

---

# **Freescale MQX™ RTCS™ User's Guide**

Document Number: MQXRTCSUG  
Rev. 8  
06/2012

## ***How to Reach Us:***

### **Home Page:**

[www.freescale.com](http://www.freescale.com)

### **E-mail:**

[support@freescale.com](mailto:support@freescale.com)

### **USA/Europe or Locations Not Listed:**

Freescale Semiconductor  
Technical Information Center, CH370  
1300 N. Alma School Road  
Chandler, Arizona 85224  
+1-800-521-6274 or +1-480-768-2130  
[support@freescale.com](mailto:support@freescale.com)

### **Europe, Middle East, and Africa:**

Freescale Halbleiter Deutschland GmbH  
Technical Information Center  
Schatzbogen 7  
81829 Muenchen, Germany  
+44 1296 380 456 (English)  
+46 8 52200080 (English)  
+49 89 92103 559 (German)  
+33 1 69 35 48 48 (French)  
[support@freescale.com](mailto:support@freescale.com)

### **Japan:**

Freescale Semiconductor Japan Ltd.  
Headquarters  
ARCO Tower 15F  
1-8-1, Shimo-Meguro, Meguro-ku,  
Tokyo 153-0064, Japan  
0120 191014 or +81 3 5437 9125  
[support.japan@freescale.com](mailto:support.japan@freescale.com)

### **Asia/Pacific:**

Freescale Semiconductor China Ltd.  
Exchange Building 23F  
No. 118 Jianguo Road  
Chaoyang District  
Beijing 100022  
China  
+86 10 5879 8000  
[support.asia@freescale.com](mailto:support.asia@freescale.com)

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals", must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.



Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© 1994-2008 ARC™ International. All rights reserved.

© Freescale Semiconductor, Inc. 2008-2012. All rights reserved.

Document Number: MQXRTCSUG

Rev. 8

06/2012

## Chapter 1

### Before You Begin

1.1	About This Book	15
1.2	Where to Go for More Information	15
1.3	Conventions	15
1.3.1	Product Names	15
1.3.2	Tips	15
1.3.3	Notes	15
1.3.4	Cautions	16

## Chapter 2

### Setting Up the RTCS

2.1	Introduction	17
2.2	Supported Protocols and Policies	17
2.3	RTCS Included with Freescale MQX RTOS	17
2.3.1	Protocol Stack Architecture	21
2.4	Setting Up the RTCS	22
2.5	Defining RTCS Protocols	22
2.6	Changing RTCS Creation Parameters	23
2.7	Creating RTCS	23
2.8	Changing RTCS Running Parameters	23
2.8.1	Enabling IP Forwarding	23
2.8.2	Bypassing TCP Checksums	24
2.9	Initializing Device Interfaces	24
2.9.1	Initializing Interfaces to Ethernet Devices	24
2.9.2	Initializing Interfaces to Point-to-Point Devices	24
2.10	Adding Device Interfaces to RTCS	25
2.10.1	Removing Device Interfaces from RTCS	25
2.11	Binding IP Addresses to Device Interfaces	25
2.11.1	Unbinding IP Addresses from Device Interfaces	25
2.12	Adding Gateways	25
2.12.1	Adding Default Gateways	25
2.12.2	Adding Gateways to a Specific Route	25
2.12.3	Removing Gateways	25
2.13	Downloading and Running a Boot File	26
2.14	Enabling RTCS Logging	26
2.15	Starting Network Address Translation	26
2.15.1	Changing Inactivity Timeouts	27
2.15.2	Specifying Port Ranges	27
2.15.3	Disabling NAT Application-Level Gateways	27
2.15.4	Getting NAT Statistics	28
2.15.5	Supported Protocols	28
2.15.6	Example: Setting Up RTCS	30
2.16	Compile-Time Options	31

2.16.1 Recommended Settings .....	32
2.16.2 Configuration Options and Default Settings .....	32
2.16.3 Application specific default settings .....	37
2.16.4 HTTP Server default configuration .....	39

## Chapter 3 Using Sockets

3.1 Before You Begin .....	41
3.2 Protocols Supported .....	41
3.3 Socket Definition .....	41
3.4 Socket Options .....	42
3.5 Comparison of Datagram and Stream Sockets .....	42
3.6 Datagram Sockets .....	42
3.6.1 Connectionless .....	42
3.7 Unreliable Transfer .....	42
3.8 Block-Oriented .....	43
3.9 Stream Sockets .....	43
3.10 Connection-Based .....	43
3.11 Reliable Transfer .....	43
3.12 Character-Oriented .....	43
3.13 Creating and Using Sockets .....	43
3.14 Creating Sockets .....	45
3.15 Changing Socket Options .....	45
3.16 Binding Sockets .....	45
3.17 Using Datagram Sockets .....	45
3.18 Setting Datagram-Socket Options .....	45
3.19 Transferring Datagram Data .....	46
3.19.1 Buffering .....	46
3.19.2 Pre-Specifying a Peer .....	46
3.20 Shutting Down Datagram Sockets .....	46
3.21 Using Stream Sockets .....	46
3.22 Changing Stream-Socket Options .....	46
3.23 Establishing Stream-Socket Connections .....	47
3.23.1 Establishing Stream-Socket Connections Passively .....	47
3.23.2 Establishing Stream-Socket Connections Actively .....	47
3.24 Getting Stream-Socket Names .....	47
3.25 Sending Stream Data .....	47
3.26 Receiving Stream Data .....	48
3.27 Buffering Data .....	48
3.28 Improving the Throughput of Stream Data .....	48
3.29 Shutting Down Stream Sockets .....	49
3.29.1 Shutting Down Gracefully .....	49
3.29.2 Shutting Down with an Abort Operation .....	49
3.30 Example .....	50

## Chapter 4

### Point-to-Point Drivers

4.1	Before You Begin	53
4.2	PPP and PPP Driver	53
4.2.1	LCP Configuration Options	53
4.2.2	Configuring PPP Driver	55
4.2.3	Changing Authentication	57
4.2.4	Initializing PPP Links	60
4.2.5	Getting PPP Statistics	60
4.2.6	Example: Using PPP Driver	60
4.3	PPP over Ethernet Driver	60
4.3.1	Setting Up PPP over Ethernet Driver	61
4.3.2	Examples: Using PPP over Ethernet Driver	62

## Chapter 5

### RTCS Applications

5.1	Before You Begin	69
5.2	DHCP Client	69
5.2.1	Example: Setting Up and Using DHCP Client	70
5.3	DHCP Server	70
5.3.1	Example: Setting Up and Modifying DHCP Server	71
5.4	DNS Resolver	71
5.4.1	Setting Up DNS Resolver	71
5.4.2	Using DNS Resolver	71
5.4.3	Communicating with a DNS Server	72
5.4.4	Using DNS Services	72
5.5	Echo Server	72
5.6	EDS Server	73
5.7	FTP Client	73
5.8	FTP Server	73
5.8.1	Communicating with an FTP Client	73
5.9	HTTP Server	74
5.9.1	Compile Time Configuration	74
5.9.2	Basic Usage	74
5.9.3	Providing Static Content	74
5.9.4	Dynamic Content — CGI-Like Pages	75
5.9.5	Dynamic Content — ASP-Like Page Callbacks	76
5.10	IPCFG — High-Level Network Interface Management	77
5.11	IWCFG — High-Level Wireless Network Interface Management	78
5.12	SNMP Agent	78
5.12.1	Configuring SNMP Agent	79
5.12.2	Starting SNMP Agent	79
5.12.3	Communicating with SNMP Clients	79
5.12.4	Defining Management Information Base (MIB)	79

5.12.5 Processing the MIB File .....	85
5.12.6 Standard MIB Included In RTCS .....	85
5.13 SNMP Client .....	85
5.14 Telnet Client .....	86
5.15 Telnet Server .....	86
5.16 TFTP Client .....	86
5.17 TFTP Server .....	86
5.17.1 Configuring TFTP Server .....	86
5.17.2 Starting TFTP Server .....	87
5.18 Quote of the Day Service .....	87
5.19 Typical RTCS IP Packet Paths .....	87

## Chapter 6 Rebuilding

6.1 Why to Rebuild RTCS .....	89
6.2 Before You Begin .....	89
6.3 RTCS Directory Structure .....	90
6.4 RTCS Build Projects in Freescale MQX .....	90
6.4.1 Post-Build Processing .....	90
6.4.2 Build Targets .....	91
6.5 Rebuilding Freescale MQX RTCS .....	91

## Chapter 7 Function Reference

7.1 Function Listing Format .....	93
7.1.1 function_name() .....	93
7.1.2 _iopcb_open() .....	95
7.1.3 _iopcb_ppphdlc_init() .....	96
7.1.4 _iopcb_pppoe_client_destroy() .....	97
7.1.5 _iopcb_pppoe_client_init() .....	98
7.1.6 _pppoe_client_stats() .....	101
7.1.7 _pppoe_server_destroy() .....	102
7.1.8 _pppoe_server_if_add() .....	103
7.1.9 _pppoe_server_if_remove() .....	104
7.1.10 _pppoe_server_if_stats() .....	105
7.1.11 _pppoe_server_init() .....	106
7.1.12 _pppoe_server_session_stats() .....	107
7.1.13 accept() .....	108
7.1.14 ARP_stats() .....	110
7.1.15 bind() .....	112
7.1.16 connect() .....	114
7.1.17 DHCP_find_option() .....	116
7.1.18 DHCP_option_addr() .....	117
7.1.19 DHCP_option_addrlist() .....	118

7.1.20 DHCP_option_int16()	119
7.1.21 DHCP_option_int32()	120
7.1.22 DHCP_option_int8()	121
7.1.23 DHCP_option_string()	121
7.1.24 DHCP_option_variable()	123
7.1.25 DHCPCLNT_find_option()	124
7.1.26 DHCPCLNT_release()	125
7.1.27 DHCPDRV_init()	126
7.1.28 DHCPDRV_ippool_add()	128
7.1.29 DHCPDRV_set_config_flag_off()	129
7.1.30 DHCPDRV_set_config_flag_on()	130
7.1.31 DNS_init()	131
7.1.32 ECHOSRV_init()	132
7.1.33 EDS_init()	133
7.1.34 ENET_get_stats()	134
7.1.35 ENET_initialize()	135
7.1.36 FTP_close()	136
7.1.37 FTP_command()	137
7.1.38 FTP_command_data()	138
7.1.39 FTPd_init()	139
7.1.40 FTP_open()	143
7.1.41 FTPDRV_init()	145
7.1.42 gethostbyaddr()	146
7.1.43 gethostbyname()	147
7.1.44 getpeername()	149
7.1.45 getsockname()	151
7.1.46 getsockopt()	152
7.1.47 httpd_default_params()	153
7.1.48 httpd_init()	154
7.1.49 httpd_server_init()	155
7.1.50 httpd_server_run()	156
7.1.51 httpd_server_poll()	157
7.1.52 HTTPD_SET_PARAM_ROOT_DIR	158
7.1.53 HTTPD_SET_PARAM_INDEX_PAGE	159
7.1.54 HTTPD_SET_PARAM_FN_TBL	160
7.1.55 HTTPD_SET_PARAM_CGI_TBL	161
7.1.56 ICMP_stats()	162
7.1.57 IGMP_stats()	163
7.1.58 IP_stats()	164
7.1.59 IPIF_stats()	165
7.1.60 ipcfg_init_device()	166
7.1.61 ipcfg_init_interface()	168
7.1.62 ipcfg_bind_boot()	170
7.1.63 ipcfg_bind_dhcp()	171
7.1.64 ipcfg_bind_dhcp_wait()	173

7.1.65 ipcfg_bind_staticip()	175
7.1.66 ipcfg_get_device_number()	176
7.1.67 ipcfg_add_interface()	177
7.1.68 ipcfg_get_ihandle()	178
7.1.69 ipcfg_get_mac()	179
7.1.70 ipcfg_get_state()	180
7.1.71 ipcfg_get_state_string()	181
7.1.72 ipcfg_get_desired_state()	182
7.1.73 ipcfg_get_link_active()	183
7.1.74 ipcfg_get_dns_ip()	184
7.1.75 ipcfg_add_dns_ip()	185
7.1.76 ipcfg_del_dns_ip()	186
7.1.77 ipcfg_get_ip()	187
7.1.78 ipcfg_get_tftp_serveraddress()	188
7.1.79 ipcfg_get_tftp_servername()	189
7.1.80 ipcfg_get_boot_filename()	190
7.1.81 ipcfg_poll_dhcp()	191
7.1.82 ipcfg_task_create()	192
7.1.83 ipcfg_task_destroy()	193
7.1.84 ipcfg_task_status()	194
7.1.85 ipcfg_task_poll()	195
7.1.86 ipcfg_unbind()	196
7.1.87 iwcfg_set_essid()	197
7.1.88 iwcfg_get_essid()	198
7.1.89 iwcfg_commit()	199
7.1.90 iwcfg_set_mode()	200
7.1.91 iwcfg_get_mode()	201
7.1.92 iwcfg_set_wep_key()	202
7.1.93 iwcfg_get_wep_key()	203
7.1.94 iwcfg_set_passphrase()	204
7.1.95 iwcfg_get_passphrase()	205
7.1.96 iwcfg_set_sec_type()	206
7.1.97 iwcfg_get_sectype()	207
7.1.98 iwcfg_set_power()	208
7.1.99 iwcfg_set_scan()	209
7.1.100listen()	211
7.1.101MIB1213_init()	212
7.1.102MIB_find_objectname()	213
7.1.103MIB_set_objectname()	214
7.1.104NAT_close()	215
7.1.105NAT_init()	216
7.1.106NAT_stats()	217
7.1.107ping()	218
7.1.108PPP_initialize()	219
7.1.109recv()	220



7.1.110	recvfrom()	222
7.1.111	RTCS_attachsock()	224
7.1.112	RTCS_create()	226
7.1.113	RTCS_detachsock()	227
7.1.114	RTCS_exec_TFTP_BIN()	228
7.1.115	RTCS_exec_TFTP_COFF()	230
7.1.116	RTCS_exec_TFTP_SREC()	231
7.1.117	RTCS_gate_add()	233
7.1.118	RTCS_gate_add_metric()	234
7.1.119	RTCS_gate_remove()	235
7.1.120	RTCS_gate_remove_metric()	236
7.1.121	RTCS_geterror()	237
7.1.122	RTCS_if_add()	238
7.1.123	RTCS_if_bind()	239
7.1.124	RTCS_if_bind_BOOTP()	240
7.1.125	RTCS_if_bind_DHCP()	242
7.1.126	RTCS_if_bind_DHCP_flagged()	244
7.1.127	RTCS_if_bind_DHCP_timed()	247
7.1.128	RTCS_if_bind_IPCP()	249
7.1.129	RTCS_if_rebind_DHCP()	251
7.1.130	RTCS_if_remove()	254
7.1.131	RTCS_if_unbind()	255
7.1.132	RTCS_load_TFTP_BIN()	256
7.1.133	RTCS_load_TFTP_COFF()	257
7.1.134	RTCS_load_TFTP_SREC()	258
7.1.135	RTCS_ping()	259
7.1.136	RTCS_request_DHCP_inform()	260
7.1.137	RTCS_selectall()	261
7.1.138	RTCS_selectset()	263
7.1.139	RTCSLOG_disable()	265
7.1.140	RTCSLOG_enable()	266
7.1.141	send()	267
7.1.142	sendto()	270
7.1.143	setsockopt()	272
7.1.144	shutdown()	288
7.1.145	SNMP_init()	290
7.1.146	SNMP_trap_warmStart()	291
7.1.147	SNMP_trap_coldStart()	292
7.1.148	SNMP_trap_authenticationFailure()	293
7.1.149	SNMP_trap_linkDown()	294
7.1.150	SNMP_trap_myLinkDown()	295
7.1.151	SNMP_trap_linkUp()	296
7.1.152	SNMP_trap_userSpec()	297
7.1.153	SNMPv2_trap_warmStart()	298
7.1.154	SNMPv2_trap_coldStart()	299

7.1.155	SNMPv2_trap_authenticationFailure()	300
7.1.156	SNMPv2_trap_linkDown()	301
7.1.157	SNMPv2_trap_linkUp()	302
7.1.158	SNMPv2_trap_userSpec()	303
7.1.159	SNTP_init()	304
7.1.160	SNTP_oneshot()	306
7.1.161	socket()	307
7.1.162	TCP_stats()	308
7.1.163	TELNET_connect()	309
7.1.164	TELNETSRV_init()	310
7.1.165	TFTPSRV_access()	312
7.1.166	TFTPSRV_init()	313
7.1.167	UDP_stats()	314
7.2	Functions Listed by Service	315

## Chapter 8 Data Types

8.1	Data Types for Compiler Portability	319
8.2	Other Data Types	320
8.3	Alphabetical List of RTCS Data Structures	320
8.3.1	_iopcb_handle, _iopcb_table	321
8.3.2	ARP_STATS	323
8.3.3	BOOTP_DATA_STRUCT	325
8.3.4	DHCP_DATA_STRUCT	326
8.3.5	DHCPSRV_DATA_STRUCT	327
8.3.6	ENET_STATS	328
8.3.7	HOSTENT_STRUCT	331
8.3.8	HTTPD_CGI_LINK_STRUCT	332
8.3.9	HTTPD_FN_LINK_STRUCT	333
8.3.10	HTTPD_PARAMS_STRUCT	334
8.3.11	HTTPD_ROOT_DIR_STRUCT	336
8.3.12	HTTPD_SESSION_STRUCT	337
8.3.13	HTTPD_STRUCT	339
8.3.14	ICMP_STATS	340
8.3.15	IGMP_STATS	344
8.3.16	in_addr	345
8.3.17	ip_mreq	346
8.3.18	IP_STATS	347
8.3.19	IPCFG_IP_ADDRESS_DATA	350
8.3.20	IPCP_DATA_STRUCT	351
8.3.21	IPIF_STATS	354
8.3.22	nat_ports	356
8.3.23	NAT_STATS	357
8.3.24	nat_timeouts	358
8.3.25	PPPOE_CLIENT_INIT_DATA_STRUCT	359

8.3.26	PPPOE_SERVER_INIT_DATA_STRUCT	361
8.3.27	PPPOE_SESSION_STATS_STRUCT	363
8.3.28	PPPOE_IF_STATS_STRUCT	364
8.3.29	PPP_SECRET	368
8.3.30	RTCS_ERROR_STRUCT	369
8.3.31	RTCS_IF_STRUCT	370
8.3.32	RTCS_protocol_table	372
8.3.33	RTCS_TASK	373
8.3.34	RTCSMIB_VALUE	374
8.3.35	sockaddr_in	375
8.3.36	TCP_STATS	376
8.3.37	UDP_STATS	381
8.4	LCP	389
8.5	SNTP	390
8.6	IPsec	390
8.7	NAT	390



## Revision History

To provide the most up-to-date information, the revision of our documents on the World Wide Web will be the most current. Your printed copy may be an earlier revision. To verify you have the latest information available, refer to <http://www.freescale.com/mqx>.

The following revision history table summarizes changes contained in this document.

Revision Number	Revision Date	Description of Changes
Rev. 0	01/2009	Initial Release.
Rev. 1	04/2009	Minor formatting updates for MQX 3.2.
Rev. 2	04/2009	Minor formatting updates for MQX 3.2.1
Rev. 3	01/2010	Updated for MQX 3.5. Description of setsockopt call changed.
Rev. 4	07/2010	“Changing RTCS Creation Parameters” section updated.
Rev. 5	02/2011	MQX Embedded -> Freescale MQX. Description of RTCS Logging updated.
Rev. 6	04/2011	IWCFG description added, IPCFG description updated. Examples and features not supported in the current MQX release were labeled. HTTP Server chapter updated.
Rev. 7	12/2011	Description of ENET_initialize() function parameters updated. “Example: Using PPP Driver” section updated.
Rev. 8	06/2012	Several typos corrected in chapters 3.2, 7.1.111 (example), 2.16.2.33 - 2.16.2.35.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc.  
© Freescale Semiconductor, Inc., 2008-2012. All rights reserved.



# Chapter 1 Before You Begin

## 1.1 About This Book

This book is a guide and reference manual for using the MQX™ RTCS™ Embedded TCP/IP Stack, which is part of Freescale MQX Real-Time Operating System distribution.

This *RTCS™ User's Guide* is written for experienced software developers, who have a working knowledge of the C and C++ languages and their target processor.

## 1.2 Where to Go for More Information

- The release notes document accompanying the Freescale MQX release provides information that was not available at the time this user's guide was published.
- The *MQX User's Guide* describes, how to create embedded applications that use the MQX RTOS.
- The *MQX Reference* describes prototypes for the MQX API.

## 1.3 Conventions

This section explains terminology and other conventions used in this manual.

### 1.3.1 Product Names

- RTCS: In this book, we use RTCS as the abbreviation for the MQX™ RTCS™ full-featured TCP/IP stack.
- MQX: MQX is used as the abbreviation for the MQX™ Real-Time Operating System.

### 1.3.2 Tips

Tips point out useful information.

<b>TIP</b>	If your CD-ROM drive is designated by another drive letter, substitute that drive letter in the command.
------------	--

### 1.3.3 Notes

Notes point out important information.

<b>NOTE</b>	Non-strict semaphores do not have priority inheritance.
-------------	---

## 1.3.4 Cautions

Cautions tell you about commands or procedures that could have unexpected or undesirable side effects or could be dangerous to your files or your hardware.

<b>CAUTION</b>	If you modify MQX data types, some tools might not operate properly.
----------------	--



## Chapter 2 Setting Up the RTCS

### 2.1 Introduction

This chapter describes how to configure, create, and set up the RTCS, so that it is ready to use with sockets.

For information about	See
Data types mentioned in this chapter	<a href="#">Chapter 8, “Data Types”</a>
PPP Driver and PPP over Ethernet Driver	<a href="#">Chapter 4, “Point-to-Point Drivers”</a>
Protocols	<a href="#">Section Appendix A, “Protocols and Policies”</a>
Prototypes for functions mentioned in this chapter	<a href="#">Chapter 7, “Function Reference”</a>
Sockets	<a href="#">Chapter 3, “Using Sockets”</a>

### 2.2 Supported Protocols and Policies

[Figure 2-1](#) shows the protocols and policies that are discussed in this manual. For more information about protocols, see the table below and [Section Appendix A, “Protocols and Policies.”](#)

### 2.3 RTCS Included with Freescale MQX RTOS

The RTCS stack included in Freescale MQX RTOS distribution is based on the ARC RTCS version 2.97. Parts of this document may refer to features not available in the Freescale MQX RTCS. Please read the Release Notes document, accompanying the Freescale MQX RTOS, to see if there are any new RTCS features supported.

The major changes in the RTCS introduced in Freescale MQX RTOS distribution are:

- The RTCS is now distributed within the Freescale MQX RTOS package. Also, the RTCS adopts version numbering of the Freescale MQX RTOS distribution (starts with 3.0).
- The RTCS build process and compile-time configuration follows the same principles as other MQX core libraries (see more details in [Chapter 6, “Rebuilding”](#)).
- The RTCS Shell and all shell functions were removed from RTCS library, and were moved to a separate library in the Freescale MQX distribution.
- Freescale MQX contains just the core parts of the original RTCS package. The IPsec, PPPoE, SNMPv3, and some other components are not included in the distribution (although this document may still refer to such features).
- A new HTTP server functionality was added in the Freescale MQX release.

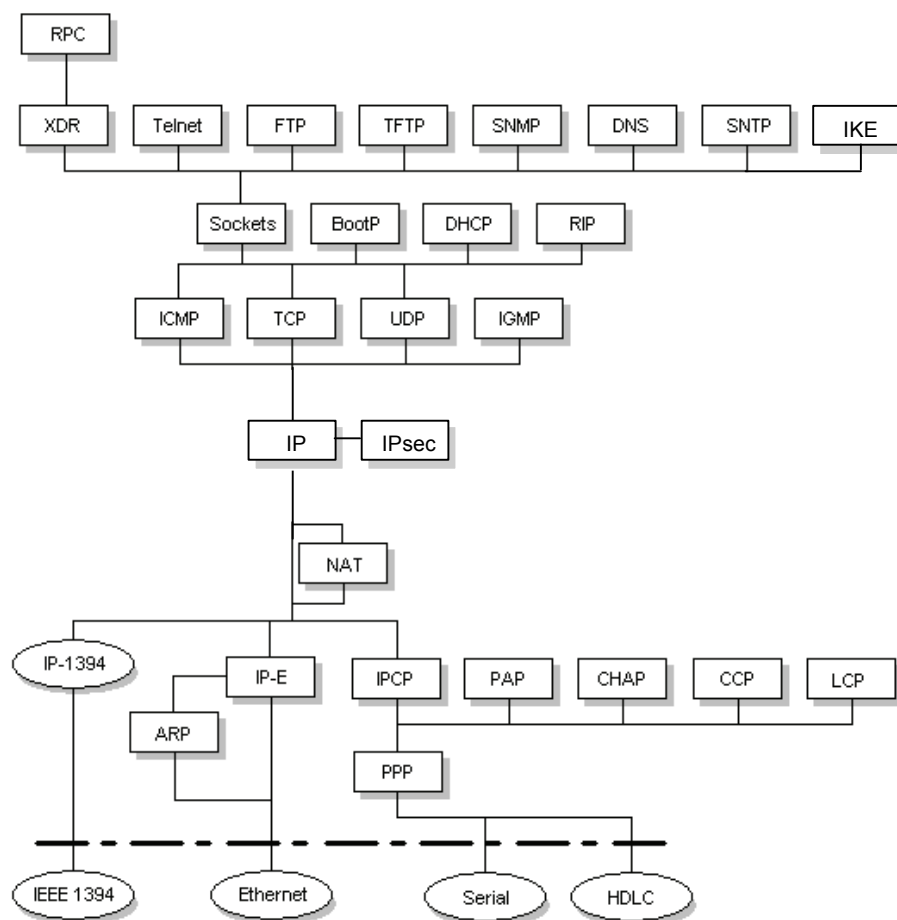


Figure 2-1. Protocols and Policies Discussed in This Manual

Table 2-1. RTCS Features

Protocol or policy	Description	RFC
ARP	Address Resolution Protocol for ethernet	826
Assigned Numbers	RFC 1700 is outdated; for current numbers, see <a href="http://www.iana.org/numbers.html">http://www.iana.org/numbers.html</a> .	
BootP	Bootstrap Protocol	951, 1542
CCP	Compression Control Protocol (used by PPP)	1692
CHAP	Challenge Handshake Authentication Protocol (used by PPP)	1334
CIDR	Classless Inter-Domain Routing	1519

**Table 2-1. RTCS Features** (continued)

Protocol or policy	Description	RFC
DHCP	Dynamic Host Configuration Protocol	2131
DHCP Options	DHCP Options and BootP vendor extensions	2132
DNS	Domain Names: implementation and specification	1035
Echo	Echo protocol	862
EDS	Winsock client/server	—
Ethernet		(IEEE 802.3)
FTP	File Transfer Protocol	959
HDLC	High-Level Data Link Control protocol	(ISO 3309)
HTTP	Hypertext Transport Protocol	2068
ICMP	Internet Control Message Protocol	792
IGMP	Internet Group Management Protocol	1112
IP	Internet Protocol	791, 919, 922
	Broadcasting internet datagrams in the presence of subnets	922
	Internet Standard Subnetting Procedure	950
IPCP	Internet Protocol Control Protocol (used by PPP)	1332
IP-E	A standard for the transmission of IP datagrams over ethernet networks	894
IPIP	IP in IP tunneling	1853
LCP	Link Control Protocol (used by PPP)	1661, 1570
MD5	RSA Data Security Inc. MD5 Message-Digest Algorithm	1321
MIB	Management Information Base (part of SNMPv2)	1902, 1907
NAT	Network Address Translation	
	Traditional IP Network Address Translator (Traditional NAT)	3022
	IP Network Address Translator (NAT) terminology and considerations	2663
PAP	Password Authentication Protocol (used by PPP)	1334

**Table 2-1. RTCS Features** (continued)

Protocol or policy	Description	RFC
ping	Implemented with ICMP Echo message	792
PPP	Point-to-Point Protocol	1661
PPP (HDLC-like framing)	PPP in HDLC-like framing	1662
PPP LCP Extensions		1570
PPPoE	PPP over Ethernet	2516
Quote	Quote of the Day protocol	865
Reqs	Requirements for internet hosts:	
	Communication layers	1122
	Application and Support protocols	1123
	Requirements for IP version 4 routers	1812
RIP	Routing Information Protocol	2453
RPC	Remote Procedure Call protocol	1057
RTCS loaders	S-records, COFF, BIN	—
SMI	Structure of Management Information	1155
SNMPv1	Simple Network Management Protocol, version 1	1157
SNMPv1 MIB	SNMPv1 Management Information Base	1213
SNMPv2	SNMP version 2	1902 – 1907
SNMPv2 MIB	SNMPv2 Management Information Base	1902, 1907
SNMPv3	SNMPv3	2570, 2571, 2572, 2574, 2575
SNTP	Simple Network Time Protocol	2030
TCP	Transmission Control Protocol	793
Telnet	Telnet protocol specification	854
TFTP	Trivial File Transfer Protocol	1350
UDP	User Datagram Protocol	768
XDR	External Data Representation protocol	1014

### 2.3.1 Protocol Stack Architecture

Figure 2-2 shows the architecture of the RTCS stack, and how the RTCS communicates with layers below and above it.

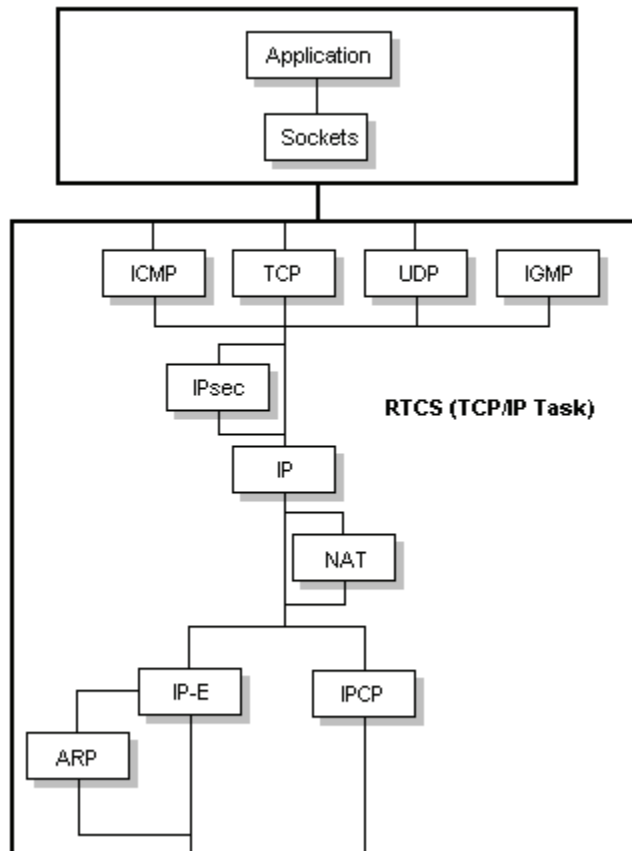


Figure 2-2. Protocol Stack Architecture

## 2.4 Setting Up the RTCS

An application follows a set of general steps to set up the RTCS. The steps are summarized in [Figure 2-3](#) and described in subsequent sections.

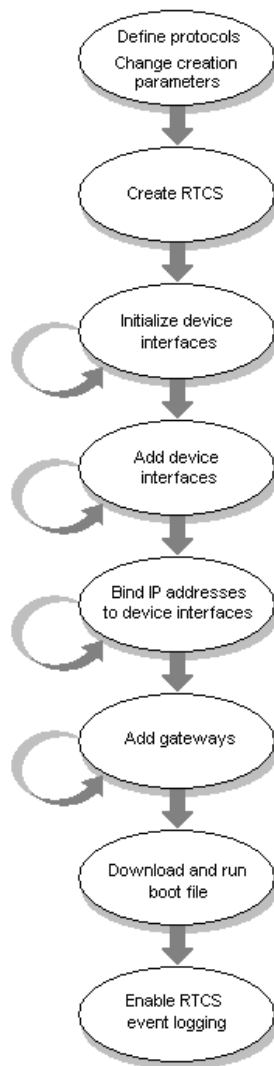


Figure 2-3. Steps to Set Up the RTCS

## 2.5 Defining RTCS Protocols

When an application creates RTCS, it uses a protocol table to determine, which protocols to start, and in which order to start them. Refer to [Section 8.3.32, “RTCS\\_protocol\\_table”](#) in [Chapter 8, “Data Types”](#) for the list of available protocols. You can add or remove protocols using the instructions provided there, or provide your own table.

## 2.6 Changing RTCS Creation Parameters

RTCS uses some global variables, when an application creates it. All the variables have default values, most of which you need never change. If you want to change the values, the application must do so before it creates RTCS; that is, before it calls **RTCS\_create()**.

To change:	From this default value:	Change this creation variable:
Priority of RTCS tasks (because you must assign priorities to all the tasks that you write, RTCS lets you change the priority of RTCS tasks so that it fits with your design).	6	_RTCSTASK_priority (see caution below)
If the priority of RTCS tasks is too low, RTCS might miss received packets or violate the timing specifications for a protocol.		
Additional stack size that is needed for DHCP and IPCP callback functions (for PPP).	0	_RTCSTASK_stacksize
Maximum number of packet control blocks (PCBs) that RTCS uses.	32	_RTCSPCB_max
Pool that RTCS should allocate memory from. If 0, system pool will be used. If a different pool needs to be used the memory pool id must be provided. Example: _RTCS_mem_pool = _mem_create_pool(ADR, SIZE)	0	_RTCS_mem_pool

## 2.7 Creating RTCS

To create RTCS, call **RTCS\_create()**, which allocates resources that RTCS needs, and creates RTCS tasks.

## 2.8 Changing RTCS Running Parameters

RTCS uses some global variables, after an application has created them. All the variables have default values, most of which you need never change. If you want to change the values, an application can do so anytime after it creates RTCS; that is, anytime after it calls **RTCS\_create()**.

To do this:	Change this variable to TRUE:
To enable IP forwarding and Network Address Translation (required for NAT or IPSHield).	_IP_forward
To not verify the TCP checksums on incoming packets.	_TCP_bypass_rx
To not generate the TCP checksums on outgoing packets.	_TCP_bypass_tx

### 2.8.1 Enabling IP Forwarding

This parameter provides the ability to route packets between network interfaces (required for NAT or IPSHield).

## 2.8.2 Bypassing TCP Checksums

In isolated networks, if the performance of data transfer is an issue, you might want to bypass the generation and verification of TCP checksums.

If you bypass the verification of TCP checksums on incoming packets, RTCS does not detect errors that occur in the data stream. However, the probability of these errors is low, because the underlying layer also includes a checksum that detects errors in the data stream.

<b>Note</b>	If you bypass the generation of TCP checksums on outgoing packets, you violate the TCP specification.
-------------	---

## 2.9 Initializing Device Interfaces

RTCS supports any driver written to a published standard, such as PPP, IPCP, and PPP over Ethernet.

Because RTCS is independent of devices, it has no built-in knowledge of the device or devices that an application is using or plans to use to connect to a network. Therefore, an application must:

- Initialize each interface to each device.
- Put each interface in a state, such that the interface can send and receive network traffic.
- Dynamically add to RTCS each interface per supported device.

When the application initializes an interface to a device, the initialization function returns a handle to the interface. The application subsequently references this device handle to add the interface to RTCS, and bind IP addresses to it.

### 2.9.1 Initializing Interfaces to Ethernet Devices

Before an application can use an interface to the ethernet device, it must initialize the device-driver interface by calling **ENET\_initialize()**. The function does the following:

- It initializes the ethernet hardware, and makes it ready to send and receive ethernet packets.
- It installs the ethernet driver's interrupt service routine (ISR).
- It sets up the send and receive buffers, which are usually representations of the ethernet device's own buffers.
- It allocates and initializes the ethernet device handle, which the application subsequently uses with other functions from the ethernet driver API (**ENET\_get\_stats()**) and from the RTCS API.

#### 2.9.1.1 Getting Ethernet Statistics

To get statistics about ethernet interfaces, call **ENET\_get\_stats()**, passing to it the device handle to the interface.

### 2.9.2 Initializing Interfaces to Point-to-Point Devices

Point-to-point devices include devices that use PPP, and PPP over Ethernet. For information about initializing interfaces to point-to-point devices see [Chapter 4, “Point-to-Point Drivers.”](#)



## 2.10 Adding Device Interfaces to RTCS

After an application has initialized device interfaces, it adds each interface to RTCS by calling **RTCS\_if\_add()** with the device handle.

### 2.10.1 Removing Device Interfaces from RTCS

To remove a device interface from RTCS, call **RTCS\_if\_remove()** with the device handle.

## 2.11 Binding IP Addresses to Device Interfaces

After an application has added device interfaces to RTCS, it binds one or more IP addresses to each.

An application can bind IP addresses to device interfaces in a number of ways.

To do this:		Call:
Bind an IP address that the application specifies.		<b>RTCS_if_bind()</b>
Bind an IP address that is obtained by using:		
	BootP	<b>RTCS_if_bind_BOOTP()</b>
	DHCP	<b>RTCS_if_bind_DHCP()</b>
	IPCP (the only method that can be used for PPP)	<b>RTCS_if_bind_IPCP()</b>

### 2.11.1 Unbinding IP Addresses from Device Interfaces

To unbind an IP address from a device interface, call **RTCS\_if\_unbind()**.

## 2.12 Adding Gateways

RTCS uses gateways to communicate with remote subnets. Although an application usually adds gateways when it sets up the RTCS, it can do so anytime. To add a gateway, call **RTCS\_gate\_add()** with the IP address of the gateway and a network mask.

### 2.12.1 Adding Default Gateways

To add a default gateway, call:

```
RTCS_gate_add(ip_address, 0, 0)
```

### 2.12.2 Adding Gateways to a Specific Route

To add a gateway with address *ip\_address* to reach subnet 192.168.1.0/24, call:

```
RTCS_gate_add(ip_address, 0xC0A80100, 0xFFFFFFFF00)
```

### 2.12.3 Removing Gateways

To remove a gateway, call **RTCS\_gate\_remove()**.

## 2.13 Downloading and Running a Boot File

After an application has bound at least one IP address to each interface, it can download and run a boot file. The format of the boot file depends on the output of the compiler that you use.

To get a boot file of this format and download and run the boot file:	Call:
Binary code	<b>RTCS_exec_TFTP_BIN()</b>
Common Object File Format	<b>RTCS_exec_TFTP_COFF()</b>
Motorola S-Records	<b>RTCS_exec_TFTP_SREC()</b>

## 2.14 Enabling RTCS Logging

You can enable RTCS event logging in the MQX kernel log. Performance analysis tools can use kernel-log data to analyze, how an application operates, and how it uses resources.

Before you enable RTCS logging, you must have MQX (RTCS library) compiled with `RTCS_CFG_LOGGING` defined to 1 (for kernel log compilation parameters read *MQX User's Guide*).

In application, user must create the kernel log and enable RTCS logging (`KLOG_RTCS_FUNCTIONS`) - better description for kernel log can be found in *MQX User's Guide*. Final step to enable RTCS event logging is calling **RTCSLOG\_enable()** with required event mask. To disable RTCS event logging, call **RTCSLOG\_disable()**.

## 2.15 Starting Network Address Translation

NAT allows sites using private addresses to initiate uni-directional, outbound access to a host on an external network. Network address port translation is supported.

When NAT is enabled, a block of external, routable IP addresses is reserved by the NAT router (RTCS in this case) to represent the private, unroutable addresses of the hosts behind the border router. A large pool of hosts can share the NAT connection with a small pool of routable addresses.

When a packet leaves the private network, the border router translates the source IP address to an address from the reserved pool, and also translates the source transport identifier (TCP/UDP port or ICMP query ID) to a random number of its choosing. When responses come back, the border router is able to untranslate the random NAT-flow identifier, map that info back to the original sender IP address, and transport identifier of the host on the private network.

The router translates the destination address and related fields of all inbound packets into the addresses, transport IDs, and related fields of hosts on the private network.

To start Network Address Translation, the application calls **NAT\_init()** with the private network address and the subnet mask of the private network. For Network Address Translation to begin, the global RTCS running parameter `_IP_forward` must be TRUE.

At initialization time, space for an internal configuration structure is allocated. The configuration structure:

- Partitions the address space.
- Maintains state information.
- Points to a list of application-level gateways.
- Provides connection-timeout settings for inactive sessions.
- Identifies the ports and ICMP query IDs that are managed through NAT on the private network.

### 2.15.1 Changing Inactivity Timeouts

Once started, NAT uses the RTCS event queue to monitor sessions between a private and public host. An event timer is used to determine, when a session is over. The amount of time to wait, before terminating an inactive UDP or TCP session, is defined in the *nat.h* header file, and is dynamically configurable through the **setsockopt()** function.

When **setsockopt()** is called, the application passes to it the address of the NAT timeout structure, *nat\_timeouts*. The structure provides three inactivity timeout values for:

- TCP sessions — default timeout is 15 minutes.
- UDP or ICMP sessions — default timeout is five minutes.
- TCP sessions, in which a FIN or RST bit has been set — default timeout is two minutes.

All three values are overwritten each time the application provides a *nat\_timeouts* structure. To avoid changing an existing timeout value, the application must supply a zero value for that particular timeout.

### 2.15.2 Specifying Port Ranges

During a session, NAT uses all ports within a specified range, as defined in the *nat.h* header file. The range of ports can be changed dynamically through the **setsockopt()** function, which accepts a NAT port structure, *nat\_ports*. The structure provides the lower and higher bound of port numbers used by NAT (TCP, UDP, and ICMP ID). By default, the minimum port number is 10000, and the maximum port number is 20000.

The minimum and maximum port numbers are overwritten each time the application provides a *nat\_ports* structure. To avoid changing an existing port number, the application must supply a zero value for the minimum or maximum.

The application must not use reserved ports, and ICMP queries should not use these ports as sequence numbers. When the session is over, NAT performs address unbinding and cleans up automatically.

### 2.15.3 Disabling NAT Application-Level Gateways

The active TFTP ALG and FTP ALG are resident on the NAT device when NAT is started. If they are not needed to perform application-specific payload monitoring and alterations, they can be disabled by

redefining the *NAT\_alg\_table* table at compile time. The table corrects and acknowledges numbers with source or destination port TFTP and FTP.

The *NAT\_alg\_table* table is defined in *natalg.c*. It contains an array of function pointers to ALGs. An application can use only the ALGs that are in the table. When you remove an ALG from the table, RTCS does not link the associated code with your application.

By default, the table is defined as follows:

```
NAT_ALG NAT_alg_table[] = {
    NAT_ALG_TFTP,
    NAT_ALG_FTP,
    NAT_ALG_ENDLIST
};
```

To disable TFTP, FTP, and NAT payload monitoring and alterations, redefine the table as follows at compile time:

```
NAT_ALG NAT_alg_table[] = {
    NAT_ALG_ENDLIST
};
```

## 2.15.4 Getting NAT Statistics

Statistics are supplied through a *NAT\_STATS* structure, which is defined in *nat.h*. To get NAT statistics, the application calls [NAT\\_stats\(\)](#).

## 2.15.5 Supported Protocols

The Freescale MQX implementation of NAT supports communications using the following protocols:

- TCP and UDP sessions that do not contain port or address information in their data
- ICMP
- HTTP
- Telnet
- RPC and Portmapper
- Echo
- Quote of the day
- TFTP and FTP

NAT has no effect on packets that are passed between hosts inside the private network, regardless of the protocol that is being used to transfer the packet. For more information about NAT, see [Section Appendix A, “Protocols and Policies.”](#)

### 2.15.5.1 Limitations

The Freescale MQX implementation of NAT does not support:

- IGMP and IP multicast modes
- fragmented TCP and UDP packets
- IKE and IPsec
- SNMP
- public DNS queries of private hosts
- H.323
- peer-to-peer connections (Only the private host can initiate a connection to the public host.)

In addition, the Freescale MQX implementation of NAT can operate only on a border router for a single private network.

**Table 2-2. Summary: Setup Functions**

<b>NAT_close</b>	Stops Network Address Translation.
<b>NAT_init</b>	Starts Network Address Translation.
<b>RTCS_create</b>	Creates the RTCS.
<b>RTCS_exec_TFTP_BIN</b>	Downloads and runs a binary file.
<b>RTCS_exec_TFTP_COFF</b>	Downloads and runs a COFF file.
<b>RTCS_exec_TFTP_SREC</b>	Downloads and runs an S-Record file.
<b>RTCS_gate_add</b>	Adds a gateway to RTCS.
<b>RTCS_gate_remove</b>	Removes a gateway from RTCS.
<b>RTCS_if_add</b>	Adds a device interface to RTCS.
<b>RTCS_if_bind</b>	Binds an IP address to a device interface.
<b>RTCS_if_bind_BOOTP</b>	Uses BootP to get an IP address to bind to a device interface.
<b>RTCS_if_bind_DHCP</b>	Uses DHCP to get an IP address to bind to a device interface.
<b>RTCS_if_bind_IPCP</b>	Binds an IP address to a PPP link.
<b>RTCS_if_remove</b>	Removes a device interface from RTCS.
<b>RTCS_if_unbind</b>	Unbinds an IP address from a device interface.
<b>RTCSLOG_enable</b>	Enables RTCS event logging.
<b>RTCSLOG_disable</b>	Disables RTCS event logging.
<b>setsockopt</b>	Sets the NAT options.

## 2.15.6 Example: Setting Up RTCS

Set up RTCS with one PPP device and one ethernet device.

```
_rtcs_if_handle  ihandle;
uint_32          error;

/* For Ethernet driver: */
_enet_handle     ehandle;

/* For PPP Driver: */
FILE_PTR         pfile;
_iopcb_handle    pio;
_ppp_handle      phandle;
IPCP_DATA_STRUCT ipcp_data;
LWSEM_STRUCT     ppp_sem;
static void      PPP_linkup (pointer lwsem){_lwsem_post(lwsem);}

/* Change the priority: */
_RTCSTASK_priority = 7;

error = RTCS_create();

if (error) {
    printf("\nFailed to create RTCS, error = %X", error);
    return;
}

/* Enable IP forwarding: */
_IP_forward = TRUE;

/* Set up the Ethernet driver: */
error = ENET_initialize(ENET_DEVICE, enet_local, 0, &ehandle);
if (error) {
    printf("\nFailed to initialize Ethernet driver: %s",
        ENET_strerror(error));
    return;
}
error = RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);
if (error) {
    printf("\nFailed to add interface for Ethernet, error = %x",
        error);
    return;
}
error = RTCS_if_bind(ihandle, enet_ipaddr, enet_ipmask);
if (error) {
    printf("\nFailed to bind interface for Ethernet, error = %x",
        error);
    return;
}
printf("\nEthernet device %d bound to %X",
    ENET_DEVICE, enet_ipaddr);

/*Set up PPP Driver: */
pfile = fopen(PPP_DEVICE, NULL);
pio = _iopcb_ppphdlc_init(pfile);
error = PPP_initialize(pio, &phandle);
if (error) {
```

```

    printf("\nFailed to initialize PPP Driver: %x", error);
    return;
}
_iopcb_open(pio, PPP_lowerup, PPP_lowerdown, phandle);
error = RTCS_if_add(phandle, RTCS_IF_PPP, &ihandle);
if (error) {
    printf("\nFailed to add interface for PPP, error = %x", error);
    return;
}
_lwsem_create(&ppp_sem, 0);
_mem_zero(&ipcp_data, sizeof(ipcp_data));
ipcp_data.IP_UP = PPP_linkup;
ipcp_data.IP_DOWN = NULL;
ipcp_data.IP_PARAM = &ppp_sem;
ipcp_data.ACCEPT_LOCAL_ADDR = FALSE;
ipcp_data.ACCEPT_REMOTE_ADDR = FALSE;
ipcp_data.LOCAL_ADDR = PPP_LOCADDR;
ipcp_data.REMOTE_ADDR = PPP_PEERADDR;
ipcp_data.DEFAULT_NETMASK = TRUE;
ipcp_data.DEFAULT_ROUTE = TRUE;
error = RTCS_if_bind_IPCP(ihandle, &ipcp_data);
if (error) {
    printf("\nFailed to bind interface for PPP, error = %x", error);
    return;
}
_lwsem_wait(&ppp_sem);
printf("\nPPP device %s bound to %X", PPP_DEVICE, ipcp_data.LOCAL_ADDR);

/* Install a default gateway: */
RTCS_gate_add(GATE_ADDR, INADDR_ANY, INADDR_ANY);

```

## 2.16 Compile-Time Options

RTCS is built with certain features that you can include or exclude by changing the value of compile-time configuration options. If you change a value, you must rebuild RTCS. For information about rebuilding RTCS, see [Chapter 6, “Rebuilding.”](#)

Similarly as the PSP, BSP, or other system libraries included in the Freescale MQX RTOS, the RTCS build projects takes its compile-time configuration options from the central user-configuration file *user\_config.h*. This file is located in board-specific subdirectory in top-level *config* folder.

The list of all configuration macros and their default values is defined in the *source\include\rtscsf.h* file. This file is not intended to be modified by user. Thanks to proper include search paths set in the RTCS build project, the *rtscsf.h* file includes the *user\_config.h* file from the board-specific configuration directory and uses the configuration options suitable for the given board.

To do this:	Set the option value to:
Include the option.	1
Exclude the option.	0

## 2.16.1 Recommended Settings

The settings that you choose for compile-time configuration options depend on the requirements of your application. Table 2-3 illustrates some common settings that you might want to use as you develop your application.

Table 2-3. Recommended Compile-Time Settings

Option	Default	Debug	Speed	Size
RTCSCFG_CHECK_ADDRSIZE	1	1	0	0
RTCSCFG_CHECK_ERRORS	1	1	0	0
RTCSCFG_CHECK_MEMORY_ALLOCATION_ERRORS	1	1	1	1
RTCSCFG_CHECK_VALIDITY	1	1	0	0
RTCSCFG_IP_DISABLE_DIRECTED_BROADCAST	0	0	0	0
RTCSCFG_LINKOPT_8021Q_PRIO	0	0, 1	0, 1	0, 1
RTCSCFG_LINKOPT_8023	0	0, 1	0, 1	0, 1
RTCSCFG_LOG_PCB	1	1	0	0
RTCSCFG_LOG_SOCKET_API	1	1	0	0

## 2.16.2 Configuration Options and Default Settings

The default values are defined in *rtcs/include/rtcscfg.h*. You may override the settings from the *user\_config.h* user configuration file.

### 2.16.2.1 RTCSCFG\_CHECK\_ADDRSIZE

By default, for functions that take a parameter that is a pointer to [sockaddr\\_in](#), RTCS determines whether the *addrlen* field is at least *sizeof(sockaddr\_in)* bytes.

If *addrlen* is not at least this size, RTCS does either of the following:

- It returns an error, when these functions are called:
  - **bind()**
  - **connect()**
  - **sendto()**
- It performs a partial copy operation, when these functions are called:
  - **accept()**
  - **getsockname()**
  - **getpeername()**
  - **recvfrom()**



### 2.16.2.2 RTCS\_CFG\_CHECK\_ERRORS

By default, RTCS API functions perform error checking on their parameters.

### 2.16.2.3 RTCS\_CFG\_CHECK\_MEMORY\_ALLOCATION\_ERROR

By default, RTCS API functions perform error checking, when they allocate memory.

### 2.16.2.4 RTCS\_CFG\_CHECK\_VALIDITY

By default, RTCS accesses its internal data structures, it determines, whether the VALID field in the structures is valid.

### 2.16.2.5 RTCS\_CFG\_IP\_DISABLE\_DIRECTED\_BROADCAST

By default, RTCS receives and forwards directed broadcast datagrams. Set this value to 1 (one) to reduce the risk of Smurf ICMP echo-request DoS attacks

### 2.16.2.6 RTCS\_CFG\_BOOTP\_RETURN\_YIADDR

When RTCS\_CFG\_BOOTP\_RETURN\_YIADDR is 1, the BOOTP\_DATA\_STRUCT has an additional field, which will be filled in with the YIADDR field of the BOOTREPLY.

### 2.16.2.7 RTCS\_CFG\_UDP\_ENABLE\_LBOUND\_MULTICAST

When RTCS\_CFG\_UDP\_ENABLE\_LBOUND\_MULTICAST is 1, locally bound sockets that are members of multicast groups will be able to receive messages sent to both their unicast and multicast addresses.

### 2.16.2.8 RTCS\_CFG\_LINKOPT\_8021Q\_Prio

By default, RTCS does not send and receive Ethernet 802.1Q priority tags. Set this value to 1 (one) to have RTCS send and receive Ethernet 802.1Q priority tags

### 2.16.2.9 RTCS\_CFG\_LINKOPT\_8023

By default, RTCS sends and receives Ethernet II frames. Set this value to 1 (one) to have RTCS send and receive both Ethernet 802.3 and Ethernet II frames.

### 2.16.2.10 RTCS\_CFG\_DISCARD\_SELF\_BCASTS

By default, controls whether or not to discard all broadcast packets that we sent, as they are likely echoes from older hubs.

### 2.16.2.11 RTCS\_MINIMUM\_FOOTPRINT

Default 0. Set to 1 to enable RTCS optimizations for small RAM devices. Setting this parameter 1 causes the RTCS\_CFG\_FEATURE\_DEFAULT setting to 0 automatically.

### **2.16.2.12 RTCSCFG\_FEATURE\_DEFAULT**

This parameter is used to determine the default enable/disable state of RTCS features.

### **2.16.2.13 RTCSCFG\_ENABLE\_ICMP**

Default value RTCSCFG\_FEATURE\_DEFAULT. Set to 1 to add support for ICMP protocol.

### **2.16.2.14 RTCSCFG\_ENABLE\_IGMP**

Default value RTCSCFG\_FEATURE\_DEFAULT. Set to 1 to add support for IGMP protocol.

### **2.16.2.15 RTCSCFG\_ENABLE\_NAT**

Default 0. Set to 1 for add support for NAT functionality.

### **2.16.2.16 RTCSCFG\_ENABLE\_DNS**

Default value RTCSCFG\_FEATURE\_DEFAULT. Set to 1 to add support for DNS.

### **2.16.2.17 RTCSCFG\_ENABLE\_LWDNS**

Default 0. Set to 1 for add implement light weight DNS functionality only.

### **2.16.2.18 RTCSCFG\_ENABLE\_IPIP**

Default value RTCSCFG\_FEATURE\_DEFAULT. Set to 1 to to add support for IPIP.

### **2.16.2.19 RTCSCFG\_ENABLE\_RIP**

Default value RTCSCFG\_FEATURE\_DEFAULT. Set to 1 to add support for RIP.

### **2.16.2.20 RTCSCFG\_ENABLE\_SNMP**

Default value RTCSCFG\_FEATURE\_DEFAULT. Set to 1 to add support for SNMP.

### **2.16.2.21 RTCSCFG\_ENABLE\_IP\_REASSEMBLY**

Default value RTCSCFG\_FEATURE\_DEFAULT, add support for IP packet reassembling.

### **2.16.2.22 RTCSCFG\_ENABLE\_LOOPBACK**

Default value RTCSCFG\_FEATURE\_DEFAULT. Set to 1 to enable loopback interface.

### **2.16.2.23 RTCSCFG\_ENABLE\_UDP**

Default value RTCSCFG\_FEATURE\_DEFAULT. Set to 1 to add support for UDP protocol.

#### **2.16.2.24 RTCSCFG\_ENABLE\_TCP**

Default value RTCSCFG\_FEATURE\_DEFAULT. Set to 1 to add support for TCP protocol.

#### **2.16.2.25 RTCSCFG\_ENABLE\_STATS**

Default value RTCSCFG\_FEATURE\_DEFAULT. Set to 1 to add support for network traffic statistics.

#### **2.16.2.26 RTCSCFG\_ENABLE\_GATEWAYS**

Default value RTCSCFG\_FEATURE\_DEFAULT. Set to 1 to add support for gateways.

#### **2.16.2.27 RTCSCFG\_ENABLE\_VIRTUAL\_ROUTES**

Default value RTCSCFG\_FEATURE\_DEFAULT. Must be 1 for PPP or tunneling.

#### **2.16.2.28 RTCSCFG\_USE\_KISS\_RNG**

Default 0. Must be 1 for PPP or tunneling.

#### **2.16.2.29 RTCSCFG\_ENABLE\_ARP\_STATS**

Default value RTCSCFG\_FEATURE\_DEFAULT. Set to 1 to enable ARP packet statistics.

#### **2.16.2.30 RTCSCFG\_PCBS\_INIT**

PCB (Packet Control Block) initial allocated count. Override in application by setting the \_RTCSPCB\_init global variable.

#### **2.16.2.31 RTCSCFG\_PCBS\_GROW**

PCB (Packet Control Block) allocation grow granularity. Override in application by setting the \_RTCSPCB\_grow global variable.

#### **2.16.2.32 RTCSCFG\_PCBS\_MAX**

PCB (Packet Control Block) maximum allocated count. Override in application by setting the \_RTCSPCB\_max global variable.

#### **2.16.2.33 RTCSCFG\_MSGPOOL\_INIT**

RTCS message pool initial size. Override in application by setting the \_RTCS\_msgpool\_init variable.

#### **2.16.2.34 RTCSCFG\_MSGPOOL\_GROW**

RTCS message pool growing granularity. Override in application by setting the \_RTCS\_msgpool\_grow variable.

### **2.16.2.35 RTCSCFG\_MSGPOOL\_MAX**

RTCS message pool maximal size. Override in application by setting the `_RTCS_msgpool_max` variable.

### **2.16.2.36 RTCSCFG\_SOCKET\_PART\_INIT**

RTCS socket pre-allocated count. Override in application by setting the `_RTCS_socket_part_init`.

### **2.16.2.37 RTCSCFG\_SOCKET\_PART\_GROW**

RTCS socket allocation grow granularity. Override in application by setting the `_RTCS_socket_part_grow`.

### **2.16.2.38 RTCSCFG\_SOCKET\_PART\_MAX**

RTCS socket maximum count. Override in application by setting the `_RTCS_socket_part_max`.

### **2.16.2.39 RTCSCFG\_UDP\_MAX\_QUEUE\_SIZE**

UDP maximum queue size. Override in application by setting the `_UDP_max_queue_size`.

### **2.16.2.40 RTCSCFG\_ENABLE\_UDP\_STATS**

Set to 0 for disable UDP statistics.

### **2.16.2.41 RTCSCFG\_ENABLE\_TCP\_STATS**

Set to 0 for disable TCP statistics.

### **2.16.2.42 RTCSCFG\_TCP\_MAX\_CONNECTIONS**

Default value 0. Maximum number of simultaneous connections allowed. Define as 0 for no limit.

### **2.16.2.43 RTCSCFG\_TCP\_MAX\_HALF\_OPEN**

Default value 0. Maximum number of simultaneous half open connections allowed. Define as 0 to disable the SYN attack recovery feature.

### **2.16.2.44 RTCSCFG\_ENABLE\_RIP\_STATS**

Default value `RTCSCFG_ENABLE_STATS`, enable RIP statistics.

### **2.16.2.45 RTCSCFG\_QUEUE\_BASE**

Override in application by setting `_RTCSQUEUE_base`.

### **2.16.2.46 RTCSCFG\_STACK\_SIZE**

Override in application by setting `_RTCSTACK_stacksize`.

### 2.16.2.47 RTCSCFG\_LOG\_PCB

By default, RTCS logs packet generation and parsing in the MQX kernel log, subject to whether the application calls **RTCSLOG\_enable()**. Set this value to 0 (zero) to have RTCS not log packets, even if the application calls **RTCSLOG\_enable()**.

### 2.16.2.48 RTCSCFG\_LOG\_SOCKET\_API

By default, RTCS logs socket API calls in the MQX kernel log, subject to whether the application calls **RTCSLOG\_enable()**. Set this value to 0 (zero) to have RTCS not log socket API calls, even if the application calls **RTCSLOG\_enable()**.

## 2.16.3 Application specific default settings

### 2.16.3.1 FTP Client

#### 2.16.3.1.1 FTPCCFG\_SMALL\_FILE\_PERFORMANCE\_ENANCEMENT

Set to 1 - better performance for small files - less than 4MB.

#### 2.16.3.1.2 FTPCCFG\_BUFFER\_SIZE

FTP Client buffer size.

#### 2.16.3.1.3 FTPCCFG\_WINDOW\_SIZE

FTP Client maximum TCP packet size.

### 2.16.3.2 FTP Server

#### 2.16.3.2.1 FTPDCFG\_SHUTDOWN\_OPTION

Flags used in shutdown() for close connection. Default value FLAG\_ABORT\_CONNECTION.

#### 2.16.3.2.2 FTPDCFG\_DATA\_SHUTDOWN\_OPTION

Flags used in shutdown() for data termination. Default value FLAG\_CLOSE\_TX.

#### 2.16.3.2.3 FTPDCFG\_USES\_MFS

Enable MFS support.

#### 2.16.3.2.4 FTPDCFG\_ENABLE\_MULTIPLE\_CLIENTS

Enable simultaneous client connections.

#### **2.16.3.2.5 FTPDCFG\_ENABLE\_USERNAME\_AND\_PASSWORD**

Set to 1 for request user name and password for connect to server.

#### **2.16.3.2.6 FTPDCFG\_ENABLE\_RENAME**

Default value 1.

#### **2.16.3.2.7 FTPDCFG\_WINDOW\_SIZE**

Maximum TCP packet size. Override in application by setting FTPd\_window\_size.

#### **2.16.3.2.8 FTPDCFG\_BUFFER\_SIZE**

FTP Server buffer size. Override in application by setting FTPd\_buffer\_size

#### **2.16.3.2.9 FTPDCFG\_CONNECT\_TIMEOUT**

Connection timeout.

#### **2.16.3.2.10 FTPDCFG\_SEND\_TIMEOUT**

Sending timeout.

#### **2.16.3.2.11 FTPDCFG\_TIMEWAIT\_TIMEOUT**

The timeout.

### **2.16.3.3 Telnet**

#### **2.16.3.3.1 TELNETDCFG\_BUFFER\_SIZE**

Telnet Server buffer size.

#### **2.16.3.3.2 TELNETDCFG\_NOWAIT**

Enable nonblocking functionality. Default value FALSE.

#### **2.16.3.3.3 TELNETDCFG\_ENABLE\_MULTIPLE\_CLIENTS**

Enable simultaneous client connections. Default value RTCSCFG\_FEATURE\_DEFAULT.

#### **2.16.3.3.4 TELENETDCFG\_CONNECT\_TIMEOUT**

Connection timeout.

#### **2.16.3.3.5 TELENETDCFG\_SEND\_TIMEOUT**

Sending timeout.

### 2.16.3.3.6 TELENETDCFG\_TIMEWAIT\_TIMEOUT

The timeout.

### 2.16.3.4 SNMP

#### 2.16.3.4.1 RTCSCFG\_ENABLE\_SNMP\_STATS

Enable SNMP statistics. Default value RTCSCFG\_ENABLE\_STATS.

### 2.16.3.5 IPCFG

#### 2.16.3.5.1 RTCSCFG\_IPCFG\_ENABLE\_DNS

Enable DNS name resolving (depends on [RTCSCFG\\_ENABLE\\_DNS](#), [RTCSCFG\\_ENABLE\\_UDP](#) and [RTCSCFG\\_ENABLE\\_LWDNS](#))

#### 2.16.3.5.2 RTCSCFG\_IPCFG\_ENABLE\_DHCP

Enable DHCP binding (depends on [RTCSCFG\\_ENABLE\\_UDP](#)).

#### 2.16.3.5.3 RTCSCFG\_IPCFG\_ENABLE\_BOOT

Enable TFTP names processing and BOOT binding.

### 2.16.4 HTTP Server default configuration

#### 2.16.4.1 HTTPDCFG\_POLL\_MODE

Default 1. Set to 1 to run HTTP Server in poll mode (all sessions handled by a single task). Set to 0 to handle each HTTP session in a different task.

#### 2.16.4.2 HTTPDCFG\_DEF\_PORT

HTTP Server listen port. Default value 80. Override in application when initializing the HTTP server.

#### 2.16.4.3 HTTPDCFG\_DEF\_INDEX\_PAGE

HTTP Server index page filename. Default value “index.htm”. Override in application when initializing the HTTP server.

#### 2.16.4.4 HTTPDCFG\_DEF\_SES\_CNT

Maximal HTTP server session count - count of simultaneous evaluated requests. Default value 2. Override in application when initializing the HTTP server.

#### **2.16.4.5 HTTPDCFG\_DEF\_URL\_LEN**

Maximal evaluated URL length. Default value 128. Override in application when initializing the HTTP server.

#### **2.16.4.6 HTTPDCFG\_DEF\_AUTH\_LEN**

Maximal length for evaluated authorization string in http request header. Default value 16. Override in application when initializing the HTTP server. Override in application when initializing the HTTP server.

#### **2.16.4.7 HTTPDCFG\_MAX\_BYTES\_TO\_SEND**

Maximal send length in step - block size. Default value 512.

#### **2.16.4.8 HTTPDCFG\_MAX\_SCRIPT\_LN**

Maximal evaluated script line length. Default value 16.

#### **2.16.4.9 HTTPDCFG\_RECV\_BUF\_LEN**

Receiving temporary buffer size. Default value 32.

#### **2.16.4.10 HTTPDCFG\_MAX\_HEADER\_LEN**

Maximal response header length. Default value 256.

#### **2.16.4.11 HTTPDCFG\_SES\_TO**

Session timeout. Default value 20000ms.

#### **2.16.4.12 HTTPCFG\_TX\_WINDOW\_SIZE**

Maximum transmit packet size.

#### **2.16.4.13 HTTPCFG\_RX\_WINDOW\_SIZE**

Maximum receive packet size.



## Chapter 3 Using Sockets

### 3.1 Before You Begin

This chapter describes, how to use RTCS and its sockets. After an application sets up RTCS, it uses a socket interface to communicate with other applications or servers over a TCP/IP network.

For information about	See
Data types mentioned in this chapter	<a href="#">Chapter 8, “Data Types”</a>
MQX	<i>MQX User’s Guide</i> <i>MQX Reference</i>
Protocols	<a href="#">Section Appendix A, “Protocols and Policies”</a>
Prototypes for functions mentioned in this chapter	<a href="#">Chapter 7, “Function Reference”</a>
Setting up RTCS	<a href="#">Chapter 2, “Setting Up the RTCS”</a>

<b>Note</b>	Remember, you can change RTCS running parameters anytime. RTCS uses some global variables after an application has created it. All the variables have default values, most of which you need never change. If you want to change the values, an application can do so anytime after it creates RTCS; that is, anytime after it calls <b>RTCS_create()</b> .
-------------	---

### 3.2 Protocols Supported

RTCS sockets provide an interface to the following protocols:

- TCP
- UDP

### 3.3 Socket Definition

A socket is an abstraction that identifies an endpoint and includes:

- A type of socket; one of:
  - datagram (uses UDP)
  - stream (uses TCP)
- A socket address, which is identified by:
  - port number

— IP address

A socket might have a remote endpoint.

### 3.4 Socket Options

Each socket has socket options, which define characteristics of the socket, such as:

- checksum calculations
- ethernet-frame characteristics
- IGMP membership
- non-blocking (nowait options)
- push operations
- sizes of send and receive buffers
- timeouts

### 3.5 Comparison of Datagram and Stream Sockets

Table 3-1 gives an overview of the differences between datagram and stream sockets.

**Table 3-1. Datagram and Stream Sockets**

	<b>Datagram socket</b>	<b>Stream socket</b>
Protocol	UDP	TCP
Connection-based	No	Yes
Reliable transfer	No	Yes
Transfer mode	Block	Character

### 3.6 Datagram Sockets

#### 3.6.1 Connectionless

A datagram socket is connectionless in that an application uses a socket without first establishing a connection. Therefore, an application specifies the destination address and destination port number for each data transfer. An application can pre-specify a remote endpoint for a datagram socket, if desired.

### 3.7 Unreliable Transfer

A datagram socket is used for datagram-based data transfer, which does not acknowledge the transfer. Because delivery is not guaranteed, the application is responsible for ensuring that the data is acknowledged when necessary.

## 3.8 Block-Oriented

A datagram socket is block-oriented. This means that when an application sends a block of data, the bytes of data remain together. If an application writes a block of data of, say, 100 bytes, RTCS sends the data to the destination in a single packet, and the destination receives 100 bytes of data.

## 3.9 Stream Sockets

## 3.10 Connection-Based

A stream-socket connection is uniquely defined by an address-port number pair for each of the two endpoints in the connection. For example, a connection to a Telnet server uses the local IP address with a local port number, and the server's IP address with port number 23.

## 3.11 Reliable Transfer

A stream socket provides reliable, end-to-end data transfer. To use stream sockets, a client establishes a connection to a peer, transfers data, and then closes the connection. Barring physical disconnection, RTCS guarantees that all sent data is received in sequence.

## 3.12 Character-Oriented

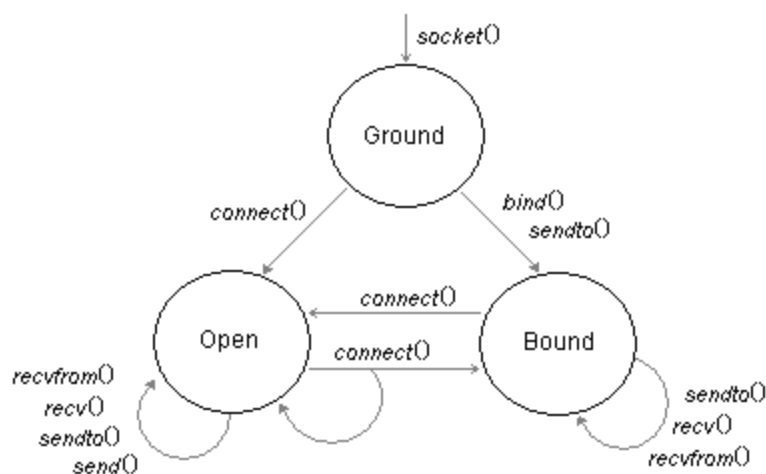
A stream socket is character-oriented. This means that RTCS might split or merge bytes of data, as it sends the data from one protocol stack to another. An application on a stream socket might perform, for example, two successive write operations of 100 bytes each, and RTCS might send the data to the destination in a single packet. The destination might then receive the data using, for example, four successive read operations of 50 bytes each.

## 3.13 Creating and Using Sockets

An application follows the following general steps to create and use sockets. The steps are summarized in the following diagrams and described in subsequent sections.

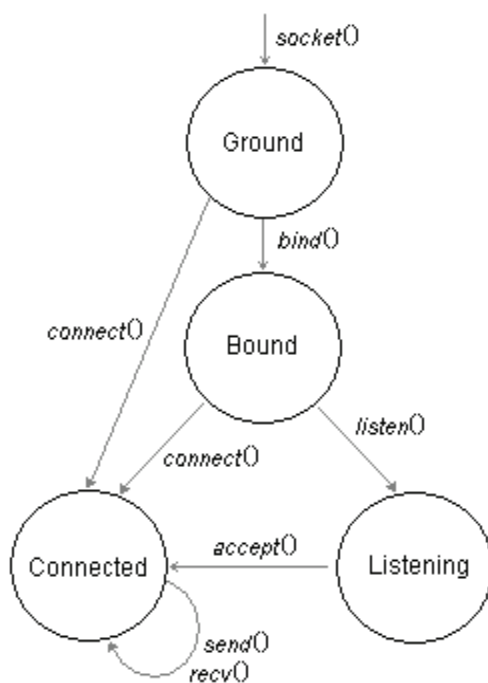
- Create a new socket by calling **socket()**, indicating whether the socket is a datagram socket or a stream socket.
- Bind the socket to a local address by calling **bind()**.
- If the socket is a stream socket, assign a remote IP address by doing one of the following:
  - Calling **connect()**.
  - Calling **listen()** followed by **accept()**.
- Send data by calling **sendto()** for a datagram socket, or **send()** for a stream socket.
- Receive data by calling **recvfrom()** for a datagram socket, or **recv()** for a stream socket.
- When data transfer is finished, optionally destroy the socket by calling **shutdown()**.

The process for datagram sockets is illustrated in Figure 3-1.



**Figure 3-1. Creating and Using Datagram Sockets (UDP)**

The process for stream sockets is illustrated in Figure 3-2.



**Figure 3-2. Creating and Using Stream Sockets (TCP)**

## 3.14 Creating Sockets

To create a socket, an application calls **socket()** and specifies, whether the socket is a datagram socket or a stream socket. The function returns a socket handle, which the application subsequently uses to access the socket.

## 3.15 Changing Socket Options

When RTCS creates a socket, it sets all the socket options to default values. To change the value of certain options, an application must do so before it binds the socket. An application can change other options anytime.

All socket options and their default values are described in the listing for **setsockopt()** in [Chapter 7, “Function Reference.”](#)

## 3.16 Binding Sockets

After an application creates a socket and optionally changes or sets socket options, it must bind the socket to a local port number by calling **bind()**. The function defines the endpoint of the local socket by the local IP address and port number, where the application defined the local IP address by calling **RTCS\_if\_bind()**, while it was setting up RTCS.

You can specify the local port number as any number, but if you specify zero, RTCS chooses an unused port number. To determine the port number that RTCS chose, call **getsockopt()**.

After the application binds the socket, how it uses the socket depends on whether the socket is a datagram socket or a stream socket. The description of using datagram sockets follows.

## 3.17 Using Datagram Sockets

## 3.18 Setting Datagram-Socket Options

By default, RTCS uses IGMP, and, by default, a socket is not in any group. The application can change the following socket options for the socket:

- IGMP add membership
- IGMP drop membership
- send nowait
- checksum bypass

For information about the options, see the listing for **setsockopt()** in [Chapter 7, “Function Reference.”](#)

For information about how to change the default behavior so that RTCS does not use IGMP, see [Section 2.5, “Defining RTCS Protocols.”](#)

## 3.19 Transferring Datagram Data

An application transfers data by making calls to **sendto()** or **send()**, and **recvfrom()** or **recv()**. With each call, RTCS either sends or receives one UDP datagram, which contains up to 65,507 bytes of data. If an application specifies more data, the functions return an error.

The functions **send()** and **sendto()** return, when the data is passed to the ethernet interface.

The functions **recv()** and **recvfrom()** return, when the socket port receives the packet, or immediately, if a queued packet is already at the port. The receive buffer should be at least as large as the largest datagram that the application expects to receive. If a packet overruns the receive buffer, RTCS truncates the packet and discards the truncated data.

### 3.19.1 Buffering

By default, **send()** and **sendto()** do not buffer outgoing data. This behavior can be changed by using either the `OPT_SEND_NOWAIT` socket option, or the `RTCS_MSG_NONBLOCK` send flag.

For incoming data, RTCS matches the data, packet by packet, to **recv()** or **recvfrom()** calls that the application makes. If a packet arrives and one of the **recv()** and **recvfrom()** calls is not waiting for data, RTCS queues the packet.

### 3.19.2 Pre-Specifying a Peer

An application can optionally pre-specify a peer by calling **connect()**. Pre-specification has the following effect:

- The **send()** function can be used to send a datagram to the peer that is specified in the call to **connect()**. Calls to **send()** fail, if **connect()** has not been called previously.
- The behavior of **sendto()** is unchanged. It is not restricted to the specified peer.
- The functions **recv()** or **recvfrom()** return datagrams that have been sent by the specified peer only.

## 3.20 Shutting Down Datagram Sockets

An application can shut down a datagram socket by calling **shutdown()**. Before the function returns, the following actions occur:

- Outstanding calls to **recvfrom()** return immediately.
- RTCS discards received packets that are queued for the socket and frees their buffers.

When **shutdown()** returns, the socket handle is invalid, and the application can no longer use the socket.

## 3.21 Using Stream Sockets

## 3.22 Changing Stream-Socket Options

An application can change the value of certain stream-socket options anytime. For details, see the listing for **setsockopt()** in [Chapter 7, “Function Reference.”](#)

## 3.23 Establishing Stream-Socket Connections

An application can establish a connection to a stream socket in one of the following ways:

- Passively — by listening for incoming connection requests (by calling **listen()** followed by **accept()**).
- Actively — by generating a connection request (by calling **connect()**).

### 3.23.1 Establishing Stream-Socket Connections Passively

By calling **listen()**, an application can passively put an unconnected socket into a listening state, after which the local socket endpoint responds to a single incoming connection request.

After it calls **listen()**, the application calls **accept()**, which returns a new socket handle, and lets the application accept the incoming connection request. Usually, the application calls **accept()** immediately after it calls **listen()**. The application uses the new socket handle for all communication with the specified remote endpoint, until one or both endpoints close the connection. The original socket remains in the listening state, and continues to be referenced by the initial socket handle that **socket()** returned.

The new socket that the listen-accept mechanism creates, inherits the socket options of the parent socket.

### 3.23.2 Establishing Stream-Socket Connections Actively

By calling **connect()**, an application can actively establish a stream-socket connection to the remote endpoint that the function specifies. If the remote endpoint is not in the listening state, **connect()** fails. Depending on the state of the remote endpoint, **connect()** fails immediately or after the time that the connect-timeout socket option specifies.

If the remote endpoint accepts the connection, the application uses the original socket handle for all its communication with that remote endpoint, and RTCS maintains the connection until either or both endpoints close the connection.

## 3.24 Getting Stream-Socket Names

After an application establishes a stream-socket connection, it can get the identifiers for the local endpoint (by calling **getsockname()**) and for the remote endpoint (by calling **getpeername()**).

## 3.25 Sending Stream Data

An application sends data on a stream socket by calling **send()**. When the function returns depends on the values of the send nowait (OPT\_SEND\_NOWAIT) socket option. An application can change the value by calling **setsockopt()**.

Send nowait (non-blocking I/O)	send() returns when:
FALSE (default)	TCP has buffered all data, but not necessarily sent it.
TRUE	Immediately (the result is a filled or partially filled buffer).

### 3.26 Receiving Stream Data

An application receives data on a stream socket by calling **recv()**. The application passes the function a buffer, into which RTCS places the incoming data. When the function returns depends on the values of the receive-nowait (**OPT\_RECEIVE\_NOWAIT**) and receive-push (**OPT\_RECEIVE\_PUSH**) socket options. The application can change the values by calling **setsockopt()**.

Receive nowait (non-blocking I/O)	Receive push (delay transmission)	recv() returns when:
FALSE (default)	TRUE (default)	One of: A push flag in the data is received. Supplied buffer is completely filled with incoming data. Receive timeout expires (the default receive timeout is an unlimited time).
FALSE (default)	FALSE	Either: Supplied buffer is completely filled with incoming data. Receive timeout expires.
TRUE	(Ignored)	Immediately after it polls TCP for any data in the internal receive buffer.

### 3.27 Buffering Data

The size of the RTCS per-socket send buffer is determined by the socket option that controls the size of the send buffer. RTCS copies data into its send buffer from the buffer that the application supplies. As the peer acknowledges the data, RTCS releases space in its buffer. If the buffer is full, calls to **send()** with the send-push (**OPT\_SEND\_PUSH**) socket option **FALSE** block, until the remote endpoint acknowledges some or all of the data.

The size of the RTCS per-socket receive buffer is determined by the socket option that controls the size of the receive buffer. RTCS uses the buffer to hold incoming data when there are no outstanding calls to **recv()**. When the application calls **recv()**, RTCS copies data from its buffer to the buffer that the application supplies, and, consequently, the remote endpoint can send more data.

### 3.28 Improving the Throughput of Stream Data

- Include the push flag in sent data only where the flag is needed; that is, at the end of a stream of data.
- Specify the largest possible send and receive buffers to reduce the amount of work that the application and RTCS do.
- When you call **recv()**, call it again immediately to reduce the amount of data that RTCS must copy into its receive buffer.
- Specify the size of the send and receive buffers to be multiples of the maximum packet size.
- Call **send()** with an amount of data that is a multiple of the maximum packet size.



## 3.29 Shutting Down Stream Sockets

An application can shut down a stream socket by calling **shutdown()** with a parameter that indicates how the socket is to be shut down: either gracefully or with an abort operation (TCP reset). The function always returns immediately.

Before **shutdown()** returns, outstanding calls to **send()** and **recv()** return immediately, and RTCS discards any data that is in its receive buffer for the socket.

### 3.29.1 Shutting Down Gracefully

If the socket is to be shut down gracefully, RTCS tries to deliver all the data that is in its send buffer for the socket. As specified by the TCP specification, RTCS maintains the socket connection for four minutes after the remote endpoint disconnects.

### 3.29.2 Shutting Down with an Abort Operation

If the socket is to be shut down with an abort operation, the following actions occur:

- RTCS immediately discards the socket and the socket's internal send and receive buffers.
- The remote endpoint frees its socket immediately after it sends all the data that is in its send buffer.

**Table 3-2. Summary: Socket Functions**

<b>accept()</b>	Accepts the next incoming stream connection and clones the socket to create a new socket, which services the connection.
<b>bind()</b>	Identifies the local application endpoint by providing a port number.
<b>connect()</b>	Establishes a stream connection with an application endpoint or sets a remote endpoint for a datagram socket.
<b>getpeername()</b>	Determines the peer address-port number endpoint of a connected socket.
<b>getsockname()</b>	Determines the local address-port number endpoint of a bound socket.
<b>getsockopt()</b>	Gets the value of a socket option.
<b>listen()</b>	Allows incoming stream connections to be received on the port number that is identified by a socket.
<b>recv()</b>	Receives data on a stream or datagram socket.
<b>recvfrom()</b>	Receives data on a datagram socket.
<b>RTCS_attachsock()</b>	Gets access to a socket that is owned by another task.
<b>RTCS_detachsock()</b>	Relinquishes ownership of a socket.
<b>RTCS_geterror()</b>	Gets the reason why an RTCS function returned an error for the socket.
<b>RTCS_selectall()</b>	Waits for activity on any socket that a caller owns.
<b>RTCS_selectset()</b>	Waits for activity on any socket in a set of sockets.
<b>send()</b>	Sends data on a stream socket or on a datagram socket, for which a remote endpoint has been specified.
<b>sendto()</b>	Sends data on a datagram socket.
<b>setsockopt()</b>	Sets the value of a socket option.
<b>shutdown()</b>	Shuts down a connection and discards the socket.
<b>socket()</b>	Creates a socket.

### 3.30 Example

A Quote of the Day server sets up a datagram socket and a stream socket. The server then loops forever. If the stream socket receives a connection request, the server accepts it and sends a quote. If the datagram socket receives data, the server sends a quote.

```

sockaddr_in  laddr, raddr;
uint_32      sock, listensock;
int_32       length;
uint_32      index;
uint_32      error;
uint_16      rlen;

```

```

/* Set up the UDP port (Quote server services port 17): */

```

```

laddr.sin_family      = AF_INET;
laddr.sin_port        = 17;
laddr.sin_addr.s_addr = INADDR_ANY;

/* Create a datagram socket: */
sock = socket(PF_INET, SOCK_DGRAM, 0);
if (sock == RTCS_SOCKET_ERROR) {
    printf("\nFailed to create datagram socket.");
    _task_block();
}
/* Bind the datagram socket to the UDP port: */
error = bind(sock, &laddr, sizeof(laddr));
if (error != RTCS_OK) {
    printf("\nFailed to bind datagram - 0x%lx.", error);
    _task_block();
}
/* Create a stream socket: */
sock = socket(PF_INET, SOCK_STREAM, 0);
if (sock == RTCS_SOCKET_ERROR) {
    printf("\nFailed to create the stream socket.");
    _task_block();
}
/* Bind the stream socket to a TCP port: */
error = bind(sock, &laddr, sizeof(laddr));
if (error != RTCS_OK) {
    printf("\nFailed to bind the stream socket - 0x%lx", error);
    _task_block();
}
/* Set up the stream socket to listen on the TCP port: */
error = listen(sock, 0);
if (error != RTCS_OK) {
    printf("\nlisten() failed - 0x%lx", error);
    _task_block();
}
listensock = sock;
printf("\n\nQuote Server is active on port 17.\n");

index = 0;
for (;;) {
    sock = RTCS_selectall(0);
    if (sock == listensock) {
        /* Connection requested; accept it. */
        rlen = sizeof(raddr);
        sock = accept(listensock, &raddr, &rlen);
        if (sock == RTCS_SOCKET_ERROR) {
            printf("\naccept() failed, error 0x%lx",
                RTCS_geterror(listensock));
            continue;
        }
        /* Send back a quote: */
        send(sock, Quotes[index], strlen(Quotes[index]) + 1, 0);
        _time_delay(1000);
        shutdown(sock, FLAG_CLOSE_TX);
    } else {
        /* Datagram socket received data. */
        memset(&raddr, 0, sizeof(raddr));
        rlen = sizeof(raddr);
    }
}

```

## Using Sockets

```
length = recvfrom(sock, NULL, 0, 0, &raddr, &rlen);
if (length == RTCS_ERROR) {
    printf("\nError %x receiving from %d.%d.%d.%d,%d",
        RTCS_geterror(sock),
        (raddr.sin_addr.s_addr >> 24) & 0xFF,
        (raddr.sin_addr.s_addr >> 16) & 0xFF,
        (raddr.sin_addr.s_addr >> 8) & 0xFF,
        raddr.sin_addr.s_addr & 0xFF,
        raddr.sin_port);
    continue;
}
/* Send back a quote: */
sendto(sock, Quotes[index], strlen(Quotes[index]) + 1, 0,
    &raddr, rlen);
}
++index;
if (Quotes[index] == NULL) {
    index = 0;
}
}
```

## Chapter 4 Point-to-Point Drivers

### 4.1 Before You Begin

This chapter describes, how to set up and use the following point-to-point drivers:

- PPP Driver
- [PPP over Ethernet Driver](#)

For information about	See
Data types mentioned in this chapter	<a href="#">Chapter 8 "Data Types"</a>
MQX	<i>MQX User's Guide</i> <i>MQX Reference</i>
Protocols	<a href="#">Appendix A "Protocols and Policies"</a>
Prototypes for functions mentioned in this chapter	<a href="#">Chapter 7, "Function Reference"</a>
Setting up RTCS	<a href="#">Chapter 2 "Setting Up the RTCS"</a>
Using RTCS and sockets	<a href="#">Chapter 3 "Using Sockets"</a>

### 4.2 PPP and PPP Driver

PPP Driver conforms to RFC 1661, which is a standard protocol for transporting multi-protocol datagrams over point-to-point links. As such, PPP Driver supplies:

- A method to encapsulate multi-protocol datagrams.
- HDLC-like framing for asynchronous serial devices.
- Link Control Protocol (LCP) to establish, configure, and test the data-link connection.
- One network-control protocol (IPCP) to establish and configure IP.

#### 4.2.1 LCP Configuration Options

The following table lists the LCP configuration options that PPP Driver negotiates. It lists the default values that RFC 1661 specifies and PPP Driver uses. The table also indicates, for which option an application can change the default value. A description of each option follows the table.

Configuration option		Default	See also
ACCM	Asynchronous Control Character Map	0xFFFFFFFF	<a href="#">Section 4.2.2, “Configuring PPP Driver”</a>
ACFC	Address- and Control-Field Compression	FALSE	—
AP	Authentication Protocol (You cannot change the default value of the AP option itself, but you can change the default values of global variables that define the authentication protocol.)	(none)	<a href="#">Section 4.2.2, “Configuring PPP Driver”</a>
MRU	Maximum Receive Unit	1500	—
PFC	Protocol-Field Compression	FALSE	—

#### 4.2.1.1 ACCM

ACCM is a 32-bit mask, where each bit corresponds to a character from 0x00 to 0x1F. The least-significant bit corresponds to 0x00; the most significant to 0x1F. For each bit that is set to one, PPP Driver escapes the corresponding character every time it sends the character over the link.

Because not all processors number bits in the same way, we define bit zero to be the least-significant bit.

The driver sends escaped characters as two bytes in the following order:

- HDLC escaped character (0x7D).
- Escaped character with bit five toggled.

For example, if bit zero of the ACCM is one, every 0x00 byte to be sent over the link is sent as the two bytes 0x7D and 0x20.

PPP Driver always insists on the ACCM as a minimal ACCM for both sides of the link.

An application can change the default value for ACCM. For example, if XON/XOFF flow control is used over the link, an application should set ACCM to 0x000A0000, which escapes XON (0x11) and XOFF (0x13), whenever they occur in a frame.

#### 4.2.1.2 ACFC

By default, ACFC is FALSE, so PPP Driver does not compress the *Address* field and *Control* field in PPP frames. If ACFC becomes TRUE, the driver omits the fields and assumes that they are always 0xFF (for *Address* field) and 0x03 (for *Control* field). To avoid ambiguity when *Protocol* field compression is enabled (that is, when the PFC configuration option is TRUE) and the first *Data* field octet is 0x03, RFC 1661 (PPP) prohibits the use of 0x00FF as the value of the *Protocol* field (which is the protocol number).

PPP Driver always tries to negotiate ACFC.

### 4.2.1.3 AP

On some links, a peer must authenticate itself before it can exchange network-layer packets. PPP Driver supports these authentication protocols:

- PAP
- CHAP

For more information about authentication, and how to change the default values of the global variables that determine the authentication protocol, see [Section 4.2.2, “Configuring PPP Driver.”](#)

### 4.2.1.4 MRU

By default, PPP Driver does not negotiate the MRU, but is prepared to advertise any MRU that is up to 1500 bytes. Additionally, in accordance with RFC 791 (IP), PPP Driver accepts from the peer any MRU that is no fewer than 68 bytes.

### 4.2.1.5 PFC

By default, PFC is FALSE, so PPP Driver does not compress the *Protocol* field. If PFC becomes TRUE, the driver sends the *Protocol* field as a single byte, whenever its value (the protocol number) does not exceed 0x00FF. That is, the most significant byte is not sent if it is zero.

PPP Driver always tries to negotiate PFC.

## 4.2.2 Configuring PPP Driver

PPP Driver uses some global variables, whose default values are assigned according to RFC 1661.

An application can change the configuration of PPP Driver by assigning its own values to the global variables before it initializes PPP Driver for any link; that is, before the first time that it calls [PPP\\_initialize\(\)](#).

To change:	From this default:	Change this global variable:
Additional stack size needed for PPP Driver.	0	_PPPTASK_stacksize
Authentication info for CHAP.	"" NULL NULL	_PPP_CHAP_LNAME _PPP_CHAP_LSECRETS _PPP_CHAP_RSECRETS
Authentication info for PAP.	NULL NULL	_PPP_PAP_LSECRET _PPP_PAP_RSECRETS
Initial timeout (in milliseconds) for PPP Driver's restart timer, when the timer becomes active. The driver doubles the timeout every time the timer expires, until the timeout reaches _PPP_MAX_XMIT_TIMEOUT.	3000	_PPP_MIN_XMIT_TIMEOUT
Maximum timeout (in milliseconds) for PPP Driver's restart timer.	10000	_PPP_MAX_XMIT_TIMEOUT
Minimal ACCM that LCP accepts for both link directions, when PPP Driver configures a link (for information about ACCM, see <a href="#">Section 4.2.1.1, "ACCM"</a> ).	0xFFFF FFFF	_PPP_ACCM
Number of times, while it negotiates link configuration that LCP sends configure-request packets before abandoning.	10	_PPP_MAX_CONF_RETRIES
Number of times, while PPP Driver is closing a link, and before it enters the Closed or Stopped state that it sends terminate-request packets, without receiving a corresponding terminate-ACK packet.	2	_PPP_MAX_TERM_RETRIES
Number of times, while PPP Driver is negotiating link configuration that it sends consecutive configure-NAK packets, before it assumes that the negotiation is not converging, at which time it starts to send configure-reject packets instead.	5	_PPP_MAX_CONF_NAKS
Priority of PPP Driver tasks. (Since you must assign priorities to all the tasks that you write, RTCS lets you change the priority of PPP Driver tasks so that it fits with your design.)	6	_PPPTASK_priority



## 4.2.3 Changing Authentication

By default, PPP Driver does not use an authentication protocol, although it does support the following:

- PAP
- CHAP

Each protocol uses ID-password pairs (PPP\_SECRET structure). For details of the structure, see the listing for [PPP\\_SECRET](#) in [Chapter 8, “Data Types.”](#)

### 4.2.3.1 PAP

PPP Driver, either as the client or the server, controls PAP with two global variables:

- `_PPP_PAP_LSECRET`

One of:

- NULL (LCP does not let the peer request the PAP protocol).
- Pointer to the ID-password pair (PPP\_SECRET) to use, when we authenticate ourselves to the peer.

- `_PPP_PAP_RSECRETS`

One of:

- NULL (LCP does not require that the peer authenticates itself).
- Pointer to a NULL-terminated array of all the ID-password pairs (PPP\_SECRET) to use, when authenticating the peer. LCP requires that the peer authenticates itself. If the peer rejects negotiation of the PAP authentication protocol, LCP terminates the link immediately, when the link reaches the opened state.

### 4.2.3.2 CHAP

PPP Driver controls CHAP with the following global variables:

- `_PPP_CHAP_LNAME`
- Pointer to a NULL-terminated string. On the server side, it is the server's name. On the client side, it is the client's name.
- `_PPP_CHAP_LSECRETS`

One of:

- NULL (LCP does not let the peer request the CHAP protocol).
- Pointer to a NULL-terminated array of ID-password pairs (PPP\_SECRET) to use, when we authenticate ourselves to the peer.
- `_PPP_CHAP_RSECRETS`

One of:

- NULL (LCP does not require that the peer authenticates itself).
- Pointer to a NULL-terminated array of all the ID-password pairs (PPP\_SECRET) to use, when authenticating the peer. LCP requires that the peer authenticates itself. If the peer rejects

negotiation of the CHAP authentication protocol, LCP terminates the link immediately, when the link reaches the opened state.

### 4.2.3.3 Example: Setting Up PAP and CHAP Authentication

#### 4.2.3.4 PAP — Client Side

The user *arc* has the password *password1*.

On the client side for PAP authentication, initialize the global variables as follows.

```
char myname[]          = "arc";
char mysecret[]        = "password1";
PPP_SECRET PAP_secret = {sizeof(myname)-1,
                          sizeof(myscret)-1,
                          myname,
                          mysecret};
_PPP_PAP_LSECRET       = &PAP_secret;
```

#### 4.2.3.5 CHAP — Client Side

CHAP is more flexible in that it lets you have a different password on each host that you might want to connect to. User *arc* has two accounts, using:

- Password *password1* on host *server1*.
- Password *password2* on host *server2*.

On the client side, initialize the global variables as follows:

```
char myname[]          = "arc";
char server1[]         = "server1";
char mysecret1[]       = "password1";
char server2[]         = "server2";
char mysecret2[]       = "password2";
PPP_SECRET CHAP_secrets[] = {{sizeof(server1)-1,
                               sizeof(myscret1)-1,
                               server1, mysecret1},
                              {sizeof(server2)-1,
                               sizeof(myscret2)-1,
                               server2,
                               mysecret2},
                              {0, 0, NULL, NULL}};
_PPP_CHAP_LNAME        = myname;
_PPP_CHAP_LSECRETS     = CHAP_secrets;
```

In this example, RTCS is running on host *server*, and there are three users.

User	Password
arc1	password1
arc2	password2
arc3	password3

#### 4.2.3.6 PAP — Server Side

On the server side for PAP authentication, initialize the global variables as follows:

```
char user1[]      = "arc1";
char secret1[]    = "password1";
char user2[]      = "arc2";
char secret2[]    = "password2";
char user3[]      = "arc3";
char secret3[]    = "password3";
PPP_SECRET secrets[] = {{sizeof(user1)-1,
                          sizeof(secret1)-1,
                          user1,
                          secret1},
                        {sizeof(user2)-1,
                          sizeof(secret2)-1,
                          user2,
                          secret2},
                        {sizeof(user3)-1,
                          sizeof(secret3)-1,
                          user3,
                          secret3},
                        {0, 0, NULL, NULL}
                       };
_PPP_PAP_RSECRETS = secrets;
```

#### 4.2.3.7 CHAP — Server Side

On the server side for CHAP authentication, initialize the global variables as follows:

```
char myname[]     = "server";
char user1[]      = "arc1";
char secret1[]    = "password1";
char user2[]      = "arc2";
char secret2[]    = "password2";
char user3[]      = "arc3";
char secret3[]    = "password3";
PPP_SECRET secrets[] = {{sizeof(user1)-1,
                          sizeof(secret1)-1,
                          user1,
                          secret1},
                        {sizeof(user2)-1,
                          sizeof(secret2)-1,
                          user2,
                          secret2},
                        {sizeof(user3)-1,
                          sizeof(secret3)-1,
                          user3,
                          secret3},
                        {0, 0, NULL, NULL}
                       };
```

```

        user3,
        secret3},
        {0, 0, NULL, NULL}
    };
_PPP_CHAP_LNAME      = myname;
_PPP_CHAP_RSECRETS   = secrets;

```

## 4.2.4 Initializing PPP Links

Before an application can use a PPP link, it must initialize the link by calling **PPP\_initialize()**. The function does the following for the link:

- It allocates and initializes internal data structures and a PPP handle, which it returns.
- It installs PPP callback functions that service the link.
- It initializes LCP and CCP.
- It creates send and receive tasks to service the link.
- It puts the link into the Initial state.

### 4.2.4.1 Using Multiple PPP Links

An application can use multiple PPP links by calling **PPP\_initialize()** for each link.

## 4.2.5 Getting PPP Statistics

To get statistics about PPP links, call **IPIF\_stats()**.

**Table 4-1. Summary: Using PPP Driver**

<b>PPP_initialize()</b>	Initializes PPP Driver (LCP or CCP) for a PPP link.
<b>PPP_SECRET</b>	Authentication passwords.
<b>IPIF_stats()</b>	Gets statistics about PPP links.

## 4.2.6 Example: Using PPP Driver

See [Section 2.15.6, “Example: Setting Up RTCS.”](#)

PPP server and PPP client functionality is demonstrated in the RTCS shell example application, see [.../rtcs/examples/shell](#).

## 4.3 PPP over Ethernet Driver

PPP over Ethernet Driver conforms to RFC 2516, which is a standard protocol for building PPP sessions and encapsulating PPP packets over the ethernet.

**Note**

PPPoE is not supported by this MQX release and will be added in the future MQX versions.

## 4.3.1 Setting Up PPP over Ethernet Driver

### 4.3.1.1 On the Client Side

On the client side, take these general steps to set up and use PPP over Ethernet (PPPoE) Client.

- Initialize an ethernet driver by calling **ENET\_initialize()**, which returns an ethernet handle.
- In a **PPPOE\_CLIENT\_INIT\_DATA\_STRUCT**, initialize the *EHANDLE* field with the ethernet handle.
- Initialize PPPoE Client by calling **\_iopcb\_pppoe\_client\_init()** with the **PPPOE\_CLIENT\_INIT\_DATA\_STRUCT** to get an I/O PCB handle.
- Initialize PPP Driver by calling **PPP\_initialize()** with the I/O PCB handle to get a PPP handle.
- Continue as for PPP Driver.

### 4.3.1.2 On the Server Side

On the server side, take these general steps to set up and use PPPoE Server:

- Initialize an ethernet driver by calling **ENET\_initialize()**, which returns an ethernet handle.
- Initialize **PPPOE\_SERVER\_INIT\_DATA\_STRUCT** and provide callback functions to be referenced through the *SESSION\_UP*, *SESSION\_DOWN*, and *AC\_NAME* fields (see [Section 4.3.2, “Examples: Using PPP over Ethernet Driver”](#)).
- Initialize PPPoE Server by calling **\_pppoe\_server\_init()** with the **PPPOE\_SERVER\_INIT\_DATA\_STRUCT** to get a PPPoE Server handle.
- Call **\_pppoe\_server\_if\_add()** with the ethernet handle and PPPoE Server handle to register the ethernet interface with PPPoE Server, and open discovery and session protocols for the ethernet port.
- Continue as for PPP Driver.

**Table 4-2. Summary: Using PPP over Ethernet Driver**

<b><a href="#">_iopcb_pppoe_client_destroy()</a></b>	Destroys the PPPoE Client task and frees the allocated resources.
<b><a href="#">_iopcb_pppoe_client_init()</a></b>	Initializes PPPoE Client.
<b><a href="#">_pppoe_client_stats()</a></b>	Gets a pointer to the statistics for the PPPoE Client.
<b><a href="#">_pppoe_server_destroy()</a></b>	Destroys the PPPoE Server task and frees the allocated resources.
<b><a href="#">_pppoe_server_if_add()</a></b>	Adds an ethernet interface to the PPPoE Server.
<b><a href="#">_pppoe_server_if_remove()</a></b>	Removes the ethernet interface to the PPPoE Server.
<b><a href="#">_pppoe_server_if_stats()</a></b>	Gets a pointer to statistics on the ethernet interface.
<b><a href="#">_pppoe_server_init()</a></b>	Initializes PPPoE Server.
<b><a href="#">_pppoe_server_session_stats()</a></b>	Gets a pointer to statistics on the PPP session.

## 4.3.2 Examples: Using PPP over Ethernet Driver

### 4.3.2.1 Example: Initializing the Ethernet Device and PPPoE Server

```

void Main_task (uint_32);
void init_ppp_session(pointer,pointer,pointer);
void remove_ppp_session(pointer,pointer,pointer);

TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
/* Task number,   Entry point,   Stack, Pri, String,   Auto? */
{1,              Main_task,      2000,  9,   "Main",   MQX_AUTO_START_TASK},
{0,              0,              0,     0,   0,       0}
};

typedef struct {
    _ppp_handle      PPP_HANDLE;
    _iopcb_handle     IO_PCB_HANDLE;
    uint_32           LOCAL_ADDRESS;
    uint_32           REMOTE_ADDRESS;
    _rtcs_if_handle   IF_HANDLE;
} SERVER_APP_CFG_STRUCT, _PTR_ SERVER_APP_CFG_STRUCT_PTR;

/*
** Initialize global variables
*/
_enet_address enet_local = ENET_ENETADDR;
SERVER_APP_CFG_STRUCT GLOBAL_APP_CFG[MAX_CONNECTION];
_rtcs_msgqueue APP_MSGQ;

static void PPP_session_up_down (pointer msg) {RTCS_msgqueue_trysend (&APP_MSGQ,msg);} /*
Endbody */

/*TASK*-----
*
* Function Name   : Main_task
* Returned Value  : void
* Comments       :
*
*END-----*/
void Main_task
(
    uint_32 temp
)
{ /* Body */
    _rtcs_if_handle  ihandle;
    char_ptr         taskname;
    uint_32           error,i,address,time;
    _enet_handle      ehandle;
    PPPOE_SERVER_INIT_DATA_STRUCT_PTR init_ptr;
    _pppoe_srv_handle srv_handle;
    uint_16           pingid = 0;
    uint_32 PingTargetAddr;
    SERVER_APP_CFG_STRUCT_PTR app_info;

    taskname = "RTCS";
    error = RTCS_create();

```

```

if (error) {
    printf("\n%s failed to initialize, error = %X", taskname,
        error);
    _task_block();
} /* Endif */
/* Enable IP forwarding */
_IP_forward = TRUE;

/* Initialize the Ethernet device */
error = ENET_initialize(ENET_DEVICE, enet_local, 0, &ehandle);
if (error) {
    printf("\nENET initialize: %s", ENET_strerror(error));
    _task_block();
}

address = REMOTE_ADDRESS_BASE;
for (i=0;i<MAX_CONNECTION;i++) {
    GLOBAL_APP_CFG[i].PPP_HANDLE = NULL;
    GLOBAL_APP_CFG[i].IO_PCB_HANDLE = NULL;
    GLOBAL_APP_CFG[i].LOCAL_ADDRESS = SERVER_ADDRESS;
    GLOBAL_APP_CFG[i].REMOTE_ADDRESS = address + i;
    GLOBAL_APP_CFG[i].IF_HANDLE = NULL;
} /* Endfor */

/* initialize the init structure */
init_ptr =
    _mem_alloc_zero(sizeof(PPPOE_SERVER_INIT_DATA_STRUCT));
init_ptr->SESSION_UP = init_ppp_session;
init_ptr->SESSION_DOWN = remove_ppp_session;
init_ptr->AC_NAME = AC_NAME_STRING;
init_ptr->PARAM = NULL;
/* use default values for other values */

RTCS_msgqueue_create(&APP_MSGQ);

_pppoe_server_init(&srv_handle,init_ptr);
_pppoe_server_if_add (srv_handle,ehandle);
_PPP_ACCM = 0;
printf("\nPPPoE server ready\n");

while (TRUE) {
    app_info = RTCS_msgqueue_receive (&APP_MSGQ,0);
    if (app_info->PPP_HANDLE) {
        printf("\nConnection established: REMOTE_IP = %lx\n",
            app_info->REMOTE_ADDRESS);
        /* Initialization of IP specific application could start
           here */
    } else {
        printf("\nConnection closed: REMOTE_IP = %lx\n",
            app_info->REMOTE_ADDRESS);
    } /* Endif */
} /* Endwhile */
} /* Endbody */

void init_ppp_session(pointer pio, pointer phandle,pointer parm)
{ /* Body */
    uint_32 error,i;

```

```

    IPCP_DATA_STRUCT  ipcp_data;
    _rtcs_if_handle    ihandle;
    _iopcb_handle      iopcb = (_iopcb_handle)pio;
    boolean            max_connect = TRUE;

    _PPP_ACCM = 0;
    _iopcb_open(iopcb, PPP_lowerup, PPP_lowerdown, phandle);
    error = RTCS_if_add(phandle, RTCS_IF_PPP, &ihandle);
    if (error) {
        printf("\nIF add failed, error = %lx", error);
    } /* Endif */

    /*
    ** search for an IP address to give out
    */
    for (i=0;i<MAX_CONNECTION;i++) {
        if (GLOBAL_APP_CFG[i].PPP_HANDLE ==NULL) {
            max_connect = FALSE;
            break;
        } /* Endif */
    } /* Endfor */

    if (max_connect) {
        /* (or modify the function so that it returns FALSE */
        return ;
    } /* Endif */

    /* save the session information */
    GLOBAL_APP_CFG[i].PPP_HANDLE = phandle;
    GLOBAL_APP_CFG[i].IO_PCB_HANDLE = pio;
    GLOBAL_APP_CFG[i].IF_HANDLE = ihandle;

    _mem_zero(&ipcp_data, sizeof(ipcp_data));
    /* server configuration */
    ipcp_data.IP_UP = PPP_session_up_down;
    ipcp_data.IP_DOWN = PPP_session_up_down;
    ipcp_data.IP_PARAM = &GLOBAL_APP_CFG[i];
    ipcp_data.ACCEPT_LOCAL_ADDR = FALSE;
    ipcp_data.ACCEPT_REMOTE_ADDR = FALSE;
    ipcp_data.LOCAL_ADDR = GLOBAL_APP_CFG[i].LOCAL_ADDRESS;
    ipcp_data.REMOTE_ADDR =
        GLOBAL_APP_CFG[i].REMOTE_ADDRESS;
    ipcp_data.DEFAULT_NETMASK = TRUE;
    ipcp_data.NETMASK = 0xFFFFFFFF;
    ipcp_data.DEFAULT_ROUTE = FALSE;
    error = RTCS_if_bind_IPCP(ihandle, &ipcp_data);
    if (error) {
        printf("\nIF bind failed, error = %lx", error);
    } /* Endif */

} /* Endbody */

void remove_ppp_session(pointer pio, pointer phandle, pointer parm)
{ /* Body */
    uint_32 i;
    /* find the session we are removing */
    for (i=0;i<MAX_CONNECTION;i++) {

```



```

        if (GLOBAL_APP_CFG[i].PPP_HANDLE== phandle) {
            break;
        } /* Endif */
    } /* Endfor */
    GLOBAL_APP_CFG[i].PPP_HANDLE = NULL;
    GLOBAL_APP_CFG[i].IO_PCB_HANDLE = NULL;
    GLOBAL_APP_CFG[i].IF_HANDLE = NULL;

} /* Endbody */

```

### 4.3.2.2 Example: Initializing the Ethernet Device and PPPoE Client

```

_enet_address enet_local = ENET_ENETADDR;
static void PPP_linkup (pointer lwsem) {_lwsem_post(lwsem);} /* Endbody */

TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
/* Task number,   Entry point,   Stack, Pri, String,   Auto? */
{1,              Main_task,      2000,  9,   "Main",   MQX_AUTO_START_TASK},
{0,              0,              0,      0,   0,        0}
};

/*TASK*-----
*
* Function Name   : Main_task
* Returned Value  : void
* Comments       :
*
*END-----*/

void Main_task
(
    uint_32 temp
)
{ /* Body */
    _rtcs_if_handle  ihandle;
    char_ptr         taskname;
    uint_32          error,time,i;
    _enet_handle     ehandle;
    _iopcb_handle     pio;
    _ppp_handle       phandle;
    IPCP_DATA_STRUCT ipcp_data;
    PPPOE_CLIENT_INIT_DATA_STRUCT_PTR init_ptr;
    LWSEM_STRUCT      ppp_sem;
    uint_16           pingid = 0;
    uint_32 PingTargetAddr;

    taskname = "RTCS";
    error = RTCS_create();
    if (error) {
        printf("\n%s failed to initialize, error = %X", taskname,
            error);
    }
}

```

```

    _task_block();
} /* Endif */
/* Enable IP forwarding */
_IP_forward = TRUE;

/* Initialize the Ethernet device */
error = ENET_initialize(ENET_DEVICE, enet_local, 0, &ehandle);
if (error) {
    printf("\nENET initialize: %s", ENET_strerror(error));
    _task_block();
}

/* initialize the init structure */
init_ptr =
    _mem_alloc_zero(sizeof(PPPOE_CLIENT_INIT_DATA_STRUCT));
init_ptr->EHANDLE = ehandle;

_lwsem_create(&ppp_sem, 0);

/* use the default values for the remaining variables */
pio = _iopcb_pppoe_client_init(init_ptr);
if (pio) {
    printf("\nPPPOE client Initialized.");
} /* Endif */

_PPP_ACCM = 0;
error = PPP_initialize(pio, &phandle);
if (error) {
    printf("\nPPP initialize: %lx", error);
    _task_block();
} /* Endif */
_iopcb_open(pio, PPP_lowerup, PPP_lowerdown, phandle);
error = RTCS_if_add(phandle, RTCS_IF_PPP, &ihandle);
if (error) {
    printf("\nIF add failed, error = %lx", error);
    _task_block();
} /* Endif */

_mem_zero(&ipcp_data, sizeof(ipcp_data));
ipcp_data.IP_UP = PPP_linkup;
ipcp_data.IP_DOWN = NULL;
ipcp_data.IP_PARAM = (pointer)&ppp_sem;
ipcp_data.ACCEPT_LOCAL_ADDR = TRUE;
ipcp_data.LOCAL_ADDR = INADDR_ANY;
ipcp_data.ACCEPT_REMOTE_ADDR = TRUE;
ipcp_data.REMOTE_ADDR = INADDR_ANY;
ipcp_data.DEFAULT_NETMASK = TRUE;
ipcp_data.NETMASK = 0;
ipcp_data.DEFAULT_ROUTE = TRUE;
ipcp_data.NEG_LOCAL_DNS = FALSE;
ipcp_data.ACCEPT_LOCAL_DNS = 0;
ipcp_data.LOCAL_DNS = 0;
ipcp_data.NEG_REMOTE_DNS = FALSE;
ipcp_data.ACCEPT_REMOTE_DNS = 0;
ipcp_data.REMOTE_DNS = 0;

error = RTCS_if_bind_IPCP(ihandle, &ipcp_data);

```

```

if (error) {
    printf("\nIF bind failed, error = %lx", error);
    _task_block();
} /* Endif */
printf("\nTrying to connect...\n");
_lwsem_wait(&ppp_sem); /* block the task until connection */
printf("\nConnection established with the server\n");
printf("\nMy IP_address = %lx", IPCP_get_local_addr(ihandle));
PingTargetAddr = IPCP_get_peer_addr(ihandle);

i = 0;
while (TRUE) {
    time = 5000; /* 5 seconds */
    error = RTCS_ping(PingTargetAddr, &time, ++pingid);
    if (error == RTCSERR_ICMP_ECHO_TIMEOUT) {
        printf("Request timed out\n");
        i++;
        if (i>10) {
            break;
        } /* Endif */
    } else if (error) {
        printf("Error 0x%04X\n", error);
    } else {
        printf("Reply from 0x%X: time=%ldms\n",
            PingTargetAddr, time);
        if ((time<1000)) {
            _time_delay(1000-time);
        } /* Endif */
    } /* Endif */
} /* Endwhile */

_iopcb_close(pio);
printf("\nClient connection closed\n");
_task_block();
} /* Endbody */

```



## Chapter 5 RTCS Applications

### 5.1 Before You Begin

This chapter describes RTCS applications that implement servers and clients for the application-layer protocols that RTCS supports.

For information about	See
Data types mentioned in this chapter	<a href="#">Chapter 8, “Data Types”</a>
MQX	<i>MQX User’s Guide</i> <i>MQX Reference</i>
Protocols	<a href="#">Section Appendix A, “Protocols and Policies”</a>
Prototypes for functions mentioned in this chapter	<a href="#">Chapter 7, “Function Reference”</a>
Setting up the RTCS	<a href="#">Chapter 2, “Setting Up the RTCS”</a>
Using RTCS and sockets	<a href="#">Chapter 3, “Using Sockets”</a>

### 5.2 DHCP Client

The Dynamic Host Configuration Protocol (DHCP) is a binding protocol, as described in RFC 2131. Freescale MQX DHCP Client is based on RFC 2131. The protocol allows a DHCP client to acquire TCP/IP configuration information from a DHCP server, even before having an IP address and mask. DHCP client must be used with RTCS: it cannot be ported to a different internet stack.

By default, the RTCS DHCP client probes the network with an ARP request for the offered IP address, when it receives an offer from a server in response to its discoverer. If a host on the network answers the ARP, the client does not accept the server’s offer; instead it sends a decline to the server’s offer and sends out a new discover. You can disable probing by being sure not to set `DHCP_SEND_PROBE` among the flags defined in *dhcp.h*, when calling **RTCS\_if\_bind\_DHCP\_flagged()**.

**Table 5-1. Summary: Setting Up DHCP Client**

Add the following to the option list that <code>RTCS_if_bind_DHCP()</code> uses:	
<code>DHCP_option_addr()</code>	IP address
<code>DHCP_option_addrlist()</code>	List of IP addresses
<code>DHCP_option_int8()</code>	8-bit value
<code>DHCP_option_int16()</code>	16-bit value
<code>DHCP_option_int32()</code>	32-bit value
<code>DHCP_option_string()</code>	String
<code>DHCP_option_variable()</code>	Variable-length option
<code>RTCS_if_bind_DHCP()</code>	Gets an IP address using DHCP and binds it to the device interface.
<code>DHCPCLNT_find_option()</code>	Searches a DHCP message for a specific option type.

### 5.2.1 Example: Setting Up and Using DHCP Client

See `RTCS_if_bind_DHCP()` in Chapter 7, “Function Reference.”

## 5.3 DHCP Server

DHCP server allocates network addresses and delivers initialization parameters to client hosts that request them. For more information, see RFC 2131. Freescale MQX DHCP Server is based on RFC 2131.

By default, the RTCS DHCP server probes the network for a requested IP address before issuing the address to a client. If the server receives a response, it sends a NAK reply and waits for the client to request a new address. To disable probing, pass the `DHCP_SRV_FLAG_DO_PROBE` flag to `DHCP_SRV_set_config_flag_off()`.

**Table 5-2. Summary: Using DHCP Server**

Add the following to the option list that <code>DHCP_SRV_ippool_add()</code> uses:	
<code>DHCP_option_addr()</code>	IP address
<code>DHCP_option_addrlist()</code>	List of IP addresses
<code>DHCP_option_int8()</code>	8-bit value
<code>DHCP_option_int16()</code>	16-bit value
<code>DHCP_option_int32()</code>	32-bit value
<code>DHCP_option_string()</code>	String
<code>DHCP_option_variable()</code>	Variable-length option
<code>DHCP_SRV_init()</code>	Creates DHCP server.
<code>DHCP_SRV_ippool_add()</code>	Assigns a block of IP addresses to DHCP server.

### 5.3.1 Example: Setting Up and Modifying DHCP Server

See [DHCP\\_SRV\\_init\(\)](#) in [Chapter 7, “Function Reference.”](#)

## 5.4 DNS Resolver

DNS Resolver is an agent that retrieves information, such as a host address or mail information, based on a domain name by querying a DNS server. DNS Resolver implements a client based on the DNS protocol (see RFC 1035).

### 5.4.1 Setting Up DNS Resolver

To setup DNS resolver, modify the following lines in `\source\if\dnshosts.c`:

```
char DNS_Local_network_name[] = ".";
char DNS_Local_server_name[] = "ns.arc.com.";
DNS_SLIST_STRUCT DNS_First_Local_server[] = {{(uchar_PTR_)DNS_Local_server_name, 0,
INADDR_LOOPBACK, 0,0,0,0, DNS_A, DNS_IN }};
```

<b>Note</b>	<i>DNS_SLIST_STRUCT</i> is defined in <code>\source\include\dns.h</code> .
-------------	--

For example, for a local server with the name `DnsServer` on local network `arc.com`, with IP address `10.10.0.120`:

```
char DNS_Local_network_name[] = ".";
char DNS_Local_server_name[] = "DnsServer.arc.com.";
DNS_SLIST_STRUCT DNS_First_Local_server[] =
{{(uchar_PTR_)DNS_Local_server_name, 0, 0x0A0A0078, 0, 0, 0, 0,
DNS_A, DNS_IN }};
```

The following is also valid:

```
char DNS_Local_network_name[] = "arc.com.";
char DNS_Local_server_name[] = "DnsServer";
DNS_SLIST_STRUCT DNS_First_Local_server[] =
{{(uchar_PTR_)DNS_Local_server_name, 0, 0x0A0A0078, 0, 0, 0, 0, DNS_A, DNS_IN }};
```

Calling **DNS\_init()** starts DNS services.

**Table 5-3. Summary: Setting Up DNS Resolver**

<i>DNS_SLIST_STRUCT</i>	DNS server list <i>struct</i> .
<b>DNS_init()</b>	Starts DNS services.

### 5.4.2 Using DNS Resolver

DNS Resolver retrieves information, such as a host address or mail information, based on a domain name. The DNS server to, which DNS Resolver sends its queries, depends on the local server name. To change the default value of the local server name, see [Section 5.4.2.1, “Changing Default Names.”](#)

If a query is successful, the DNS server sends a reply to DNS Resolver. DNS Resolver caches the reply, so that it needs not to make the query again for the lifetime of the resource record, which is defined in the reply. DNS Resolver checks the cache before it makes any query to a DNS server.

### 5.4.2.1 Changing Default Names

If you want DNS Resolver to append a local domain name other than the default, modify the global variable *DNS\_Local\_network\_name*.

If you want to use a DNS server other than the default, modify the global variable *DNS\_Local\_server\_name*.

Name	Defined in source\ifdhshosts.c as global variable	Default value
Local domain	<i>DNS_Local_network_name</i>	". "
Local server	<i>DNS_Local_server_name</i>	""

### 5.4.3 Communicating with a DNS Server

DNS Resolver communicates with a DNS server; the server is not a part of RTCS. The DNS server either provides the answer to a query or a referral to another DNS server.

### 5.4.4 Using DNS Services

RTCS provides functions for obtaining information about servers on the network by address or by name. To get the *HOSTENT\_STRUCT* for an IP address, use function [gethostbyaddr\(\)](#). To get the *HOSTENT\_STRUCT* for a host name, use function [gethostbyname\(\)](#).

**Table 5-4. Summary: Using DNS Services**

<b>gethostbyaddr()</b>	Gets the <i>HOSTENT_STRUCT</i> for an IP address.
<b>gethostbyname()</b>	Gets the <i>HOSTENT_STRUCT</i> for a host name.

## 5.5 Echo Server

Echo Server implements a server that complies with the Echo protocol (RFC 862). The echo service sends any data that it receives back to the originating source .

To start Echo Server, an application calls [ECHOSRV\\_init\(\)](#) with the name of the task that implements the Echo protocol, the task's priority, and its stack size.

<b>Note</b>	When the server is started, the application should make the priority of the task lower than the TCP/IP task; that is, make the task's priority 7, 8, 9, or greater. See information on the <i>_RTCSTASK_priority</i> variable in <a href="#">Section 2.6, "Changing RTCS Creation Parameters"</a> .
-------------	---

Echo Server communicates with a client on the host; the client is not part of RTCS.



## 5.6 EDS Server

EDS Server communicates with a host that is running a performance analysis tool available from Freescale MQX. The tool initiates a connection between the host and target systems, so that TCP/IP packets can be sent over a TCP or UDP connection. EDS Server listens and responds to commands without the need for a debugger. This lets you debug an embedded application from a host computer that is running a performance analysis tool.

When an application starts the EDS Server task through `EDS_init()`, you can establish a connection using the performance analysis tool. Set the configuration settings in the performance analysis tool to match the characteristics of the link. EDS Server assumes a default port number of 5002. You can change this value by changing the following line in *source/apps/eds.c*:

```
#define EDS_PORT          5002
```

## 5.7 FTP Client

To initiate an FTP session, the application calls `FTPd_init()`. Once the FTP session has started, the client issues commands to the FTP server using functions `FTP_command()` and `FTP_command_data()`. The client calls `FTP_close()` to close the FTP session.

## 5.8 FTP Server

The File Transfer Protocol (FTP) is used to transfer files from a remote computer according to RFC 959. The server consists of a protocol interpreter and a data transfer process.

To start FTP Server, an application calls `FTPSRV_init()` with the name of the task that implements FTP, the task's priority, and its stack size.

### Note

When the server is started, the application should make the priority of the task lower than the TCP/IP task; that is, make the task's priority 7, 8, 9, or greater. See information on the `_RTCSTASK_priority` variable in [Section 2.6, "Changing RTCS Creation Parameters"](#).

### 5.8.1 Communicating with an FTP Client

FTP Server waits for an FTP client to connect to it. As defined by RFC 959, FTP Server accepts the following commands from clients:

- **abor** — aborts the previous command and any related transfer of data.
- **acct** — enters account information.
- **help** — displays information about a command.
- **pass** — enters a password.
- **port** — specifies a port number for a data connection.
- **quit** — ends the FTP session.
- **retr** — retrieves a file from the server.
- **stor** — sends a file to the server.
- **user** — enters a user name.

## 5.9 HTTP Server

Hypertext Transfer Protocol (HTTP) server is a simple web server that handles, evaluates, and responses to HTTP requests. Depending on the configuration and incoming client requests, it returns static file system content (web pages, style sheets, images ...) or content dynamically generated by callback routines.

### 5.9.1 Compile Time Configuration

HTTPDCFG\_POLL\_MODE - configures HTTP server for "polling mode". The user needs to poll the server periodically from a single task.

HTTPDCFG\_STATIC\_TASKS - configures the HTTP server for "static-tasks mode". The server creates sessions servicing tasks in advance during an initialization phase. The tasks are not finished after session is closed are recycled for next sessions.

HTTPDCFG\_DYNAMIC\_TASKS - configures the HTTP server for "dynamic-tasks mode". The server creates new task for each new session and terminates the task when session is closed. This method does dynamic memory allocation in runtime.

### 5.9.2 Basic Usage

An easy way to start the HTTP Server with default parameters is to call [httpd\\_server\\_init\(\)](#) for server initialization followed by [httpd\\_server\\_run\(\)](#) to create HTTP server task (one or more tasks — depending on settings).

```
server = httpd_server_init((HTTPD_ROOT_DIR_STRUCT*)root_dir, "\\index.html");
httpd_server_run(server);
```

There is also an option to run the server in poll mode (HTTPDCFG\_POLL\_MODE = 1), without creating a dedicated task. After the server is initialized with [httpd\\_server\\_init\(\)](#), an application should call [httpd\\_server\\_poll\(\)](#) periodically in the background.

```
server = httpd_server_init((HTTPD_ROOT_DIR_STRUCT*)root_dir, "\\index.html");

while (1)
{
    httpd_server_poll(server, 1);
}
```

### 5.9.3 Providing Static Content

One of the key parameters to the HTTP server initialization is an array describing HTTP root directories. Each directory in this array is a mapping between virtual-web directory, root directory, and path to physical filesystem directory.

The following example shows mapping of two web root directories:

- The web root directory (for example `your.server.com/`) to the root of the `tf:` filesystem.
- The usb subdirectory (for example `your.server.com/usb`) to the root of `c:` filesystem.

```
const HTTPD_ROOT_DIR_STRUCT root_dir[] = {  
    { "", "tfs:" },  
    { "usb", "c:" },  
    { 0, 0 }  
};
```

### 5.9.4 Dynamic Content — CGI-Like Pages

An application may register so-called CGI (Common Gateway Interface) callback functions with the HTTP server. The function is called back from the HTTP server when the client requests the assigned CGI file to be retrieved (for example `your.server.com/cginame.cgi`).

The following declaration shows an example of CGI assingment map.

```
const HTTPD_CGI_LINK_STRUCT cgi_lnk_tbl[] = {
    { "ipstat", cgi_ipstat},
    { "icmpstat",          cgi_icmpstat},
    { "udpstat",           cgi_udpstat},
    { "tcpstat",           cgi_tcpstat},
    { "rtcdata",           cgi_rtc_data},
    { 0, 0 }
};
```

The CGI callback functions and their assigned pages are registered by calling

```
HTTPD_SET_PARAM_CGI_TBL(server, (HTTPD_CGI_LINK_STRUCT*)cgi_lnk_tbl);
```

any time after the HTTP server is initialized and before it is run.

### 5.9.5 Dynamic Content — ASP-Like Page Callbacks

Special ASP (Active Server Pages) tags (<% x %>) may be embedded in the HTML files to provide a customized content without the need for generating the full server response (as in the case of CGI handler). The client application may register a callback function, which is called anytime such an ASP tag is processed. The function is then able to generate customized content back to the client.

The following example shows a function, which generates a visibility value based on the USB stick status:

```
static void usb_status_fn(HTTPD_SESSION_STRUCT *session)
{
    if (USB_Stick.VALUE)
        httpd_sendstr(session->sock, "visible");
    else
        httpd_sendstr(session->sock, "hidden");
}

const HTTPD_FN_LINK_STRUCT fn_lnk_tbl[] = {
    { "usb_status_fn",  usb_status_fn },
    { 0, 0 }
};
```

The callback array is registered in a similar way the CGI pages are registered by calling:

```
HTTPD_SET_PARAM_FN_TBL(server, (HTTPD_FN_LINK_STRUCT*)fn_lnk_tbl);
```

anytime after the HTTP server is initialized and before it is run.

The HTML page may contain a special tag in the style string:

```
....
....
<li style="visibility:<% usb_status_fn %>">
<a href="usb/index.htm">Browse USB Mass Storage Device</a>
</li>
....
....
```

## 5.10 IPCFG — High-Level Network Interface Management

IPCFG is a set of high level functions wrapping some of the RTCS network interface management functions described in [Section 2.11, “Binding IP Addresses to Device Interfaces”](#). The IPCFG system may be used to monitor the Ethernet link status and call the appropriate “bind” functions automatically.

In the current version, the IPCFG supports automatic binding of static IP address or automated renewal of DHCP-assigned addresses. It may operate in its own independently running task or in a polling mode.

The IPCFG API functions are all prefixed with **ipcfg\_** prefix. See the functions reference chapter for more details.

The usage procedure of IPCFG is as follows:

1. Create RTCS as described in previous sections ([RTCS\\_create\(\)](#))
2. Initialize network device using [ipcfg\\_init\\_device\(\)](#).
3. Use one of the **ipcfg\_bind\_**xxx functions to bind the interface to an IP address, mask and gateway.
4. You can start the link status monitoring task ([ipcfg\\_task\\_create\(\)](#)) to automatically rebind in case of Ethernet cable is re-attached. Another method to handle this monitoring is to call [ipcfg\\_task\\_poll\(\)](#) periodically in an existing task.
5. You can acquire bind information using various **iocfg\_get\_**xxx functions.

The whole IPCFG functionality is demonstrated in the *ipconfig* command in shell. See its implementation in the *shell/source/rtcs/sh\_ipconfig.c* source code file.

Part of IPCFG functionality depends on what RTCS features are enabled or disabled in the *user\_config.h* configuration file. Any time this configuration is changed, the RTCS library and all applications must be rebuilt.

IPCFG functionality is affected by following defines:

- **RTCSCFG\_ENABLE\_GATEWAYS** - must be set non-zero to enable reaching devices behind gateways within the network. Without this feature, IPCFG ignores all gateway-related data.
- **RTCSCFG\_IPCFG\_ENABLE\_DNS** - must be set non-zero to enable DNS name resolving in IPCFG. Note that DNS functionality also depends on **RTCSCFG\_ENABLE\_DNS**, **RTCSCFG\_ENABLE\_UDP** and **RTCSCFG\_ENABLE\_LWDNS**.
- **RTCSCFG\_IPCFG\_ENABLE\_DHCP** - must be set non-zero to enable DHCP binding in IPCFG. Note that DHCP also depends on **RTCSCFG\_ENABLE\_UDP**.
- **RTCSCFG\_IPCFG\_ENABLE\_BOOT** - must be set non-zero to enable TFTP names processing and BOOT binding

## 5.11 IWCFG — High-Level Wireless Network Interface Management

IWCFG is a set of high level functions wrapping some of wireless configuration management functions. It is used to set the parameters of the network interface which are specific to the wireless operation (for example ESSID). Iwconfig may also be used to display those parameters.

All these parameters are device dependent. Each driver will provide only some of them depending on hardware support, and the range of values may change. Please refer to the documentation main page of each device for details.

The IWCFG API functions are all prefixed with **iwcfg\_** prefix. See the functions reference chapter for more details.

The usage procedure of IWCFG is as follows:

1. Create RTCS as described in previous sections (**RTCS\_create()**)
2. Initialize network device using **ipcfg\_init\_device()**.
3. Initialize wifi device using followed commands:  
**iwcfg\_set\_essid()**  
**iwcfg\_set\_passphrase()**  
**iwcfg\_set\_wep\_key()**  
**iwcfg\_set\_sec\_type()**  
**iwcfg\_set\_mode()**
4. Use one of the **ipcfg\_bind\_**xxx functions to bind the interface to an IP address, mask and gateway.

## 5.12 SNMP Agent

The Simple Network Management Protocol (SNMP) is used to manage TCP/IP-based internet objects. Objects such as hosts, gateways, and terminal servers that have an SNMP agent can perform network-management functions in response to requests from network-management stations.

The Freescale MQX SNMPv1 Agent conforms to the following RFCs:

- RFC 1155
- RFC 1157
- RFC 1212
- RFC 1213

The Freescale MQX SNMPv2c Agent is based on the following RFCs:

- RFC 1905
- RFC 1906

## 5.12.1 Configuring SNMP Agent

SNMP Agent uses several constants defined in *snmpcfg.h*. Those values may be overridden in *user\_config.h*.

	Constant	Default value
Community strings that SNMPv1 and SNMPv2c use.	<b>SNMPCFG_COMMUNITY_LIST</b>	"public"
Size of the static buffer for receiving responses and the static buffer for generating responses (RFCs 1157 and 1906 require it to be at least 484 bytes).	<b>SNMPCFG_BUFFER_SIZE</b>	512
Value of the variable <i>system.sysDescr</i> .	<b>SNMPCFG_SYSDESCR</b>	"RTCS version 3.0"
Value of the variable <i>system.sysServices</i> .	<b>SNMPCFG_SYSSERVICES</b>	8

## 5.12.2 Starting SNMP Agent

To start the SNMP Agent (server), an application calls:

- **MIB1213\_init()**, which installs the standard MIBs that are defined in RFC 1213. This function (or any other MIB initialization functions) must be called before **SNMP\_init()**.
- **SNMP\_init()** with the name of the task that implements the agent, the task's priority and its stack size initializes and runs the agent. Alternatively the **SNMP\_init\_with\_traps()** function may be called with the same arguments plus a pointer to list of trap recipients.

### Note

When the service is started, the application should make the priority of the task lower than the TCP/IP task; that is, make the task's priority 7, 8, 9, or greater. See information on the *\_RTCSTASK\_priority* variable in [Section 2.6, "Changing RTCS Creation Parameters"](#).

## 5.12.3 Communicating with SNMP Clients

SNMP Agent communicates with a client on the host network-management station; the client is not a part of RTCS.

## 5.12.4 Defining Management Information Base (MIB)

The MIB database objects (nodes) are described with a special-syntax definition ("def") file. The definition file is processed by the *mib2c* script, which generates set of initialized *RTCSMIB\_NODE* structures and a bit of infrastructure code. The structures contain pointers to parent, child, and sibling nodes so they effectively implement the MIB tree database in memory. Each node structure also points to a "value" structure (*RTCSMIB\_VALUE*), which contains the actual MIB node data (or function pointer in case of run-time-generated values).

As the MIB tree typically does not need to be changed in run-time, the node structures may be declared "const" and put into read-only memory (this is how the script actually generates them).

The definition file is split into two sections separated by a `%%` separator placed on a single line:

- *Object-definition section* — contains definition of the MIB objects, one object per line.

- *Verbatim C code section* — the second part of the file is copied verbatim to the output file.

#### 5.12.4.1 MIB Definition File: Object Definition

Each MIB object is defined on a single line of this format:

```
objectname parent.number [type access status [index index index ...]]
```

Only the first two parameters (*objectname* and *parent.number* are required). Other parameters are optional, depending on kind of the MIB object being defined. All parameters can be described as follows:

- *objectname* [required] — the object name. It should be a valid C identifier as this name appears in structure and function names in the generated code.
- *parent* [required] — the name of the parent object.
- *number* [required] — child index within the parent object.
- *type* [required for leaf nodes] — the standard ASN.1 encoded type. One of:
  - INTEGER
  - OCTET (for OCTET STRING)
  - OBJECT (for OBJECT IDENTIFIER)
  - SEQUENCE (for SEQUENCE and SEQUENCE OF)
  - IpAddress
  - Counter
  - Gauge
  - TimeTicks
  - Opaque
- *access* [required for leaf nodes] — object accessibility
  - read-only
  - read-write
  - write-only
  - not-accessible
- *status* [required for leaf nodes] — this field is ignored, but should be present for leaf-node definition.
- *index* [required for table row objects] — row identifier (object name); one for each of the table-row indices. Each such index must be subsequently defined as a variable object with the table entry as its parent.



## Examples

- Object definition for the system subtree (object that is a non-leaf node). Defines object `system` as the child number one of node `mib-2`:

```
system mib-2.1
```

- Object definition for the `sysDescr` variable in the system subtree. `sysDescr` is child number one of node `system`. It is a variable of type OCTET STRING, it is read-only, and its implementation is mandatory (this information is not used).

```
sysDescr system.1 OCTET read-only mandatory
```

- Object definition for the `udpEntry` table entry. The line defines the format of a `udpEntry` entry in the `udpTable` table. The entry is indexed by variables `udpAddr` and `udpPort`. The object definition for `udpAddr` and one for `udpPort` should refer the `udpEntry` as their parent.

```
udpEntry udpTable.1 SEQUENCE not-accessible mandatory udpAddr udpPort
udpAddr udpEntry.1 IpAddress read-only mandatory
udpPort udpEntry.2 INTEGER read-only mandatory
```

## Special Lines

- Comment lines.* Lines that begin with `--` and have text on the same line are treated as comments by the code-generation script:

```
-- This is a comment
```

- Type-definition lines.* Line that begins with `%%` defines type based on an existing one:

```
%% new_type existing_type
```

- Separator line.* A line that consists only of two percent signs `%%` and separates the object-definition section from the verbatim C-code section. The code-generator script copies all lines following the separator line to the output C source file.

### 5.12.4.2 MIB Definition File: Verbatim C Code

The C code, generated by the script, references other variables and functions that must be provided by user. Such a user code may be placed anywhere in the application, but it may be a good idea to keep it in the same file with the MIB-definition lines.

The following table summarizes what user code is needed for different kinds of MIB objects:

MIB Object	User C Code Required
Root object in the definition file (the one without parent defined in the same definition file)	A call to <code>RTCSMIB_mib_add(&amp;MIBNODE_&lt;objectname&gt;)</code> registers the object with the SNMP agent.
No-leaf object node.	No user code required. The generated <code>RTCSMIB_NODE</code> structure only contains pointers to other node structures.
Leaf object node (variable object).	The instance of <code>RTCSMIB_VALUE</code> structure named as <code>MIBVALUE_&lt;objectname&gt;</code> .
Table object	A function to map table indices to instances. The function name should be <code>MIB_find_&lt;objectname&gt;()</code> ,
Writable variable object	A function to perform a set operation. The function name should be <code>MIB_set_&lt;objectname&gt;()</code> ,

## Variable Objects

In the verbatim code section, the user should provide implementation of `RTCSMIB_VALUE` structures for all (readable) variable “leaf” objects. The structure is defined as

```
typedef struct rtcsmib_value
{
    uint_32 TYPE;
    pointer PARAM;
} RTCSMIB_VALUE, _PTR_ RTCSMIB_VALUE_PTR ;
```

In this structure, the user specifies the type and method used to retrieve the object value in the application. There are actually two types of information attached to each MIB object:

- One is based directly on the MIB standard type and is attached to the `RTCSMIB_NODE` structure.
- The `TYPE` information attached to `RTCSMIB_VALUE` structure. This type value is used in conjunction with `PARAM` member. See the table below for more details.

MIB Object type	TYPE	PARAM type casting	Description
INTEGER, whose value SNMP agent computes when SNMP manager performs GET	RTCSMIB_NODETYPE_INT_CONST	int_32	Constant signed integer supplied directly as the PARAM value.
	RTCSMIB_NODETYPE_INT_PTR	int_32 *	Pointer to signed integer value.
	RTCSMIB_NODETYPE_INT_FN	RTCSMIB_INT_FN_PTR function pointer: int_32 function(pointer)	Pointer to function that takes an instance pointer (void *), returning the signed int_32 value.
	RTCSMIB_NODETYPE_UINT_CONST	uint_32	Constant unsigned integer supplied directly as the PARAM value.
	RTCSMIB_NODETYPE_UINT_PTR	uint_32 *	Pointer to unsigned integer value.
	RTCSMIB_NODETYPE_UINT_FN	RTCSMIB_UINT_FN_PTR function pointer uint_32 function(pointer)	Pointer to function that takes an instance pointer (void *), returning the unsigned uint_32 value.
NULL-terminated OCTET STRING, whose value SNMP agent computes when SNMP manager performs GET	RTCSMIB_NODETYPE_DISPSTR_FN	uchar_ptr	PARAM points to C string directly.
	RTCSMIB_NODETYPE_DISPSTR_FN	RTCSMIB_UINT_FN_PTR function pointer uchar_ptr function(pointer)	Pointer to function that takes an instance pointer (void *), returning the C string pointer.
OCTET STRING, whose value SNMP agent computes when SNMP manager performs GET	RTCSMIB_NODETYPE_OCTSTR_FN	RTCSMIB_OCTSTR_FN_PTR function pointer uchar_ptr function(pointer, uint_32_PTR_);	Pointer to function that takes an instance pointer (void *), returning address of a static buffer that contains value and length of variable object (must be static, because SNMP does not free it).
OBJECT ID	RTCSMIB_NODETYPE_OID_PTR	RTCSMIB_NODE_PTR	Pointer to Address of an initialized RTCSMIB_NODE variable.
	RTCSMIB_NODETYPE_OID_FN	RTCSMIB_OID_FN_PTR function pointer RTCSMIB_NODE_PTR function(pointer)	Pointer to function that takes an instance pointer (void *), returning address of an initialized RTCSMIB_NODE structure.

## Table-Row Objects

For each variable object that is in a table, you must provide `MIB_find_objectname()` function, where *objectname* is the name of the variable object. See the *1213.c* file in the *rtcs/source/snmp* for the example.

```
boolean MIB_find_objectname
(
    uint_32 op, /* IN */
    pointer index, /* IN */
    pointer _PTR_ instance /* OUT */
)
```

## Writable Objects

For each variable object that is writable, you must provide `MIB_set_objectname()` function, where *objectname* is the name of the variable object. See the *1213.c* file in the *rtcs/source/snmp* for the example.

```
uint_32 MIB_set_objectname
(
    pointer instance, /* IN */
    uchar_ptr value_ptr, /* OUT */
    uint_32 value_len /* OUT */
)
```

- *instance* — NULL (if *objectname* is not in a table) or is a pointer returned by `MIB_find_objectname()`
- *value\_ptr* — Pointer to the value, to which the object is to be set.
- *value\_len* — Length of the value in bytes.

In case the *objectname* is an INTEGER (ASN.1 encoded), you can simplify the parsing by using the built-in function:

```
RTCSMIB_int_read(value_ptr, value_len);
```

The `MIB_set_objectname()` function should return one of the following codes:

- `SNMP_ERROR_noError` — The operation is successful.
- `SNMP_ERROR_wrongValue` — Value cannot be assigned, because it is illegal.
- `SNMP_ERROR_inconsistentValue` — Value is legal, but it cannot be assigned (other reason).
- `SNMP_ERROR_wrongLength` — *value\_len* is incorrect for this object type.
- `SNMP_ERROR_resourceUnavailable` — There are not enough resources.
- `SNMP_ERROR_genErr` — Any other reason.

## 5.12.5 Processing the MIB File

There are several helper AWK scripts accompanying the RTCS installation:

- *def2c.awk* should be used to generate the output C file. This file should be added to project and compiled by standard C compiler together with RTCS library or end the application.

Use this script as:

```
gawk -f def2c.awk mymib.def > mymib.c
```

- *def2mib.awk* may be used to compile the definition file to a standard MIB syntax acceptable by majority of SNMP browsers.

Use this script as:

```
gawk -f def2mib.awk mymib.def > mymib.mib
```

- *mib2def.awk* may be used in early development stages when a standard MIB description file is available. This script generates the first part of the definition file (no user code is generated).

Use this script as:

```
gawk -f mib2def.awk test.mib > test.def
```

## 5.12.6 Standard MIB Included In RTCS

There are two MIBs included and compiled by default with RTCS library.

- The standard MIB, as defined by RFC1213.
- MIB, providing MQX-specific information.

Custom MIB database can be defined as a part of application (see example application in *rtcs/examples/snmp*).

## 5.13 SNTP Client

RTCS provides an SNTP Client that is based on RFC 2030 (Simple Network Time Protocol).

The SNTP Client offers two different interfaces. One is used as a function call that sets the time to the current time, and the other interface starts a SNTP Client task that updates the local time at regular intervals.

**Table 5-5. Summary: SNTP Client Services**

<b>SNTP_init()</b>	Starts the SNTP Client task.
<b>SNTP_oneshot()</b>	Sets the time using the SNTP protocol.

## 5.14 Telnet Client

Telnet Client implements a client that complies with the Telnet protocol specification, RFC 854. A Telnet connection establishes a network virtual terminal configuration between two computers with dissimilar character sets. The *server* host provides a service to the *user* host that initiated the communication.

To start a TCP/IP-based Telnet Client, an application calls **TELNET\_connect()**.

## 5.15 Telnet Server

Telnet Server implements a server that complies with the Telnet protocol specification, RFC 854.

To start Telnet Server, an application calls **TELNETSRV\_init()** with the name of the task that implements the server, the task's priority, its stack size, and a pointer to the task that the server starts, when a client initiates a connection.

<b>Note</b>	When the server is started, the application should make the priority of the task lower than the TCP/IP task; (that is, make the task's priority 7, 8, 9, or greater). See information on the <code>_RTCSTASK_priority</code> variable in <a href="#">Section 2.6, "Changing RTCS Creation Parameters"</a> .
-------------	---

Telnet Server listens on a stream socket. When the Telnet Client initiates a connection, the server creates a new task and redirects the new task's I/O to the socket.

## 5.16 TFTP Client

TFTP Client implements a client that complies with the TFTP (see RFC 1350).

TFTP Client sends a request message to port 69.

## 5.17 TFTP Server

TFTP Server implements a server that complies with the Trivial File Transfer Protocol, TFTP (see RFC 1350). TFTP enables files to be moved between computers on different UDP networks.

### 5.17.1 Configuring TFTP Server

By default, the maximum number of TFTP transactions (**TFTPSRV\_MAX\_TRANSACTIONS**) is 20 (defined in *tftp.h*). If you change the default value, you must recompile TFTP Server.

RTCS provides **TFTPSRV\_access()**, which allows all read accesses and denies all write accesses. You can change its behavior to suit your needs.

### 5.17.2 Starting TFTP Server

To start TFTP Server, an application calls `TFTPSRV_init()` with the name of the task that implements TFTP, the task's priority, and its stack size. We recommend a stack size of at least 1000 bytes. Increase it only if you increase the value of `TFTPSRV_MAX_TRANSACTIONS`.

**Note**

When the server is started, the application should make the priority of the task lower than the TCP/IP task; that is, make the task's priority 7, 8, 9, or greater. See information on the `_RTCSTASK_priority` variable in [Section 2.6, "Changing RTCS Creation Parameters"](#).

## 5.18 Quote of the Day Service

**Note**

Quote of the Day client and server examples are not part of this MQX release.

RTCS provides example code that implements a Quote of the Day client and server. This service is not part of the RTCS library, but you might find it useful as a template to write your own service.

The examples can be found in the following subdirectories of the `\examples` folder:

- client example — `\gotdclnt\gotdclnt.c`
- server example — `\gotdsrv\gotdsrvr.c`

The Quote of the Day server example implements a server that complies with the Quote of the Day protocol (RFC 865). The server task, `QUOTE_server`, listens to TCP connections or UDP datagrams on port 17. When a client request is received, the server sends a quote back. Sample quotations are provided in `\gotdsrv\quotes.c`.

The client task, `QUOTE_client`, connects to the server, gets the quote, displays it, and then closes the connection.

## 5.19 Typical RTCS IP Packet Paths

[Figure 5-1](#) is a diagram of typical code paths for IP packet handling in RTCS applications. This is a generic illustration only, for general purposes, such as finding good locations for setting a breakpoint. The functions listed are internal to RTCS. The driver's input and output interfaces are specific to the media-interface driver software, such as an ethernet driver.

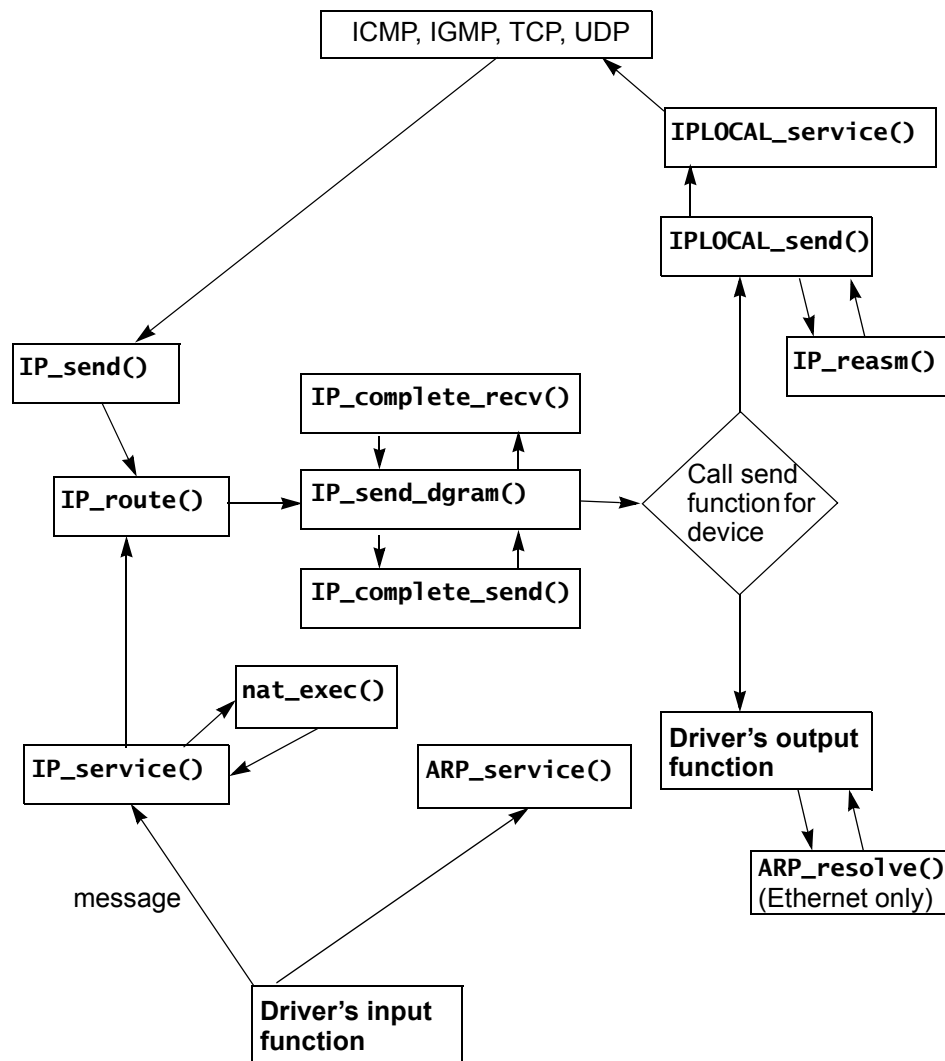


Figure 5-1. Typical RTCS Packet-Processing Paths



## Chapter 6 Rebuilding

### 6.1 Why to Rebuild RTCS

You need to rebuild RTCS, if you do any of the following:

- Change compiler options (for example optimization level).
- Change RTCS compile-time configuration options.
- Incorporate changes that you made to RTCS source code.

<b>CAUTION</b>	We do not recommend you to modify RTCS data structures. If you do, some of the components in the Precise Solution™ Host Tools family of host software-development tools might not perform correctly. Modify RTCS data structures only if you are very experienced with RTCS.
----------------	---

### 6.2 Before You Begin

Before you rebuild RTCS, we recommend that you:

- Read the *MQX User Guide* document for MQX RTOS rebuild instructions. A very similar concept applies also to the RTCS.
- Read the MQX Release Notes that accompany Freescale MQX to get information that is specific to your target environment and hardware.
- Have the required tools for your target environment:
  - compiler
  - assembler
  - linker
- Be familiar with the RTCS directory structure and re-build instructions, as they are described in the release notes document, and also the instructions provided in the following sections.

## 6.3 RTCS Directory Structure

The following table shows the RTCS directory structure.

<i>config</i>			The main configuration directory.
	<board>		Board-specific directory, which contains the main configuration file ( <i>user_config.h</i> ).
<i>rtcs</i>			Root directory for RTCS within the Freescale MQX distribution.
	<i>\build</i>		
		<i>\codewarrior</i>	CodeWarrior-specific build files (project files).
	<i>\examples</i>		
		<i>\example</i>	Source files (.c) for the example and the example's build project.
	<i>\source</i>		All RTCS source code files.
<i>\lib</i>			
	<i>\&lt;board&gt;.&lt;comp&gt;\rtcs</i>		RTCS library files built for your hardware and environment.

## 6.4 RTCS Build Projects in Freescale MQX

The RTCS build project is constructed very much like the other core library projects included in Freescale MQX RTOS. The build project for a given development environment (for example CodeWarrior) is located in the *rtcs\build\<compiler>* directory. Although the RTCS code is not specific to any particular board nor to processor derivative, a separate RTCS build project exists for each supported board. Also the resulting library file is built into a board-specific output directory in *lib\<board>.<compiler>*.

The main reason why is this board-independent code built into the board-specific output directory, is because it may be configured for each board separately. The compile-time user-configuration file is taken from board-specific directory *config\<board>*. In other words, the user may want to build the resulting library code differently for two different boards.

See the *MQX User Guide* for more details about user configuration files or about how to create customized configurations and build projects.

### 6.4.1 Post-Build Processing

All RTCS build projects are configured to generate the resulting binary library file in the top-level *lib\<board>.<compiler>\rtcs* directory. For example the CodeWarrior libraries for the M52259EVB board are built into the *lib\m52259evb.cw\rtcs* directory.

The RTCS build project is also set up to execute post-build batch file, which copies all the public header files to the destination directory. This makes the output *\lib* directory the only place accessed by

the application code. The projects of MQX applications, which need to use the RTCS services, do not need to make any reference to the RTCS source tree at all.

### 6.4.2 Build Targets

CodeWarrior development environment enables to have multiple build configurations, so-called build targets. All projects in the Freescale MQX RTCS contain at least two build targets:

- Debug Target — compiler optimizations are set low to enable easy debugging. Libraries built using this target are named with “\_d” postfix (for example *lib\m52259evb.cw\rtcs\rtcs\_d.a*).
- Release Target — compiler optimizations are set to maximum to achieve the smallest code size and fast execution. The resulting code is very hard to debug. Generated library name does not get any postfix (for example *lib\m52259evb.cw\rtcs\rtcs.a*).

## 6.5 Rebuilding Freescale MQX RTCS

Rebuilding the MQX RTCS library is a simple task, which involves only opening the proper build project in the development environment and building it. Don't forget to select the proper build target to be built or build all targets.

For specific information about rebuilding MQX RTCS and the example applications, see the release notes that accompany the Freescale MQX distribution.



# Chapter 7 Function Reference

## 7.1 Function Listing Format

This is the general format of an entry for a function, compiler intrinsic, or macro.

### 7.1.1 **function\_name()**

A short description of what function **function\_name()** does.

#### Synopsis

Provides a prototype for function **function\_name()**.

```
<return_type> function_name(  
    <type_1> parameter_1,  
    <type_2> parameter_2,  
    ...  
    <type_n> parameter_n)
```

#### Parameters

*parameter\_1 [in]* — Pointer to x  
*parameter\_2 [out]* — Handle for y  
*parameter\_n [in/out]* — Pointer to z

Parameter passing is categorized as follows:

- *In* means the function uses one or more values in the parameter you give it, without storing any changes.
- *Out*
- *Out* means the function saves one or more values in the parameter you give it. You can examine the saved values to find out useful information about your application.
- *In/out*
- *In/out* means the function changes one or more values in the parameter you give it, and saves the result. You can examine the saved values to find out useful information about your application.

#### Description

Describes the function **function\_name()**. This section also describes any special characteristics or restrictions that might apply:

- Function blocks, or might block under certain conditions.
- Function must be started as a task.
- Function creates a task.

- Function has pre-conditions that might not be obvious.
- Function has restrictions or special behavior.

### Return Value

Specifies any value or values returned by function **function\_name()**.

### See Also

Lists other functions or data types related to function **function\_name()**.

### Example

Provides an example (or a reference to an example) that illustrates the use of function **function\_name()**.

### Function Listings

This section provides function listings in alphabetical order.

## 7.1.2 `_iopcb_open()`

Opens the I/O PCB driver for PPP.

### Synopsis

```
void _iopcb_open(
    _iopcb_handle  ioppp,
    _CODE_PTR_    PPP_lowerup(),
    _CODE_PTR_    PPP_lowerdown(),
    _ppp_handle    PPP_handle)
```

### Parameters

*ioppp* [in] — I/O PCB handle.

*PPP\_lowerup()* [in] — Pointer to callback function to use, when the lower layer is up.

*PPP\_lowerdown()* [in] — Pointer to the callback function to use, when the lower layer is down.

*PPP\_handle* [in] — Pointer to the PPP interface handle from **PPP\_initialize()**

### Description

Function **\_iopcb\_open()** opens the I/O PCB driver for PPP using handle *ioppp* (returned by **\_iopcb\_ppphdlc\_init()** or **\_iopcb\_pppoe\_client\_init()**), and saves *PPP\_lowerup()*, *PPP\_lowerdown()*, and *PPP\_handle*:

- When the frame driver is ready to send and receive frames, it calls *PPP\_lowerup()* with *PPP\_handle*.
- When the frame driver can no longer send or receive frames, it calls *PPP\_lowerdown()* with *PPP\_handle*.

Under some circumstances, the frame driver calls *PPP\_lowerup()* multiple times. For example, if it needs to dial a modem, the frame driver calls *PPP\_lowerup()* every time a connection is established, and *PPP\_lowerdown()* every time carrier is lost.

### Return Value

None

### See Also

- [\\_iopcb\\_ppphdlc\\_init\(\)](#)
- [\\_iopcb\\_pppoe\\_client\\_init\(\)](#)
- [PPP\\_initialize\(\)](#)

### Example

See [Section 2.15.6, “Example: Setting Up RTCS.”](#)

### 7.1.3 `_iopcb_ppphdlc_init()`

Initializes the driver for the HDLC-like framing device, and gets a handle to the device.

#### Synopsis

```
_iopcb_handle _iopcb_ppphdlc_init(  
    FILE_PTR    device)
```

#### Parameters

*device [in]* — Asynchronous serial device handle

#### Description

Function `_iopcb_ppphdlc_init()` uses the asynchronous serial device handle returned by `fopen()` to initialize the driver for the HDLC-like framing device, and returns a handle to the device.

#### Return Value

- I/O PCB handle (success)
- Error (failure)

#### See Also

- [\\_iopcb\\_open\(\)](#)

#### Example

See [Section 2.15.6, “Example: Setting Up RTCS.”](#)



## 7.1.4 `_iopcb_pppoe_client_destroy()`

Destroys the PPPoE Client task.

### Synopsis

```
void _iopcb_pppoe_client_destroy(  
    pointer      ppp_handle,  
    _iopcb_handle iopcb)
```

### Parameters

*ppp\_handle* [in] — PPP handle

*iopcb* [in] — I/O PCB handle for the session

### Description

Function `_iopcb_pppoe_client_destroy()` destroys the PPPoE Client task, and frees the resources that are allocated to the PPPoE Client task.

### Return Value

None.

### See Also

- [\\_iopcb\\_pppoe\\_client\\_init\(\)](#)

## 7.1.5 `_iopcb_pppoe_client_init()`

Initializes PPPoE Client.

### Synopsis

```
_iopcb_handle _iopcb_pppoe_client_init(
    PPPOE_CLIENT_INIT_DATA_STRUCT_PTR init)
```

### Parameters

*init [in]* — pointer to *PPPOE\_CLIENT\_INIT\_DATA\_STRUCT*

### Description

Function `_iopcb_pppoe_client_init()` initializes the driver for the PPP over Ethernet framing device and returns a handle to the device.

### Return Value

- I/O PCB handle (success)
- NULL (failure)

### See Also

- [\\_iopcb\\_open\(\)](#)
- *PPPOE\_CLIENT\_INIT\_DATA\_STRUCT*

### Example

The following example sets up RTCS with a PPP over Ethernet device.

```
_rtcs_if_handle  ihandle;
uint_32          error;

/* For Ethernet driver: */
_enet_handle  ehandle;

/* For PPPOE Driver: */
pio;
_ppp_handle    phandle;
IPCP_DATA_STRUCT  ipcp_data;
LWSEM_STRUCT    ppp_sem;
PPPOE_CLIENT_INIT_DATA_STRUCT_PTR  init_ptr;
static char      MySecretName[64];
static char      MySecretPassword[64];
static PPP_SECRET MySecrets[2];
char_ptr  login_string;
char_ptr  password;

static void      PPP_linkup (pointer lwsem){_lwsem_post(lwsem);}

error = RTCS_create();

if (error) {
    printf("\nFailed to create RTCS, error = %X", error);
    return;
}
```

```

/* Enable IP forwarding: */
_IP_forward = TRUE;

/* Set up the Ethernet driver: */
error = ENET_initialize(ENET_DEVICE, enet_local, 0, &ehandle);
if (error) {
    printf("\nFailed to initialize Ethernet driver: %s",
        ENET_strerror(error));
    return;
}

/*Set up PPPOE Driver: */
init_ptr = _mem_alloc_zero(sizeof(PPPOE_CLIENT_INIT_DATA_STRUCT));
init_ptr->EHANDLE = ehandle;
/* use the default values for rest of the variables */
pio = _iopcb_pppoe_client_init(init_ptr);
error = PPP_initialize(pio, &phandle);
if (error) {
    printf("\nFailed to initialize PPP Driver: %x", error);
    return;
}
_iopcb_open(pio, PPP_lowerup, PPP_lowerdown, phandle);
error = RTCS_if_add(phandle, RTCS_IF_PPP, &ihandle);
if (error) {
    printf("\nFailed to add interface for PPP, error = %x", error);
    return;
}
_lwsem_create(&ppp_sem, 0);

_mem_zero(&ipcp_data, sizeof(ipcp_data));
ipcp_data.IP_UP = PPP_linkup;
ipcp_data.IP_DOWN = NULL;
ipcp_data.IP_PARAM = (pointer)&ppp_sem;
ipcp_data.ACCEPT_LOCAL_ADDR = TRUE;
ipcp_data.LOCAL_ADDR = INADDR_ANY;
ipcp_data.ACCEPT_REMOTE_ADDR = TRUE;
ipcp_data.REMOTE_ADDR = INADDR_ANY;
ipcp_data.DEFAULT_NETMASK = TRUE;
ipcp_data.NETMASK = 0;
ipcp_data.DEFAULT_ROUTE = TRUE;
ipcp_data.NEG_LOCAL_DNS = FALSE;
ipcp_data.ACCEPT_LOCAL_DNS = 0;
ipcp_data.LOCAL_DNS = 0;
ipcp_data.NEG_REMOTE_DNS = FALSE;
ipcp_data.ACCEPT_REMOTE_DNS = 0;
ipcp_data.REMOTE_DNS = 0;

login_string = "<login_name>";
password = "<password>"

strcpy(MySecretName, login_string);
strcpy(MySecretPassword, password);

if (ENABLE_CHAP) {
    MySecrets[1].PPP_ID_LENGTH = MySecrets[1].PPP_PW_LENGTH = 0;
    _PPP_CHAP_LSECRETS = &MySecrets[1];
}

```

```

    _PPP_CHAP_LNAME = MySecretName;
    MySecrets[0].PPP_PW_PTR = MySecretPassword;
    MySecrets[0].PPP_PW_LENGTH = strlen(MySecretPassword);
    _PPP_PAP_LSECRET = NULL;
} else {
    MySecrets[0].PPP_ID_PTR = MySecretName;
    MySecrets[0].PPP_ID_LENGTH = strlen(MySecretName);
    MySecrets[0].PPP_PW_PTR = MySecretPassword;
    MySecrets[0].PPP_PW_LENGTH = strlen(MySecretPassword);
    _PPP_PAP_LSECRET = &MySecrets[0];
    _PPP_CHAP_LSECRETS = NULL;
    _PPP_CHAP_LNAME = NULL;
} /* EndIf */

error = RTCS_if_bind_IPCP(ihandle, &ipcp_data);
if (error) {
    printf("\nFailed to bind interface for PPP, error = %x", error);
    return;
}
_lwsem_wait(&ppp_sem);
printf("Connection established with the server");

```

## 7.1.6 `_pppoe_client_stats()`

Gets a pointer to the statistics for PPP over Ethernet Driver.

### Synopsis

```
PPPOEIF_STATS_STRUCT_PTR _pppoe_client_stats(  
    _iopcb_handle pio)
```

### Parameters

*pio* [in] — I/O PCB handle

### Description

Function `_pppoe_client_stats()` returns a pointer to statistics for PPP over Ethernet Driver. Parameter *pio* is returned by `_iopcb_pppoe_client_init()`.

### Return Value

- Pointer to a `PPPOEIF_STATS_STRUCT` structure (success)
- NULL (failure: *pio* was invalid)

### See Also

- [\\_iopcb\\_pppoe\\_client\\_init\(\)](#)
- [PPPOEIF\\_STATS\\_STRUCT](#)

### 7.1.7 `_pppoe_server_destroy()`

Destroy the PPPoE Server tasks.

#### Synopsis

```
uint_32 _pppoe_server_destroy(  
    _pppoe_srv_handle pppoe_handle)
```

#### Parameters

*pppoe\_handle* [in] — PPPoE Server handle

#### Description

Function `_pppoe_server_destroy()` destroys the PPPoE Server task and frees the resources that are allocated to that task.

#### Return Value

- **PPPOE\_OK** (success)
- Error code (failure)

#### See Also

- [\\_pppoe\\_server\\_init\(\)](#)

## 7.1.8 `_pppoe_server_if_add()`

Adds an ethernet interface to the PPPoE Server.

### Synopsis

```
uint_32 _pppoe_server_if_add(  
    _pppoe_srv_handle pppoe_handle,  
    _enet_handle      enet_handle)
```

### Parameters

*pppoe\_handle [in]* — PPPoE Server handle

*enet\_handle [in]* — Ethernet port handle

### Description

Function `_pppoe_server_if_add()` adds an ethernet interface to the PPPoE Server, and opens discovery and session protocols for the Ethernet port.

### Return Value

- **PPPOE\_OK** (success)
- Error code (failure)

### See Also

- [\\_pppoe\\_server\\_if\\_remove\(\)](#)

## 7.1.9 `_pppoe_server_if_remove()`

Removes the ethernet interface to the PPPoE Server.

### Synopsis

```
uint_32 _pppoe_server_if_remove(  
    _pppoe_srv_handle pppoe_handle,  
    _enet_handle      enet_handle)
```

### Parameters

*pppoe\_handle* [in] — PPPoE Server handle  
*enet\_handle* [in] — Ethernet interface handle

### Description

Function `_pppoe_server_if_remove()` removes the ethernet interface to the PPPoE Server, and closes discovery and session protocols for the ethernet port. The ethernet interface must be previously registered by a call to `_pppoe_server_if_add()`.

### Return Value

- **PPPOE\_OK** (success)
- Error code (failure)

### See Also

- [\\_pppoe\\_server\\_if\\_add\(\)](#)



### 7.1.10 `_pppoe_server_if_stats()`

Gets a pointer to statistics on the ethernet interface.

#### Synopsis

```
uint_32  _pppoe_server_if_stats(  
    _pppoe_srv_handle  pppoe_handle,  
    _enet_handle       enet_handle)
```

#### Parameters

*pppoe\_handle* [in] — PPPoE server handle  
*enet\_handle* [in] — Ethernet interface handle

#### Description

Function `_pppoe_server_if_stats()` returns a pointer to the statistics on the ethernet interface for the PPPoE server.

#### Return Value

- Pointer to a **PPPOEIF\_STATS\_STRUCT** structure (success)
- Error code (failure)

#### See Also

- [\\_pppoe\\_server\\_if\\_add\(\)](#)
- [\\_pppoe\\_server\\_if\\_remove\(\)](#)
- [PPPOEIF\\_STATS\\_STRUCT](#)

## 7.1.11 \_pppoe\_server\_init()

Initializes PPPoE Server.

### Synopsis

```
uint_32 _pppoe_server_init(  
    _pppoe_srv_handle _PTR_      pppoe_handle,  
    PPPOE_SERVER_INIT_DATA_STRUCT_PTR pppoe_init_data)
```

### Parameters

*pppoe\_handle* [out] — PPPoE Server handle  
*pppoe\_init\_data* [in] — Initialization parameters

### Description

Function **\_pppoe\_server\_init()** initializes the PPPoE Server so the PPPoE Server can respond to PPP over Ethernet discovery packets sent via the Ethernet Driver.

### Return Value

- **PPPOE\_OK** (success)
- Error code (failure)

### See Also

- [\*PPPOE\\_SERVER\\_INIT\\_DATA\\_STRUCT\*](#)

## 7.1.12 `_pppoe_server_session_stats()`

Gets a pointer to statistics on the PPP session.

### Synopsis

```
PPPOE_SESSION_STATS_STRUCT_PTR  
_pppoe_server_session_stats(  
    _iopcb_handle  iopcb)
```

### Parameters

*iopcb* [*in*] — I/O PCB handle

### Description

Function `_pppoe_server_session_stats()` provides statistics on the PPP session using the I/O PCB handle *iopcb*.

### Return Value

- Pointer to a `PPPOE_SESSION_STATS_STRUCT` structure (success)
- Error code (failure)

### See Also

- [`PPPOE\_SESSION\_STATS\_STRUCT`](#)

### 7.1.13 accept()

Creates a new stream socket to accept incoming connections from the remote endpoint.

#### Synopsis

```
uint_32 accept(
    uint_32      socket,
    sockaddr_in  _PTR_ peeraddr,
    uint_16      _PTR_  addrlen)
```

#### Parameters

*socket [in]* — Handle for the parent stream socket.

*peeraddr [out]* — Pointer to where to place the remote endpoint identifier.

*addrlen [in/out]* — When passed in Pointer to the length, in bytes, of the location *peeraddr* points to. When passed out: Full size, in bytes, of the remote-endpoint identifier.

#### Description

The function accepts incoming connections by creating a new stream socket for the connections. The parent socket (*socket*) must be in the listening state; it remains in the listening state after each new socket is created from it.

The new socket created by **accept()** inherits the link-layer options from the listening socket. The new socket has the same local endpoint and socket options as the parent; the remote endpoint is the originator of the connection.

This function blocks until an incoming connection is available.

#### Return Value

- Handle for a new stream socket (success)
- *RTCS\_SOCKET\_ERROR* (failure)

#### See Also

- [bind\(\)](#)
- [connect\(\)](#)
- [listen\(\)](#)
- [socket\(\)](#)

#### Example

```
uint_32      handle;
uint_32      child_handle;
sockaddr_in  remote_sin;
uint_16      remote_addrlen;
uint_32      status;

...

status = listen(handle, 0);
if (status != RTCS_OK) {
    printf("\nError, listen() failed with error code %lx", status);
```

```
} else {
    remote_addrlen = sizeof(remote_sin);
    child_handle = accept(handle, &remote_sin, &remote_addrlen);
    if (child_handle != RTCS_SOCKET_ERROR) {
        printf("\nConnection accepted from %lx, port %d",
            remote_sin.sin_addr, remote_sin.sin_port);
    } else {
        status = RTCS_geterror(handle);
        if (status == RTCS_OK) {
            printf("\nConnection reset by peer");
        } else {
            printf("Error, accept() failed with error code %lx",
                status);
        }
    }
}
```

### 7.1.14 ARP\_stats()

Gets a pointer to the ARP statistics that RTCS collects for the interface.

#### Synopsis

```
ARP_STATS_PTR ARP_stats(
    _rtcs_if_handle rtcs_if_handle)
```

#### Parameters

*rtcs\_if\_handle* [in] — RTCS interface handle from **RTCS\_if\_add()**.

#### Return Value

- Pointer to the *ARP\_STATS* structure for *rtcs\_if\_handle* (success).
- NULL (failure: *rtcs\_if\_handle* is invalid).

#### See Also

- [ENET\\_get\\_stats\(\)](#)
- [ICMP\\_stats\(\)](#)
- [IP\\_stats\(\)](#)
- [IPIF\\_stats\(\)](#)
- [RTCS\\_if\\_add\(\)](#)
- [TCP\\_stats\(\)](#)
- [UDP\\_stats\(\)](#)
- [ARP\\_STATS](#)

#### Example

Use RTCS statistics functions to display received-packets statistics.

```
void display_rx_stats(void)
{
    IP_STATS_PTR    ip;
    IGMP_STATS_PTR  igmp;
    IPIF_STATS      ipif;
    ICMP_STATS_PTR  icmp;
    UDP_STATS_PTR   udp;
    TCP_STATS_PTR   tcp;
    ARP_STATS_PTR   arp;
    _rtcs_if_handle ihandle;
    _enet_handle    ehandle;

    ENET_initialize(ENET_DEVICE, enet_local, 0, &ehandle);
    RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);

    ip    = IP_stats();
    igmp  = IGMP_stats();
    ipif  = IPIF_stats(ihandle);
    icmp  = ICMP_stats();
    udp   = UDP_stats();
    tcp   = TCP_stats();
    arp   = ARP_stats(ihandle);
}
```

```
printf("\n%d IP packets received", ip->ST_RX_TOTAL);  
printf("\n%d IGMP packets received", igmp->ST_RX_TOTAL);  
printf("\n%d IPIF packets received", ipif->ST_RX_TOTAL);  
printf("\n%d TCP packets received", tcp->ST_RX_TOTAL);  
printf("\n%d UDP packets received", udp->ST_RX_TOTAL);  
printf("\n%d ICMP packets received", icmp->ST_RX_TOTAL);  
printf("\n%d ARP packets received", arp->ST_RX_TOTAL);  
}
```

## 7.1.15 bind()

Binds the local address to the socket.

### Synopsis

```
uint_32 bind(
    uint_32          socket,
    sockaddr_in *_PTR localaddr,
    uint_16          addrlen)
```

### Parameters

*socket* [in] — Socket handle for the socket to bind.

*localaddr* [in] — Pointer to the local endpoint identifier, to which to bind *socket* (see description).

*addrlen* [in] — Length in bytes of what *localaddr* points to.

### Description

The following *localaddr* input values are required:

sockaddr_in field	Required input value
sin_family	AF_INET
sin_port	One of: <ul style="list-style-type: none"> <li>Local port number for the socket.</li> <li>Zero (to determine the port number that RTCS chooses, call <b>getsockname()</b>).</li> </ul>
sin_addr	One of: <ul style="list-style-type: none"> <li>IP address that was previously bound with a call to one of the <b>RTCS_if_bind</b> functions.</li> <li><b>INADDR_ANY</b>.</li> </ul>

Usually, TCP/IP servers bind to **INADDR\_ANY**, so that one instance of the server can service all IP addresses.

This function blocks, but RTCS immediately services the command, and is replied to by the socket layer.

### Return Value

- **RTCS\_OK** (success)
- Specific error code (failure)

### See Also

- **RTCS\_if\_bind** family of functions
- [socket\(\)](#)
- [sockaddr\\_in](#)



## Example

Binds a socket to port number 2010.

```
uint_32      sock;
sockaddr_in  local_sin;
uint_32      result;
...
sock = socket(AF_INET, SOCK_DGRAM, 0);
if (sock == RTCS_SOCKET_ERROR)
{
    printf("\nError, socket create failed");
    return;
}
memset((char *) &local_sin, 0, sizeof(local_sin));
local_sin.sin_family = AF_INET;
local_sin.sin_port = 2010;
local_sin.sin_addr.s_addr = INADDR_ANY;
result = bind(sock, &local_sin, sizeof (sockaddr_in));
if (status != RTCS_OK)
    printf("\nError, bind() failed with error code %lx", result);
```

## 7.1.16 connect()

Connects the stream socket to the remote endpoint, or sets a remote endpoint for a datagram socket.

### Synopsis

```
uint_32 connect (
    uint_32      socket,
    sockaddr_in  _PTR_ destaddr,
    uint_16      addrlen)
```

### Parameters

- socket* [in] — Handle for the stream socket to connect.
- destaddr* [in] — Pointer to the remote endpoint identifier.
- addrlen* [in] — Length in bytes of what *destaddr* points to.

### Description

The **connect()** function might be used multiple times. Whenever **connect()** is called, the current endpoint is replaced by the new one.

A connection can be dissolved by calling **connect()** and specifying an address family of *AF\_UNSPEC*. This dissolves the association, places the socket into the bound state, and returns the error code *RTCSERR\_SOCK\_INVALID\_AF*.

If **connect()** fails, the socket is left in a bound state (no remote endpoint).

When used with stream sockets, the function fails, if the remote endpoint:

- Rejects the connection request, which it might do immediately.
- Is unreachable, which causes the connection timeout to expire.

If the function is successful, the application can use the socket to transfer data.

When used with datagram sockets, the function has the following effects:

- The **send()** function can be used instead of **sendto()** to send a datagram to *destaddr*.
- The behavior of **sendto()** is unchanged: it can still be used to send a datagram to any peer.
- The socket receives datagrams from *destaddr* only.

This task blocks, until the connection is accepted, or until the connection-timeout socket option expires.

### Return Value

- *RTCS\_OK* (success)
- Specific error code (failure)

### See Also

- [accept\(\)](#)
- [bind\(\)](#)
- [getsockopt\(\)](#)

- [listen\(\)](#)
- [setsockopt\(\)](#)
- [socket\(\)](#)

### Example: Stream Socket

```

uint_32      sock;
uint_32      child_handle;
sockaddr_in  remote_sin;
uint_16      remote_addrlen = sizeof(sockaddr_in);
uint_32      result;
...

/* Connect to 192.203.0.83, port 2011: */
memset((char *) &remote_sin, 0, sizeof(sockaddr_in));
remote_sin.sin_family      = AF_INET;
remote_sin.sin_port        = 2011;
remote_sin.sin_addr.s_addr = 0xC0A80001; /* 192.168.0.1 */

result = connect(sock, &remote_sin, remote_addrlen);

if (result != RTCS_OK)
{
    printf("\nError--connect() failed with error code %lx.",
                                                    result);
} else {
    printf("\nConnected to %lx, port %d.",
          remote_sin.sin_addr.s_addr, remote_sin.sin_port);
}

```

## 7.1.17 DHCP\_find\_option()

Searches a DHCP message for a specific option type.

### Synopsis

```
uchar_ptr DHCP_find_option(
    uchar_ptr  msgptr,
    uint_32    msglen,
    uchar      option)
```

### Parameters

*msgptr* [in/out] — Pointer to the DHCP message.

*msglen* [in/out] — Pointer to the number of bytes in the message.

*option* [in/out] — Option type to search for (see RFC 2131).

### Description

The *msgptr* pointer points to an option in the DHCP message, which is formatted according to RFCs 2131 and 2132. The application is responsible for parsing options and reading the values.

The returned pointer must be passed to one of the *ntohl* or *ntohs* macros to extract the value of the option. The macros can convert the value into host-byte order.

### Return Value

- Pointer to the specified option in the DHCP message in network-byte order (success).
- NULL (no option of the specified type exists).

### See Also

- [DHCPCLNT\\_find\\_option\(\)](#)

### Example

```
/* Get a pointer to the start of the DHCP server's name from a
   packet (like a DH_OFFER packet) recieved from the server */

uchar_PTR_ buffer_ptr; /* This is a DHCP packet recieved
                           from a server */

uint_32 buffer_size;
uchar_PTR_ optptr;

optptr = DHCPCLNT_find_option(buffer_ptr, buffer_size, DHCP_OPT_SERVERNAME);
```

## 7.1.18 DHCP\_option\_addr()

Adds the IP address to the list of DHCP options for DHCP Server.

### Synopsis

```
boolean  DHCP_option_addr(
    uchar_ptr  _PTR_  optptr,
    uint_32    _PTR_  optlen,
    uchar      opttype,
    _ip_address optval)
```

### Parameters

*optptr* [in/out] — Pointer to the option list.

*optlen* [in/out] — Pointer to the number of bytes remaining in the option list:  
in before *optval* is added.

Passed out after *optval* is added.

*opttype* [in] — Option type to add to the list (see RFC 2132).

*optval* [in] — IP address to add.

### Description

Function **DHCP\_option\_addr()** adds IP address *optval* to the list of DHCP options for the DHCP server. The application subsequently passes parameter *optptr* (pointer to the option list) to **DHCPSRV\_ippool\_add()**.

### Return Value

- TRUE (success)
- FALSE (failure: not enough room in the option list)

### See Also

- [DHCPCLNT\\_find\\_option\(\)](#)
- [DHCPSRV\\_ippool\\_add\(\)](#)
- [DHCP\\_option\\_addrlist\(\)](#)
- [DHCP\\_option\\_int8\(\)](#)
- [DHCP\\_option\\_int16\(\)](#)
- [DHCP\\_option\\_int32\(\)](#)
- [DHCP\\_option\\_string\(\)](#)
- [DHCP\\_option\\_variable\(\)](#)

### Example

See [DHCPSRV\\_init\(\)](#).

## 7.1.19 DHCP\_option\_addrlist()

Adds the list of IP addresses to the list of DHCP options for DHCP Server.

### Synopsis

```
boolean  DHCP_option_addrlist(
    uchar_ptr    _PTR_  optptr,
    uint_32      _PTR_  optlen,
    uchar        opttype,
    _ip_address  _PTR_  optval,
    uint_32      listlen)
```

### Parameters

*optptr* [in/out] — Pointer to the option list.

*optlen* [in/out] — Pointer to the number of bytes remaining in the option list:

Passed in before *optval* is added.

Passed out after *optval* is added.

*opttype* [in] — Option type to add to the list (see RFC 2132).

*optval* [in] — Pointer to list of IP addresses.

*listlen* [in] — Number of IP addresses in the list.

### Description

Function **DHCP\_option\_addrlist()** adds the list of IP addresses referenced by *optval* to the list of DHCP options for the DHCP Server. The application subsequently passes parameter *optptr* (pointer to the option list) to **DHCPSRV\_ippool\_add()**.

### Return Value

- TRUE (success)
- FALSE (failure: not enough room in the option list)

### See Also

- [DHCPCLNT\\_find\\_option\(\)](#)
- [DHCPSRV\\_ippool\\_add\(\)](#)
- [DHCP\\_option\\_addr\(\)](#)
- [DHCP\\_option\\_int8\(\)](#)
- [DHCP\\_option\\_int16\(\)](#)
- [DHCP\\_option\\_int32\(\)](#)
- [DHCP\\_option\\_string\(\)](#)
- [DHCP\\_option\\_variable\(\)](#)

### Example

See [DHCPSRV\\_init\(\)](#).

## 7.1.20 DHCP\_option\_int16()

Adds a 16-bit value to the list of DHCP options for DHCP Server.

### Synopsis

```
boolean DHCP_option_int16(
    uchar_ptr _PTR_ optptr,
    uint_32   _PTR_ optlen,
    uchar     opttype,
    uint_16   optval)
```

### Parameters

*optptr* [in/out] — Pointer to the option list.

*optlen* [in/out] — Pointer to the number of bytes remaining in the option list:

Passed in before *optval* is added.

Passed out after *optval* is added.

*opttype* [in] — Option type to add to the list (see RFC 2132).

*optval* [in] — Value to add.

### Description

Function **DHCP\_option\_int16()** adds the 16-bit value *optval* to the list of DHCP options for DHCP Server. The application subsequently passes parameter *optptr* (pointer to the option list) to **DHCPSRV\_ippool\_add()**.

### Return Value

- TRUE (success)
- FALSE (failure: not enough room in the option list)

### See Also

- [DHCPCLNT\\_find\\_option\(\)](#)
- [DHCPSRV\\_ippool\\_add\(\)](#)
- [DHCP\\_option\\_addr\(\)](#)
- [DHCP\\_option\\_addrlist\(\)](#)
- [DHCP\\_option\\_int8\(\)](#)
- [DHCP\\_option\\_int32\(\)](#)
- [DHCP\\_option\\_string\(\)](#)
- [DHCP\\_option\\_variable\(\)](#)

### Example

See [DHCPSRV\\_init\(\)](#).

## 7.1.21 DHCP\_option\_int32()

Adds a 32-bit value to the list of DHCP options for DHCP Server.

### Synopsis

```
boolean DHCP_option_int32(
    uchar_ptr _PTR_ optptr,
    uint_32   _PTR_ optlen,
    uchar     opttype,
    uint_32   optval)
```

### Parameters

*optptr* [in/out] — Pointer to the option list.

*optlen* [in/out] — Pointer to the number of bytes remaining in the option list:

Passed in before *optval* is added.

Passed out after *optval* is added.

*opttype* [in] — Option type to add to the list (see RFC 2132).

*optval* [in] — Value to add.

### Description

Function **DHCP\_option\_int32()** adds a 32-bit value to the list of DHCP options for DHCP Server. The application subsequently passes parameter *optptr* (pointer to the option list) to **DHCPSRV\_ippool\_add()**.

### Return Value

- TRUE (success)
- FALSE (failure: not enough room in the option list)

### See Also

- [DHCPCLNT\\_find\\_option\(\)](#)
- [DHCPSRV\\_ippool\\_add\(\)](#)
- [DHCP\\_option\\_addr\(\)](#)
- [DHCP\\_option\\_addrlist\(\)](#)
- [DHCP\\_option\\_int8\(\)](#)
- [DHCP\\_option\\_int16\(\)](#)
- [DHCP\\_option\\_string\(\)](#)
- [DHCP\\_option\\_variable\(\)](#)

### Example

See [RTCS\\_if\\_bind\\_DHCP\(\)](#) and [DHCPSRV\\_init\(\)](#).



### 7.1.22 DHCP\_option\_int8()

Adds an 8-bit value to the list of DHCP options for DHCP Server.

#### Synopsis

```
boolean  DHCP_option_int8(
    uchar_ptr  _PTR_  optptr,
    uint_32    _PTR_  optlen,
    uchar      opttype,
    uchar      optval)
```

#### Description

Function **DHCP\_option\_int8()** adds an 8-bit value to the list of DHCP options for DHCP Server. The application subsequently passes parameter *optptr* (pointer to the option list) to **DHCPSRV\_ippool\_add()**.

#### Parameters

*optptr* [in/out] — Pointer to the option list.

*optlen* [in/out] — Pointer to the number of bytes remaining in the option list:

Passed in before *optval* is added.

Passed out after *optval* is added.

*opttype* [in] — Option type to add to the list (see RFC 2132).

*optval* [in] — Value to add.

#### Return Value

- TRUE (success)
- FALSE (failure: not enough room in the option list)

#### See Also

- [DHCPCLNT\\_find\\_option\(\)](#)
- [DHCPSRV\\_ippool\\_add\(\)](#)
- [DHCP\\_option\\_addr\(\)](#)
- [DHCP\\_option\\_addrlist\(\)](#)
- [DHCP\\_option\\_int16\(\)](#)
- [DHCP\\_option\\_int32\(\)](#)
- [DHCP\\_option\\_string\(\)](#)
- [DHCP\\_option\\_variable\(\)](#)

#### Example

See [DHCPSRV\\_init\(\)](#).

### 7.1.23 DHCP\_option\_string()

Adds a string to the list of DHCP options for DHCP Server.

## Synopsis

```
uint_32 DHCP_option_string(
    uchar_ptr _PTR_ optptr,
    uint_32 _PTR_ optlen,
    uchar opttype,
    char_ptr optval)
```

## Description

Function **DHCP\_option\_string()** adds a string to the list of DHCP options for the DHCP Server. The application subsequently passes parameter *optptr* (pointer to the option list) to **DHCPSRV\_ippool\_add()**.

## Parameters

*optptr* [in/out] — Pointer to the option list.

*optlen* [in/out] — Pointer to the number of bytes remaining in the option list:

Passed in before *optval* is added.

Passed out after *optval* is added.

*opttype* [in] — Option type to add to the list (see RFC 2132).

*optval* [in] — String to add.

## Return Value

- TRUE (success)
- FALSE (failure: not enough room in the option list)

## See Also

- [DHCPCLNT\\_find\\_option\(\)](#)
- [DHCPSRV\\_ippool\\_add\(\)](#)
- [DHCP\\_option\\_addr\(\)](#)
- [DHCP\\_option\\_addrlist\(\)](#)
- [DHCP\\_option\\_int8\(\)](#)
- [DHCP\\_option\\_int16\(\)](#)
- [DHCP\\_option\\_int32\(\)](#)
- [DHCP\\_option\\_variable\(\)](#)

## Example

See [DHCPSRV\\_init\(\)](#).

## 7.1.24 DHCP\_option\_variable()

Adds a variable-length option to a list of DHCP options for DHCP Server.

### Synopsis

```
uint_32 DHCP_option_variable(
    uchar_ptr _PTR_ optptr,
    uint_32   _PTR_ optlen,
    uchar     opttype,
    uchar     _PTR_ optdata,
    uint_32   datalen)
```

### Parameters

*optptr* [in/out] — Pointer to the option list.

*optlen* [in/out] — Pointer to the number of bytes remaining in the option list:

Passed in before *optval* is added.

Passed out after *optval* is added.

*opttype* [in] — Option type to add to the list (see RFC 2132).

*optdata* [in] — Sequence of bytes to add.

*datalen* [in] — Number of bytes *optdata* points to.

### Description

Function **DHCP\_option\_variable()** adds a variable-length option to a list of DHCP options for DHCP Server. Use this function to create the *optptr* buffer that you pass to **DHCPSRV\_ippool\_add()** and **RTCS\_if\_bind\_DHCP()**.

### Return Value

- TRUE (success)
- FALSE (failure)

### See Also

- [DHCPCLNT\\_find\\_option\(\)](#)
- [DHCPSRV\\_ippool\\_add\(\)](#)
- [DHCP\\_option\\_addr\(\)](#)
- [DHCP\\_option\\_addrlist\(\)](#)
- [DHCP\\_option\\_int8\(\)](#)
- [DHCP\\_option\\_int16\(\)](#)
- [DHCP\\_option\\_int32\(\)](#)
- [DHCP\\_option\\_string\(\)](#)
- [RTCS\\_if\\_bind\\_DHCP\(\)](#)

### Example

See [RTCS\\_if\\_bind\\_DHCP\(\)](#).

## 7.1.25 DHCPCLNT\_find\_option()

Searches a DHCP message for a specific option type.

### Synopsis

```
uchar_ptr DHCPCLNT_find_option(  
    uchar_ptr msgptr,  
    uint_32   msglen,  
    uchar     option)
```

### Parameters

*msgptr* [in/out] — Pointer to the DHCP message.

*msglen* [in/out] — Pointer to the number of bytes in the message.

*option* [in/out] — Option type to search for (see RFC 2131).

### Description

The *msgptr* pointer points to an option in the DHCP message, which is formatted according to RFCs 2131 and 2132. The application is responsible for parsing options and reading the values.

The returned pointer must be passed to one of the *ntohl* or *ntohs* macros to extract the value of the option. The macros can be used to convert the value into host-byte order.

### Return Value

- Pointer to the specified option in the DHCP message in network-byte order (success).
- NULL (no option of the specified type exists).

### See Also

- [DHCP\\_find\\_option\(\)](#)

## 7.1.26 DHCPCLNT\_release()

Releases a DHCP Client no longer needed.

### Synopsis

```
uchar_ptr DHCPCLNT_release(
    _rtcs_if_handle handle)
```

### Parameters

*handle [in]* — Pointer to the interface no longer needed.

### Description

Use function **DHCPCLNT\_release()** to release a DHCP client, when your application no longer needs it.

Function **DHCPCLNT\_release()** does the following:

- It cancels timer events in the DHCP state machine.
- It sets the state to RELEASING (resulting in the release of resources with this state).
- It unbinds from an interface.
- It stops listening on the DHCP port.
- It releases resources.

### Return Value

- *void* (success)
- Error code (failure)

### See Also

- [RTCS\\_if\\_bind\\_DHCP\(\)](#)

### Example

```
_rtcs_if_handle ihandle;
/* start RTCS task, add an interface and bind it with
   RTCS_if_bind_DHCP */
/* do some stuff with the interface */
/* all done */
DHCPCLNT_release(ihandle);
```

## 7.1.27 DHCP\_SRV\_init()

Starts DHCP Server.

### Synopsis

```
uint_32 DHCP_SRV_init(
    char_ptr  name,
    uint_32   priority,
    uint_32   stacksize)
```

### Parameters

*name* [in] — Name of the server's task.  
*priority* [in] — Priority for the server's task.  
*stacksize* [in] — Stack size for the server's task.

### Description

Function **DHCP\_SRV\_init()** starts the DHCP server and creates *DHCP\_SRV\_task*.

### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

### See Also

- [DHCPCLNT\\_find\\_option\(\)](#)
- [DHCP\\_option\\_addr\(\)](#)
- [DHCP\\_option\\_addrlist\(\)](#)
- [DHCP\\_option\\_int8\(\)](#)
- [DHCP\\_option\\_int16\(\)](#)
- [DHCP\\_option\\_int32\(\)](#)
- [DHCP\\_option\\_string\(\)](#)
- [DHCP\\_option\\_variable\(\)](#)

### Example

Start DHCP Server and set up its options:

```
DHCP_SRV_DATA_STRUCT dhcpsrv_data;
uchar                dhcpsrv_options[200];
_ip_address          routers[3];
uchar_ptr            optptr;
uint_32              optlen;
uint_32              error;

/* Start DHCP Server: */
error = DHCP_SRV_init("DHCP server", 7, 2000);
if (error != RTCS_OK) {
    printf("\nFailed to initialize DHCP Server, error %x", error);
    return;
```

```

}
printf("\nDHCP Server running");

/* Fill in the required parameters: */
/* 192.168.0.1: */
dhcpsrv_data.SERVERID = 0xC0A80001;
/* Infinite leases: */
dhcpsrv_data.LEASE = 0xFFFFFFFF;
/* 255.255.255.0: */
dhcpsrv_data.MASK = 0xFFFFFFFF0;
/* TFTP server address: */
dhcpsrv_data.SADDR = 0xC0A80002;
memset(dhcpsrv_data.SNAME, 0, sizeof(dhcpsrv_data.SNAME));
memset(dhcpsrv_data.FILE, 0, sizeof(dhcpsrv_data.FILE));

/* Fill in the options: */
optptr = dhcpsrv_options;
optlen = sizeof(dhcpsrv_options);
/* Default IP TTL: */
DHCP_SRV_option_int8(&optptr, &optlen, 23, 64);
/* MTU: */
DHCP_SRV_option_int16(&optptr, &optlen, 26, 1500);
/* Renewal time: */
DHCP_SRV_option_int32(&optptr, &optlen, 58, 3600);
/* Rebinding time: */
DHCP_SRV_option_int32(&optptr, &optlen, 59, 5400);
/* Domain name: */
DHCP_SRV_option_string(&optptr, &optlen, 15, "arc.com");
/* Broadcast address: */
DHCP_SRV_option_addr(&optptr, &optlen, 28, 0xC0A800FF);
/* Router list: */
routers[0] = 0xC0A80004;
routers[1] = 0xC0A80005;
routers[2] = 0xC0A80006;
DHCP_SRV_option_addrlist(&optptr, &optlen, 3, routers, 3);

/* Serve addresses 192.168.0.129 to 192.168.0.135 inclusive: */
DHCP_SRV_ippool_add(0xC0A80081, 7, &dhcpsrv_data, dhcpsrv_options,
                   optptr - dhcpsrv_options);

```

## 7.1.28 DHCP\_SRV\_ippool\_add()

Gives DHCP Server the block of IP addresses to serve.

### Synopsis

```
uint_32  DHCP_SRV_ippool_add(
    _ip_address      ipstart,
    uint_32          ipnum,
    DHCP_SRV_DATA_STRUCT_PTR params_ptr,
    uchar_ptr        optptr,
    uint_32          optlen)
```

### Parameters

*ipstart* [in] — First IP address to give.

*ipnum* [in] — Number of IP addresses to give.

*params\_ptr* [in] — Pointer to the configuration information that is associated with the IP addresses.

*optptr* [in] — Pointer to the optional configuration information that is associated with the IP addresses.

*optlen* [in] — Number of bytes that *optptr* points to.

### Description

Function **DHCP\_SRV\_ippool\_add()** gives the DHCP server the block of IP addresses it serves. The DHCP Server task must be created (by calling **DHCP\_SRV\_init()**) before you call this function.

### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

### See Also

- [DHCPCLNT\\_find\\_option\(\)](#)
- [DHCP\\_option\\_addr\(\)](#)
- [DHCP\\_option\\_addrlist\(\)](#)
- [DHCP\\_option\\_int8\(\)](#)
- [DHCP\\_option\\_int16\(\)](#)
- [DHCP\\_option\\_int32\(\)](#)
- [DHCP\\_option\\_string\(\)](#)
- [DHCP\\_option\\_variable\(\)](#)
- [DHCP\\_SRV\\_init\(\)](#)
- [DHCP\\_SRV\\_DATA\\_STRUCT](#)

### Example

See [DHCP\\_SRV\\_init\(\)](#).



## 7.1.29 DHCPDRV\_set\_config\_flag\_off()

Disables address probing.

### Synopsis

```
uint_32 DHCPDRV_set_config_flag_off (
    uint_32                                flag)
```

### Parameters

*flag* [in] — DHCP server address-probing flag

### Description

By default, the RTCS DHCP server probes the network for a requested IP address before issuing the address to a client. If the server receives a response, it sends a NAK reply and waits for the client to request a new address. You can disable probing to reduce overhead in time and traffic. To do so, pass the DHCPDRV\_FLAG\_DO\_PROBE flag to **DHCPDRV\_set\_config\_flag\_off()**.

This function may be called any time after **DHCPDRV\_init()**.

### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

### See Also

- [DHCPDRV\\_set\\_config\\_flag\\_on\(\)](#)
- [DHCPDRV\\_init\(\)](#)

### Example

```
#define DHCP_DO_PROBING 1
int dhcp_do_probing = DHCP_DO_PROBING;
/*init*/
/*setup*/
if (dhcp_do_probing) {
    DHCPDRV_set_config_flag_on(DHCPDRV_FLAG_DO_PROBE);
}
else {
    DHCPDRV_set_config_flag_off(DHCPDRV_FLAG_DO_PROBE);
}
```

### 7.1.30 DHCPDRV\_set\_config\_flag\_on()

Re-enables address probing.

#### Synopsis

```
uint_32 DHCPDRV_set_config_flag_on (
    uint_32 flag
```

#### Parameters

flag [in] — DHCP server address-probing flag

#### Description

By default, the RTCS DHCP server probes the network for a requested IP address before issuing the address to a client. If the server receives a response, it sends a NAK reply and waits for the client to request a new address. If you have previously disabled probing, pass the *DHCPDRV\_FLAG\_DO\_PROBE* flag to **DHCPDRV\_set\_config\_flag\_on()** to reenables probing.

#### Return Value

- **RTCS\_OK** (success)
- Error code (failure)

#### See Also

- [DHCPDRV\\_set\\_config\\_flag\\_off\(\)](#)
- [DHCPDRV\\_init\(\)](#)

#### Example

```
#define DHCP_DO_PROBING 1
int dhcp_do_probing = DHCP_DO_PROBING;
/*init*/
/*setup*/
if (dhcp_do_probing) {
    DHCPDRV_set_config_flag_on(DHCPDRV_FLAG_DO_PROBE);
}
else {
    DHCPDRV_set_config_flag_off(DHCPDRV_FLAG_DO_PROBE);
}
```

### 7.1.31 DNS\_init()

Starts a DNS client in order to use DNS services.

#### Synopsis

```
uint_32  DNS_init(void)
```

#### Description

Function **DNS\_init()** starts a DNS client in order to use DNS services, and creates *DNS\_Resolver\_task*.

Before your application calls the function, it should bind an IP address to an interface by calling one of the **RTCS\_if\_bind** family of functions.

#### Return Value

- *RTCS\_OK* (success)
- Error code: The function returns an error if it cannot do any of the following:
  - Allocate memory for DNS control structures.
  - Create a temporary datagram socket.
  - Detach from the temporary socket.
  - Create *DNS\_Resolver\_task*.

#### See Also

- [gethostbyaddr\(\)](#)
- [gethostbyname\(\)](#)
- [RTCS\\_if\\_bind\(\)](#)
- [RTCS\\_if\\_bind\\_BOOTP\(\)](#)
- [RTCS\\_if\\_bind\\_DHCP\(\)](#)
- [RTCS\\_if\\_bind\\_IPCP\(\)](#)

## 7.1.32 ECHOSRV\_init()

Starts RFC 862 Echo Server.

### Synopsis

```
uint_32  ECHOSRV_init(  
    char_ptr  name,  
    uint_32   priority  
    uint_32   stacksize)
```

### Parameters

*name* [in] — Name of the server's task.  
*priority* [in] — Priority of the server's task.  
*stacksize* [in] — Stack size for the server's task.

### Description

Function **ECHOSRV\_init()** starts the RFC 862 Echo Server and creates *ECHO\_task*. We recommend that you make *priority* lower than the *priority* of the RTCS task; that is, make it a higher number.

### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

### Example

```
error = ECHOSRV_init("Echo server", 7, 1000);
```

### 7.1.33 EDS\_init()

Starts Embedded Debug Server (EDS server).

#### Synopsis

```
uint_32  EDS_init(  
    char_ptr      name,  
    uint_32       priority,  
    uint_32       stacksize)
```

#### Parameters

*name* [in] — Name of EDS Server (Winsock) task.

*priority* [in] — Priority of EDS Server (Winsock).

*stacksize* [in] — Stack size for EDS Server (Winsock) task.

#### Description

The function starts the EDS task, which listens on UDP and TCP ports 5002, and Creates *EDS\_task*. When the Integrated Profiler (running on a host computer) establishes a connection with the server, the server allows the Integrated Profiler to communicate with the EDS task.

We recommend that you make *priority* lower than the *priority* of the RTCS task; that is, make it a higher number.

#### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

### 7.1.34 ENET\_get\_stats()

Gets a pointer to the ethernet statistics that RTCS collects for the ethernet interface.

#### Synopsis

```
ENET_STATS_PTR  ENET_get_stats(
    _enet_handle  _PTR_  handle)
```

#### Parameters

handle [in] — Pointer to the Ethernet handle

#### Description

The function is not a part of RTCS. If you are using MQX, the function is available to you and you can use it. If you are porting RTCS to another operating system, the application must supply the function.

#### Return Value

Pointer to the *ENET\_STATS* structure.

#### See Also

- [ICMP\\_stats\(\)](#)
- [IP\\_stats\(\)](#)
- [IPIF\\_stats\(\)](#)
- [RTCS\\_if\\_add\(\)](#)
- [TCP\\_stats\(\)](#)
- [UDP\\_stats\(\)](#)
- [ENET\\_STATS](#)

#### Example

```
ENET_STATS_PTR  enet;
_enet_handle    ehandle;

...
enet = ENET_get_stats();
printf("\n%d Ethernet packets received", enet->ST_RX_TOTAL);
```

### 7.1.35 ENET\_initialize()

Initializes the interface to the ethernet device.

#### Synopsis

```
uint_32  ENET_initialize(
    uint_32 device_num,
    _enet_address address,
    uint_32 flags,
    _enet_handle _PTR_ enet_handle)
```

#### Parameters

*device\_num* [in] — Device number for the device to initialize.

*address* [in] — Ethernet address of the device to initialize.

*flags* [in] — One of the following:

non-zero (use the ethernet address from the device's EEPROM).

Zero (use *address*).

THIS PARAMETER IS NOT USED ANYMORE AND IS IGNORED!

*enet\_handle* [out] — Pointer to the ethernet handle for the device interface.

#### Description

The function is not a part of RTCS. If you are using MQX, the function is available to you and you can use it. If you are porting RTCS to another operating system, the application must supply the function.

<b>Note</b>	This function can be called only once per device number.
-------------	--

The function does the following:

- It initializes the ethernet hardware and makes it ready to send and receive ethernet packets.
- It installs the ethernet interrupt service routine.
- It sets up send and receive buffers, which are usually a representation of the ethernet device's own buffers.
- It allocates and initializes the ethernet handle, which the upper layer uses with other functions from the Ethernet Driver API and from the RTCS API.

#### Return Value

- *ENET\_OK* (success)
- Ethernet error code (failure)

#### Example

See [Section 2.15.6, “Example: Setting Up RTCS.”](#)

### 7.1.36 FTP\_close()

Terminates an FTP session.

#### Synopsis

```
int_32  FTP_close(  
    pointer    handle,  
    FILE_PTR   ctrl_fd)
```

#### Parameters

*handle* [in] — FTP session handle

*ctrl\_fd* [in] — Device to write control-connection responses to

#### Description

Function **FTP\_close()** issues a **QUIT** command to the FTP server, closes the control connection, and then frees any resources that were allocated to the FTP session handle.

#### Return Value

- The FTP response code (success)
- -1 (failure)

#### See Also

- [FTPd\\_init\(\)](#)

#### Example

See [FTPd\\_init\(\)](#).



### 7.1.37 FTP\_command()

Issues a command to the FTP server.

#### Synopsis

```
int_32  FTP_command(  
    pointer    handle,  
    char_ptr   command,  
    FILE_PTR   ctrl_fd)
```

#### Parameters

*handle* [in] — FTP session handle.

*command* [in] — FTP command.

*ctrl\_fd* [in] — Device to write control-connection responses to.

#### Description

Function **FTP\_command()** sends a command to the FTP server.

#### Return Value

- The FTP response code (success)
- -1 (failure)

#### See Also

- [FTP\\_command\\_data\(\)](#)

#### Example

See [FTPD\\_init\(\)](#).

### 7.1.38 FTP\_command\_data()

Issues a command to the FTP server that requires a data connection.

#### Synopsis

```
int_32  FTP_command(
    pointer    handle,
    char_ptr   command,
    FILE_PTR   ctrl_fd,
    FILE_PTR   data_fd,
    uint_32    flags)
```

#### Parameters

*handle* [in] — FTP session handle.

*command* [in] — FTP command.

*ctrl\_fd* [in] — Device to write control-connection responses to.

*data\_fd* [in] — Device for the data connection.

*flags* [in] — Options for the data connection.

#### Description

Function **FTP\_command\_data()** sends a command to the FTP server, opens a data connection, and then performs a data transfer.

Parameter *flags* is a bitwise **OR** of the following:

- the connection mode, which must be one of the following:
  - FTPMODE\_DEFAULT — the client will use the default port for the data connection.
  - FTPMODE\_PORT — the client will choose an unused port and issue a PORT command.
  - FTPMODE\_PASV — the client will issue a PASV command.
- the data-transfer direction, which must be one of:
  - FTPDIR\_RECV — the client will read data from the data connection and write it to *data\_fd*.
  - FTPDIR\_SEND — the client will read data from *data\_fd* and send it to the data connection.

#### Return Value

- The FTP response code (success)
- -1 (failure)

#### See Also

- [FTP\\_command\(\)](#)

#### Example

See [FTPD\\_init\(\)](#).

## 7.1.39 FTPd\_init()

Starts the FTP Server.

### Synopsis

```
uint_32  FTPd_init(
    char_ptr      name,
    uint_32       priority,
    uint_32       stacksize)
```

### Parameters

*name* [in] — Name of FTP Server task.

*priority* [in] — Priority of FTP Server task (we recommend that you make the priority lower than the priority of the RTCS task; that is, make it a higher number).

*stacksize* [in] — Stack size for FTP Server task.

*shell* [in] — Shell task that FTP Server starts, when a client initiates a connection (see description).

### Description

Function **FTPd\_init()** starts Telnet Server and creates *FTPSRV\_task*.

A sample FTP Server is included in the examples/shell directory. The FTP Server allows any number of users to connect from a remote workstation using an FTP client.

The FTP Server optionally supports usernames and passwords. To enable usernames and passwords, the global *FTPd\_userfile* must be set to the name of the file containing the usernames and passwords. The username and password file contains one line for each username password combination, in one of the following formats:

- username
- username:password
- username:password:info

If the username is specified without a password, no password is required. The info field is ignored.

This is a valid sample password file:

```
guest
anonymous
user1:pass1
user2:pass2:
user3:pass3:other stuff\
```

To disable usernames and passwords, set *FTPd\_userfile* to NULL.

The commands supported by the FTP Server are configurable. The application must initialize a NULL terminated global variables *FTPd\_COMMAND\_STRUCT FTPd\_commands[]* with the supported commands and *char FTPd\_rootdir[]* with the default FTP root directory path.

Available commands are:

Function	FTP Command String	Description
<i>FTPd_cd</i>	cwd, xcwd	Changes directory.
<i>FTPd_cdup</i>	cdup	Change to parent directory.
<i>FTPd_dele</i>	dele	Deletes file.
<i>FTPd_help</i>	help	Help — returns list of supported commands.
<i>FTPd_list</i>	list	Lists files.
<i>FTPd_mkdir</i>	mkd, xmkd	Makes directory.
<i>FTPd_nlst</i>	nlst	Lists files.
<i>FTPd_noop</i>	noop	No operation.
<i>FTPd_opts</i>	opts	Sets options — always returns “bad option.”
<i>FTPd_pass</i>	pass	Specifies password.
<i>FTPd_pasv</i>	pasv	Enters passive mode.
<i>FTPd_port</i>	port	Specifies port.
<i>FTPd_pwd</i>	pwd, xpwd	Prints working directory.
<i>FTPd_quit</i>	quit	Quits.
<i>FTPd_retr</i>	retr	Retrieves file.
<i>FTPd_rmdir</i>	rmd, xrm	Removes directory.
<i>FTPd_site</i>	site	Gets site information.
<i>FTPd_size</i>	size	Gets file size.
<i>FTPd_stor</i>	stor	Stores file.
<i>FTPd_syst</i>	syst	System.
<i>FTPd_type</i>	type	Sets type (ascii or binary).
<i>FTPd_unimplemented</i>	abor, acct	Returns unimplemented command.
<i>FTPd_user</i>	user	Specifies user name.
<i>FTPd_feat</i>	feat	request a descriptive list of server-supported features
<i>FTPd_rmd</i>	rmd	remove a remote directory
<i>FTPd_rnfr</i>	rnfr	rename from
<i>FTPd_rnto</i>	rnto	rename to
<i>FTPd_site</i>	site	site-specific commands
<i>FTPd_size</i>	size	return the size of a file

The FTP Server may be started or stopped from the shell, by including the *Shell\_FTPd* function in the shell command list.

*FTPD* *init* initializes a new FTP Server (found in the examples directory), which has the following enhancements:

- It uses revised file system abstraction, allowing better support for MFS and TargetFFS.
- It supports multiple simultaneous FTP sessions.
- It uses a command table, so the user can configure the supported commands.

We recommend the use of **FTPD\_init()** in place of **FTPSRV\_init()**.

### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

### Example

```
#include <mqx.h>
#include <rtcs.h>
#include "ftpd.h"

// ftp root dir
const char FTPd_rootdir[] = {"c:\\\\"};

//ftp commands
const FTPd_COMMAND_STRUCT FTPd_commands[] = {
    { "abor", FTPd_unimplemented },
    { "acct", FTPd_unimplemented },
    { "cdup", FTPd_cdup },
    { "cwd", FTPd_cd },
    { "feat", FTPd_feat },
    { "help", FTPd_help },
    { "dele", FTPd_dele },
    { "list", FTPd_list },
    { "mkd", FTPd_mkdir },
    { "noop", FTPd_noop },
    { "nlst", FTPd_nlst },
    { "opts", FTPd_opts },
    { "pass", FTPd_pass },
    { "pasv", FTPd_pasv },
    { "port", FTPd_port },
    { "pwd", FTPd_pwd },
    { "quit", FTPd_quit },
    { "rnfr", FTPd_rnfr },
    { "rnto", FTPd_rnto },
    { "retr", FTPd_retr },
    { "stor", FTPd_stor },
    { "rmd", FTPd_rmdir },
    { "site", FTPd_site },
    { "size", FTPd_size },
    { "syst", FTPd_syst },
    { "type", FTPd_type },
    { "user", FTPd_user },
    { "xcwd", FTPd_cd },
    { "xmkd", FTPd_mkdir },
    { "xpwd", FTPd_pwd },
    { "xrmd", FTPd_rmdir },
    { NULL, NULL }
```

## Function Reference

```
};  
FTPD_userfile = "userfile:";  
  
/* Start FTP Server: */  
error = FTPd_init("FTP server", 7, 2000);  
if (error) return error;  
printf("\nFTP Server is running");  
return 0;
```

## 7.1.40 FTP\_open()

Starts an FTP session.

### Synopsis

```
int_32  FTP_open(
    pointer_PTR_  handle_ptr,
    _ip_address   server_addr,
    FILE_PTR      ctrl_fd)
```

### Parameters

*handle\_ptr* [in] — FTP session handle.  
*server\_addr* [in] — IP address of the FTP server.  
*ctrl\_fd* [in] — Device to write control-connection responses to.

### Description

This function establishes a connection to the specified FTP server. If successful, the functions **FTP\_command()** and **FTP\_command\_data()** can be called to issue commands to the FTP Server.

### Return Value

- An FTP response code (success)
- -1 (failure)

### See Also

- [FTP\\_close\(\)](#)

### Example

```
#include <mqx.h>
#include <bsp.h>
#include <rtcs.h>

void main_task
(
    uint_32  dummy
)
{ /* Body */
    pointer ftphandle;
    int_32  response;

    response = FTP_open(&ftphandle, SERVER_ADDRESS, stdout);
    if (response == -1) {
        printf("Couldn't open FTP session\n");
        return;
    } /* Endif */

    response = FTP_command(ftphandle, "USER anonymous\r\n",
        stdout);

    /* response 3xx means Password Required */
    if ((response >= 300) && (response < 400)) {
```

```
        response = FTP_command(ftphandle, "PASS password\r\n",
                                stdout);
    } /* Endif */

    /* response 2xx means Logged In */
    if ((response >= 200) && (response < 300)) {
        response = FTP_command_data(ftphandle, "LIST\r\n", stdout,
                                    stdout, FTPMODE_PORT | FTPDIR_RECV);
    } /* Endif */

    FTP_close(ftphandle, stdout);

} /* Endbody */
```



### 7.1.41 FTPSRV\_init()

Starts the FTP Server.

#### Synopsis

```
uint_32  FTPSRV_init(
    char_ptr  name,
    uint_32   priority
    uint_32   stacksize)
```

#### Parameters

*name* [in] — Name of the server's task.  
*priority* [in] — Priority of the server's task.  
*stacksize* [in] — Stack size for the server's task.

#### Description

Function **FTPSRV\_init()** starts the FTP Server with task priority *priority* (we recommend that you make the priority lower than the priority of the RTCS task; that is, make it a higher number).

It also creates *FTPSRV\_task*.

#### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

#### Example

```
uint_32  error;

/* Start FTP Server: */
error = FTPSRV_init("FTP server", 7, 1000);
if (error) return error;
printf("\nFTP Server is running");
return 0;
```

## 7.1.42 gethostbyaddr()

Gets the *HOSTENT\_STRUCT* structure for an IP address.

### Synopsis

```
hostent _PTR_ gethostbyaddr(
    const char _PTR_ addr_ptr,
    uint_32 len,
    uint_32 type)
```

### Parameters

*addr\_ptr [in]* — Pointer to the IP address in numeric form.  
*len [in]* — Length of the address; must be sizeof(struct in\_addr).  
*type [in]* — Type of address; must be *AF\_INET*.

### Description

If the function is successful, a static *HOSTENT\_STRUCT* is overwritten every time that the function is called.

### Return Value

- Pointer to a *HOSTENT\_STRUCT* structure (success)
- NULL (failure)

### See Also

- [gethostbyname\(\)](#)
- [HOSTENT\\_STRUCT](#)

### Example

```
struct in_addr      my_ip_address;
struct hostent _PTR_ hostname;

/* Initialize my_ip_address.s_addr: */
hostname = gethostbyaddr(&my_ip_address,
                        sizeof(my_ip_address),
                        AF_INET);
printf("Hostname is %s.\n", hostname->h_name);
```

## 7.1.43 gethostbyname()

Gets the *HOSTENT\_STRUCT* structure for a host name.

### Synopsis

```
HOSTENT_STRUCT_PTR gethostbyname(  
    char_ptr name)
```

### Parameters

*name* [in] — Pointer to a string that is a properly formatted domain name (see description).  
 Pointer to a string that is a properly formatted domain name (see description).

### Return Value

- Pointer to a *HOSTENT\_STRUCT* structure (success)
- NULL (failure; see table)

If:	Function returns:
More than eight aliases are encountered or the alias names the loop.	Immediately
Name does not exist in the public name space.	Name error
Name is an alias.	Canonical name and its IP address
Query is successful.	Name and its IP address
Query times out and no response is received.	Timeout error

### Description

This function provides information on server *name*, where *name* is a domain name or IP address.

For a full description of the requirements for formatting *name*, see RFCs 1034 and 1035. If *name* is terminated by a period (.), the name is an absolute domain name (NULL follows the period, and NULL is the default name for the root server of any domain tree). If the string is not terminated by a period, the name is a relative domain name. For more information on setting up and using DNS Resolver, see [Section 5.4, “DNS Resolver.”](#)

If the function is successful, a static *HOSTENT\_STRUCT* is overwritten every time that the function is called.

The following fields in the *HOSTENT\_STRUCT* always have the following values:

Field	Value
<i>h_addrtype</i>	AF_INET
<i>h_length</i>	sizeof(struct in_addr)

**See Also**

- [DNS\\_init\(\)](#)
- [gethostbyaddr\(\)](#)
- [HOSTENT\\_STRUCT](#)

**Example**

```

HOSTENT_STRUCT
char
char_ptr
char_ptr
char_ptr
uint_32
_ip_addr

host;
string[30];
name;
alias1;
alias2;
type, length;
ip;

strcpy(string, "sparky.com");
host = gethostbyname(string);
if (host != NULL) {
    name    = host->h_name;
    alias1  = host->h_aliases[0];
    alias2  = host->h_aliases[1];
    type    = host->h_addrtype;
    length  = host->h_length;
    ip      = *(uint_32_ptr)host->h_addr_list[0];
}

```

## 7.1.44 getpeername()

Gets the remote-endpoint identifier of a socket.

### Synopsis

```
uint_32  getpeername(
    uint_32      socket,
    sockaddr_in  _PTR_ name,
    uint_16      _PTR_ namelen)
```

### Parameters

*socket* [*in*] — Handle for the stream socket.

*name* [*out*] — Pointer to a placeholder for the remote-endpoint identifier of the socket.

*namelen* [*in/out*] — When passed in: Pointer to the length, in bytes, of what *name* points to.

When passed out: Full size, in bytes, of the remote-endpoint identifier.

### Description

Function **getpeername()** finds the remote-endpoint identifier of socket *socket* as was determined by **connect()** or **accept()**. This function blocks, but the command is immediately serviced and replied to.

### Return Value

- *RTCS\_OK* (success)
- Specific error code (failure)

### See Also

- [accept\(\)](#)
- [connect\(\)](#)
- [getsockname\(\)](#)
- [socket\(\)](#)

### Example

```
uint_32      handle;
sockaddr_in  remote_sin;
uint_32      status;
uint_16      namelen;

...

namelen = sizeof (sockaddr_in);
status = getpeername(handle, &remote_sin, &namelen);
if (status != RTCS_OK)
{
    printf("\nError, getpeername() failed with error code %lx",
        status);
} else {
    printf("\nRemote address family is %x", remote_sin.sin_family);
    printf("\nRemote port is %d", remote_sin.sin_port);
}
```

## Function Reference

```
printf("\nRemote IP address is %lx",
    remote_sin.sin_addr.s_addr);
}
```

## 7.1.45 getsockname()

Gets the local-endpoint identifier of the socket.

### Synopsis

```
uint_32  getsockname(
    uint_32      socket,
    sockaddr_in  _PTR_ name,
    uint_16      _PTR_ namelen)
```

### Parameters

*socket* [in] — Socket handle

*name* [out] — Pointer to a placeholder for the remote-endpoint identifier of the socket.

*namelen* [in/out] — When passed in: Pointer to the length, in bytes, of what *name* points to.

When passed out: Full size, in bytes, of the remote-endpoint identifier.

### Description

Function **getsockname()** returns the local endpoint for the socket as was defined by **bind()**. This function blocks, but the command is immediately serviced and replied to.

### Return Value

- **RTCS\_OK** (success)
- Specific error code (failure)

### See Also

- [bind\(\)](#)
- [getpeername\(\)](#)
- [socket\(\)](#)

### Example

```
uint_32      handle;
sockaddr_in  local_sin;
uint_32      status;
uint_16      namelen;

...

namelen = sizeof (sockaddr_in);
status = getsockname(handle, &local_sin, &namelen);

if (status != RTCS_OK)
{
    printf("\nError, getsockname() failed with error code %lx",
        status);
} else {
    printf("\nLocal address family is %x", local_sin.sin_family);
    printf("\nLocal port is %d", local_sin.sin_port);
    printf("\nLocal IP address is %lx", local_sin.sin_addr.s_addr);
}
```

## 7.1.46 getsockopt()

Gets the value of the socket option.

### Synopsis

```
uint_32  getsockopt (
    uint_32      socket,
    int_32       level,
    uint_32      optname,
    pointer      optval,
    uint_32      _PTR_ optlen)
```

### Parameters

*socket [in]* — Socket handle.

*level [in]* — Protocol level, at which the option resides.

*optname [in]* — Option name (see description).

*optval [in/out]* — Pointer to the option value.

*optlen [in/out]* — When passed in: Size of *optval* in bytes.

When passed out: Full size, in bytes, of the option value.

### Description

An application can get all socket options for all protocol levels. For a complete description of socket options and protocol levels, see **setsockopt()**. This function blocks, but the command is immediately serviced and replied to.

### Return Value

- *RTCS\_OK* (success)
- Specific error code (failure)

### See Also

- [setsockopt\(\)](#)



## 7.1.47 httpd\_default\_params()

Initializes the HTTP server parameter structure to default values.

### Synopsis

```
HTTPD_PARAMS_STRUCT* httpd_default_params(
    HTTPD_PARAMS_STRUCT *params)
```

### Parameters

*params* [in] — pointer to parameter structure which will be set to default values. If NULL, the structure is allocated dynamically.

### Description

This function prepares HTTP server parameter structure and sets all its members to default values. If *params* argument is NULL, the function allocates the parameter structure dynamically. Default parameter values are defined as constants in internal HTTP header files and can be overridden by *user\_config.h* user configuration file.

This function should be called if a HTTP server is to be run with other than default parameters (compiled-in during RTCS and HTTP build). The parameters and structure returned by this function may be further modified before it is passed to [httpd\\_init\(\)](#) function.

### Return Value

Pointer to HTTP parameters structure. If a valid pointer is passed to the function in *params* argument, the returned pointer is equal to this value. If NULL is passed as *params* argument, this function returns pointer to a newly allocated memory containing the initialized structure.

### See Also

- [httpd\\_init\(\)](#)
- [httpd\\_server\\_init\(\)](#)

### Example

```
/* allocate default values */
params = httpd_default_params(NULL);

if (params)
{
    /* change some parameter values */
    params->root_dir = (HTTPD_ROOT_DIR_STRUCT*)root_dir;
    params->index_page = "\\index.html";
    params->max_ses = 1;

    /* initialize HTTP */
    server = httpd_init(params);
}
```

## 7.1.48 httpd\_init()

This function initializes the HTTP server.

### Synopsis

```
HTTPD_STRUCT* httpd_init(
    HTTPD_PARAMS_STRUCT *params)
```

### Parameters

*params* [in] — pointer to parameter structure to be used by the HTTP server. This should not be NULL.

### Description

This is the main HTTP initialization function which should be called before the server is started. This function uses the information passed in the parameter structure to allocate internal memory buffers and to initialize internal server and session structures.

After the HTTP server is initialized by this call, some of the server parameters can still be changed using one of the **HTTP\_SET\_xxx** calls (macros). However, parameters like port number or number of sessions can not be changed after this call.

### Return Value

Pointer to HTTP to server structure.

### See Also

- [httpd\\_default\\_params\(\)](#)
- [httpd\\_server\\_init\(\)](#)

### Example

```
/* allocate default values */
params = httpd_default_params(NULL);

if (params)
{
    /* change some parameter values */
    params->root_dir = (HTTPD_ROOT_DIR_STRUCT*)root_dir;
    params->index_page = "\\index.html";
    params->max_ses = 1;

    /* initialize HTTP */
    server = httpd_init(params);
}
```

## 7.1.49 `httpd_server_init()`

This function initializes the HTTP server task using the default parameters.

### Synopsis

```
HTTPD_STRUCT* httpd_server_init(
    HTTPD_ROOT_DIR_STRUCT *root_dir,
    const char *index_page)
```

### Parameters

*root\_dir* [in] — pointer to `HTTPD_ROOT_DIR_STRUCT` which contains web server root directories (mapping between web directories and physical filesystem paths).

*index\_page* [in] — Filename of the default index page (relative to root directory)

### Description

This function is a simple wrapper around `httpd_default_params()` and `httpd_init()` functions. Use this call to prepare the HTTP to be started with the default (compiled-in) settings.

After the HTTP server is initialized by this call, some of the server parameters can still be changed using one of the `HTTP_SET_xxx` calls (macros). However, parameters like port number or number of sessions can not be changed after this call.

### Return Value

Pointer to HTTP to server structure.

### See Also

- [httpd\\_default\\_params\(\)](#)
- [httpd\\_server\\_run\(\)](#)
- [httpd\\_server\\_poll\(\)](#)

### Example

```
HTTPD_ROOT_DIR_STRUCT root_dir[] = {
    { "", "tfs:" },
    { 0, 0 } // table termination record
};

...

HTTPD_STRUCT *server;
server = httpd_server_init(root_dir, "\\mqx.html");
HTTPD_SET_PARAM_CGI_TBL(server, cgi_lnk_tbl);
HTTPD_SET_PARAM_FN_TBL(server, fn_lnk_tbl);

httpd_server_run(server);
```

## 7.1.50 httpd\_server\_run()

This function starts the HTTP server task (or tasks).

### Synopsis

```
int httpd_server_run(HTTPD_STRUCT *server)
```

### Parameters

*server* [in] — pointer to main server structure [HTTPD\\_STRUCT](#) returned by [httpd\\_init\(\)](#) or [httpd\\_server\\_init\(\)](#)

### Description

Depending on HTTPDCFG\_POLL\_MODE compilation parameter (from *user\_config.h*), this function starts the HTTP server tasks or tasks:

- With HTTPDCFG\_POLL\_MODE set non-zero, single HTTP server task is started. In this mode, all sessions are handled (polled) from a single task. Alternatively, the HTTP server may be polled from the caller's task by calling [httpd\\_server\\_poll\(\)](#) periodically.
- With HTTPDCFG\_POLL\_MODE set zero, multiple tasks are started, one for each HTTP session.

### Return Value

Returns zero if HTTP task (or all tasks) were started successfully, otherwise it returns non-zero number. If the return value is positive, it reports how many session tasks could not be created. If the return value is negative, the single HTTP server task failed to start.

### See Also

- [httpd\\_server\\_init\(\)](#)

### Example

```
#define HTTPD_SEPARATE_TASK 0 /* select how to run the HTTP server */

HTTPD_ROOT_DIR_STRUCT root_dir[] = {
    { "", "tfs:" },
    { 0, 0 } // table termination record
};

...

HTTPD_STRUCT *server;
server = httpd_server_init(root_dir, "\\index.html");

#if HTTPD_SEPARATE_TASK || !HTTPDCFG_POLL_MODE
    httpd_server_run(server);
    /* .. code continues here */
#else
while (1)
{
    httpd_server_poll(server, 1);
    /* user stuff come here - only non blocking calls */
}
#endif
```

## 7.1.51 httpd\_server\_poll()

Single-step the HTTP server handling.

### Synopsis

```
void httpd_server_poll(
    HTTPD_STRUCT *server,
    int to)
```

### Parameters

*server* [in] — pointer to main server structure [HTTPD\\_STRUCT](#) returned by [httpd\\_init\(\)](#) or [httpd\\_server\\_init\(\)](#)

*to* [in] — timeout for blocking functions during the poll processing - for example socket functions

### Description

If the HTTP server runs in polled mode (HTTPDCFG\_POLL\_MODE set non-zero in user\_config.h), an application can handle server processing by calling **httpd\_server\_poll()** periodically.

### Return Value

none

### See Also

- [httpd\\_init\(\)](#)
- [httpd\\_server\\_init\(\)](#)

### Example

```
HTTPD_ROOT_DIR_STRUCT root_dir[] = {
    { "", "tfs:" },
    { 0, 0 } // table termination record
};

HTTPD_STRUCT *server;
server = httpd_server_init(root_dir, "\\index.html");

while (1)
{
    httpd_server_poll(server, 1);
    /* user stuff come here - only non blocking calls */
}
```

## 7.1.52 HTTPD\_SET\_PARAM\_ROOT\_DIR

Macro setting the HTTP server root directory mapping.

### Synopsis

```
HTTPD_SET_PARAM_ROOT_DIR(server, val)
```

### Parameters

*server* [in] — pointer to server structure returned by [httpd\\_init\(\)](#) or [httpd\\_server\\_init\(\)](#)

*val* [in] — pointer to root directories table (array of `HTTPD_ROOT_DIR_STRUCT`)

### Description

This macro sets the root directory mapping array pointer in the HTTP server structure.

### Example

```
HTTPD_ROOT_DIR_STRUCT root_dir[] =
{
    { "", "tfs:" },
    { "usb", "c:" },
    { 0, 0 } // table termination record
};

....

/* allocate default values */
params = httpd_default_params(NULL);

if (params)
{
    server = httpd_init(params);
    HTTPD_SET_PARAM_ROOT_DIR(server, root_dir);
}
```

### 7.1.53 HTTPD\_SET\_PARAM\_INDEX\_PAGE

Macro setting the HTTP server default page name.

#### Synopsis

```
HTTPD_SET_PARAM_INDEX_PAGE(server, val)
```

#### Parameters

*server* [in] — pointer to server structure returned by [httpd\\_init\(\)](#) or [httpd\\_server\\_init\(\)](#)

*val* [in] — pointer to string with index page filename (`char_ptr` type)

#### Description

This macro sets the file name (relative path to) the default HTTP server page.

#### Example

```
HTTPD_ROOT_DIR_STRUCT root_dir[] =
{
    { "", "tfs:" },
    { "usb", "c:" },
    { 0, 0 } // table termination record
};

....

/* allocate default values */
params = httpd_default_params(NULL);

if (params)
{
    server = httpd_init(params);
    HTTPD_SET_PARAM_ROOT_DIR(server, root_dir);
    HTTPD_SET_PARAM_INDEX_PAGE(server, "\\index.htm");
}
```

## 7.1.54 HTTPD\_SET\_PARAM\_FN\_TBL

Macro setting the HTTP server ASP-like callback functions table.

### Synopsis

```
HTTPD_SET_PARAM_FN_TBL(server, val)
```

### Parameters

*server* [in] — pointer to server structure returned by [httpd\\_init\(\)](#) or [httpd\\_server\\_init\(\)](#)

*val* [in] — pointer to script support functions table (array of `HTTPD_FN_LINK_STRUCT`)

### Example

```
HTTPD_FN_LINK_STRUCT cgi_lnk_tbl[] =
{
    { "time", fn_time},
    { 0, 0 } // table termination record
};

... // server initialization

HTTPD_SET_PARAM_FN_TBL(server, fn_lnk_tbl);
```



## 7.1.55 HTTPD\_SET\_PARAM\_CGI\_TBL

Macro setting the HTTP server CGI callback functions table.

### Synopsis

```
HTTPD_SET_PARAM_CGI_TBL(server, val)
```

### Parameters

*server* [in] — pointer to server structure returned by [httpd\\_init\(\)](#) or [httpd\\_server\\_init\(\)](#)

*val* [in] — pointer to CGI functions table (array of `HTTPD_CGI_LINK_STRUCT`)

### Example

```
HTTPD_CGI_LINK_STRUCT cgi_lnk_tbl[] = {
    { "ipstat",          cgi_ipstat},
    { "icmpstat",        cgi_icmpstat},
    { "udpstat",          cgi_udpstat},
    { "tcpstat",          cgi_tcpstat},
    { "analog",           cgi_analog_data},
    { "rtcdata",           cgi_rtc_data},
    { "toggleled1",       cgi_toggle_led1},
    { "toggleled2",       cgi_toggle_led2},
    { "toggleled3",       cgi_toggle_led3},
    { "toggleled4",       cgi_toggle_led4},
    { 0, 0 } // table termination record
};

... // server initialization

HTTPD_SET_PARAM_CGI_TBL(server, cgi_lnk_tbl);
```

## 7.1.56 ICMP\_stats()

Gets a pointer to the ICMP statistics.

### Synopsis

```
ICMP_STATS_PTR  ICMP_stats(void)
```

### Description

Function **ICMP\_stats()** takes no parameters, and returns a pointer to the ICMP statistics that RTCS collects.

### Return Value

Pointer to the *ICMP\_STATS* structure.

### See Also

- [ARP\\_stats\(\)](#)
- [ENET\\_get\\_stats\(\)](#)
- [ICMP\\_stats\(\)](#)
- [IP\\_stats\(\)](#)
- [IPIF\\_stats\(\)](#)
- [TCP\\_stats\(\)](#)
- [UDP\\_stats\(\)](#)
- *ICMP\_STATS*

### Example

See [ARP\\_stats\(\)](#).

## 7.1.57 IGMP\_stats()

Gets a pointer to the IGMP statistics.

### Synopsis

```
IGMP_STATS_PTR IGMP_stats(void)
```

### Description

Function **IGMP\_stats()** takes no parameters, and returns a pointer to the IGMP statistics that RTCS collects.

### Return Value

Pointer to the *IGMP\_STATS* structure.

### See Also

- [ARP\\_stats\(\)](#)
- [ENET\\_get\\_stats\(\)](#)
- [ICMP\\_stats\(\)](#)
- [IP\\_stats\(\)](#)
- [IPIF\\_stats\(\)](#)
- [TCP\\_stats\(\)](#)
- [UDP\\_stats\(\)](#)
- *IGMP\_STATS*

### Example

See [ARP\\_stats\(\)](#).

## 7.1.58 IP\_stats()

Gets a pointer to the IP statistics.

### Synopsis

```
IP_STATS_PTR IP_stats(void)
```

### Description

Function **IP\_stats()** takes no parameters and returns a pointer to the IP statistics that RTCS collects.

### Return Value

Pointer to the *IP\_STATS* structure.

### See Also

- [ARP\\_stats\(\)](#)
- [ENET\\_get\\_stats\(\)](#)
- [ICMP\\_stats\(\)](#)
- [IGMP\\_stats\(\)](#)
- [IPIF\\_stats\(\)](#)
- [TCP\\_stats\(\)](#)
- [UDP\\_stats\(\)](#)
- *IP\_STATS*

### Example

See [ARP\\_stats\(\)](#).

## 7.1.59 IPIF\_stats()

Gets a pointer to the IPIF statistics that RTCS collects for the device interface.

### Synopsis

```
IPIF_STATS_PTR IPIF_stats(  
    _rtcs_if_handle rtcs_if_handle)
```

### Parameters

*rtcs\_if\_handle* [in] — RTCS interface handle.

### Description

Function **IPIF\_stats()** returns a pointer to the IPIF statistics that RTCS collects for the device interface.

### Return Value

- Pointer to the *IPIF\_STATS* structure (success)
- NULL (failure: *rtcs\_if\_handle* is invalid)

### See Also

- [ARP\\_stats\(\)](#)
- [ENET\\_get\\_stats\(\)](#)
- [ICMP\\_stats\(\)](#)
- [IGMP\\_stats\(\)](#)
- [IP\\_stats\(\)](#)
- [TCP\\_stats\(\)](#)
- [UDP\\_stats\(\)](#)
- [IPIF\\_STATS](#)

### Example

See [ARP\\_stats\(\)](#).

## 7.1.60 ipcfg\_init\_device()

Initializes the Ethernet device, adds network interface and setups the IPCFG context for it.

### Synopsis

```
uint_32 ipcfg_init_device(
    uint_32 device,
    _enet_address mac)
```

### Parameters

*device* [in] — device identification (index)

*mac* [in] — Ethernet MAC address

### Description

This function initializes the ethernet device (calls ENET\_initialize internally), adds network interface (RTCS\_if\_add) to the RTCS and sets up ipcfg context for the device.

### Return Value

- *IPCFG\_OK* (success)
- *RTCSERR\_IPCFG\_BUSY*
- *RTCSERR\_IPCFG\_DEVICE\_NUMBER*
- *RTCSERR\_IPCFG\_INIT*

### See Also

- [ipcfg\\_init\\_interface\(\)](#)
- [RTCS\\_if\\_add\(\)](#)

### Example

```
#define ENET_IPADDR  IPADDR(192,168,1,4)
#define ENET_IPMASK  IPADDR(255,255,255,0)
#define ENET_IPGATEWAY  IPADDR(192,168,1,1)

uint_32 setup_network(void)
{
    uint_32          error;
    IPCFG_IP_ADDRESS_DATA ip_data;
    _enet_address     enet_address;

    ip_data.ip = ENET_IPADDR;
    ip_data.mask = ENET_IPMASK;
    ip_data.gateway = ENET_IPGATEWAY;

    /* Create TCP/IP task */
    error = RTCS_create();
    if (error) return error;

    /* Get the Ethernet address of the device */
    ENET_get_mac_address (BSP_DEFAULT_ENET_DEVICE, ENET_IPADDR, enet_address);

    /* Initialize the Ethernet device */
    error = ipcfg_init_device (BSP_DEFAULT_ENET_DEVICE, enet_address);
```

```
if (error) return error;

/* Bind Ethernet device to network using constant (static) IP address information */
error = ipcfg_bind_staticip(BSP_DEFAULT_ENET_DEVICE, &ip_data);
if (error) return error;

return 0;
}
```

## 7.1.61 ipcfg\_init\_interface()

Setups IPCFG context for already initialized device and its interface.

### Synopsis

```
uint_32 ipcfg_init_interface(
    uint_32 device_number,
    _rtcs_if_handle ihandle)
```

### Parameters

*device\_number* [in] — device number  
*ihandle* [in] — interface handle

### Description

This function sets up the IPCFG context for network interface already initialized by other RTCS calls.

### Return Value

- *IPCFG\_OK* (success)
- *RTCSERR\_IPCFG\_BUSY*
- *RTCSERR\_IPCFG\_DEVICE\_NUMBER*
- *RTCSERR\_IPCFG\_INIT*

### See Also

- [ipcfg\\_init\\_device\(\)](#)

### Example

```
#define ENET_IPADDR  IPADDR(192,168,1,4)
#define ENET_IPMASK  IPADDR(255,255,255,0)
#define ENET_IPGATEWAY  IPADDR(192,168,1,1)
```

```
uint_32 setup_network(void)
{
    uint_32          error;
    IPCFG_IP_ADDRESS_DATA ip_data;
    _enet_address    enet_address;
    _enet_handle     ehandle;
    _rtcs_if_handle  ihandle;

    ip_data.ip = ENET_IPADDR;
    ip_data.mask = ENET_IPMASK;
    ip_data.gateway = ENET_IPGATEWAY;

    error = RTCS_create();
    if (error) return error;

    ENET_get_mac_address (BSP_DEFAULT_ENET_DEVICE, ENET_IPADDR, enet_address);

    error = ENET_initialize(BSP_DEFAULT_ENET_DEVICE, enet_address, 0, &ehandle);
    if (error) return error;

    error = RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);
    if (error) return error;
```



```
error = ipcfg_init_interface(BSP_DEFAULT_ENET_DEVICE, ihandle);  
if (error) return error;  
  
return ipcfg_bind_autoip(BSP_DEFAULT_ENET_DEVICE, &ip_data);  
}
```

## 7.1.62 ipcfg\_bind\_boot()

Binds Ethernet device to network using the BOOT protocol.

### Synopsis

```
uint_32 ipcfg_bind_boot(
    uint_32 device)
```

### Parameters

*device [in]* — device identification

### Description

This function tries to bind the device to network using BOOT protocol. It also gathers information about TFTP server and file to download. It is blocking function, i.e. doesn't return until the process is finished or error occurs.

Any failure during bind leaves the network interface in unbound state.

### Return Value

- *IPCFG\_OK* (success)
- *RTCSERR\_IPCFG\_BUSY*
- *RTCSERR\_IPCFG\_DEVICE\_NUMBER*
- *RTCSERR\_IPCFG\_INIT*
- *RTCSERR\_IPCFG\_BIND*

### See Also

- [ipcfg\\_unbind\(\)](#)

### Example

```
#define ENET_IPADDR  IPADDR(192,168,1,4)
#define ENET_IPMASK  IPADDR(255,255,255,0)
#define ENET_IPGATEWAY  IPADDR(192,168,1,1)

uint_32 setup_network(void)
{
    uint_32          error;
    _enet_address    enet_address;

    error = RTCS_create();
    if (error) return error;

    ENET_get_mac_address (BSP_DEFAULT_ENET_DEVICE, ENET_IPADDR, enet_address);

    error = ipcfg_init_device(BSP_DEFAULT_ENET_DEVICE, enet_address);
    if (error) return error;

    error = ipcfg_bind_boot(BSP_DEFAULT_ENET_DEVICE);
    if (error) return error;

    TFTPtip = ipcfg_get_tftp_serveraddress(BSP_DEFAULT_ENET_DEVICE);
    TFTPserver = ipcfg_get_tftp_servername(BSP_DEFAULT_ENET_DEVICE);
    TFTPfile = ipcfg_get_boot_filename(BSP_DEFAULT_ENET_DEVICE);
}
```

### 7.1.63 ipcfg\_bind\_dhcp()

Binds Ethernet device to network using DHCP protocol (polling mode).

#### Synopsis

```
uint_32 ipcfg_bind_dhcp(
    uint_32 device,
    boolean try_auto_ip)
```

#### Parameters

*device [in]* — device identification

*try\_auto\_ip [in]* — try the auto-ip automatic assign address if DHCP binding fails

#### Description

This function initiates the process of binding the device to network using the DHCP protocol. As the DHCP address resolving may take up to one minute, there are two separate non-blocking functions related to the DHCP binding.

**ipcfg\_bind\_dhcp()** must be called first, repeatedly, till it returns a result other than `RTCSERR_IPCFG_BUSY`. In case this function returns `IPCFG_OK`, the process may continue by calling **ipcfg\_poll\_dhcp()** periodically again until the result is other than `RTCSERR_IPCFG_BUSY`.

Both functions must be called with same value of the first two parameters.

According to second parameter, additional auto IP binding can take place after DHCP fails.

The polling process should be aborted if any of the two functions return result other than `RTCS_OK` or `RTCSERR_IPCFG_BUSY`. In this case, the network interface is left in unbound state.

An alternative (blocking) method of DHCP bind is **ipcfg\_bind\_dhcp\_wait()**. See the example below how this call is implemented internally.

#### Return Value

- `IPCFG_OK` (success)
- `RTCSERR_IPCFG_BUSY`
- `RTCSERR_IPCFG_DEVICE_NUMBER`
- `RTCSERR_IPCFG_INIT`
- `RTCSERR_IPCFG_BIND`

#### See Also

- **ipcfg\_poll\_dhcp()**
- **ipcfg\_unbind()**
- **ipcfg\_bind\_dhcp\_wait()**

## Example

```
uint_32 ipcfg_bind_dhcp_wait(uint_32 device, boolean try_auto_ip,
                             IPCFG_IP_ADDRESS_DATA_PTR auto_ip_data)
{
    uint_32 result = IPCFG_OK;

    do
    {
        if (result == RTCSEERR_IPCFG_BUSY) _time_delay(200);
        result = ipcfg_bind_dhcp(device, try_auto_ip);
    } while (result == RTCSEERR_IPCFG_BUSY);
    if (result != IPCFG_OK) return result;
    do
    {
        _time_delay (200);
        result = ipcfg_poll_dhcp(device, try_auto_ip, auto_ip_data);
    } while (result == RTCSEERR_IPCFG_BUSY);
    return result;
}
```

## 7.1.64 ipcfg\_bind\_dhcp\_wait()

Binds Ethernet device to network using DHCP protocol (blocking mode).

### Synopsis

```
uint_32 ipcfg_bind_dhcp_wait(
    uint_32 device,
    boolean try_auto_ip,
    IPCFG_IP_ADDRESS_DATA_PTR auto_ip_data)
```

### Parameters

*device* [in] — device identification

*try\_auto\_ip* [in] — try the auto-ip automatic assign address if DHCP binding fails

*auto\_ip\_data* [in] — ip, mask and gateway information used by auto-IP binding (may be NULL)

### Description

This function tries to bind the device to network using the DHCP protocol, optionally followed by auto IP bind if DHCP fails. It is blocking function, i.e. doesn't return until the process is finished or error occurs.

According to second parameter, an additional auto IP binding can take place if DHCP fails. When the third parameter is NULL, the last successful bind information is used as an input to auto IP binding.

Any failure during bind leaves the network interface in unbound state.

### Return Value

- *IPCFG\_OK* (success)
- *RTCSERR\_IPCFG\_BUSY*
- *RTCSERR\_IPCFG\_DEVICE\_NUMBER*
- *RTCSERR\_IPCFG\_INIT*
- *RTCSERR\_IPCFG\_BIND*

### See Also

- [ipcfg\\_bind\\_dhcp\(\)](#)
- [ipcfg\\_poll\\_dhcp\(\)](#)

### Example

```
#define ENET_IPADDR  IPADDR(192,168,1,4)
#define ENET_IPMASK  IPADDR(255,255,255,0)
#define ENET_IPGATEWAY IPADDR(192,168,1,1)

uint_32 setup_network(void)
{
    uint_32          error;
    IPCFG_IP_ADDRESS_DATA auto_ip_data;
    _enet_address    enet_address;

    auto_ip_data.ip = ENET_IPADDR;
    auto_ip_data.mask = ENET_IPMASK;
    auto_ip_data.gateway = ENET_IPGATEWAY;
```

## Function Reference

```
error = RTCS_create();  
if (error) return error;  
  
ENET_get_mac_address (BSP_DEFAULT_ENET_DEVICE, ENET_IPADDR, enet_address);  
  
error = ipcfg_init_device(BSP_DEFAULT_ENET_DEVICE, enet_address);  
if (error) return error;  
  
return ipcfg_bind_dhcp_wait(BSP_DEFAULT_ENET_DEVICE, TRUE, &auto_ip_data);  
}
```

## 7.1.65 `ipcfg_bind_staticip()`

Binds Ethernet device to network using constant (static) IP address information.

### Synopsis

```
uint_32 ipcfg_bind_staticip(
    uint_32 device,
    IPCFG_IP_ADDRESS_DATA_PTR static_ip_data)
```

### Parameters

*device* [in] — device identification

*static\_ip\_data* [in] — pointer to ip, mask and gateway structure

### Description

This function tries to bind device to network using given IP address information. If the address is already used, an error is returned. This is blocking function, i.e. doesn't return until the process is finished or error occurs.

Any failure during bind leaves the network interface in unbound state.

### Return Value

- *IPCFG\_OK* (success)
- *RTCSERR\_IPCFG\_BUSY*
- *RTCSERR\_IPCFG\_DEVICE\_NUMBER*
- *RTCSERR\_IPCFG\_INIT*
- *RTCSERR\_IPCFG\_BIND*

### See Also

- [ipcfg\\_unbind\(\)](#)

### Example

See [ipcfg\\_init\\_device\(\)](#)

## 7.1.66 ipcfg\_get\_device\_number()

Returns the Ethernet device number for given RTCS interface.

### Synopsis

```
uint_32 ipcfg_get_device_number(  
    _rtcs_if_handle ihandle)
```

### Parameters

*ihandle [in]* — interface handle

### Description

Simple function returning the Ethernet device number by giving an RTCS interface handle.

### Return Value

Device number if successful, otherwise -1.

### See Also

- [ipcfg\\_get\\_ihandle\(\)](#)



## 7.1.67 ipcfg\_add\_interface()

Add new interface and returns corresponding device number.

### Synopsis

```
uint_32 ipcfg_add_interface(  
    uint_32 device_number,  
    _rtcs_if_handle ihandle)
```

### Parameters

*device\_number [in]* — device number

*ihandle [in]* — interface handle

### Description

Internally, this function makes the association between ihandle and the device number.

### Return Value

Device number if successful, otherwise -1.

### See Also

- [ipcfg\\_get\\_ihandle\(\)](#)
- [ipcfg\\_get\\_device\\_number\(\)](#)

## 7.1.68 ipcfg\_get\_ihandle()

Returns the RTCS interface handle for given Ethernet device number.

### Synopsis

```
_rtcs_if_handle ipcfg_get_ihandle(  
    uint_32 device)
```

### Parameters

*device [in]* — device identification

### Description

Simple function returning the RTCS interface handle by giving an Ethernet device number.

### Return Value

Interface handle if successful, NULL otherwise.

### See Also

- [ipcfg\\_get\\_device\\_number\(\)](#)

## 7.1.69 ipcfg\_get\_mac()

Returns the Ethernet MAC address.

### Synopsis

```
boolean ipcfg_get_mac(  
    uint_32 device,  
    _enet_address mac)
```

### Parameters

*device [in]* — device identification

*mac [in]* — pointer to mac address structure

### Description

Simple function returning the Ethernet MAC address by giving Ethernet device number.

### Return Value

TRUE if successfull (MAC address filled), otherwise FALSE.

## 7.1.70 ipcfg\_get\_state()

Returns the IPCFG state for a given Ethernet device.

### Synopsis

```
IPCFG_STATE ipcfg_get_state(  
    uint_32 device)
```

### Parameters

*device [in]* — device identification

### Description

This function returns an immediate state of Ethernet device as it is evaluated by the IPCFG engine.

### Return Value

Actual IPCFG status (`enum IPCFG_STATE` value).

One of

- `IPCFG_STATE_INIT`
- `IPCFG_STATE_UNBOUND`
- `IPCFG_STATE_BUSY`
- `IPCFG_STATE_STATIC_IP`
- `IPCFG_STATE_DHCP_IP`
- `IPCFG_STATE_AUTO_IP`
- `IPCFG_STATE_DHCPAUTO_IP`
- `IPCFG_STATE_BOOT`

### See Also

- [ipcfg\\_get\\_state\\_string\(\)](#)
- [ipcfg\\_get\\_desired\\_state\(\)](#)

### Example

### 7.1.71 ipcfg\_get\_state\_string()

Converts IPCFG status value to string.

#### Synopsis

```
const char_ptr ipcfg_get_state_string(  
    IPCFG_STATE state)
```

#### Parameters

*state* [in] — status identification

#### Description

This function may be used to display the IPCFG status value in text messages.

#### Return Value

Pointer to status string or NULL.

#### See Also

- [ipcfg\\_get\\_state\(\)](#)
- [ipcfg\\_get\\_desired\\_state\(\)](#)

## 7.1.72 ipcfg\_get\_desired\_state()

Returns the target IPCFG state for a given Etherent device.

### Synopsis

```
IPCFG_STATE ipcfg_get_desired_state(  
    uint_32 device)
```

### Parameters

*device [in]* — device identification

### Description

This function returns the target state the user requires to reach with the given Ethernet device.

### Return Value

The desired IPCFG status (`enum IPCFG_STATE` value).

One of

- `IPCFG_STATE_UNBOUND`
- `IPCFG_STATE_STATIC_IP`
- `IPCFG_STATE_DHCP_IP`
- `IPCFG_STATE_AUTO_IP`
- `IPCFG_STATE_DHCPAUTO_IP`
- `IPCFG_STATE_BOOT`

### See Also

- [ipcfg\\_get\\_state\\_string\(\)](#)
- [ipcfg\\_get\\_state\(\)](#)

### 7.1.73 ipcfg\_get\_link\_active()

Returns immediate Ethernet link state.

#### Synopsis

```
boolean ipcfg_get_link_active
      uint_32 device
```

#### Parameters

*device [in]* — device identification

#### Description

This function returns the immediate Ethernet link status of a given device.

#### Return Value

*TRUE* if link active, *FALSE* otherwise

#### See Also

- [ipcfg\\_get\\_state\\_string\(\)](#)
- [ipcfg\\_get\\_state\(\)](#)
- [ipcfg\\_get\\_desired\\_state\(\)](#)

## 7.1.74 ipcfg\_get\_dns\_ip()

Returns the  $n$ -th DNS IP address from the registered DNS list.

### Synopsis

```
_ip_address ipcfg_get_dns_ip(  
    uint_32 device,  
    uint_32 n)
```

### Parameters

*device* [in] — device identification

*n* [in] — DNS IP address index

### Description

This function may be used to retrieve all DNS addresses registered (manually or by DHCP binding process) with the given Ethernet device.

### Return Value

DNS IP address. Zero if  $n$ -th address is not available.

### See Also

- [ipcfg\\_add\\_dns\\_ip\(\)](#)
- [ipcfg\\_del\\_dns\\_ip\(\)](#)



### 7.1.75 ipcfg\_add\_dns\_ip()

Registers the DNS IP address with the Etherent device.

#### Synopsis

```
boolean ipcfg_add_dns_ip (  
    uint_32 device,  
    _ip_address address)
```

#### Parameters

*device* [in] — device identification

*address* [in] — DNS IP address to add

#### Description

This function adds the DNS IP address to the list assigned to given Ethernet device and starts the DNS machine, if not running already.

#### Return Value

*TRUE* if successful, *FALSE* otherwise

#### See Also

- [ipcfg\\_get\\_dns\\_ip\(\)](#)
- [ipcfg\\_del\\_dns\\_ip\(\)](#)

## 7.1.76 ipcfg\_del\_dns\_ip()

Unregisters the DNS IP address.

### Synopsis

```
boolean ipcfg_del_dns_ip (  
    uint_32 device,  
    _ip_address address)
```

### Parameters

*device* [in] — device identification

*address* [in] — DNS IP address to be removed

### Description

This function removes the DNS IP address from the list assigned to given Ethernet device.

### Return Value

*TRUE* if successful, *FALSE* otherwise

### See Also

- [ipcfg\\_get\\_dns\\_ip\(\)](#)
- [ipcfg\\_add\\_dns\\_ip\(\)](#)

### 7.1.77 ipcfg\_get\_ip()

Returns an immediate IP address information bound to Ethernet device.

#### Synopsis

```
boolean ipcfg_get_ip(  
    uint_32 device,  
    IPCFG_IP_ADDRESS_DATA_PTR data)
```

#### Parameters

*device* [in] — device identification

*data* [in] — pointer to IP address information (IP address, mask and gateway)

#### Description

This function returns the immediate IP address information bound to given Ethernet device.

#### Return Value

TRUE if successful and *data* structure filled. FALSE in case of error.

#### See Also

- [ipcfg\\_get\\_dns\\_ip\(\)](#)

### 7.1.78 ipcfg\_get\_tftp\_serveraddress()

Returns TFTP server address, if any.

#### Synopsis

```
_ip_address ipcfg_get_tftp_serveraddress(  
    uint_32 device)
```

#### Parameters

*device [in]* — device identification

#### Description

This function returns the last TFTP server address if such was assigned by the last BOOTP bind process.

#### Return Value

The TFTP server IP address.

#### See Also

- [ipcfg\\_get\\_tftp\\_servername\(\)](#)
- [ipcfg\\_get\\_boot\\_filename\(\)](#)

### 7.1.79 ipcfg\_get\_tftp\_servername()

Returns TFTP servername, if any.

#### Synopsis

```
uchar_ptr ipcfg_get_tftp_serveraddress(uint_32 device)
```

#### Parameters

*device [in]* — device identification

#### Description

This function returns the last TFTP server name if such was assigned by the last DHCP or BOOTP bind process.

#### Return Value

Pointer to server name string.

#### See Also

- [ipcfg\\_get\\_tftp\\_serveraddress\(\)](#)
- [ipcfg\\_get\\_boot\\_filename\(\)](#)

## 7.1.80 ipcfg\_get\_boot\_filename()

Returns the TFTP boot filename, if any.

### Synopsis

```
uchar_ptr ipcfg_get_boot_filename(uint_32 device)
```

### Parameters

*device [in]* — device identification

### Description

This function returns the last boot file name if such was assigned by the last DHCP or BOOTP bind process.

### Return Value

Pointer to boot filename string.

### See Also

- [ipcfg\\_get\\_tftp\\_serveraddress\(\)](#)
- [ipcfg\\_get\\_tftp\\_servername\(\)](#)

## 7.1.81 ipcfg\_poll\_dhcp()

Polls (finishes) the Ethernet device DHCP binding process.

### Synopsis

```
uint_32 ipcfg_poll_dhcp(
    uint_32 device,
    boolean try_auto_ip,
    IPCFG_IP_ADDRESS_DATA_PTR auto_ip_data)
```

### Parameters

*device [in]* — device identification

*try\_auto\_ip [in]* — try the auto-ip automatic assign address if DHCP binding fails

*auto\_ip\_data [in]* — ip, mask and gateway address information to be used if DHCP bind fails

### Description

See [ipcfg\\_bind\\_dhcp\(\)](#).

### Return Value

- *IPCFG\_OK* (success)
- *RTCSERR\_IPCFG\_BUSY*
- *RTCSERR\_IPCFG\_DEVICE\_NUMBER*
- *RTCSERR\_IPCFG\_INIT*
- *RTCSERR\_IPCFG\_BIND*

### See Also

- [ipcfg\\_bind\\_dhcp\(\)](#)

### Example

## 7.1.82 ipcfg\_task\_create()

Creates and starts the IPCFG Ethernet link status-monitoring task.

### Synopsis

```
uint_32 ipcfg_task_create(
    uint_32 priority,
    uint_32 task_period_ms)
```

### Parameters

*priority* [in] — task priority

*task\_period\_ms* [in] — task polling period in milliseconds

### Description

Link status task periodically checks Ethernet link status of each initialized Ethernet device. If the link is lost, the task automatically unbinds the interface. When the link goes on again, the task tries to bind the interface to network using information from last successful bind operation.

If the device was unbound by calling [ipcfg\\_unbind\(\)](#), the task leaves the interface in unbound state.

An alternative way to monitor the Ethernet link status (without a separate task) is to call [ipcfg\\_task\\_poll\(\)](#) periodically in the user's task.

### Return Value

- *MQX\_OK* (success)
- *MQX\_DUPLICATE\_TASK\_TEMPLATE\_INDEX*
- *MQX\_INVALID\_TASK\_ID*

### See Also

- [ipcfg\\_task\\_destroy\(\)](#)
- [ipcfg\\_task\\_status\(\)](#)
- [ipcfg\\_task\\_poll\(\)](#)

### Example

```
void main(uint_32 param)
{
    setup_network();
    ipcfg_task_create(8, 1000);
    if (! ipcfg_task_stats()) _task_block();

    ...

    ipcfg_task_destroy(TRUE);
    while (1)
    {
        _time_delay(1000);
        ipcfg_task_poll();
    }
}
```



### 7.1.83 ipcfg\_task\_destroy()

Signals the exit request to the IPCFG task.

#### Synopsis

```
void ipcfg_task_destroy(  
    boolean wait_task_finish)
```

#### Parameters

*wait\_task\_finish [in]* — wait for task exit if TRUE

#### Description

This functions sets an internal flag which is checked during each pass of Ethernet link status monitoring task. The task exits as soon as it completes the immediate operation.

According to parameter this function may wait for task destruction.

#### Return Value

none

#### See Also

- [ipcfg\\_task\\_create\(\)](#)
- [ipcfg\\_task\\_status\(\)](#)
- [ipcfg\\_task\\_poll\(\)](#)

#### Example

See [ipcfg\\_task\\_create\(\)](#).

## 7.1.84 **ipcfg\_task\_status()**

Checks whether the IPCFG Ethernet link status monitorin task is running.

### Synopsis

```
boolean ipcfg_task_status(void)
```

### Description

This function returns TRUE if link status monitoring task is currently running, returns FALSE otherwise.

### Return Value

TRUE if task is running.

FALSE if task is not running.

### See Also

- [ipcfg\\_task\\_create\(\)](#)
- [ipcfg\\_task\\_destroy\(\)](#)
- [ipcfg\\_task\\_poll\(\)](#)

### Example

See [ipcfg\\_task\\_create\(\)](#).

## 7.1.85 ipcfg\_task\_poll()

One step of the IPCFG Ethernet link status monitoring task.

### Synopsis

```
boolean ipcfg_task_poll(void)
```

### Description

This function executes one step of the link status monitoring task. This function may be called periodically in any user's task to emulate the task operation. The task itself doesn't need to be created in this case.

### Return Value

*TRUE* if the immediate bind process finished (stable state).

*FALSE* if task is in the middle of bind operation (function should be called again).

### See Also

- [ipcfg\\_task\\_create\(\)](#)
- [ipcfg\\_task\\_destroy\(\)](#)
- [ipcfg\\_task\\_status\(\)](#)

### Example

See [ipcfg\\_task\\_create\(\)](#).

## 7.1.86 ipcfg\_unbind()

Unbinds the Ethernet device from network.

### Synopsis

```
uint_32 ipcfg_unbind(  
    uint_32 device)
```

### Parameters

*device [in]* — device identification

### Description

This function releases the IP address information bound to a given device. It is blocking function, i.e. doesn't return until the process is finished or error occurs.

### Return Value

- *IPCFG\_OK (success)*
- *RTCSERR\_IPCFG\_BUSY*
- *RTCSERR\_IPCFG\_DEVICE\_NUMBER*
- *RTCSERR\_IPCFG\_INIT*

### See Also

- [ipcfg\\_bind\\_staticip\(\)](#)
- [ipcfg\\_bind\\_dhcp\(\)](#)

### Example

```
void main(uint_32 param)  
{  
    setup_network();  
  
    ...  
  
    ipcfg_unbind();  
    while (1) {};  
}
```

## 7.1.87 iwcfg\_set\_essid()

### Synopsis

```
uint_32 iwcfg_set_essid
(
    uint_32 dev_num,
    char_ptr essid
)
```

### Parameters

*dev\_num [in]* — Device identification (index).

*essid [in]* — Pointer to ESSID (Extended Service Set Identifier) string.

### Description

This function sets to device identified IP interface structure ESSID. Device must be initialized before. ESSID comes into effect only when user commits his changes. The ESSID is used to identify cells which are part of the same virtual network.

### Return Value

- ENET\_OK (success)
- ENET\_ERROR
- ENETERR\_INVALID\_DEVICE

### Example

```
#define SSID                "NGZG"
#define DEFAULT_DEVICE     1
int_32                     error;

/* IP configuration */
error = RTCS_create();
ENET_get_mac_address (DEFAULT_DEVICE, ENET_IPADDR, enet_address);
error = ipcfig_init_device (DEFAULT_DEVICE, enet_address);
/* Set SSID */
iwcfg_set_essid (DEFAULT_DEVICE, SSID);
iwcfg_commit( DEFAULT_DEVICE );
/* end of IP configuration */
error = ipcfig_bind_staticip (DEFAULT_DEVICE, &ip_data);
```

## 7.1.88 iwcfg\_get\_essid()

### Synopsis

```
uint_32 iwcfg_get_essid
(
    uint_32 dev_num,
    char_ptr essid
)
```

### Parameters

*dev\_num [in]* — Device identification (index).

*essid [out]* — Extended Service Set Identifier string.

### Description

This function returns ESSID for selected device.

### Return Value

- ENET\_OK (success)
- ENET\_ERROR
- ENETERR\_INVALID\_DEVICE

### Example

```
#define DEFAULT_DEVICE 1
char[20] ssid_name;

iwcfg_get_essid (DEFAULT_DEVICE, &ssid_name);
```

## 7.1.89 iwcfg\_commit()

### Synopsis

```
uint_32 iwcfg_commit
(
    uint_32 dev_num
)
```

### Parameters

*dev\_num [in]* — Device identification (index).

### Description

Commits the requested change. Some cards may not apply changes done immediately (they may wait to aggregate the changes). This command forces the card to apply all pending changes.

### Return Value

- ENET\_OK (success)
- ENETERR\_INVALID\_DEVICE
- Other device specific errors

### Example

```
#define SSID                "NGZG"
#define DEFAULT_DEVICE 1

/* initialize rtcs before */
iwcfg_set_essid (DEFAULT_DEVICE, SSID);
iwcfg_commit (DEFAULT_DEVICE);
```

## 7.1.90 iwcfg\_set\_mode()

### Synopsis

```
uint_32 iwcfg_set_mode
(
    uint_32 dev_num,
    char_ptr mode
)
```

### Parameters

*dev\_num [in]* — Device identification (index).

*mode [in]* — Wifi device mode, accepted values are "managed" and "adhoc".

### Description

Set the operating mode of the device, which depends on the network topology. The mode can be Ad-Hoc (network composed of only one cell and without Access Point) or Managed (node connects to a network composed of many Access Points, with roaming).

### Return Value

- ENET\_OK (success)
- ENETERR\_INVALID\_DEVICE
- Other device specific errors

### Example

```
#define DEMOCFG_SECURITY "none"
#define DEMOCFG_SSID     "NGZG"
#define DEMOCFG_NW_MODE  "managed"
#define DEFAULT_DEVICE   1
```

```
error = RTCS_create();
```

```
ip_data.ip = ENET_IPADDR;
ip_data.mask = ENET_IPMASK;
ip_data.gateway = ENET_IPGATEWAY;
```

```
ENET_get_mac_address (DEFAULT_DEVICE, ENET_IPADDR, enet_address); error = ipcfg_init_device
(DEFAULT_DEVICE, enet_address);
iwcfg_set_essid (DEFAULT_DEVICE, DEMOCFG_SSID );
iwcfg_set_sec_type (DEFAULT_DEVICE, DEMOCFG_SECURITY);
iwcfg_set_mode (DEFAULT_DEVICE, DEMOCFG_NW_MODE);
error = ipcfg_bind_staticip (DEFAULT_DEVICE, &ip_data);
```



## 7.1.91 iwcfg\_get\_mode()

### Synopsis

```
uint_32 iwcfg_get_mode
(
    uint_32 dev_num
    char_ptr mode
)
```

### Parameters

*dev\_num [in]* — Device identification (index).

*mode [out]* — Current wifi mode (string).

### Description

Return current wifi module mode. Possible values are "managed" or "adhoc".

### Return Value

- ENET\_OK (success)
- ENETERR\_INVALID\_DEVICE

### Example

```
#define DEFAULT_DEVICE 1
char[20] ssid_name;

iwcfg_get_mode (DEFAULT_DEVICE, &ssid_name);
```

## 7.1.92 iwcfg\_set\_wep\_key()

### Synopsis

```
uint_32 iwcfg_set_wep_key
(
    uint_32 dev_num,
    char_ptr wep_key,
    uint_32 key_len,
    uint_32 key_index
)
```

### Parameters

*dev\_num [in]* — Device identification (index).

*wep\_key [in]* — Wep\_key.

*key\_len [in]* — Length of the key.

*key\_index [in]* — Additional optional device specific parameters. Index must be lower than 256.

### Description

Set wep key to wifi device.

### Return Value

- ENET\_OK (success)
- ENETERR\_INVALID\_DEVICE

### Example

```
iwcfg_set_wep_key (DEFAULT_DEVICE, DEMOCFG_WEP_KEY, strlen(DEMOCFG_WEP_KEY),
DEMOCFG_WEP_KEY_INDEX);
```

## 7.1.93 iwcfg\_get\_wep\_key()

### Synopsis

```
uint_32 iwcfg_get_wep_key
(
    uint_32 dev_num,
    char_ptr wep_key,
    uint_32 key_index
)
```

### Parameters

*dev\_num [in]* — Device identification (index).

*wep\_key [in]* — Wep\_key.

*key\_index [in]* — Additional optional device specific parameters. Index must be lower than 256.

### Description

Get the wep key.

### Return Value

- ENET\_OK (success)
- ENETERR\_INVALID\_DEVICE

## 7.1.94 iwcfg\_set\_passphrase()

### Synopsis

```
uint_32 iwcfg_set_passphrase
(
    uint_32 dev_num,
    char_ptr passphrase
)
```

### Parameters

*dev\_num [in]* — Device identification (index).

*passphrase [in]* — SSID passphrase.

### Description

Set wpa passphrase.

### Return Value

- ENET\_OK (success)
- ENETERR\_INVALID\_DEVICE

### Example

```
#define DEMOCFG_SECURITY "wpa"
#define DEMOCFG_SSID     "NGZG"
#define DEMOCFG_NW_MODE  "managed"
#define DEMOCFG_PASSPHRASE "abcdefgh"
#define DEFAULT_DEVICE   1

error = RTCS_create();

ip_data.ip = ENET_IPADDR;
ip_data.mask = ENET_IPMASK;
ip_data.gateway = ENET_IPGATEWAY;

ENET_get_mac_address (DEFAULT_DEVICE, ENET_IPADDR, enet_address) error = ipcfg_init_device
(DEFAULT_DEVICE, enet_address);
iwcfg_set_essid (DEFAULT_DEVICE, DEMOCFG_SSID);
iwcfg_set_passphrase (DEFAULT_DEVICE, DEMOCFG_PASSPHRASE);
iwcfg_set_sec_type (DEFAULT_DEVICE, DEMOCFG_SECURITY);
iwcfg_set_mode (DEFAULT_DEVICE, DEMOCFG_NW_MODE);
error = ipcfg_bind_staticip (DEFAULT_DEVICE, &ip_data);
```

## 7.1.95 iwcfg\_get\_passphrase()

### Synopsis

```
uint_32 iwcfg_get_passphrase
(
    uint_32 dev_num,
    char_ptr passphrase
)
```

### Parameters

*dev\_num [in]* — Device identification (index).

*passphrase [out]* — SSID passphrase (string).

### Description

Get the wpa passphrase from initialized wifi device.

### Return Value

- ENET\_OK (success)
- ENETERR\_INVALID\_DEVICE

## 7.1.96 iwcfg\_set\_sec\_type()

### Synopsis

```
uint_32 iwcfg_set_sec_type
(
    uint_32 dev_num,
    char_ptr sec_type
)
```

### Parameters

*dev\_num* [in] — Device identification (index).

*sec\_type* [in] — Security type. Accepted values are "none", "wep", "wpa", "wpa2".

### Description

Set security type to device.

### Return Value

- ENET\_OK (success)
- ENETERR\_INVALID\_DEVICE

### Example

See the iwcfg\_set\_passphrase example.

## 7.1.97 iwcfg\_get\_sectype()

### Synopsis

```
uint_32 iwcfg_get_sec_type
(
    uint_32 dev_num,
    char_ptr sec_type
)
```

### Parameters

*dev\_num* [in] — Device identification (index).

*sec\_type* [out] — Security type (string).

### Description

Get security type from device. Possible values are "none", "wep", "wpa", "wpa2".

### Return Value

- ENET\_OK (success)
- ENETERR\_INVALID\_DEVICE

## 7.1.98 iwcfg\_set\_power()

### Synopsis

```
uint_32 iwcfg_set_power
(
    uint_32 dev_num,
    uint_32 pow_val,
    uint_32 flags
)
```

### Parameters

*dev\_num [in]* — Device identification (index).

*pow\_val [in]* — Power in dBm.

*flags [in]* — Device specific options.

### Description

For cards supporting multiple transmit powers, sets the transmit power in dBm. If W is the power in Watt, the power in dBm is  $P = 30 + 10 \cdot \log(W)$ . In addition, on and off enable and disable the radio, and auto and fixed enable and disable power control (if those features are available).

### Return Value

- ENET\_OK (success)
- ENETERR\_INVALID\_DEVICE



## 7.1.99 iwcfg\_set\_scan()

### Synopsis

```
uint_32 iwcfg_set_scan
(
    uint_32 dev_num,
    char_ptr ssid
)
```

### Parameters

*dev\_num [in]* — Device identification (index).

*ssid [in]* — Not used yet.

### Description

This will find all available networks and print them in format. The format is wifi vendor dependent.

ssid = tplink - SSID name

bssid = 94:c:6d:a5:51:b - SSID's MAC address

channel = 1 - channel

strength = ##### - signal strength in graphics

indicator = 183 - signal strength

### Return Value

- ENET\_OK (success)
- ENETERR\_INVALID\_DEVICE

### Example

```
#define SSID                "NGZG"
int_32                      error;

/* IP configuration */
error = RTCS_create();
ENET_get_mac_address (DEFAULT_DEVICE, ENET_IPADDR, enet_address);
error = ipcfig_init_device (DEFAULT_DEVICE, ENET_IPADDR);

/* scan for networks */
iwcfg_set_scan (DEFAULT_DEVICE, NULL);
```

#### Example output:

```
ssid = tplink
bssid = 94:c:6d:a5:51:b
channel = 1
strength = #####
indicator = 183
```

## Function Reference

```
ssid = Faz  
bssid = 0:21:91:12:da:cc  
channel = 1  
strength = ####.  
indicator = 172  
---  
  
scan done.
```

## 7.1.100 listen()

Puts the stream socket into the listening state.

### Synopsis

```
uint_32  listen(  
    uint_32  socket,  
    uint_16  backlog)
```

### Parameters

*socket* [in] — Socket handle

*backlog* [in] — Ignored

### Description

Putting the stream into the listening state allows incoming connection requests from remote endpoints. After the application calls **listen()**, it should call **accept()** to attach new sockets to the incoming requests.

This function blocks, but the command is immediately serviced and replied to.

### Return Value

- *RTCS\_OK* (success)
- Specific error code (failure)

### See Also

- [accept\(\)](#)
- [bind\(\)](#)
- [socket\(\)](#)

### Example

See [accept\(\)](#).

## 7.1.101 MIB1213\_init()

Initializes the MIB-1213.

### Synopsis

```
void MIB1213_init(void)
```

### Description

The function installs the standard MIBs defined in RFC 1213. If the function is not called, SNMP Agent cannot access the MIB.

### See Also

- [SNMP\\_init\(\)](#)

### Example

See [SNMP\\_init\(\)](#).

## 7.1.102 MIB\_find\_objectname()

Find object in table.

### Synopsis

```
boolean MIB_find_objectname(uint_32 op, pointer index, pointer _PTR_ instance)
```

### Parameters

op [in]

*index [in]* — Pointer to a structure that contains the table index.

instance [out]

### Description

For each variable object that is in a table, you must provide `MIB_find_objectname()`, where `objectname` is the name of the variable object. The function gets an instance pointer.

### Return Value

- *SNMP\_ERROR\_noError* (success)
- *SNMP\_ERROR\_wrongValue*
- *SNMP\_ERROR\_inconsistentValue*
- *SNMP\_ERROR\_wrongLength*
- *SNMP\_ERROR\_resourceUnavailable*
- *SNMP\_ERROR\_genErr*

### See Also

- [SNMP\\_init\(\)](#)
- [MIB1213\\_init\(\)](#)

### Example

### 7.1.103 MIB\_set\_objectname()

Set name for writable object in table.

#### Synopsis

```
uint_32 MIB_set_objectname(pointer instance, uchar_ptr value_ptr, uint_32 value_len)
```

#### Parameters

*instance* [in]

*value\_ptr* [out] — Pointer to the value to which to set objectname.

*value\_len* [out] — Length in bytes of the value.

#### Description

For each writable variable object, you must provide MIB\_set\_objectname(), where objectname is the name of the variable object.

#### See Also

- [SNMP\\_init\(\)](#)
- [MIB1213\\_init\(\)](#)
- [MIB\\_find\\_objectname\(\)](#)

#### Example

### 7.1.104 NAT\_close()

Stops Network Address Translation.

#### Synopsis

```
uint_32 NAT_close(void)
```

#### Return Value

- *RTCS\_OK* (success)

#### See Also

- [NAT\\_init\(\)](#)

## 7.1.105 NAT\_init()

Starts Network Address Translation.

### Synopsis

```
uint_32 NAT_init(  
    _ip_address  prv_network,  
    _ip_address  prv_netmask)
```

### Parameters

*prv\_network* [in] — Private-network address

*prv\_netmask* [in] — Private-network subnet mask

### Description

Freescall MQX NAT starts working only when network address translation has started (by a call to **NAT\_init()**) and the *\_IP\_forward* global running parameter is TRUE.

Function **NAT\_init()** enables all the application-level gateways that are defined in the *NAT\_alg\_table*. For more information, see [Section 2.15.3, “Disabling NAT Application-Level Gateways.”](#)

You can use this function to restart Network Address Translation after you call **NAT\_close()**.

### Return Value

- *RTCS\_OK* (success)
- *RTCSERR\_OUT\_OF\_MEMORY* (failure)
- *RTCSERR\_INVALID\_PARAMETER* (failure)

### See Also

- [NAT\\_close\(\)](#)
- [NAT\\_stats\(\)](#)
- *nat\_ports*
- *nat\_timeouts*
- *NAT\_STATS*



## 7.1.106 NAT\_stats()

Gets Network Address Translation statistics.

### Synopsis

```
NAT_STATS_PTR NAT_stats(void)
```

### Return Value

- Pointer to the *NAT\_STATS* structure (success)
- NULL (failure: **NAT\_init()** has not been called)

### See Also

- [NAT\\_init\(\)](#)
- [NAT\\_STATS](#)

## 7.1.107 ping()

See [RTCS\\_ping\(\)](#).

## 7.1.108 PPP\_initialize()

Initializes PPP Driver for the PPP link.

### Synopsis

```
uint_32  PPP_initialize(
    _iopcb_handle  device,
    _ppp_handle _PTR_ ppp_handle)
```

### Parameters

*device [in]* — I/O stream to use

*ppp\_handle [out]* — Pointer to the PPP handle

### Description

Function **PPP\_initialize()** fails, if RTCS cannot do any one of the following:

- Allocate memory for the PPP state structure or initialize a lightweight semaphore to protect it.
- Initialize LCP or CCP.
- Allocate a pool of message buffers.
- Create the PPP send and receive tasks.

### Return Value

- *PPP\_OK* (success)
- Error code (failure)

### See Also

- [\\_iopcb\\_handle](#), [\\_iopcb\\_table](#)

### Example

See [Section 2.15.6](#), “Example: Setting Up RTCS.”

## 7.1.109 recv()

Provides RTCS with incoming buffer.

### 7.1.109.1 Synopsis

```
int_32  recv(
    uint_32      socket,
    char*_PTR    buffer,
    uint_32      buflen,
    uint_32      flags
)
```

#### Parameters

*socket [in]* — Handle for the connected stream socket.

*buffer [out]* — Pointer to the buffer, in which to place received data.

*buflen [in]* — Size of buffer in bytes.

*flags [in]* — Flags to underlying protocols. One of the following:

*RTCS\_MSG\_PEEK* — for a UDP socket, receives a datagram but does not consume it (ignored for stream sockets).

Zero — ignore.

#### Description

Function **recv()** provides RTCS with a buffer for data incoming on a stream or datagram socket.

When the *flags* parameter is *RTCS\_MSG\_PEEK*, the same datagram is received the next time **recv()** or **recvfrom()** is called.

If the function returns **RTCS\_ERROR**, the application can call **RTCS\_geterror()** to determine the reason for the error.

NOTE	If the peer gracefully closed the connection, <b>recv()</b> returns <i>RTCS_ERROR</i> , rather than zero as BSD 4.4 specifies. A subsequent call to <b>RTCS_geterror()</b> returns <i>RTCSERR_TCP_CONN_CLOSING</i> .
------	--

#### Stream Socket

If the receive-nowait socket option is TRUE, RTCS immediately copies internally buffered data (up to *buflen* bytes) into the buffer (at *buffer*), and **recv()** returns. If the receive-wait socket option is TRUE, **recv()** blocks, until the buffer is full or the receive-push socket option is satisfied.

If the receive-push socket option is TRUE, a received TCP push flag causes **recv()** to return with whatever data has been received. If the receive-push socket option is FALSE, RTCS ignores incoming TCP push flags, and **recv()** returns when enough data has been received to fill the buffer.

#### Datagram Socket

The **recv()** function on a datagram socket is identical to **recvfrom()** with NULL *fromaddr* and *fromlen* pointers. The **recv()** function is normally used on a connected socket.

## Stream Socket

```
uint_32  handle;
char     buffer[20000];
uint_32  count;

...
count = recv(handle, buffer, 20000, 0);
if (count == RTCS_ERROR)
{
    printf("\nError, recv() failed with error code %lx",
           RTCS_geterror(handle));
} else {
    printf("\nReceived %ld bytes of data.", count);
}
```

## 7.1.110 recvfrom()

Provides RTCS with the buffer, in which to place data that is incoming on the datagram socket.

### Synopsis

```
int_32 recvfrom(
    uint_32          socket,
    char *_PTR_      buffer,
    uint_32          buflen,
    uint_32          flags,
    sockaddr_in *_PTR_ fromaddr,
    uint_16_ptr      fromlen)
```

### Parameters

*socket* [in] — Handle for the datagram socket.

*buffer* [out] — Pointer to the buffer, in which to place received data.

*buflen* [in] — Size of buffer in bytes.

*flags* [in] — Flags to underlying protocols. One of the following:

*RTCS\_MSG\_PEEK* — receives a datagram but does not consume it.

Zero — ignore.

*fromaddr* [out] — Source socket address of the message.

*fromlen* [in/out] — When passed in: Size of the *fromaddr* buffer.

When passed out: Size of the socket address stored in the *fromaddr* buffer, or, if the provided buffer was too small (socket-address was truncated), the length before truncation.

### Description

If a remote endpoint has been specified with **connect()**, only datagrams from that source will be received.

When the *flags* parameter is **RTCS\_MSG\_PEEK**, the same datagram is received the next time **recv()** or **recvfrom()** is called.

If *fromlen* is NULL, the socket address is not written to *fromaddr*. If *fromaddr* is NULL and the value of *fromlen* is not NULL, the result is unspecified.

If the function returns *RTCS\_ERROR*, the application can call **RTCS\_geterror()** to determine the reason for the error.

This function blocks until data is available or an error occurs.

### Return Value

- Number of bytes received (success)
- *RTCS\_ERROR* (failure)

### See Also

- [bind\(\)](#)
- [RTCS\\_geterror\(\)](#)
- [sendto\(\)](#)

- **socket()**

### Example

Receive up to 500 bytes of data.

```
uint_32      handle;
sockaddr_in  remote_sin;
uint_32      count;
char         my_buffer[500];
uint_16      remote_len = sizeof(remote_sin);

...

count = recvfrom(handle, my_buffer, 500, 0,
                  &remote_sin, &remote_len);

if (count == RTCS_ERROR)
{
    printf("\nrecvfrom() failed with error %lx",
          RTCS_geterror(handle));
} else {
    printf("\nReceived %ld bytes of data.", count);
}
```

## 7.1.111 RTCS\_attachsock()

Takes ownership of the socket.

### Synopsis

```
uint_32 RTCS_attachsock(
    uint_32  socket)
```

### Parameters

*socket [in]* — Socket handle

### Description

The function adds the calling task to the socket's list of owners.

This function blocks, although the command is serviced and responded to immediately.

### Return Value

- New socket handle (success)
- *RTCS\_SOCKET\_ERROR* (failure)

### See Also

- [accept\(\)](#)
- [RTCS\\_detachsock\(\)](#)

### Example

A main task loops to accept connections. When it accepts a connection, it creates a child task to manage the connection: it relinquishes control of the socket by calling **RTCS\_detachsock()**, and then creates the child with the accepted socket handle as the initial parameter.

```
while (TRUE) {
    /* Issue ACCEPT: */
    TELNET_accept_skt =
        accept(TELNET_listen_skt, &peer_addr, &addr_len);
    if (TELNET_accept_skt != RTCS_SOCKET_ERROR) {
        /* Transfer the socket and create the child task to look after
           the socket: */
        if (RTCS_detachsock(TELNET_accept_skt) == RTCS_OK) {
            child_task = (_task_create(LOCAL_ID, CHILD, TELNET_accept_skt);
        } else {
            printf("\naccept() failed, error
                0x%lx", RTCS_geterror(TELNET_accept_skt));
        }
    }
}
```

The child attaches itself to the socket for which the main task transferred ownership.

```
void TELNET_Child_task
(
    uint_32  socket_handle
)
{
    /* Attach the socket to this task: */
    printf("\nCHILD - about to attach the socket.");
    socket_handle = RTCS_attachsock(socket_handle);
}
```



```
if (socket_handle != RTCS_SOCKET_ERROR) {  
    /* Continue managing the socket. */  
} else {  
    ...  
}
```

## 7.1.112 RTCS\_create()

Creates RTCS.

### Synopsis

```
uint_32 RTCS_create(void)
```

### Description

This function allocates resources that RTCS needs and creates TCP/IP task.

### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

### See Also

- [RTCS\\_if\\_add\(\)](#)
- [RTCS\\_if\\_bind\(\)](#)

### Example

See [Section 2.15.6](#), “Example: Setting Up RTCS.”

### 7.1.113 RTCS\_detachsock()

Relinquishes ownership of the socket.

#### Synopsis

```
uint_32  RTCS_detachsock(  
    uint_32  socket)
```

#### Parameters

*socket* [in] — Socket handle

#### Description

The function removes the calling task from the socket's list of owners.

Parameter *socket* is returned by one of the following:

- **socket()**
- **accept()**
- **RTCS\_attachsock()**

This function blocks, although the command is serviced and responded to immediately.

#### Return Value

- *RTCS\_OK* (success)
- Specific error code (failure)

#### See Also

- [accept\(\)](#)
- [RTCS\\_attachsock\(\)](#)
- [socket\(\)](#)

#### Example

See [RTCS\\_attachsock\(\)](#).

## 7.1.114 RTCS\_exec\_TFTP\_BIN()

Download and run the binary boot file.

### Synopsis

```
uint_32 RTCS_exec_TFTP_BIN(
    _ip_address  server,
    char_ptr     filename,
    uchar_ptr    download_address,
    uchar_ptr    run_address)
```

### Parameters

*server* [in] — IP address of the TFTP Server, from which to get the file.

*filename* [in] — Name of the file to download.

*download\_address* [in] — Address, to which to download the file.

*run\_address* [in] — Address, at which to start to run the file.

### Description

This function downloads the binary file from the TFTP Server and runs the file. This function does not return if it succeeds.

You can usually find the *server* and *filename* in the structure fields shown in [Table 7-1](#):

**Table 7-1. Boot File Server and File Names**

Operation	Function	Fields	Structure
BootP	<b>RTCS_if_bind_BOOTP()</b>	<ul style="list-style-type: none"> <li>SADDR</li> <li>BOOTFILE</li> </ul>	<i>BOOTP_DATA_STRUCT</i>
DHCP	<b>RTCS_if_bind_DHCP()</b>	<ul style="list-style-type: none"> <li>SADDR</li> <li>FILE</li> </ul>	<i>DHCPSRV_DATA_STRUCT</i>

### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

### See Also

- [RTCS\\_create\(\)](#)
- [RTCS\\_exec\\_TFTP\\_COFF\(\)](#)
- [RTCS\\_exec\\_TFTP\\_SREC\(\)](#)
- [RTCS\\_load\\_TFTP\\_BIN\(\)](#)
- [BOOTP\\_DATA\\_STRUCT](#)

## Example

Initialize RTCS using BootP, download the binary boot file, and run it.

```
uint_32 boot_function(void) {
    BOOTP_DATA_STRUCT boot_data;
    _enet_handle      ehandle;
    _rtcs_if_handle    ihandle;
    uint_32            error;

    error = ENET_initialize(0, enet_local, 0, &ehandle);
    if (error) return error;
    error = RTCS_create();
    if (error) return error;
    error = RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);
    if (error) return error;

    memset(&boot_data, 0, sizeof(boot_data));
    error = RTCS_if_bind_BOOTP(ihandle, &boot_data);
    if (error) return error;

    printf("\nDownloading the boot file...\n");

    error = RTCS_exec_TFTP_BIN(boot_data.SADDR,
                              (char_ptr)boot_data.BOOTFILE,
                              (uchar_ptr)DOWNLOAD_ADDR,
                              (uchar_ptr)RUN_ADDR);

    return error;
}
```

## 7.1.115 RTCS\_exec\_TFTP\_COFF()

Downloads and runs the COFF boot file.

### Synopsis

```
uint_32  RTCS_exec_TFTP_COFF(  
    _ip_address  server,  
    char_ptr     filename)
```

### Description

The function downloads the COFF file from the TFTP Server, decodes the file, and runs it.

For information on the values of *server* and *filename*, see [Table 7-1](#).

### Parameters

*server* [in] — IP address of the TFTP Server, from which to get the file.

*filename* [in] — Name of the file to download.

### Return Value

- Nothing (*RTCS\_OK*) on success
- Error code on failure

### See Also

- [RTCS\\_create\(\)](#)
- [RTCS\\_exec\\_TFTP\\_BIN\(\)](#)
- [RTCS\\_exec\\_TFTP\\_SREC\(\)](#)
- [BOOTP\\_DATA\\_STRUCT](#)

## 7.1.116 RTCS\_exec\_TFTP\_SREC()

Downloads and runs the S-Record boot file.

### Synopsis

```
uint_32 RTCS_exec_TFTP_SREC(
    _ip_address  server,
    char_ptr     filename)
```

### Description

This function downloads the Motorola S-Record file from the TFTP Server, decodes the file, and runs it.

For information on the values of *server* and *filename*, see [Table 7-1](#).

### Parameters

*server* [in] — IP address of the TFTP server, from which to get the file.

*filename* [in] — Name of the file to download.

### Return Value

- Nothing (*RTCS\_OK*) on success
- Error code on failure

### See Also

- [RTCS\\_create\(\)](#)
- [RTCS\\_exec\\_TFTP\\_BIN\(\)](#)
- [RTCS\\_exec\\_TFTP\\_COFF\(\)](#)
- [BOOTP\\_DATA\\_STRUCT](#)

### Example

Initialize RTCS using BootP, download the S-Record file, and run it.

```
uint_32 boot_function(void)
{
    BOOTP_DATA_STRUCT  boot_data;
    _enet_handle        ehandle;
    _rtcs_if_handle     ihandle;
    uint_32             error;

    error = ENET_initialize(0, enet_local, 0, &ehandle);
    if (error) return error;

    error = RTCS_create();
    if (error) return error;

    error = RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);
    if (error) return error;

    memset(&boot_data, 0, sizeof(boot_data));
    error = RTCS_if_bind_BOOTP(ihandle, &boot_data);
    if (error) return error;

    printf("\nDownloading the boot file...\n");
```

```
error = RTCS_exec_TFTP_SREC (boot_data.SADDR,  
                             (char_ptr)boot_data.BOOTFILE);  
return error;  
}
```



### 7.1.117 RTCS\_gate\_add()

Adds the gateway to RTCS.

#### Synopsis

```
uint_32 RTCS_gate_add(
    _ip_address gateway,
    _ip_address network,
    _ip_address netmask)
```

#### Parameters

*gateway* [in] — IP address of the gateway.

*network* [in] — IP network, in which the gateway is located.

*netmask* [in] — Network mask for *network*.

#### Description

Function **RTCS\_gate\_add()** adds gateway *gateway* to RTCS with metric zero.

#### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

#### See Also

- [RTCS\\_gate\\_remove\(\)](#)
- [RTCS\\_if\\_bind\\*](#) family of functions

#### Example

Add a default gateway.

```
error = RTCS_gate_add(GATE_ADDR, INADDR_ANY, INADDR_ANY);
```

### 7.1.118 RTCS\_gate\_add\_metric()

Adds a gateway to the RTCS routing table and assign it's metric.

#### Synopsis

```
uint_32  RTCS_gate_add_metric(
        _ip_address  gateway,
        _ip_address  network,
        _ip_address  netmask
        _uint_16     metric)
```

#### Parameters

*gateway* [in] — IP address of the gateway.

*network* [in] — IP network, in which the gateway is located.

*netmask* [in] — Network mask for *network*.

*metric* [in] — Gateway metric on a scale of zero to 65535.

#### Description

Function **RTCS\_gate\_add\_metric()** associates metric *metric* with gateway *gateway*.

#### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

#### See Also

- [RTCS\\_gate\\_remove\\_metric\(\)](#)
- [RTCS\\_if\\_bind\\*](#) family of functions

#### Example

```
RTCS_gate_add_metric(GATE_ADDR, INADDR_ANY, INADDR_ANY, 42)
```

### 7.1.119 RTCS\_gate\_remove()

Removes a gateway from the routing table.

#### Synopsis

```
uint_32 RTCS_gate_remove(
    _ip_address gateway,
    _ip_address network,
    _ip_address netmask)
```

#### Parameters

*gateway [in]* — IP address of the gateway

*network [in]* — IP network in which the gateway is located

*netmask [in]* — Network mask for *network*

#### Description

Function **RTCS\_gate\_remove()** removes gateway *gateway* from the routing table.

#### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

#### See Also

- [RTCS\\_gate\\_add\(\)](#)

#### Example

Remove the default gateway.

```
error = RTCS_gate_remove(GATE_ADDR, INADDR_ANY, INADDR_ANY);
```

### 7.1.120 RTCS\_gate\_remove\_metric()

Removes a specific gateway from the routing table.

#### Synopsis

```
uint_32 RTCS_gate_remove_metric(
    _ip_address gateway,
    _ip_address network,
    _ip_address netmask
    _uint_16    metric)
```

#### Parameters

*gateway [in]* — IP address of the gateway  
*network [in]* — IP network in which the gateway is located  
*netmask [in]* — Network mask for *network*  
*metric [in]* — Gateway metric on a scale of 0 to 65535

#### Description

Function **RTCS\_gate\_remove\_metric()** removes a specific gateway from the routing table, if it matches the network, netmask, and metric.

#### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

#### See Also

- [RTCS\\_gate\\_add\\_metric\(\)](#)

#### Example

```
error = RTCS_gate_remove_metric
        (GATE_ADDR, INADDR_ANY, INADDR_ANY, 42)
```

### 7.1.121 RTCS\_geterror()

Gets the reason why the RTCS function returned an error for the socket.

#### Synopsis

```
uint_32 RTCS_geterror(  
    uint_32 socket)
```

#### Parameters

*socket [in]* — Socket handle

#### Description

This function does not block. Use this function, if **accept()** returns **RTCS\_SOCKET\_ERROR** or any of the following functions return **RTCS\_ERROR**:

- **recv()**
- **recvfrom()**
- **send()**
- **sendto()**

#### Return Value

- **RTCS\_OK** (no socket error)
- Last error code for the socket

#### See Also

- [accept\(\)](#)
- [recv\(\)](#)
- [recvfrom\(\)](#)
- [send\(\)](#)
- [sendto\(\)](#)

#### Example

See **accept()**, **recv()**, **recvfrom()**, **send()**, and **sendto()**.

## 7.1.122 RTCS\_if\_add()

Adds device interface to RTCS.

### Synopsis

```
uint_32 RTCS_if_add(
    pointer          dev_handle,
    RTCS_IF_STRUCT_PTR callback_ptr,
    _rtcs_if_handle_PTR_ rtcs_if_handle)
```

### Parameters

*dev\_handle* [in] — Handle from **ENET\_initialize()** or **PPP\_initialize()**.

*callback\_ptr* [in] — One of the following:

Pointer to the callback functions for the device interface.

*RTCS\_IF\_ENET* (Ethernet only: uses default callback functions for Ethernet interfaces).

*RTCS\_IF\_LOCALHOST* (uses default callback functions for local loopback).

*RTCS\_IF\_PPP* (PPP only: uses default callback functions for PPP interfaces).

*rtcs\_if\_handle* [out] — Pointer to the RTCS interface handle.

### Description

The application uses the RTCS interface handle to call **RTCS\_if\_bind** functions.

### Return Value

- **RTCS\_OK** (success)
- Error code (failure)

### See Also

- [ENET\\_initialize\(\)](#)
- [PPP\\_initialize\(\)](#)
- [RTCS\\_create\(\)](#)
- [RTCS\\_if\\_bind\(\)](#)
- [RTCS\\_IF\\_STRUCT](#)

### Example

See [Section 2.15.6](#), “Example: Setting Up RTCS.”

### 7.1.123 RTCS\_if\_bind()

Binds the IP address and network mask to the device interface.

#### Synopsis

```
uint_32 RTCS_if_bind(
    _rtcs_if_handle rtcs_if_handle,
    _ip_address      address,
    _ip_address      netmask)
```

#### Parameters

*rtcs\_if\_handle* [in] — RTCS interface handle  
*address* [in] — IP address for the device interface  
*netmask* [in] — Network mask for the interface

#### Description

Function **RTCS\_if\_bind()** binds IP address *address* and network mask *netmask* to the device interface associated with handle *rtcs\_if\_handle*. Parameter *rtcs\_if\_handle* is returned by **RTCS\_if\_add()**.

#### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

#### See Also

- [RTCS\\_if\\_add\(\)](#)
- [RTCS\\_if\\_bind\\_BOOTP\(\)](#)
- [RTCS\\_if\\_bind\\_DHCP\(\)](#)
- [RTCS\\_if\\_bind\\_DHCP\\_flagged\(\)](#)
- [RTCS\\_if\\_rebind\\_DHCP\(\)](#)

#### Example

See [Section 2.15.6](#), “Example: Setting Up RTCS.”

## 7.1.124 RTCS\_if\_bind\_BOOTP()

Gets an IP address using BootP and binds it to the device interface.

### Synopsis

```
uint_32 RTCS_if_bind_BOOTP(
    _rtcs_if_handle    rtcs_if_handle,
    BOOTP_DATA_STRUCT_PTR data_ptr)
```

### Parameters

*rtcs\_if\_handle* [in] — RTCS interface handle from  
*data\_ptr* [in/out] — Pointer to BootP data

### Description

This function uses BootP to assign an IP address, determines a boot file to download, and determines the server, from which to download it. Parameter *rtcs\_if\_handle* is returned by **RTCS\_if\_add()**.

### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

### See Also

- [RTCS\\_if\\_add\(\)](#)
- [RTCS\\_if\\_bind\(\)](#)
- [RTCS\\_if\\_bind\\_DHCP\(\)](#)
- [RTCS\\_if\\_bind\\_IPCP\(\)](#)
- [BOOTP\\_DATA\\_STRUCT](#)

### Example

```
BOOTP_DATA_STRUCT boot_data;

uint_32 boot_function(void)
{
    BOOTP_DATA_STRUCT  boot_data;
    _enet_handle        ehandle;
    _rtcs_if_handle     ihandle;
    uint_32             error;

    error = ENET_initialize(0, enet_local, 0, &ehandle);
    if (error) return error;

    error = RTCS_create();
    if (error) return error;

    error = RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);
    if (error) return error;

    memset(&boot_data, 0, sizeof(boot_data));
    error = RTCS_if_bind_BOOTP(ihandle, &boot_data);
    if (error) return error;
```



```
error = RTCS_exec_TFTP_SREC (boot_data.SADDR,  
                             (char_ptr)boot_data.BOOTFILE);  
  
return error;  
}
```

## 7.1.125 RTCS\_if\_bind\_DHCP()

Gets an IP address using DHCP and binds it to the device interface.

### Synopsis

```
uint_32  RTCS_if_bind_DHCP(
    _rtcs_if_handle  rtcs_if_handle,
    DHCP_DATA_STRUCT_PTR  callback_ptr,
    char_ptr  optptr,
    uint_32  optlen)
```

### Parameters

*rtcs\_if\_handle* [in] — RTCS interface handle.

*callback\_ptr* [in] — Pointer to the callback functions for DHCP.

*optptr* [in] — One of the following:

- pointer to the buffer of DHCP params (see RFC 2132)
- NULL

*optlen* [in] — Number of bytes in the buffer pointed to by *optptr*.

### Description

Function **RTCS\_if\_bind\_DHCP()** uses DHCP to get an IP address and bind it to the device interface. Parameter *rtcs\_if\_handle* is returned by **RTCS\_if\_add()**.

This function blocks until DHCP completes initialization, but not until it binds the interface.

### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

### See Also

- [RTCS\\_if\\_add\(\)](#)
- [RTCS\\_if\\_bind\(\)](#)
- [RTCS\\_if\\_bind\\_BOOTP\(\)](#)
- [RTCS\\_if\\_bind\\_DHCP\\_flagged\(\)](#)
- [RTCS\\_if\\_bind\\_DHCP\\_timed\(\)](#)
- [RTCS\\_if\\_bind\\_IPCP\(\)](#)
- [DHCP\\_DATA\\_STRUCT](#)

### Example

```
_enet_handle  ehandle;
_rtcs_if_handle  ihandle;
uint_32  error;
uint_32  optlen = 100; /* Use the size that you need for
                        the number of params that you
                        are using with DHCP */

uchar  option_array[100];
uchar _PTR_ optptr;
```

```

DHCP_DATA_STRUCT params;
uchar          parm_options[3] = {DHCHOPT_SERVERNAME,
                                  DHCHOPT_FILENAME,
                                  DHCHOPT_FINGER_SRV};

error = ENET_initialize(0, enet_local, 0, &ehandle);
if (error) {
    printf("\nFailed to initialize Ethernet driver: %s.",
           ENET_strerror(error));
    return;
}

error = RTCS_create();
if (error != RTCS_OK) {
    printf("\nFailed to create RTCS, error = %x.", error);
    return;
}

error = RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);
if (error) {
    printf("\nFailed to add the interface, error = %x.", error);
    return;
}

/* You supply the following functions; if any is NULL, DHCP Client
   follows its default behavior. */
params.CHOICE_FUNC = DHCPCLNT_test_choice_func;
params.BIND_FUNC   = DHCPCLNT_test_bind_func;
params.UNBIND_FUNC = DHCPCLNT_test_unbind_func;

optptr = option_array;
/* Fill in the requested params: */
/* Request a three-minute lease: */
DHCP_option_int32(&optptr, &optlen, DHCHOPT_LEASE, 180);
/* Request a TFTP Server, FILENAME, and Finger Server: */
DHCP_option_variable(&optptr, &optlen, DHCHOPT_PARAMLIST,
                    parm_options, 3);

error = RTCS_if_bind_DHCP(ihandle, &params, option_array,
                        optptr - option_array);
if (error) {
    printf("\nDHCP boot failed, error = %x.", error);
    return;
}

/* Use the network interface when it is bound. */

```

## 7.1.126 RTCS\_if\_bind\_DHCP\_flagged()

Gets an IP address using DHCP and binds it to the device interface using parameters defined by the flags in *dhcp.h*.

### 7.1.126.1 Synopsis

```
uint_32 RTCS_if_bind_DHCP_flagged(
    _rtcs_if_handle    rtcs_if_handle,
    DHCP_DATA_STRUCT_PTR params,
    char_ptr           optptr,
    uint_32            optlen)
```

#### Parameters

*rtcs\_if\_handle* [in] — RTCS interface handle.

*params* [in] — Optional parameters

*params*->CHOICE\_FUNC

*params*->BIND\_FUNC

*params*->REBIND\_FUNC

*params*->UNBIND\_FUNC

*params*->FAILURE\_FUNC

*params*->FLAGS

*optptr* [in] — One of the following:

Pointer to the buffer of DHCP params (see RFC 2132).

NULL

*optlen* [in] — Number of bytes in the buffer pointed to by *optptr*.

#### Description

Function **RTCS\_if\_bind\_DHCP\_flagged()** uses DHCP to get an IP address and bind it to the device interface. The *TCPIP\_PARM\_IF\_DHCP* structure is defined in *dhcp\_prv.h*. The FLAGS are defined in *dhcp.h*. Parameter *rtcs\_if\_handle* is returned by **RTCS\_if\_add()**.

To have the DHCP client accept offered IP addresses without probing the network, do not set *DHCP\_SEND\_PROBE* in *params*->FLAGS.

This function blocks until DHCP completes initialization, but not until it binds the interface.

#### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

#### See Also

- [RTCS\\_if\\_add\(\)](#)
- [RTCS\\_if\\_bind\(\)](#)
- [RTCS\\_if\\_bind\\_BOOTP\(\)](#)
- [RTCS\\_if\\_bind\\_IPCP\(\)](#)

- *DHCP\_DATA\_STRUCT*

### Example

```

_enet_handle      ehandle;
_rtcs_if_handle   ihandle;
uint_32           error;
uint_32           optlen = 100; /* Use the size that you need for
                                the number of params that you
                                are using with DHCP */

uchar             option_array[100];
uchar_PTR         optptr;
DHCP_DATA_STRUCT  params;
uchar             parm_options[3] = {DHCHOPT_SERVERNAME,
                                      DHCHOPT_FILENAME,
                                      DHCHOPT_FINGER_SRV};

error = ENET_initialize(0, enet_local, 0, &ehandle);
if (error) {
    printf("\nFailed to initialize Ethernet driver: %s.",
          ENET_strerror(error));
    return;
}

error = RTCS_create();
if (error != RTCS_OK) {
    printf("\nFailed to create RTCS, error = %x.", error);
    return;
}

error = RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);
if (error) {
    printf("\nFailed to add the interface, error = %x.", error);
    return;
}

/* You supply the following functions; if any is NULL, DHCP Client
   follows its default behavior. */
params.FLAGS = 0;
params.FLAGS |= DHCP_SEND_INFORM_MESSAGE;
params.FLAGS |= DHCP_MAINTAIN_STATE_ON_INFINITE_LEASE;
params.FLAGS |= DHCP_SEND_PROBE;
params.CHOICE_FUNC = DHCPCLNT_test_choice_func;
params.BIND_FUNC   = DHCPCLNT_test_bind_func;
params.UNBIND_FUNC = DHCPCLNT_test_unbind_func;

optptr = option_array;
/* Fill in the requested params: */
/* Request a three-minute lease: */
DHCP_option_int32(&optptr, &optlen, DHCHOPT_LEASE, 180);
/* Request a TFTP Server, FILENAME, and Finger Server: */
DHCP_option_variable(&optptr, &optlen, DHCHOPT_PARAMLIST,
                    parm_options, 3);

error = RTCS_if_bind_DHCP(ihandle, &params, option_array,
                          optptr - option_array);
if (error) {
    printf("\nDHCP boot failed, error = %x.", error);
}

```

## Function Reference

```
    return;  
}  
/* Use the network interface when it is bound. */
```

### 7.1.127 RTCS\_if\_bind\_DHCP\_timed()

Gets an IP address using DHCP and binds it to the device interface within a timeout.

#### Synopsis

```
uint_32 RTCS_if_bind_DHCP_timed(
    _rtcs_if_handle    rtcs_if_handle,
    DHCP_DATA_STRUCT_PTR params,
    char_ptr           optptr,
    uint_32             optlen)
```

#### Parameters

*rtcs\_if\_handle* [in] — RTCS interface handle.

*params* [in] — Optional parameters

*params*->CHOICE\_FUNC

*params*->BIND\_FUNC

*params*->REBIND\_FUNC

*params*->UNBIND\_FUNC

*params*->FAILURE\_FUNC

*params*->FLAGS

*optptr* [in] — One of the following:

Pointer to the buffer of DHCP params (see RFC 2132).

NULL.

*optlen* [in] — Number of bytes in the buffer pointed to by *optptr*.

#### Description

Function **RTCS\_if\_bind\_DHCP\_timed()** uses DHCP to get an IP address and bind it to the device interface. If the interface does not bind via DHCP within the timeout limit, the client stops trying to bind and exits. Parameter *rtcs\_if\_handle* is returned by **RTCS\_if\_add()**.

This function blocks until DHCP completes initialization, but not until it binds the interface.

#### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

#### See Also

- [RTCS\\_if\\_add\(\)](#)
- [RTCS\\_if\\_bind\(\)](#)
- [RTCS\\_if\\_bind\\_BOOTP\(\)](#)
- [RTCS\\_if\\_bind\\_IPCP\(\)](#)
- [DHCP\\_DATA\\_STRUCT](#)

## Example

```

_enet_handle      ehandle;
_rtcs_if_handle   ihandle;
uint_32           error;
uint_32           optlen = 100; /* Use the size that you need for
                                the number of params that you
                                are using with DHCP */

uchar             option_array[100];
uchar_PTR         optptr;
DHCP_DATA_STRUCT  params;
uchar             parm_options[3] = {DHCHOPT_SERVERNAME,
                                      DHCHOPT_FILENAME,
                                      DHCHOPT_FINGER_SRV};

uint_32           timeout = 120; /* two minutes*/

error = ENET_initialize(0, enet_local, 0, &ehandle);
if (error) {
    printf("\nFailed to initialize Ethernet driver: %s.",
          ENET_strerror(error));
    return;
}

error = RTCS_create();
if (error != RTCS_OK) {
    printf("\nFailed to create RTCS, error = %x.", error);
    return;
}

error = RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);
if (error) {
    printf("\nFailed to add the interface, error = %x.", error);
    return;
}

/* You supply the following functions; if any is NULL, DHCP Client
   follows its default behavior. */
params.CHOICE_FUNC = DHCPCLNT_test_choice_func;
params.BIND_FUNC   = DHCPCLNT_test_bind_func;
params.UNBIND_FUNC = DHCPCLNT_test_unbind_func;

optptr = option_array;
/* Fill in the requested params: */
/* Request a three-minute lease: */
DHCP_option_int32(&optptr, &optlen, DHCHOPT_LEASE, 180);
/* Request a TFTP Server, FILENAME, and Finger Server: */
DHCP_option_variable(&optptr, &optlen, DHCHOPT_PARAMLIST,
                    parm_options, 3);

error = RTCS_if_bind_DHCP_timed(ihandle, &params, option_array,
                                optptr - option_array, timeout);
if (error) {
    printf("\nDHCP boot failed, error = %x.", error);
    return;
}

/* Use the network interface if it successfully binds. Check
   after the timeout value to see if it did bind. */

```



## 7.1.128 RTCS\_if\_bind\_IPCP()

Binds an IP address to the PPP device interface.

### Synopsis

```
uint_32 RTCS_if_bind_IPCP(
    _rtcs_if_handle    rtcs_if_handle,
    IPCP_DATA_STRUCT_PTR data_ptr)
```

### Parameters

*rtcs\_if\_handle* [in] — RTCS interface handle for PPP device.

*data\_ptr* [in] — Pointer to the IPCP data.

### Description

Function **RTCS\_if\_bind\_IPCP()** is the only way to bind an IP address to a PPP device interface.

The function starts to negotiate IPCP over the PPP interface that is specified by *rtcs\_if\_handle* (returned by **RTCS\_if\_add()**). The function returns immediately; it does not wait until IPCP has completed negotiation. The *IPCP\_DATA\_STRUCT* contains configuration parameters and a set of application callback functions that RTCS is to call when certain events occur. For details, see *IPCP\_DATA\_STRUCT* in Chapter 8, “Data Types.”

### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

### See Also

- [PPP\\_initialize\(\)](#)
- [RTCS\\_if\\_add\(\)](#)
- [RTCS\\_if\\_bind\(\)](#)
- [IPCP\\_DATA\\_STRUCT](#)

### Example

Initialize PPP and bind to the interface.

```
void boot_done(pointer sem) {
    _lwsem_post(sem);
}

int_32 init_ppp(void)
{
    FILE_PTR          pppfile;
    _iopcb_handle     pppio;
    _ppp_handle       phandle;
    _rtcs_if_handle    ihandle;
    IPCP_DATA_STRUCT  ipcp_data;
    LWSEM_STRUCT       boot_sem;

    pppfile = fopen("ittya:", NULL);
    if (pppfile == NULL) return -1;
```

```
pppio = _iopcb_ppphdlc_init(pppfile);
if (pppio == NULL) return -1;
error = PPP_initialize(pppio, &phandle);
if (error) return error;
_iopcb_open(pppio, PPP_lowerup, PPP_lowerdown, phandle);
error = RTCS_if_add(phandle, RTCS_IF_PPP, &ihandle);
if (error) return error;

_lwsem_create(&boot_sem, 0);
memset(&ipcp_data, 0, sizeof(ipcp_data));
ipcp_data.IP_UP = boot_done;
ipcp_data.IP_DOWN = NULL;
ipcp_data.IP_PARAM = &boot_sem;
ipcp_data.ACCEPT_LOCAL_ADDR = FALSE;
ipcp_data.ACCEPT_REMOTE_ADDR = FALSE;
ipcp_data.LOCAL_ADDR = PPP_LOCADDR;
ipcp_data.REMOTE_ADDR = PPP_PEERADDR;
ipcp_data.DEFAULT_NETMASK = TRUE;
ipcp_data.DEFAULT_ROUTE = TRUE;

error = RTCS_if_bind_IPCP(ihandle, &ipcp_data);
if (error) return error;

_lwsem_wait(&boot_sem);
printf("IPCP is up\n");
return 0;
}
```

## 7.1.129 RTCS\_if\_rebind\_DHCP()

Binds a previously used IP address to the device interface.

### Synopsis

```
uint_32 RTCS_if_rebind_DHCP(
    _rtcs_if_handle    rtcs_if_handle,
    _ip_address        address,
    _ip_address        netmask,
    uint_32            lease,
    _ip_address        server,
    DHCP_DATA_STRUCT_PTR params,
    uchar_ptr          optptr,
    uint_32            optlen)
```

### Parameters

*handle [in]* — RTCS interface handle.

*address [in]* — IP address for the interface.

*netmask [in]* — IP address of the network or subnet mask for the interface.

*lease [in]* — Duration in seconds of the lease.

*server [in]* — IP address of the DHCP Server.

*params* — Optional parameters

*params->CHOICE\_FUNC*

*params->BIND\_FUNC*

*params->REBIND\_FUNC*

*params->UNBIND\_FUNC*

*params->FAILURE\_FUNC*

*params->FLAGS*

*optptr [in]* — One of the following:

Pointer to the buffer of DHCP options (see RFC 2132).

NULL.

*optlen [in]* — Number of bytes in the buffer pointed to by *optptr*.

### Description

Function **RTCS\_if\_rebind\_DHCP()** uses DHCP to get an IP address and bind it to the device interface. Parameter *rtcs\_if\_handle* is returned by **RTCS\_if\_add()**.

This function blocks until DHCP completes initialization, but not until it binds the interface.

### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

### See Also

- [RTCS\\_if\\_add\(\)](#)
- [RTCS\\_if\\_bind\(\)](#)
- [RTCS\\_if\\_bind\\_BOOTP\(\)](#)
- [RTCS\\_if\\_bind\\_DHCP\\_flagged\(\)](#)
- [RTCS\\_if\\_bind\\_DHCP\\_timed\(\)](#)
- [RTCS\\_if\\_bind\\_IPCP\(\)](#)
- [DHCP\\_DATA\\_STRUCT](#)

## Example

```

_enet_handle      ehandle;
_rtcs_if_handle   ihandle;
uint_32           error;
uint_32           optlen = 100; /* Make large enough for the number
                                of your DHCP options */

uchar             option_array[100];
uchar_PTR         optptr;
DHCP_DATA_STRUCT  params;
uchar             parm_options[3] = {DHCHOPT_SERVERNAME,
                                      DHCHOPT_FILENAME,
                                      DHCHOPT_FINGER_SRV};

in_addr           rebind_address, rebind_mask, rebind_server;
uint_32           lease = 28800; /* 8 Hours, in seconds */

error = ENET_initialize(0, enet_local, 0, &ehandle);
if (error) {
    printf("\nFailed to initialize Ethernet driver: %s.",
          ENET_strerror(error));
    return;
}
error = RTCS_create();
if (error != RTCS_OK) {
    printf("\nFailed to create RTCS, error = %x.", error);
    return;
}
error = RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);
if (error) {
    printf("\nFailed to add the interface, error = %x.", error);
    return;
}
/* You supply the following functions; if any is NULL, DHCP Client
   follows its default behavior. */
params.CHOICE_FUNC = DHCPCLNT_test_choice_func;
params.BIND_FUNC   = DHCPCLNT_test_bind_func;
params.UNBIND_FUNC = DHCPCLNT_test_unbind_func;
optptr = option_array;
/* Fill in the requested options: */
/* Request a three-minute lease: */
DHCP_option_int32(&optptr, &optlen, DHCHOPT_LEASE, 180);
/* Request a TFTP Server, FILENAME, and Finger Server: */
DHCP_option_variable(&optptr, &optlen, DHCHOPT_PARAMLIST,
                    parm_options, 3);
error = inet_aton ("192.168.1.100", &rebind_address);
error |= inet_aton ("255.255.255.0", &rebind_mask);
error |= inet_aton ("192.168.1.2", &rebind_server);

```

```
if (error) {
    printf("\nFailed to convert IP addresses from dotted decimal, error = %x.", error);
    return;
}
error = RTCS_if_rebind_DHCP(ihandle,
                           rebind_address,
                           rebind_mask,
                           lease,
                           rebind_server,
                           &params,
                           option_array,
                           optptr - option_array);

if (error) {
    printf("\nDHCP boot failed, error = %x.", error);
    return;
}
```

### 7.1.130 RTCS\_if\_remove()

Removes the device interface from RTCS.

#### Synopsis

```
uint_32 RTCS_if_remove(  
    _rtcs_if_handle rtcs_if_handle)
```

#### Parameters

*rtcs\_if\_handle [in]* — RTCS interface handle.

#### Description

Function **RTCS\_if\_remove()** removes the device interface associated with *rtcs\_if\_handle* (returned by **RTCS\_if\_add()**) from RTCS.

#### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

#### See Also

- [RTCS\\_if\\_add\(\)](#)
- [RTCS\\_if\\_rebind\\_DHCP\(\)](#)

### 7.1.131 RTCS\_if\_unbind()

Unbinds the IP address from the device interface.

#### Synopsis

```
uint_32 RTCS_if_unbind(  
    _rtcs_if_handle rtcs_if_handle,  
    _ip_address      address)
```

#### Parameters

*rtcs\_if\_handle* [in] — RTCS interface handle.

*address* [in] — IP address to unbind.

#### Description

Function **RTCS\_if\_unbind()** unbinds IP address *address* from the device interface associated with *rtcs\_if\_handle*. Parameter *rtcs\_if\_handle* is returned by **RTCS\_if\_add()**.

#### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

#### See Also

- [RTCS\\_if\\_add\(\)](#)
- [RTCS\\_if\\_bind\(\)](#)
- [RTCS\\_if\\_bind\\_BOOTP\(\)](#)
- [RTCS\\_if\\_bind\\_DHCP\(\)](#)
- [RTCS\\_if\\_bind\\_IPCP\(\)](#)
- [RTCS\\_if\\_rebind\\_DHCP\(\)](#)

### 7.1.132 RTCS\_load\_TFTP\_BIN()

Downloads the binary file.

#### Synopsis

```
uint_32  RTCS_load_TFTP_BIN(
        _ip_address  server,
        char_ptr     filename,
        uchar_ptr    start_download_address)
```

#### Parameters

*server* [in] — IP address of the TFTP Server.

*filename* [in] — Name of the file to download.

*start\_download\_address* [in] — Address, at which to download the file.

#### Description

This function downloads the binary file from the TFTP Server. It is the same as **RTCS\_exec\_TFTP\_BIN()**, with the exception that it does not run the file after it downloads the file. For information on the values of *server* and *filename*, see [Table 7-1](#).

#### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

#### See Also

- [RTCS\\_exec\\_TFTP\\_BIN\(\)](#)
- [RTCS\\_if\\_bind\\_BOOTP\(\)](#)
- [BOOTP\\_DATA\\_STRUCT](#)



### 7.1.133 RTCS\_load\_TFTP\_COFF()

Downloads the COFF boot file.

#### Synopsis

```
uint_32  RTCS_load_TFTP_COFF(  
    _ip_address  server,  
    char_ptr     filename)
```

#### Parameters

*server* [in] — IP address of the TFTP Server.

*filename* [in] — Name of the file to download.

#### Description

This function downloads the binary file from the TFTP Server. This function is the same as **RTCS\_exec\_TFTP\_COFF()**, with the exception that it does not run the file after it downloads the file. For information on the values of *server* and *filename*, see [Table 7-1](#).

#### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

#### See Also

- [RTCS\\_exec\\_TFTP\\_COFF\(\)](#)
- [RTCS\\_if\\_bind\\_BOOTP\(\)](#)
- [BOOTP\\_DATA\\_STRUCT](#)

## 7.1.134 RTCS\_load\_TFTP\_SREC()

Downloads the S-Record file.

### Synopsis

```
uint_32  RTCS_load_TFTP_SREC(  
    _ip_address  server,  
    char_ptr     filename)
```

### Parameter

*server* [in] — IP address of the TFTP Server.

*filename* [in] — Name of the file to download.

### Description

This function downloads the S-Record file from the TFTP Server. This function is the same as **RTCS\_exec\_TFTP\_SREC()**, with the exception that it does not run the file after it downloads the file. For information on the values of *server* and *filename*, see [Table 7-1](#).

### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

### See Also

- [RTCS\\_exec\\_TFTP\\_SREC\(\)](#)
- [RTCS\\_if\\_bind\\_BOOTP\(\)](#)
- [BOOTP\\_DATA\\_STRUCT](#)

### 7.1.135 RTCS\_ping()

Sends an ICMP echo-request packet to the IP address and waits for a reply.

#### Synopsis

```
uint_32  RTCS_ping(
    ip_address    address,
    uint_32_ptr   timeout,
    uint_16       id)
```

#### Parameters

*address [in]* — IP address, to which to send the packet.

*timeout [in/out]* — When passed in, one of the following:

Pointer to the maximum time to wait for a reply.

Zero (waits indefinitely).

When passed out, pointer to the round-trip time.

*id [in]* — User ID for the echo request.

#### Description

Function **RTCS\_ping()** is the RTCS implementation of **ping**. It sends an ICMP echo-request packet to IP address *address* and waits for a reply.

#### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

### 7.1.136 RTCS\_request\_DHCP\_inform()

Requests a DHCP information message.

#### Synopsis

```
uint_32 RTCS_request_DHCP_inform(
    _rtcs_if_handle    handle,
    uchar_ptr          optptr,
    uint_32             optlen,
    _ip_address         client_addr,
    _ip_address         server_addr,
    void               (_CODE_PTR_ inform_func) (uchar _PTR_,
    uint_32, _rtcs_if_handle))
```

#### Parameters

*handle [in]* — RTCS interface handle.

*optptr [in]* — One of the following:

Pointer to the buffer of DHCP options (see RFC 2132)

NULL.

*optlen [in]* — Number of bytes in the buffer pointed to by *optptr*.

*client\_addr [in]* — IP address, where the application is bound.

*server\_addr [in]* — IP address of the server, for which information is needed.

*inform\_func* — Function to call, when DHCP is finished.

#### Description

Function **RTCS\_request\_DHCP\_inform()** requests an information message about server *server*.

#### Return Value

- Server DHCP information (success)
- Error code (failure)

## 7.1.137 RTCS\_selectall()

Waits for activity on any socket that the caller owns.

### Synopsis

```
uint_32 RTCS_selectall(
    uint_32 timeout)
```

### Parameters

*timeout [in]* — One of the following:

Maximum number of milliseconds to wait for activity.

Zero (waits indefinitely).

–1 (does not block).

### Description

If *timeout* is not –1, the function blocks, until activity is detected on any socket that the calling task owns. *Activity* consists of any of the following.

Socket	Receives
Unbound datagram	Datagrams.
Listening stream	Connection requests.
Connected stream	Data or Shutdown requests that are initiated by the remote endpoint.

### Return Value

- Socket handle (activity was detected)
- Zero (*timeout* expired)
- *RTCS\_SOCKET\_ERROR* (error)

### See Also

- [RTCS\\_attachsock\(\)](#)
- [RTCS\\_detachsock\(\)](#)
- [RTCS\\_selectset\(\)](#)

### Example

Echo data on TCP port number seven.

```
int_32 servsock;
int_32 connsock;
int_32 status;
SOCKET_ADDRESS_STRUCT addrpeer;
uint_16 addrlen;
char buf[500];
int_32 count;
uint_32 error

/* create a stream socket and bind it to port 7: */
```

```
error = listen(servsock, 0);
if (error != RTCS_OK) {
    printf("\nlisten() failed, status = %d", error);
    return;
}

for (;;) {
    connsock = RTCS_selectall(0);

    if (connsock == RTCS_SOCKET_ERROR) {
        printf("\nRTCS_selectall() failed!");
    } else if (connsock == servsock) {
        status = accept(servsock, &addrpeer, &addrlen);
        if (status == RTCS_SOCKET_ERROR)
            printf("\naccept() failed!");
    } else {
        count = recv(connsock, buf, 500, 0);
        if (count <= 0)
            shutdown(connsock, FLAG_CLOSE_TX);
        else
            send(connsock, buf, count, 0);
    }
}
```

## 7.1.138 RTCS\_selectset()

Waits for activity on any socket in the set of sockets.

### Synopsis

```
uint_32  RTCS_selectset(
    pointer  socket,
    uint_32  count,
    uint_32  timeout)
```

### Parameters

- socket* [in] — Pointer to an array of sockets.
- count* [in] — Number of sockets in the array.
- timeout* [in] — One of the following:
  - Maximum number of milliseconds to wait for activity.
  - Zero (waits indefinitely).
  - 1 (does not block).

### Description

If *timeout* is not –1, the function blocks, until activity is detected on at least one of the sockets in the set. For a description of what constitutes *activity*, see [RTCS\\_selectall\(\)](#).

### Return Value

- Socket handle (activity was detected)
- Zero (*timeout* expired)
- `RTCS_SOCKET_ERROR` (error)

### See Also

- [RTCS\\_selectall\(\)](#)

### Example

Echo UDP data that is received on ports 2010, 2011, and 2012.

```
int_32      socklist[3];
sockaddr_in local_sin;
uint_32     result;

...

memset((char *) &local_sin, 0, sizeof(local_sin));

local_sin.sin_family = AF_INET;
local_sin.sin_addr.s_addr = INADDR_ANY;

local_sin.sin_port = 2010;
socklist[0] = socket(AF_INET, SOCK_DGRAM, 0);
result = bind(socklist[0], &local_sin, sizeof (sockaddr_in));

local_sin.sin_port = 2011;
socklist[1] = socket(AF_INET, SOCK_DGRAM, 0);
```

```
result = bind(socklist[1], &local_sin, sizeof (sockaddr_in));

local_sin.sin_port = 2012;
socklist[2] = socket(AF_INET, SOCK_DGRAM, 0);
result = bind(socklist[2], &local_sin, sizeof (sockaddr_in));

while (TRUE) {
    sock = RTCS_selectset(socklist, 3, 0);

    rlen = sizeof(raddr);
    length = recvfrom(sock, buffer, BUFFER_SIZE, 0, &raddr, &rlen);
    sendto(sock, buffer, length, 0, &raddr, rlen);
}
```



### 7.1.139 RTCSLOG\_disable()

Disables RTCS logging.

#### Synopsis

```
void RTCSLOG_disable(  
    uint_32 logtype)
```

#### Parameters

*logtype* [in] — Class or classes of entries to stop logging.

#### Description

The function disables RTCS event logging in the MQX kernel log. *logtype* is a bitwise **OR** of either of the following:

- *RTCS\_LOGCTRL\_FUNCTION* — Logs all socket API calls.
- *RTCS\_LOGCTRL\_PCB* — Logs packet generation and parsing.
- Alternatively, *logtype* can be *RTCS\_LOGCTRL\_ALL* to disable all classes of log entries.

#### See Also

#### RTCSLOG\_enable()

#### Example

See [RTCSLOG\\_enable\(\)](#).

## 7.1.140 RTCSLOG\_enable()

Enables RTCS logging.

### Synopsis

```
void RTCSLOG_enable(
    uint_32 logtype)
```

### Parameters

*logtype* [in] — Class or classes of entries to start logging.

### Description

The function enables RTCS event logging in the MQX kernel log. *logtype* is a bitwise **OR** of any of the following:

- *RTCS\_LOGCTRL\_FUNCTION* — Logs all socket API calls.
- *RTCS\_LOGCTRL\_PCB* — Logs packet generation and parsing.
- Alternatively, *logtype* can be *RTCS\_LOGCTRL\_ALL* to enable all classes of log entries.

RTCS log entries are written into the kernel log. Therefore, the kernel log must have been created prior to enabling RTCS logging.

In addition, the socket API log entries belong to the kernel log functions group in the kernel. To log socket API calls, this group must be enabled using the MQX function **\_klog\_control()**.

### See Also

- [RTCSLOG\\_disable\(\)](#)
- **\_klog\_create()** in *MQX Reference Manual*
- **\_klog\_control()** in *MQX Reference Manual*

### Example

Create the kernel log.

```
_klog_create(16384, 0);
/* Tell MQX to log RTCS functions */
_klog_control(KLOG_ENABLED | KLOG_FUNCTIONS_ENABLED |
    RTCSLOG_FNBASE, TRUE);
/* Tell RTCS to start logging */
RTCSLOG_enable(RTCS_LOGCTRL_ALL);

/* ... */

/* Tell RTCS to stop logging */
RTCSLOG_disable(RTCS_LOGCTRL_ALL);
```

## 7.1.141 send()

Sends data on the stream socket, or on a datagram socket, for which a remote endpoint has been specified.

### Synopsis

```
int_32 send(
    uint_32      socket,
    char *_PTR_  buffer,
    uint_32      buflen,
    uint_32      flags)
```

### Parameters

*socket [in]* — Handle for the socket, on which to send data.

*buffer [in]* — Pointer to the buffer of data to send.

*buflen [in]* — Number of bytes in the buffer (no restriction).

*flags [in]* — For datagram sockets only: Flags to underlying protocols, selected from three independent groups. Perform a bitwise **OR** of one flag only from one or more of the groups described in [Section , “Flags,”](#) below.

### Description

Function **send()** sends data on a stream socket, or on a datagram socket, for which a remote endpoint has been specified.

#### Stream Socket

RTCS packetizes the data (at *buffer*) into TCP packets and delivers the packets reliably and sequentially to the connected remote endpoint.

If the send-nowait socket option is TRUE, RTCS immediately copies the data into the internal send buffer for the socket, to a maximum of *buflen*. The function then returns.

If the send-push socket option is TRUE, RTCS appends a push flag to the last packet that it uses to send the buffer; all data is sent immediately, taking into account the capabilities of the remote endpoint buffer.

#### Datagram Socket

If a remote endpoint is specified using **connect()**, **send()** is identical to **sendto()** using the specified remote endpoint. If a remote endpoint is not specified, **send()** returns *RTCS\_ERROR*.

The *flags* parameter is for datagram sockets only. The override is temporary and lasts for the current call to **send()** only. Setting *flags* to *RTCS\_MSG\_NOLOOP* is useful when broadcasting or multicasting a datagram to several destinations. When *flags* is set to *RTCS\_MSG\_NOLOOP*, the datagram is not duplicated for the local host interface.

## Flags

### Group 1:

- *RTCS\_MSG\_BLOCK* — overrides the *OPT\_SEND\_NOWAIT* datagram socket option; makes it behave as if it was FALSE.
- *RTCS\_MSG\_NONBLOCK* — overrides the *OPT\_SEND\_NOWAIT* datagram socket option; makes it behave as if it was TRUE

### Group 2:

- *RTCS\_MSG\_CHKSUM* — overrides the *OPT\_CHECKSUM\_BYPASS* checksum bypass option; makes it behave as if it was FALSE.
- *RTCS\_MSG\_NOCHKSUM* — overrides the *OPT\_CHECKSUM\_BYPASS* checksum bypass option; makes it behave as though it is TRUE.

### Group 3:

- *RTCS\_MSG\_NOLOOP* — does not send the datagram to the loopback interface.
- Zero — ignore.

## Return Value

- Number of bytes sent (success)
- *RTCS\_ERROR* (failure)

If the function returns **RTCS\_ERROR**, the application can call **RTCS\_geterror()** to determine the cause of the error.

## See Also

- [accept\(\)](#)
- [bind\(\)](#)
- [getsockopt\(\)](#)
- [listen\(\)](#)
- [recv\(\)](#)
- [RTCS\\_geterror\(\)](#)
- [setsockopt\(\)](#)
- [shutdown\(\)](#)
- [socket\(\)](#)

## Example: Stream Socket

```
uint_32  handle;
char     buffer[20000];
uint_32  count;

...

count = send(handle, buffer, 20000, 0);
if (count == RTCS_ERROR)
    printf("\nError, send() failed with error code %lx",
```

```
RTCS_geterror(handle);
```

## 7.1.142 sendto()

Sends data on the datagram socket.

### Synopsis

```
int_32 sendto(
    uint_32      socket,
    char *_PTR_  buffer,
    uint_32      buflen,
    uint_16      flags,
    sockaddr_in *_PTR_ destaddr,
    uint_16      addrlen)
```

### Parameters

*socket [in]* — Handle for the socket, on which to send data.

*buffer [in]* — Pointer to the buffer of data to send.

*buflen [in]* — Number of bytes in the buffer (no restriction).

*flags [in]* — Flags to underlying protocols, selected from three independent groups. Perform a bitwise **OR** of one flag only from one or more of the groups described under [Section , “Flags.”](#)

### Description

The function sends the data (at *buffer*) as a UDP datagram to the remote endpoint (at *destaddr*).

This function can also be used when a remote endpoint has been prespecified through **connect()**. The datagram is sent to *destaddr*, even if it is different than the prespecified remote endpoint.

If the socket address has been prespecified, you can call **sendto()** with *destaddr* set to NULL and *addrlen* equal to zero: this combination sends to the prespecified address. Calling **sendto()** with *destaddr* set to NULL and *addrlen* equal to zero without first having prespecified the destination will result in an error.

The override is temporary and lasts for the current call to **sendto()** only. Setting *flags* to *RTCS\_MSG\_NOLOOP* is useful when broadcasting or multicasting a datagram to several destinations. When *flags* is set to *RTCS\_MSG\_NOLOOP*, the datagram is not duplicated for the local host interface.

If the function returns *RTCS\_ERROR*, the application can call **RTCS\_geterror()** to determine the cause of the error.

This function blocks, but the command is immediately serviced and replied to.

### Return Value

- Number of bytes sent (success)
- *RTCS\_ERROR* (failure)

### See Also

- [setsockopt\(\)](#)
- [bind\(\)](#)
- [recvfrom\(\)](#)
- [RTCS\\_geterror\(\)](#)
- [socket\(\)](#)

## Example

Send 500 bytes of data to IP address 192.203.0.54, port number 678.

```
uint_32      handle;
sockaddr_in  remote_sin;
uint_32      count;
char         my_buffer[500];
...
for (i=0; i < 500; i++) my_buffer[i]= (i & 0xff);
memset((char *) &remote_sin, 0, sizeof(sockaddr_in));

remote_sin.sin_family = AF_INET;
remote_sin.sin_port = 678;
remote_sin.sin_addr.s_addr = 0xC0CB0036;

count = sendto(handle, my_buffer, 500, 0, &remote_sin,
               sizeof(sockaddr_in));
if (count != 500)
    printf("\nsendto() failed with count %ld and error %lx",
           count, RTCS_geterror(handle));
```

## 7.1.143 setsockopt()

Sets the value of the socket option.

### Synopsis

```
uint_32  setsockopt(
    uint_32  socket,
    uint_32  level,
    uint_32  optname,
    pointer  optval,
    uint_32  optlen)
```

### Parameters

*socket* [in] — One of the following:

- if *level* is anything but *SOL\_NAT*, handle for the socket whose option is to be changed.
- if *level* is *SOL\_NAT*, *socket* is ignored.

*level* [in] — Protocol levels, at which the option resides:

*SOL\_IGMP*  
*SOL\_LINK*  
*SOL\_NAT*  
*SOL\_SOCKET*  
*SOL\_TCP*  
*SOL\_UDP*  
*SOL\_IP*

*optname* [in] — Option name (see [Section](#) , “Description”).

*optval* [in] — Pointer to the option value.

*optlen* [in] — Number of bytes that *optval* points to.

### Return Value

- *RTCS\_OK* (success)
- Specific error code (failure)

### See Also

- [bind\(\)](#)
- [getsockopt\(\)](#)
- [ip\\_mreq](#)
- [nat\\_ports](#)
- [nat\\_timeouts](#)



## Description

You can set most socket options by calling **setsockopt()**. However, the following options cannot be set; you can use them only with **getsockopt()**:

- IGMP get membership
- receive Ethernet 802.1Q priority tags
- receive Ethernet 802.3 frames
- socket error
- socket type

The user-changeable options have default values. If you want to change the value of some of the options, you must do so before you bind the socket. For other options, you can change the value anytime after the socket is created.

This function blocks, but the command is immediately serviced and replied to.

<b>NOTE</b>	Some options can be temporarily overridden for datagram sockets. For more information, see <b>send()</b> and <b>sendto()</b> .
-------------	--

## Options

This section describes the socket options.

### Checksum Bypass

<b>Option name</b>	<i>OPT_CHECKSUM_BYPASS</i> (can be overridden)
<b>Protocol level</b>	<i>SOL_UDP</i>
<b>Values</b>	<ul style="list-style-type: none"> <li>• TRUE (RTCS sets the checksum field of sent datagram packets to zero, and the generation of checksums is bypassed).</li> <li>• FALSE (RTCS generates checksums for sent datagram packets).</li> </ul>
<b>Default value</b>	FALSE
<b>Change</b>	Before bound
<b>Socket type</b>	Datagram
<b>Comments</b>	—

### Connect Timeout

<b>Option name</b>	<i>OPT_CONNECT_TIMEOUT</i>
<b>Protocol level</b>	<i>SOL_TCP</i>
<b>Values</b>	≥ 180,000 (RTCS maintains the connection for this number of milliseconds).
<b>Default value</b>	480,000 (eight minutes).

<b>Change</b>	Before bound
<b>Socket type</b>	Stream
<b>Comments</b>	Connect timeout corresponds to R2 (as defined in RFC 793) and is sometimes called the hard timeout. It indicates how much time RTCS spends attempting to establish a connection before it gives up. If the remote endpoint does not acknowledge a sent segment within the connect timeout (as would happen if a cable breaks, for example), RTCS shuts down the socket connection, and all function calls that use the connection return.

## Receive Wait/Nowait

<b>Option name</b>	<i>OPT_RECEIVE_NOWAIT</i>
<b>Protocol level</b>	<i>SOL_UDP</i>
<b>Values</b>	<ul style="list-style-type: none"> <li>• TRUE (<b>recv()</b> and <b>recvfrom()</b> return immediately, regardless of whether data to be received is present).</li> <li>• FALSE (<b>recv()</b> and <b>recvfrom()</b> wait until data to be received is present).</li> </ul>
<b>Default value</b>	FALSE
<b>Change</b>	Anytime
<b>Socket type</b>	Datagram
<b>Comments</b>	—

## IGMP Add Membership

<b>Option name</b>	<i>RTCS_SO_IGMP_ADD_MEMBERSHIP</i>
<b>Protocol level</b>	<i>SOL_IGMP</i>
<b>Values</b>	—
<b>Default value</b>	Not in a group
<b>Change</b>	Anytime
<b>Socket type</b>	Datagram
<b>Comments</b>	<p>IGMP must be in the RTCS protocol table.</p> <p>To join a multicast group:</p> <pre>uint_32      sock; struct ip_mreq group;  group.imr_multiaddr = multicast_ip_address; group.imr_interface = local_ip_address; error = setsockopt(sock, SOL_IGMP,     RTCS_SO_IGMP_ADD_MEMBERSHIP, &amp;group,     sizeof(group));</pre>

## IGMP Drop Membership

<b>Option name</b>	<i>RTCS_SO_IGMP_DROP_MEMBERSHIP</i>
<b>Protocol level</b>	<i>SOL_IGMP</i>
<b>Values</b>	—
<b>Default value</b>	Not in a group
<b>Change</b>	After the socket is created
<b>Socket type</b>	Datagram
<b>Comments</b>	<p>IGMP must be in the RTCS protocol table.</p> <p>To leave a multicast group:</p> <pre>uint_32      sock; struct ip_mreq group;  group.imr_multiaddr = multicast_ip_address; group.imr_interface = local_ip_address; error = setsockopt(sock, SOL_IGMP,     RTCS_SO_IGMP_DROP_MEMBERSHIP, &amp;group,     sizeof(group));</pre>

## IGMP Get Membership

<b>Option name</b>	<i>RTCS_SO_IGMP_GET_MEMBERSHIP</i>
<b>Protocol level</b>	<i>SOL_IGMP</i>
<b>Values</b>	—
<b>Default value</b>	Not in a group
<b>Change</b>	— (use with <a href="#">getsockopt()</a> only; returns value in <i>optval</i> ).
<b>Socket type</b>	Datagram
<b>Comments</b>	—

### Initial Retransmission Timeout

<b>Option name</b>	<i>OPT_RETRANSMISSION_TIMEOUT</i>
<b>Protocol level</b>	<i>SOL_TCP</i>
<b>Values</b>	$\geq 15$ ms (see comments)
<b>Default value</b>	3000 (three seconds)
<b>Change</b>	Before bound
<b>Socket type</b>	Stream
<b>Comments</b>	Value is a first, best guess of the round-trip time for a stream socket packet. RTCS attempts to resend the packet, if it does not receive an acknowledgment in this time. After a connection is established, RTCS determines the retransmission timeout, starting from this initial value. If the initial retransmission timeout is not longer than the end-to-end acknowledgment time expected on the socket, the connect timeout will expire prematurely.

### Keep-Alive Timeout

<b>Option name</b>	<i>OPT_KEEPAIVE</i>
<b>Protocol level</b>	<i>SOL_TCP</i>
<b>Values</b>	<ul style="list-style-type: none"> <li>Zero (RTCS does not probe the remote endpoint).</li> <li>Non-zero (if the connection is idle, RTCS periodically probes the remote endpoint, an action that detects, whether the remote endpoint is still present).</li> </ul>
<b>Default value</b>	Zero minutes
<b>Change</b>	Before bound
<b>Socket type</b>	Stream
<b>Comments</b>	The option is not a standard feature of the TCP/IP specification and generates unnecessary periodic network traffic.

## Maximum Retransmission Timeout

<b>Option name</b>	<i>OPT_MAXRTO</i>
<b>Protocol level</b>	<i>SOL_TCP</i>
<b>Values</b>	<ul style="list-style-type: none"><li>• Non-zero (maximum value for the retransmission timer's exponential backoff).</li><li>• Zero (RTCS uses the default value, which is 2 times the maximum segment lifetime [MSL]. Since the MSL is 2 minutes, the MTO is 4 minutes)</li></ul>
<b>Default value</b>	Zero milliseconds
<b>Change</b>	Before bound
<b>Socket type</b>	Stream
<b>Comments</b>	The retransmission timer is used for multiple retransmissions of a segment.

## NAT Inactivity Timeout

<b>Option name</b>	<i>RTCS_SO_NAT_TIMEOUTS</i>
<b>Protocol level</b>	<i>SOL_NAT</i>
<b>Values</b>	See comments
<b>Default value</b>	See comments
<b>Change</b>	After the socket is created
<b>Socket type</b>	Datagram or stream
<b>Comments</b>	An application-supplied <i>nat_timeouts</i> structure defines inactivity timeout values.

## NAT Port Numbers

<b>Option name</b>	<i>RTCS_SO_NAT_PORTS</i>
<b>Protocol level</b>	<i>SOL_NAT</i>
<b>Values</b>	See comments
<b>Default value</b>	See comments
<b>Change</b>	After the socket is created
<b>Socket type</b>	Datagram or stream
<b>Comments</b>	An application-supplied <i>nat_ports</i> structure defines port numbers.

## No Nagle Algorithm

<b>Option name</b>	<i>OPT_NO_NAGLE_ALGORITHM</i>
<b>Protocol level</b>	<i>SOL_TCP</i>
<b>Values</b>	<ul style="list-style-type: none"> <li>• TRUE (RTCS does not use the Nagle algorithm to coalesce short segments).</li> <li>• FALSE (to reduce network congestion, RTCS uses the Nagle algorithm [defined in RFC 896] to coalesce short segments).</li> </ul>
<b>Default value</b>	FALSE
<b>Change</b>	Before bound
<b>Socket type</b>	Stream
<b>Comments</b>	If an application intentionally sends short segments, it can improve efficiency by setting the option to TRUE.

## Receive Ethernet 802.1Q Priority Tags

Option name	<i>RTCS_SO_LINK_RX_8021Q_PRIO</i>
Protocol level	<i>SOL_LINK</i>
Values	<ul style="list-style-type: none"> <li>• -1 (last received frame did not have an Ethernet 802.1Q priority tag).</li> <li>• 0..7 (last received frame had an Ethernet 802.1Q priority tag with the specified priority).</li> </ul>
Default value	—
Change	— (use with <a href="#">getsockopt()</a> only; returns value in <i>optval</i> ).
Socket type	Stream (Ethernet)
Comments	Returned information is for the last frame that the socket received.

## Receive Ethernet 802.3 Frames

Option name	<i>RTCS_SO_LINK_RX_8023</i>
Protocol level	<i>SOL_LINK</i>
Values	<ul style="list-style-type: none"> <li>• TRUE (last received frame was an 802.3 frame).</li> <li>• FALSE (last received frame was an Ethernet II frame).</li> </ul>
Default value	—
Change	— (use with <a href="#">getsockopt()</a> only; returns value in <i>optval</i> )
Socket type	Stream (Ethernet)
Comments	Returned information is for the last frame that the socket received.

## Receive Nowait

Option name	<i>OPT_RECEIVE_NOWAIT</i>
Protocol level	<i>SOL_TCP</i>
Values	<ul style="list-style-type: none"> <li>• TRUE (<a href="#">recv()</a> returns immediately, regardless of whether there is data to be received).</li> <li>• FALSE (<a href="#">recv()</a> waits until there is data to be received).</li> </ul>
Default value	FALSE
Change	Anytime
Socket type	Stream
Comments	—

## Receive Push

Option name	<i>OPT_RECEIVE_PUSH</i>
Protocol level	<i>SOL_TCP</i>
Values	<ul style="list-style-type: none"> <li>TRUE (<b>recv()</b>) returns immediately if it receives a push flag from the remote endpoint, even if the specified receive buffer is not full).</li> <li>FALSE (<b>recv()</b>) ignores push flags and returns only when its buffer is full, or if the receive timeout expires).</li> </ul>
Default value	TRUE
Change	Anytime
Socket type	Stream
Comments	—

## Receive Timeout

Option name	<i>OPT_RECEIVE_TIMEOUT</i>
Protocol level	<i>SOL_TCP</i>
Values	<ul style="list-style-type: none"> <li>Zero (RTCS waits indefinitely for incoming data during a call to <b>recv()</b>).</li> <li>Non-zero (RTCS waits for this number of milliseconds for incoming data during a call to <b>recv()</b>).</li> </ul>
Default value	Zero milliseconds
Change	Anytime
Socket type	Stream
Comments	When the timeout expires, <b>recv()</b> returns with whatever data that has been received.

## Receive-Buffer Size

Option name	<i>OPT_RBSIZE</i>
Protocol level	<i>SOL_TCP</i>
Values	Recommended to be a multiple of the maximum segment size, where the multiple is at least three.
Default value	4380 bytes
Change	Before bound
Socket type	Stream
Comments	When the socket is bound, RTCS allocates a receive buffer of the specified number of bytes, which controls how much received data RTCS can buffer for the socket.



## Send Ethernet 802.1Q Priority Tags

<b>Option name</b>	<i>RTCS_SO_LINK_TX_8021Q_PRIO</i>
<b>Protocol level</b>	<i>SOL_LINK</i>
<b>Values</b>	<ul style="list-style-type: none"> <li>–1 (RTCS does not include Ethernet 802.1Q priority tags)</li> <li>0..7 (RTCS includes Ethernet 802.1Q priority tags with the specified priority)</li> </ul>
<b>Default value</b>	–1
<b>Change</b>	Anytime
<b>Socket type</b>	Stream (Ethernet)
<b>Comments</b>	—

## Send Ethernet 802.3 Frames

<b>Option name</b>	<i>RTCS_SO_LINK_TX_8023</i>
<b>Protocol level</b>	<i>SOL_LINK</i>
<b>Values</b>	<ul style="list-style-type: none"> <li>TRUE (RTCS sends 802.3 frames).</li> <li>FALSE (RTCS sends Ethernet II frames).</li> </ul>
<b>Default value</b>	FALSE
<b>Change</b>	Anytime
<b>Socket type</b>	Stream (Ethernet)
<b>Comments</b>	Returns information for the last frame that the socket received.

## Send Nowait (Datagram Socket)

<b>Option name</b>	<i>OPT_SEND_NOWAIT</i> (can be overridden)
<b>Protocol level</b>	<i>SOL_UDP</i>
<b>Values</b>	<ul style="list-style-type: none"> <li>TRUE (RTCS buffers every datagram and <b>send()</b> or <b>sendto()</b> returns immediately).</li> <li>FALSE (task that calls <b>send()</b> or <b>sendto()</b> blocks until the datagram has been transmitted; datagrams are not copied).</li> </ul>
<b>Default value</b>	FALSE
<b>Change</b>	Anytime
<b>Socket type</b>	Datagram
<b>Comments</b>	—

## Send Nowait (Stream Socket)

Option name	<i>OPT_SEND_NOWAIT</i>
Protocol level	<i>SOL_TCP</i>
Values	<ul style="list-style-type: none"> <li>TRUE (task that calls <b>send()</b> does not wait if data is waiting to be sent; RTCS buffers the outgoing data, and <b>send()</b> returns immediately).</li> <li>FALSE (task that calls <b>send()</b> waits if data is waiting to be sent).</li> </ul>
Default value	FALSE
Change	Anytime
Socket type	Stream
Comments	—

## Send Push

Option name	<i>OPT_SEND_PUSH</i>
Protocol level	<i>SOL_TCP</i>
Values	<ul style="list-style-type: none"> <li>TRUE (if possible, RTCS appends a send-push flag to the last packet in the segment of the data that is associated with <b>send()</b> and immediately sends the data. A call to <b>send()</b> might block until another task calls <b>send()</b> for that socket).</li> <li>FALSE (before it sends a packet, RTCS waits until it has received enough data from the host to completely fill the packet).</li> </ul>
Default value	TRUE
Change	Anytime
Socket type	Stream
Comments	—

## Send Timeout

Option name	<i>OPT_SEND_TIMEOUT</i>
Protocol level	<i>SOL_TCP</i>
Values	<ul style="list-style-type: none"> <li>Zero (RTCS waits indefinitely for outgoing data during a call to <b>send()</b>).</li> <li>Non-zero (RTCS waits for this number of milliseconds for incoming data during a call to <b>send()</b>).</li> </ul>
Default value	Four minutes
Change	Anytime
Socket type	Stream
Comments	When the timeout expires, <b>send()</b> returns

## Send-Buffer Size

<b>Option name</b>	<i>OPT_TBSIZE</i>
<b>Protocol level</b>	<i>SOL_TCP</i>
<b>Values</b>	Recommended to be a multiple of the maximum segment size, where the multiple is at least three.
<b>Default value</b>	4380 bytes
<b>Change</b>	Before bound
<b>Socket type</b>	Stream
<b>Comments</b>	When the socket is bound, RTCS allocates a send buffer of the specified number of bytes, which controls how much sent data RTCS can buffer for the socket.

## Socket Error

<b>Option name</b>	<i>OPT_SOCKET_ERROR</i>
<b>Protocol level</b>	<i>SOL_SOCKET</i>
<b>Values</b>	—
<b>Default value</b>	—
<b>Change</b>	— (use with <a href="#">getsockopt()</a> only; returns value in <i>optval</i> )
<b>Socket type</b>	Datagram or stream
<b>Comments</b>	Returns the last error for the socket.

## Socket Type

<b>Option name</b>	<i>OPT_SOCKET_TYPE</i>
<b>Protocol level</b>	<i>SOL_SOCKET</i>
<b>Values</b>	—
<b>Default value</b>	—
<b>Change</b>	— (use with <a href="#">getsockopt()</a> only; returns value in <i>optval</i> )
<b>Socket type</b>	Datagram or stream
<b>Comments</b>	Returns the type of socket ( <i>SOCK_DGRAM</i> or <i>SOCK_STREAM</i> ).

## Timewait Timeout

<b>Option name</b>	<i>OPT_TIMEWAIT_TIMEOUT</i>
<b>Protocol level</b>	<i>SOL_TCP</i>
<b>Values</b>	> Zero milliseconds
<b>Default value</b>	Two times the maximum segment lifetime (which is a constant).
<b>Change</b>	Before bound
<b>Socket type</b>	Stream
<b>Comments</b>	Returned information is for the last frame that the socket received.

## RX Destination Address

<b>Option name</b>	RTCS_SO_IP_RX_DEST
<b>Protocol level</b>	<i>SOL_IP</i>
<b>Values</b>	—
<b>Default value</b>	—
<b>Change</b>	— (use with <a href="#">getsockopt()</a> only; returns value in <i>optval</i> ).
<b>Socket type</b>	Datagram or stream
<b>Comments</b>	Returns destination address of the last frame that the socket received.

## Time to Live - RX

<b>Option name</b>	RTCS_SO_IP_RX_TTL
<b>Protocol level</b>	<i>SOL_IP</i>
<b>Values</b>	—
<b>Default value</b>	—
<b>Change</b>	— (use with <a href="#">getsockopt()</a> only; returns value in <i>optval</i> ).
<b>Socket type</b>	Datagram or stream
<b>Comments</b>	Gets the TTL (time to live) field of incoming packets. Returned information is for the last frame that the socket received.

## Type of Service

<b>Option name</b>	RTCS_SO_IP_RX_TOS
<b>Protocol level</b>	<i>SOL_IP</i>
<b>Values</b>	—
<b>Default value</b>	—

<b>Change</b>	— (use with <a href="#">getsockopt()</a> only; returns value in <i>optval</i> ).
<b>Socket type</b>	Datagram or stream
<b>Comments</b>	Returns the TOS (type of service) field of incoming packets. Returned information is for the last frame that the socket received.

### Time to Live - TX

<b>Option name</b>	RTCS_SO_IP_TX_TTL
<b>Protocol level</b>	<i>SOL_IP</i>
<b>Values</b>	TTL field of the IP header in outgoing datagrams
<b>Default value</b>	64
<b>Change</b>	Anytime
<b>Socket type</b>	Datagram or stream
<b>Comments</b>	Sets or gets the TTL (time to live) field of outgoing packets.

### Local Address

<b>Option name</b>	RTCS_SO_IP_LOCAL_ADDR
<b>Protocol level</b>	<i>SOL_IP</i>
<b>Values</b>	—
<b>Default value</b>	—
<b>Change</b>	— (use with <a href="#">getsockopt()</a> only; returns value in <i>optval</i> ).
<b>Socket type</b>	Datagram or stream
<b>Comments</b>	Returns local IP address.

## Examples

**Example 7-1. Changing the Send-Push Option to FALSE**


---

```

uint_32  handle;
uint_32  opt_length = sizeof(uint_32);
uint_32  opt_value = FALSE;
uint_32  status;
...
status = setsockopt(handle, 0, OPT_SEND_PUSH,
                    &opt_value, opt_length);
if (status != RTCS_OK)
    printf("\nsetsockopt() failed with error %lx", status);

status = getsockopt(handle, 0, OPT_SEND_PUSH,
                    &opt_value, (uint_32_ptr *)&opt_length);
if (status != RTCS_OK)
    printf("\ngetsockopt() failed with error %lx", status);

```

**Example 7-2. Changing the Receive-Nowait Option to TRUE**


---

```

uint_32  handle;
uint_32  opt_length = sizeof(uint_32);
uint_32  opt_value = TRUE;
uint_32  status;
...
status = setsockopt(handle, 0, OPT_RECEIVE_NOWAIT,
                    &opt_value, opt_length);
if (status != RTCS_OK)
    printf("\nError, setsockopt() failed with error %lx", status);

```

**Example 7-3. Changing the Checksum-Bypass Option to TRUE**


---

```

uint_32  handle;
uint_32  opt_length = sizeof(uint_32);
uint_32  opt_value = TRUE;
uint_32  status;
...
status = setsockopt(handle, SOL_UDP, OPT_CHECKSUM_BYPASS,
                    &opt_value, opt_length);
if (status != RTCS_OK)
    printf("\nError, setsockopt() failed with error %lx", status);

```

**Example 7-4. Changing Maximum Port Number Option**


---

Change the maximum port number used by Freescale MQX NAT to 30000 and do not change the minimum port number.

```

nat_ports  ports;
uint_32    error;

ports.port_min = 0;           /* No modification */
ports.port_max = 30000;

error = setsockopt(RTCS_SOCKET_ERROR, SOL_NAT, RTCS_SO_NAT_PORTS,
                  &ports, sizeof(ports));

```

Change the TCP and UDP inactivity timeouts  
 Change the TCP and UDP inactivity timeout values and do not change the FIN timeout value.

```

nat_timeouts    nat_touts;
uint_32         error;

nat_touts.timeout_tcp = 700000; /* Time in milliseconds */
nat_touts.timeout_udp = 500000; /* Time in milliseconds */
nat_touts.timeout_fin = 0;      /* No modification */

error = setsockopt(RTCS_SOCKET_ERROR, SOL_NAT,
                  RTCS_SO_nat_timeouts, &nat_touts,
                  sizeof(nat_touts));

```

### Example 7-5. Changing the TX TTL

---

```

uint_32  handle;
uint_32  status;
uint_8   opt_value = 64;
...
status = setsockopt(handle, SOL_IP, RTCS_SO_IP_TX_TTL,
                   (void *)&opt_value, sizeof(opt_value));
if (status != RTCS_OK)
    printf("\nError, setsockopt() failed with error %lx", status);

```

## 7.1.144 shutdown()

Shuts down the socket.

### Synopsis

```
uint_32 shutdown(
    uint_32 socket,
    uint_16 how)
```

### Parameters

*socket* [*in*] — Handle of the socket to shut down.

*how* [*in*] — One of the following (see description):

*FLAG\_CLOSE\_TX*

*FLAG\_ABORT\_CONNECTION*

### Description

Note that after calling **shutdown()**, the application can no longer use *socket*.

The **shutdown()** blocks, but the command is processed and returns immediately.

Type of socket	Value of <i>how</i>	Action
Datagram	Ignored	<ul style="list-style-type: none"> <li>Shuts down <i>socket</i> immediately.</li> <li>Calls to <b>recvfrom()</b> return immediately.</li> <li>Discards queued incoming packets.</li> </ul>
Unconnected stream	Ignored	Shuts down <i>socket</i> immediately.
Connected stream	FLAG_CLOSE_TX	<ul style="list-style-type: none"> <li>Gracefully shuts down <i>socket</i>, ensuring that all sent data is acknowledged.</li> <li>Calls to <b>send()</b> and <b>recv()</b> return immediately.</li> <li>If RTCS is originating the disconnection, it maintains the internal socket context for four minutes (twice the maximum TCP segment lifetime) after the remote endpoint closes the connection.</li> </ul>
	FLAG_ABORT_CONNECTION	<ul style="list-style-type: none"> <li>Immediately discards the internal socket context.</li> <li>Sends a TCP reset packet to the remote endpoint.</li> <li>Calls to <b>send()</b> and <b>recv()</b> return immediately.</li> </ul>

### Return Value

- RTCS\_OK*
- Specific error code



## See Also

- [socket\(\)](#)

## Example

```
uint_32  handle;  
uint_32  status;  
...  
status = shutdown(handle, 0);  
if (status != RTCS_OK)  
    printf("\nError, shutdown() failed with error code %lx",  
          status);
```

## 7.1.145 SNMP\_init()

Starts SNMP Agent.

### Synopsis

```
uint_32  SNMP_init(
    char_ptr  name,
    uint_32   priority
    uint_32   stacksize)
```

### Parameters

*name* [in] — Name of the SNMP Agent task.

*priority* [in] — Priority of the SNMP Agent task (we recommend that you make the priority lower than the priority of the RTCS task; that is, make it a higher number).

*stacksize* [in] — Stack size for the SNMP Agent task.

### Description

This function starts the SNMP Agent and creates the SNMP task.

### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

### See Also

- [MIB1213\\_init\(\)](#)

### Example

```
uint_32  error;

/* register the RFC1213 MIB */
MIB1213_init();

/* Start SNMP Agent: */
error = SNMP_init("SNMP agent", 7, 1000);
if (error)
    return error;

printf("\nSNMP Agent is running");
```

## 7.1.146 SNMP\_trap\_warmStart()

### Synopsis

```
void SNMP_trap_warmStart(void)
```

### Description

This function sends a warm start trap type 1/0. SNMP trap version 1.

### Return Value

### See Also

- [SNMPv2\\_trap\\_warmStart\(\)](#)

## 7.1.147 SNMP\_trap\_coldStart()

### Synopsis

```
void SNMP_trap_coldStart(void)
```

### Description

This function sends a cold start trap type 0/0. SNMP trap version 1.

### Return Value

### See Also

- [SNMPv2\\_trap\\_coldStart\(\)](#)

## 7.1.148 SNMP\_trap\_authenticationFailure()

### Synopsis

```
void SNMP_trap_authenticationFailure(void)
```

### Description

This function sends an authentication failure trap type 4/0. SNMP trap version 1.

### Return Value

### See Also

- [SNMPv2\\_trap\\_authenticationFailure\(\)](#)

## 7.1.149 SNMP\_trap\_linkDown()

### Synopsis

```
void SNMP_trap_linkDown(pointer ihandle)
```

### Parameters

*ihandle [in]* — interface index

### Description

This function sends a link down trap type 2/0. SNMP trap version 1.

### Return Value

### See Also

- [SNMPv2\\_trap\\_linkDown\(\)](#)

## 7.1.150 SNMP\_trap\_myLinkDown()

### Synopsis

```
void SNMP_trap_myLinkDown(pointer ihandle)
```

### Parameters

*ihandle [in]* — enterprise specific interface index

### Description

This function sends a link down trap type 2/0 for enterprise specific device. SNMP trap version 1.

### Return Value

### See Also

- [SNMPv2\\_trap\\_linkDown\(\)](#)

## 7.1.151 SNMP\_trap\_linkUp()

### Synopsis

```
void SNMP_trap_linkUp(pointer ihandle)
```

### Parameters

*ihandle [in]* — interface index

### Description

This function sends a link up trap type 3/0. SNMP trap version 1.

### Return Value

### See Also

- [SNMPv2\\_trap\\_linkUp\(\)](#)



## 7.1.152 SNMP\_trap\_userSpec()

### Synopsis

```
void SNMP_trap_userSpec(  
    RTCSMIB_NODE_PTR trap_node,  
    uint_32 spec_trap,  
    RTCSMIB_NODE_PTR enterprises)
```

### Parameters

*trap\_node* [in] — user specific trap node

*spec\_trap* [in] — user specific trap type

*enterprises* [in] — enterprises node

### Description

This function sends user specified trap 6/spec\_trap type 1 message.

### Return Value

### See Also

- [SNMP\\_trap\\_userSpec\(\)](#)

## 7.1.153 SNMPv2\_trap\_warmStart()

### Synopsis

```
void SNMPv2_trap_warmStart(void)
```

### Description

This function sends warm start trap type 2 message.

### Return Value

### See Also

- [SNMP\\_trap\\_warmStart\(\)](#)

## 7.1.154 SNMPv2\_trap\_coldStart()

### Synopsis

```
void SNMPv2_trap_coldStart(void)
```

### Description

This function sends cold start trap type 2 message.

### Return Value

### See Also

- [SNMP\\_trap\\_coldStart\(\)](#)

## 7.1.155 SNMPv2\_trap\_authenticationFailure()

### Synopsis

```
void SNMPv2_trap_authenticationFailure(void)
```

### Description

This function sends authentication failure trap type 2 message.

### Return Value

### See Also

- [SNMP\\_trap\\_authenticationFailure\(\)](#)

## 7.1.156 SNMPv2\_trap\_linkDown()

### Synopsis

```
void SNMPv2_trap_linkDown(pointer ihandle)
```

### Parameters

*ihandle* [in] — interface index

### Description

This function sends link down trap type 2 message.

### Return Value

### See Also

- [SNMP\\_trap\\_linkDown\(\)](#)

## 7.1.157 SNMPv2\_trap\_linkUp()

### Synopsis

```
void SNMPv2_trap_linkUp(pointer ihandle)
```

### Parameters

*ihandle* [in] — interface index

### Description

This function sends link up trap type 2 message.

### Return Value

### See Also

- [SNMP\\_trap\\_linkUp\(\)](#)

## 7.1.158 SNMPv2\_trap\_userSpec()

### Synopsis

```
void SNMPv2_trap_userSpec(  
    RTCSMIB_NODE_PTR trap_node)
```

### Parameters

*trap\_node [in]* — user specific trap node

### Description

This function sends user specified trap type 2 message.

### Return Value

### See Also

- [SNMP\\_trap\\_userSpec\(\)](#)

## 7.1.159 SNTP\_init()

Starts the SNTP Client task.

### Synopsis

```
uint_32 SNTP_init(
    char_ptr      name,
    uint_32       priority,
    uint_32       stacksize,
    _ip_address    destination,
    uint_32       poll)
```

### Parameters

*name* [in] — Name of the SNTP Client task.

*priority* [in] — Priority of SNTP Client task (we recommend that you make the priority lower than the priority of the RTCS task; that is, make it a higher number).

*stacksize* [in] — Stack size for the SNTP Client task.

*destination* [in] — Where SNTP time requests are sent. One of the following:

- IP address of the time server (unicast mode).
- A local broadcast address or multicast group (anycast mode).

*poll* [in] — Time to wait between time updates (must be between one and 4294967 seconds).

### Description

The function starts the SNTP Client task that will first update the local time, and then wait for a number of seconds as specified by *poll*. Once this time has expired, the SNTP Client repeats the same cycle. The local time is set in UTC (coordinated universal time).

The SNTP Client task works in unicast or anycast mode.

### Return Value

- *RTCS\_OK* (success).
- *RTCSERR\_INVALID\_PARAMETER* (failure) resulting from either *destination* not being specified, or *poll* is out of range.
- Specific error code (failure) resulting from **socket()** and **bind()** calls.

### See Also

- [socket\(\)](#)
- [bind\(\)](#)
- [SNTP\\_oneshot\(\)](#)

### Example

```
uint_32 error;

/*
** Start the SNTP Client task with the following settings:
** Task Name: SNTP Client
** Priority: 7
** Stacksize: 1000
```



```
** Server address: 142.123.203.66 = 0x8E7BCB42
** Poll interval: every 100 seconds
*/

error = SNTP_init("SNTP client", 7, 1000, 0x8E7BCB42, 100);
if (error) return error;
printf("The SNTP client task is running");
return 0;
```

## 7.1.160 SNTP\_oneshot()

Sets the time in UTC time using the SNTP protocol.

### Synopsis

```
uint_32 SNTP_oneshot(  
    _ip_address  destination,  
    uint_32      timeout)
```

### Parameters

*destination* [in] — Where SNTP time requests are sent. One of:

- IP address of the time server (unicast mode).
- a local broadcast address or multicast group (anycast mode).

*timeout* [in] — Amount of time (in milliseconds) to continue trying to obtain the time using SNTP.

### Description

This function sends an SNTP packet and waits for a reply. If a reply is received before *timeout* elapses, the time is set. If no reply is received within the specified time, *RTCSERR\_TIMEOUT* is returned. The local time is set in UTC (coordinated universal time).

The SNTP Client task works in unicast or anycast mode.

### Return Value

- *RTCS\_OK* (success).
- *RTCSERR\_INVALID\_PARAMETER* (failure) resulting from *destination* not being specified.
- *RTCSERR\_TIMEOUT* (failure) due to expiry of *timeout* value before SNTP could successfully receive the time.
- Error code (failure).

### See Also

- [SNTP\\_init\(\)](#)

## 7.1.161 socket()

Creates the socket.

### Synopsis

```
uint_32  socket(  
    uint_16  protocol_family,  
    uint_16  type,  
    uint_16  protocol)
```

### Parameters

*protocol\_family* [in] — Protocol family; must be *PF\_INET* (protocol family, IP addressing).

*type* [in] — Type of socket; one of the following:

*SOCK\_STREAM*

*SOCK\_DGRAM*

*protocol* [in] — Unused

### Description

The application uses the socket handle to subsequently use the socket. This function blocks, although the command is serviced and responded to immediately.

### Return Value

- Socket handle (success)
- *RTCS\_SOCKET\_ERROR* (failure)

### See Also

- [bind\(\)](#)

### Example

See [bind\(\)](#).

## 7.1.162 TCP\_stats()

Gets a pointer to TCP statistics.

### Synopsis

```
TCP_STATS_PTR TCP_stats(void)
```

### Description

Function **TCP\_stats()** takes no parameters. It returns the TCP statistics that RTCS collects.

### Return Value

Pointer to the *TCP\_STATS* structure.

### See Also

- [ARP\\_stats\(\)](#)
- [ENET\\_get\\_stats\(\)](#)
- [ICMP\\_stats\(\)](#)
- [IGMP\\_stats\(\)](#)
- [IP\\_stats\(\)](#)
- [IPIF\\_stats\(\)](#)
- [UDP\\_stats\(\)](#)
- *TCP\_STATS*

### Example

See [ARP\\_stats\(\)](#).

### 7.1.163 TELNET\_connect()

Starts Telnet Client, which starts the shell that accepts a command to start a Telnet session with a Telnet server.

#### Synopsis

```
uint_32  TELNET_connect(  
    _ip_address  ipaddress)
```

#### Parameters

*ipaddress [in]* — IP address to connect to.

#### Description

If a user enters *telnet* at the shell prompt, the shell prompts for the IP address of a Telnet server. The Telnet client creates a stream socket, binds it, and connects it to Telnet server. When the socket is connected, the client sends to the server any characters that the user types and displays on the console any characters that it receives from the server.

#### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

## 7.1.164 TELNETSRV\_init()

Starts the Telnet Server.

### Synopsis

```
uint_32  TELNETSRV_init(
    char_ptr      name,
    uint_32       priority,
    uint_32       stacksize,
    RTCS_TASK_PTR shell)
```

### Parameters

*name* [in] — Name of Telnet Server task.

*priority* [in] — Priority of Telnet Server task (we recommend that you make the priority lower than the priority of the RTCS task; that is, make it a higher number).

*stacksize* [in] — Stack size for Telnet Server task.

*shell* [in] — Shell task that Telnet Server starts when a client initiates a connection (see description).

### Description

Function **TELNETSRV\_init()** starts Telnet Server and creates *TELNETSRV\_task*.

Telnet Server listens on a stream socket. Every time a client initiates a connection, the server creates a new shell task and redirects the new task's I/O to the connected socket.

Command processing is done by the specified shell, which may be the Shell function provided. When using the Shell function, an alternate command list may be specified in order to restrict the commands available remotely.

The Telnet server may be started or stopped from the shell, by including the *Shell\_Telnetd* function in the shell command list.

```
#include <rtcs.h>
#include "shell.h"
#include "sh_rtcs.h"

#define SHELL_TELNETD_PRIO      7
#define SHELL_TELNETD_STACK    1000

// A restricted list of shell commands
SHELL_COMMAND_STRUCT Telnetd_shell_commands [] = {
    { "cd",          Shell_cd },
    { "dir",         Shell_dir },
    { "exit",        Shell_exit },
    { "ftp",         Shell_FTP_client },
    { "gethbn",      Shell_get_host_by_name },
    { "help",        Shell_help },
    { "netstat",     Shell_netstat },
    { "ping",        Shell_ping },
    { "pwd",         Shell_pwd },
    { "read",        Shell_read },
    { "telnet",      Shell_Telnet_client },
    { "tftp",        Shell_TFTP_client },
```

```

    { "type",      Shell_type },
    { "?",        Shell_command_list },
    { NULL,       NULL }
};

RTCS_TASK Telnetd_shell_template = {"Telnet_shell", 8, 2000, Telnetd_shell_fn, NULL};

void Telnetd_shell_fn (pointer dummy)
{
    Shell(Telnetd_shell_commands, NULL);
}

void main_task( uint_32 temp )

    /* Start the telnet server */
    result = TELNETSRV_init("Telnet_server", SHELL_TELNETD_PRIO,
        SHELL_TELNETD_STACK, &Telnetd_shell_template ); }

```

### Return Value

- *RTCS\_OK* (success)
- Error code (failure)

### See Also

- [TELNET\\_connect\(\)](#)
- [RTCS\\_TASK](#)

## 7.1.165 TFTP\_SRV\_access()

Decides, whether to allow access to a TFTP client.

### Synopsis

```
boolean TFTP_SRV_access(  
    char_ptr  string_ptr,  
    uint_16   request_type)
```

### Parameters

*string\_ptr [in]* — String name that identifies requested device

*request\_type [in]* — Type of access requested; one of the following:

*TFTPOP\_RRQ*

*TFTPOP\_WRQ*

### Description

TFTP Server calls the function every time a TFTP client initiates a read request or a write request. The function that accompanies RTCS allows all read access and denies all write access. If you want to enforce different access restriction, you can supply your own function to override the one that accompanies RTCS.

### Return Value

- TRUE (allow access)
- FALSE (deny access)

### See Also

- [TFTP\\_SRV\\_init\(\)](#)



## 7.1.166 TFTPSPRV\_init()

Starts TFTP Server.

### Synopsis

```
uint_32 TFTPSPRV_init(
    char_ptr    name,
    uint_32     priority,
    uint_32     stacksize)
```

### Parameters

*name* [in] — String name to assign to TFTP Server task.

*priority* [in] — Priority to assign to TFTP Server task (we recommend that you make the priority lower than the priority of the RTCS task; that is, make it a higher number).

*stacksize* [in] — Number of bytes to allocate for the TFTP Server task stack (see description).

### Description

This function creates TFTP Server task and blocks until TFTP Server task has completed its initialization.

We recommend a stack size of at least 1000 bytes. Increase it only if you increase the value of *TFTPSPRV\_MAX\_TRANSACTIONS*, whose default value (20) is defined in *tftp.h*.

### Return Value

- *RTCS\_OK* (success)
- RTCS error code (failure)

### See Also

- [TFTPSPRV\\_access\(\)](#)

### Example

```
uint_32 error;

/* Start TFTP Server: */
error = TFTPSPRV_init("TFTP server", 7, 1000);
if (error) return error;
printf("\nTFTP Server is running.");
return 0;
```

## 7.1.167 UDP\_stats()

Gets a pointer to UDP statistics.

### Synopsis

```
UDP_STATS_PTR  UDP_stats(void)
```

### Description

Function **UDP\_stats()** gets a pointer to the UDP statistics that RTCS collects.

### Return Value

Pointer to the *UDP\_STATS* structure.

### See Also

- [ARP\\_stats\(\)](#)
- [ENET\\_get\\_stats\(\)](#)
- [ICMP\\_stats\(\)](#)
- [IGMP\\_stats\(\)](#)
- [IP\\_stats\(\)](#)
- [IPIF\\_stats\(\)](#)
- [TCP\\_stats\(\)](#)
- [ARP\\_STATS](#)

### Example

See [ARP\\_stats\(\)](#).

## 7.2 Functions Listed by Service

Table 7-2.

Service	Functions
DHCP Client	<a href="#">RTCS_if_bind_DHCP()</a> <a href="#">DHCPCLNT_find_option()</a>
DHCP Server	<a href="#">DHCP*</a> <a href="#">DHCPSRV*</a>
DNS Resolver	<a href="#">DNS_init()</a> <a href="#">gethostbyaddr()</a> <a href="#">gethostbyname()</a>
Echo Server	<a href="#">ECHOSRV_init()</a>
EDS Server (Winsock)	<a href="#">DNS_init()</a>
Ethernet Driver	<a href="#">ENET_get_stats()</a> (part of MQX) <a href="#">ENET_initialize()</a> (part of MQX)
FTP Client	<a href="#">FTP_close()</a> <a href="#">FTP_command()</a> <a href="#">FTP_command_data()</a> <a href="#">FTPd_init()</a>
FTP Server	<a href="#">FTPSRV_init()</a>
HDLC	<a href="#">_iopcb_ppphdlc_init()</a>
HTTP Server	<a href="#">httpd_default_params()</a> <a href="#">httpd_server_init()</a> <a href="#">httpd_server_run()</a> <a href="#">httpd_server_poll()</a>
I/O PCB driver	<a href="#">_iopcb_open()</a> <a href="#">_iopcb_ppphdlc_init()</a> <a href="#">_iopcb_pppoe_client_init()</a>

Table 7-2. (continued)

IPCFG	<a href="#">ipcfg_init_device()</a> <a href="#">ipcfg_init_interface()</a> <a href="#">ipcfg_bind_boot()</a> <a href="#">ipcfg_bind_dhcp()</a> <a href="#">ipcfg_bind_dhcp_wait()</a> <a href="#">ipcfg_bind_staticip()</a> <a href="#">ipcfg_get_device_number()</a> <a href="#">ipcfg_add_interface()</a> <a href="#">ipcfg_get_ihandle()</a> <a href="#">ipcfg_get_mac()</a> <a href="#">ipcfg_get_state()</a> <a href="#">ipcfg_get_state_string()</a> <a href="#">ipcfg_get_desired_state()</a> <a href="#">ipcfg_get_link_active()</a> <a href="#">ipcfg_get_dns_ip()</a> <a href="#">ipcfg_add_dns_ip()</a> <a href="#">ipcfg_del_dns_ip()</a> <a href="#">ipcfg_get_ip()</a> <a href="#">ipcfg_get_tftp_serveraddress()</a> <a href="#">ipcfg_get_tftp_servername()</a> <a href="#">ipcfg_get_boot_filename()</a> <a href="#">ipcfg_poll_dhcp()</a> <a href="#">ipcfg_task_create()</a> <a href="#">ipcfg_task_destroy()</a> <a href="#">ipcfg_task_status()</a> <a href="#">ipcfg_task_poll()</a> <a href="#">ipcfg_unbind()</a>
IWCFG	<a href="#">iwcfg_set_essid()</a> <a href="#">iwcfg_get_essid()</a> <a href="#">iwcfg_commit()</a> <a href="#">iwcfg_set_mode()</a> <a href="#">iwcfg_get_mode()</a> <a href="#">iwcfg_set_wep_key()</a> <a href="#">iwcfg_get_wep_key()</a> <a href="#">iwcfg_set_passphrase()</a> <a href="#">iwcfg_get_passphrase()</a> <a href="#">iwcfg_set_sec_type()</a> <a href="#">iwcfg_get_sectype()</a> <a href="#">iwcfg_set_power()</a> <a href="#">iwcfg_set_scan()</a>
MIB	<a href="#">MIB1213_init()</a>
NAT	<a href="#">NAT_init()</a> <a href="#">NAT_close()</a> <a href="#">NAT_stats()</a>
PPP Driver	<a href="#">PPP_initialize()</a> <a href="#">IPIF_stats()</a>

Table 7-2. (continued)

PPP over Ethernet	_iopcb_pppoe_client_destroy() _iopcb_pppoe_client_init() _pppoe_client_stats() _pppoe_server_destroy() _pppoe_server_if_add() _pppoe_server_if_remove() _pppoe_server_if_stats() _pppoe_server_init() _pppoe_server_session_stats()
RTCS	RTCS_create() RTCS_exec_TFTP_BIN() RTCS_exec_TFTP_COFF() RTCS_exec_TFTP_SREC() RTCS_gate_add() RTCS_gate_remove() RTCS_if_add() RTCS_if_bind() RTCS_if_bind_BOOTP() RTCS_if_bind_DHCP() RTCS_if_bind_IPCP() RTCS_if_remove() RTCS_if_unbind() RTCS_load_TFTP_BIN() RTCS_load_TFTP_COFF() RTCS_load_TFTP_SREC() RTCS_ping() RTCSLOG_disable() RTCSLOG_enable()
SNMP Agent	SNMP_init() SNMP_trap_warmStart() SNMP_trap_coldStart() SNMP_trap_authenticationFailure() SNMP_trap_linkDown() SNMP_trap_myLinkDown() SNMP_trap_linkUp() SNMP_trap_userSpec() SNMPv2_trap_warmStart() SNMPv2_trap_coldStart() SNMPv2_trap_authenticationFailure() SNMPv2_trap_linkDown() SNMPv2_trap_linkUp() SNMPv2_trap_userSpec() MIB1213_init() MIB_find_objectname() MIB_set_objectname()
SNTP Client	SNTP_init() SNTP_oneshot()

Table 7-2. (continued)

Sockets	<b>accept()</b> <b>bind()</b> <b>connect()</b> <b>getpeername()</b> <b>getsockname()</b> <b>getsockopt()</b> <b>listen()</b> <b>recv()</b> <b>recvfrom()</b> <b>RTCS_attachsock()</b> <b>RTCS_detachsock()</b> <b>RTCS_geterror()</b> <b>RTCS_selectall()</b> <b>RTCS_selectset()</b> <b>send()</b> <b>sendto()</b> <b>setsockopt()</b> <b>shutdown()</b> <b>socket()</b>
Statistics	<b>ARP_stats()</b> <b>ENET_get_stats()</b> (part of MQX) <b>ICMP_stats()</b> <b>IGMP_stats()</b> <b>IP_stats()</b> <b>IPIF_stats()</b> <b>NAT_stats()</b> <b>TCP_stats()</b> <b>UDP_stats()</b>
Telnet Client	<b>TELNET_connect()</b>
Telnet Server	<b>TELNETSRV_init()</b>
TFTP Server	<b>TFTPSRV_access()</b> <b>TFTPSRV_init()</b>

## Chapter 8 Data Types

### 8.1 Data Types for Compiler Portability

Name	Bytes	From	To	Description
<b>boolean</b>	4	0	Not zero	Non-zero = TRUE Zero = FALSE
<b>ieee_double</b>	8	2.225074 E-308	1.7976923 E+308	Double-precision IEEE floating-point number
<b>ieee_single</b>	4	8.43E-37	3.37E+38	Single-precision IEEE floating-point number
<b>pointer</b>	4	0	0xFFFFFFFF	Generic pointer
<b>char</b>	1	-128	127	Signed character
<b>char_ptr</b>	4	0	0xFFFFFFFF	Pointer to <b>char</b>
<b>uchar</b>	1	0	255	Unsigned character
<b>uchar_ptr</b>	4	0	0xFFFFFFFF	Pointer to <b>uchar</b>
<b>int_8</b>	1	-128	127	Signed character
<b>int_8_ptr</b>	4	0	0xFFFFFFFF	Pointer to <b>int_8</b>
<b>uint_8</b>	1	0	255	Unsigned character
<b>uint_8_ptr</b>	4	0	0xFFFFFFFF	Pointer to <b>uint_8</b>
<b>int_16</b>	2	-2 <sup>15</sup>	(2 <sup>15</sup> )-1	Signed 16-bit integer
<b>int_16_ptr</b>	4	0	0xFFFFFFFF	Pointer to <b>int_16</b>
<b>uint_16</b>	2	0	(2 <sup>16</sup> )-1	Unsigned 16-bit integer
<b>uint_16_ptr</b>	4	0	0xFFFFFFFF	Pointer to <b>uint_16</b>
<b>int_32</b>	4	-2 <sup>31</sup>	(2 <sup>31</sup> )-1	Signed 32-bit integer
<b>int_32_ptr</b>	4	0	0xFFFFFFFF	Pointer to <b>int_32</b>
<b>uint_32</b>	4	0	(2 <sup>32</sup> )-1	Unsigned 32-bit integer
<b>uint_32_ptr</b>	4	0	0xFFFFFFFF	Pointer to <b>uint_32</b>
<b>int_64</b>	8	-2 <sup>63</sup>	(2 <sup>63</sup> )-1	Signed 64-bit integer
<b>int_64_ptr</b>	4	0	0xFFFFFFFF	Pointer to <b>int_64</b>
<b>uint_64</b>	8	0	(2 <sup>64</sup> )-1	Unsigned 64-bit integer
<b>uint_64_ptr</b>	4	0	0xFFFFFFFF	Pointer to <b>uint_64</b>

## 8.2 Other Data Types

RTCS data type	MQX data type	Defined in	Notes
—	<b>_PTR_</b>	<i>psptypes.h</i> as * (for a particular processor type)	In MQX source
<b>_enet_address</b>	<b>uchar[6]</b>	<i>enet.h</i>	In MQX source
<b>_enet_handle</b>	<b>pointer</b>	<i>enet.h</i>	In MQX source
<b>_ip_address</b>	<b>uint_32</b>	<i>rtcs.h</i>	
<b>_ppp_handle</b>	<b>pointer</b>	<i>ppp.h</i>	
<b>_pppoe_srv_handle</b>	<b>pointer</b>	<i>pppoe.h</i>	
<b>_rtcs_if_handle</b>	<b>pointer</b>	<i>rtcs.h</i>	
<b>_task_id</b>	<b>uint_32</b>	<i>mqx.h</i>	In MQX source
<b>bool_t</b>	<b>boolean</b>	<i>rpctypes.h</i>	
<b>caddr_t</b>	<b>char_ptr</b>	<i>rpctypes.h</i>	
<b>enum_t</b>	<b>uint_16</b> or <b>uint_32</b> (depends on the compiler)	<i>rpctypes.h</i>	
<b>u_char</b>	<b>uchar</b>	<i>rpctypes.h</i>	
<b>u_int</b>	<b>uint_32</b>	<i>rpctypes.h</i>	
<b>u_long</b>	<b>uint_32</b>	<i>rpctypes.h</i>	
<b>u_short</b>	<b>uint_16</b>	<i>rpctypes.h</i>	

## 8.3 Alphabetical List of RTCS Data Structures

This section provides an alphabetical list of RTCS data structures with the following information:

- Function
- Definition
- Fields



### 8.3.1 `_iopcb_handle`, `_iopcb_table`

A variable of `_iopcb_handle` structure is an input parameter to `PPP_initialize()`.

```
typedef struct _iopcb_table {
    uint_32  (_CODE_PTR_  OPEN)  (struct _iopcb_table _PTR_,
                                   void (_CODE_PTR_) (pointer),
                                   void (_CODE_PTR_) (pointer),
                                   pointer);

    uint_32  (_CODE_PTR_  CLOSE) (struct _iopcb_table _PTR_);
    PCB_PTR  (_CODE_PTR_  READ)  (struct _iopcb_table _PTR_,
                                   uint_32);
    void      (_CODE_PTR_  WRITE) (struct _iopcb_table _PTR_,
                                   PCB_PTR,
                                   uint_32);
    uint_32  (_CODE_PTR_  IOCTL) (struct _iopcb_table _PTR_,
                                   uint_32,
                                   pointer);
} _PTR_ _iopcb_handle;
```

#### **OPEN**

Called by PPP Driver to open a link.

- First parameter — pointer to an I/O handle.
- Second parameter — pointer to a function that PPP Driver uses to put the link down.
- Third parameter — pointer to a function that PPP Driver uses to put the link up.
- Fourth parameter — the parameter for the up and down functions.

Returns a status code.

#### **CLOSE**

Called by PPP Driver to close a link and free memory.

- Parameter — pointer to an I/O handle.

Returns a status code.

#### **READ**

Called by PPP Driver to receive data.

- First parameter — pointer to an I/O handle.
- Second parameter — flags (ignored; must be zero).

Returns a pointer to a PCB.

#### **WRITE**

Called by PPP Driver to send data.

- First parameter — pointer to an I/O handle.
- Second parameter — pointer to a PCB to send.

- Third parameter — Flags:
  - Zero: use negotiated options.
  - One: use default HDLC options.

## **IOCTL**

Called by PPP Driver to store and set I/O control commands.

- First parameter — pointer to an I/O handle.
- Second parameter — command to use.
- Third parameter — pointer to the value of the command.

Returns a status code.

### 8.3.2 ARP\_STATS

A pointer to this structure is returned by [ARP\\_stats\(\)](#).

```
typedef struct {
    uint_32      ST_RX_TOTAL;
    uint_32      ST_RX_MISSED;
    uint_32      ST_RX_DISCARDED;
    uint_32      ST_RX_ERRORS;

    uint_32      ST_TX_TOTAL;
    uint_32      ST_TX_MISSED;
    uint_32      ST_TX_DISCARDED;
    uint_32      ST_TX_ERRORS;

    RTCS_ERROR_STRUCT ERR_RX;
    RTCS_ERROR_STRUCT ERR_TX;

    uint_32      ST_RX_REQUESTS;
    uint_32      ST_RX_REPLIES;

    uint_32      ST_TX_REQUESTS;
    uint_32      ST_TX_REPLIES;

    uint_32      ST_ALLOCS_FAILED;
    uint_32      ST_CACHE_HITS;
    uint_32      ST_CACHE_MISSES;
    uint_32      ST_PKT_DISCARDS;
} ARP_STATS, _PTR_ ARP_STATS_PTR;
```

#### ST\_RX\_TOTAL

Received (total).

#### ST\_RX\_MISSED

Received (discarded due to lack of resources).

#### ST\_RX\_DISCARDED

Received (discarded for all other reasons).

#### ST\_RX\_ERRORS

Received (with internal errors).

#### ST\_TX\_TOTAL

Transmitted (total).

#### ST\_TX\_MISSED

Transmitted (discarded due to lack of resources).

#### ST\_TX\_DISCARDED

Transmitted (discarded for all other reasons).

#### ST\_TX\_ERRORS

Transmitted (with internal errors).

**ERR\_RX**

RX error information.

**ERR\_TX**

TX error information.

**ST\_RX\_REQUESTS**

Valid ARP requests received.

**ST\_RX\_REPLIES**

Valid ARP replies received.

**ST\_TX\_REQUESTS**

ARP requests sent.

**ST\_TX\_REPLIES**

ARP replies sent.

**ST\_ALLOCS\_FAILED**

**ARP\_alloc()** returned NULL.

**ST\_CACHE\_HITS**

ARP cache hits.

**ST\_CACHE\_MISSES**

ARP cache misses.

**ST\_PKT\_DISCARDS**

Data packets discarded due to a missing ARP entry.

### 8.3.3 BOOTP\_DATA\_STRUCT

A pointer to this structure is an input parameter to [RTCS\\_if\\_bind\\_BOOTP\(\)](#).

```
typedef struct bootp_data_struct
{
    _ip_address  SADDR;
    uchar        SNAME[64];
    uchar        BOOTFILE[128];
    uchar        OPTIONS[64];
} BOOTP_DATA_STRUCT, _PTR_ BOOTP_DATA_STRUCT_PTR;
```

#### **SADDR**

IP address of the boot file server.

#### **SNAME**

Host name that corresponds to *SADDR*.

#### **BOOTFILE**

Boot file to load.

#### **OPTIONS**

BootP options.

### 8.3.4 DHCP\_DATA\_STRUCT

A pointer to this structure in a parameter to **RTCS\_if\_bind\_DHCP()**.

```
typedef struct {
    int_32      (_CODE_PTR_ CHOICE_FUNC) (uchar _PTR_, uint_32);
    void        (_CODE_PTR_ BIND_FUNC)   (uchar _PTR_, uint_32,
                                           _rtcs_if_handle);
    boolean     (_CODE_PTR_ UNBIND_FUNC) (_rtcs_if_handle);
} DHCP_DATA_STRUCT, _PTR_ DHCP_DATA_STRUCT_PTR;
```

#### CHOICE\_FUNC

Called every time a server receives a DHCP OFFER. If *CHOICE\_FUNC* is NULL, RTCS attempts to bind with the first offer it receives.

- First parameter — pointer to the **OFFER** packet.
- Second parameter — length of the **OFFER** packet.

Returns -1 to reject the packet.

Returns zero to accept the packet.

#### BIND\_FUNC

Called every time DHCP gets a lease. If *BIND\_FUNC* is NULL, RTCS does not modify the behavior of the DHCP Client; the function is for notification purposes only.

- First parameter — pointer to the ACK packet.
- Second parameter — length of the packet.
- Third parameter — handle passed to *RTCS\_if\_bind\_DHCP()*.

#### UNBIND\_FUNC

Called when a lease expires and was not renewed. If *UNBIND\_FUNC* is NULL, RTCS terminates DHCP.

- Parameter — handle passed to **RTCS\_if\_bind\_DHCP()**.

Returns TRUE to attempt to get a new lease.

Returns FALSE to leave the interface unbound.

### 8.3.5 DHCP\_SRV\_DATA\_STRUCT

A pointer to this structure is an input parameter to [DHCP\\_SRV\\_ippool\\_add\(\)](#).

```
typedef struct dhcpsrv_data_struct {  
    _ip_address  SERVERID;  
    uint_32      LEASE;  
    _ip_address  MASK;  
    _ip_address  SADDR;  
    uchar        SNAME[64];  
    uchar        FILE[128];  
} DHCP_SRV_DATA_STRUCT, _PTR_ DHCP_SRV_DATA_STRUCT_PTR;
```

#### SERVERID

IP address of the server.

#### LEASE

Maximum allowable lease length.

#### MASK

Subnet mask.

#### SADDR

*SADDR* field in the DHCP packet header.

#### SNAME

*SNAME* field in the DHCP packet header.

#### FILE

*FILE* field in the DHCP packet header.

### 8.3.6 ENET\_STATS

A pointer to this structure is returned by [ENET\\_get\\_stats\(\)](#).

```
typedef struct {
    uint_32  ST_RX_TOTAL;
    uint_32  ST_RX_MISSED;
    uint_32  ST_RX_DISCARDED;
    uint_32  ST_RX_ERRORS;

    uint_32  ST_TX_TOTAL;
    uint_32  ST_TX_MISSED;
    uint_32  ST_TX_DISCARDED;
    uint_32  ST_TX_ERRORS;
    uint_32  ST_TX_COLLHIST[16];

    uint_32  ST_RX_ALIGN;
    uint_32  ST_RX_FCS;
    uint_32  ST_RX_RUNT;
    uint_32  ST_RX_GIANT;
    uint_32  ST_RX_LATECOLL;
    uint_32  ST_RX_OVERRUN;

    uint_32  ST_TX_SQE;
    uint_32  ST_TX_DEFERRED;
    uint_32  ST_TX_LATECOLL;
    uint_32  ST_TX_EXCESSCOLL;
    uint_32  ST_TX_CARRIER;
    uint_32  ST_TX_UNDERRUN;
} ENET_STATS, _PTR_ ENET_STATS_PTR;
```

#### ST\_RX\_TOTAL

Received (total).

#### ST\_RX\_MISSED

Received (missed packets).

#### ST\_RX\_DISCARDED

Received (discarded due to unrecognized protocol).

#### ST\_RX\_ERRORS

Received (discarded due to error on reception).

#### ST\_TX\_TOTAL

Transmitted (total).

#### ST\_TX\_MISSED

Transmitted (discarded because transmit ring was full).



**ST\_TX\_DISCARDED**

Transmitted (discarded because packet was bad packet).

**ST\_TX\_ERRORS**

Transmitted (errors during transmission).

**ST\_TX\_COLLHIST**

Transmitted (collision histogram).

The following stats are for physical errors or conditions.

**ST\_RX\_ALIGN**

Frame alignment errors.

**ST\_RX\_FCS**

CRC errors.

**ST\_RX\_RUNT**

Runt packets received.

**ST\_RX\_GIANT**

Giant packets received.

**ST\_RX\_LATECOLL**

Late collisions.

**ST\_RX\_OVERRUN**

DMA overruns.

**ST\_TX\_SQE**

Heartbeats lost.

**ST\_TX\_DEFERRED**

Transmissions deferred.

**ST\_TX\_LATECOLL**

Late collisions.

**ST\_TX\_EXCESSCOLL**

Excessive collisions.

**ST\_TX\_CARRIER**

Carrier sense lost.

**ST\_TX\_UNDERRUN**

DMA underruns.

### 8.3.7 HOSTENT\_STRUCT

A pointer to this structure is returned by the socket functions [gethostbyaddr\(\)](#) and [gethostbyname\(\)](#).

```
typedef struct hostent
{
    char_ptr      h_name;
    char_ptr      _PTR_ h_aliases;
    int_16        h_addrtype;
    int_16        h_length;
    char_ptr      _PTR_ h_addr_list;
} HOSTENT_STRUCT, _PTR_ HOSTENT_STRUCT_PTR;
```

#### **h\_name**

Pointer to the NULL-terminated character string that is the official name of the host.

#### **h\_aliases**

NULL-terminated array of alternate names for the host.

#### **h\_addrtype**

Type of address being returned (always *AF\_INET*).

#### **h\_length**

Length in bytes of the address.

#### **h\_addr\_list**

Pointer to a list of pointers to the network addresses for the host (each host address is represented as a series of bytes in network byte order; they are not ASCII strings).

## 8.3.8 HTTPD\_CGI\_LINK\_STRUCT

CGI callback structure. See [HTTPD\\_PARAMS\\_STRUCT](#).

```
typedef int(*CGI_CALLBACK)(HTTPD_SESSION_STRUCT*);

typedef struct httpd_cgi_link_struct {
    char cgi_name[HTTPDCFG_MAX_SCRIPT_LN + 1];
    CGI_CALLBACK callback;
} HTTPD_CGI_LINK_STRUCT;
```

### 8.3.8.1 Fields

#### 8.3.8.1.1 cgi\_name

callback function alias - used as name for requested page

#### 8.3.8.1.2 callback

callback function

### 8.3.9 HTTPD\_FN\_LINK\_STRUCT

Function callback link structure - one row in table functions callback table. See [HTTPD\\_PARAMS\\_STRUCT](#).

```
typedef void(*FN_CALLBACK) (HTTPD_SESSION_STRUCT*);
```

```
typedef struct httpd_fn_link_struct {  
    char fn_name[HTTPDCFG_MAX_SCRIPT_LN + 1];  
    FN_CALLBACK callback;  
} HTTPD_FN_LINK_STRUCT;
```

#### **fn\_name**

callback function alias - used as command in inlined script

#### **callback**

callback function

### 8.3.10 HTTPD\_PARAMS\_STRUCT

```
typedef struct httpd_params_struct {
    unsigned short port;
    unsigned int max_uri;
    unsigned int max_auth;

#ifdef HTTPDCFG_POLL_MODE
    unsigned int max_ses;
    unsigned int max_line;
#endif

    HTTPD_ROOT_DIR_STRUCT *root_dir;
    char *index_page;

    // callback functions
    HTTPD_CGI_LINK_STRUCT *cgi_lnk_tbl;
    HTTPD_FN_LINK_STRUCT *fn_lnk_tbl;
    HTTPD_AUTH_CALLBACK auth_fn;

    char *page401;
    char *page403;
    char *page404;
} HTTPD_PARAMS_STRUCT;
```

#### **port**

HTTP Server listening port.

#### **max\_uri**

maximal URI string length

#### **max\_auth**

maximal auth string length

#### **max\_ses**

maximal count of sessions

#### **max\_line**

maximal evaluated line length

#### **root\_dir**

pointer to root dir structure

#### **index\_page**

pointer to index page - full path and name

#### **cgi\_lnk\_tbl**

cgi function callback table. See [HTTPD\\_CGI\\_LINK\\_STRUCT](#).

#### **fn\_lnk\_tbl**

function callback table (dynamic web pages). See [HTTPD\\_FN\\_LINK\\_STRUCT](#).

**auth\_fn**

callback for authentication function

### 8.3.11 HTTPD\_ROOT\_DIR\_STRUCT

```
typedef struct httpd_root_dir_struct {  
    char *alias;  
    char *path;  
} HTTPD_ROOT_DIR_STRUCT;
```

**alias**

symbolic name (alias) for path

**path**

absolut path



## 8.3.12 HTTPD\_SESSION\_STRUCT

HTTP session structure — contains run-time data for session.

```
typedef struct httpd_session_struct {
    HTTPD_SES_STATE state;
    int valid;
    unsigned int keep_alive;
    int sock;

    HTTPD_REQ_STRUCT request;
    HTTPD_RES_STRUCT response;
    int header;
    int req_lines;
    int remain;
    HTTPD_TIME_STRUCT time;

    char recv_buf[HTTPDCFG_RECV_BUF_LEN + 1];
    char *recv_rd;
    int recv_used;

#ifdef HTTPDCFG_POLL_MODE
    char *line;
    int line_used;
#endif
} HTTPD_SESSION_STRUCT
```

### state

actual session status

### valid

describe session validity

### keep\_alive

connection persistence

### sock

socket used by session

### request

http request data storage

### response

http response data storage

### header

flag for header sending

### req\_lines

### remain

### time

## Data Types

state start time in ticks

### **recv\_buf**

temporary receiving buffer

### **recv\_rd**

reading pointer in recv\_buf

### **recv\_used**

recv\_buf used size

### **line**

### **line\_used**

### 8.3.13 HTTPD\_STRUCT

Main HTTP server structure.

```
typedef struct httpd_struct {  
    HTTPD_PARAMS_STRUCT *params;  
  
    // runtime data  
    int sock;  
    HTTPD_SESSION_STRUCT **session;  
} HTTPD_STRUCT;
```

#### **params**

pointer to server parameters structure

#### **sock**

runtime data - listen socket

#### **session**

runtime data - field of pointers to session specific structure

### 8.3.14 ICMP\_STATS

A pointer to this structure is returned by [ICMP\\_stats\(\)](#).

```
typedef struct {
    uint_32          ST_RX_TOTAL;
    uint_32          ST_RX_MISSED;
    uint_32          ST_RX_DISCARDED;
    uint_32          ST_RX_ERRORS;

    uint_32          ST_TX_TOTAL;
    uint_32          ST_TX_MISSED;
    uint_32          ST_TX_DISCARDED;
    uint_32          ST_TX_ERRORS;

    RTCS_ERROR_STRUCT ERR_RX;
    RTCS_ERROR_STRUCT ERR_TX;

    uint_32          ST_RX_BAD_CODE;
    uint_32          ST_RX_BAD_CHECKSUM;
    uint_32          ST_RX_SMALL_DGRAM;
    uint_32          ST_RX_RD_NOTGATE;

    uint_32          ST_RX_DESTUNREACH;
    uint_32          ST_RX_TIMEEXCEED;
    uint_32          ST_RX_PARMPROB;
    uint_32          ST_RX_SRCQUENCH;
    uint_32          ST_RX_REDIRECT;
    uint_32          ST_RX_ECHO_REQ;
    uint_32          ST_RX_ECHO_REPLY;
    uint_32          ST_RX_TIME_REQ;
    uint_32          ST_RX_TIME_REPLY;
    uint_32          ST_RX_INFO_REQ;
    uint_32          ST_RX_INFO_REPLY;
    uint_32          ST_RX_OTHER;

    uint_32          ST_TX_DESTUNREACH;
    uint_32          ST_TX_TIMEEXCEED;
    uint_32          ST_TX_PARMPROB;
    uint_32          ST_TX_SRCQUENCH;
    uint_32          ST_TX_REDIRECT;
    uint_32          ST_TX_ECHO_REQ;
    uint_32          ST_TX_ECHO_REPLY;
    uint_32          ST_TX_TIME_REQ;
    uint_32          ST_TX_TIME_REPLY;
    uint_32          ST_TX_INFO_REQ;
    uint_32          ST_TX_INFO_REPLY;
    uint_32          ST_TX_OTHER;
} ICMP_STATS, _PTR_ ICMP_STATS_PTR;
```

#### 8.3.14.0.1 ST\_RX\_TOTAL

Total number of received packets.

#### ST\_RX\_MISSED

Incoming packets discarded due to lack of resources.

**ST\_RX\_DISCARDED**

Incoming packets discarded for all other reasons.

**ST\_RX\_ERRORS**

Internal errors detected while processing an incoming packet.

**ST\_TX\_TOTAL**

Total number of transmitted packets.

**ST\_TX\_MISSED**

Packets to be sent that were discarded due to lack of resources.

**ST\_TX\_DISCARDED**

Packets to be sent that were discarded for all other reasons.

**ST\_TX\_ERRORS**

Internal errors detected while trying to send a packet.

**ERR\_RX**

RX error information.

**ERR\_TX**

TX error information.

The following are included in *ST\_RX\_DISCARDED*.

**ST\_RX\_BAD\_CODE**

Datagrams with unrecognized code.

**ST\_RX\_BAD\_CHECKSUM**

Datagrams with an invalid checksum.

**ST\_RX\_SMALL\_DGRAM**

Datagrams smaller than the header.

**ST\_RX\_RD\_NOTGATE**

Redirects received from a non-gateway.

Stats on each *ICMP* type.

**ST\_RX\_DESTUNREACH**

Received Destination Unreachables.

**ST\_RX\_TIMEEXCEED**

Received Time Exceeded.

**ST\_RX\_PARMPROB**

Received Parameter Problems.

**ST\_RX\_SRCQUENCH**

Received Source Quenches.

**ST\_RX\_REDIRECT**

Received Redirects.

**ST\_RX\_ECHO\_REQ**

Received Echo Requests.

**ST\_RX\_ECHO\_REPLY**

Received Echo Replies.

**ST\_RX\_TIME\_REQ**

Received Timestamp Requests.

**ST\_RX\_TIME\_REPLY**

Received Timestamp Replies.

**ST\_RX\_INFO\_REQ**

Received Information Requests.

**ST\_RX\_INFO\_REPLY**

Received Information Replies.

**ST\_RX\_OTHER**

Received all other types.

**ST\_TX\_DESTUNREACH**

Transmitted Destination Unreachables.

**ST\_TX\_TIMEEXCEED**

Transmitted Time Exceeded.

**ST\_TX\_PARMPROB**

Transmitted Parameter Problems.

**ST\_TX\_SRCQUENCH**

Transmitted Source Quenches.

**ST\_TX\_REDIRECT**

Transmitted Redirects.

**ST\_TX\_ECHO\_REQ**

Transmitted Echo Requests.

**ST\_TX\_ECHO\_REPLY**

Transmitted Echo Replies.

**ST\_TX\_TIME\_REQ**

Transmitted Timestamp Requests.

**ST\_TX\_TIME\_REPLY**

Transmitted Timestamp Replies.

**ST\_TX\_INFO\_REQ**

Transmitted Information Requests.

**ST\_TX\_INFO\_REPLY**

Transmitted Information Replies.

**ST\_TX\_OTHER**

Transmitted all other types.

### 8.3.15 IGMP\_STATS

A pointer to this structure is returned by **IGMP\_stats()**.

```
typedef struct {

    uint_32  ST_RX_TOTAL;
    uint_32  ST_RX_MISSED;
    uint_32  ST_RX_DISCARDED;
    uint_32  ST_RX_ERRORS;

    uint_32  ST_TX_TOTAL;
    uint_32  ST_TX_MISSED;
    uint_32  ST_TX_DISCARDED;
    uint_32  ST_TX_ERRORS;

    RTCS_ERROR_STRUCT ERR_RX;
    RTCS_ERROR_STRUCT ERR_TX;

    uint_32  ST_RX_BAD_TYPE;
    uint_32  ST_RX_BAD_CHECKSUM;
    uint_32  ST_RX_SMALL_DGRAM;
    uint_32  ST_RX_QUERY;
    uint_32  ST_RX_REPORT;

    uint_32  ST_TX_QUERY;
    uint_32  ST_TX_REPORT;

} IGMP_STATS, _PTR_ IGMP_STATS_PTR;
```

#### ST\_RX\_BAD\_TYPE

Datagrams with unrecognized code.

#### ST\_RX\_BAD\_CHECKSUM

Datagrams with invalid checksum.

#### ST\_RX\_SMALL\_DGRAM

Datagrams smaller than header.

#### ST\_RX\_QUERY

Received queries.

#### ST\_RX\_REPORT

Received reports.

#### ST\_TX\_QUERY

Transmitted queries.

#### ST\_TX\_REPORT

Transmitted reports.



### 8.3.16 in\_addr

Structure of address fields in the following structures:

- *ip\_mreq*
- *sockaddr\_in*

```
typedef struct in_addr {  
    _ip_address s_addr;  
} in_addr;
```

#### **s\_addr**

IP address.

### 8.3.17 ip\_mreq

IP multicast group.

```
typedef struct ip_mreq {  
    in_addr  imr_multiaddr;  
    in_addr  imr_interface;  
} ip_mreq;
```

#### **imr\_multiaddr**

Multicast IP address.

#### **imr\_interface**

Local IP address.

### 8.3.18 IP\_STATS

A pointer to this structure is returned by `IP_stats()`.

```
typedef struct {
    uint_32          ST_RX_TOTAL;
    uint_32          ST_RX_MISSED;
    uint_32          ST_RX_DISCARDED;
    uint_32          ST_RX_ERRORS;

    uint_32          ST_TX_TOTAL;
    uint_32          ST_TX_MISSED;
    uint_32          ST_TX_DISCARDED;
    uint_32          ST_TX_ERRORS;

    RTCS_ERROR_STRUCT ERR_RX;
    RTCS_ERROR_STRUCT ERR_TX;

    uint_32          ST_RX_HDR_ERRORS;
    uint_32          ST_RX_ADDR_ERRORS;
    uint_32          ST_RX_NO_PROTO;
    uint_32          ST_RX_DELIVERED;
    uint_32          ST_RX_FORWARDED;

    uint_32          ST_RX_BAD_VERSION;
    uint_32          ST_RX_BAD_CHECKSUM;
    uint_32          ST_RX_BAD_SOURCE;
    uint_32          ST_RX_SMALL_HDR;
    uint_32          ST_RX_SMALL_DGRAM;
    uint_32          ST_RX_SMALL_PKT;
    uint_32          ST_RX_TTL_EXCEEDED;
    uint_32          ST_RX_FRAG_RECVD;
    uint_32          ST_RX_FRAG_REASMD;
    uint_32          ST_RX_FRAG_DISCARDED;

    uint_32          ST_TX_FRAG_SENT;
    uint_32          ST_TX_FRAG_FRAGD;
    uint_32          ST_TX_FRAG_DISCARDED
} IP_STATS, _PTR_ IP_STATS_PTR;
```

#### ST\_RX\_TOTAL

Total number of received packets.

#### ST\_RX\_MISSED

Incoming packets discarded due to lack of resources.

#### ST\_RX\_DISCARDED

Incoming packets discarded for all other reasons.

#### ST\_RX\_ERRORS

Internal errors detected while processing an incoming packet.

#### ST\_TX\_TOTAL

Total number of transmitted packets.

## **ST\_TX\_MISSED**

Packets to be sent that were discarded due to lack of resources.

## **ST\_TX\_DISCARDED**

Packets to be sent that were discarded for all other reasons.

## **ST\_TX\_ERRORS**

Internal errors detected while trying to send a packet.

## **ERR\_RX**

RX error information.

## **ERR\_TX**

TX error information.

## **ST\_RX\_HDR\_ERRORS**

Discarded (error in the IP header).

## **ST\_RX\_ADDR\_ERRORS**

Discarded (illegal destination).

## **ST\_RX\_NO\_PROTO**

Datagrams larger than the frame.

## **ST\_RX\_DELIVERED**

Datagrams delivered to the upper layer.

## **ST\_RX\_FORWARDED**

Datagrams forwarded.

The following are included in *ST\_RX\_DISCARDED* and *ST\_RX\_HDR\_ERRORS*.

## **ST\_RX\_BAD\_VERSION**

Datagrams with the version not equal to four.

## **ST\_RX\_BAD\_CHECKSUM**

Datagrams with an invalid checksum.

## **ST\_RX\_BAD\_SOURCE**

Datagrams with an invalid source address.

## **ST\_RX\_SMALL\_HDR**

Datagrams with a header too small.

## **ST\_RX\_SMALL\_DGRAM**

Datagrams smaller than the header.

**ST\_RX\_SMALL\_PKT**

Datagrams larger than the frame.

**ST\_RX\_TTL\_EXCEEDED**

Datagrams to route with TTL = 0.

**ST\_RX\_FRAG\_RECVD**

Received IP fragments.

**ST\_RX\_FRAG\_REASMD**

Reassembled datagrams.

**ST\_RX\_FRAG\_DISCARDED**

Discarded fragments.

**ST\_TX\_FRAG\_SENT**

Sent fragments.

**ST\_TX\_FRAG\_FRAGD**

Fragmented datagrams.

**ST\_TX\_FRAG\_DISCARDED**

Fragmentation failures.

### 8.3.19 IPCFG\_IP\_ADDRESS\_DATA

Interface address structure.

```
typedef uint_32 _ip_address;  
  
typedef struct ipcfg_ip_address_data  
{  
    _ip_address ip;  
    _ip_address mask;  
    _ip_address router;  
} IPCFG_IP_ADDRESS_DATA, _PTR_ IPCFG_IP_ADDRESS_DATA_PTR;
```

**ip**

ip address

**mask**

mask

**route**

gateway

## 8.3.20 IPCP\_DATA\_STRUCT

A pointer to this structure is a parameter of [RTCS\\_if\\_bind\\_IPCP\(\)](#).

```
typedef struct {
    void (_CODE_PTR_ IP_UP) (pointer);
    void (_CODE_PTR_ IP_DOWN) (pointer);
    pointer IP_PARAM;
    unsigned ACCEPT_LOCAL_ADDR : 1;
    unsigned ACCEPT_REMOTE_ADDR : 1;
    unsigned DEFAULT_NETMASK : 1;
    unsigned DEFAULT_ROUTE : 1;
    unsigned NEG_LOCAL_DNS : 1;
    unsigned NEG_REMOTE_DNS : 1;
    unsigned ACCEPT_LOCAL_DNS : 1;
    /*Ignored if NEG_LOCAL_DNS = 0. */
    unsigned ACCEPT_REMOTE_DNS : 1;
    /*Ignored if NEG_REMOTE_DNS = 0. */
    unsigned : 0;

    _ip_address LOCAL_ADDR;
    _ip_address REMOTE_ADDR;
    _ip_address NETMASK;
    /* Ignored if DEFAULT_NETMASK = 1. */
    _ip_address LOCAL_DNS;
    /* Ignored if NEG_LOCAL_DNS = 0. */
    _ip_address REMOTE_DNS;
    /* Ignored if NEG_REMOTE_DNS = 0. */

} IPCP_DATA_STRUCT, _PTR_ IPCP_DATA_STRUCT_PTR;
```

### IP\_UP

### IP\_DOWN

### IP\_PARAM

RTCS calls	With	When IPCP successfully
<i>IP_UP</i>	<i>IP_PARAM</i>	Enters the opened state.
<i>IP_DOWN</i>	<i>IP_PARAM</i>	Leaves the opened state.

### ACCEPT\_LOCAL\_ADDR

### LOCAL\_ADDR

IPCP attempts to negotiate *LOCAL\_ADDR* as its local IP address.

If <b>ACCEPT_LOCAL_ADDR</b> is:	IPCP does this
TRUE	Allows the peer to negotiate a different local IP address.
FALSE	Accepts only <i>LOCAL_ADDR</i> as its local IP address.

### ACCEPT\_REMOTE\_ADDR

### REMOTE\_ADDR

IPCP attempts to negotiate *REMOTE\_ADDR* as the peer IP address.

If <b>ACCEPT_REMOTE_ADDR</b> is:	IPCP does this
TRUE	Allows the peer to negotiate a different peer IP address.
FALSE	Accepts only <i>REMOTE_ADDR</i> as its peer IP address.

## NETMASK

### DEFAULT\_NETMASK

If <b>DEFAULT_NETMASK</b> is:	IPCP does this
TRUE	Dynamically calculates the link's netmask based on the negotiated local and peer IP addresses.
FALSE	IPCP always uses <i>NETMASK</i> as the netmask.

## DEFAULT\_ROUTE

If *DEFAULT\_ROUTE* is TRUE, IPCP installs the peer as a default gateway in the IP routing table.

### ACCEPT\_LOCAL\_DNS

### NEG\_LOCAL\_DNS

### LOCAL\_DNS

Controls, whether RTCS negotiates the address of a DNS server to be used by the local resolver.

If *ACCEPT\_LOCAL\_DNS* is TRUE, a peer can override *LOCAL\_DNS*.

If <b>NEG_LOCAL_DNS</b> is:	IPCP does this
TRUE	Attempts to negotiate <i>LOCAL_DNS</i> as the DNS server address that is to be used by the local resolver.
FALSE	Does not attempt to negotiate a DNS server address for the local resolver.

### ACCEPT\_REMOTE\_DNS

### NEG\_REMOTE\_DNS

### REMOTE\_DNS



Controls, whether RTCS negotiates the address of a DNS server to be used by the peer resolver. If *ACCEPT\_REMOTE\_DNS* is TRUE, a peer can override *REMOTE\_DNS*.

If <i>NEG_REMOTE_DNS</i> is	IPCP does this
TRUE	Attempts to negotiate <i>REMOTE_DNS</i> as the DNS server address that is to be used by the peer resolver
FALSE	Does not attempt to negotiate a DNS server address for the peer resolver

### 8.3.21 IPIF\_STATS

A pointer to this structure is returned by [IPIF\\_stats\(\)](#).

```
typedef struct {
    uint_32      ST_RX_TOTAL;
    uint_32      ST_RX_MISSED;
    uint_32      ST_RX_DISCARDED;
    uint_32      ST_RX_ERRORS;

    uint_32      ST_TX_TOTAL;
    uint_32      ST_TX_MISSED;
    uint_32      ST_TX_DISCARDED;
    uint_32      ST_TX_ERRORS;

    RTCS_ERROR_STRUCT ERR_RX;
    RTCS_ERROR_STRUCT ERR_TX;

    uint_32      ST_RX_OCTETS;
    uint_32      ST_RX_UNICAST;
    uint_32      ST_RX_MULTICAST;
    uint_32      ST_RX_BROADCAST;

    uint_32      ST_TX_OCTETS;
    uint_32      ST_TX_UNICAST;
    uint_32      ST_TX_MULTICAST;
    uint_32      ST_TX_BROADCAST;
} IPIF_STATS, _PTR_ IPIF_STATS_PTR;
```

#### ST\_RX\_TOTAL

Total number of received packets.

#### ST\_RX\_MISSED

Incoming packets discarded due to lack of resources.

#### ST\_RX\_DISCARDED

Incoming packets discarded for all other reasons.

#### ST\_RX\_ERRORS

Internal errors detected while processing an incoming packet.

#### ST\_TX\_TOTAL

Total number of transmitted packets.

#### ST\_TX\_MISSED

Packets to be sent that were discarded due to lack of resources.

#### ST\_TX\_DISCARDED

Packets to be sent that were discarded for all other reasons.

#### ST\_TX\_ERRORS

Internal errors detected while trying to send a packet.

**ERR\_RX**

RX error information.

**ERR\_TX**

TX error information.

**ST\_RX\_OCTETS**

Total bytes received.

**ST\_RX\_UNICAST**

Unicast packets received.

**ST\_RX\_MULTICAST**

Multicast packets received.

**ST\_RX\_BROADCAST**

Broadcast packets received.

**ST\_TX\_OCTETS**

Total bytes sent.

**ST\_TX\_UNICAST**

Unicast packets sent.

**ST\_TX\_MULTICAST**

Multicast packets sent.

**ST\_TX\_BROADCAST**

Broadcast packets sent.

### 8.3.22 nat\_ports

Used by Freescale MQX NAT to control the range of ports between and including the minimum and maximum ports specified.

```
typedef struct {  
    uint_16  port_min;  
    uint_16  port_max;  
} nat_ports;
```

#### **PORT\_MIN**

Minimum port number.

#### **PORT\_MAX**

Maximum port number.

### 8.3.23 NAT\_STATS

Network address translation statistics.

```
typedef struct {
    uint_32  ST_SESSIONS;
    uint_32  ST_SESSIONS_OPEN;
    uint_32  ST_SESSIONS_OPEN_MAX;

    uint_32  ST_PACKETS_TOTAL;
    uint_32  ST_PACKETS_BYPASS;
    uint_32  ST_PACKETS_PUB_PRV;
    uint_32  ST_PACKETS_PUB_PRV_ERR;
    uint_32  ST_PACKETS_PRV_PUB;
    uint_32  ST_PACKETS_PRV_PUB_ERR;
} NAT_STATS, _PTR_ NAT_STATS_PTR;
```

#### ST\_SESSIONS

Total amount of sessions created to date.

#### ST\_SESSIONS\_OPEN

Number of sessions currently open.

#### ST\_SESSIONS\_OPEN\_MAX

Maximum number of sessions open simultaneously to date.

#### ST\_PACKETS\_TOTAL

Number of packets processed by Freescale MQX NAT.

#### ST\_PACKETS\_BYPASS

Number of unmodified packets.

#### ST\_PACKETS\_PUB\_PRV

Number of packets from public to private realm.

#### ST\_PACKETS\_PUB\_PRV\_ERR

Number of packets from public to private realm with errors (packets that have errors are discarded).

#### ST\_PACKETS\_PRV\_PUB

Number of packets from private to public realm.

#### ST\_PACKETS\_PRV\_PUB\_ERR

Number of packets from private to public realm with errors (packets that have errors are discarded).

### 8.3.24 nat\_timeouts

Used by Freescale MQX NAT to determine inactivity timeout settings.

```
typedef struct {  
    uint_32  timeout_tcp;  
    uint_32  timeout_fin;  
    uint_32  timeout_udp;  
} nat_timeouts;
```

#### **TIMEOUT\_TCP**

Inactivity timeout setting for a TCP session.

#### **TIMEOUT\_FIN**

Inactivity timeout setting for a TCP session, in which a FIN or RST bit has been set.

#### **TIMEOUT\_UDP**

Inactivity timeout setting for a UDP or ICMP session.

### 8.3.25 PPPOE\_CLIENT\_INIT\_DATA\_STRUCT

A pointer to this structure is the parameter to [\\_iopcb\\_pppoe\\_client\\_init\(\)](#).

```
typedef struct pppoe_client_init_data_struct {
    pointer          EHANDLE;
    char_ptr         SERVICE_NAME;
    char_ptr         AC_NAME;
    boolean          HOST_UNIQUE;

    uint_32          RTX_TASK_PRIORITY;
    uint_32          RTX_TASK_STACK;

    uint_32          RTX_MIN_TIMEOUT;
    uint_32          RTX_MAX_TIMEOUT;
    uint_32          RTX_MAX_RETRY;

    boolean          SEND_PADI_FOR_EVER;

    void             (_CODE_PTR_ CONNECTION_TIME_OUT)(pointer);
    uint_32          (_CODE_PTR_ SEND_PADI_TAGS_EXTRA)(uchar_ptr);
    uint_32          (_CODE_PTR_ SEND_PADR_TAGS_EXTRA)(uchar_ptr);
    boolean          (_CODE_PTR_ PARSE_PADO_TAGS_EXTRA)(pointer);
    boolean          (_CODE_PTR_ PARSE_PADS_TAGS_EXTRA)(pointer);
} PPPOE_CLIENT_INIT_DATA_STRUCT, _PTR_
  PPPOE_CLIENT_INIT_DATA_STRUCT_PTR;
```

#### EHANDLE

Pointer to the initialized Ethernet handle from [ENET\\_initialize\(\)](#). The application must initialize the field.

#### SERVICE\_NAME

Pointer to the service name to open for the session. If you set the field to NULL, it is ignored.

#### AC\_NAME

Pointer to the name of the access concentrator to negotiate a session with. If you set the field to NULL, any access concentrator is used and the first access concentrator that responds with a PADO packet is accepted.

#### HOST\_UNIQUE

If you set the field to TRUE, the host unique ID is used for the client session. If you set the field to FALSE, the host unique ID is not used.

#### RTX\_TASK\_PRIORITY

Task priority for *PPPOE\_rtx\_task*. If you set the field to zero, [\\_iopcb\\_pppoe\\_client\\_init\(\)](#) sets it to the default value (six).

#### RTX\_TASK\_STACK

Extra stack space needed for *PPPOE\_rtx\_task*.

**RTX\_MIN\_TIMEOUT**

Minimum time to wait before retransmitting a discovery packet. If you set the field to zero, `_iopcb_pppoe_client_init()` sets it to the default value (3000, which is three seconds).

**RTX\_MAX\_TIMEOUT**

Maximum time to wait before retransmitting a discovery packet. If you set the field to zero, `_iopcb_pppoe_client_init()` sets it to the default value (10000, which is ten seconds).

**RTX\_MAX\_RETRY**

Number of requests to make before the connection request fails. If you set the field to zero, `_iopcb_pppoe_client_init()` sets it to the default value (ten).

**SEND\_PADI\_FOR\_EVER**

If you set the field to TRUE, PADI packets are sent until a reply is received from the access concentrator. If you set the field to FALSE, PADI packets are no longer sent if a reply is not received from the access concentrator.

**CONNECTION\_TIME\_OUT**

Callback function that can inform the application of the PADI timeout. The `_iopcb_handle` is passed to the callback function. If the `SEND_PADI_FOR_EVER` field is TRUE, the function is not called. If you set the field to NULL, it is ignored.

**SEND\_PADI\_TAGS\_EXTRA**

Callback function that can send extra tags (for example vendor-specific tags) with PADI packets. The `uchar_ptr` parameter should be returned to the driver. If you set the field to NULL, it is ignored.

**SEND\_PADR\_TAGS\_EXTRA**

Callback function that can send extra tags with PADR packets. The parameter of type `uchar_ptr` should be returned to the driver. If you set the field to NULL, it is ignored.

**PARSE\_PADO\_TAGS\_EXTRA**

Callback function that can parse extra tags with PADO packets. If you set the field to NULL, it is ignored.

**PARSE\_PADS\_TAGS\_EXTRA**

Callback function that can parse extra tags with PADS packets. If you set the field to NULL, it is ignored.



### 8.3.26 PPPOE\_SERVER\_INIT\_DATA\_STRUCT

PPPoE Server parameter configuration structure.

```
typedef struct pppoe_server_init_data_struct {
    char_ptr      SERVICE_NAME;
    char_ptr      AC_NAME;
    uint_32       SERVER_TASK_PRIORITY;
    uint_32       SERVER_TASK_STACK;
    uint_32       ECHO_TIMEOUT;
    uint_32       ECHO_MAX_RETRY;
    void (_CODE_PTR_) SESSION_UP (pointer,pointer,pointer);
    void (_CODE_PTR_) SESSION_DOWN (pointer,pointer,pointer);
    pointer       PARAM;
    uint_32 (_CODE_PTR_) SEND_PADO_TAGS_EXTRA (uchar_ptr);
    uint_32 (_CODE_PTR_) SEND_PADS_TAGS_EXTRA (uchar_ptr);
    boolean (_CODE_PTR_) PARSE_PADI_TAGS_EXTRA (pointer);
    boolean (_CODE_PTR_) PARSE_PADR_TAGS_EXTRA (pointer);
} PPPOE_SERVER_INIT_DATA_STRUCT, _PTR_ PPPOE_SERVER_INIT_DATA_STRUCT_PTR;
```

#### SERVICE\_NAME

Pointer to the service name to open for the session. If you set the field to NULL, it is ignored.

#### AC\_NAME

Pointer to the access concentrator name.

#### SERVER\_TASK\_PRIORITY

Task priority for the PPPoE Server task. If you set the field to zero, **\_pppoe\_server\_init()** sets it to the default value (six).

#### SERVER\_TASK\_STACK

Extra stack space for the PPPoE Server task.

#### ECHO\_TIMEOUT

Maximum time to wait before retransmitting an echo packet. If you set the field to zero, **\_pppoe\_server\_init()** sets it to the default value (10 000, which is ten seconds).

#### ECHO\_MAX\_RETRY

Number of echo requests to make before closing the client connection. If you set the field to zero, **\_pppoe\_server\_init()** sets it to the default value (six).

#### SESSION\_UP

Callback function to call after a session is established with the client.

#### SESSION\_DOWN

Callback function to call after a session is terminated.

#### PARAM

Parameter to the *SESSION\_UP* or *SESSION\_DOWN* callback function.

### **SEND\_PADO\_TAGS\_EXTRA**

Callback function that can parse extra tags (such as vendor-specific tags) with PADO packets. If you set the field to NULL, it is ignored.

### **SEND\_PADS\_TAGS\_EXTRA**

.Callback function that can parse extra tags with PADS packets. If you set the field to NULL, it is ignored.

### **SEND\_PADI\_TAGS\_EXTRA**

Callback function that can send extra tags (for example vendor-specific tags) with PADI packets. The parameter should be returned to the driver. If you set the field to NULL, it is ignored.

### **SEND\_PADR\_TAGS\_EXTRA**

.Callback function that can send extra tags with PADR packets. The parameter should be returned to the driver. If you set the field to NULL, it is ignored.

### 8.3.27 PPPOE\_SESSION\_STATS\_STRUCT

Statistics for the PPP session that is registered with the PPPoE Server.

```
typedef struct pppoe_session_stats_struct{
    uint_32  ST_RX_TOTAL;
    uint_32  ST_RX_MISSED;
    uint_32  ST_RX_DISCARDED;
    uint_32  ST_RX_ERRORS;

    uint_32  ST_TX_TOTAL;
    uint_32  ST_TX_MISSED;
    uint_32  ST_TX_DISCARDED;
    uint_32  ST_TX_ERRORS;

    uint_32  ST_RX_UNICAST;
    uint_32  ST_RX_BROADCAST;
} PPPOE_SESSION_STATS_STRUCT, _PTR_
  PPPOE_SESSION_STATS_STRUCT_PTR;
```

#### ST\_RX\_TOTAL

Total number of received packets to the session.

#### ST\_RX\_MISSED

Incoming packets discarded due to lack of resources.

#### ST\_RX\_DISCARDED

Incoming packets discarded for all other reasons.

#### ST\_RX\_ERRORS

Internal errors detected while processing an incoming packet.

#### ST\_TX\_TOTAL

Total number of transmitted packets.

#### ST\_TX\_MISSED

Packets to be sent that were discarded due to lack of resources.

#### ST\_TX\_DISCARDED

Packets to be sent that were discarded for all other reasons.

#### ST\_TX\_ERRORS

Internal errors detected while trying to send a packet.

#### ST\_RX\_UNICAST

Unicast packets received.

#### ST\_RX\_BROADCAST

Broadcast packets received.

## 8.3.28 PPPOEIF\_STATS\_STRUCT

Statistics for the PPP over Ethernet driver layer.

```
typedef struct pppoeif_stats_struct{
    uint_32  ST_RX_TOTAL;
    uint_32  ST_RX_MISSED;
    uint_32  ST_RX_DISCARDED;
    uint_32  ST_RX_ERRORS;

    uint_32  ST_TX_TOTAL;
    uint_32  ST_TX_MISSED;
    uint_32  ST_TX_DISCARDED;
    uint_32  ST_TX_ERRORS;

    uint_32  ST_RX_OCTETS;
    uint_32  ST_RX_UNICAST;
    uint_32  ST_RX_BROADCAST;
    uint_32  ST_RX_MULTICAST;

    uint_32  ST_TX_UNICAST;
    uint_32  ST_TX_BROADCAST;

    uint_32  ST_TX_PADITRANSMITTED;
    uint_32  ST_TX_PADOTRANSMITTED;

    uint_32  ST_TX_PADRTRANSMITTED;
    uint_32  ST_TX_PADSTRANSMITTED;
    uint_32  ST_TX_PADTTRANSMITTED;
    uint_32  ST_TX_GENERIC_ERRORS_TRANSMITTED;

    uint_32  ST_RX_PADIREJECTED;
    uint_32  ST_RX_PADRREJECTED;
    uint_32  ST_RX_PADSRECEIVED;
    uint_32  ST_RX_PADORECEIVED;
    uint_32  ST_RX_PADTRECEIVED;
    uint_32  ST_RX_GENERIC_ERRORS_RECEIVED;
    uint_32  ST_RX_MALFORMED_PACKETS;
    uint_32  ST_RX_MULTIPLE_PADO_RECEIVED;
    uint_32  ST_RX_SERVICENAMEERRORS;
    uint_32  ST_RX_ACSYSTEMERRORS;
    uint_32  ST_RX_PADI;

    uint_16  SERVICE_NAME_ERROR_TAG_LENGTH;
    char_ptr SERVICE_NAME_ERROR_DATA;

    uint_16  AC_SYSTEM_ERROR_TAG_LENGTH;
    char_ptr AC_SYSTEM_ERROR_DATA;

    uint_16  GENERIC_ERROR_TAG_LENGTH;
    char_ptr GENERIC_ERROR_DATA;
} PPPOEIF_STATS_STRUCT, _PTR_ PPPOEIF_STATS_STRUCT_PTR;
```

### ST\_RX\_TOTAL

Total number of received packets to the driver.

### ST\_RX\_MISSED

Incoming packets discarded due to lack of resources.

**ST\_RX\_DISCARDED**

Incoming packets discarded for all other reasons.

**ST\_RX\_ERRORS**

Internal errors detected while processing an incoming packet.

**ST\_TX\_TOTAL**

Total number of transmitted packets.

**ST\_TX\_MISSED**

Packets to be sent that were discarded due to lack of resources.

**ST\_TX\_DISCARDED**

Packets to be sent that were discarded for all other reasons.

**ST\_TX\_ERRORS**

Internal errors detected while trying to send a packet.

**ST\_RX\_OCTETS**

Total bytes received.

**ST\_RX\_UNICAST**

Unicast packets received.

**ST\_RX\_BROADCAST**

Broadcast packets received.

**ST\_RX\_MULTICAST**

Multicast packets received.

**ST\_TX\_UNICAST**

Unicast packets sent.

**ST\_TX\_BROADCAST**

Broadcast packets sent.

**ST\_TX\_PADITRANSMITTED**

Number of transmitted PADI packets.

**ST\_TX\_PADOTRANSMITTED**

Number of transmitted PADO packets.

**ST\_TX\_PADRTRANSMITTED**

Number of transmitted PADR packets.

**ST\_TX\_PADSTRANSMITTED**

Number of transmitted PADS packets.

**ST\_TX\_PADTTRANSMITTED**

Number of transmitted PADT packets.

**ST\_TX\_GENERIC\_ERRORS\_TRANSMITTED**

Number of generic error packets transmitted.

**ST\_RX\_PADIREJECTED**

Number of PADI packets rejected.

**ST\_RX\_PADRREJECTED**

Number of PADR rejected.

**ST\_RX\_PADSRECEIVED**

Number of received PADS packets.

**ST\_RX\_PADORECEIVED**

Number of received PADO packets.

**ST\_RX\_PADTRECEIVED**

Number of received PADT packets.

**ST\_RX\_GENERIC\_ERRORS\_RECEIVED**

Number of generic error packets received.

**ST\_RX\_MALFORMED\_PACKETS**

Number of received malformed packets.

**ST\_RX\_MULTIPLE\_PADO\_RECEIVED**

Number of multiple PADO packets received.

**ST\_RX\_SERVICENAMEERRORS**

Number of service name error packets received.

**ST\_RX\_ACSYSTEMERRORS**

Number of AC system errors received.

**ST\_RX\_PADI**

Number of PADI packets received.

**SERVICE\_NAME\_ERROR\_TAG\_LENGTH**

Length of the service name error data string.

**SERVICE\_NAME\_ERROR\_DATA**

Data pointer to the most recent error string received.

**AC\_SYSTEM\_ERROR\_TAG\_LENGTH**

Length of the AC system error data string.

**AC\_SYSTEM\_ERROR\_DATA**

Data pointer to the most recent error string.

**GENERIC\_ERROR\_TAG\_LENGTH**

Length of the generic error data string.

**GENERIC\_ERROR\_DATA**

Data pointer to the length of the AC system error data string.

### 8.3.29 PPP\_SECRET

Used by PPP Driver for PAP and CHAP authentication of peers.

```
typedef struct {  
    uint_16    PPP_ID_LENGTH;  
    uint_16    PPP_PW_LENGTH;  
    char_ptr   PPP_ID_PTR;  
    char_ptr   PPP_PW_PTR;  
} PPP_SECRET, _PTR_ PPP_SECRET_PTR;
```

#### PPP\_ID\_LENGTH

Number of bytes in the array at *PPP\_ID\_PTR*.

#### PPP\_PW\_LENGTH

Number of bytes in the array at *PPP\_PW\_PTR*.

#### PPP\_ID\_PTR

Pointer to an array that represents a remote entity's ID, such as a host name or user ID.

#### PPP\_PW\_PTR

Pointer to an array that represents the password that is associated with the remote entity's ID.



### 8.3.30 RTCS\_ERROR\_STRUCT

Statistics for protocol errors; the structure that is included as fields *ERR\_TX* and *ERR\_RX* in the following statistics structures:

- *ARP\_STATS*
- *ICMP\_STATS*
- *IGMP\_STATS*
- *IP\_STATS*
- *IPIF\_STATS*
- *TCP\_STATS*
- *UDP\_STATS*

```
typedef struct {
    uint_32    ERROR;
    uint_32    PARM;
    _task_id   TASK_ID;
    uint_32    TASKCODE;
    pointer     MEMPTR;
    boolean     STACK;
} RTCS_ERROR_STRUCT, _PTR_ RTCS_ERROR_STRUCT_PTR;
```

#### **ERROR**

Code that describes the protocol error.

#### **PARM**

Parameters that are associated with the protocol error.

#### **TASK\_ID**

Task ID of the task that set the code.

#### **TASKCODE**

Task error code of the task that set the code.

#### **MEMPTR**

Highest core-memory address that MQX has allocated.

#### **STACK**

Whether the stack for the task that set the code is past its limit.

### 8.3.31 RTCS\_IF\_STRUCT

Callback functions for a device interface. A pointer to this structure is a parameter to [RTCS\\_if\\_add\(\)](#). To use the default table for an interface, use the constant that is defined in the following table.

Interface	Parameter to RTCS_if_add()
Ethernet	RTCS_IF_ENET
Local loopback	RTCS_IF_LOCALHOST
PPP	RTCS_IF_PPP

```
typedef struct {
    uint_32 (_CODE_PTR_ OPEN) (struct ip_if _PTR_);
    uint_32 (_CODE_PTR_ CLOSE) (struct ip_if _PTR_);
    uint_32 (_CODE_PTR_ SEND) (struct ip_if _PTR_,
                               struct rtcspcb _PTR_,
                               _ip_address,
                               _ip_address);
    uint_32 (_CODE_PTR_ JOIN) (struct ip_if _PTR_,
                               _ip_address);
    uint_32 (_CODE_PTR_ LEAVE) (struct ip_if _PTR_,
                                _ip_address);
} RTCS_IF_STRUCT, _PTR_ RTCS_IF_STRUCT_PTR;
```

The IP interface structure (*ip\_if*) contains information to let RTCS send packets (ethernet) or datagrams (PPP).

#### OPEN

Called by RTCS to register with a packet driver (ethernet) or to open a link (PPP).

- Parameter — pointer to the IP interface structure.

Returns a status code.

#### CLOSE

Called by RTCS to unregister with the packet driver (ethernet) or to close the link (PPP).

- Parameter — pointer to the IP interface structure.

Returns a status code.

#### SEND

Called by RTCS to send a packet (ethernet) or datagram (PPP).

- First parameter — pointer to the IP interface structure.
- Second parameter — pointer to the packet (ethernet) or datagram (PPP) to send.
- Third parameter:
  - For ethernet: Protocol to use (*ENETPROT\_IP* or *ENETPROT\_ARP*).
  - For PPP: Next-hop source address.
- Fourth parameter:
  - For ethernet: IP address of the destination.

- For PPP: Next-hop destination address.

Returns a status code.

## **JOIN**

Called by RTCS to join a multicast group (not used for PPP interfaces).

- First parameter — pointer to the IP interface structure.
- Second parameter — IP address of the multicast group.

Returns a status code.

## **LEAVE**

Called by RTCS to leave a multicast group (not used for PPP interfaces).

- First parameter—Pointer to the IP interface structure.
- Second parameter—IP address of the multicast group.

Returns a status code.

### 8.3.32 RTCS\_protocol\_table

A NULL-terminated table that defines the protocols that RTCS initializes and starts when RTCS is created. RTCS initializes the protocols in the order that they appear in the table. An application can use only the protocols that are in the table. If you remove a protocol from the table, RTCS does not link the associated code with your application, an action that reduces the code size.

```
extern uint_32 (_CODE_PTR_ RTCS_protocol_table[]) (void);
```

#### Protocols Supported

##### RTCSPROT\_IGMP

Internet Group Management Protocol — used for multicasting.

##### RTCSPROT\_UDP

User Datagram Protocol — connectionless datagram service.

##### RTCSPROT\_TCP

Transmission Control Protocol — reliable connection-oriented stream service.

##### RTCSPROT\_RIP

Routing Information Protocol — requires UDP.

#### Default RTCS Protocol Table

You can either define your own protocol table or use the following default table, which RTCS provides in *ifrtcsinit.c*:

```
uint_32 (_CODE_PTR_ RTCS_protocol_table[]) (void) = {
    RTCSPROT_IGMP,
    RTCSPROT_UDP,
    RTCSPROT_TCP,
    RTCSPROT_IPIP,
    NULL
};
```

### 8.3.33 RTCS\_TASK

Definition for Telnet Server shell task.

```
typedef struct {  
    char_ptr      NAME;  
    uint_32       PRIORITY;  
    uint_32       STACKSIZE;  
    void (_CODE_PTR_) START(pointer);  
    pointer       ARG;  
} RTCS_TASK, _PTR_ RTCS_TASK_PTR;
```

#### **NAME**

Name of the task.

#### **PRIORITY**

Task priority.

#### **STACKSIZE**

Stack size for the task.

#### **START**

Task entry point.

#### **ARG**

Parameter for the task.

### 8.3.34 RTCSMIB\_VALUE

```
typedef struct rtcsmib_value {  
    uint_32 TYPE;  
    pointer PARAM;  
} RTCSMIB_VALUE, _PTR_ RTCSMIB_VALUE_PTR;
```

#### TYPE

Value type.

#### PARAM

### 8.3.35 sockaddr\_in

Structure for a socket-endpoint identifier.

```
typedef struct sockaddr_in
{
    uint_16  sin_family;
    uint_16  sin_port;
    in_addr  sin_addr;
} sockaddr_in;
```

#### **sin\_family**

Address family type.

#### **sin\_port**

Port number.

#### **sin\_addr**

IP address.

### 8.3.36 TCP\_STATS

A pointer to this structure is returned by [TCP\\_stats\(\)](#).

```
typedef struct {
    uint_32      ST_RX_TOTAL;
    uint_32      ST_RX_MISSED;
    uint_32      ST_RX_DISCARDED;
    uint_32      ST_RX_ERRORS;

    uint_32      ST_TX_TOTAL;
    uint_32      ST_TX_MISSED;
    uint_32      ST_TX_DISCARDED;
    uint_32      ST_TX_ERRORS;

    RTCS_ERROR_STRUCT ERR_RX;
    RTCS_ERROR_STRUCT ERR_TX;

    uint_32      ST_RX_BAD_PORT;
    uint_32      ST_RX_BAD_CHECKSUM;
    uint_32      ST_RX_BAD_OPTION;
    uint_32      ST_RX_BAD_SOURCE;
    uint_32      ST_RX_SMALL_HDR;
    uint_32      ST_RX_SMALL_DGRAM;
    uint_32      ST_RX_SMALL_PKT;
    uint_32      ST_RX_BAD_ACK;
    uint_32      ST_RX_BAD_DATA;
    uint_32      ST_RX_LATE_DATA;
    uint_32      ST_RX_OPT_MSS;
    uint_32      ST_RX_OPT_OTHER;

    uint_32      ST_RX_DATA;
    uint_32      ST_RX_DATA_DUP;
    uint_32      ST_RX_ACK;
    uint_32      ST_RX_ACK_DUP;
    uint_32      ST_RX_RESET;
    uint_32      ST_RX_PROBE;
    uint_32      ST_RX_WINDOW;

    uint_32      ST_RX_SYN_EXPECTED;
    uint_32      ST_RX_ACK_EXPECTED;
    uint_32      ST_RX_SYN_NOT_EXPECTED;
    uint_32      ST_RX_MULTICASTS;

    uint_32      ST_TX_DATA;
    uint_32      ST_TX_DATA_DUP;
    uint_32      ST_TX_ACK;
    uint_32      ST_TX_ACK_DELAYED;
    uint_32      ST_TX_RESET;
    uint_32      ST_TX_PROBE;
    uint_32      ST_TX_WINDOW;

    uint_32      ST_CONN_ACTIVE;
    uint_32      ST_CONN_PASSIVE;
    uint_32      ST_CONN_OPEN;
    uint_32      ST_CONN_CLOSED;
    uint_32      ST_CONN_RESET;
    uint_32      ST_CONN_FAILED;
}
```



```
uint_32          ST_CONN_ABORTS;  
} TCP_STATS, _PTR_ TCP_STATS_PTR;
```

**ST\_RX\_TOTAL**

Total number of received packets.

**ST\_RX\_MISSED**

Incoming packets discarded due to lack of resources.

**ST\_RX\_DISCARDED**

Incoming packets discarded for all other reasons.

**ST\_RX\_ERRORS**

Internal errors detected while processing an incoming packet.

**ST\_TX\_TOTAL**

Total number of transmitted packets.

**ST\_TX\_MISSED**

Packets to be sent that were discarded due to lack of resources.

**ST\_TX\_DISCARDED**

Packets to be sent that were discarded for all other reasons.

**ST\_TX\_ERRORS**

Internal errors detected while trying to send a packet.

**ERR\_RX**

RX error information.

**ERR\_TX**

TX error information.

The following are included in *ST\_RX\_DISCARDED*.

**ST\_RX\_BAD\_PORT**

Segments with the destination port zero.

**ST\_RX\_BAD\_CHECKSUM**

Segments with an invalid checksum.

**ST\_RX\_BAD\_OPTION**

Segments with invalid options.

**ST\_RX\_BAD\_SOURCE**

Segments with an invalid source.

**ST\_RX\_SMALL\_HDR**

Segments with the header too small.

**ST\_RX\_SMALL\_DGRAM**

Segments smaller than the header.

**ST\_RX\_SMALL\_PKT**

Segments larger than the frame.

**ST\_RX\_BAD\_ACK**

Received ACK for unsent data.

**ST\_RX\_BAD\_DATA**

Received data outside the window.

**ST\_RX\_LATE\_DATA**

Received data after close.

**ST\_RX\_OPT\_MSS**

Segments with the MSS option set.

**ST\_RX\_OPT\_OTHER**

Segments with other options.

**ST\_RX\_DATA**

Data segments received.

**ST\_RX\_DATA\_DUP**

Duplicate data received.

**ST\_RX\_ACK**

ACKs received.

**ST\_RX\_ACK\_DUP**

Duplicate ACKs received.

**ST\_RX\_RESET**

RST segments received.

**ST\_RX\_PROBE**

Window probes received.

**ST\_RX\_WINDOW**

Window updates received.

**ST\_RX\_SYN\_EXPECTED**

Expected SYN, not received.

**ST\_RX\_ACK\_EXPECTED**

Expected ACK, not received.

**ST\_RX\_SYN\_NOT\_EXPECTED**

Received SYN, not expected.

**ST\_RX\_MULTICASTS**

Multicast packets.

**ST\_TX\_DATA**

Data segments sent.

**ST\_TX\_DATA\_DUP**

Data segments retransmitted.

**ST\_TX\_ACK**

ACK-only segments sent.

**ST\_TX\_ACK\_DELAYED**

Delayed ACKs sent.

**ST\_TX\_RESET**

RST segments sent.

**ST\_TX\_PROBE**

Window probes sent.

**ST\_TX\_WINDOW**

Window updates sent.

**ST\_CONN\_ACTIVE**

Active open operations.

**ST\_CONN\_PASSIVE**

Passive open operations.

**ST\_CONN\_OPEN**

Established connections.

**ST\_CONN\_CLOSED**

Graceful shutdown operations.

**ST\_CONN\_RESET**

Ungraceful shutdown operations.

**ST\_CONN\_FAILED**

Failed open operations.

**ST\_CONN\_ABORTS**

Abort operations.

### 8.3.37 UDP\_STATS

A pointer to this structure is returned by [UDP\\_stats\(\)](#).

```
typedef struct {
    uint_32      ST_RX_TOTAL;
    uint_32      ST_RX_MISSED;
    uint_32      ST_RX_DISCARDED;
    uint_32      ST_RX_ERRORS;

    uint_32      ST_TX_TOTAL;
    uint_32      ST_TX_MISSED;
    uint_32      ST_TX_DISCARDED;
    uint_32      ST_TX_ERRORS;

    RTCS_ERROR_STRUCT ERR_RX;
    RTCS_ERROR_STRUCT ERR_TX;

    uint_32      ST_RX_BAD_PORT;
    uint_32      ST_RX_BAD_CHECKSUM;
    uint_32      ST_RX_SMALL_DGRAM;
    uint_32      ST_RX_SMALL_PKT;
    uint_32      ST_RX_NO_PORT;
} UDP_STATS, _PTR_ UDP_STATS_PTR;
```

#### ST\_RX\_TOTAL

Total number of received packets.

#### ST\_RX\_MISSED

Incoming packets discarded due to lack of resources.

#### ST\_RX\_DISCARDED

Incoming packets discarded for all other reasons.

#### ST\_RX\_ERRORS

Internal errors detected while processing an incoming packet.

#### ST\_TX\_TOTAL

Total number of transmitted packets.

#### ST\_TX\_MISSED

Packets to be sent that were discarded due to lack of resources.

#### ST\_TX\_DISCARDED

Packets to be sent that were discarded for all other reasons.

#### ST\_TX\_ERRORS

Internal errors detected while trying to send a packet.

#### ERR\_RX

RX error information.

**ERR\_TX**

TX error information.

The following stats are included in *ST\_RX\_DISCARDED*.

**ST\_RX\_BAD\_PORT**

Datagrams with the destination port zero.

**ST\_RX\_BAD\_CHECKSUM**

Datagrams with an invalid checksum.

**ST\_RX\_SMALL\_DGRAM**

Datagrams smaller than the header.

**ST\_RX\_SMALL\_PKT**

Datagrams larger than the frame.

**ST\_RX\_NO\_PORT**

Datagrams for a closed port.

## Appendix A Protocols and Policies

### A.1 Ethernet

Ethernet (IEEE 802.3) is the physical layer that RTCS supports. RFC 894 (a standard for the transmission of IP datagrams over ethernet networks) defines, how IP datagrams are sent in ethernet frames.

Properties of ethernet include:

- It is not deterministic.
- Delivery is unreliable (not guaranteed).
- All hosts on an ethernet network can receive all packets.
- Minimum frame length is 64 bytes.
- Maximum frame length is 1518 bytes.

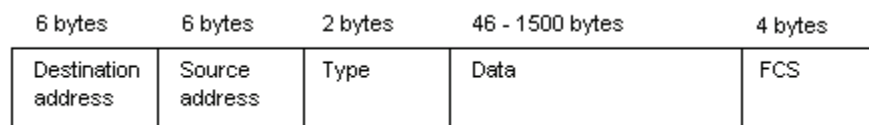


Figure A-1. Ethernet Frame

### A.2 ARP

Address Resolution Protocol (RFC 826) resolves a logical IP address to a physical ethernet address.

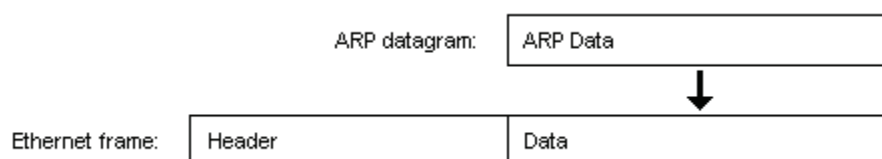
ARP maintains a local list of IP addresses and their corresponding ethernet addresses in a data structure called the ARP cache. When ARP initializes, the ARP cache is empty; that is, it contains no IP-to-ethernet address pairs. When a source host prepares a packet to send to a destination IP address on the local subnet, ARP examines its ARP cache to determine, whether it already knows the destination ethernet address. If ARP does not already know the ethernet address (which is the case immediately after ARP initializes), ARP broadcasts on the local subnet an ARP request that asks all hosts on the subnet, whether they are the destination IP address. All the hosts receive the ARP request, but only the destination host replies. The destination host sends directly to the source host (that is, it does not use a broadcast message) an ARP reply that contains the destination host's ethernet address.

When the source host receives the ARP reply, ARP places the destination host IP address and ethernet address in the ARP cache. ARP includes a timestamp with each entry and deletes the entry after two minutes.

0	8	16	24	31
Hardware type		Protocol type		
HLen	PLen		Operation	
Sender hardware address (octets 0-3)				
Sender hardware address (octets 4-5)		Sender IP (octets 0-1)		
Sender IP (octets 2-3)		Target hardware address (octets 0-1)		
Target hardware address (octets 2-5)				
Target IP				

**Figure A-2. ARP Datagram**

In an ethernet frame that contains an ARP datagram, the *Type* field contains 0x806.



**Figure A-3. ARP Datagram in an Ethernet Frame**

## A.3 IP

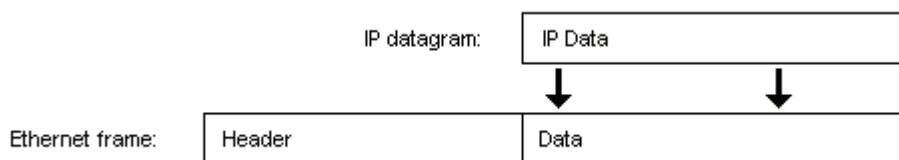
Internet Protocol (RFC 791) lets applications view multiple, interconnected physical networks as one single, logical network. IP provides an unreliable, connectionless datagram transport protocol between hosts in the logical network.

0	8	16	24	31
Vers	HLen	Service type	Total length	
Identification			Flags	Fragment offset
Time to live		Protocol	Header checksum	
Source IP address				
Destination IP address				
IP options (optional)				Padding
Data				

**Figure A-4. IP Datagram**



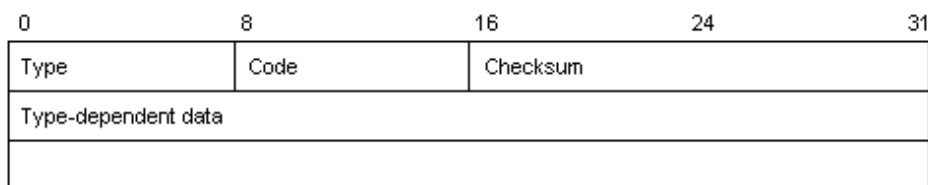
In an ethernet frame that contains an IP datagram, the *Type* field contains 0x800.



**Figure A-5. IP Datagram in an Ethernet Frame**

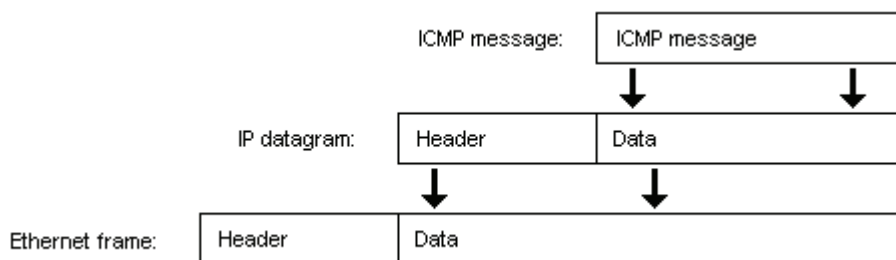
## A.4 ICMP

IP uses Internet Control Message Protocol (RFC 792) to send and receive error and status information.



**Figure A-6. ICMP Message**

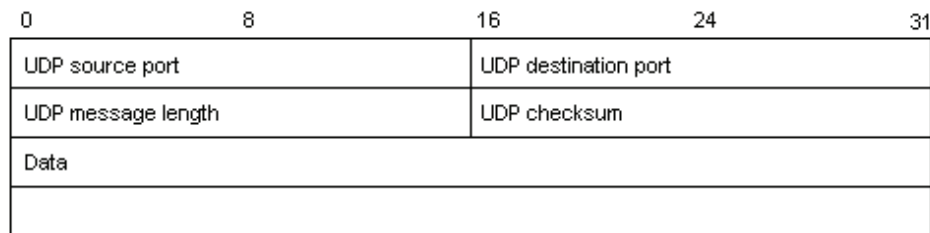
In an IP datagram that contains an ICMP message, the *Protocol* field contains one.



**Figure A-7. ICMP Message in an Ethernet Frame**

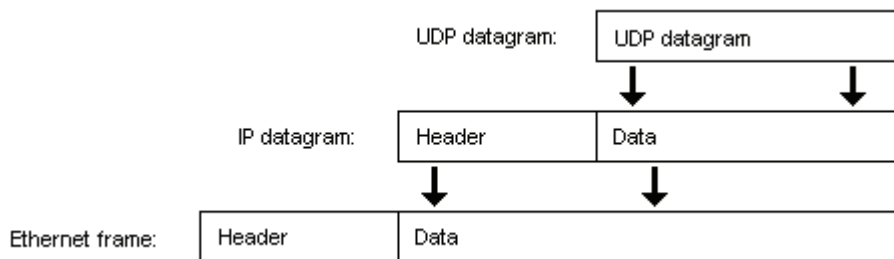
## A.5 UDP

User Datagram Protocol (RFC 768) provides the same unreliable, connectionless datagram transport protocol as IP. In addition, UDP adds to IP the concept of a source and a destination port, which lets multiple applications on source and destination hosts have independent communication paths. That is, an IP communication path is defined by the source IP address and the destination IP address. A UDP communication path is defined by the source port on the source host and the destination port on the destination host. Therefore, with UDP, it is possible to have multiple, independent communication paths between a source host and a destination host.



**Figure A-8. UDP Datagram**

In an IP datagram that contains a UDP datagram, the *Protocol* field contains 17.



**Figure A-9. UDP Datagram in an Ethernet Frame**

## A.6 TCP

Transmission Control Protocol (RFC 793) provides a reliable, stream-oriented transport protocol. TCP, like UDP, incorporates the concept of source and destination ports. However, TCP applications deal with connections, not endpoints. With UDP, any endpoint (IP address and port number) can communicate with any other endpoint. With TCP, before communication is possible, source and destination endpoints must first define a connection.

TCP differs from UDP in that TCP is:

- reliable
- stream-oriented
- connection-based
- buffered

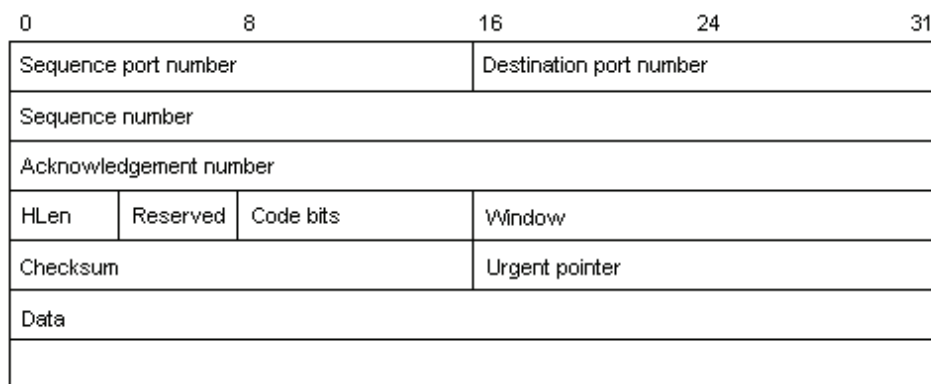


Figure A-10. TCP Segment

In an IP datagram that contains a TCP segment, the *Protocol* field contains six.

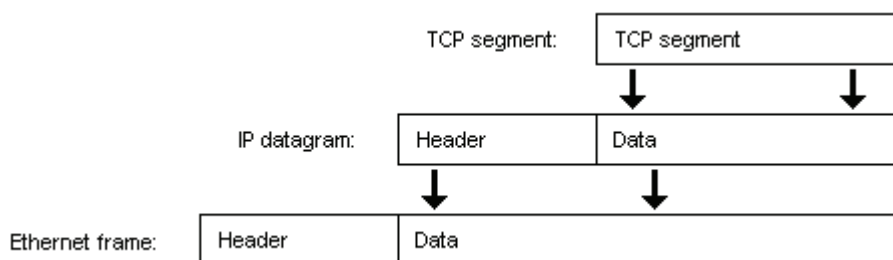


Figure A-11. TCP Segment in an Ethernet Frame

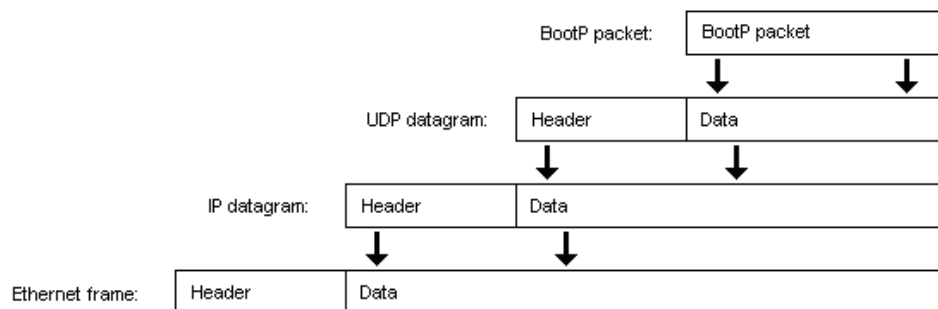
## A.7 BootP

Bootstrap Protocol (RFC 951) is used to get an IP address based on an ethernet address, load an executable boot file, and run that file.

BootP is built on top of UDP/IP and one of FTP, TFTP, or SFTP. The RTCS implementation of BootP uses TFTP. Applications that use BootP require a client and a server. RTCS provides the BootP client.

Bootstrapping consists of two phases:

- Phase one — The client determines its IP address, the server's IP address, and the boot filename using BootP. The client can override any of these values by specifying any of them.
- Phase two — The client transfers the file using TFTP.

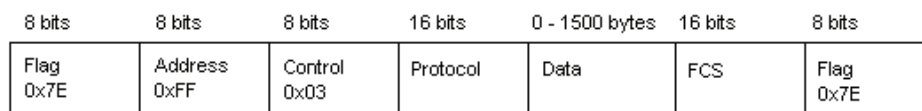


**Figure A-12. BootP Packet in an Ethernet Frame**

## A.8 HDLC

To encapsulate datagrams, PPP uses HDLC-like framing (RFC 1662). HDLC is an ISO protocol, defined in:

- ISO/IEC 3309:1991 (HDLC frame structure)
- ISO/IEC 4335:1991 (HDLC elements of procedures)



**Figure A-13. PPP Frame**

### A.8.1 Flag

Each frame begins and ends with a *Flag* field (0x7E), which PPP uses to synchronize frames. Only one flag is required between two frames. Two consecutive *Flag* fields constitute an empty frame, which PPP silently discards and does not count as an FCS error.

### A.8.2 Address

Always contains 0xFF, which is the HDLC all-stations (that is, broadcast) address. Individual station addresses are not assigned.

### A.8.3 Control

Always contains 0x03, the HDLC unnumbered information (UI) command.

### A.8.4 Protocol

Identifies the datagram that is encapsulated in the *Data* field. Values are listed in RFC 1700 (Assigned Numbers).

### A.8.5 Data

Contains the encapsulated packet.

### A.8.6 FCS

The frame-check sequence by default uses CCITT-16, and is calculated over all bits of the *Address*, *Control*, *Protocol*, and *Data* fields.

## 8.4 LCP

PPP uses Link Control Protocol (RFC 1661 (PPP) and RFC 1570 (LCP Extensions)) to negotiate options for a link.

In the process of maintaining the link, the PPP link goes through states, as shown in [Figure A-14](#).

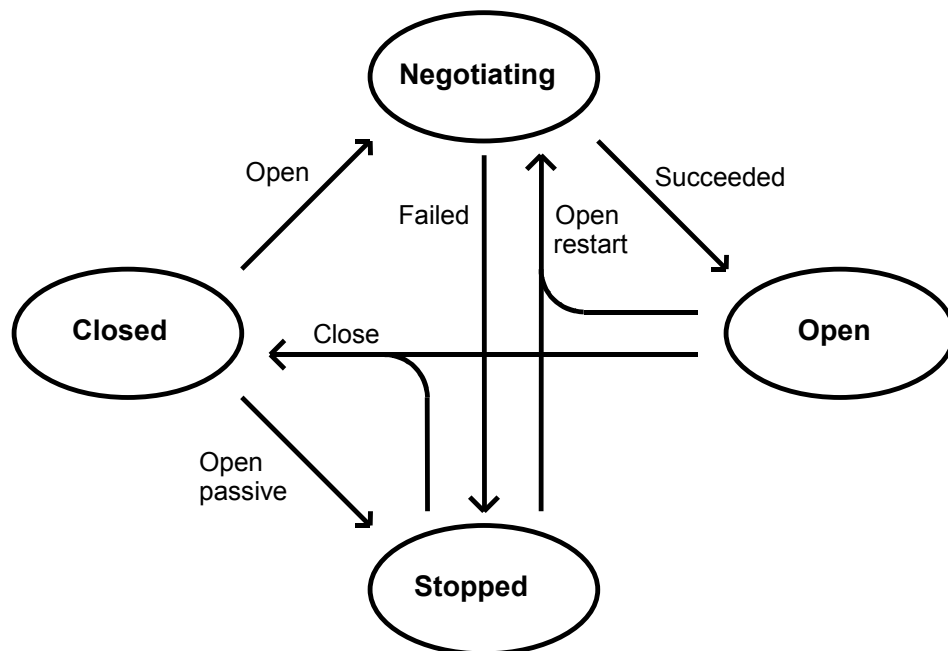


Figure A-14. PPP State Diagram

In the Closed state, PPP does not accept requests from the peer to open the link, nor does it allow the host to send packets to the peer.

In the Stopped state, PPP accepts requests from the peer to open the link, but still does not allow the host to send packets to the peer.

For the link to be opened, the link configuration must be negotiated first. If the negotiation is successful, the link is in the Open state, and available for an application to use. If the negotiation is not successful, the link is in the Stopped state.

## 8.5 SNTP

Simple Network Time Protocol (RFC 2030) operates over UDP at the IP layer for IPv4 to synchronize computer clocks on the Internet. RTCS clients can operate in unicast (point-to-point) or anycast (multi-point-to-point) mode.

### A.8.7 Unicast Mode

The client sends a request to a time server at its unicast address, then waits for a reply. The reply must contain the time, round-trip delay, and local clock offset relative to the server.

### A.8.8 Anycast Mode

The client sends a request to a local-broadcast or multicast-group address. One or more servers might reply with a unicast address. The client binds to the first received reply.

## 8.6 IPsec

IPsec (IP security) defines a set of protocols and cryptographic algorithms for creating secure IP traffic sessions between IPsec hosts. For more information, refer to one of the following RFCs:

- *PF\_KEY Key Management API, Version 2* (RFC 2367)
- *Security Architecture for the Internet Protocol* (RFC 2401)
- *IP Authentication Header* (RFC 2402)
- *The Use of HMAC-MD5-96 within ESP and AH* (RFC 2403)
- *The Use of HMAC-SHA-1-96 within ESP and AH* (RFC 2404)
- *The ESP DES-CBC Cipher Algorithm With Explicit IV* (RFC 2405)
- *IP Encapsulating Security Payload (ESP)* (RFC 2406)
- *HMAC: Keyed-Hashing for Message Authentication* (RFC 2104)
- *IP Security Document Roadmap* (RFC 2411)
- *The NULL Encryption Algorithm and Its Use With IPsec* (RFC 2410)

## 8.7 NAT

NAT helps to solve the problem of IP-address depletion. Under NAT, a few IP address ranges are reserved as private realms, and are not forwarded on the Internet. They can thus be reused by multiple organizations without risking address conflict. Public IP addresses must be globally unique; private IP addresses may be reused by any organization and need only be locally unique inside the organization. A NAT router acts as a gateway between the two realms. The router maps reusable, local IP addresses to globally unique addresses, and the other way around.

NAT allows hosts in a private network to transparently communicate with hosts outside of the network. NAT runs on the router that connects the private network to a public network, and it modifies all outbound packets that pass through the router by making the router the source of the packet.

When a reply is received for a specific packet, the router modifies the packet by setting the destination to be the private host that originally sent the packet.

For more information about NAT, see the following RFCs:

- *The IP Network Address Translator (NAT) (RFC 3022)*
- *IP Network Address Translator (NAT) Terminology and Considerations (RFC 2663)*

<b>NOTE</b>	When IP security (IPsec) is being used, the contents of IP headers (including the source and destination addresses) are protected from modification. Therefore, NAT and IPsec cannot be used together.
-------------	--





# Glossary

## A

---

### AC

Access concentrator in PPP over Ethernet. In the Discovery stage, a host (the client) discovers an access concentrator (the server). Based on the network topology, there might be more than one access concentrator, with which the host can communicate. The Discovery stage allows the host to discover all access concentrators, and then select one. When Discovery completes successfully, both the host and the selected access concentrator have the information they will use to build their point-to-point connection over ethernet.

### ACCM

Asynchronous control character map. One of the LCP configuration options for PPP Driver that you can modify. It is a set of control characters that have special meaning when they are sent over a PPP link and must, therefore, be escaped, when they are sent in a frame.

### active task

The task that is currently running on the processor, that is, the highest-priority ready task.

### address

When it is used alone, *address* refers to an IP address or a hardware address.

- All-stations address — HDLC term for broadcast address.
- Broadcast address — a hardware address or an IP address that refers to all hosts on a medium. The HDLC term for this is all-stations address.
- Ethernet address — ethernet term for the hardware address. A 48-bit address, where the least-significant bit of the first octet signifies, whether the address is a unicast address (the bit is zero) or a multicast address (the bit is one). An ethernet address is sometimes called a MAC address.
- Hardware address — address of a physical interface. A hardware address is sometimes called a media address.
- IP address — a 32-bit quantity that represents a point of attachment in an internet. It consists of a network portion and a host portion. The IP address is a multicast address, if the four most significant bits of the first octet have the binary value 1110. An IP address is sometimes called a logical address. See also address class and network mask. An IP address is sometimes called a logical address.
- Logical address — see IP address.
- MAC address — see ethernet address.
- Media address — see physical address.
- Multicast address — see ethernet address and IP address.
- Network address — a 32-bit address consisting of the network portion of an IP address and zeros in the host portion.
- Physical address — a sequent of bits that a medium uses to address a host. A physical address is sometimes called a media address.
- Unicast address — see ethernet address and IP address.

## **address class**

A method to determine the boundary between the network and host portions of an IP address. For example, Class A, Class B. See also network mask and CIDR.

## **address family**

The field in *sockaddr\_in* structure that identifies the family that RTCS supports for sockets. See also [AF\\_INET](#), [protocol family](#).

## **address-mask**

See [network mask](#).

## **address resolution**

Mapping an IP address onto a physical address.

## **Address Resolution Protocol**

See [ARP](#).

## **ACFC**

*Address-* and *Control*-field compression. One of the LCP configuration options for PPP Driver that you cannot change.

## **Address- and Control-fields compression**

See [ACFC](#).

## **AF\_INET**

Internet address family, using IP addresses. It is the address family that RTCS supports for sockets.

See also [PF\\_INET](#).

## **AH**

Authentication Header security protocol.

## **ALG**

See application-level gateway.

## **all-stations address**

See [address](#).

## **AP**

Authentication protocol for host authentication. One of the LCP configuration options for PPP Driver. The default is no AP. You can change it to CHAP, PAP, or both.

## **API**

Application Programming Interface.

## **application-level gateway**

Application-level gateways connect host and target applications transparently through NAT devices. Any application-level gateway can interact with NAT to set up or use state information and modify application-specific payload (add protocol-specific information to the NAT session entry). Any application that carries (or uses) an IP address inside the application will not work with NAT, unless the NAT is configured to do the appropriate translation. See also [NAT](#).

## **ARP**

Address Resolution Protocol, which resolves a logical IP address to a physical ethernet address.

## **ARP cache**

The cache that ARP uses to maintain a list of IP-to-ethernet address pairs.

## Assigned Numbers

RFC 1700.

## Asynchronous Control Character Map

See [ACCM](#).

## authentication

- Host authentication — a process whereby PPP determines, whether a host can use a link. PPP lets the host use the link only if authentication passes. See also [CHAP](#) and [PAP](#).
- Message authentication — the process whereby SNMPv3 associates a message with a particular originating entity during a session.

## authentication protocol

Used for host authentication only. See also [AP](#).

# B

---

## big-endian

The endian-format, in which the most-significant byte is the first byte in the word. This endian-format is also the network byte order. See also byte-swap, host byte order, and little-endian.

## BIN

Binary code. One format of a boot file. See also [COFF](#) and [SREC](#).

## binary code

See [BIN](#).

## bind (verb)

The action of completing a socket's local endpoint identification by providing the port number. Before it is bound, the socket is called an unbound socket; after, it is called a bound socket.

## block (verb)

When the active task blocks, MQX removes it from the ready queue and makes another task active.

See also [dispatch \(verb\)](#).

## boot file

Also called executable image and executable file.

## BootP

Bootstrap Protocol. The protocol that is used to get an IP address based on an ethernet address, load an executable file, and run that file.

## Bootstrap Protocol

See [BootP](#).

## broadcast address

See [address](#).

## BSD

See [UNIX BSD 4.4](#).

## **byte-swap**

Converts from one endian format to the other. See also [big-endian](#), [host byte order](#), [little-endian](#), and [network byte order](#).

# **C**

---

## **cache**

See [ARP cache](#).

## **callback function**

A function that an application provides and RTCS calls, when RTCS detects a certain event. The application can provide callback functions for DHCP (by calling **RTCS\_if\_bind\_DHCP()**) and for PPP (by calling **RTCS\_if\_bind\_IPCP()**).

## **carrier sense**

Each interface on a network can listen for carrier wave to determine, whether another interface is using the network. See also [CSMA/CD](#).

## **carrier sense multiple access with collision detection**

See [CSMA/CD](#).

## **CCITT**

International Telegraph and Telephone Consultative Committee. An international organization that sets standards for data communications, including algorithms for computing CRC checksums (such as [CCITT-16](#) and [CCITT-32](#)).

## **CCITT-16**

An algorithm proposed by CCITT for calculating a 16-bit CRC checksum.

## **CCITT-32**

An algorithm proposed by CCITT for calculating a 32-bit CRC checksum.

## **CCP**

Compression Control Protocol, which is used by PPP Driver.

## **Challenge Handshake Authentication Protocol**

See [CHAP](#).

## **CHAP**

Challenge Handshake Authentication Protocol. One of the authentication protocols that PPP Driver supports. See also [PAP](#).

## **checksum**

See CRC.

## **CIDR**

Classless Inter-Domain Routing.

## **Classless Inter-Domain Routing**

See [CIDR](#).

## **COFF**

Common Object File Format. One format of a boot file. See also [BIN](#) and [SREC](#).

## **collision**

Because signals take a finite time to travel from one end of a system to the other, the first bits of a sent frame do not reach all parts of the network simultaneously. Therefore, two interfaces might sense that the network is idle, and start to send their frames simultaneously. When this happens, there is a collision. See also [collision detection](#) and [CSMA/CD](#).

## **collision detection**

If ethernet detects the collision of signals, it stops the transmission and re-sends the frame.

## **Common Object File Format**

See [COFF](#).

## **Compression Control Protocol**

See [CCP](#).

## **configuration options**

See the individual LCP configuration options for PPP Driver: [ACCM](#), [ACFC](#), [AP](#), [MRU](#), and [PFC](#).

## **configure a link**

Negotiate configuration options for a PPP link.

## **connection**

A logical binding between two endpoints. For PPP, the term is link.

## **connection-based mode**

A transport protocol that has three distinct phases:

- Establishment — two or more users are bound to a connection.
- Data transfer — data is exchanged between the users.
- Release — the binding is terminated.

## **connectionless mode**

A transport protocol that has a single phase involving control mechanisms, such as addressing, in addition to data transfer.

## **CRC**

Cyclic redundancy check; a type of checksum. A small integer computed from a sequence of bits that is used to detect errors following the transmission of the bits. See also [CCITT-16](#) and [CCITT-32](#).

## **CSMA/CD**

Carrier sense multiple access with collision detection. The type of bus that ethernet uses to access a LAN.

With CSMA/CD, before an interface sends data to another interface on the network, it listens for carrier wave to determine, whether any interface is using the network. (The ability of all interfaces to listen for carrier wave is called carrier sense.) If no carrier wave is detected, the data is sent.

## **Cyclic Redundancy Check**

See [CRC](#).

## D

---

### **datagram**

The unit of transmission in the network layer (such as UDP or IP). A datagram can be mapped to one or more packets that are passed to the data-link layer. Sometimes called a message.

### **data-link layer**

That portion of the OSI model that is responsible for transmission, framing, and error control over a single communications link.

### **data-link layer protocol**

A protocol that controls data transmission and error detection over a physical medium.

### **delivery service**

See transport protocol.

### **device**

An object (hardware) at the physical layer. In RTCS, there are two types of devices:

- Ethernet devices (use IP-E)
- Serial devices (use PPP and IPCP)

### **device driver**

Software that communicates with and manages a device. In RTCS, there are two types of device drivers: ethernet drivers and PPP Driver.

### **device interface**

A number that identifies an interface for a device. A device can have multiple interfaces.

### **device number**

A 32-bit number that identifies the interface on a device.

### **DHCP**

Dynamic Host Configuration Protocol.

### **direct routing**

The process of sending an IP datagram, when the sender and destination are on the same IP network.

See also [indirect routing](#).

### **discard silently**

Discard the packet and do not process it in a way that creates network traffic. Examples of processing that can occur include logging the error and recording the event in a statistics counter.

### **discovery packet**

Packets during the Discovery stage of PPP over Ethernet; any one of [PADI packet](#), [PADO packet](#), [PADR packet](#), [PADS packet](#), or [PADT packet](#).

### **dispatch (verb)**

When MQX dispatches a task, it is in the process of examining the ready queues to determine, which task it will make active. MQX makes active the highest-priority task that has been the longest in the ready queue.

See also [block \(verb\)](#) and [schedule \(verb\)](#).

### **DNS**

Domain Name System. The system and the protocol that are defined in RFC 1035 — *Domain Names: Implementation and Specification*.

## **domain name server**

A server that implements the DNS protocol, which is defined in RFC 1035. Sometimes called a DNS server.

## **Domain Name System**

See DNS.

## **DoS**

A denial-of-service attack is characterized by an explicit attempt by attackers to prevent legitimate users of a service from using the service. Examples include attempts to flood a network, thereby preventing legitimate network traffic; attempts to disrupt connections between two machines, thereby preventing access to a service; attempts to prevent a particular individual from accessing a service; and attempts to disrupt service to a specific system or person.

## **dotted decimal notation**

A convention for writing IP addresses in textual format; for example, 192.100.140.2.

## **download**

Get a file from a server.

## **driver**

See *device driver*.

## **Dynamic Host Configuration Protocol**

See *DHCP*.

# **E**

---

## **encapsulate**

Add a header and perhaps a trailer to a layer's unit of transmission, to create the unit of transmission for the next-lower layer. IP encapsulates an ICMP message to create an IP datagram by prepending the IP header. Ethernet encapsulates an IP datagram to create an ethernet frame by prepending the ethernet header and appending the FCS.

## **endian format**

Byte ordering for processor's words: one of big-endian (the most significant byte is the first byte in the word) or little-endian (the most significant byte is the last byte in the word).

## **endpoint**

The IP address, transport protocol, and port number that define one end of a socket.

## **ENET**

Refers to an ethernet driver.

## **ESP**

Encapsulating Security Payload security protocol.

## **Ethernet**

Can refer to:

- IEEE 802.3 signaling protocol.
- An ethernet controller (hardware). Sometimes called an ethernet device and a MAC (media access controller).
- An ethernet driver (software).

## **ethernet address**

See [address](#).

## **executable image**

See [boot file](#).

## **executable file**

See [boot file](#).

## **External Data Representation**

See [XDR](#).

# **F**

---

## **FALSE**

Not TRUE; that is, zero.

## **FCS**

Frame Check Sequence. The result of an algorithm (usually a CRC) for determining, whether an error occurred during transmission of a frame.

## **File Transfer Protocol**

See [FTP](#).

## **finite lease**

DHCP defines mechanisms, whereby it can assign an IP address to a client for a limited time, which means it can reassign the IP address to another client later.

## **fragment**

One of:

- An IP datagram that contains only a portion of the payload from a larger IP datagram.
- Part of an RTCS PCB.

## **fragmentation**

The process of splitting an IP datagram into fragments.

## **frame**

The unit of transmission at the data-link layer, usually an ethernet frame. A frame includes a header, a trailer, or both, and some payload. A frame usually encapsulates one packet (the unit of transmission at the network layer); the exception occurs, when an IP datagram is fragmented into multiple smaller datagrams.

## **Frame Check Sequence**

See [FCS](#).

## **FTP**

File Transfer Protocol.



## G

---

### **gateway**

A layer-three (network-layer) relay. Sometimes called a router.

## H

---

### **hardware address**

See [address](#).

### **hardware interface**

Demarcation between hardware and software.

### **HDLC**

High-Level Data Link Control. A link-layer protocol proposed by ISO.

### **High-Level Data Link Control**

See [HDLC](#).

### **host**

An entity with an IP address.

### **host byte order**

The endian format of the host computer: one of big-endian or little-endian. See also [byte-swap](#) and [network byte order](#).

### **host identifier**

That portion of an IP address that identifies a host on the IP network. For example, for 192.168.0.1 and 192.168.0.2, one and two are the host identifiers. Sometimes called host number.

### **host name**

An identity, usually a textual identifier, that is associated with a host.

### **host-number**

See host-identifier.

### **HTTP**

Hypertext Transfer Protocol.

### **hub**

The ethernet term for a layer-one (physical-layer) relay. The general term is switch.

### **Hypertext Transfer Protocol**

See [HTTP](#).

**I/O**

Input/output. Refers to the transfer of commands or data across an interface.

**ICMP**

Internet Control Message Protocol.

**IGMP**

Internet Group Management Protocol.

**indirect routing**

The process of sending an IP datagram to a router, which forwards the datagram to its destination. See also direct routing.

**interface**

One of:

- API (software interface).
- Demarcation between hardware and software (hardware interface).

**interface layer**

The layer in the internet suite of protocols that is responsible for transmission on a single physical network.

**Internet Control Message Protocol**

See [ICMP](#).

**Internet Group Management Protocol**

See [IGMP](#).

**Internet Protocol**

See [IP](#).

**Internet Protocol Control Protocol**

See [IPCP](#).

**interoperate**

In RTCS: the ability to communicate with a host that uses another protocol, or another version of a protocol; for example, with RTCS, an application can communicate fully with hosts that support RIPv2, and interoperate with hosts that support RIPv1.

**IP**

Internet Protocol. The network protocol that offers a connectionless-mode network protocol.

**IP address**

See [address](#).

**IPCP**

Internet Protocol Control Protocol. For RTCS, it is the network-control protocol for IP.

**ISO**

International Organization for Standardization, an international organization that sets standards for network protocols, such as HDLC.

---

## K

---

### KB

Kilobyte.

## L

---

### LAN

Local Area Network. Any technology that provides high-speed transfer over a geographically limited area.

### LANCE

Local Area Network Controller for Ethernet. An ethernet device from AMD Inc.

### LCP

Link Control Protocol. The protocol that PPP uses to negotiate options over a link.

### lease

See *finite lease*.

### library

See *API*.

### link (noun)

A network of two hosts that use PPP to communicate. See also *configure a link*.

### Link Control Protocol

See *LCP*.

### link layer

See *data-link layer*.

### little-endian

The endian format, in which the most significant byte is the last byte in the word. See also *big-endian*, *byte-swap*, *host byte order*, *network byte order*.

### local

For a datagram or stream socket, the endpoint or part of the endpoint (such as the IP address or port number) that is “here.” It is the endpoint that is associated with the function that called **socket()**. See also *remote*.

For a PPP link, it refers to the end of the link that is “here.” It is the endpoint that is associated with the function that called **PPP\_initialize()**. See also *peer*.

### Local Area Network

See *LAN*.

### logical address

See *address*.

### loopback address

See *address*.

## M

---

### **MAC**

Medium Access Control. The MAC mechanism is based on CSMA/CD.

Media Access Controller. See *Ethernet controller* under [Ethernet](#).

### **MAC address**

See [address](#).

### **Management Information Base**

See [MIB](#).

### **mask**

See [network mask](#).

### **maximum receive unit**

See [MRU](#).

### **maximum transmission unit**

See [MTU](#).

### **MB**

Megabyte.

### **Mb**

Megabit.

### **MD5**

Message-digest algorithm. Used by PPP Driver for CHAP authentication.

### **media**

Plural of medium.

### **media address**

See [address](#).

### **medium**

The physical connection between hosts; for example, a cable. Singular of media.

### **Medium Access Control**

See [MAC](#).

### **message**

See [datagram](#).

### **message-digest algorithm**

See [MD5](#).

### **MIB**

Management Information Base (RFCs 1213 and 2287).

### **MQX**

MQX. The RTOS that RTCS uses. MQX is an RTOS for single-processor, multi-processor, and distributed-processor, embedded, real-time applications.

### **MRU**

Maximum receive unit. One of the LCP configuration options for PPP Driver that you cannot change.

## MTU

Maximum transmission unit. The largest amount of user data (for example, the largest size of an IP datagram) that can be sent in a single frame on a particular medium.

## multicast address

See [address](#).

## multicast group

One of:

- Set of hosts that have joined a multicast group.
- Same as the multicast address.

## multihomed

A host that has multiple IP addresses.

## multiple access

All interfaces are equally able to send frames onto the network. No interface has a higher priority than another. See also [CSMA/CD](#).

# N

---

## Nagle algorithm

RTCS uses the Nagle algorithm (defined in RFC 896) to coalesce short segments for stream sockets.

## name

See [host name](#).

## name server

A server that maps host names to their associated attributes.

## NAT

Network Address Translation. The ‘traditional’ or ‘outbound’ method of routing IPv4 datagrams transparently, in which public IP addresses are mapped to unregistered private addresses and the other way around. The router between the public and private networks performs session-based translation. See also [application-level gateway](#) and [session](#).

## NCP

Network-Control Protocol. Network-control protocols negotiate a PPP link for use by a network-layer protocol. See also [IPCP](#).

## negotiate

See [configure a link](#).

## net

See [network](#).

## network

A physically connected set of hosts.

## network address

See [address](#).

### **network byte order**

Big-endian format. So that computers with different endian formats can communicate, the Internet protocols specify this canonical byte-order convention for data transmitted over the network. See also [host byte order](#).

### **Network Control Protocol**

See [NCP](#).

### **network-identifier**

That portion of an IP address that corresponds to a network or the internet.

### **network layer**

That portion of the OSI model that is responsible for data transfer across the network, independent of both the media that make up the underlying subnetworks and the topology of the subnetworks.

### **network-layer protocol**

A protocol that routes datagrams from the original source to the final destination over physical networks. Because network-layer protocols are independent of the medium, they use link-layer protocols to send packets over individual physical networks.

### **network mask**

A 32-bit quantity that indicates, which bits in an IP address refer to the network portion. Sometimes called net mask or address mask.

### **node**

See [host](#).

### **NULL FCS**

No Frame Check Sequence for PPP frames. See also [CCITT-16](#) and [CCITT-32](#).

## **O**

---

### **opaque data**

Data, whose interpretation is unknown by a layer, because the layer does not have the XDR translation function.

### **Open Systems Interconnection**

See [OSI](#).

### **option**

In RTCS, the following have options associated with them:

- DHCP
- IGMP
- IPCP
- PPP links
- sockets

### **OSI**

Open Systems Interconnection. An international body to facilitate communication among computers of different manufacturers and technologies.

## P

---

### **packet**

The unit of transmission at the network layer and passed between the network layer and the data-link layer. A packet is usually mapped to one frame (the unit of transmission at the data-link layer); the exception occurs, when an IP datagram is fragmented into multiple, smaller datagrams. See also [segment](#).

### **Packet Control Block**

See [PCB](#).

### **PADI packet**

Active Discovery Initiation packet in the Discovery process of PPP over Ethernet.

### **PADO packet**

Active Discovery Offer packet in the Discovery process of PPP over Ethernet.

### **PADR packet**

Active Discovery Request packet in the Discovery process of PPP over Ethernet.

### **PADS packet**

Active Discovery Session-confirmation packet in the Discovery process of PPP over Ethernet.

### **PADT packet**

Active Discovery Terminate packet in the Discovery process of PPP over Ethernet.

### **PAP**

Password Authentication Protocol. One of the authentication protocols that PPP Driver supports. See also [CHAP](#).

### **passive open**

The sequence of events that occurs when an application entity informs TCP that it is willing to accept connections.

### **Password Authentication Protocol**

See [PAP](#).

### **payload**

Data portion in an IP datagram.

### **PCB**

Packet Control Block, which RTCS uses to hold a packet.

### **peer**

The non-local end of a PPP link. See also [local](#), [remote](#).

### **PFC**

Protocol-Field Compression. One of the LCP configuration options for PPP Driver that you cannot change.

### **PF\_INET**

Protocol family, using IP addresses. It is the one protocol family that RTCS supports for sockets.

See also [AF\\_INET](#).

### **physical address**

See [address](#).

### **physical layer**

That portion of the OSI model that is responsible for the electromechanical interface to the communications media.

## **ping**

A program that tests IP-level connectivity from one IP address to another.

## **Point-to-Point Protocol**

See *PPP*.

## **port**

See *port number*.

## **Portmapper**

The RPC program that maps remote programs to the port numbers, to which they were bound.

## **port number**

A 16-bit number that identifies an application entity to a transport protocol. Sometimes called a port.

## **POSIX**

Portable Operating System Interface, produced by IEEE and standardized by ANSI and ISO.

## **PPP**

Point-to-Point Protocol. A link-layer protocol for sending multi-protocol datagrams over point-to-point links.

## **PPP Driver**

The RTCS component that manages PPP devices.

## **PPP over Ethernet**

(PPPoE) A protocol that specifies, how to build PPP sessions and encapsulate PPP packets over ethernet. PPPoE has two distinct stages — a Discovery stage and a PPP Session stage. When a host wants to initiate a PPPoE session, it must first perform Discovery to identify the ethernet MAC address of the peer and establish a PPPoE session ID. While PPP defines a peer-to-peer relationship, Discovery is inherently a client-server relationship. In the Discovery stage, a host (the client) discovers an access concentrator (the server). Based on the network topology, there might be more than one access concentrator, with which the host can communicate. The Discovery stage allows the host to discover all access concentrators, and then select one. When Discovery completes successfully, both the host and the selected access concentrator have the information they will use to build their point-to-point connection over ethernet.

## **MQX**

The RTOS that RTCS uses. MQX is an RTOS for single-processor, multi-processor, and distributed-processor, embedded real-time applications.

## **RTCS**

A real-time embedded internet stack.

## **procedure**

See *remote procedure*.

## **program**

See *remote program*.

## **protocol family**

A collection of transport protocols for sockets. RTCS supports one: *PF\_INET* (protocol family using IP addresses). See also *address family*.

## **Protocol-Field Compression**

See *PFC*.

## **push**

The push flag, which forces data delivery through a stream socket.



---

## Q

---

### Quote Protocol

Quote of the Day Protocol.

## R

---

### ready queue

MQX maintains a linked list of ready queues, one ready queue for each task priority. Each ready queue holds tasks that have the specific priority and that are in the ready state. To distinguish it from other ready queues, the ready queue that a task is in is referred to as the task's ready queue.

### ready task

A task that is on the ready queue for its priority.

### remote

The non-local endpoint (or part of the endpoint) of a stream socket. See also *local*, *peer*.

### remote procedure

An application makes a remote procedure call by calling `clnt_call()`.

### Remote Procedure Call

See *RPC*.

### remote program

A collection of remote procedures.

### Request for Comments

See *RFC*.

### RFC

Request for Comments. The Internet Standards Committee documentation for protocol standards.

### RIP

Routing Information Protocol. RTCS supports RIPv2 and interoperates with RIPv1.

### Round-Trip Time

See *RTT*.

### router

See *gateway*.

### routing

See *direct routing* or *indirect routing*.

### Routing Information Protocol

See *RIP*.

### RPC

Remote Procedure Call protocol. See also *XDR*.

## **RPC program**

The description of a set of RPC procedures. Along with the procedures, the description includes their return values, parameters, and the data types for the return values and parameters. One RPC program is usually associated with one socket.

## **RTCS**

RTCS, an embedded internet stack, optimized to run on the MQX RTOS.

## **RTOS**

Real-Time Operating System.

## **RTT**

Round-Trip Time. The time it takes for a signal to get from one end of the complete media system and back.

# **S**

---

## **schedule (verb)**

MQX finds the next ready task and makes it the active task.

## **segment**

The unit of transmission for TCP; the TCP term for *packet*.

## **server**

An application that is above the transport layer and that usually has a well-known endpoint.

## **service name**

In PPP over Ethernet, the service name tag can indicate an ISP name or a class or quality of service.

## **session**

The set of traffic that is managed as a unit for Network Address Translation (NAT). UDP sessions are identified by the tuple of source IP address, source UDP port, target IP address, and target UDP port. ICMP query sessions are identified by the tuple of source IP address, ICMP query ID, and target IP address. All other sessions are identified through the tuple of source IP address, target IP address, and IP protocol. See also *NAT*.

## **session stage (PPPoE)**

PPPoE has two distinct stages — a Discovery stage and a PPP Session stage.

## **silently discard**

See *discard silently*.

## **Simple Network Management Protocol**

See *SNMP*.

## **SMI**

Structure of Management Information.

## **SNMP**

Simple Network Management Protocol.

## **SOCK\_DGRAM**

Literal that indicates a datagram socket type.

## **SOCK\_STREAM**

Literal that indicates a stream socket type.

## **socket**

An object that an application uses to communicate with a remote endpoint. A socket is independent of the transport protocol.

## **socket type**

Datagram (*SOCK\_DGRAM*) or stream (*SOCK\_STREAM*).

## **software interface**

See [API](#).

## **SREC**

Motorola S-Record. A file format for boot files. See also [BIN](#) and [COFF](#).

## **S-Record**

See [SREC](#).

## **station**

An ethernet-equipped computer.

## **stream**

The sequential nature of TCP data transfers.

## **Structure of Management Information**

See [SMI](#).

## **switch**

See [hub](#).

## **system resource**

In the MQX sense, allocated memory might be a system resource, which means that any task can use it and any task can free it.

# **T**

---

## **task**

A body of C code (usually an infinite loop) that performs some function. There can be multiple instances of a task. Unless it would create confusion, the instance of a task is simply called the task.

## **task error code**

The error code that MQX assigns to a task, if certain calls to MQX functions cause an error.

## **TCP**

Transmission Control Protocol.

## **TCP/IP**

The entire IP stack.

## **Telnet**

A network protocol for a virtual terminal.

## **TFTP**

Trivial File Transfer Protocol.

## **Transmission Control Protocol**

See [TCP](#).

### **transport layer**

That portion of the OSI system that is responsible for the reliability and multiplexing of data transfer across the network, over and above, which the network layer provides.

### **transport-layer protocol**

A protocol that provides transparent transfer of data between network hosts. Transport-layer protocols rely on network-layer protocols to route datagrams over a network.

### **transport protocol**

One of UDP or TCP. Sometimes called a delivery service.

## **Trivial File Transfer Protocol**

See [TFTP](#).

### **TRUE**

Not FALSE; that is, any non-zero value.

## **U**

---

### **UDP**

User Datagram Protocol.

### **UI command**

Unnumbered Information command (HDLC).

### **UNIX BSD 4.4**

Berkeley distribution of the UNIX operating system, version 4.3.

### **unnumbered information command**

See [UI command](#).

### **User Datagram Protocol**

See [UDP](#).

## **W**

---

### **wait**

Blocking I/O on a stream socket, where wait implies wait for data to be sent or received.

## **X**

---

### **XDR**

External Data Representation protocol. Used with RPC.

## Index Symbols

- `_iocb_handle`, 217
- `_iocb_table`, 217
- `_IP_forward`, 11
- `_PPP_ACCM`, 38
- `_PPP_CHAP_LNAME`, 37
- `_PPP_CHAP_LSECRETS`, 37
- `_PPP_CHAP_RSECRETS`, 37
- `_PPP_MAX_CONF_NAKS`, 38
- `_PPP_MAX_CONF_RETRIES`, 38
- `_PPP_MAX_TERM_RETRIES`, 38
- `_PPP_MAX_XMIT_TIMEOUT`, 37
- `_PPP_MIN_XMIT_TIMEOUT`, 37
- `_PPP_PAP_LSECRET`, 37
- `_PPP_PAP_RSECRETS`, 37
- `_PPPTASK_priority`, 38
- `_PPPTASK_stacksize`, 37
- `_RTCSPCB_max`, 11
- `_RTCSTASK_priority`, 11
- `_RTCSTASK_stacksize`, 11
- `_TCP_bypass_rx`, 11
- `_TCP_bypass_tx`, 11

## A

- ACCM, 35, 36
  - minimal, 37
- ACFC, 36
- address family (internet), 84, 117, 118, 226
- address field structure (in `_addr`), 231
- address probing (DHCP), 52, 101, 102
- AF\_INET, 84, 117, 118, 226
- AP, 36
- ARC NAT
  - ALGs, disabling, 15
  - enabling, 11, 132
  - enabling and starting, 14
  - inactivity timeout socket option, 190
  - inactivity timeout structure (nat\_timeouts), 242
  - inactivity timeouts, changing, 15
  - limitations, 16
  - port numbers socket option, 191
  - port ranges, specifying, 15
  - port-numbers structure (nat\_ports), 240
  - protocols, supported, 16
  - starting, 132
  - statistics, 16, 133
  - statistics (NAT\_STATS), 241
  - stopping, 131
  - timeouts, changing, 15
- ARP, 6, 265

- statistics

- getting, 82
- structure, 219

- ARP\_STATS structure, 219

- Assigned Numbers protocol, 6, 270

- authentication protocols, 37

- changing, 38

## B

- BIN boot files

- loading, 171

- loading and running, 14, 144

- binding IP addresses to device interfaces, 13, 155

- using BootP, 156

- using DHCP, 158, 160, 162, 166

- using IPCP, 164

- boot files

- loading

- BIN format, 171

- COFF format, 172

- S-Record format, 173

- loading and running

- BIN format, 14, 144

- COFF format, 14, 146

- S-Record format, 14, 147

- BootP, 6, 269

- performing, 13

- using to bind an IP address, 13, 156

- BOOTP\_DATA\_STRUCT, 221

- bootstrapping, 269

- performing, 13

- Phase 2, performing, 14, 144, 146, 147

- Broadcast Datagrams protocol, 7

- Broadcasting Internet Datagrams in the Presence of Subnets, 7

- BSD 4.4, 23

## C

- call graphs *see* code paths

- callback functions

- DHCP, 11, 158, 160, 162, 166

- Ethernet, 154, 253

- IPCP, 11, 164

- PPP, 41, 67, 154, 253

- PPP over Ethernet, 244, 245, 246

- CCITT-16, 271

- CCP, 6, 41, 135

- CHAP, 6, 37, 38, 39

- example, 39

- checksum bypass socket option, 187

- checksums (RTCS), bypassing, 11

- CIDR, 6

- client (DHCP), releasing, 97

- code paths, 58

- code size, reducing, 255
- COFF boot files
  - loading, 172
  - loading and running, 14, 146
- community string (SNMPv1, SNMPv2c), 56
- compile-time options, configuring RTCS, 19
  - recommended settings, 19
- configuring RTCS
  - compile-time options, 19
    - recommended settings, 19
  - creation parameters, 11
  - running parameters, 11, 23
- connect timeout socket option, 187, 188
- connections, establishing for stream sockets, 29
  - accept(), 80
  - connect(), 86
  - listen(), 129
- creation parameters (RTCS), 11

## D

- data types
  - for portability, 215
- datagram sockets
  - about, 24
  - comparison with stream sockets, 24
- data
  - receiving, 28, 138
  - sending, 28, 184
- options
  - getting, 124
  - setting, 27, 186
- shutting down, 28
- using, 27
- debugging, embedded, 55
- debugging, over Winsock connection, 55
- device interfaces
  - adding to RTCS, 13, 154
  - binding to IP addresses, 13, 155
    - using BootP, 156
    - using DHCP, 158, 160, 162, 166
    - using IPCP, 164
- initializing, 12
  - Ethernet, 12, 107
  - point-to-point, 12
  - PPP, 41, 135
- removing from RTCS, 13, 169
- unbinding IP addresses, 13, 170
- DHCP, 6
  - callback functions, 11, 158, 160, 162, 166
  - Client, 51
    - example, 98, 158, 160, 162, 167
    - releasing, 97
  - options, adding
    - 16-bit quantity, 91

- 32-bit quantity, 92
- 8-bit quantity, 93
- IP address, 89
- IP address list, 90
- string, 94
  - variable-length quantity, 95
- options, searching for, 88, 96
- performing, 13
- Server, 52
  - adding block of IP addresses, 100
  - information, 175
  - starting, 98
  - using to bind an IP address, 13, 158, 160, 162, 166
- DHCP Options and BootP Vendor Extensions, 6
- DHCP\_DATA\_STRUCT, 222
- DHCPSRV\_DATA\_STRUCT, 223
- dhshosts.c, 54
- directory structure (RTCS), 61
- DNS, 7
  - Client, 53
    - services, 54
    - starting, 103
  - host from IP address, getting, 117
  - host from name, getting, 118
- DNS Resolver, 53
- DNS server, 53
- DNS\_Local\_network\_name, 54
- DNS\_Local\_server\_name, 54
- document conventions, 3
- DoS attacks (Smurf), reducing risk of, 21

## E

- Echo protocol, 7
  - Server, 54
    - starting, 104
- EDS Server (Winsock)
  - opening connection to, 55
  - starting, 105
- embedded debugging, 55
- enet.h, 216
- ENET\_STATS structure, 224
- Ethernet
  - callback functions, 154, 253
- Ethernet 802.1Q priority tags, 191, 194
- Ethernet 802.3 frames, 192, 194
- Ethernet connection, for embedded debugging, 55
- Ethernet drivers
  - initializing, 12, 107
  - statistics
    - getting, 12, 106
    - structure, 224
- Ethernet II frames, 21
- Ethernet protocol, 7, 265
- Ethernet802.1Q priority tags, 21

Ethernet802.3 frames, 21

#### examples

- binary boot file, downloading and running, 144

- DHCP Client, 98, 158, 160, 162, 167

- echoing data

  - TCP, 176

  - UDP, 178

- FTP

  - server, starting, 116

  - session, starting, 114

- gateways

  - adding default, 149

  - removing default, 151

- host HOSTENT\_STRUCT, getting by name, 119

- host name, getting by address, 117

- IP addresses, assigning with BootP, 156

- NAT, max port number, 198

- PAP and CHAP authentication, setting up, 39

- PPP

  - binding to interface, 164

- PPP Driver, 17

- PPP over Ethernet, 70

- RTCS

  - creating and setting up, 17

  - logging, 181

  - received-packets statistics, displaying, 82

- SNMP Agent, 201

- SNTP Client, 202

- sockets

  - accepting incoming connections, 80

  - binding to port number, 84

  - checksum-bypass option, 198

  - local endpoint, getting, 122

  - max NAT port number option, 198

  - receive-nowait option, 197

  - receiving data

    - datagram, 139

    - stream, 137

  - remote endpoint, getting, 120

  - sending data

    - datagram, 185

    - stream, 183

  - send-push option, 197

  - shutting down, 200

  - stream, connecting to, 87

  - using, 140

  - various timeout options, 198

- sockets, using, 32

- S-Record boot file, downloading and running, 147

- TFTP Server, 211

## F

FCS, 271

FLAG\_ABORT\_CONNECTION, 199

FLAG\_CLOSE\_TX, 199

flowcharts, 58

FTP, 7

- Client, 55

  - issuing commands to FTP server, 109

  - issuing commands to FTP server that requires data connection, 110

  - starting a session, 114

  - terminating a session, 108

- Server, 55

  - starting, 116

## G

gateway metric, 150, 152

gateways

- adding to RTCS, 13, 149

- metrics, 150, 152

- removing from RTCS, 14, 151

get host by address (DNS), 117

get host by name (DNS), 118

get peer name (stream sockets), 120

get socket name (stream sockets), 122

## H

hard timeout socket option, 188

HDLC, 7, 270

- escape character, 36

HDLC-like framing device (PPP)

- initializing driver, 68

host by address (DNS), getting, 117

host by name (DNS), getting, 118

HOSTENT\_STRUCT, 226

HTTP, 7

## I

I/O PCB

- handle

  - destroying

    - for PPPoE Client, 69

  - getting

    - for HDLC, 68

    - for PPPoE Client, 70

  - structure, 217

- opening driver for PPP, 67

- table, structure, 217

ICMP, 7, 267

- statistics

  - getting, 125

  - structure, 227

ICMP\_STATS structure, 227

IGMP, 7

- add membership socket option, 188

- drop membership socket option, 189
- get membership socket option, 189
- in RTCS protocol table, 255
- statistics
  - getting, 126
  - structure, 230
- IGMP\_STATS structure, 230
- in\_addr structure, 231
- INADDR\_ANY, 84
- information
  - on DHCP server, 175
- initial transmission timeout socket option, 189
- Internet Standard Subnetting Procedure, 7
- IP, 7, 266
  - ICMP and, 267
  - statistics
    - getting, 127
- IP addresses
  - binding to device interfaces, 13, 155
    - using BootP, 156
    - using DHCP, 158, 160, 162, 166
    - using IPCP, 164
  - unbinding device interfaces, 13, 170
- IP forwarding
  - enabling, 11
- IP multicast group structure (ip\_mreq), 232
- ip\_mreq structure, 232
- IPCP, 7, 35
  - callback functions, 11, 164
  - using to bind an IP address, 13, 164
- IPCP\_DATA\_STRUCT, 236
- IP-E, 7
- IPIF
  - statistics
    - getting, 42, 128
- IPIP, 7
- IPsec, 272

## K

- keep-alive timeout socket option, 190

## L

- LCP, 7, 35, 41, 135, 271
  - configuration optionsspan\_lcp\_options, 35
- links (PPP)
  - initializing, 12, 41, 135
- logging
  - disabling, 14, 180
  - enabling, 14, 181

## M

- maximum retransmission timeout socket option, 190

- maximum segment lifetime, 197, 199
- MD5, 7
- metric, for gateway, 150, 152
- MIB, 7
- MIB-1213, installing, 57, 130
- MRU, 36, 37

## N

- NAT, 7, 272
- NAT\_alg\_table, 16
- nat\_ports structure, 240
- NAT\_STATS structure, 241
- nat\_timeouts structure, 242
- network address translation
  - starting, 132
  - statistics, 133
  - stopping, 131
- no Nagle algorithm socket option, 191

## O

- OPT\_CHECKSUM\_BYPASS, 187
- OPT\_CONNECT\_TIMEOUT, 187
- OPT\_KEEPAIVE, 190
- OPT\_MAXRTO, 190
- OPT\_NO\_NAGLE\_ALGORITHM, 191
- OPT\_RBSIZE, 193
- OPT\_RECEIVE\_NOWAIT, 188, 192
- OPT\_RECEIVE\_PUSH, 193
- OPT\_RECEIVE\_TIMEOUT, 193
- OPT\_RETRANSMISSION\_TIMEOUT, 189
- OPT\_SEND\_NOWAIT, 194, 195
- OPT\_SEND\_PUSH, 195
- OPT\_SEND\_TIMEOUT, 195
- OPT\_SOCKET\_ERROR, 196
- OPT\_SOCKET\_TYPE, 196
- OPT\_TBSIZE, 196
- OPT\_TIMEWAIT\_TIMEOUT, 197

## P

- PAP, 7, 36, 37, 38
  - example, 39
- parameters (RTCS)
  - creation, 11
  - running, 11, 23
- PCBs
  - max (RTCS), 11
  - opening driver for PPP, 67
- PCB handle
  - destroying
    - for PPPoE Client, 69
  - getting
    - for HDLC, 68



- for PPPoE Client, 70
  - structure, 217
- PCB table, structure, 217
- peer name (stream sockets), getting, 120
- PF\_INET, 205
- PFC, 36, 37
- ping, 7, 174
- PPP, 7, 35
  - callback functions, 41, 67, 154, 253
  - frame, 270
- PPP Driver, 35
  - authentication protocols, 37
    - changing, 38
    - example, 39
  - configuring, 37
  - example, 17
  - function summary, 42
  - HDLC-like framing device, initializing, 68
  - I/O PCB driver, opening, 67
  - initializing, 12, 41, 135
  - IP addresses, binding to, 164
  - LCP configuration optionsspan\_lcp\_options, 35
  - PPP over Ethernet framing device, destroying, 69
  - PPP over Ethernet framing device, initializing, 70
  - stack size, additional, 37
  - statistics
    - getting, 42, 128
  - task priority, 38
- PPP in HDLC-like Framing, 7
- PPP LCP Extensions, 7
- PPP links, initializing, 41, 135
- PPP over Ethernet
  - callback functions, 244, 245, 246
  - destroying, 69
  - example, 70
  - function summary, 43
  - initializing, 12, 70
  - setting up, 42
  - statistics (PPPOE\_SESSION\_STATS\_STRUCT), 247
  - statistics (PPPOEIF\_STATS), 248
  - statistics, getting, 73, 77, 79
- ppp.h, 216
- PPP\_SECRET structure, 251
- PPPoE, 7
- PPPoE Server
  - adding an Ethernet interface, 75
  - destroying task, 74
  - initializing, 78
  - removing an Ethernet interface, 76
  - statistics
    - getting Ethernet, 77
    - getting PPP session, 79
- pppoe.h, 216
- PPPOE\_CLIENT\_INIT\_DATA\_STRUCT structure, 243
- PPPOE\_SERVER\_INIT\_DATA\_STRUCT structure, 245

- PPPOE\_SESSION\_STATS\_STRUCT structure, 247
- PPPOEIF\_STATS structure, 248
- priority of tasks
  - PPP Driver, 38
  - RTCS, 11
- protocol family (internet), 205
- protocol stack (RTCS), 9
- protocols (RTCS)
  - defining, 10

## Q

- Quote of the Day protocol, 7, 58
- Quote of the day service, 58

## R

- R2 timeout socket option, 188
- receive buffer size socket option, 193
- receive Ethernet 802.1Q priority tags socket option, 191
- receive Ethernet 802.3 frames socket option, 192
- receive nowait socket option, 30, 192
- receive push socket option, 30, 193
- receive timeout socket option, 193
- reducing code size, 255
- references
  - documentation, 3
- release DHCP client, 97
- Requirements for Internet Hosts, 7
- Requirements for IP Version 4 Routers, 7
- retransmission timeout socket option, 189
- RFCs supported, 6
- RIP, 7
  - in RTCS protocol table, 255
- round-trip time (stream sockets), 190
- RPC, 7
- rpctypes.h, 216
- RTCS
  - checksums, bypassing, 11
  - compile-time options, 19
    - recommended settings, 19
  - creating, 11, 142
  - directory structure, 61
  - example (creating and setting up), 17
  - function summary, 17
  - gateways
    - adding, 13, 149
    - metrics, 150, 152
    - removing, 14, 151
  - logging, 14
  - parameters
    - creation, 11
    - running, 11, 23
  - PCBs (max), 11
  - protocol stack, 9

- protocols
  - defining, 10
  - setting up
    - overview, 10
  - socket error, getting, 153
  - stack size, additional, 11
  - statistics
    - Ethernet
      - getting, 12, 106
      - structure, 224
    - PPP
      - getting, 42, 128
  - task priority, 11
- RTCS applications
  - DHCP Client, 51
  - DHCP Server, 52
  - DNS Resolver, 53
  - Echo Server, 54
  - EDS Server (Winsock), 55
  - FTP Client, 55
  - FTP Server, 55
  - SNMP Agent, 56
  - SNTP Client, 57
  - Telnet Client, 57
  - Telnet Server, 57
  - TFTP Client, 57
  - TFTP Server, 58
- rtcs.h, 216
- RTCS\_ERROR\_STRUCT, 252
- RTCS\_IF\_ENET, 154
- RTCS\_IF\_LOCALHOST, 154
- RTCS\_IF\_PPP, 154
- RTCS\_IF\_STRUCT, 253
- RTCS\_protocol\_table, 255
- RTCS\_protocol\_table, 255
- RTCS\_SO\_IGMP\_ADD\_MEMBERSHIP, 188
- RTCS\_SO\_IGMP\_DROP\_MEMBERSHIP, 189
- RTCS\_SO\_IGMP\_GET\_MEMBERSHIP, 189
- RTCS\_SO\_LINK\_RX\_8021Q\_PRIO, 191
- RTCS\_SO\_LINK\_RX\_8023, 192
- RTCS\_SO\_LINK\_TX\_8021Q\_PRIO, 194
- RTCS\_SO\_LINK\_TX\_8023, 194
- RTCS\_SO\_NAT\_PORTS, 191
- RTCS\_SO\_NAT\_TIMEOUTS, 190
- RTCS\_TASK structure, 256
- rtcsf.h, 19
- RTCSCFG\_CHECK\_ADDR\_SIZE, 20
- RTCSCFG\_CHECK\_ERRORS, 20
- RTCSCFG\_CHECK\_MEMORY\_ALLOCATION\_ERROR, 20
- RTCSCFG\_CHECK\_VALIDITY, 20
- RTCSCFG\_IP\_DISABLE\_DIRECTED\_BROADCAST, 21
- RTCSCFG\_LINKOPT\_8021Q\_PRIO, 21
- RTCSCFG\_LINKOPT\_8023, 21
- RTCSCFG\_LOG\_PCB, 21
- RTCSCFG\_LOG\_SOCKET\_API, 21

- rtcsinit.c, 255
- RTCSlite, 6
- running parameters (RTCS), 11, 23

## S

- secret structure (PPP\_SECRET), 251
- segment lifetime, max, 197, 199
- send buffer size socket option, 196
- send Ethernet 802.1Q priority tags socket option, 194
- send Ethernet 802.3 frames socket option, 194
- send nowait socket option, 29, 182, 194, 195
- send push socket option, 182, 195
- send timeout socket option, 195
- server information (DHCP), 175
- SMI, 7
- Smurf DoS attacks, reducing risk of, 21
- SNMP Agent, 56
  - starting, 201
- snmpcfg.h, 56
- SNMPCFG\_BUFFER\_SIZE, 56
- SNMPCFG\_COMMUNITY, 56
- SNMPCFG\_SYSDDESCR, 56
- SNMPCFG\_SYSSERVICES, 56
- SNMPv1, 8
  - community string, 56
- SNMPv1 MIB, 8
- SNMPv2, 8
- SNMPv2 MIB, 8
- SNMPv2c, 56
  - community string, 56
- SNMPv3, 8
- SNTP, 272
- SNTP Client
  - services, 57
  - starting, 202
  - starting time, 204
- SOCK\_DGRAM, 205
- SOCK\_STREAM, 205
- sockaddr\_in structure, 257
- socket error socket option, 196
- socket name (stream sockets), getting, 122
- socket options
  - getting, 124
  - round-trip time for stream sockets, 190
  - setting, 27, 186
    - for datagram sockets, 27
    - for stream sockets, 28
- socket type socket option, 196
- socket, send timeout, 195
- sockets
  - aborting stream sockets, 31
  - activity
    - waiting on any in a set, 178
    - waiting on any owned by task, 176

- binding, 27, 84
- comparison of datagram and stream, 24
- connections, establishing for stream sockets, 29
  - accept(), 80
  - connect(), 86
  - listen(), 129
- creating, 27, 205
- definition, 23
- endpoint identifier structure (sockaddr\_in), 257
- error, getting, 153
- example, 32
- function summary, 32
- names, getting for stream sockets, 29, 122
- ownership
  - relinquishing (RTCS\_detachsock()), 143
  - taking (RTCS\_attachsock()), 140
- peer names, getting for stream sockets, 120
- shutting down stream sockets, 31, 199
- using, summary, 25
- SOL\_IGMP, 186
- SOL\_LINK, 186
- SOL\_NAT, 186
- SOL\_SOCKET, 186
- SOL\_TCP, 186
- SOL\_UDP, 186
- S-Record boot files
  - loading, 173
  - loading and running, 14, 147
- stack size, additional
  - PPP Driver, 37
  - RTCS, 11
- Standard for the Transmission of IP Datagrams over Ethernet Networks, 7
- statistics
  - ARP
    - getting, 82
    - structure, 219
  - Ethernet, 12
    - getting, 106
    - structure, 224
  - ICMP
    - getting, 125
    - structure, 227
  - IGMP
    - getting, 126
    - structure, 230
  - IP
    - getting, 127
  - IPIF
    - getting, 42, 128
  - PPP
    - getting, 42, 128
  - PPP over Ethernet
    - getting, 73, 77, 79
  - TCP

- getting, 206
- UDP
  - getting, 212
- stream sockets
  - aborting, 31
  - about, 25
  - comparison with datagram sockets, 24
  - connections, establishing, 29
    - accept(), 80
    - connect(), 86
    - listen(), 129
  - data
    - buffering, 30
    - receiving, 30, 136
    - sending, 29, 182
    - throughput, improving, 30
  - names, getting, 29, 122
  - options
    - getting, 124
    - setting, 28, 186
  - peer names, getting, 120
  - round-trip time, 190
  - shutting down, 31, 199
  - using, 28
- Subnetting Procedure, 7
- system.sysDescr, 56
- system.sysService, 56

## T

- task priority
  - PPP Driver, 38
  - RTCS, 11
- TCP, 8, 268
  - checksums, bypassing, 11
  - in RTCS protocol table, 255
  - statistics
    - getting, 206
- TCP reset, 31
- Telnet
  - Client, 57
    - starting, 207
  - protocol, 8
  - Server, 57
    - starting, 111, 208
    - task structure (RTCS\_TASK), 256
- TFTP, 8
  - Client, 57
  - Server, 58
    - access to a TFTP client, allowing, 210
    - starting, 211
- tftp.h, 58, 211
- TFTPOP\_RRQ, 210
- TFTPOP\_WRQ, 210
- TFTPSRV\_MAX\_TRANSACTIONS, 58, 211

timeout, socket send, 195  
timewait timeout socket option, 197  
transmit buffer size socket option, 196  
transmit Ethernet 802.1Q priority tags socket option, 194  
transmit Ethernet 802.3 frames socket option, 194

## U

UDP, 8, 267  
    in RTCS protocol table, 255  
    statistics  
        getting, 212

UI command, 270  
unbinding IP addresses from device interfaces, 13, 170  
UNIX BSD 4.4, 23  
unnumbered information command, 270

## W

Winsock connection for embedded debugging, 55

## X

XDR, 8