`	Some network timer management fault(s).
`NET_DBG_SF_TMR_TYPE`	Network timer invalid type.
`NET_DBG_SF_TMR_ID`	Network timer invalid id.
`NET_DBG_SF_TMR_LINK_TYPE`	Network timer invalid link type.
`NET_DBG_SF_TMR_LINK_UNUSED`	Network timer link unused.
`NET_DBG_SF_TMR_LINK_BACK_TO_TMR`	Network timer invalid link back to same timer.
`NET_DBG_SF_TMR_LINK_TO_TMR`	Network timer invalid link back to timer.
`NET_DBG_SF_TMR_POOL_TYPE`	Network timer invalid pool type.
`NET_DBG_SF_TMR_POOL_ID`	Network timer invalid pool id.
`NET_DBG_SF_TMR_POOL_DUP`	Network timer pool contains duplicate timer(s).
`NET_DBG_SF_TMR_POOL_NBR_MAX`	Network timer pool number of timers greater than maximum number of timers.
`NET_DBG_SF_TMR_LIST_TYPE`	Network Timer Task list invalid type.
`NET_DBG_SF_TMR_LIST_ID`	Network Timer Task list invalid id.
`NET_DBG_SF_TMR_LIST_DUP`	Network Timer Task list contains duplicate timer(s).
`NET_DBG_SF_TMR_LIST_NBR_MAX`	Network Timer Task list number of timers greater than maximum number of timers.
`NET_DBG_SF_TMR_LIST_NBR_USED`	Network Timer Task list number of timers NOT equal to number of used timers.
`NET_DBG_SF_TMR_USED_IN_POOL`	Network timer used but in pool.

NET_DBG_SF_TMR_UNUSED_NOT_IN_POOL . . Network timer unused but NOT in pool.
NET_DBG_SF_TMR_UNUSED_IN_LIST. . . . . . . Network timer unused but in Timer Task list.

Required Configuration

Available only if NET_DBG_CFG_DBG_STATUS_EN is enabled (see Section 3.02.02).

Notes / Warnings

None.

A.07.18 NetDbg_MonTaskStatusGetRsrcLost()

Return whether any **µC/TCP-IP** resources are currently lost.

See Section A.07.14 for more information.

Files

net_dbg.h/net_dbg.c

Prototype

```
NET_DBG_STATUS  NetDbg_MonTaskStatusGetRsrcLost(void);
```

A.07.19 NetDbg_MonTaskStatusGetRsrcLo()

Return whether any **µC/TCP-IP** resources are currently low.

See Section A.07.15 for more information.

Files

net_dbg.h/net_dbg.c

Prototype

```
NET_DBG_STATUS  NetDbg_MonTaskStatusGetRsrcLo(void);
```

A.08 ICMP Functions

A.08.01 NetICMP_CfgTxSrcQuenchTh()

Configure ICMP transmit source quench entry's access transmit threshold.

Files

net_icmp.h/net_icmp.c

Prototype

```
CPU_BOOLEAN  NetICMP_CfgTxSrcQuenchTh(CPU_INT16U  th);
```

Arguments

th	Desired number of received IP packets from a specific IP source host that trips the transmission of an additional ICMP Source Quench Error Message.

Returned Value

DEF_OK	ICMP transmit source quench threshold configured.
DEF_FAIL,	otherwise.

Required Configuration

None.

Notes / Warnings

None.

A.09 IGMP Functions

A.09.01 NetIGMP_HostGrpJoin()

Join a host group.

Files

net_igmp.h/net_igmp.c

Prototype

```
void  NetIGMP_HostGrpJoin (NET_IF_NBR    if_nbr,
                           NET_IP_ADDR   addr_grp,
                           NET_ERR      *perr);
```

Arguments

`if_nbr` Interface number to join host group.

`addr_grp` IP address of host group to join.

`perr` Pointer to variable that will receive the return error code from this function :

NET_IGMP_ERR_NONE
NET_IGMP_ERR_INVALID_ADDR_GRP
NET_IGMP_ERR_HOST_GRP_NONE_AVAIL
NET_IGMP_ERR_HOST_GRP_INVALID_TYPE
NET_IF_ERR_INVALID_IF
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_LOCK

Returned Value

DEF_OK, if host group successfully joined.
DEF_FAIL, otherwise.

Required Configuration

Available only if NET_IP_CFG_MULTICAST_SEL is configured for transmit and receive multicasting (see Section 3.09.02).

Notes / Warnings

'addr_grp' MUST be in host-order.

A.09.02 **NetIGMP_HostGrpLeave()**

Leave a host group.

Files

net_igmp.h/net_igmp.c

Prototype

```
void  NetIGMP_HostGrpLeave (NET_IF_NBR    if_nbr,
                             NET_IP_ADDR   addr_grp,
                             NET_ERR      *perr);
```

Arguments

if_nbr Interface number to leave host group.

addr_grp IP address of host group to leave.

perr Pointer to variable that will receive the return error code from this function :

- NET_IGMP_ERR_NONE
- NET_IGMP_ERR_HOST_GRP_NOT_FOUND
- NET_ERR_INIT_INCOMPLETE
- NET_OS_ERR_LOCK

Returned Value

DEF_OK, if host group successfully left.
DEF_FAIL, otherwise.

Required Configuration

Available only if NET_IP_CFG_MULTICAST_SEL is configured for transmit and receive multicasting (see Section 3.09.02).

Notes / Warnings

'addr_grp' MUST be in host-order.

A.10 Network Interface Functions

A.10.01 NetIF_Add()

Add a network device and hardware as a network interface.

Files

net_if.h/net_if.c

Prototype

```
NET_IF_NBR  NetIF_Add(void      *if_api,
                      void      *dev_api,
                      void      *dev_cfg,
                      void      *phy_api,
                      void      *phy_cfg,
                      NET_ERR   *perr);
```

Arguments

if_api
Pointer to the desired link-layer API for this network interface and device hardware. In most cases, the desired link-layer interface will point to an Ethernet API.

dev_api
Pointer to the desired device driver API for this network interface.

dev_cfg
Pointer to a configuration structure used to configure the device hardware for the specific network interface.

phy_api
Pointer to an optional physical layer device driver API for this network interface. In most cases, a generic physical layer device API will be used, but for Ethernet devices that have non-MII or non-RMII compliant physical layer components, another device-specific physical layer device driver API may be necessary.

phy_cfg
Pointer to a configuration structure used to configure the physical layer hardware for the specific network interface.

perr
Pointer to variable that will receive the return error code from this function:
NET_IF_ERR_NONE
NET_IF_ERR_NULL_PTR
NET_IF_ERR_INVALID_IF

```
NET_IF_ERR_INVALID_CFG
NET_IF_ERR_NONE_AVAIL
NET_BUF_ERR_POOL_INIT
NET_BUF_ERR_INVALID_POOL_TYPE
NET_BUF_ERR_INVALID_POOL_ADDR
NET_BUF_ERR_INVALID_POOL_SIZE
NET_BUF_ERR_INVALID_POOL_QTY
NET_BUF_ERR_INVALID_SIZE
NET_OS_ERR_INIT_DEV_TX_RDY
NET_OS_ERR_INIT_DEV_TX_RDY_NAME
NET_OS_ERR_LOCK
```

Returned Value

Network interface number, if device and hardware successfully added;
`NET_IF_NBR_NONE`, otherwise.

Required Configuration

None.

Notes / Warnings

The first network interface added and started is the default interface used for all default communication. See also Sections A.11.01 & A.11.02.

Both physical layer API and configuration parameters **MUST** either be specified or passed `NULL` pointers.

Additional error codes may be returned by the specific interface or device driver.

A.10.02 **NetIF_AddrHW_Get()**

Get network interface's hardware address.

Files

net_if.h/net_if.c

Prototype

```
void  NetIF_AddrHW_Get(NET_IF_NBR   if_nbr,
                       CPU_INT08U  *paddr_hw,
                       CPU_INT08U  *paddr_len,
                       NET_ERR     *perr);
```

Arguments

`if_nbr` Network interface number to get the hardware address.

`paddr_hw` Pointer to variable that will receive the hardware address.

`paddr_len` Pointer to a variable to pass the length of the address buffer pointed to by `paddr_hw` and return the actual size of the returned hardware address, if NO errors.

`perr` Pointer to variable that will receive the return error code from this function:

NET_IF_ERR_NONE
NET_IF_ERR_NULL_PTR
NET_IF_ERR_NULL_FNCT
NET_IF_ERR_INVALID_IF
NET_IF_ERR_INVALID_CFG
NET_IF_ERR_INVALID_ADDR_LEN
NET_OS_ERR_LOCK

Returned Value

None.

Required Configuration

None.

Notes / Warnings

The hardware address is returned in network-order; i.e. the pointer to the hardware address points to the highest-order byte.

Additional error codes may be returned by the specific interface or device driver.

A.10.03 NetIF_AddrHW_IsValid()

Validate a network interface hardware address.

Files

net_if.h/net_if.c

Prototype

```
CPU_BOOLEAN  NetIF_AddrHW_IsValid(NET_IF_NBR   if_nbr,
                                  CPU_INT08U  *paddr_hw,
                                  NET_ERR     *perr);
```

Arguments

`if_nbr` Network interface number to validate the hardware address.

`paddr_hw` Pointer to a network interface hardware address.

`perr` Pointer to variable that will receive the return error code from this function:

NET_IF_ERR_NONE
NET_IF_ERR_NULL_PTR
NET_IF_ERR_NULL_FNCT
NET_IF_ERR_INVALID_IF
NET_IF_ERR_INVALID_CFG
NET_OS_ERR_LOCK

Returned Value

DEF_YES, if hardware address valid;
DEF_NO, otherwise.

Required Configuration

None.

Notes / Warnings

None.

A.10.04 NetIF_AddrHW_Set()

Set network interface's hardware address.

Files

net_if.h/net_if.c

Prototype

```
void  NetIF_AddrHW_Set(NET_IF_NBR   if_nbr,
                       CPU_INT08U  *paddr_hw,
                       CPU_INT08U   addr_len,
                       NET_ERR     *perr);
```

Arguments

`if_nbr` Network interface number to set hardware address.

`paddr_hw` Pointer to a hardware address.

`addr_len` Length of hardware address.

`perr` Pointer to variable that will receive the return error code from this function:

NET_IF_ERR_NONE
NET_IF_ERR_NULL_PTR
NET_IF_ERR_NULL_FNCT
NET_IF_ERR_INVALID_IF
NET_IF_ERR_INVALID_CFG
NET_IF_ERR_INVALID_STATE
NET_IF_ERR_INVALID_ADDR
NET_IF_ERR_INVALID_ADDR_LEN
NET_OS_ERR_LOCK

Returned Value

None.

Required Configuration

None.

Notes / Warnings

The hardware address **MUST** be in network-order; i.e. the pointer to the hardware address **MUST** point to the highest-order byte.

The network interface **MUST** be stopped **BEFORE** setting a new hardware address, which does **NOT** take effect until the interface is re-started.

Additional error codes may be returned by the specific interface or device driver.

A.10.05 NetIF_CfgPerfMonPeriod()

Configure Network Interface Performance Monitor Handler timeout.

Files

net_if.h/net_if.c

Prototype

```
CPU_BOOLEAN  NetIF_CfgPerfMonPeriod(CPU_INT16U  timeout_ms);
```

Arguments

timeout_ms	Desired value for Network Interface Performance Monitor Handler timeout (in milliseconds).

Returned Value

DEF_OK,	Network Interface Performance Monitor Handler timeout configured;
DEF_FAIL,	otherwise.

Required Configuration

Available only if NET_CTR_CFG_STAT_EN is enabled (see Section 3.04.01).

Notes / Warnings

None.

A.10.06 NetIF_CfgPhyLinkPeriod()

Configure Network Interface Physical Link State Handler timeout.

Files

net_if.h/net_if.c

Prototype

```
CPU_BOOLEAN  NetIF_CfgPhyLinkPeriod(CPU_INT16U  timeout_ms);
```

Arguments

timeout_ms	Desired value for Network Interface Link State Handler timeout (in milliseconds).

Returned Value

DEF_OK,	Network Interface Physical Link State Handler timeout configured;
DEF_FAIL,	otherwise.

Required Configuration

None.

Notes / Warnings

None.

A.10.07 NetIF_IO_Ctrl()

Handle network interface &/or device specific (I/O) control(s).

Files

net_if.h/net_if.c

Prototype

```
void  NetIF_IO_Ctrl(NET_IF_NBR   if_nbr,
                    CPU_INT08U   opt,
                    void        *p_data,
                    NET_ERR     *perr);
```

Arguments

if_nbr Network interface number to handle (I/O) controls.

opt Desired I/O control option code to perform; additional control options may be defined by the device driver:
NET_IF_IO_CTRL_LINK_STATE_GET
NET_IF_IO_CTRL_LINK_STATE_UPDATE

p_data Pointer to variable that will receive the I/O control information.

perr Pointer to variable that will receive the return error code from this function:
NET_IF_ERR_NONE
NET_IF_ERR_NULL_PTR
NET_IF_ERR_NULL_FNCT
NET_IF_ERR_INVALID_IF
NET_IF_ERR_INVALID_CFG
NET_IF_ERR_INVALID_IO_CTRL_OPTNET_OS_ERR_LOCK

Returned Value

None.

Required Configuration

None.

Notes / Warnings

Additional error codes may be returned by the specific interface or device driver.

A.10.08 NetIF_IsEn()

Validate network interface as enabled.

Files

net_if.h/net_if.c

Prototype

```
CPU_BOOLEAN  NetIF_IsEn(NET_IF_NBR   if_nbr,
                        NET_ERR     *perr);
```

Arguments

if_nbr	Network interface number to validate.
perr	Pointer to variable that will receive the return error code from this function: NET_IF_ERR_NONE NET_IF_ERR_INVALID_IF NET_OS_ERR_LOCK

Returned Value

DEF_YES,	network interface valid and enabled;
DEF_NO,	network interface invalid or disabled.

Required Configuration

None.

Notes / Warnings

None.

A.10.09 NetIF_IsEnCfgd()

Validate configured network interface as enabled.

Files

net_if.h/net_if.c

Prototype

```
CPU_BOOLEAN  NetIF_IsEnCfgd(NET_IF_NBR   if_nbr,
                            NET_ERR     *perr);
```

Arguments

if_nbr Network interface number to validate.

perr Pointer to variable that will receive the return error code from this function:

NET_IF_ERR_NONE
NET_IF_ERR_INVALID_IF
NET_OS_ERR_LOCK

Returned Value

DEF_YES, network interface valid and enabled;
DEF_NO, network interface invalid or disabled.

Required Configuration

None.

Notes / Warnings

None.

A.10.10 NetIF_IsValid()

Validate network interface number.

Files

net_if.h/net_if.c

Prototype

```
CPU_BOOLEAN  NetIF_IsValid(NET_IF_NBR    if_nbr,
                           NET_ERR      *perr);
```

Arguments

if_nbr Network interface number to validate.

perr Pointer to variable that will receive the return error code from this function:

NET_IF_ERR_NONE
NET_IF_ERR_INVALID_IF
NET_OS_ERR_LOCK

Returned Value

DEF_YES, network interface number valid;

DEF_NO, network interface number invalid / **NOT** yet configured.

Required Configuration

None.

Notes / Warnings

None.

A.10.11 NetIF_IsValidCfgd()

Validate configured network interface number.

Files

net_if.h/net_if.c

Prototype

```
CPU_BOOLEAN  NetIF_IsValidCfgd(NET_IF_NBR   if_nbr,
                               NET_ERR     *perr);
```

Arguments

if_nbr Network interface number to validate.

perr Pointer to variable that will receive the return error code from this function:

NET_IF_ERR_NONE
NET_IF_ERR_INVALID_IF
NET_OS_ERR_LOCK

Returned Value

DEF_YES, network interface number valid;
DEF_NO, network interface number invalid / **NOT** yet configured or reserved.

Required Configuration

None.

Notes / Warnings

None.

A.10.12 NetIF_LinkStateGet()

Get network interface's last known physical link state.

Files

net_if.h/net_if.c

Prototype

```
CPU_BOOLEAN  NetIF_LinkStateGet(NET_IF_NBR    if_nbr,
                                NET_ERR      *perr);
```

Arguments

if_nbr	Network interface number to get last known physical link state.
perr	Pointer to variable that will receive the return error code from this function: NET_IF_ERR_NONE NET_IF_ERR_INVALID_IF NET_OS_ERR_LOCK

Returned Value

NET_IF_LINK_UP,	if **NO** errors and network interface's last known physical link state was 'UP';
NET_IF_LINK_DOWN,	otherwise.

Required Configuration

None.

Notes / Warnings

Use NetIF_IO_Ctrl() with option NET_IF_IO_CTRL_LINK_STATE_GET to get a network interface's current physical link state.

A.10.13 NetIF_MTU_Get()

Get network interface’s MTU.

Files

net_if.h/net_if.c

Prototype

```
NET_MTU  NetIF_MTU_Get(NET_IF_NBR   if_nbr,
                       NET_ERR     *perr);
```

Arguments

if_nbr — Network interface number to get MTU.

perr — Pointer to variable that will receive the return error code from this function:

NET_IF_ERR_NONE
NET_IF_ERR_INVALID_IF
NET_OS_ERR_LOCK

Returned Value

Network interface’s MTU, if NO errors;
0, otherwise.

Required Configuration

None.

Notes / Warnings

None.

A.10.14 NetIF_MTU_Set()

Set network interface's MTU.

Files

net_if.h/net_if.c

Prototype

```
void  NetIF_MTU_Set(NET_IF_NBR   if_nbr,
                    NET_MTU      mtu,
                    NET_ERR     *perr);
```

Arguments

if_nbr Network interface number to set MTU.

mtu Desired maximum transmission unit size to configure.

perr Pointer to variable that will receive the return error code from this function:

NET_IF_ERR_NONE
NET_IF_ERR_NULL_FNCT
NET_IF_ERR_INVALID_IF
NET_IF_ERR_INVALID_CFG
NET_IF_ERR_INVALID_MTU
NET_OS_ERR_LOCK

Returned Value

None.

Required Configuration

None.

Notes / Warnings

Additional error codes may be returned by the specific interface or device driver.

A.10.15 NetIF_Start()

Start a network interface.

Files

net_if.h/net_if.c

Prototype

```
void  NetIF_Start(NET_IF_NBR   if_nbr,
                  NET_ERR     *perr);
```

Arguments

if_nbr   Network interface number to start.

perr   Pointer to variable that will receive the return error code from this function:

NET_IF_ERR_NONE
NET_IF_ERR_NULL_FNCT
NET_IF_ERR_INVALID_IF
NET_IF_ERR_INVALID_CFG
NET_IF_ERR_INVALID_STATE
NET_OS_ERR_LOCK

Returned Value

None.

Required Configuration

None.

Notes / Warnings

Additional error codes may be returned by the specific interface or device driver.

A.10.16 NetIF_Stop()

Stop a network interface.

Files

net_if.h/net_if.c

Prototype

```
void  NetIF_Stop(NET_IF_NBR   if_nbr,
                 NET_ERR     *perr);
```

Arguments

if_nbr   Network interface number to stop.

perr   Pointer to variable that will receive the return error code from this function:

NET_IF_ERR_NONE
NET_IF_ERR_NULL_FNCT
NET_IF_ERR_INVALID_IF
NET_IF_ERR_INVALID_CFG
NET_IF_ERR_INVALID_STATE
NET_OS_ERR_LOCK

Returned Value

None.

Required Configuration

None.

Notes / Warnings

Additional error codes may be returned by the specific interface or device driver.

A.11 IP Functions

A.11.01 NetIP_CfgAddrAdd()

Add a statically-configured IP host address, subnet mask, & default gateway to an interface.

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_CfgAddrAdd(NET_IF_NBR    if_nbr,
                              NET_IP_ADDR   addr_host,
                              NET_IP_ADDR   addr_subnet_mask,
                              NET_IP_ADDR   addr_dflt_gateway,
                              NET_ERR      *perr);
```

Arguments

`if_nbr` Interface number to configure.

`addr_host` Desired IP address to add to this interface.

`addr_subnet_mask` Desired IP address subnet mask.

`addr_dflt_gateway` Desired IP default gateway address.

`perr` Pointer to variable that will receive the return error code from this function:

NET_IP_ERR_NONE
NET_IP_ERR_INVALID_ADDR_HOST
NET_IP_ERR_INVALID_ADDR_GATEWAY
NET_IP_ERR_ADDR_CFG_STATE
NET_IP_ERR_ADDR_TBL_FULL
NET_IP_ERR_ADDR_CFG_IN_USE
NET_IF_ERR_INVALID_IF
NET_OS_ERR_LOCK

Returned Value

DEF_OK,	if valid IP address, subnet mask, & default gateway statically-configured;
DEF_FAIL,	otherwise.

Required Configuration

None.

Notes / Warnings

IP addresses **MUST** be configured in host-order.

An interface may be configured with either:
(1) One or more statically- configured IP addresses (default configuration) **OR**
(2) Exactly one dynamically-configured IP address (see Section A.11.02).

If an interface's address(s) are dynamically-configured, **NO** statically-configured address(s) may be added until all dynamically-configured address(s) are removed.

The maximum number of IP address(s) configured on any interface is limited to `NET_IP_CFG_IF_MAX_NBR_ADDR` (see Section 3.09.01).

Note that on the default interface, the first IP address added will be the default address used for all default communication. See also Section A.10.01.

A host **MAY** be configured without a gateway address to allow communication only with other hosts on its local network. However, any configured gateway address **MUST** be on the same network as the configured host IP address—i.e. the network portion of the configured IP address and the configured gateway addresses **MUST** be identical.

A.11.02 NetIP_CfgAddrAddDynamic()

Add a dynamically-configured IP host address, subnet mask, & default gateway to an interface.

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_CfgAddrAdd(NET_IF_NBR    if_nbr,
                              NET_IP_ADDR   addr_host,
                              NET_IP_ADDR   addr_subnet_mask,
                              NET_IP_ADDR   addr_dflt_gateway,
                              NET_ERR      *perr);
```

Arguments

if_nbr	Interface number to configure.
addr_host	Desired IP address to add to this interface.
addr_subnet_mask	Desired IP address subnet mask.
addr_dflt_gateway	Desired IP default gateway address.
perr	Pointer to variable that will receive the return error code from this function: NET_IP_ERR_NONE NET_IP_ERR_INVALID_ADDR_HOST NET_IP_ERR_INVALID_ADDR_GATEWAY NET_IP_ERR_ADDR_CFG_STATE NET_IP_ERR_ADDR_TBL_FULL NET_IP_ERR_ADDR_CFG_IN_USE NET_IF_ERR_INVALID_IF NET_OS_ERR_LOCK

Returned Value

DEF_OK,	if valid IP address, subnet mask, & default gateway dynamically configured;
DEF_FAIL,	otherwise.

Required Configuration

None.

Notes / Warnings

IP addresses **MUST** be configured in host-order.

An interface may be configured with either:
(1) One or more statically- configured IP addresses (see Section A.11.01) **OR**
(2) Exactly one dynamically-configured IP address.

This function should **ONLY** be called by appropriate network application function(s) [e.g. DHCP initialization functions]. However, if your application attempts to dynamically configure IP address(s), it **MUST** call `NetIP_CfgAddrAddDynamicStart()` before calling `NetIP_CfgAddrAddDynamic()`.

Note that on the default interface, the first IP address added will be the default address used for all default communication. See also Section A.10.01.

A host **MAY** be configured without a gateway address to allow communication only with other hosts on its local network. However, any configured gateway address **MUST** be on the same network as the configured host IP address—i.e. the network portion of the configured IP address and the configured gateway addresses **MUST** be identical.

A.11.03 NetIP_CfgAddrAddDynamicStart()

Start dynamic IP address configuration for an interface.

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_CfgAddrAddDynamicStart(NET_IF_NBR   if_nbr,
                                          NET_ERR     *perr);
```

Arguments

if_nbr Interface number to start dynamic address configuration.

perr Pointer to variable that will receive the return error code from this function:

NET_IP_ERR_NONE
NET_IP_ERR_ADDR_CFG_STATE
NET_IP_ERR_ADDR_CFG_IN_PROGRESS
NET_IF_ERR_INVALID_IF
NET_OS_ERR_LOCK

Returned Value

DEF_OK, if dynamic IP address configuration successfully started;
DEF_FAIL, otherwise.

Required Configuration

None.

Notes / Warnings

This function should **ONLY** be called by appropriate network application function(s) [e.g. DHCP initialization functions]. However, if your application attempts to dynamically configure IP address(s), it **MUST** call `NetIP_CfgAddrAddDynamicStart()` before calling `NetIP_CfgAddrAddDynamic()`.

A.11.04 NetIP_CfgAddrAddDynamicStop()

Stop dynamic IP address configuration for an interface.

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_CfgAddrAddDynamicStop(NET_IF_NBR   if_nbr,
                                         NET_ERR     *perr);
```

Arguments

if_nbr — Interface number to stop dynamic address configuration.

perr — Pointer to variable that will receive the return error code from this function:
NET_IP_ERR_NONE
NET_IP_ERR_ADDR_CFG_STATE
NET_IF_ERR_INVALID_IF
NET_OS_ERR_LOCK

Returned Value

DEF_OK, if dynamic IP address configuration successfully stopped;
DEF_FAIL, otherwise.

Required Configuration

None.

Notes / Warnings

This function should **ONLY** be called by appropriate network application function(s) [e.g. DHCP initialization functions]. However, if your application attempts to dynamically configure IP address(s), it must call NetIP_CfgAddrAddDynamicStop() **ONLY** after calling NetIP_CfgAddrAddDynamicStart() and dynamic IP address configuration has failed.

A.11.05 NetIP_CfgAddrRemove()

Remove a configured IP host address from an interface.

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_CfgAddrRemove(NET_IF_NBR    if_nbr,
                                 NET_IP_ADDR   addr_host,
                                 NET_ERR      *perr);
```

Arguments

if_nbr Interface number to remove configured IP host address.

addr_host IP address to remove.

perr Pointer to variable that will receive the return error code from this function:

NET_IP_ERR_NONE
NET_IP_ERR_INVALID_ADDR_HOST
NET_IP_ERR_ADDR_CFG_STATE
NET_IP_ERR_ADDR_TBL_EMPTY
NET_IP_ERR_ADDR_NOT_FOUND
NET_IF_ERR_INVALID_IF
NET_OS_ERR_LOCK

Returned Value

DEF_OK, if interface's configured IP host address successfully removed;
DEF_FAIL, otherwise.

Required Configuration

None.

Notes / Warnings

None.

A.11.06 NetIP_CfgAddrRemoveAll()

Remove all configured IP host address(s) from an interface.

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_CfgAddrRemoveAll(NET_IF_NBR   if_nbr,
                                    NET_ERR     *perr);
```

Arguments

if_nbr Interface number to remove all configured IP host address(s).

perr Pointer to variable that will receive the return error code from this function:

NET_IP_ERR_NONE
NET_IP_ERR_ADDR_CFG_STATE
NET_IF_ERR_INVALID_IF
NET_OS_ERR_LOCK

Returned Value

DEF_OK, if all interface's configured IP host address(s) successfully removed;
DEF_FAIL, otherwise.

Required Configuration

None.

Notes / Warnings

None.

A.11.07 NetIP_CfgFragReasmTimeout()

Configure IP fragment reassembly timeout.

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_CfgFragReasmTimeout(CPU_INT08U  timeout_sec);
```

Arguments

timeout_sec Desired value for IP fragment reassembly timeout (in seconds).

Returned Value

DEF_OK IP fragment reassembly timeout successfully configured.

DEF_FAIL, otherwise.

Required Configuration

None.

Notes / Warnings

Fragment reassembly timeout is the maximum time allowed between received fragments of the same IP datagram.

A.11.08 NetIP_GetAddrDfltGateway()

Get the default gateway IP address for a host's configured IP address.

Files

net_ip.h/net_ip.c

Prototype

```
NET_IP_ADDR  NetIP_GetAddrDfltGateway(NET_IP_ADDR   addr,
                                      NET_ERR      *perr);
```

Arguments

addr Configured IP host address.

perr Pointer to variable that will receive the return error code from this function:

NET_IP_ERR_NONE
NET_IP_ERR_INVALID_ADDR_HOST
NET_OS_ERR_LOCK

Returned Value

Configured IP host address's default gateway (in host-order), if NO errors;
NET_IP_ADDR_NONE, otherwise.

Required Configuration

None.

Notes / Warnings

ALL IP addresses in host-order.

A.11.09 NetIP_GetAddrHost()

Get an interface's configured IP host address(s).

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_GetAddrHost(NET_IF_NBR          if_nbr,
                               NET_IP_ADDR        *paddr_tbl,
                               NET_IP_ADDRS_QTY   *paddr_tbl_qty,
                               NET_ERR            *perr);
```

Arguments

if_nbr — Interface number to get configured IP host address(s).

paddr_tbl — Pointer to IP address table that will receive the IP host address(s) in host-order for this interface.

paddr_tbl_qty — Pointer to a variable to:

Pass the size of the address table, in number of IP addresses, pointed to by paddr_tbl.

Return the actual number of IP addresses, if NO errors;
Return 0, otherwise.

perr — Pointer to variable that will receive the return error code from this function:

NET_IP_ERR_NONE
NET_IP_ERR_NULL_PTR
NET_IP_ERR_ADDR_NONE_AVAIL
NET_IP_ERR_ADDR_CFG_IN_PROGRESS
NET_IP_ERR_ADDR_TBL_SIZE
NET_IF_ERR_INVALID_IF
NET_OS_ERR_LOCK

Returned Value

DEF_OK, if interface's configured IP host address(s) successfully returned;
DEF_FAIL, otherwise.

Required Configuration

None.

Notes / Warnings

ALL IP addresses returned in host-order.

A.11.10 NetIP_GetAddrSubnetMask()

Get the IP address subnet mask for a host's configured IP address.

Files

net_ip.h/net_ip.c

Prototype

```
NET_IP_ADDR  NetIP_GetAddrSubnetMask(NET_IP_ADDR   addr,
                                     NET_ERR      *perr);
```

Arguments

addr Configured IP host address.

perr Pointer to variable that will receive the return error code from this function:

NET_IP_ERR_NONE
NET_IP_ERR_INVALID_ADDR_HOST
NET_OS_ERR_LOCK

Returned Value

Configured IP host address's subnet mask (in host-order), if NO errors;
NET_IP_ADDR_NONE, otherwise.

Required Configuration

None.

Notes / Warnings

ALL IP addresses in host-order.

A.11.11 NetIP_IsAddrBroadcast()

Validate an IP address as the limited broadcast IP address.

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_IsAddrBroadcast(NET_IP_ADDR  addr);
```

Arguments

addr	IP address to validate.

Returned Value

DEF_YES,	if IP address is a limited broadcast IP address;
DEF_NO,	otherwise.

Required Configuration

None.

Notes / Warnings

IP address **MUST** be in host-order.

The broadcast IP address is 255.255.255.255.

A.11.12 NetIP_IsAddrClassA()

Validate an IP address as a Class-A IP address.

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_IsAddrClassA(NET_IP_ADDR  addr);
```

Arguments

addr	IP address to validate.

Returned Value

DEF_YES,	if IP address is a Class-A IP address;
DEF_NO,	otherwise.

Required Configuration

None.

Notes / Warnings

IP address **MUST** be in host-order.

Class-A IP addresses have their most significant bit be '0'.

A.11.13 NetIP_IsAddrClassB()

Validate an IP address as a Class-B IP address.

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_IsAddrClassB(NET_IP_ADDR  addr);
```

Arguments

addr	IP address to validate.

Returned Value

DEF_YES,	if IP address is a Class-B IP address;
DEF_NO,	otherwise.

Required Configuration

None.

Notes / Warnings

IP address **MUST** be in host-order.

Class-B IP addresses have their most significant bits be '10'.

A.11.14 NetIP_IsAddrClassC()

Validate an IP address as a Class-C IP address.

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_IsAddrClassC(NET_IP_ADDR  addr);
```

Arguments

addr	IP address to validate.

Returned Value

DEF_YES,	if IP address is a Class-C IP address;
DEF_NO,	otherwise.

Required Configuration

None.

Notes / Warnings

IP address **MUST** be in host-order.

Class-C IP addresses have their most significant bits be '110'.

A.11.15 NetIP_IsAddrHost()

Validate an IP address as one the host's IP address(s).

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_IsAddrHost(NET_IP_ADDR  addr);
```

Arguments

addr	IP address to validate.

Returned Value

DEF_YES,	if IP address is any one of the host's IP address(s);
DEF_NO,	otherwise.

Required Configuration

None.

Notes / Warnings

IP address **MUST** be in host-order.

A.11.16 NetIP_IsAddrHostCfgd()

Validate an IP address as one the host's configured IP address(s).

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_IsAddrHostCfgd(NET_IP_ADDR  addr);
```

Arguments

addr	IP address to validate.

Returned Value

DEF_YES,	if IP address is any one of the host's configured IP address(s);
DEF_NO,	otherwise.

Required Configuration

None.

Notes / Warnings

IP address **MUST** be in host-order.

A.11.17 NetIP_IsAddrLocalHost()

Validate an IP address as a 'Localhost' IP address.

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_IsAddrLocalHost(NET_IP_ADDR  addr);
```

Arguments

addr	IP address to validate.

Returned Value

DEF_YES,	if IP address is a 'Localhost' IP address;
DEF_NO,	otherwise.

Required Configuration

None.

Notes / Warnings

IP address **MUST** be in host-order.

Localhost IP addresses are any host address in the '127.<host>' subnet.

A.11.18 NetIP_IsAddrLocalLink()

Validate an IP address as a link-local IP address.

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_IsAddrLocalLink(NET_IP_ADDR  addr);
```

Arguments

addr	IP address to validate.

Returned Value

DEF_YES,	if IP address is a link-local IP address;
DEF_NO,	otherwise.

Required Configuration

None.

Notes / Warnings

IP address **MUST** be in host-order.

Link-local IP addresses are any host address in the '169.254.<host>' subnet.

A.11.19 NetIP_IsAddrsCfgdOnIF()

Check if any IP address(s) are configured on an interface.

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_IsAddrsHostCfgdOnIF(NET_IF_NBR   if_nbr,
                                      NET_ERR     *perr);
```

Arguments

if_nbr Interface number to check for configured IP host address(s).

perr Pointer to variable that will receive the return error code from this function:

NET_IP_ERR_NONE
NET_IF_ERR_INVALID_IF
NET_OS_ERR_LOCK

Returned Value

DEF_YES, if **ANY** IP host address(s) are configured on the interface;
DEF_NO, otherwise.

Required Configuration

None.

Notes / Warnings

None.

A.11.20 NetIP_IsAddrThisHost()

Validate an IP address as the 'This Host' initialization IP address.

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_IsAddrThisHost(NET_IP_ADDR  addr);
```

Arguments

addr	IP address to validate.

Returned Value

DEF_YES,	if IP address is a 'This Host' initialization IP address;
DEF_NO,	otherwise.

Required Configuration

None.

Notes / Warnings

IP address **MUST** be in host-order.

The 'This Host' initialization IP address is 0.0.0.0.

A.11.21 NetIP_IsValidAddrHost()

Validate an IP address as a valid IP host address.

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_IsValidAddrHost(NET_IP_ADDR  addr_host);
```

Arguments

addr_host IP host address to validate.

Returned Value

DEF_YES, if valid IP host address;
DEF_NO, otherwise.

Required Configuration

None.

Notes / Warnings

IP address **MUST** be in host-order.

A valid IP host address must **NOT** be one of the following:

1. This Host . . . . See Section A.11.20
2. Specified Host
3. Localhost . . . . See Section A.11.17
4. Limited Broadcast . . . . . . See Section A.11.11
5. Directed Broadcast

A.11.22 NetIP_IsValidAddrHostCfgd()

Validate an IP address as a valid, configurable IP host address.

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN  NetIP_IsValidAddrHostCfgd(NET_IP_ADDR  addr_host,
                                       NET_IP_ADDR  addr_subnet_mask);
```

Arguments

addr_host	IP host address to validate.
addr_subnet_mask	IP host address subnet mask.

Returned Value

DEF_YES,	if configurable IP host address;
DEF_NO,	otherwise.

Required Configuration

None.

Notes / Warnings

IP addresses **MUST** be in host-order.

A configurable IP host address must **NOT** be one of the following:

1. This Host . . . . See Section A.11.20
2. Specified Host
3. Localhost . . . . See Section A.11.17
4. Limited Broadcast . . . . . . See Section A.11.11
5. Directed Broadcast
6. Subnet Broadcast

A.11.23 NetIP_IsValidAddrSubnetMask()

Validate an IP address subnet mask.

Files

net_ip.h/net_ip.c

Prototype

```
CPU_BOOLEAN NetIP_IsValidAddrSubnetMask(NET_IP_ADDR addr_subnet_mask);
```

Arguments

addr_subnet_mask	IP host address subnet mask.

Returned Value

DEF_YES,	if valid IP address subnet mask;
DEF_NO	otherwise.

Required Configuration

None.

Notes / Warnings

IP address **MUST** be in host-order.

A.12 Network Socket Functions

A.12.01 NetSock_Accept() / accept() TCP

Wait for new socket connections on a listening server socket (see Section A.12.27). When a new connection arrives and the TCP handshake has successfully completed, a new socket ID is returned for the new connection with the remote host's address and port number returned in the socket address structure.

Files

net_sock.h/net_sock.c
net_bsd.h/net_bsd.c

Prototypes

```
NET_SOCK_ID  NetSock_Accept(NET_SOCK_ID         sock_id,
                            NET_SOCK_ADDR      *paddr_remote,
                            NET_SOCK_ADDR_LEN  *paddr_len,
                            NET_ERR            *perr);

int  accept(        int          sock_id,
            struct  sockaddr    *paddr_remote,
                    socklen_t   *paddr_len);
```

Arguments

`sock_id` This is the socket ID returned by `NetSock_Open()`/`socket()` when the socket was created. This socket is assumed to be bound to an address and listening for new connections (see Section A.12.27).

`paddr_remote` Pointer to a socket address structure (see Section 6.01) to return the remote host address of the new accepted connection.

`paddr_len` Pointer to the size of the socket address structure which **MUST** be passed the size of the socket address structure [e.g. `sizeof(NET_SOCK_ADDR_IP)`]. Returns size of the accepted connection's socket address structure.

`perr` Pointer to variable that will receive the return error code from this function:
NET_SOCK_ERR_NONE
NET_SOCK_ERR_NULL_PTR

```
NET_SOCK_ERR_NONE_AVAIL
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_CLOSED
NET_SOCK_ERR_INVALID_SOCK
NET_SOCK_ERR_INVALID_FAMILY
NET_SOCK_ERR_INVALID_TYPE
NET_SOCK_ERR_INVALID_STATE
NET_SOCK_ERR_INVALID_OP
NET_SOCK_ERR_CONN_ACCEPT_Q_NONE_AVAIL
NET_SOCK_ERR_CONN_SIGNAL_TIMEOUT
NET_SOCK_ERR_CONN_FAIL
NET_SOCK_ERR_FAULT
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_LOCK
```

Returned Value

Returns a non-negative socket descriptor ID for the new accepted connection, if successful; `NET_SOCK_BSD_ERR_ACCEPT/-1`, otherwise.

If the socket is configured for non-blocking, a return value of `NET_SOCK_BSD_ERR_ACCEPT/-1` may indicate that the no requests for connection were queued when `NetSock_Accept()/accept()` was called. In this case, the server can 'poll' for a new connection at a later time.

Required Configuration

`NetSock_Accept()` is available only if either `NET_CFG_TRANSPORT_LAYER_SEL` is configured for TCP (see Section 3.12) and/or `NET_UDP_CFG_APP_API_SEL` is configured for sockets (see Section 3.13.01).

In addition, `accept()` is available only if `NET_BSD_CFG_API_EN` is enabled (see Section 3.15.12).

Notes / Warnings

See Section 6.01 for socket address structure formats.

A.12.02 **NetSock_Bind() / bind() TCP/UDP**

Assign network addresses to sockets. Typically, server sockets bind to addresses but client sockets do not. Servers may bind to one of the local host's addresses but usually bind to the wildcard address (`NET_SOCK_ADDR_IP_WILDCARD/INADDR_ANY`) on a specific, well-known port number. Whereas client sockets usually bind to one of the local host's addresses but with a random port number (by configuring the socket address structure's port number field with a value of `0`).

Files

`net_sock.h/net_sock.c`
`net_bsd.h/net_bsd.c`

Prototypes

```
NET_SOCK_RTN_CODE  NetSock_Bind(NET_SOCK_ID          sock_id,
                                NET_SOCK_ADDR       *paddr_local,
                                NET_SOCK_ADDR_LEN    addr_len,
                                NET_ERR             *perr);


int  bind(        int          sock_id,
          struct  sockaddr    *paddr_local,
                  socklen_t    addr_len);
```

Arguments

`sock_id` This is the socket ID returned by `NetSock_Open()/socket()` when the socket was created.

`paddr_local` Pointer to a socket address structure (see Section 6.01) which contains the local host address to bind the socket to.

`addr_len` Size of the socket address structure which **MUST** be passed the size of the socket address structure [e.g. `sizeof(NET_SOCK_ADDR_IP)`].

`perr` Pointer to variable that will receive the return error code from this function:

`NET_SOCK_ERR_NONE`
`NET_SOCK_ERR_NOT_USED`
`NET_SOCK_ERR_CLOSED`
`NET_SOCK_ERR_INVALID_SOCK`
`NET_SOCK_ERR_INVALID_FAMILY`
`NET_SOCK_ERR_INVALID_PROTOCOL`
`NET_SOCK_ERR_INVALID_TYPE`

NET_SOCK_ERR_INVALID_STATE
NET_SOCK_ERR_INVALID_OP
NET_SOCK_ERR_INVALID_ADDR
NET_SOCK_ERR_ADDR_IN_USE
NET_SOCK_ERR_PORT_NBR_NONE_AVAIL
NET_SOCK_ERR_CONN_FAIL
NET_IF_ERR_INVALID_IF
NET_IP_ERR_ADDR_NONE_AVAIL
NET_IP_ERR_ADDR_CFG_IN_PROGRESS
NET_CONN_ERR_NULL_PTR
NET_CONN_ERR_NOT_USED
NET_CONN_ERR_NONE_AVAIL
NET_CONN_ERR_INVALID_CONN
NET_CONN_ERR_INVALID_FAMILY
NET_CONN_ERR_INVALID_TYPE
NET_CONN_ERR_INVALID_PROTOCOL_IX
NET_CONN_ERR_INVALID_ADDR_LEN
NET_CONN_ERR_ADDR_NOT_USED
NET_CONN_ERR_ADDR_IN_USE
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_LOCK

Returned Value

NET_SOCK_BSD_ERR_NONE/0, if successful;
NET_SOCK_BSD_ERR_BIND/-1, otherwise.

Required Configuration

NetSock_Bind() is available only if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01).

In addition, bind() is available only if NET_BSD_CFG_API_EN is enabled (see Section 3.15.12).

Notes / Warnings

See Section 6.01 for socket address structure formats.

Sockets may bind to any of the host's configured addresses, any localhost address (127.x.y.z network; e.g. 127.0.0.1), any link-local address (169.254.y.z network; e.g. 169.254.65.111), as well as the wildcard address (NET_SOCK_ADDR_IP_WILDCARD/INADDR_ANY, i.e. 0.0.0.0).

Sockets may bind to specific port numbers or request a random, ephemeral port number by configuring the socket address structure's port number field with a value of `0`. Sockets may **NOT** bind to a port number that is within the configured range of random port numbers (see Sections 3.15.02 & 3.15.07):

```
NET_SOCK_CFG_PORT_NBR_RANDOM_BASE <= RandomPortNbrs <=
(NET_SOCK_CFG_PORT_NBR_RANDOM_BASE + NET_SOCK_CFG_NBR_SOCK + 10)
```

A.12.03 **NetSock_CfgTimeoutConnAcceptDflt() TCP**

Set socket's connection accept timeout to configured-default value.

Files

net_sock.h/net_sock.c

Prototype

```
void  NetSock_CfgTimeoutConnAcceptDflt(NET_SOCK_ID   sock_id,
                                       NET_ERR      *perr);
```

Arguments

sock_id — This is the socket ID returned by NetSock_Open()/socket() when the socket was created **OR** by NetSock_Accept()/accept() when a connection was accepted.

perr — Pointer to variable that will receive the return error code from this function:

NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_INVALID_SOCK
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_INVALID_TIME
NET_OS_ERR_LOCK

Returned Value

None.

Required Configuration

Available only if NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12).

Notes / Warnings

None.

A.12.04 **NetSock_CfgTimeoutConnAcceptGet_ms() TCP**

Get socket's connection accept timeout value.

Files

net_sock.h/net_sock.c

Prototype

```
CPU_INT32U NetSock_CfgTimeoutConnAcceptGet_ms(NET_SOCK_ID   sock_id,
                                               NET_ERR      *perr);
```

Arguments

sock_id — This is the socket ID returned by NetSock_Open()/socket() when the socket was created **OR** by NetSock_Accept()/accept() when a connection was accepted.

perr — Pointer to variable that will receive the return error code from this function:
NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_INVALID_SOCK
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_LOCK

Returned Value

0, on **ANY** errors;
NET_TMR_TIME_INFINITE, if infinite (i.e. **NO** timeout) value configured;
Timeout in number of milliseconds, otherwise.

Required Configuration

Available only if NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12).

Notes / Warnings

None.

A.12.05 **NetSock_CfgTimeoutConnAcceptSet()** TCP

Set socket's connection accept timeout value.

Files

net_sock.h/net_sock.c

Prototype

```
void  NetSock_CfgTimeoutConnAcceptSet(NET_SOCK_ID   sock_id,
                                      CPU_INT32U    timeout_ms,
                                      NET_ERR      *perr);
```

Arguments

sock_id
This is the socket ID returned by NetSock_Open()/socket() when the socket was created **OR** by NetSock_Accept()/accept() when a connection was acepted.

timeout_ms
Desired timeout value:
NET_TMR_TIME_INFINITE, if infinite (i.e. **NO** timeout) value desired.
In number of milliseconds, otherwise.

perr
Pointer to variable that will receive the return error code from this function:
NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_INVALID_SOCK
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_INVALID_TIME
NET_OS_ERR_LOCK

Returned Value

None.

Required Configuration

Available only if NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12).

Notes / Warnings

None.

A.12.06 NetSock_CfgTimeoutConnCloseDflt() TCP

Set socket's connection close timeout to configured-default value.

Files

net_sock.h/net_sock.c

Prototype

```
void  NetSock_CfgTimeoutConnCloseDflt(NET_SOCK_ID   sock_id,
                                      NET_ERR      *perr);
```

Arguments

sock_id This is the socket ID returned by NetSock_Open()/socket() when the socket was created **OR** by NetSock_Accept()/accept() when a connection was accepted.

perr Pointer to variable that will receive the return error code from this function:

NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_INVALID_SOCK
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_INVALID_TIME
NET_OS_ERR_LOCK

Returned Value

None.

Required Configuration

Available only if NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12).

Notes / Warnings

None.

A.12.07 NetSock_CfgTimeoutConnCloseGet_ms() TCP

Get socket's connection close timeout value.

Files

net_sock.h/net_sock.c

Prototype

```
CPU_INT32U  NetSock_CfgTimeoutConnCloseGet_ms(NET_SOCK_ID   sock_id,
                                              NET_ERR      *perr);
```

Arguments

sock_id This is the socket ID returned by NetSock_Open()/socket() when the socket was created **OR** by NetSock_Accept()/accept() when a connection was accepted.

perr Pointer to variable that will receive the return error code from this function:

NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_INVALID_SOCK
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_LOCK

Returned Value

0, on ANY errors;
NET_TMR_TIME_INFINITE, if infinite (i.e. **NO** timeout) value configured;
Timeout in number of milliseconds, otherwise.

Required Configuration

Available only if NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12).

Notes / Warnings

None.

A.12.08 NetSock_CfgTimeoutConnCloseSet() TCP

Set socket's connection close timeout value.

Files

net_sock.h/net_sock.c

Prototype

```
void  NetSock_CfgTimeoutConnCloseSet(NET_SOCK_ID   sock_id,
                                     CPU_INT32U    timeout_ms,
                                     NET_ERR      *perr);
```

Arguments

sock_id — This is the socket ID returned by NetSock_Open()/socket() when the socket was created **OR** by NetSock_Accept()/accept() when a connection was accepted.

timeout_ms — Desired timeout value:
NET_TMR_TIME_INFINITE, if infinite (i.e. **NO** timeout) value desired.
In number of milliseconds, otherwise.

perr — Pointer to variable that will receive the return error code from this function:
NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_INVALID_SOCK
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_INVALID_TIME
NET_OS_ERR_LOCK

Returned Value

None.

Required Configuration

Available only if NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12).

Notes / Warnings

None.

A.12.09 **NetSock_CfgTimeoutConnReqDflt() TCP**

Set socket's connection request timeout to configured-default value.

Files

net_sock.h/net_sock.c

Prototype

```
void  NetSock_CfgTimeoutConnReqDflt(NET_SOCK_ID   sock_id,
                                    NET_ERR      *perr);
```

Arguments

`sock_id` This is the socket ID returned by `NetSock_Open()/socket()` when the socket was created **OR** by `NetSock_Accept()/accept()` when a connection was accepted.

`perr` Pointer to variable that will receive the return error code from this function:

NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_INVALID_SOCK
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_INVALID_TIME
NET_OS_ERR_LOCK

Returned Value

None.

Required Configuration

Available only if `NET_CFG_TRANSPORT_LAYER_SEL` is configured for TCP (see Section 3.12).

Notes / Warnings

None.

A.12.10 **NetSock_CfgTimeoutConnReqGet_ms() TCP**

Get socket’s connection request timeout value.

Files

net_sock.h/net_sock.c

Prototype

```
CPU_INT32U  NetSock_CfgTimeoutConnReqGet_ms(NET_SOCK_ID   sock_id,
                                            NET_ERR      *perr);
```

Arguments

sock_id — This is the socket ID returned by NetSock_Open()/socket() when the socket was created **OR** by NetSock_Accept()/accept() when a connection was accepted.

perr — Pointer to variable that will receive the return error code from this function:
NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_INVALID_SOCK
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_LOCK

Returned Value

0, on **ANY** errors;
NET_TMR_TIME_INFINITE, if infinite (i.e. **NO** timeout) value configured;
Timeout in number of milliseconds, otherwise.

Required Configuration

Available only if NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12).

Notes / Warnings

None.

A.12.11 NetSock_CfgTimeoutConnReqSet() TCP

Set socket's connection request timeout value.

Files

net_sock.h/net_sock.c

Prototype

```
void  NetSock_CfgTimeoutConnReqSet(NET_SOCK_ID    sock_id,
                                   CPU_INT32U     timeout_ms,
                                   NET_ERR       *perr);
```

Arguments

sock_id
: This is the socket ID returned by NetSock_Open()/socket() when the socket was created **OR** by NetSock_Accept()/accept() when a connection was accepted.

timeout_ms
: Desired timeout value:
NET_TMR_TIME_INFINITE, if infinite (i.e. **NO** timeout) value desired.
In number of milliseconds, otherwise.

perr
: Pointer to variable that will receive the return error code from this function:
NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_INVALID_SOCK
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_INVALID_TIME
NET_OS_ERR_LOCK

Returned Value

None.

Required Configuration

Available only if NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12).

Notes / Warnings

None.

A.12.12 **NetSock_CfgTimeoutRxQ_Dflt() TCP/UDP**

Set socket's connection receive queue timeout to configured-default value.

Files

net_sock.h/net_sock.c

Prototype

```
void  NetSock_CfgTimeoutRxQ_Dflt(NET_SOCK_ID   sock_id,
                                 NET_ERR      *perr);
```

Arguments

`sock_id` This is the socket ID returned by `NetSock_Open()/socket()` when the socket was created **OR** by `NetSock_Accept()/accept()` when a connection was accepted.

`perr` Pointer to variable that will receive the return error code from this function:

```
NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_INVALID_SOCK
NET_SOCK_ERR_INVALID_TYPE
NET_SOCK_ERR_INVALID_PROTOCOL
NET_TCP_ERR_CONN_NOT_USED
NET_TCP_ERR_INVALID_CONN
NET_CONN_ERR_NOT_USED
NET_CONN_ERR_INVALID_CONN
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_INVALID_TIME
NET_OS_ERR_LOCK
```

Returned Value

None.

Required Configuration

Available only if either `NET_CFG_TRANSPORT_LAYER_SEL` is configured for TCP (see Section 3.12) and/or `NET_UDP_CFG_APP_API_SEL` is configured for sockets (see Section 3.13.01).

Notes / Warnings

None.

A.12.13 NetSock_CfgTimeoutRxQ_Get_ms() TCP/UDP

Get socket's receive queue timeout value.

Files

net_sock.h/net_sock.c

Prototype

```
CPU_INT32U  NetSock_CfgTimeoutRxQ_Get_ms(NET_SOCK_ID   sock_id,
                                         NET_ERR      *perr);
```

Arguments

sock_id — This is the socket ID returned by NetSock_Open()/socket() when the socket was created **OR** by NetSock_Accept()/accept() when a connection was accepted.

perr — Pointer to variable that will receive the return error code from this function:

NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_INVALID_SOCK
NET_SOCK_ERR_INVALID_TYPE
NET_SOCK_ERR_INVALID_PROTOCOL
NET_TCP_ERR_CONN_NOT_USED
NET_TCP_ERR_INVALID_CONN
NET_CONN_ERR_NOT_USED
NET_CONN_ERR_INVALID_CONN
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_LOCK

Returned Value

0, on **ANY** errors;
NET_TMR_TIME_INFINITE, if infinite (i.e. **NO** timeout) value configured;
Timeout in number of milliseconds, otherwise.

Required Configuration

Available only if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01).

Notes / Warnings

None.

A.12.14 NetSock_CfgTimeoutRxQ_Set() TCP/UDP

Set socket's connection receive queue timeout value.

Files

net_sock.h/net_sock.c

Prototype

```
void  NetSock_CfgTimeoutRxQ_Set(NET_SOCK_ID   sock_id,
                                CPU_INT32U    timeout_ms,
                                NET_ERR      *perr);
```

Arguments

sock_id This is the socket ID returned by NetSock_Open()/socket() when the socket was created **OR** by NetSock_Accept()/accept() when a connection was accepted.

timeout_ms Desired timeout value:
NET_TMR_TIME_INFINITE, if infinite (i.e. **NO** timeout) value desired.
In number of milliseconds, otherwise.

perr Pointer to variable that will receive the return error code from this function:
NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_INVALID_SOCK
NET_SOCK_ERR_INVALID_TYPE
NET_SOCK_ERR_INVALID_PROTOCOL
NET_TCP_ERR_CONN_NOT_USED
NET_TCP_ERR_INVALID_CONN
NET_CONN_ERR_NOT_USED
NET_CONN_ERR_INVALID_CONN
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_INVALID_TIME
NET_OS_ERR_LOCK

Returned Value

None.

Required Configuration

Available only if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01).

Notes / Warnings

None.

A.12.15 NetSock_CfgTimeoutTxQ_Dflt() TCP

Set socket's connection transmit queue timeout to configured-default value.

Files

net_sock.h/net_sock.c

Prototype

```
void  NetSock_CfgTimeoutTxQ_Dflt(NET_SOCK_ID   sock_id,
                                 NET_ERR      *perr);
```

Arguments

sock_id This is the socket ID returned by NetSock_Open()/socket() when the socket was created **OR** by NetSock_Accept()/accept() when a connection was accepted.

perr Pointer to variable that will receive the return error code from this function:

NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_INVALID_SOCK
NET_SOCK_ERR_INVALID_TYPE
NET_SOCK_ERR_INVALID_PROTOCOL
NET_TCP_ERR_CONN_NOT_USED
NET_TCP_ERR_INVALID_CONN
NET_CONN_ERR_NOT_USED
NET_CONN_ERR_INVALID_CONN
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_INVALID_TIME
NET_OS_ERR_LOCK

Returned Value

None.

Required Configuration

Available only if NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12).

Notes / Warnings

None.

A.12.16 NetSock_CfgTimeoutTxQ_Get_ms() TCP

Get socket's transmit queue timeout value.

Files

net_sock.h/net_sock.c

Prototype

```
CPU_INT32U  NetSock_CfgTimeoutTxQ_Get_ms(NET_SOCK_ID   sock_id,
                                         NET_ERR      *perr);
```

Arguments

sock_id This is the socket ID returned by NetSock_Open()/socket() when the socket was created **OR** by NetSock_Accept()/accept() when a connection was accepted.

perr Pointer to variable that will receive the return error code from this function:

NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_INVALID_SOCK
NET_SOCK_ERR_INVALID_TYPE
NET_SOCK_ERR_INVALID_PROTOCOL
NET_TCP_ERR_CONN_NOT_USED
NET_TCP_ERR_INVALID_CONN
NET_CONN_ERR_NOT_USED
NET_CONN_ERR_INVALID_CONN
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_LOCK

Returned Value

0, on ANY errors;
NET_TMR_TIME_INFINITE, if infinite (i.e. **NO** timeout) value configured;
Timeout in number of milliseconds, otherwise.

Required Configuration

Available only if NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12).

Notes / Warnings

None.

A.12.17 NetSock_CfgTimeoutTxQ_Set() TCP

Set socket's connection transmit queue timeout value.

Files

net_sock.h/net_sock.c

Prototype

```
void  NetSock_CfgTimeoutTxQ_Set(NET_SOCK_ID   sock_id,
                                CPU_INT32U    timeout_ms,
                                NET_ERR      *perr);
```

Arguments

sock_id — This is the socket ID returned by NetSock_Open()/socket() when the socket was created **OR** by NetSock_Accept()/accept() when a connection was accepted.

timeout_ms — Desired timeout value:
NET_TMR_TIME_INFINITE, if infinite (i.e. **NO** timeout) value desired.
In number of milliseconds, otherwise.

perr — Pointer to variable that will receive the return error code from this function:
NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_INVALID_SOCK
NET_SOCK_ERR_INVALID_TYPE
NET_SOCK_ERR_INVALID_PROTOCOL
NET_TCP_ERR_CONN_NOT_USED
NET_TCP_ERR_INVALID_CONN
NET_CONN_ERR_NOT_USED
NET_CONN_ERR_INVALID_CONN
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_INVALID_TIME
NET_OS_ERR_LOCK

Returned Value

None.

Required Configuration

Available only if `NET_CFG_TRANSPORT_LAYER_SEL` is configured for TCP (see Section 3.12).

Notes / Warnings

None.

A.12.18 NetSock_Close() / close() TCP/UDP

Terminate communication and free a socket.

Files

net_sock.h/net_sock.c
net_bsd.h/net_bsd.c

Prototypes

```
NET_SOCK_RTN_CODE  NetSock_Close(NET_SOCK_ID   sock_id,
                                 NET_ERR      *perr);

int  close(int  sock_id);
```

Arguments

sock_id — This is the socket ID returned by NetSock_Open()/socket() when the socket was created **OR** by NetSock_Accept()/accept() when a connection was accepted.

perr — Pointer to variable that will receive the return error code from this function:

NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_CLOSED
NET_SOCK_ERR_INVALID_SOCK
NET_SOCK_ERR_INVALID_FAMILY
NET_SOCK_ERR_INVALID_STATE
NET_SOCK_ERR_CLOSE_IN_PROGRESS
NET_SOCK_ERR_CONN_SIGNAL_TIMEOUT
NET_SOCK_ERR_CONN_FAIL
NET_SOCK_ERR_FAULT
NET_CONN_ERR_NULL_PTR
NET_CONN_ERR_NOT_USED
NET_CONN_ERR_INVALID_CONN
NET_CONN_ERR_INVALID_ADDR_LEN
NET_CONN_ERR_ADDR_IN_USE
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_LOCK

Returned Value

NET_SOCK_BSD_ERR_NONE/0, if successful;
NET_SOCK_BSD_ERR_CLOSE/-1, otherwise.

Required Configuration

NetSock_Close() is available only if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01).

In addition, close() is available only if NET_BSD_CFG_API_EN is enabled (see Section 3.15.12).

Notes / Warnings

After closing a socket, NO further operations should be performed with the socket.

A.12.19 NetSock_Conn() / connect() TCP/UDP

Connect a local socket to a remote socket address. If the local socket was not previously bound to a local address and port, the socket is bound to the default interface's default address and a random port number. When successful, a connected socket has access to both local and remote socket addresses.

Although both UDP and TCP sockets may both connect to remote servers or hosts, UDP and TCP connections are inherently different:

For TCP sockets, `NetSock_Conn()/connect()` returns successfully only after completing the three-way TCP handshake with the remote TCP host. Success implies the existence of a dedicated connection to the remote socket similar to a telephone connection. This dedicated connection is maintained for the life of the connection until one or both sides close the connection.

For UDP sockets, `NetSock_Conn()/connect()` merely saves the remote socket's address for the local socket for convenience. All UDP datagrams from the socket will be transmitted to the remote socket. This pseudo-connection is not permanent and may be re-configured at any time.

Files

net_sock.h/net_sock.c
net_bsd.h/net_bsd.c

Prototypes

```
NET_SOCK_RTN_CODE  NetSock_Connect(NET_SOCK_ID         sock_id,
                                   NET_SOCK_ADDR      *paddr_remote,
                                   NET_SOCK_ADDR_LEN   addr_len,
                                   NET_ERR            *perr);

int  connect(        int         sock_id,
             struct  sockaddr   *paddr_remote,
                     socklen_t   addr_len);
```

Arguments

sock_id This is the socket ID returned by `NetSock_Open()/socket()` when the socket was created.

paddr_remote Pointer to a socket address structure (see Section 6.01) which contains the remote socket address to connect the socket to.

`addr_len` Size of the socket address structure which **MUST** be passed the size of the socket address structure [e.g. `sizeof(NET_SOCK_ADDR_IP)`].

`perr` Pointer to variable that will receive the return error code from this function:

```
NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_CLOSED
NET_SOCK_ERR_INVALID_SOCK
NET_SOCK_ERR_INVALID_FAMILY
NET_SOCK_ERR_INVALID_PROTOCOL
NET_SOCK_ERR_INVALID_TYPE
NET_SOCK_ERR_INVALID_STATE
NET_SOCK_ERR_INVALID_OP
NET_SOCK_ERR_INVALID_ADDR
NET_SOCK_ERR_INVALID_ADDR_LEN
NET_SOCK_ERR_ADDR_IN_USE
NET_SOCK_ERR_PORT_NBR_NONE_AVAIL
NET_SOCK_ERR_CONN_SIGNAL_TIMEOUT
NET_SOCK_ERR_CONN_IN_USE
NET_SOCK_ERR_CONN_FAIL
NET_SOCK_ERR_FAULT
NET_IF_ERR_INVALID_IF
NET_IP_ERR_ADDR_NONE_AVAIL
NET_IP_ERR_ADDR_CFG_IN_PROGRESS
NET_CONN_ERR_NULL_PTR
NET_CONN_ERR_NOT_USED
NET_CONN_ERR_NONE_AVAIL
NET_CONN_ERR_INVALID_CONN
NET_CONN_ERR_INVALID_FAMILY
NET_CONN_ERR_INVALID_TYPE
NET_CONN_ERR_INVALID_PROTOCOL_IX
NET_CONN_ERR_INVALID_ADDR_LEN
NET_CONN_ERR_ADDR_NOT_USED
NET_CONN_ERR_ADDR_IN_USE
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_LOCK
```

Returned Value

NET_SOCK_BSD_ERR_NONE/0, if successful;
NET_SOCK_BSD_ERR_CONN/-1, otherwise.

Required Configuration

NetSock_Conn() is available only if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01).

In addition, connect() is available only if NET_BSD_CFG_API_EN is enabled (see Section 3.15.12).

Notes / Warnings

See Section 6.01 for socket address structure formats.

A.12.20 NET_SOCK_DESC_CLR() / FD_CLR() TCP/UDP

Remove a socket file descriptor ID as a member of a file descriptor set.

See also Appendix A.12.32.

Files

`net_sock.h`

Prototype

```
NET_SOCK_DESC_CLR(desc_nbr, pdesc_set);
```

Arguments

`desc_nbr` This is the socket file descriptor ID returned by `NetSock_Open()`/`socket()` when the socket was created **OR** by `NetSock_Accept()`/`accept()` when a connection was accepted.

`pdesc_set` Pointer to a socket file descriptor set.

Returned Value

None.

Required Configuration

Available only if either `NET_CFG_TRANSPORT_LAYER_SEL` is configured for TCP (see Section 3.12) and/or `NET_UDP_CFG_APP_API_SEL` is configured for sockets (see Section 3.13.01) **AND** if `NET_SOCK_CFG_SEL_EN` is enabled (see Section 3.15.04).

In addition, `FD_CLR()` is available only if `NET_BSD_CFG_API_EN` is enabled (see Section 3.15.12).

Notes / Warnings

`NetSock_Sel()`/`select()` checks or waits for available operations or error conditions on any of the socket file descriptor members of a socket file descriptor set.

No errors are returned even if the socket file descriptor ID or the file descriptor set is invalid, or the socket file descriptor ID is **NOT** set in the file descriptor set.

A.12.21 **NET_SOCK_DESC_COPY() TCP/UDP**

Copy a file descriptor set to another file descriptor set.

See also Appendix A.12.32.

Files

`net_sock.h`

Prototype

```
NET_SOCK_DESC_COPY(pdesc_set_dest, pdesc_set_src);
```

Arguments

`pdesc_set_dest`	Pointer to the destination socket file descriptor set.
`pdesc_set_src`	Pointer to the source socket file descriptor set to copy.

Returned Value

None.

Required Configuration

Available only if either `NET_CFG_TRANSPORT_LAYER_SEL` is configured for TCP (see Section 3.12) and/or `NET_UDP_CFG_APP_API_SEL` is configured for sockets (see Section 3.13.01) AND if `NET_SOCK_CFG_SEL_EN` is enabled (see Section 3.15.04).

Notes / Warnings

`NetSock_Sel()`/`select()` checks or waits for available operations or error conditions on any of the socket file descriptor members of a socket file descriptor set.

No errors are returned even if either file descriptor set is invalid.

A.12.22 NET_SOCK_DESC_ INIT() / FD_ZERO() TCP/UDP

Initialize/zero-clear a file descriptor set.

See also Appendix A.12.32.

Files

net_sock.h

Prototype

```
NET_SOCK_DESC_INIT(pdesc_set);
```

Arguments

pdesc_set   Pointer to a socket file descriptor set.

Returned Value

None.

Required Configuration

Available only if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01) AND if NET_SOCK_CFG_SEL_EN is enabled (see Section 3.15.04).

In addition, FD_ZERO() is available only if NET_BSD_CFG_API_EN is enabled (see Section 3.15.12).

Notes / Warnings

NetSock_Sel()/select() checks or waits for available operations or error conditions on any of the socket file descriptor members of a socket file descriptor set.

No errors are returned even if the file descriptor set is invalid.

A.12.23 NET_SOCK_DESC_IS_SET() / FD_IS_SET() TCP/UDP

Check if a socket file descriptor ID is a member of a file descriptor set.

See also Appendix A.12.32.

Files

`net_sock.h`

Prototype

```
NET_SOCK_DESC_IS_SET(desc_nbr, pdesc_set);
```

Arguments

`desc_nbr` This is the socket file descriptor ID returned by `NetSock_Open()`/`socket()` when the socket was created **OR** by `NetSock_Accept()`/`accept()` when a connection was accepted.

`pdesc_set` Pointer to a socket file descriptor set.

Returned Value

`1`, if the socket file descriptor ID is a member of the file descriptor set;
`0`, otherwise.

Required Configuration

Available only if either `NET_CFG_TRANSPORT_LAYER_SEL` is configured for TCP (see Section 3.12) and/or `NET_UDP_CFG_APP_API_SEL` is configured for sockets (see Section 3.13.01) **AND** if `NET_SOCK_CFG_SEL_EN` is enabled (see Section 3.15.04).

In addition, `FD_IS_SET()` is available only if `NET_BSD_CFG_API_EN` is enabled (see Section 3.15.12).

Notes / Warnings

`NetSock_Sel()`/`select()` checks or waits for available operations or error conditions on any of the socket file descriptor members of a socket file descriptor set.

`0` is returned if the socket file descriptor ID or the file descriptor set is invalid.

A.12.24 NET_SOCK_DESC_SET() / FD_SET() TCP/UDP

Add a socket file descriptor ID as a member of a file descriptor set.

See also Appendix A.12.32.

Files

net_sock.h

Prototype

```
NET_SOCK_DESC_SET(desc_nbr, pdesc_set);
```

Arguments

desc_nbr This is the socket file descriptor ID returned by NetSock_Open()/socket() when the socket was created **OR** by NetSock_Accept()/accept() when a connection was accepted.

pdesc_set Pointer to a socket file descriptor set.

Returned Value

None.

Required Configuration

Available only if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01) **AND** if NET_SOCK_CFG_SEL_EN is enabled (see Section 3.15.04).

In addition, FD_SET() is available only if NET_BSD_CFG_API_EN is enabled (see Section 3.15.12).

Notes / Warnings

NetSock_Sel()/select() checks or waits for available operations or error conditions on any of the socket file descriptor members of a socket file descriptor set.

No errors are returned even if the socket file descriptor ID or the file descriptor set is invalid, or the socket file descriptor ID is **NOT** cleared in the file descriptor set.

A.12.25 NetSock_GetConnTransportID()

Gets a socket's transport layer connection handle ID (e.g. TCP connection ID) if available.

Files

net_sock.h/net_sock.c

Prototype

```
NET_CONN_ID  NetSock_GetConnTransportID(NET_SOCK_ID   sock_id,
                                        NET_ERR      *perr);
```

Arguments

sock_id — This is the socket ID returned by NetSock_Open()/socket() when the socket was created **OR** by NetSock_Accept()/accept() when a connection was accepted.

perr — Pointer to variable that will receive the return error code from this function:

NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_INVALID_SOCK
NET_SOCK_ERR_INVALID_TYPE
NET_CONN_ERR_NOT_USED
NET_CONN_ERR_INVALID_CONN
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_LOCK

Returned Value

Socket's transport connection handle ID (e.g. TCP connection ID), if **NO** errors;
NET_CONN_ID_NONE, otherwise.

Required Configuration

Available only if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01).

Notes / Warnings

None.

A.12.26 NetSock_IsConn()

Check if a socket is connected to a remote socket.

Files

net_sock.h/net_sock.c

Prototype

```
CPU_BOOLEAN  NetSock_IsConn(NET_SOCK_ID   sock_id,
                            NET_ERR      *perr);
```

Arguments

sock_id This is the socket ID returned by NetSock_Open()/socket() when the socket was created **OR** by NetSock_Accept()/accept() when a connection was accepted.

perr Pointer to variable that will receive the return error code from this function:

NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_INVALID_SOCK
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_LOCK

Returned Value

DEF_YES, if the socket is valid and connected;
DEF_NO, otherwise.

Required Configuration

Available only if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01).

Notes / Warnings

None.

A.12.27 **NetSock_Listen() / listen() TCP**

Set a socket to accept incoming connections. The socket must already be bound to a local address. If successful, incoming TCP connection requests addressed to the socket's local address will be queued until accepted by the socket (see Section A.12.01).

Files

net_sock.h/net_sock.c
net_bsd.h/net_bsd.c

Prototypes

```
NET_SOCK_RTN_CODE  NetSock_Listen(NET_SOCK_ID       sock_id,
                                  NET_SOCK_Q_SIZE   sock_q_size,
                                  NET_ERR          *perr);
int  listen(int  sock_id,
            int  sock_q_size);
```

Arguments

sock_id — This is the socket ID returned by NetSock_Open()/socket() when the socket was created.

sock_q_size — Maximum number of new connections allowed to be waiting. In other words, this argument specifies the maximum queue length of pending connections while the listening socket is busy servicing the current request.

perr — Pointer to variable that will receive the return error code from this function:

NET_SOCK_ERR_NONE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_CLOSED
NET_SOCK_ERR_INVALID_SOCK
NET_SOCK_ERR_INVALID_FAMILY
NET_SOCK_ERR_INVALID_PROTOCOL
NET_SOCK_ERR_INVALID_TYPE
NET_SOCK_ERR_INVALID_STATE
NET_SOCK_ERR_INVALID_OP
NET_SOCK_ERR_CONN_FAIL
NET_CONN_ERR_NOT_USED
NET_CONN_ERR_INVALID_CONN
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_LOCK

Returned Value

NET_SOCK_BSD_ERR_NONE/0, if successful;
NET_SOCK_BSD_ERR_LISTEN/-1, otherwise.

Required Configuration

NetSock_Listen() is available only if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01).

In addition, listen() is available only if NET_BSD_CFG_API_EN is enabled (see Section 3.15.12).

Notes / Warnings

Available only for stream-type sockets (e.g. TCP sockets).

A.12.28 **NetSock_Open() / socket() TCP/UDP**

Create a datagram (i.e. UDP) or stream (i.e. TCP) type socket.

Files

net_sock.h/net_sock.c

net_bsd.h/net_bsd.c

Prototypes

```
NET_SOCK_ID  NetSock_Open(NET_SOCK_PROTOCOL_FAMILY   protocol_family,
                          NET_SOCK_TYPE              sock_type,
                          NET_SOCK_PROTOCOL          protocol,
                          NET_ERR                   *perr);

int  socket(int  protocol_family,
            int  sock_type,
            int  protocol);
```

Arguments

`protocol_family` This field establishes the socket *protocol family domain*. Always use `NET_SOCK_FAMILY_IP_V4/PF_INET` for TCP/IP sockets.

`sock_type` Socket type:

- `NET_SOCK_TYPE_DATAGRAM/PF_DGRAM` .... for datagram sockets (i.e. UDP)
- `NET_SOCK_TYPE_STREAM/PF_STREAM` .... for stream sockets (i.e. TCP)

`NET_SOCK_TYPE_DATAGRAM` sockets preserve message boundaries. Applications that exchange single request and response messages are examples of datagram communication.

`NET_SOCK_TYPE_STREAM` sockets provides a reliable byte-stream connection, where bytes are received from the remote application in the same order as they were sent. File transfer and terminal emulation are examples of applications that require this type of protocol.

`protocol` Socket protocol:

NET_SOCK_PROTOCOL_UDP/IPPROTO_UDP .... for UDP
NET_SOCK_PROTOCOL_TCP/IPPROTO_TCP .... for TCP
0 .... for default-protocol:
UDP for NET_SOCK_TYPE_DATAGRAM/PF_DGRAM
TCP for NET_SOCK_TYPE_DATAGRAM/PF_DGRAM

`perr` Pointer to variable that will receive the return error code from this function:

NET_SOCK_ERR_NONE
NET_SOCK_ERR_NONE_AVAIL
NET_SOCK_ERR_INVALID_FAMILY
NET_SOCK_ERR_INVALID_PROTOCOL
NET_SOCK_ERR_INVALID_TYPE
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_LOCK

The table below shows you the different ways you can specify the three arguments.

TCP/IP PROTOCOL	ARGUMENTS		
	protocol_family	sock_type	protocol
UDP	NET_SOCK_FAMILY_IP_V4	NET_SOCK_TYPE_DATAGRAM	NET_SOCK_PROTOCOL_UDP
UDP	NET_SOCK_FAMILY_IP_V4	NET_SOCK_TYPE_DATAGRAM	0
TCP	NET_SOCK_FAMILY_IP_V4	NET_SOCK_TYPE_STREAM	NET_SOCK_PROTOCOL_TCP
TCP	NET_SOCK_FAMILY_IP_V4	NET_SOCK_TYPE_STREAM	0

Returned Value

Returns a non-negative socket descriptor ID for the new socket connection, if successful; NET_SOCK_BSD_ERR_OPEN/-1 otherwise.

Required Configuration

NetSock_Open() is available only if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01).

In addition, socket() is available only if NET_BSD_CFG_API_EN is enabled (see Section 3.15.12).

Notes / Warnings

The family, type, and protocol of a socket is fixed once a socket is created. In other words, you cannot change a TCP stream socket to a UDP datagram socket (or vice versa) at run-time.

To connect two sockets, both sockets must share the same socket family, type, and protocol.

A.12.29 NetSock_PoolStatGet()

Get Network Sockets' statistics pool.

Files

net_sock.h/net_sock.c

Prototype

```
NET_STAT_POOL  NetSock_PoolStatGet(void);
```

Arguments

None.

Returned Value

Network Sockets' statistics pool, if NO errors;
NULL statistics pool, otherwise.

Required Configuration

Available only if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01).

Notes / Warnings

None.

A.12.30 NetSock_PoolStatResetMaxUsed()

Reset Network Sockets' statistics pool's maximum number of entries used.

Files

net_sock.h/net_sock.c

Prototype

```
void  NetSock_PoolStatResetMaxUsed(void);
```

Arguments

None.

Returned Value

None.

Required Configuration

Available only if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01).

Notes / Warnings

None.

A.12.31 NetSock_RxData() / recv() TCP/UDP NetSock_RxDataFrom() / recvfrom() TCP/UDP

Copy up to a specified number of bytes received from a remote socket into an application memory buffer.

Files

net_sock.h/net_sock.c
net_bsd.h/net_bsd.c

Prototypes

```
NET_SOCK_RTN_CODE NetSock_RxData(NET_SOCK_ID             sock_id,
                                 void                   *pdata_buf,
                                 CPU_INT16U              data_buf_len,
                                 CPU_INT16S              flags,
                                 NET_ERR                *perr);

NET_SOCK_RTN_CODE NetSock_RxDataFrom(NET_SOCK_ID         sock_id,
                                     void               *pdata_buf,
                                     CPU_INT16U          data_buf_len,
                                     CPU_INT16S          flags,
                                     NET_SOCK_ADDR      *paddr_remote,
                                     NET_SOCK_ADDR_LEN  *paddr_len,
                                     void               *pip_opts_buf,
                                     CPU_INT08U          ip_opts_buf_len,
                                     CPU_INT08U         *pip_opts_len,
                                     NET_ERR            *perr);

ssize_t  recv(int      sock_id,
              void    *pdata_buf,
             _size_t   data_buf_len,
              int      flags);

ssize_t  recvfrom(    int             sock_id,
                      void           *pdata_buf,
                     _size_t          data_buf_len,
                      int             flags,
             struct  sockaddr        *paddr_remote,
                     socklen_t       *paddr_len);
```

Arguments

`sock_id`	This is the socket ID returned by `NetSock_Open()`/`socket()` when the socket was created **OR** by `NetSock_Accept()`/`accept()` when a connection was accepted.
`pdata_buf`	Pointer to the application memory buffer to receive data.
`data_buf_len`	Size of the destination application memory buffer (in bytes).

`flags`	Specifies receive options. You can logically OR the flags. Valid flags are:
	`NET_SOCK_FLAG_NONE/0` .... No socket flags selected
	`NET_SOCK_FLAG_RX_DATA_PEEK/`
	`MSG_PEEK`.......................... Receive socket data without consuming it
	`NET_SOCK_FLAG_RX_NO_BLOCK/`
	`MSG_DONTWAIT`..................... Receive socket data without blocking
	
	In most cases, you would set `flags` to `NET_SOCK_FLAG_NONE/0`.

`paddr_remote`	Pointer to a socket address structure (see Section 6.01) to return the remote host address that sent the received data.
`paddr_len`	Pointer to the size of the socket address structure which **MUST** be passed the size of the socket address structure [e.g. `sizeof(NET_SOCK_ADDR_IP)`]. Returns size of the accepted connection's socket address structure.
`pip_opts_buf`	Pointer to buffer to receive possible IP options.
`pip_opts_len`	Pointer to variable that will receive the return size of any received IP options.

`perr`	Pointer to variable that will receive the return error code from this function:
	`NET_SOCK_ERR_NONE`
	`NET_SOCK_ERR_NULL_PTR`
	`NET_SOCK_ERR_NULL_SIZE`
	`NET_SOCK_ERR_NOT_USED`
	`NET_SOCK_ERR_CLOSED`
	`NET_SOCK_ERR_INVALID_SOCK`
	`NET_SOCK_ERR_INVALID_FAMILY`

NET_SOCK_ERR_INVALID_PROTOCOL
NET_SOCK_ERR_INVALID_TYPE
NET_SOCK_ERR_INVALID_STATE
NET_SOCK_ERR_INVALID_OP
NET_SOCK_ERR_INVALID_FLAG
NET_SOCK_ERR_INVALID_ADDR_LEN
NET_SOCK_ERR_INVALID_DATA_SIZE
NET_SOCK_ERR_CONN_FAIL
NET_SOCK_ERR_FAULT
NET_SOCK_ERR_RX_Q_EMPTY
NET_SOCK_ERR_RX_Q_CLOSED
NET_ERR_RX
NET_CONN_ERR_NULL_PTR
NET_CONN_ERR_NOT_USED
NET_CONN_ERR_INVALID_CONN
NET_CONN_ERR_INVALID_ADDR_LEN
NET_CONN_ERR_ADDR_NOT_USED
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_LOCK

Returned Value

Positive number of bytes received, if successful;
NET_SOCK_BSD_RTN_CODE_CONN_CLOSED/0, if the socket is closed;
NET_SOCK_BSD_ERR_RX/-1, otherwise.

Blocking vs Non-Blocking

The default setting for **µC/TCP-IP** is blocking. However, you can change that at compile time by setting the NET_SOCK_CFG_BLOCK_SEL (see Section 3.15.04) to one of the following values:

NET_SOCK_BLOCK_SEL_DFLT sets blocking mode to the default, or blocking, unless modified by run-time options.

NET_SOCK_BLOCK_SEL_BLOCK sets the blocking mode to blocking. This means that a socket receive function will wait forever, until at least one byte of data is available to return or the socket connection is closed, unless a timeout is specified by NetSock_CfgTimeoutRxQ_Set() [see Section A.12.14].

NET_SOCK_BLOCK_SEL_NO_BLOCK sets the blocking mode to non-blocking. This means that a socket receive function will NOT wait but immediately return either any available data, socket connection closed, or an error indicating no available data or other possible socket faults. Your application will have to 'poll' the socket on a regular basis to receive data.

The current version of **µC/TCP-IP** selects blocking or non-blocking at compile time for all sockets. A future version may allow the selection of blocking or non-blocking at the individual socket level. However, each socket receive call can pass the `NET_SOCK_FLAG_RX_NO_BLOCK/MSG_DONTWAIT` flag to disable blocking on that call.

Required Configuration

`NetSock_RxData()/NetSock_RxDataFrom()` is available only if either `NET_CFG_TRANSPORT_LAYER_SEL` is configured for TCP (see Section 3.12) and/or `NET_UDP_CFG_APP_API_SEL` is configured for sockets (see Section 3.13.01).

In addition, `recv()/recvfrom()` is available only if `NET_BSD_CFG_API_EN` is enabled (see Section 3.15.12).

Notes / Warnings

TCP sockets typically use `NetSock_RxData()/recv()`, whereas UDP sockets typically use `NetSock_RxDataFrom()/recvfrom()`.

For stream sockets (i.e. TCP), bytes are guaranteed to be received in the same order as they were transmitted, without omissions.

For datagram sockets (i.e. UDP), each receive returns the data from exactly one send but datagram order and delivery is not guaranteed. Also, if the application memory buffer is not big enough to receive an entire datagram, the datagram's data is truncated to the size of the memory buffer and the remaining data is discarded.

Only some receive flag options are implemented. If other flag options are requested, an error is returned so that flag options are **NOT** silently ignored.

A.12.32 **NetSock_Sel() / select() TCP/UDP**

Check if any sockets are ready for available read or write operations or error conditions.

Files

net_sock.h/net_sock.c
net_bsd.h/net_bsd.c

Prototypes

```
NET_SOCK_RTN_CODE  NetSock_Sel(NET_SOCK_QTY       sock_nbr_max,
                               NET_SOCK_DESC     *psock_desc_rd,
                               NET_SOCK_DESC     *psock_desc_wr,
                               NET_SOCK_DESC     *psock_desc_err,
                               NET_SOCK_TIMEOUT  *ptimeout,
                               NET_ERR           *perr);

int  select(        int       desc_nbr_max,
            struct  fd_set   *pdesc_rd,
            struct  fd_set   *pdesc_wr,
            struct  fd_set   *pdesc_err,
            struct  timeval  *ptimeout);
```

Arguments

sock_nbr_max — Specifies the maximum number of socket file descriptors in the file descriptor sets.

psock_desc_rd — Pointer to a set of socket file descriptors to :
(a) Check for available read operations.
(b) (1) Return the actual socket file descriptors ready for available read operations, if no errors;
(2) Return the initial, non-modified set of socket file descriptors, on any errors;
(3) Return a null-valued (i.e. zero-cleared) descriptor set, if any timeout expires.

psock_desc_wr — Pointer to a set of socket file descriptors to :
(a) Check for available write operations.
(b) (1) Return the actual socket file descriptors ready for available write operations, if no errors;
(2) Return the initial, non-modified set of socket file descriptors, on any errors;

(3) Return a null-valued (i.e. zero-cleared) descriptor set, if any timeout expires.

`psock_desc_err` Pointer to a set of socket file descriptors to :
(a) Check for any available socket errors.
(b) (1) Return the actual socket file descriptors ready with any pending errors;
(2) Return the initial, non-modified set of socket file descriptors, on any errors;
(3) Return a null-valued (i.e. zero-cleared) descriptor set, if any timeout expires.

`ptimeout` Pointer to a timeout argument.

`perr` Pointer to variable that will receive the return error code from this function:
NET_SOCK_ERR_NONE
NET_SOCK_ERR_TIMEOUT
NET_ERR_INIT_INCOMPLETE
NET_SOCK_ERR_INVALID_DESC
NET_SOCK_ERR_INVALID_TIMEOUT
NET_SOCK_ERR_INVALID_SOCK
NET_SOCK_ERR_INVALID_TYPE
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_EVENTS_NBR_MAX
NET_OS_ERR_LOCK

Returned Value

Returns the number of sockets ready with available operations, if successful;
NET_SOCK_BSD_RTN_CODE_TIMEOUT/0, upon timeout;
NET_SOCK_BSD_ERR_SEL/-1, otherwise.

Required Configuration

`NetSock_Sel()` is available only if either NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12) and/or NET_UDP_CFG_APP_API_SEL is configured for sockets (see Section 3.13.01) **AND** if NET_SOCK_CFG_SEL_EN is enabled (see Section 3.15.04).

In addition, `select()` is available only if NET_BSD_CFG_API_EN is enabled (see Section 3.15.12).

Notes / Warnings

Supports socket file descriptors **ONLY** (i.e. socket ID numbers).

Use descriptor macro's to prepare and decode socket file descriptor sets (see Sections A.12.20 – A.12.24).

See '`net_sock.c  NetSock_Sel()` Note #3' for more details.

A.12.33 NetSock_TxData() / send() TCP/UDP NetSock_TxDataTo() / sendto() TCP/UDP

Copy bytes from an application memory buffer into a socket to send to a remote socket.

Files

net_sock.h/net_sock.c
net_bsd.h/net_bsd.c

Prototypes

```
NET_SOCK_RTN_CODE  NetSock_TxData(NET_SOCK_ID    sock_id,
                                  void          *p_data,
                                  CPU_INT16U     data_len,
                                  CPU_INT16S     flags,
                                  NET_ERR       *perr);

NET_SOCK_RTN_CODE  NetSock_TxDataTo(NET_SOCK_ID         sock_id,
                                    void               *p_data,
                                    CPU_INT16U          data_len,
                                    CPU_INT16S          flags,
                                    NET_SOCK_ADDR      *paddr_remote,
                                    NET_SOCK_ADDR_LEN   addr_len,
                                    NET_ERR            *perr);

ssize_t  send (int       sock_id,
               void     *p_data,
              _size_t    data_len,
               int       flags);

ssize_t  sendto(        int         sock_id,
                        void       *p_data,
                       _size_t      data_len,
                        int         flags,
                struct  sockaddr   *paddr_remote,
                        socklen_t   addr_len);
```

Arguments

sock_id — This is the socket ID returned by NetSock_Open()/socket() when the socket was created **OR** by NetSock_Accept()/accept() when a connection was accepted.

`p_data` Pointer to the application data memory buffer to send.

`data_len` Size of the application data memory buffer (in bytes).

`flags` Specifies transmit options. You can logically OR the flags. Valid flags are:

`NET_SOCK_FLAG_NONE/0` .... No socket flags selected
`NET_SOCK_FLAG_TX_NO_BLOCK/`
`MSG_DONTWAIT` .... Send socket data without blocking

In most cases, you would set flags to `NET_SOCK_FLAG_NONE/0`.

`paddr_remote` Pointer to a socket address structure (see Section 6.01) which contains the remote socket address to send the data to.

`addr_len` Size of the socket address structure which **MUST** be passed the size of the socket address structure [e.g. `sizeof(NET_SOCK_ADDR_IP)`].

`perr` Pointer to variable that will receive the return error code from this function:

```
NET_SOCK_ERR_NONE
NET_SOCK_ERR_NULL_PTR
NET_SOCK_ERR_NOT_USED
NET_SOCK_ERR_CLOSED
NET_SOCK_ERR_INVALID_SOCK
NET_SOCK_ERR_INVALID_FAMILY
NET_SOCK_ERR_INVALID_PROTOCOL
NET_SOCK_ERR_INVALID_TYPE
NET_SOCK_ERR_INVALID_STATE
NET_SOCK_ERR_INVALID_OP
NET_SOCK_ERR_INVALID_FLAG
NET_SOCK_ERR_INVALID_DATA_SIZE
NET_SOCK_ERR_INVALID_CONN
NET_SOCK_ERR_INVALID_ADDR
NET_SOCK_ERR_INVALID_ADDR_LEN
NET_SOCK_ERR_INVALID_PORT_NBR
NET_SOCK_ERR_ADDR_IN_USE
NET_SOCK_ERR_PORT_NBR_NONE_AVAIL
NET_SOCK_ERR_CONN_FAIL
NET_SOCK_ERR_FAULT
NET_ERR_TX
```

NET_IF_ERR_INVALID_IF
NET_IP_ERR_ADDR_NONE_AVAIL
NET_IP_ERR_ADDR_CFG_IN_PROGRESS
NET_CONN_ERR_NULL_PTR
NET_CONN_ERR_NOT_USED
NET_CONN_ERR_INVALID_CONN
NET_CONN_ERR_INVALID_FAMILY
NET_CONN_ERR_INVALID_TYPE
NET_CONN_ERR_INVALID_PROTOCOL_IX
NET_CONN_ERR_INVALID_ADDR_LEN
NET_CONN_ERR_ADDR_IN_USE
NET_ERR_INIT_INCOMPLETE
NET_OS_ERR_LOCK

Returned Value

Positive number of bytes (queued to be) sent, if successful;
NET_SOCK_BSD_RTN_CODE_CONN_CLOSED/0, if the socket is closed;
NET_SOCK_BSD_ERR_TX/-1, otherwise.

Note that a positive return value does not mean that the message was successfully delivered to the remote socket, just that it was sent or queued for sending.

Blocking vs Non-Blocking

The default setting for **µC/TCP-IP** is blocking. However, you can change that at compile time by setting the NET_SOCK_CFG_BLOCK_SEL (see Section 3.15.04) to one of the following values:

NET_SOCK_BLOCK_SEL_DFLT sets blocking mode to the default, or blocking, unless modified by run-time options.

NET_SOCK_BLOCK_SEL_BLOCK sets the blocking mode to blocking. This means that a socket transmit function will wait forever, until it can send (or queue to send) at least one byte of data or the socket connection is closed, unless a timeout is specified by NetSock_CfgTimeoutTxQ_Set() [see Section A.12.17].

NET_SOCK_BLOCK_SEL_NO_BLOCK sets the blocking mode to non-blocking. This means that a socket transmit function will NOT wait but immediately return as much data sent (or queued to be sent), socket connection closed, or an error indicating no available memory to send (or queue) data or other possible socket faults. Your application will have to 'poll' the socket on a regular basis to transmit data.

The current version of **µC/TCP-IP** selects blocking or non-blocking at compile time for all sockets. A future version may allow the selection of blocking or non-blocking at the individual socket level. However, each socket transmit call can pass the `NET_SOCK_FLAG_TX_NO_BLOCK/MSG_DONTWAIT` flag to disable blocking on that call.

Despite these socket-level blocking options, the current version of **µC/TCP-IP** possibly blocks at the the device driver when waiting for the availability of a device's transmitter.

Required Configuration

`NetSock_TxData()/NetSock_TxDataTo()` is available only if either `NET_CFG_TRANSPORT_LAYER_SEL` is configured for TCP (see Section 3.12) and/or `NET_UDP_CFG_APP_API_SEL` is configured for sockets (see Section 3.13.01).

In addition, `send()/sendto()` is available only if `NET_BSD_CFG_API_EN` is enabled (see Section 3.15.12).

Notes / Warnings

TCP sockets typically use `NetSock_TxData()/send()`, whereas UDP sockets typically use `NetSock_TxDataTo()/sendto()`.

For datagram sockets (i.e. UDP), each receive returns the data from exactly one send but datagram order and delivery is not guaranteed. Also, if the receive memory buffer is not big enough to receive an entire datagram, the datagram's data is truncated to the size of the memory buffer and the remaining data is discarded.

For datagram sockets (i.e. UDP), all data is sent atomically—i.e. each call to send data **MUST** be sent in a single, complete datagram. Since **µC/TCP-IP** does **NOT** currently support IP transmit fragmentation, if a datagram socket attempts to send data greater than a single datagram, then the socket send is aborted and **NO** socket data is sent.

Only some transmit flag options are implemented. If other flag options are requested, an error is returned so that flag options are **NOT** silently ignored.

A.13 TCP Functions

A.13.01 NetTCP_ConnCfgMaxSegSizeLocal()

Configure TCP connection's local maximum segment size.

Files

net_tcp.h/net_tcp.c

Prototype

```
CPU_BOOLEAN  NetTCP_ConnCfgMaxSegSizeLocal(NET_TCP_CONN_ID   conn_id_tcp,
                                           NET_TCP_SEG_SIZE  max_seg_size);
```

Arguments

conn_id_tcp	TCP connection handle ID to configure local maximum segment size.
max_seg_size	Desired maximum segment size.

Returned Value

DEF_OK,	TCP connection's local maximum segment size successfully configured, if NO errors;
DEF_FAIL,	otherwise.

Required Configuration

Available only if NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12).

Notes / Warnings

The `conn_id_tcp` argument represents the TCP connection handle — **NOT** the socket handle. The following code may be used to get the TCP connection handle and configure TCP connection parameters (see also Appendix A.12.25):

```
NET_SOCK_ID      sock_id;
NET_TCP_CONN_ID  conn_id_tcp;
NET_ERR          err;

sock_id     =  Application's TCP socket ID;  /* Get application's TCP socket     ID.    */
                                             /* Get socket's      TCP connection ID.    */
conn_id_tcp = (NET_TCP_CONN_ID)NetSock_GetConnTransportID(sock_id, &err);

if (err == NET_SOCK_ERR_NONE) {              /* If NO errors, ...                       */
    NetTCP_ConnCfg???(conn_id_tcp, ...);     /* ... configure TCP connection parameters. */
}
```

A.13.02 NetTCP_ConnCfgReTxMaxTh()

Configure TCP connection's maximum number of same segment retransmissions.

Files

net_tcp.h/net_tcp.c

Prototype

```
CPU_BOOLEAN  NetTCP_ConnCfgReTxMaxTh(NET_TCP_CONN_ID  conn_id_tcp,
                                      CPU_INT16U      nbr_max_re_tx);
```

Arguments

conn_id_tcp TCP connection handle ID to configure maximum number of same segment retransmissions.

nbr_max_re_tx Desired maximum number of same segment retransmissions.

Returned Value

DEF_OK, TCP connection's maximum number of retransmissions successfully configured, if NO errors;

DEF_FAIL, otherwise.

Required Configuration

Available only if NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12).

Notes / Warnings

The conn_id_tcp argument represents the TCP connection handle—**NOT** the socket handle. The following code may be used to get the TCP connection handle and configure TCP connection parameters (see also Appendix A.12.25):

```
NET_SOCK_ID      sock_id;
NET_TCP_CONN_ID  conn_id_tcp;
NET_ERR          err;

sock_id     =  Application's TCP socket ID;  /* Get application's TCP socket     ID.     */
                                             /* Get socket's      TCP connection ID.     */
conn_id_tcp = (NET_TCP_CONN_ID)NetSock_GetConnTransportID(sock_id, &err);

if (err == NET_SOCK_ERR_NONE) {              /* If NO errors, ...                        */
    NetTCP_ConnCfg???(conn_id_tcp, ...);     /* ... configure TCP connection parameters. */
}
```

A.13.03 NetTCP_ConnCfgReTxMaxTimeout()

Configure TCP connection's maximum retransmission timeout.

Files

net_tcp.h/net_tcp.c

Prototype

```
CPU_BOOLEAN  NetTCP_ConnCfgReTxMaxTimeout(NET_TCP_CONN_ID      conn_id_tcp,
                                          NET_TCP_TIMEOUT_SEC  timeout_sec);
```

Arguments

conn_id_tcp	TCP connection handle ID to configure retransmission maximum timeout value.
timeout_sec	Desired value for TCP connection retransmission maximum timeout (in seconds).

Returned Value

DEF_OK,	TCP connection's maximum retransmission timeout successfully configured, if NO errors;
DEF_FAIL,	otherwise.

Required Configuration

Available only if NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12).

Notes / Warnings

The `conn_id_tcp` argument represents the TCP connection handle — **NOT** the socket handle. The following code may be used to get the TCP connection handle and configure TCP connection parameters (see also Appendix A.12.25):

```
NET_SOCK_ID      sock_id;
NET_TCP_CONN_ID  conn_id_tcp;
NET_ERR          err;


sock_id     =  Application's TCP socket ID;  /* Get application's TCP socket     ID.     */
                                             /* Get socket's      TCP connection ID.     */
conn_id_tcp = (NET_TCP_CONN_ID)NetSock_GetConnTransportID(sock_id, &err);

if (err == NET_SOCK_ERR_NONE) {              /* If NO errors, ...                        */
    NetTCP_ConnCfg???(conn_id_tcp, ...);     /* ... configure TCP connection parameters. */
}
```

A.13.04 **NetTCP_ConnCfgRxWinSize()**

Configure TCP connection's receive window size.

Files

`net_tcp.h/net_tcp.c`

Prototype

```
CPU_BOOLEAN  NetTCP_ConnCfgRxWinSize(NET_TCP_CONN_ID   conn_id_tcp,
                                     NET_TCP_WIN_SIZE  win_size);
```

Arguments

`conn_id_tcp` TCP connection handle ID to configure receive window size.

`win_size` Desired receive window size.

Returned Value

`DEF_OK`, TCP connection's receive window size successfully configured, if **NO** errors;

`DEF_FAIL`, otherwise.

Required Configuration

Available only if `NET_CFG_TRANSPORT_LAYER_SEL` is configured for TCP (see Section 3.12).

Notes / Warnings

The `conn_id_tcp` argument represents the TCP connection handle—**NOT** the socket handle. The following code may be used to get the TCP connection handle and configure TCP connection parameters (see also Appendix A.12.25):

```
NET_SOCK_ID      sock_id;
NET_TCP_CONN_ID  conn_id_tcp;
NET_ERR          err;


sock_id     =  Application's TCP socket ID;  /* Get application's TCP socket     ID.     */
                                             /* Get socket's      TCP connection ID.     */
conn_id_tcp = (NET_TCP_CONN_ID)NetSock_GetConnTransportID(sock_id, &err);

if (err == NET_SOCK_ERR_NONE) {              /* If NO errors, ...                        */
    NetTCP_ConnCfg???(conn_id_tcp, ...);     /* ... configure TCP connection parameters. */
}
```

A.13.05 NetTCP_ConnCfgTxAckImmedRxdPushEn()

Configure TCP connection's transmit immediate acknowledgement for received & pushed TCP segments.

Files

net_tcp.h/net_tcp.c

Prototype

```
CPU_BOOLEAN NetTCP_ConnCfgTxAckImmedRxdPushEn(NET_TCP_CONN_ID conn_id_tcp,
                                              CPU_BOOLEAN     tx_immed_ack_en);
```

Arguments

conn_id_tcp	TCP connection handle ID to configure transmit immediate acknowledgement for received and pushed TCP segments.
tx_immed_ack_en	Desired value for TCP connection transmit immediate acknowledgement for received & pushed TCP segments:

DEF_ENABLED	TCP connection acknowledgements immediately transmitted for any pushed TCP segments received
DEF_DISABLED	TCP connection acknowledgements **NOT** immediately transmitted for any pushed TCP segments received

Returned Value

DEF_OK,	TCP connection's transmit immediate acknowledgement for received and pushed TCP segments successfully configured, if **NO** errors;
DEF_FAIL,	otherwise.

Required Configuration

Available only if NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12).

Notes / Warnings

The `conn_id_tcp` argument represents the TCP connection handle—**NOT** the socket handle. The following code may be used to get the TCP connection handle and configure TCP connection parameters (see also Appendix A.12.25):

```
NET_SOCK_ID      sock_id;
NET_TCP_CONN_ID  conn_id_tcp;
NET_ERR          err;


sock_id     =  Application's TCP socket ID;  /* Get application's TCP socket     ID.     */
                                             /* Get socket's      TCP connection ID.     */
conn_id_tcp = (NET_TCP_CONN_ID)NetSock_GetConnTransportID(sock_id, &err);

if (err == NET_SOCK_ERR_NONE) {              /* If NO errors, ...                        */
    NetTCP_ConnCfg???(conn_id_tcp, ...);     /* ... configure TCP connection parameters. */
}
```

A.13.06 NetTCP_ConnPoolStatGet()

Get TCP connections' statistics pool.

Files

net_tcp.h/net_tcp.c

Prototype

```
NET_STAT_POOL  NetTCP_ConnPoolStatGet(void);
```

Arguments

None.

Returned Value

TCP connections' statistics pool, if NO errors;
NULL statistics pool, otherwise.

Required Configuration

Available only if NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12).

Notes / Warnings

None.

A.13.07 NetTCP_ConnPoolStatResetMaxUsed()

Reset TCP connections' statistics pool's maximum number of entries used.

Files

net_tcp.h/net_tcp.c

Prototype

```
void  NetTCP_ConnPoolStatResetMaxUsed(void);
```

Arguments

None.

Returned Value

None.

Required Configuration

Available only if NET_CFG_TRANSPORT_LAYER_SEL is configured for TCP (see Section 3.12).

Notes / Warnings

None.

A.13.08 **NetTCP_InitTxSeqNbr()**

Application-defined function to initialize TCP's Initial Transmit Sequence Number Counter.

Files

`net_tcp.h/net_bsp.c`

Prototype

```
void  NetTCP_InitTxSeqNbr(void);
```

Arguments

None.

Returned Value

None.

Required Configuration

Available only if `NET_CFG_TRANSPORT_LAYER_SEL` is configured for TCP (see Section 3.12).

Notes / Warnings

The application is required to initialize TCP's Initial Transmit Sequence Number Counter.

Possible initialization methods include:

(a) Time-based initialization is one preferred method since it more appropriately provides a pseudo-random initial sequence number.

(b) Hardware-generated random number initialization is **NOT** a preferred method since it tends to produce a discrete set of pseudo-random initial sequence numbers—often the same initial sequence number.

(c) Hard-coded initial sequence number is **NOT** a preferred method since it is **NOT** random.

A.14 Network Timer Functions

A.14.01 NetTmr_PoolStatGet()

Get Network Timers' statistics pool.

Files

`net_tmr.h/net_tmr.c`

Prototype

```
NET_STAT_POOL  NetTmr_PoolStatGet(void);
```

Arguments

None.

Returned Value

Network Timers' statistics pool, if NO errors;

`NULL` statistics pool, otherwise.

Required Configuration

None.

Notes / Warnings

None.

A.14.02 NetTmr_PoolStatResetMaxUsed()

Reset Network Timers' statistics pool's maximum number of entries used.

Files

net_tmr.h/net_tmr.c

Prototype

```
void  NetTmr_PoolStatResetMaxUsed(void);
```

Arguments

None.

Returned Value

None.

Required Configuration

None.

Notes / Warnings

None.

A.15 UDP Functions

A.15.01 NetUDP_RxAppData()

Copy up to a specified number of bytes from received UDP packet buffer(s) into an application memory buffer.

Files

net_udp.h/net_udp.c

Prototype

```
CPU_INT16U        NetUDP_RxAppData(NET_BUF      *pbuf,
                                   void         *pdata_buf,
                                   CPU_INT16U    data_buf_len,
                                   CPU_INT16U    flags,
                                   void         *pip_opts_buf,
                                   CPU_INT08U    ip_opts_buf_len,
                                   CPU_INT08U   *pip_opts_len,
                                   NET_ERR      *perr);
```

Arguments

pbuf — Pointer to network buffer that received UDP datagram.

pdata_buf — Pointer to application buffer to receive application data.

data_buf_len — Size of application receive buffer (in bytes).

flags — Flags to select receive options; bit-field flags logically OR'd:

- NET_UDP_FLAG_NONE . . . . No UDP receive flags selected.
- NET_UDP_FLAG_RX_DATA_PEEK . . . Receive UDP application data without consuming the data; i.e. do **NOT** free any UDP receive packet buffer(s).

pip_opts_buf — Pointer to buffer to receive possible IP options, if **NO** errors.

`ip_opts_buf_len`	Size of IP options receive buffer (in bytes).
`pip_opts_len`	Pointer to variable that will receive the return size of any received IP options, if **NO** errors.

`perr`	Pointer to variable that will receive the return error code from this function:
	`NET_UDP_ERR_NONE`
	`NET_UDP_ERR_NULL_PTR`
	`NET_UDP_ERR_INVALID_DATA_SIZE`
	`NET_UDP_ERR_INVALID_FLAG`
	`NET_ERR_INIT_INCOMPLETE`
	`NET_ERR_RX`

Returned Value

Positive number of bytes received, if successful;
`0`, otherwise.

Required Configuration

None.

Notes / Warnings

`NetUDP_RxAppData()` **MUST** be called with the global network lock **ALREADY** acquired. Expected to be called from application's custom `NetUDP_RxAppDataHandler()` (see Section A.15.02).

Each UDP receive returns the data from exactly one send but datagram order and delivery is not guaranteed. Also, if the application memory buffer is not big enough to receive an entire datagram, the datagram's data is truncated to the size of the memory buffer and the remaining data is discarded. Therefore, the application memory buffer should be large enough to receive either the maximum UDP datagram size (i.e. 65,507 bytes) **OR** the application's expected maximum UDP datagram size.

Only some UDP receive flag options are implemented. If other flag options are requested, an error is returned so that flag options are **NOT** silently ignored.

A.15.02 NetUDP_RxAppDataHandler()

Application-defined handler to demultiplex and receive UDP packet(s) to application without sockets.

Files

net_udp.h/net_bsp.c

Prototype

```
void  NetUDP_RxAppDataHandler(NET_BUF            *pbuf,
                              NET_IP_ADDR         src_addr,
                              NET_UDP_PORT_NBR    src_port,
                              NET_IP_ADDR         dest_addr,
                              NET_UDP_PORT_NBR    dest_port,
                              NET_ERR            *perr);
```

Arguments

pbuf	Pointer to network buffer that received UDP datagram.
src_addr	Receive UDP packet's source IP address.
src_port	Receive UDP packet's source UDP port.
dest_addr	Receive UDP packet's destination IP address.
dest_port	Receive UDP packet's destination UDP port.
perr	Pointer to variable that will receive the return error code from this function: NET_APP_ERR_NONE NET_ERR_RX_DEST NET_ERR_RX

Returned Value

None.

Required Configuration

Available only if NET_UDP_CFG_APP_API_SEL is configured for application demultiplexing (see Section 3.13.01).

Notes / Warnings

NetUDP_RxAppDataHandler() **ALREADY** called with the global network lock acquired and expects to call NetUDP_RxAppData() to copy data from received UDP packets (see Section A.15.01).

If NetUDP_RxAppDataHandler() services the application data immediately within the handler function, it should do so as quickly as possible since the network's global lock remains acquired for the full duration. Thus, no other network receives or transmits can occur while NetUDP_RxAppDataHandler() executes.

NetUDP_RxAppDataHandler() may delay servicing the application data but **MUST** then:
(1) Acquire the network's global lock PRIOR to calling NetUDP_RxAppData()
(2) Release the network's global lock AFTER calling NetUDP_RxAppData()

If NetUDP_RxAppDataHandler() successfully demultiplexes the UDP packets, it should eventually call NetUDP_RxAppData() to deframe the UDP packet application data. If NetUDP_RxAppData() successfully deframes the UDP packet application data, NetUDP_RxAppDataHandler() **MUST NOT** call NetUDP_RxPktFree() to free the UDP packet's network buffer(s), since NetUDP_RxAppData() already frees the network buffer(s). And if the UDP packets were successfully demultiplexed and deframed, NetUDP_RxAppDataHandler() must return NET_APP_ERR_NONE.

However, if NetUDP_RxAppDataHandler() does **NOT** successfully demultiplex the UDP packets and therefore does **NOT** call NetUDP_RxAppData(), then NetUDP_RxAppDataHandler() should return NET_ERR_RX_DEST but must **NOT** free or discard the UDP packet network buffer(s).

But if NetUDP_RxAppDataHandler() or NetUDP_RxAppData() fails for any other reason, NetUDP_RxAppDataHandler() should call NetUDP_RxPktDiscard() to discard the UDP packet's network buffer(s) and should return NET_ERR_RX.

A.15.03 NetUDP_TxAppData()

Copy bytes from an application memory buffer to send via UDP packet(s).

Files

net_udp.h/net_udp.c

Prototype

```
CPU_INT16U  NetUDP_TxAppData(void              *p_data,
                             CPU_INT16U         data_len,
                             NET_IP_ADDR        src_addr,
                             NET_UDP_PORT_NBR   src_port,
                             NET_IP_ADDR        dest_addr,
                             NET_UDP_PORT_NBR   dest_port,
                             NET_IP_TOS         TOS,
                             NET_IP_TTL         TTL,
                             CPU_INT16U         flags_udp,
                             CPU_INT16U         flags_ip,
                             void              *popts_ip,
                             NET_ERR           *perr);
```

Arguments

p_data — Pointer to application datA.

data_len — Length of application data (in bytes).

src_addr — Source IP address.

src_port — Source UDP port.

dest_addr — Destination IP address.

dest_port — Destination UDP port.

TOS — Specific TOS to transmit UDP/IP packet.

`TTL`	Specific TTL to transmit UDP/IP packet:	
	`NET_IP_TTL_MIN` .... `1`	minimum. .... TTL transmit value
	`NET_IP_TTL_MAX` .... `255`	maximum .... TTL transmit value
	`NET_IP_TTL_DFLT`	default .... TTL transmit value
	`NET_IP_TTL_NONE` .... `0`	replace with default TTL
`flags_udp`	Flags to select UDP transmit options; bit-field flags logically OR'd:	
	`NET_UDP_FLAG_NONE`	No UDP transmit flags selected.
	`NET_UDP_FLAG_TX_CHK_SUM_DIS`	**DISABLE** UDP transmit check-sums.
	`NET_UDP_FLAG_TX_BLOCK`	Transmit UDP application data with blocking, if flag set; without blocking, if flag clear.
`flags_ip`	Flags to select IP transmit options; bit-field flags logically OR'd:	
	`NET_IP_FLAG_NONE`	**NO** IP transmit flags selected.
	`NET_IP_FLAG_TX_DONT_FRAG`	Set IP 'Don't Frag' flag.
`popts_ip`	Pointer to one or more IP options configuration data structures:	
	`NULL`	**NO** IP transmit options configuration.
	`NET_IP_OPT_CFG_ROUTE_TS`	Route &/or Internet Timestamp options configuration.
	`NET_IP_OPT_CFG_SECURITY`	Security options configuration.

`perr` Pointer to variable that will receive the return error code from this function:

```
NET_UDP_ERR_NONE
NET_UDP_ERR_NULL_PTR
NET_UDP_ERR_INVALID_DATA_SIZE
NET_UDP_ERR_INVALID_LEN_DATA
NET_UDP_ERR_INVALID_PORT_NBR
NET_UDP_ERR_INVALID_FLAG
NET_BUF_ERR_NULL_PTR
NET_BUF_ERR_NONE_AVAIL
NET_BUF_ERR_INVALID_TYPE
NET_BUF_ERR_INVALID_SIZE
NET_BUF_ERR_INVALID_IX
NET_BUF_ERR_INVALID_LEN
```

```
NET_UTIL_ERR_NULL_PTR
NET_UTIL_ERR_NULL_SIZE
NET_UTIL_ERR_INVALID_PROTOCOL
NET_ERR_TX
NET_ERR_INIT_INCOMPLETE
NET_ERR_INVALID_PROTOCOL
NET_OS_ERR_LOCK
```

Returned Value

Positive number of bytes sent, if successful;
`0`, otherwise.

Required Configuration

None.

Notes / Warnings

Each UDP datagram is sent atomically—i.e. each call to send data **MUST** be sent in a single, complete datagram. Since **µC/TCP-IP** does **NOT** currently support IP transmit fragmentation, if the application attempts to send data greater than a single UDP datagram, then the send is aborted and **NO** data is sent.

Only some UDP transmit flag options are implemented. If other flag options are requested, an error is returned so that flag options are **NOT** silently ignored.

A.16 General Network Utility Functions

A.16.01 NET_UTIL_HOST_TO_NET_16()

Convert 16-bit integer values from CPU host-order to network-order.

Files

`net_util.h`

Prototype

```
NET_UTIL_HOST_TO_NET_16(val);
```

Arguments

`val` 16-bit integer data value to convert.

Returned Value

16-bit integer value in network-order.

Required Configuration

None.

Notes / Warnings

For microprocessors that require data access to be aligned to appropriate word boundaries, `val` and any variable to receive the returned 16-bit integer **MUST** start on appropriately-aligned CPU addresses. This means that all 16-bit words **MUST** start on addresses that are multiples of 2 bytes.

A.16.02 NET_UTIL_ HOST_TO_NET_32()

Convert 32-bit integer values from CPU host-order to network-order.

Files

`net_util.h`

Prototype

```
NET_UTIL_HOST_TO_NET_32(val);
```

Arguments

`val`	32-bit integer data value to convert.

Returned Value

32-bit integer value in network-order.

Required Configuration

None.

Notes / Warnings

For microprocessors that require data access to be aligned to appropriate word boundaries, `val` and any variable to receive the returned 32-bit integer **MUST** start on appropriately-aligned CPU addresses. This means that all 32-bit words **MUST** start on addresses that are multiples of 4 bytes.

A.16.03 NET_UTIL_NET_TO_HOST_16()

Convert 16-bit integer values from network-order to CPU host- order.

Files

`net_util.h`

Prototype

```
NET_UTIL_NET_TO_HOST_16(val);
```

Arguments

`val` 16-bit integer data value to convert.

Returned Value

16-bit integer value in CPU host-order.

Required Configuration

None.

Notes / Warnings

For microprocessors that require data access to be aligned to appropriate word boundaries, `val` and any variable to receive the returned 16-bit integer **MUST** start on appropriately-aligned CPU addresses. This means that all 16-bit words **MUST** start on addresses that are multiples of 2 bytes.

A.16.04 NET_UTIL_NET_TO_HOST_32()

Convert 32-bit integer values from network-order to CPU host- order.

Files

net_util.h

Prototype

```
NET_UTIL_NET_TO_HOST_32(val);
```

Arguments

val 32-bit integer data value to convert.

Returned Value

32-bit integer value in CPU host-order.

Required Configuration

None.

Notes / Warnings

For microprocessors that require data access to be aligned to appropriate word boundaries, val and any variable to receive the returned 32-bit integer **MUST** start on appropriately-aligned CPU addresses. This means that all 32-bit words **MUST** start on addresses that are multiples of 4 bytes.

A.16.05 NetUtil_TS_Get()

Application-defined function to get the current Internet Timestamp.

Files

net_util.h/net_bsp.c

Prototype

```
NET_TS  NetUtil_TS_Get(void);
```

Arguments

None.

Returned Value

Current Internet Timestamp.

Required Configuration

None.

Notes / Warnings

RFC #791, Section 3.1 'Options : Internet Timestamp' states that "the [Internet] Timestamp is a right-justified, 32-bit timestamp in milliseconds since midnight UT [Universal Time]".

The application is responsible for providing a real-time clock with correct time-zone configuration to implement the Internet Timestamp, if possible.

A.16.06 NetUtil_TS_Get_ms()

Application-defined function to get the current millisecond timestamp.

Files

`net_util.h/net_bsp.c`

Prototype

```
NET_TS  NetUtil_TS_Get_ms(void);
```

Arguments

None.

Returned Value

Current millisecond timestamp.

Required Configuration

None.

Notes / Warnings

The application is responsible for providing a millisecond timestamp clock with adequate resolution and range to satisfy the minimum/maximum TCP RTO values (see '`net_bsp.c NetUtil_TS_Get_ms()` Note #1a').

A.17 BSD Functions

A.17.01 accept() TCP

Wait for new socket connections on a listening server socket.

See Section A.12.01 for more information.

Files

net_bsd.h/net_bsd.c

Prototype

```
int  accept(         int          sock_id,
             struct  sockaddr    *paddr_remote,
                     socklen_t   *paddr_len);
```

A.17.02 bind() TCP/UDP

Assign network addresses to sockets.

See Section A.12.02 for more information.

Files

net_bsd.h/net_bsd.c

Prototype

```
int  bind(        int          sock_id,
          struct  sockaddr    *paddr_local,
                  socklen_t    addr_len);
```

A.17.03 **close()** TCP/UDP

Terminate communication and free a socket.

See Section A.12.18 for more information.

Files

net_bsd.h/net_bsd.c

Prototype

```
int  close(int  sock_id);
```

A.17.04 **connect() TCP/UDP**

Connect a local socket to a remote socket address.

See Section A.12.19 for more information.

Files

net_bsd.h/net_bsd.c

Prototype

```
int   connect(        int          sock_id,
              struct  sockaddr    *paddr_remote,
                      socklen_t    addr_len);
```

A.17.05 **FD_CLR() TCP/UDP**

Remove a socket file descriptor ID as a member of a file descriptor set.

See Section A.12.20 for more information.

Files

net_bsd.h

Prototype

```
FD_CLR(fd, fdsetp);
```

Required Configuration

Available only if NET_BSD_CFG_API_EN is enabled (see Section 3.15.12).

A.17.06 **FD_ISSET() TCP/UDP**

Check if a socket file descriptor ID is a member of a file descriptor set.

See Section A.12.23 for more information.

Files

net_bsd.h

Prototype

```
FD_ISSET(fd, fdsetp);
```

Required Configuration

Available only if NET_BSD_CFG_API_EN is enabled (see Section 3.15.12).

A.17.07 **FD_SET() TCP/UDP**

Add a socket file descriptor ID as a member of a file descriptor set.

See Section A.12.24 for more information.

Files

`net_bsd.h`

Prototype

```
FD_SET(fd, fdsetp);
```

Required Configuration

Available only if `NET_BSD_CFG_API_EN` is enabled (see Section 3.15.12).

A.17.08 **FD_ZERO() TCP/UDP**

Initialize/zero-clear a file descriptor set.

See Section A.12.22 for more information.

Files

`net_bsd.h`

Prototype

```
FD_ZERO(fdsetp);
```

Required Configuration

Available only if `NET_BSD_CFG_API_EN` is enabled (see Section 3.15.12).

A.17.09 htonl()

Convert 32-bit integer values from CPU host-order to network-order.

See Section A.16.02 for more information.

Files

`net_bsd.h`

Prototype

```
htonl(val);
```

Required Configuration

Available only if `NET_BSD_CFG_API_EN` is enabled (see Section 3.15.12).

A.17.10 htons()

Convert 16-bit integer values from CPU host-order to network-order.

See Section A.16.01 for more information.

Files

`net_bsd.h`

Prototype

```
htons(val);
```

Required Configuration

Available only if `NET_BSD_CFG_API_EN` is enabled (see Section 3.15.12).

A.17.11 inet_addr() IPv4

Convert a string of an IPv4 address in dotted-decimal notation to an IPv4 address in host-order.

See Section A.04.03 for more information.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
in_addr_t  inet_addr(char  *paddr);
```

Arguments

`paddr` Pointer to an ASCII string that contains a dotted-decimal IPv4 address.

Returned Value

Returns the IPv4 address represented by ASCII string in host-order, if NO errors;
`-1` otherwise (i.e. `0xFFFFFFFF`).

Required Configuration

Available only if either `NET_CFG_TRANSPORT_LAYER_SEL` is configured for TCP (see Section 3.12) and/or `NET_UDP_CFG_APP_API_SEL` is configured for sockets (see Section 3.13.01) AND if `NET_BSD_CFG_API_EN` is enabled (see Section 3.15.12).

Notes / Warnings

RFC 1983 states that "dotted decimal notation ... refers [to] IP addresses of the form A.B.C.D; where each letter represents, in decimal, one byte of a four byte IP address". In other words, the dotted-decimal notation separates four decimal byte values by the dot, or period, character ('.'). Each decimal value represents one byte of the IP address starting with the most significant byte in network order.

IPv4 Address Examples :

DOTTED DECIMAL NOTATION	HEXADECIMAL EQUIVALENT
`127.0.0.1`	`0x7F000001`
`192.168.1.64`	`0xC0A80140`
`255.255.255.0`	`0xFFFFFF00`
MSB .... LSB	MSB .... LSB

MSB Most Significant Byte in Dotted-Decimal IP Address
LSB Least Significant Byte in Dotted-Decimal IP Address

The IPv4 dotted-decimal ASCII string **MUST** include **ONLY** decimal values and the dot, or period, character ('.'); **ALL** other characters are trapped as invalid, including any leading or trailing characters. The ASCII string **MUST** include exactly four decimal values separated by exactly three dot characters. Each decimal value **MUST NOT** exceed the maximum byte value (i.e. 255), or exceed the maximum number of digits for each byte (i.e. 3) including any leading zeros.

A.17.12 **inet_ntoa() IPv4**

Convert an IPv4 address in host-order into an IPv4 dotted-decimal notation ASCII string.

See Section A.04.01 for more information.

Files

net_bsd.h/net_bsd.c

Prototype

```
char  *inet_ntoa(struct  in_addr  addr);
```

Arguments

`in_addr` IPv4 address (in host-order).

Returned Value

Pointer to ASCII string of converted IPv4 address (see Notes / Warnings), if NO errors;
Pointer to `NULL`, otherwise.

Required Configuration

Available only if either `NET_CFG_TRANSPORT_LAYER_SEL` is configured for TCP (see Section 3.12) and/or `NET_UDP_CFG_APP_API_SEL` is configured for sockets (see Section 3.13.01) AND if `NET_BSD_CFG_API_EN` is enabled (see Section 3.15.12).

Notes / Warnings

RFC 1983 states that "dotted decimal notation ... refers [to] IP addresses of the form A.B.C.D; where each letter represents, in decimal, one byte of a four byte IP address". In other words, the dotted-decimal notation separates four decimal byte values by the dot, or period, character ('.'). Each decimal value represents one byte of the IP address starting with the most significant byte in network order.

IPv4 Address Examples :

DOTTED DECIMAL NOTATION	HEXADECIMAL EQUIVALENT
`127.0.0.1`	`0x7F000001`
`192.168.1.64`	`0xC0A80140`
`255.255.255.0`	`0xFFFFFF00`
MSB .... LSB	MSB .... LSB

MSB Most Significant Byte in Dotted-Decimal IP Address
LSB Least Significant Byte in Dotted-Decimal IP Address

Since the returned ASCII string is stored in a single, global ASCII string array, this function is **NOT** reentrant or thread-safe. Therefore, the returned string should be copied as soon as possible before other calls to `inet_ntoa()` are needed.

A.17.13 listen() TCP

Set a socket to accept incoming connections.

See Section A.12.27 for more information.

Files

net_bsd.h/net_bsd.c

Prototype

```
int  listen(int  sock_id,
            int  sock_q_size);
```

A.17.14 ntohl()

Convert 32-bit integer values from network-order to CPU host-order.

See Section A.16.04 for more information.

Files

net_bsd.h

Prototype

```
ntohl(val);
```

Required Configuration

Available only if NET_BSD_CFG_API_EN is enabled (see Section 3.15.12).

A.17.15 ntohs()

Convert 16-bit integer values from network-order to CPU host-order.

See Section A.16.03 for more information.

Files

net_bsd.h

Prototype

```
ntohs(val);
```

Required Configuration

Available only if NET_BSD_CFG_API_EN is enabled (see Section 3.15.12).

A.17.16 recv() / recvfrom() TCP/UDP

Copy up to a specified number of bytes received from a remote socket into an application memory buffer.

See Section A.12.31 for more information.

Files

net_bsd.h/net_bsd.c

Prototypes

```
ssize_t  recv(int      sock_id,
              void    *pdata_buf,
             _size_t   data_buf_len,
              int      flags);

ssize_t  recvfrom(        int         sock_id,
                          void       *pdata_buf,
                         _size_t      data_buf_len,
                          int         flags,
                  struct  sockaddr   *paddr_remote,
                          socklen_t  *paddr_len);
```

A.17.17 **select() TCP/UDP**

Check if any sockets are ready for available read or write operations or error conditions.

See Section A.12.32 for more information.

Files

net_bsd.h/net_bsd.c

Prototype

```
int  select(        int       desc_nbr_max,
             struct  fd_set   *pdesc_rd,
             struct  fd_set   *pdesc_wr,
             struct  fd_set   *pdesc_err,
             struct  timeval  *ptimeout);
```

A.17.18 send() / sendto() TCP/UDP

Copy bytes from an application memory buffer into a socket to send to a remote socket.

See Section A.12.33 for more information.

Files

net_bsd.h/net_bsd.c

Prototypes

```
ssize_t  send (int       sock_id,
               void     *p_data,
              _size_t    data_len,
               int       flags);

ssize_t  sendto(         int          sock_id,
                         void        *p_data,
                        _size_t       data_len,
                         int          flags,
                 struct  sockaddr    *paddr_remote,
                         socklen_t    addr_len);
```

A.17.19 socket() TCP/UDP

Create a datagram (i.e. UDP) or stream (i.e. TCP) type socket.

See Section A.12.28 for more information.

Files

net_bsd.h/net_bsd.c

Prototype

```
int  socket(int  protocol_family,
            int  sock_type,
            int  protocol);
```

Appendix

B

μC/TCP-IP Error Codes

This appendix provides a brief explanation of **μC/TCP-IP** error codes defined in `net_err.h`. Any error codes not listed here may be searched in `net_err.h` for both their numerical value and usage.

B.01 Network Error Codes

NET_ERR_INIT_INCOMPLETE	Network initialization NOT complete.
NET_ERR_INVALID_PROTOCOL	Invalid/unknown network protocol type.
NET_ERR_INVALID_TRANSACTION	Invalid/unknown network buffer pool type.
NET_ERR_RX	General receive error. Receive data discarded.
NET_ERR_RX_DEST	Destination address and/or port number not available on this host.
NET_ERR_TX	General transmit error. No data transmitted. Try delaying momentarily in order to allow additional buffers to be de-allocated before calling `send()`, `NetSock_TxData()` or `NetSock_TxDataTo()`.

B.02 ARP Error Codes

NET_ARP_ERR_CACHE_INVALID_TYPE	ARP cache type invalid or unknown.
NET_ARP_ERR_CACHE_NONE_AVAIL	No ARP cache entry structures available.
NET_ARP_ERR_CACHE_NOT_FOUND	ARP cache entry not found.
NET_ARP_ERR_CACHE_PEND	ARP cache resolution pending.
NET_ARP_ERR_INVALID_HW_ADDR_LEN	Invalid ARP hardware address length.
NET_ARP_ERR_INVALID_PROTOCOL_ADDR_LEN	Invalid ARP protocol address length.
NET_ARP_ERR_NONE	ARP operation completed successfully.
NET_ARP_ERR_NULL_PTR	Argument(s) passed `NULL` pointer.

B.03 Network ASCII Error Codes

NET_ASCII_ERR_INVALID_CHAR	Invalid ASCII character.
NET_ASCII_ERR_INVALID_CHAR_LEN	Invalid ASCII character length.
NET_ASCII_ERR_INVALID_CHAR_SEQ	Invalid ASCII character sequence.
NET_ASCII_ERR_INVALID_CHAR_VAL	Invalid ASCII character value.
NET_ASCII_ERR_INVALID_STR_LEN	Invalid ASCII string length.
NET_ASCII_ERR_NONE	ASCII operation completed successfully.
NET_ASCII_ERR_NULL_PTR	Argument(s) passed `NULL` pointer.

B.04 Network Buffer Error Codes

NET_BUF_ERR_INVALID_IX	Invalid buffer index outside DATA area.
NET_BUF_ERR_INVALID_LEN	Invalid buffer length specified outside of data area.
NET_BUF_ERR_INVALID_POOL_ADDR	Invalid network buffer pool address.
NET_BUF_ERR_INVALID_POOL_QTY	Invalid number of pool buffers configured.
NET_BUF_ERR_INVALID_POOL_TYPE	Invalid network buffer pool type.
NET_BUF_ERR_INVALID_SIZE	Invalid network buffer pool size.
NET_BUF_ERR_NONE_AVAIL	No network buffers of required size available.
NET_BUF_ERR_POOL_INIT	Network buffer pool initialization failed.

B.05 ICMP Error Codes

B.06 Network Interface Error Codes

NET_IF_ERR_INVALID_ADDR	Invalid hardware address specified.
NET_IF_ERR_INVALID_ADDR_LEN	Invalid hardware address length specified.
NET_IF_ERR_INVALID_CFG	Invalid network interface configuration specified.
NET_IF_ERR_INVALID_IF	Invalid network interface number specified.
NET_IF_ERR_INVALID_IO_CTRL_OPT	Invalid I/O control option parameter specified.
NET_IF_ERR_INVALID_MTU	Invalid hardware MTU specified.
NET_IF_ERR_INVALID_STATE	Invalid network interface state for specified operation.
NET_IF_ERR_NONE	Network interface operation completed successfully.
NET_IF_ERR_NONE_AVAIL	No network interfaces available.Try increasing NET_IF_CFG_MAX_NBR_IF in net_cfg.h.
NET_IF_ERR_NULL_FNCT	NULL interface API function pointer encountered.
NET_IF_ERR_NULL_PTR	Argument(s) passed NULL pointer.

B.07 IP Error Codes

NET_IP_ERR_ADDR_CFG_IN_PROGRESS	Interface address configuration in progress.
NET_IP_ERR_ADDR_CFG_IN_USE	Specified IP address currently in use.
NET_IP_ERR_ADDR_CFG_STATE	Invalid IP address state for attempted operation.
NET_IP_ERR_ADDR_NONE_AVAIL	No IP addresses configured.
NET_IP_ERR_ADDR_NOT_FOUND	IP address not found.
NET_IP_ERR_ADDR_TBL_EMPTY	IP address table empty.
NET_IP_ERR_ADDR_TBL_FULL	IP address table full.
NET_IP_ERR_ADDR_TBL_SIZE	Invalid IP address table size argument passed.
NET_IP_ERR_INVALID_ADDR_GATEWAY	Invalid gateway IP address.
NET_IP_ERR_INVALID_ADDR_HOST	Invalid host IP address.
NET_IP_ERR_NONE	IP operation completed successfully.
NET_IP_ERR_NULL_PTR	Argument(s) passed NULL pointer.

B.08 IGMP Error Codes

NET_IGMP_ERR_HOST_GRP_INVALID_TYPE	Invalid or unknown IGMP host group type.
NET_IGMP_ERR_HOST_GRP_NONE_AVAIL	No host group available.
NET_IGMP_ERR_HOST_GRP_NOT_FOUND	No IGMP host group found.
NET_IGMP_ERR_INVALID_ADDR_DEST	Invalid IGMP IP destination address.
NET_IGMP_ERR_INVALID_ADDR_GRP	Invalid IGMP IP host group address
NET_IGMP_ERR_INVALID_ADDR_SRC	Invalid IGMP IP source address.
NET_IGMP_ERR_INVALID_CHK_SUM	Invalid IGMP checksum.
NET_IGMP_ERR_INVALID_LEN	Invalid IGMP message lenth.
NET_IGMP_ERR_INVALID_TYPE	Invalid IGMP message type.
NET_IGMP_ERR_INVALID_VER	Invalid IGMP version.
NET_IGMP_ERR_NONE	IGMP operation completed successfully.

B.09 OS Error Codes

NET_OS_ERR_LOCK	Network global lock access **NOT** acquired. OS-implemented lock may be corrupted.

B.10 Network Socket Error Codes

NET_SOCK_ERR_ADDR_IN_USE	Socket address (IP / port number) already in use.
NET_SOCK_ERR_CLOSED	Socket already/previously closed.
NET_SOCK_ERR_CLOSE_IN_PROGRESS	Socket already closing.
NET_SOCK_ERR_CONN_ACCEPT_Q_NONE_AVAIL	Accept connection handle identifier **NOT** available.
NET_SOCK_ERR_CONN_FAIL	Socket operation failed.
NET_SOCK_ERR_CONN_IN_USE	Socket address (IP / port number) already connected.
NET_SOCK_ERR_CONN_SIGNAL_TIMEOUT	Socket operation **NOT** signaled before specified timeout.
NET_SOCK_ERR_EVENTS_NBR_MAX	Number of configured socket events is greater than the maximum number of socket events.
NET_SOCK_ERR_FAULT	Fatal socket fault; close socket immediately.
NET_SOCK_ERR_INVALID_ADDR	Invalid socket address specified.
NET_SOCK_ERR_INVALID_ADDR_LEN	Invalid socket address length specified.
NET_SOCK_ERR_INVALID_CONN	Invalid socket connection.
NET_SOCK_ERR_INVALID_DATA_SIZE	Socket receive or transmit data does not fit into the receive or transmit buffer.In the case of receive, excess data bytes are dropped; for transmit, no data is sent.
NET_SOCK_ERR_INVALID_DESC	Invalid socket descriptor number.
NET_SOCK_ERR_INVALID_FAMILY	Invalid socket family; close socket immediately.
NET_SOCK_ERR_INVALID_FLAG	Invalid socket flags specified.

NET_SOCK_ERR_INVALID_OP	Invalid socket operation; e.g. socket not in the correct state for specified socket call.
NET_SOCK_ERR_INVALID_PORT_NBR	Invalid port number specified.
NET_SOCK_ERR_INVALID_PROTOCOL	Invalid socket protocol; close socket immediately.
NET_SOCK_ERR_INVALID_SOCK	Invalid socket number specified.
NET_SOCK_ERR_INVALID_STATE	Invalid socket state; close socket immediately.
NET_SOCK_ERR_INVALID_TIMEOUT	Invalid or no timeout specified.
NET_SOCK_ERR_INVALID_TYPE	Invalid socket type; close socket immediately.
NET_SOCK_ERR_NONE	Socket operation completed successfully.
NET_SOCK_ERR_NONE_AVAIL	No available socket resources to allocate; try increasing NET_SOCK_CFG_NBR_SOCK in net_cfg.h.
NET_SOCK_ERR_NOT_USED	Socket not used; do **NOT** close or use the socket for further operations.
NET_SOCK_ERR_NULL_PTR	Argument(s) passed NULL pointer.
NET_SOCK_ERR_NULL_SIZE	Argument(s) passed NULL size.
NET_SOCK_ERR_PORT_NBR_NONE_AVAIL	Random local port number not available.
NET_SOCK_ERR_RX_Q_CLOSED	Socket receive queue closed (received FIN from peer).
NET_SOCK_ERR_RX_Q_EMPTY	Socket receive queue empty.
NET_SOCK_ERR_TIMEOUT	No socket events occurred before timeout expired.

B.11 UDP Error Codes

NET_UDP_ERR_INVALID_DATA_SIZE	UDP receive or transmit data does not fit into the receive or transmit buffer.In the case of receive, excess data bytes are dropped; for transmit, no data is sent.
NET_UDP_ERR_INVALID_FLAG	Invalid UDP flags specified.
NET_UDP_ERR_INVALID_LEN_DATA	Invalid protocol/data length.
NET_UDP_ERR_INVALID_PORT_NBR	Invalid UDP port number.
NET_UDP_ERR_NONE	UDP operation completed successfully.
NET_UDP_ERR_NULL_PTR	Argument(s) passed NULL pointer.
NET_UDP_ERR_NULL_SIZE	Argument(s) passed NULL size.

Appendix

C

µC/TCP-IP Frequently Asked Questions (FAQ)

This appendix provides a brief explanation to a variety of common questions regarding how to use **µC/TCP-IP**.

C.01 µC/TCP-IP Licensing

C.01.01 How do I obtain µC/TCP-IP source code?

If you wish to obtain the **µC/TCP-IP** source code, or if you have any other pre-sale questions, please contact sales@micrium.com.

C.01.02 How do I license µC/TCP-IP?

In order to license **µC/TCP-IP,** please contact sales@micrium.com.

C.01.03 Why should I renew maintenance for µC/TCP-IP?

Licensing **µC/TCP-IP** provides you with one year of limited technical support and maintenance and source code updates. If you want to renew the maintenance agreement for continued support and source code updates after this time, please contact sales@micrium.com.

C.01.04 How do I obtain µC/TCP-IP source code updates?

If you are under maintenance, you will be automatically emailed when source code updates become available. You can then download your available updates from ftp.micrium.com. If you are no longer under maintenance, or forgot your Micrium FTP username or password, please contact sales@micrium.com.

C.01.05 How do I obtain support for µC/TCP-IP?

Please visit the customer support section in www.micrium.com to select the type of support you require.

C.02 μC/TCP-IP Configuration and Initialization

C.02.01 How do I configure the μC/TCP-IP stack?

Please refer to Chapter 3 for information on this topic.

C.02.02 How do I know how large to configure the μC/LIB memory heap?

The **μC/LIB** memory heap is used for allocation of the following objects:

1. Transmit small buffers
2. Transmit large buffers
3. Receive large buffers
4. Network Buffers (Network Buffer header and pointer to data area)
5. DMA receive descriptors
6. DMA transmit descriptors
7. Interface data area
8. Device driver data area

The size of Network Buffer Data Areas (1, 2, 3) vary based on configuration. However, for this example, lets assume the following object sizes in bytes:

- Small transmit buffers: 256
- Large transmit buffers: 1594 for maximum sized TCP packets (1614 for maximum sized ICMP packets)
- Large receive buffers: 1518
- Size of Network Buffer: 134
- Size of DMA receive descriptor: 8
- Size of DMA transmit descriptor: 8
- Ethernet interface data area: 7
- Average Ethernet device driver data area: 108

Assume a 4-byte alignment on all memory pool objects requires a worst case disposal of 3 leading bytes for each object. In practice this is not usually true since the size of most objects tend to be even multiples of 4 and thus alignment is preserved after having aligned the start of the pool data area. However, this makes the case for allocating objects to the next greatest multiple of 4 in order to save space on the alignment of those objects.

Then the approximate memory heap size may be determined according to the following formulas:

```
nbr buf per interface     =  nbr small Tx buf +
                             nbr large Tx buf +
                             nbr large Rx buf

nbr net buf per interface =  nbr buf per interface

nbr objects               = (nbr buf per interface     +
                             nbr net buf per interface +
                             nbr Rx descriptors        +
                             nbr Tx descriptors        +
                             1 Ethernet data area      +
                             1 Device driver data area)

interface mem             = (nbr small Tx buf       *  256) +
                            (nbr large Tx buf       * 1594) +
                            (nbr large Rx buf       * 1518) +
                            (nbr buf per interface  *  134) +
                            (nbr Rx descriptors     *    8) +
                            (nbr Tx descriptors     *    8) +
                            (Ethernet IF  data area *    7) +
                            (Ethernet Drv data area *  108) +
                            (nbr objects            *    3)
total mem required        =  nbr interfaces * interface mem
```

Example:

Let's assume the following configuration:

- 10 small transmit buffers
- 10 large transmit buffers
- 10 large receive buffers
- 6 receive descriptors
- 20 transmit descriptors
- Ethernet interface (interface + device driver data area required)

```
nbr     buf per interface =  10 + 10 + 10                = 30
nbr net buf per interface =  nbr buf per interface       = 30

nbr objects               = (30 + 30 + 6  + 20 + 1  + 1) = 88

interface mem             = (10 *  256) +
                            (10 * 1594) +
                            (10 * 1518) +
                            (30 *  134) +
                            ( 6 *    8) +
                            (20 *    8) +
                            (1  *    7) +
                            (1  *  108) +
                            (88 *    3)                  = 38,553
                                                           bytes

total mem required        = 38,553 + local host memory if enabled.
```

The local host interface, when enabled, requires a similar amount of memory except that it does not require Rx and Tx descriptors, an IF data area, or a device driver data area.

The value determined by this formula is only an estimate. In some cases it may be possible to reduce the size of the **µC/LIB** memory heap by inspecting the variable `Mem_PoolHeap.SegSizeRem` after all interfaces have been successfully initialized and any additional application allocations (if applicable) have been completed.

Excess heap space, if present, may be subtracted from the lib heap size configuration macro, `LIB_MEM_CFG_HEAP_SIZE`, present in `app_cfg.h`.

C.02.03 How do I know how large to make the µC/TCP-IP task stacks?

In general, the size of the **µC/TCP-IP** task stacks is dependent on the CPU architecture and compiler used.

On ARM processors, experience has shown that configuring the task stacks to 1024 `OS_STK` entries (4,096 bytes) is sufficient for most applications. Of course, the stack sizes may be examined and reduced accordingly once the run-time behavior of the device has been analyzed and additional stack space deemed to be unnecessary.

See also Section 3.17.01.

C.02.04 How do I configure µC/TCP-IP task priorities?

We recommand configuring Network Protocol Stack task priorities in the following order:

```
NET_OS_CFG_IF_TX_DEALLOC_TASK_PRIO   (highest priority)
NET_OS_CFG_IF_RX_TASK_PRIO
NET_OS_CFG_IF_LOOPBACK_TASK_PRIO
NET_OS_CFG_TMR_TASK_PRIO             (lowest priority)
```

We recommend that the **µC/TCP-IP** Receive Task, Loopback Task, and Timer Task be lower priority than almost all other application tasks. But we recommend that the Transmit Deallocation Task be higher priority than all application tasks that use **µC/TCP-IP** network services.

Depending on the application, it may be desirable to swap the priorities of the Loopback and Receive Task in order to ensure that heavy localhost traffic does not starve physical interfaces of CPU time.

See also Section 3.17.01.

C.02.05 How do I configure µC/TCP-IP queue sizes?

Please refer to Section 3.17.02.

C.02.06 How do I initialize µC/TCP-IP?

The following example code demonstrates the initialization of two identical Network Interface Devices via a local, application developer provided function named `AppInit_TCPIP()`.

The first interface is bound to two different sets of network addresses on two separate networks. The second interface is configured to operate on one of the same networks as the first interface, but could easily be plugged into a separate network that happens to use the same address ranges.

Listing C-1 Complete Initialization Example

```
static  void  AppInit_TCPIP (void)
{
    NET_IF_NBR   if_nbr;
    NET_IP_ADDR  ip;
    NET_IP_ADDR  msk;
    NET_IP_ADDR  gateway;
    CPU_BOOLEAN  cfg_success;
    NET_ERR      err;


    Mem_Init();                                                          (1)

    err = Net_Init();                                                    (2)

    if (err != NET_ERR_NONE) {
        return;
    }

    if_nbr = NetIF_Add((void    *)&NetIF_API_Ether,                      (3)
                       (void    *)&NetDev_API_FEC,
                       (void    *)&NetDev_Cfg_FEC_0,
                       (void    *)&NetPHY_API_Generic,
                       (void    *)&NetPhy_Cfg_FEC_0,
                       (NET_ERR *)&err);
```

```
    if (err == NET_IF_ERR_NONE) {
        ip      = NetASCII_Str_to_IP((CPU_CHAR *)"192.168.1.2",   &err);   (4)
        msk     = NetASCII_Str_to_IP((CPU_CHAR *)"255.255.255.0", &err);
        gateway = NetASCII_Str_to_IP((CPU_CHAR *)"192.168.1.1",   &err);

        cfg_success = NetIP_CfgAddrAdd(if_nbr, ip, msk, gateway,  &err);   (5)


        ip      = NetASCII_Str_to_IP((CPU_CHAR *)"10.10.1.2",     &err);   (6)
        msk     = NetASCII_Str_to_IP((CPU_CHAR *)"255.255.255.0", &err);
        gateway = NetASCII_Str_to_IP((CPU_CHAR *)"10.10.1.1",     &err);

        cfg_success = NetIP_CfgAddrAdd(if_nbr, ip, msk, gateway,  &err);   (7)


        NetIF_Start(if_nbr, &err);                                         (8)
    }

    if_nbr = NetIF_Add((void    *)&NetIF_API_Ether,                         (9)
                       (void    *)&NetDev_API_FEC,
                       (void    *)&NetDev_Cfg_FEC_1,
                       (void    *)&NetPHY_API_Generic,
                       (void    *)&NetPhy_Cfg_FEC_1,
                       (NET_ERR *)&err);

    if (err == NET_IF_ERR_NONE) {
        ip      = NetASCII_Str_to_IP((CPU_CHAR *)"192.168.1.3",   &err);           (10)
        msk     = NetASCII_Str_to_IP((CPU_CHAR *)"255.255.255.0", &err);
        gateway = NetASCII_Str_to_IP((CPU_CHAR *)"192.168.1.1",   &err);

        cfg_success = NetIP_CfgAddrAdd(if_nbr, ip, msk, gateway,  &err);           (11)


        NetIF_Start(if_nbr, &err);                                                 (12)
    }
}
```

LC-1(1) Initialize **µC/LIB** memory management. Most applications call this function PRIOR to `AppInit_TCPIP()` so that other parts of the application may benefit from memory management functionality prior to initializing **µC/TCP-IP**.

LC-1(2) Initialize **µC/TCP-IP**. This function must only be called once following the call to **µC/LIB** `Mem_Init()`. The return error code should be checked for `NET_ERR_NONE` before proceeding

LC-1(3) Add the first network interface to the system. In this case, an Ethernet interface bound to a Freescale FEC hardware device and generic (MII or RMII) compliant physical layer device is being configured. Notice that the interface is using a difference device configuration structure than the second interface being added in step 8. Each interface requires a unique device configuration structure. Physical layer device configuration structures however could be re-used if the Physical layer configurations are exactly the same. The return error should be checked before starting the interface.

LC-1(4) Obtain the hexadecimal equivalents for the first set of internet addresses to configure on the first added interface.

LC-1(5) Configure the first added interface with the first set of specified addresses.

LC-1(6) Obtain the hexadecimal equivalents for the second set of internet addresses to configure on the first added interface. Notice that the same local variables have been used as when the first set of address information was configured. Once the address set has been configured to the interface, as in step 4, the local copies of the configured addresses are no longer necessary and can be overwritten with the next set of addresses to configure.

LC-1(7) Configure the first added interface with the second set of specified addresses.

LC-1(8) Start the first interface. The return error code should be checked, but this depends on whether the application will attempt to restart the interface should an error occur. This example assumes that no error occurs when starting the interface. Initialization for the first interface is now complete, and if no further initialization takes place, the first interface will respond to ICMP Echo (ping) requests on either of its configured addresses.

LC-1(9) Add the second network interface to the system. In this case, an Ethernet interface bound to a Freescale FEC hardware device and generic (MII or RMII) compliant physical layer device is being configured. Notice that the interface is using a difference device configuration structure than the first interface being added in step 2. Each interface requires a unique device configuration structure. Physical layer device configuration structures however could be re-used if the Physical layer configurations are exactly the same. The return error **should** be checked before starting the interface.

LC-1(10) Obtain the hexadecimal equivalents for the first and only set of internet addresses to configure on the second added interface.

LC-1(11) Configure the second interface with the first and only set of specified addresses.

LC-1(12) Start the second interface. The return error code should be checked, but this depends on whether the application will attempt to restart the interface should an error occur. This example assumes that no error occurs when starting the interface. Initialization for the second interface is now complete and it will respond to ICMP Echo (ping) requests on its configured address.

C.03 Network Interfaces, Devices, and Buffers

C.03.01 Network Interface Configuration

C.03.01.01 How do I add an interface?

Interfaces may be added to the stack by calling `NetIF_Add()`. Of course, each new interface requires additional BSP and the order of addition is critical for ensuring that the interface number assigned to the new interface matches the code defined within `net_bsp.c`. Please see Sections 4.01 for more information on configuring and adding interfaces.

C.03.01.02 How do I start an interface?

Interfaces may be started by calling `NetIF_Start()`. Please see Section 4.02.01 for more information on starting interfaces.

C.03.01.03 How do I stop an interface?

Interfaces may be started by calling `NetIF_Stop()`. Please see Section 4.02.02 for more information on stopping interfaces.

C.03.01.04 How do I check if an interface is enabled?

The application may check if an interface is enabled by calling `NetIF_IsEn()` or `NetIF_IsEnCfgd()`. See Sections A.10.08 & A.10.09 for more information.

C.03.02 Network and Device Buffer Configuration

C.03.02.01 Why are large transmit buffers 1594 (or 1614) bytes?

Please refer to the chapter on buffer management for more information.

C.03.02.02 How do I determine how many Rx or Tx buffers to configure?

The number of large receive, small transmit and large transmit buffers configured for a specific interface depend on several factors.

1. Desired level of performance.
2. Amount of data to be either transmitted or received.
3. Ability of the target application to either produce or consume transmitted or received data.
4. Average CPU utilization.
5. Average network utilization.

In general, the more buffers the better. However, the number of buffers can be tailored based on the application. For example, if an application recieves a lot of data but transmits very little, then it may be sufficient to define a number of small transmit buffers for operations such as TCPIP acknowledgements and allocate the remaining memory to large receive buffers. Similarly, if an application transmits and recieves little, then the buffer allocation emphasis should be on defining more transmit buffers. However, there is a caveat:

If the application is written such that the task that consumes receive data runs infrequently or the CPU utilization is high and the receiving application task(s) becomes starved for CPU time, then more receive buffers will be required.

In order to ensure the highest level of performance possible, it makes sense to define as many buffers as possible and use the interface and pool statistics data in order to refine the number after having run the application for a while. A busy network will require more receive buffers in order to handle the additional broadcast messages that will be received.

In general, you should configure at least 2 large and 2 small transmit buffers. This assumes the neither the network or CPU are very busy.

Many applications will receive properly with 4 or more large receive buffers. However, for TCP applications that move a lot of data between the target and the peer, this number may need to be higher.

Specifying too few transmit or receive buffers may lead to stalls in communication and possibly even dead-lock. Care should be taken when configuring the number of buffers.

µC/TCP-IP is often tested with configurations of 10 or more small transmit, large transmit, and large receive buffers.

C.03.02.03 How do I determine how many DMA descriptors to configure?

If your hardware device is an Ethernet MAC that supports DMA, then the number of configured receive descriptors will play an important role in determining overall performance for the configured interface.

For applications which 10 or less large receive buffers, it is desirable to configure the number of receive descriptors to that of 60% or 70% of the number of configured receive buffers.

In this example, 60% of 10 receive buffers allows for four receive buffers to be 'hanging around' the stack waiting to be processed by application tasks. While the application is processing data, the hardware may continue to receive additional frames up to the number of configured receive descriptors.

There is however a point in which configuring additional receive descriptors no longer greatly impacts performance. For applications with 20 or more buffers, the number of descriptors can be configured to 50% of the number of configured receive buffers. After this point, only the number of buffers remains a significant factor; especially for slower or busy CPU's and networks with higher utilization.

In general, if the CPU is not busy and the **µC/TCP-IP** Receive Task has opportunity to run often, then the ratio of receive descriptors to receive buffers may be reduced further for very high numbers of available receive buffers (e.g. 50 or more).

The number of transmit descriptors should be configured such that it is equal to the number of small plus the number of large transmit buffers.

These number only serve as a starting point. Your application and the environment that the device will be attached to will ultimately dictate the numer of required transmit and receive descriptors necessary for achieving maximum performance.

Specifying too few descriptors can cause communication delays.

C.03.02.04 How do I write or obtain additional device drivers?

Please contact Micrium for information about obtaining additional device drivers. If a specific driver is not available, You may be able to pay non reoccuring engineering charges and Micrium will write it for you.

Alternately, you may write your own device driver by filling in a template driver provided with the **µC/TCP-IP** source code.

Please see the ***µC/TCP-IP*** *Driver Architecture* document for more information.

C.03.03 Ethernet MAC Address

C.03.03.01 How do I obtain the MAC address of an interface?

The application may call `NetIF_AddrHW_Get()` in order to obtain the MAC address for a specific interface.

C.03.03.02 How do I change the MAC address of an interface?

The application may call `NetIF_AddrHW_Set()` in order to set the MAC address for a specific interface.

C.03.03.03 How do I obtain the MAC address of a host on my network?

In order to determine the MAC address of a host on your network, the Network Protocol Stack must have an ARP cache entry for the specified host protocol address. An application may check to see if an ARP cache entry is present by calling `NetARP_CacheGetAddrHW()`.

If an ARP cache entry is not found, the application may call `NetARP_ProbeAddrOnNet()` in order to send an ARP request to all hosts on the network. If the target host is present, an ARP reply will be received shortly and the application should wait and then call `NetARP_CacheGetAddrHW()` to see if the ARP reply has been entered into the ARP cache.

The following example shows how to obtain the Ethernet MAC address of a host on the local area network:

```
void  AppGetRemoteHW_Addr (void)
{
    NET_IP_ADDR   addr_ip_local;
    NET_IP_ADDR   addr_ip_remote;
    CPU_CHAR     *paddr_ip_remote;
    CPU_CHAR      addr_hw_str[NET_IF_ETHER_ADDR_SIZE_STR];
    CPU_INT08U    addr_hw[NET_IF_ETHER_ADDR_SIZE];
    NET_ERR       err;


                                        /* ------------- PREPARE IP ADDRs ------------- */
    paddr_ip_local  = "10.10.1.10";     /* MUST be one of host's configured IP addrs.   */
     addr_ip_local  = NetASCII_Str_to_IP((CPU_CHAR *) paddr_ip_local,
                                         (NET_ERR  *)&err);
    if (err != NET_ASCII_ERR_NONE) {
        printf("  Error #%d converting IP address %s", err, paddr_ip_local);
        return;
    }

    paddr_ip_remote = "10.10.1.50";     /* Remote host's IP addr to get hardware addr. */
     addr_ip_remote = NetASCII_Str_to_IP((CPU_CHAR *) paddr_ip_remote,
                                         (NET_ERR  *)&err);
    if (err != NET_ASCII_ERR_NONE) {
        printf("  Error #%d converting IP address %s", err, paddr_ip_remote);
        return;
    }
```

```
 addr_ip_local  = NET_UTIL_HOST_TO_NET_32(addr_ip_local);
 addr_ip_remote = NET_UTIL_HOST_TO_NET_32(addr_ip_remote);

                                    /* ------------ PROBE ADDR ON NET ------------- */
 NetARP_ProbeAddrOnNet((NET_PROTOCOL_TYPE) NET_PROTOCOL_TYPE_IP_V4,
                       (CPU_INT08U      *)&addr_ip_local,
                       (CPU_INT08U      *)&addr_ip_remote,
                       (NET_ARP_ADDR_LEN ) sizeof(addr_ip_remote),
                       (NET_ERR         *)&err);
 if (err != NET_ARP_ERR_NONE) {
     printf("  Error #%d probing address %s on network", err, addr_ip_remote);
     return;
 }

 OSTimeDly(2);                      /* Delay short time for ARP to probe network.  */

                                    /* ---- QUERY ARP CACHE FOR REMOTE HW ADDR ---- */
(void)NetARP_CacheGetAddrHW((CPU_INT08U     *)&addr_hw[0],
                            (NET_ARP_ADDR_LEN) sizeof(addr_hw_str),
                            (CPU_INT08U     *)&addr_ip_remote,
                            (NET_ARP_ADDR_LEN) sizeof(addr_ip_remote),
                            (NET_ERR        *)&err);
 switch (err) {
     case NET_ARP_ERR_NONE:
          NetASCII_MAC_to_Str((CPU_INT08U *)&addr_hw[0],
                              (CPU_CHAR   *)&addr_hw_str[0],
                              (CPU_BOOLEAN ) DEF_NO,
                              (CPU_BOOLEAN ) DEF_YES,
                              (NET_ERR    *)&err);
          if (err != NET_ASCII_ERR_NONE) {
              printf("  Error #%d converting hardware address", err);
              return;
          }

          printf("  Remote IP Addr %s  @  HW Addr %s\n\r",
                    paddr_ip_remote, &addr_hw_str[0]);
          break;
```

```
        case NET_ARP_ERR_CACHE_NOT_FOUND:
             printf("   Remote IP Addr %s  NOT     found on network\n\r", paddr_ip_remote);
             break;


        case NET_ARP_ERR_CACHE_PEND:
             printf("   Remote IP Addr %s  NOT YET found on network\n\r", paddr_ip_remote);
             break;


        case NET_ARP_ERR_NULL_PTR:
        case NET_ARP_ERR_INVALID_HW_ADDR_LEN:
        case NET_ARP_ERR_INVALID_PROTOCOL_ADDR_LEN:
        default:
             printf("  Error #%d querying ARP cache", err);
             break;
    }
}
```

C.03.04 Ethernet Phy Link State

C.03.04.01 How do I increase the rate of link state polling?

The application may increase the **µC/TCP-IP** link state polling rate by calling `NetIF_CfgPhyLinkPeriod()` (see Section A.10.06). The default value is 250ms.

C.03.04.02 How do I obtain the current link state for an interface?

µC/TCP-IP provides two mechanisms for obtaining interface link state.

1. Call a function which reads a global variable that is periodically updated.
2. Call a function which reads the current link state from the hardware.

Method 1 provides the fastest mechanism to obtain link state since it does not require communication with the physical layer device. For most applications, this mechanism is suitable and if necessary, the polling rate can be increased by calling NetIF_CfgPhyLinkPeriod().

In order to utilize method 1, the application may call NetIF_LinkStateGet() which returns NET_IF_LINK_UP or NET_IF_LINK_DOWN.

The accuracy of method 1 can be improved by using a physical layer device and driver combination that supports link state change interrupts. In this circumstance, the value of the global variable containing the link state is updated immediately following a link state change. Therefore, the polling rate can be reduced further if desired and a call to NetIF_LinkStateGet() will return the actual link state.

Method 2 requires the application to call NetIF_IO_Ctrl() with the option parameter set to either:

NET_IF_IO_CTRL_LINK_STATE_GET
NET_IF_IO_CTRL_LINK_STATE_GET_INFO.

If the application specifies NET_IF_IO_CTRL_LINK_STATE_GET, then NET_IF_LINK_UP or NET_IF_LINK_DOWN will be returned.

Alternatively, if the application specifies NET_IF_IO_CTRL_LINK_STATE_GET_INFO, then the link state details such as speed and duplex will be returned.

The advantage to method 2 is that the link state returned is the actual link state as reported by the hardware at the time of the function call. However, the overhead of communicating with the physical layer device may be high and therefore some cycles may be wasted waiting for the result since the connection bus between the CPU and the physical layer device is often only a couple of MHz.

C.03.04.03 How do I force an Ethernet Phy to a specific link state?

The generic Phy driver that comes with **µC/TCP-IP** does not provide a mechanism for disabling auto-negotation and specifying a desired link state. This restriction is required in order to remain MII register block compliant with all (R)MII compliant physical layer devices.

However, **µC/TCP-IP** does provide a mechanism for coaching the physical layer device into advertising only the desired auto-negotiation states. This may be achieved by adjusting the physical layer device configuration as specified in `net_dev_cfg.c` with alternative link speed and duplex values.

Below is an example physical layer device configuration structure.

```
NET_PHY_CFG_ETHER  NetPhy_Cfg_Generic_0 = {
    0,
    NET_PHY_BUS_MODE_MII,
    NET_PHY_TYPE_EXT,
    NET_PHY_SPD_AUTO,
    NET_PHY_DUPLEX_AUTO
};
```

The parameters `NET_PHY_SPD_AUTO` and `NET_PHY_DUPLEX_AUTO` may be changed to match any of the following settings:

```
NET_PHY_SPD_10
NET_PHY_SPD_100
NET_PHY_SPD_1000
NET_PHY_SPD_AUTO

NET_PHY_DUPLEX_HALF
NET_PHY_DUPLEX_FULL
NET_PHY_DUPLEX_AUTO
```

This mechanism is only effective when both the physical layer device attached to the target and the remote link state partner supoprt auto-negotiation.

C.04 IP Address Configuration

C.04.01 How do I convert IP addresses to and from their dotted decimal representation?

µC/TCP-IP contains functions for performing various string operations on IP addresses.

The following example shows how to use the NetASCII module in order to convert IP addresses to and from their dotted decimal representations:

```
NET_IP_ADDR  ip;
CPU_INT08U   ip_str[16];
NET_ERR      err;

ip = NetASCII_Str_to_IP((CPU_CHAR *)"192.168.1.65",  &err);

NetASCII_IP_to_Str(ip, &ip_str[0], DEF_NO, &err);
```

C.04.02 How do I statically assign one or more IP addresses to an interface?

The constant `NET_IP_CFG_IF_MAX_NBR_ADDR` specified in `net_cfg.h` determines the maximum number of IP addresses that may be assigned to an interface. You may add as many IP addresses up to the specified maximum by calling `NetIP_CfgAddrAdd()`.

Configuring an IP gateway address is not necessary when communicating only within your local network.

```
    CPU_BOOLEAN  cfg_success;

    ip      = NetASCII_Str_to_IP((CPU_CHAR *)"192.168.1.65",  perr);
    msk     = NetASCII_Str_to_IP((CPU_CHAR *)"255.255.255.0", perr);
    gateway = NetASCII_Str_to_IP((CPU_CHAR *)"192.168.1.1",   perr);

    cfg_success = NetIP_CfgAddrAdd(if_nbr, ip, msk, gateway,  perr);
```

C.04.03 How do I remove one or more statically assigned IP addresses from an interface?

Statically assigned IP addresses for a specific interface may be removed by calling `NetIP_CfgAddrRemove()`. Alternatively, the application may call `NetIP_CfgAddrRemoveAll()` in order to remove all configured static addresses for a specific interface.

C.04.04 How do I get a dynamic IP address?

You must obtain and integrate **µC/DHCPc** to dynamically assign an IP address to an interface. Please refer to the **µC/DHCPc** user manual for more information.

C.04.05 How do I obtain all the IP addresses configured on a specific interface?

The application may obtain the protocol address information for a specific interface by calling `NetIP_GetAddrHost()`. This function may return one or more configured addresses.

Similarly, the application may call `NetIP_GetAddrSubnetMask()` and `NetIP_GetAddrDfltGateway()` in order to determine the subnet mask and gateway information for a specific interface.

C.05 Socket Programming

C.05.01 How do I use µC/TCP-IP sockets to develop an application?

Refer to *Application Note AN-3003 µC/TCP-IP Socket Programming* document for code examples on this topic.

C.05.02 How do I join and leave an IGMP host group?

µC/TCP-IP supports IP multicasting with IGMP. In order to receive packets addressed to a given IP multicast group address, the stack must have been configured to support multicasting in net_cfg.h, and that host group has to be joined.

The following examples show how to join and leave an IP multicast group with **µC/TCP-IP**:

```
NET_IF_NBR   if_nbr;
NET_IP_ADDR  group_ip_addr;
NET_ERR      err;


if_nbr = NET_IF_NBR_BASE_CFGD;
group_ip_addr = NetASCII_Str_to_IP("233.0.0.1", &err);
if (err != NET_ASCII_ERR_NONE) {
    /* Handle error. */
}

NetIGMP_HostGrpJoin(if_nbr, group_ip_addr, &err);
if (err != NET_IGMP_ERR_NONE) {
    /* Handle error. */
}

[...]

NetIGMP_HostGrpLeave(if_nbr, group_ip_addr, &err);
if (err != NET_IGMP_ERR_NONE) {
    /* Handle error. */
}
```

C.05.03 How do I transmit to a multicast IP group address?

Transmitting to an IP multicast group is identical to transmitting to a unicast or broadcast address. You do, however, have to configure the stack to enable multicast transmit.

Refer to *Application Note AN-3003 µC/TCP-IP Socket Programming* document for code examples on this topic.

C.05.04 How do I receive from a multicast IP group?

You have to have joined an IP multicast group before you can receive packet from it (see C.05.02 for more information). Once this is done, receiving from a multicast group only requires a socket bound to the NET_SOCK_ADDR_IP_WILDCARD address, as shown in the following example:

```
NET_SOCK_ID        sock;
NET_SOCK_ADDR_IP   sock_addr_ip;
NET_SOCK_ADDR      addr_remote;
NET_SOCK_ADDR_LEN  addr_remote_len;
CPU_CHAR           rx_buf[100];
CPU_INT16U         rx_len;
NET_ERR            err;


sock = NetSock_Open((NET_SOCK_PROTOCOL_FAMILY) NET_SOCK_ADDR_FAMILY_IP_V4,
                    (NET_SOCK_TYPE           ) NET_SOCK_TYPE_DATAGRAM,
                    (NET_SOCK_PROTOCOL       ) NET_SOCK_PROTOCOL_UDP,
                    (NET_ERR                *)&err);
if (err != NET_SOCK_ERR_NONE) {
    /* Handle error. */
}


Mem_Set(&sock_addr_ip, (CPU_CHAR)0, sizeof(sock_addr_ip));
sock_addr_ip.AddrFamily  = NET_SOCK_ADDR_FAMILY_IP_V4;
sock_addr_ip.Addr        = NET_UTIL_HOST_TO_NET_32(NET_SOCK_ADDR_IP_WILDCARD);
sock_addr_ip.Port        = NET_UTIL_HOST_TO_NET_16(10000);
```

```
NetSock_Bind((NET_SOCK_ID      ) sock,
             (NET_SOCK_ADDR   *)&sock_addr_ip,
             (NET_SOCK_ADDR_LEN) NET_SOCK_ADDR_SIZE,
             (NET_ERR         *)&err);
if (err != NET_SOCK_ERR_NONE) {
    /* Handle error. */
}

rx_len = NetSock_RxDataFrom((NET_SOCK_ID        ) sock,
                            (void              *)&rx_buf [0],
                            (CPU_INT16U         ) BUF_SIZE,
                            (CPU_INT16S         ) NET_SOCK_FLAG_NONE,
                            (NET_SOCK_ADDR     *)&addr_remote,
                            (NET_SOCK_ADDR_LEN *)&addr_remote_len,
                            (void              *) 0,
                            (CPU_INT08U         ) 0,
                            (CPU_INT08U        *) 0,
                            (NET_ERR           *)&err);
```

C.05.05 Why does my application receive socket errors immediately after reboot?

Immediately after a network interface is added, the physical layer device is reset and network interface and device initialization begins. However, it may take up to 3 seconds for the average Ethernet physical layer device to complete auto-negotiation. During this time, the socket layer will return `NET_SOCK_ERR_LINK_DOWN` for sockets that are bound to the interface in question.

The application should attempt to retry the socket operation with a short delay between attempts until network link has been established.

C.05.06 How do I reduce the number of transitory errors (NET_ERR_TX)?

Try increasing the number of transmit buffers. Additionally, it may be helpful to add a short delay between successive calls to socket transmit functions.

C.05.07 How do I control socket blocking options?

Socket blocking options may be configured during compile time by adjusting the `net_cfg.h` macro `NET_SOCK_CFG_BLOCK_SEL` to the following values:

```
NET_SOCK_BLOCK_SEL_DFLT
NET_SOCK_BLOCK_SEL_BLOCK
NET_SOCK_BLOCK_SEL_NO_BLOCK
```

`NET_SOCK_BLOCK_SEL_DFLT` selects blocking as the default option, however, allows run-time code to override blocking settings by specifying additional socket.

`NET_SOCK_BLOCK_SEL_BLOCK` configures all sockets to always block.

`NET_SOCK_BLOCK_SEL_NO_BLOCK` configures all sockets to non blocking.

See the Sections A.12.31 & A.12.33 for more information about sockets and blocking options.

C.05.08 How do I tell if a socket is still connected to a peer?

Applications may call `NetSock_IsConn()` to determine if a socket is (still) connected to a remote socket (see Section A.12.26).

Alternatively, applications may make a non-blocking call to `recv()`, `NetSock_RxData()`, or `NetSock_RxDataFrom()` and inspect the return value. If data or a non-fatal, transitory error is returned, then the socket is still connected; otherwise, if '`0`' or a fatal error is returned, then the socket is disconnected or closed.

C.05.09 Why do I receive -1 instead of 0 when calling recv() for a closed socket?

When a remote peer closes a socket, and the target application calls one of the receive socket functions, **µC/TCP-IP** will first report that the receive queue is empty and return a -1 for both BSD and **µC/TCP-IP** socket API functions. The next call to receive will indicate that the socket has been closed by the remote peer.

This is a known issue and will be corrected in subsequent versions of **µC/TCP-IP**.

C.05.10 How do I determine which interface a UDP datagram was received on?

If a UDP socket server is bound to the 'any' address, then it is not currently possible to know which interface received the UDP datagram. This is a limitation in the BSD socket API and therefore no solution has been implemented in the **µC/TCP-IP** socket API.

In order to guarantee which interface a UDP packet was received on, the socket server will have to bind a specific interface address.

In fact, if a UDP datagram is received on a listening socket bound to the 'any' address and the application transmits a response back to the peer using the same socket, then the newly transmitted UDP datagram will be transmitted from the default interface. The default interface may or may not be the interface in which the UDP datagram originated.

C.06 µC/TCP-IP Statistics and Debug

C.06.01 How do I obtain performance statistics during run-time?

µC/TCP-IP periodically measures and estimates run-time performance on a per interface basis. The performance data is stored in the global **µC/TCP-IP** statistics data structure, `Net_StatCtrs` which is of type `NET_CTR_STATS`.

Each interface has a performance metric structure which is allocated within a single array of `NET_CTR_IF_STATS`. Each index in the array represents a different interface.

In order to access the performance metrics for a specific interface number, the application may externally access the array by viewing the variable `Net_StatCtrs.NetIF_StatCtrs[if_nbr].field_name`, where `if_nbr` represents the interface number in question, 0 for the loopback interface, and where `field_name` corresponds to one of the fields below.

Possible field names:

```
NetIF_StatRxNbrOctets
NetIF_StatRxNbrOctetsPerSec
NetIF_StatRxNbrOctetsPerSecMax
NetIF_StatRxNbrPktCtr
NetIF_StatRxNbrPktCtrPerSec
NetIF_StatRxNbrPktCtrPerSecMax
NetIF_StatRxNbrPktCtrProcessed

NetIF_StatTxNbrOctets
NetIF_StatTxNbrOctetsPerSec
NetIF_StatTxNbrOctetsPerSecMax
NetIF_StatTxNbrPktCtr
NetIF_StatTxNbrPktCtrPerSec
NetIF_StatTxNbrPktCtrPerSecMax
NetIF_StatTxNbrPktCtrProcessed
```

Please see Chapter 9 for more information.

C.06.02 How do I view error and statistics counters?

In order to access the statistics and error counters, the application may externally access the global **µC/TCP-IP** statistics array by referencing the members of the structure variable `Net_StatCtrs`.

Please see Chapter 9 for more information.

C.06.03 How do I use network debug functions to check network status conditions?

The following are some example(s) demonstrating how to use the network debug status functions:

```
NET_DBG_STATUS  net_status;
CPU_BOOLEAN     net_fault;
CPU_BOOLEAN     net_fault_conn;
CPU_BOOLEAN     net_rsrc_lost;
CPU_BOOLEAN     net_rsrc_low;

net_status     = NetDbg_ChkStatus();
net_fault      = DEF_BIT_IS_SET(net_status, NET_DBG_STATUS_FAULT);
net_fault_conn = DEF_BIT_IS_SET(net_status, NET_DBG_STATUS_FAULT_CONN);
net_rsrc_lost  = DEF_BIT_IS_SET(net_status, NET_DBG_STATUS_RSRC_LOST);
net_rsrc_lo    = DEF_BIT_IS_SET(net_status, NET_DBG_STATUS_RSRC_LO);

net_status     = NetDbg_ChkStatusTmrs();
```

C.07 µC/TCP-IP Optimization

C.07.01 How do I optimize µC/TCP-IP for additional performance?

There are several configuration combinations that can improve overall **µC/TCP-IP** performance. You may begin by adjusting the following items:

1. If using the ARM architecture, enable the assembly port optimizations.
2. Configure the **µC/TCP-IP** for speed optimization.
3. Configure optimum TCP window sizes for TCP communication.
4. Disable argument checking, statistics and error counters.

First, if you are using the ARM architecture, or other supported optimized architecture, you may include `net_util_a.asm` and `lib_mem_a.asm` in to your project and define / enable the following macros:

`app_cfg.h`: #define `uC_CFG_OPTIMIZE_ASM_EN`
`net_cfg.h`: Set `NET_CFG_OPTIMIZE_ASM_EN` to `DEF_ENABLED`

These files are generally located in the following directories:

```
C:\Micrium\Software\uC-LIB\Ports\ARM\IAR\lib_mem_a.asm
C:\Micrium\Software\uC-TCPIP-V2\Ports\ARM\IAR\net_util_a.asm
```

Second, you may compile the Network Protocol Stack with speed optimizations enabled. This can be accomplished by configurign the `net_cfg.h` macro `NET_CFG_OPTIMIZE to NET_OPTIMIZE_SPD`.

Third, configure the `net_cfg.h` macros `NET_TCP_CFG_RX_WIN_SIZE_OCTET` and `NET_TCP_CFG_TX_WIN_SIZE_OCTET`. These macros configure each TCP connections' receive & transmit window sizes. We recommend setting TCP window sizes to integer multiples of each TCP connection's maximum segment size (MSS). For example, systems with an Ethernet MSS of 1460, a value 5840 (4 * 1460) is probably a better configuration than the default window size of **`4096`** (4K).

Lastly, once you have validated your application, you may optionally disable argument checking, statistics and error counters by configuring the following macros to `DEF_DISABLED`:

```
NET_ERR_CFG_ARG_CHK_EXT_EN
NET_ERR_CFG_ARG_CHK_DBG_EN
NET_CTR_CFG_STAT_EN
NET_CTR_CFG_ERR_EN
```

C.08 Miscellaneous

C.08.01 How do I send and receive ICMP Echo Requests from the target?

µC/TCP-IP does not support sending and receiving ICMP Echo and Reply messages from the user application. However, the target is capable of receiving externally generated ICMP Echo messages and replying accordingly.

C.08.02 How do I enable TCP Keep-Alives?

µC/TCP-IP does not support TCP Keep-Alives at this time. If both ends of the connection are running different Network Protocol Stacks, you may attempt to enable TCP Keep-Alives on the remote side. Alternatively, the application will have to send something through the socket to the remote peer in order to ensure that the TCP connection remains open.

C.08.03 Can I use µC/TCP-IP for inter-process communication?

Yes, tasks can communicate with sockets via the localhost interface which must be enabled.

Appendix

D

µC/TCP-IP Licensing Policy

You can evaluate the **µC/TCP-IP** source code for FREE for 45 days, but a license is required when **µC/TCP-IP** is used commercially. The policy is as follows:

µC/TCP-IP source and object code can be used by accredited Colleges and Universities without requiring a license, as long as there is no commercial application involved. In other words, no licensing is required if **µC/TCP-IP** is used for educational purposes.

You need to obtain an 'Executable Distribution License' to embed **µC/TCP-IP** in a product that is sold with the intent to make a profit or if the product is not used for education or 'peaceful' research.

For licensing details, contact us at:

Micriµm
949 Crestview Circle
Weston, FL 33327-1848
U.S.A.
+1 954 217 2036
FAX: +1 954 217 2037
www.micrium.com
licensing@micrium.com