Freescale MQX™ RTOS RTCS™ User's Guide

MQXRTCSUG
Rev. 16
02/2014

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: MQXRTCSUG
Rev. 16, 02/2014

Chapter 1
Before You Begin

Chapter 2
Setting Up the RTCS

Chapter 3
Using Sockets

Chapter 4
Point-to-Point Drivers

Chapter 5
RTCS Applications

Chapter 6
Rebuilding

Chapter 7
Function Reference

Chapter 8
Data Types

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, see freescale.com and navigate to Design Resources>Software and Tools>AllSoftware and Tools>Freescale MQX Software Solutions.

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 RTOS version 3.2.
Rev. 2	04/2009	Minor formatting updates for MQX RTOS version 3.2.1
Rev. 3	01/2010	Updated for MQX RTOS version 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 RTOS. 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.
Rev. 9	10/2012	“Configuration Options and Default Settings” chapter updated by new options. “setsockopt()” chapter updated by new RTCS_SO_IP_TX_TOS option.
Rev. 10	11/2012	Updated by IPv6-related description.
Rev. 11	03/2013	“Configuring TFTP Server”, “TFTPSRV_access()” and “Changing RTCS Creation Parameters” sections updated.
Rev. 12	05/2013	Added HTTP driver functions and SMTP client features. This version also includes grammatical and stylistic improvements.
Rev. 13	07/2013	Updated HTTP Server chapter; Updated HTTPSRV_PARAM_STRUCT and added HTTPSRV_ALIAS.
Rev. 14	10/2013	Replaced MQX types with C99 types.
Rev. 15	12/2013	Updates specific to 4.1.0-beta release.
Rev. 16	02/2014	Added getaddrinfo() and freeaddrinfo() to Function Reference chapter.

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 RTOS release provides information that was not available at the time this user's guide was published.
- The *MQX™ RTOS User's Guide* describes how to create embedded applications that use the MQX RTOS.
- The *MQX™ RTOS Reference Manual* 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 RTOS: MQX RTOS is used as the abbreviation for the MQX Real-Time Operating System.

1.3.2 Tips

Tips point out useful information.

TIP	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.

NOTE

Non-strict semaphores do not have priority inheritance.

1.3.4 Cautions

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

CAUTION	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 with sockets.

For information about	See
Data types mentioned in this chapter	Chapter 8, Data Types"
PPP Driver and PPP over Ethernet Driver	Chapter 4, Point-to-Point Drivers"
Protocols	Section Appendix A, ?$paratext>"
Prototypes for functions mentioned in this chapter	Chapter 7, Function Reference"
Sockets	Chapter 3, Using Sockets"

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, ?$paratext>."

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 see 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:

- 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).
- RTCS build process and compile-time configuration follow the same principles as other MQX core libraries (Chapter 6, Rebuilding").
- The RTCS Shell and all shell functions are removed from RTCS library and moved to a separate library in the Freescale MQX distribution.
- Freescale MQX RTOS contains 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 is added in the Freescale MQX release.

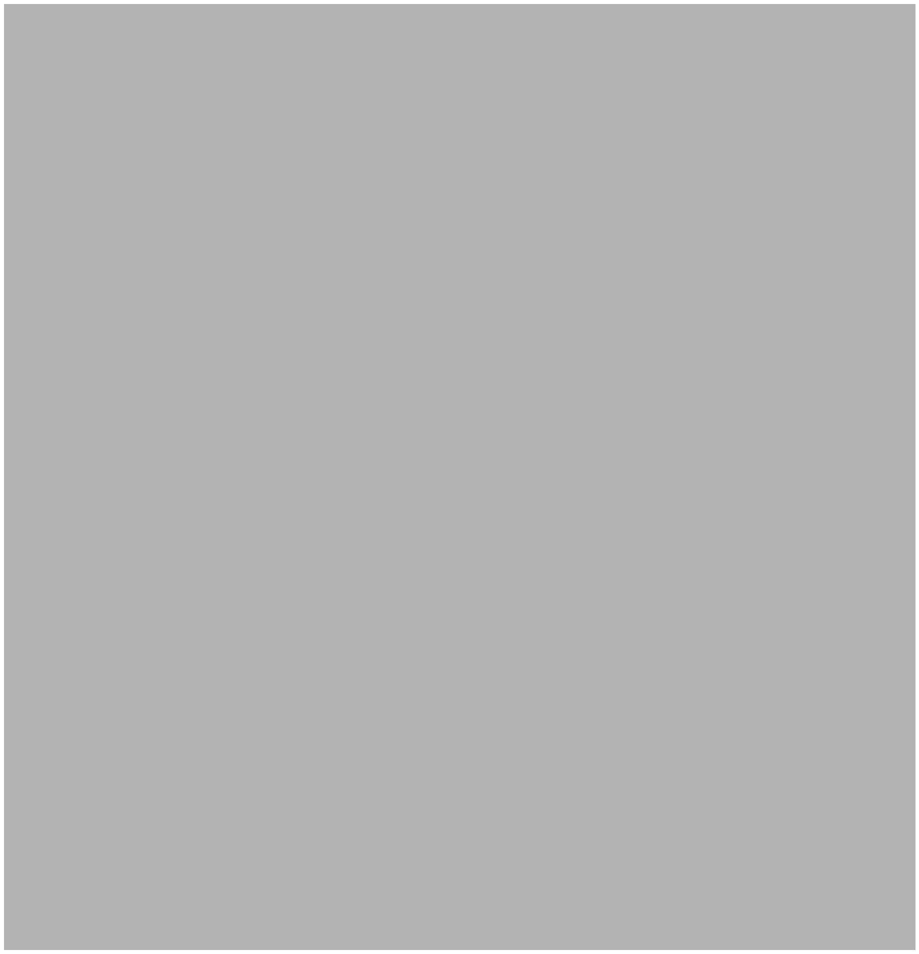

Figure 2-1. Protocols and Policies Discussed in This Manual

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 http://www.iana.org/numbers.	
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.

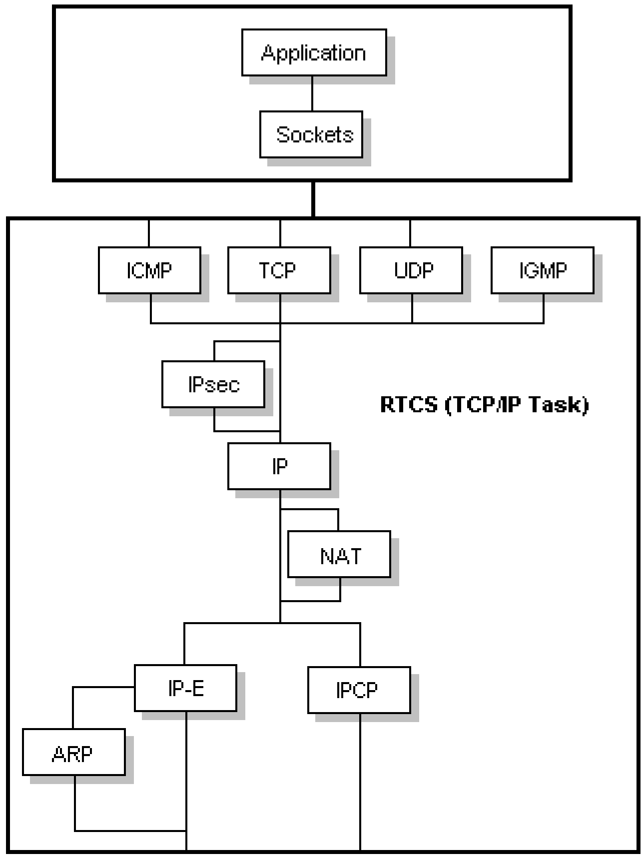

Figure 2-2. Protocol Stack Architecture

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.

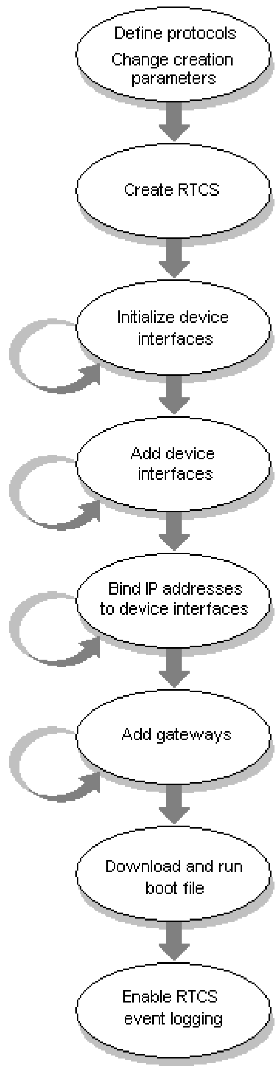

Figure 2-3. Steps to Set Up the RTCS

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. See Section 8.2.36, ?$paratext>" in Chapter 8, Data Types" for the list of available protocols. You can add or remove protocols using the instructions provided there, 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, . 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 (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 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.	4	_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, . 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 may 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.

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 any 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 that the interface can send and receive network traffic.
- Dynamically add to RTCS 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()** to it the device handle to the interface.

2.9.2 Initializing Interfaces to Point-to-Point Devices

Point-to-point 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.		**RTCS_if_bind()**
Bind an IP address that is obtained by using:		
	BootP	**RTCS_if_bind_BOOTP()**
	DHCP	**RTCS_if_bind_DHCP()**
	IPCP (the only method that can be used for PPP)	**RTCS_if_bind_IPCP()**

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, 0xFFFFFF00)`

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	**RTCS_exec_TFTP_BIN()**
Common Object File Format	**RTCS_exec_TFTP_COFF()**
Motorola S-Records	**RTCS_exec_TFTP_SREC()**

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 RTOS (RTCS library) compiled with RTCSCFG_LOGGING defined to 1 (for kernel log compilation parameters see *MQX™ RTOS User's Guide*).

In the application, an user must create the kernel log and enable RTCS logging (KLOG_RTCS_FUNCTIONS). A better description for kernel log can be found in the *MQX™ RTOS User's Guide*. Final step to enable RTCS event logging calling **RTCSLOG_enable()** with a 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. 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, a 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 the following:

- 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. 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. , 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
- Echo
- 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, ?$paratext>.”

2.15.5.1 Limitations

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

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

2.15.6 Example: Setting Up RTCS

Set up RTCS with one Ethernet device as follows:

```
_rtcs_if_handle  ihandle;
uint32_t          error;

/* For Ethernet driver: */
 _enet_handle  ehandle;

/* For PPP Driver: */
FILE_PTR          pfile;

/* 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);

/* 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\rtcscfg.h* file. This file is not intended to be modified by the user. proper include search paths set in the RTCS build project, the *rtcscfg.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 may 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	0	1	0	0
RTCSCFG_LOG_SOCKET_API	0	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, RTCS determines whether the *addrlen* field is at least *sizeof(sockaddr)* 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 RTCSCFG_CHECK_ERRORS

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

2.16.2.3 RTCSCFG_CHECK_MEMORY_ALLOCATION_ERROR

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

2.16.2.4 RTCSCFG_CHECK_VALIDITY

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

2.16.2.5 RTCSCFG_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 RTCSCFG_BOOTP_RETURN_YIADDR

When RTCSCFG_BOOTP_RETURN_YIADDR is 1, the BOOTP_DATA_STRUCT has an additional field which will be filled inwith the YIADDR field of the BOOTREPLY.

2.16.2.7 RTCSCFG_UDP_ENABLE_LBOUND_MULTICAST

When RTCSCFG_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 RTCSCFG_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 RTCSCFG_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 RTCSCFG_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 RTCSCFG_ENABLE_ICMP

Default value 1. Set to 0 to disable ICMP protocol.

2.16.2.12 RTCSCFG_ENABLE_IGMP

By default set to 0. Set to 1 to add support for IGMP protocol.

2.16.2.13 RTCSCFG_ENABLE_NAT

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

2.16.2.14 RTCSCFG_ENABLE_DNS

Default value is 0. Set to 1 to enable full DNS server and resolver.

2.16.2.15 RTCSCFG_ENABLE_LWDNS

Default 1. Set to 0 to disable lightweight name resolver.

2.16.2.16 RTCSCFG_ENABLE_IPIP

Default value is 0. Set to 1 to to add support for IPIP.

2.16.2.17 RTCSCFG_ENABLE_RIP

Default value is 0. Set to 1 to add support for RIP.

2.16.2.18 RTCSCFG_ENABLE_SNMP

Default value is 0. Set to 1 to add support for SNMP.

2.16.2.19 RTCSCFG_ENABLE_IP_REASSEMBLY

Default value is 0. Set to 1 to enable IP packet reassembling.

2.16.2.20 RTCSCFG_ENABLE_LOOPBACK

Default value is 0. Set to 1 to enable loopback interface.

2.16.2.21 RTCSCFG_ENABLE_UDP

Default value is 1. Set to 0 to disable support for UDP protocol.

2.16.2.22 RTCSCFG_ENABLE_TCP

Default value is 1. Set to 0 to disable support for TCP protocol.

2.16.2.23 RTCSCFG_ENABLE_STATS

Default value is 0. Set to 1 to add support for network trafic statistics.

2.16.2.24 RTCSCFG_ENABLE_GATEWAYS

Default value is 0. Set to 0 to disable support for gateways.

2.16.2.25 RTCSCFG_ENABLE_VIRTUAL_ROUTES

Default value is 0. Must be 1 for PPP or tunneling.

2.16.2.26 RTCSCFG_USE_KISS_RNG

Default 0. Must be 1 for PPP or tunneling.

2.16.2.27 RTCSCFG_ENABLE_ARP_STATS

Default value is 0. Set to 1 to enable ARP packet statistics.

2.16.2.28 RTCSCFG_PCBS_INIT

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

2.16.2.29 RTCSCFG_PCBS_GROW

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

2.16.2.30 RTCSCFG_PCBS_MAX

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

2.16.2.31 RTCSCFG_MSGPOOL_INIT

RTCS message pool initial size. Override in application by setting the _RTCS_msgpool_init variable.

2.16.2.32 RTCSCFG_MSGPOOL_GROW

RTCS message pool growing granularity. Override in application by setting the _RTCS_msgpool_grow variable.

2.16.2.33 RTCSCFG_MSGPOOL_MAX

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

2.16.2.34 RTCSCFG_SOCKET_PART_INIT

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

2.16.2.35 RTCSCFG_SOCKET_PART_GROW

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

2.16.2.36 RTCSCFG_SOCKET_PART_MAX

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

2.16.2.37 RTCSCFG_UDP_MAX_QUEUE_SIZE

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

2.16.2.38 RTCSCFG_ENABLE_UDP_STATS

Set to 0 for disable UDP statistics.

2.16.2.39 RTCSCFG_ENABLE_TCP_STATS

Set to 0 for disable TCP statistics.

2.16.2.40 RTCSCFG_TCP_MAX_CONNECTIONS

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

2.16.2.41 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.

2.16.2.42 RTCSCFG_ENABLE_RIP_STATS

Default value RTCSCFG_ENABLE_STATS, enable RIP statistics.

2.16.2.43 RTCSCFG_QUEUE_BASE

Override in application by setting _RTCSQUEUE_base.

2.16.2.44 RTCSCFG_STACK_SIZE

Override in application by setting _RTCSTASK_stacksize.

2.16.2.45 RTCSCFG_LOG_PCB

By default, RTCS doesn't log packet generation and parsing in the MQX kernel log. Set this value to 1 (one) to have RTCS log packets if application calls RTCSLOG_enable().

2.16.2.46 RTCSCFG_LOG_SOCKET_API

By default, RTCS doesn't log socket API calls in the MQX kernel log whether the application calls RTCSLOG_enable(). Set this value to 1 (one) to have RTCS log socket API calls.

2.16.2.47 RTCSCFG_ENABLE_IP4

Enable IPv4 Protocol support.

Default value 1.

2.16.2.48 RTCSCFG_ENABLE_IP6

Enable IPv6 Protocol support.

Default value 0.

2.16.2.49 RTCSCFG_ND6_NEIGHBOR_CACHE_SIZE

Maximum number of entries in the neighbor cache (per interface).

Default value 6.

2.16.2.50 RTCSCFG_ND6_PREFIX_LIST_SIZE

Maximum number of entries in the prefix list (per interface).

Default value 4.

2.16.2.51 RTCSCFG_ND6_ROUTER_LIST_SIZE

Maximum number of entries in the Default Router list (per interface).

Default value 2.

2.16.2.52 RTCSCFG_IP6_IF_ADDRESSES_MAX

Maximum number of IPv6 addresses per interface.

Default value 5.

2.16.2.53 RTCSCFG_IP6_IF_DNS_MAX

Maximum number of DNSv6 Server addresses that can be assigned to an interface.

Default value 2.

2.16.2.54 RTCSCFG_IP6_ REASSEMBLY

Enable IPv6 packet reassembling.

Default value 1.

2.16.2.55 RTCSCFG_IP6_ LOOPBACK_MULTICAST

Enable loopback of own IPv6 multicast packets.

Default value 0.

2.16.2.56 RTCSCFG_ND6_RDNSS

Enable Recursive DNS Server (RDNSS) Option support, according to RFC6106.

Default value 1.

2.16.2.57 RTCSCFG_ND6_RDNSS_LIST_SIZE

Maximum number of entries in the Recursive DNS Server (RDNSS) addresses list, per networking interface.

RFC6106 specifies a sufficient number of RDNSS addresses as three.

Default value 3.

2.16.2.58 RTCSCFG_ND6_DAD_TRANSMITS

Maximum number of Solicitation messages sent while performing Duplicate Address Detection on a tentative address.

Default value 1.

A value of one indicates a single transmission with no follow-up retransmissions. A value of zero indicates that Duplicate Address Detection is not performed on tentative addresses.

2.16.2.59 RTCSCFG_IP6_MULTICAST_MAX

Maximum number of unique IPv6 multicast memberships that may exist at the same time in the whole system.

Default value 10.

2.16.2.60 RTCSCFG_IP6_MULTICAST_SOCKET_MAX

Maximum number of IPv6 multicast memberships, that may exist at the same time per one socket.

Default value 1.

2.16.2.61 RTCSCFG_ENABLE_MLD

Enable Multicast Listener Discovery (MLDv1) Protocol support.

Default value 1.

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 ENET module hardware-acceleration options

ENET module implements layer 3 network acceleration functions. These functions are designed to accelerate the processing of various common networking protocols, such as IP, TCP, UDP and ICMP.

2.16.4.1 BSPCFG_ENET_HW_TX_IP_CHECKSUM

Set to 1 to enable generation of the IPv4 header checksum by the ENET module for outgoing packets. Set to 0 to disable it.

2.16.4.2 BSPCFG_ENET_HW_TX_PROTOCOL_CHECKSUM

Set to 1 to enable generation of the TCP, UDP, and ICMPv4 checksum by the ENET module for outgoing packets. Set to 0 to disable it.

2.16.4.3 BSPCFG_ENET_HW_RX_IP_CHECKSUM

Set to 1 to enable verification of the IPv4 header checksum by the ENET module for incoming packets. Set to 0 to disable it.

2.16.4.4 BSPCFG_ENET_HW_RX_PROTOCOL_CHECKSUM

Set to 1 to enable verification of the TCP, UDP and ICMPv4 checksum by the ENET module for incoming packets. Set to 0 to disable it.

2.16.4.5 BSPCFG_ENET_HW_RX_MAC_ERR

Set to 1 to enable discard of incoming frames with MAC layer (CRC, length or PHY) errors by the ENET module. Set to 0 to disable it.

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	Chapter 8, “Data Types”
MQX RTOS	*MQX RTOS User's Guide* *MQX RTOS Reference* Manual
Protocols	Section Appendix A, “Protocols and Policies”
Prototypes for functions mentioned in this chapter	Chapter 7, “Function Reference”
Setting up RTCS	Chapter 2, “Setting Up the RTCS”

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 the following:

- 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

Socket Type	Datagram socket	Stream socket
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 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.

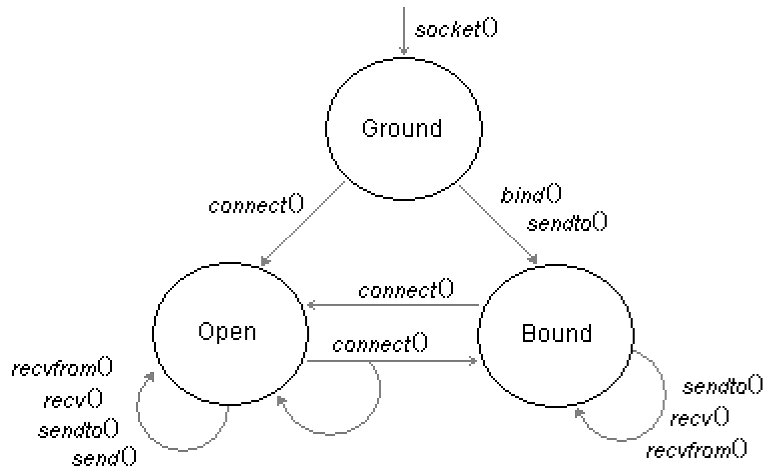

Figure 3-1. Creating and Using Datagram Sockets (UDP)

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

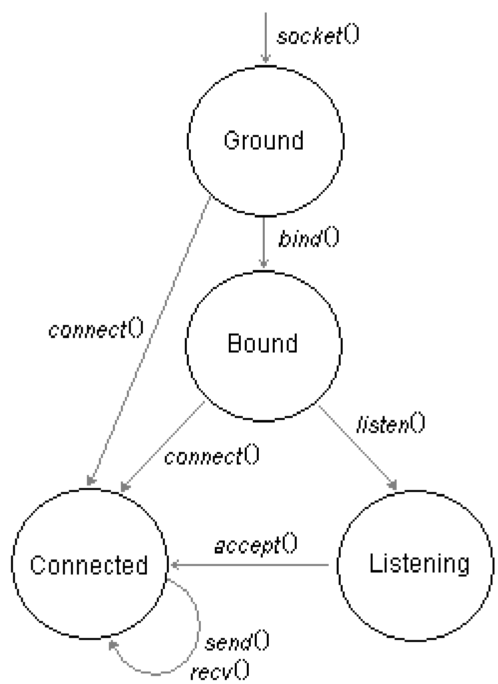

Figure 3-2. Creating and Using Stream Sockets (TCP)

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 at 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.

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. .

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 a **socket()** returned.

The new socket, which 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 it has 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 .
- 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

accept()	Accepts the next incoming stream connection and clones the socket to create a new socket, which services the connection.
bind()	Identifies the local application endpoint by providing a port number.
connect()	Establishes a stream connection with an application endpoint or sets a remote endpoint for a datagram socket.
getpeername()	Determines the peer address-port number endpoint of a connected socket.
getsockname()	Determines the local address-port number endpoint of a bound socket.
getsockopt()	Gets the value of a socket option.
listen()	Allows incoming stream connections to be received on the port number that is identified by a socket.
recv()	Receives data on a stream or datagram socket.
recvfrom()	Receives data on a datagram socket.
RTCS_attachsock()	Gets access to a socket that is owned by another task.
RTCS_detachsock()	Relinquishes ownership of a socket.
RTCS_geterror()	Gets the reason why an RTCS function returned an error for the socket.
RTCS_selectall()	Waits for activity on any socket that a caller owns.
RTCS_selectset()	Waits for activity on any socket in a set of sockets.
send()	Sends data on a stream socket or on a datagram socket, for which a remote endpoint has been specified.
sendto()	Sends data on a datagram socket.
setsockopt()	Sets the value of a socket option.
shutdown()	Shuts down a connection and discards the socket.
socket()	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;
uint32_t        sock, listensock;
int32_t         length;
uint32_t        index;
uint32_t        error;
uint16_t        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);
```

```
    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 PPP point-to-point driver.

For information about	See
Data types mentioned in this chapter	Chapter 8 "Data Types"
MQX RTOS	*MQX™ RTOS User's Guide* *MQX™ RTOS Reference Manual*
Protocols	Appendix A "Protocols and Policies"
Prototypes for functions mentioned in this chapter	Function Reference"
Setting up RTCS	Chapter 2 "Setting Up the RTCS"
Using RTCS and sockets	Chapter 3 "Using Sockets"

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	Configuring PPP Driver"
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)	Configuring PPP Driver”
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.

Since all processors do not 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. Therefore, 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 (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 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. Therefore, 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, if the most significant byte is zero, it is not sent.

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. In other words, before the first time that it calls **PPP_init()**.

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 ACCM).	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 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

Either:

— 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

Either:

— 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

Either:

- 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

Either:

- 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 *freescale* has the password *password1*.

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

```
char myname[]         = "freescale";
char mysecret[]       = "password1";
PPP_SECRET PAP_secret = {sizeof(myname)-1,
                         sizeof(mysecret)-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 the following:

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

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

```
char myname[]             = "freescale";
char server1[]            = "server1";
char mysecret1[]          = "password1";
char server2[]            = "server2";
char mysecret2[]          = "password2";
PPP_SECRET CHAP_secrets[] = {{sizeof(server1)-1,
                              sizeof(mysecret1)-1,
                              server1, mysecret1},
                             {sizeof(server2)-1,
                              sizeof(mysecret2)-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*. There are three users.

User	Password
fsl1	*password1*
fsl2	*password2*
fsl3	*password3*

4.2.3.6 PAP — Server Side

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

```
char user1[]         = "fsl1";
char secret1[]       = "password1";
char user2[]         = "fsl2";
char secret2[]       = "password2";
char user3[]         = "fsl3";
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[]         = "fsl1";
char secret1[]       = "password1";
char user2[]         = "fsl2";
char secret2[]       = "password2";
char user3[]         = "fsl3";
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_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_init(). 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.
- 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_init() 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

PPP_init()	Initializes PPP Driver (LCP or CCP) for a PPP link.
PPP_SECRET	Authentication passwords.
IPIF_stats()	Gets statistics about PPP links.

4.2.6 Example: Using PPP Driver

See Example: Setting Up RTCS.

PPP server and PPP client functionality is demonstrated in the RTCS shell example application. See *%MQX_ROOT%\rtcs\examples\shell and %MQX_ROOT%\shell\source\rtcs\sh_ppp.c*.

Chapter 5 RTCS Applications

5.1 Before You Begin

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

For information about	See
Data types mentioned in this chapter	Data Types
MQX RTOS	*MQX™ RTOS User's Guide* *MQX™ RTOS Reference* Manual
Protocols	Protocols and Policies
Prototypes for functions mentioned in this chapter	Function Reference”
Setting up the RTCS	Setting Up the RTCS
Using RTCS and sockets	Using Sockets

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 DHPC 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 making 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 RTCS_if_bind_DHCP() uses:	
DHCP_option_addr()	IP address
DHCP_option_addrlist()	List of IP addresses
DHCP_option_int8()	8-bit value

Table 5-1. Summary: Setting Up DHCP Client (continued)

DHCP_option_int16()	16-bit value
DHCP_option_int32()	32-bit value
DHCP_option_string()	String
DHCP_option_variable()	Variable-length option
RTCS_if_bind_DHCP()	Gets an IP address using DHCP and binds it to the device interface.
DHCPCLNT_find_option()	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 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 DHCPSVR_FLAG_DO_PROBE flag to **DHCPSRV_set_config_flag_off()**.

Table 5-2. Summary: Using DHCP Server

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

5.3.1 Example: Setting Up and Modifying DHCP Server

See **DHCPSRV_init()** in 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 *)DNS_Local_server_name, 0,
INADDR_LOOPBACK, 0,0,0,0, DNS_A, DNS_IN }};
```

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 *)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 *)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

DNS_SLIST_STRUCT	DNS server list *struct.*
DNS_init()	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 Changing Default Names.

If a query is successful, the DNS server sends a reply to DNS Resolver. . 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\if\dhshosts.c as global variable	Default value
Local domain	*DNS_Local_network_name*	"."
Local server	*DNS_Local_server_name*	""

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

gethostbyaddr()	Gets the HOSTENT_STRUCT for an IP address.
gethostbyname()	Gets the HOSTENT_STRUCT 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 the 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.

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 which is running a performance analysis tool available from Freescale MQX RTOS. 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 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 **FTP_open()**. 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

File Transfer protocol (FTP) is network protocol that allows users to transfer files between hosts over TCP connections. It receives commands on command port and transfers data on either active or passive data connection. Basic user authentication is supported in form of username and password. It is also possible to specify separate root directory for each user.

5.8.1 Communicating with an FTP Client

Following commands are supported in FTPSRV:

- ABOR - abort current file transfer.
- APPE [filename]- append data to file [filename].
- CWD [path] - change working directory to [path].
- CDUP - change working directory one level up.
- DELE [filename]- delete file [filename].
- EPSV - extended passive mode (IPv6).
- EPRT - extended port command (IPv6).
- FEAT - list server features.
- HELP - show server help (command list).
- LIST [dirname] - list files in directory [dirname].
- MKDIR [dirname] - create directory [dirname].
- MKD - same as MKDIR.
- NLST [dirname] - list filenames in directory [dirname].
- NOOP - no operation (empty command).
- PASS [password] - input password.
- PASV - passive transfer mode.
- PORT [host-port] - set host and port for data transfer.
- PWD - print working directory.
- QUIT - disconnect from server.
- RMDIR [dirname] - remove directory [dirname].
- RMD - same as RMDIR.
- RETR [filename] - retrieve file [filename] from server.
- RNFR [filename] - rename from [filename].
- RNTO [filename] - rename to [filename].

- SITE - site specific information.
- SIZE [filename] - get [filename] size.
- STOR [filename] - store file [filename] to server.
- SYST - get system name.
- TYPE [type] - set type of transferred data to [type].
- USER [username] - login user [username].
- XCUP - same as CDUP.
- XCWD - same as CWD.
- XMKD - same as MKDIR.
- XPWD - same as PWD.
- XRMD - same as RMDIR.

5.8.2 Compile Time Configuration

A few macros are used for setting FTP server default configuration during compile time. Default values of all of them can be found in file %MQX_PATH%\rtcs\source\include\rtcscfg.h. If you need to change any option, add required define directive to file user_config.h of your project.

- FTPSRVCFG_DEF_SERVER_PRIO: default priority of server tasks. This value is used when the HTTP server creates its main and session task. The value can be overridden by setting server_prio member of the server initialization structure to a non-zero value. By default, value of this macro is set to 8.
- FTPSRVCFG_DEF_ADDR: default server IPv4/IPv6 address. The server is listening on this address if a different value is not set by ipv4_addres or ipv6_address member (depending on selected address family) in the server initialization structure. Default value of this macro is INADDR_ANY.
- FTPSRVCFG_DEF_SES_CNT: default maximum number of sessions. This value limits maximum number of sessions (connections) to the server. Each time a new connection is established from the client, a new session is created. The value of this parameter can be overridden by setting the max_ses member of the server initialization structure. Default value is 2 sessions.
- FTPSRVCFG_TX_BUFFER_SIZE: size of the socket transmit buffer in bytes. This option cannot be overridden in runtime. Default value is 4380 bytes.
- FTPSRVCFG_RX_BUFFER_SIZE: size of the socket receive buffer in bytes. This option cannot be overridden in runtime. Default value is 1460 bytes.
- FTPSRVCFG_TIMEWAIT_TIMEOUT: timeout value for send/receive operations on the sockets in miliseconds. This option cannot be overridden in runtime. Default value is 1000 ms.
- FTPSRVCFG_SEND_TIMEOUT: timeout value for server sockets in miliseconds. This option cannot be changed during runtime. Default value is 500ms.
- FTPSRVCFG_CONNECT_TIMEOUT: hard timeout for connection establishment in miliseconds for HTTP server sockets. Cannot be changed during runtime. Default value is 5000ms.
- FTPSRVCFG_RECEIVE_TIMEOUT: timeout for the recv() function. After this timeout recv() returns with whatever data it has. Cannot be changed during runtime. Default value is 50ms.

- FTPSRVCFG_IS_ANONYMOUS: macro defining if login/password are required to run priviledged server commands. If it is set to zero (default), the login and password are required; otherwise no authentication is needed.

5.8.3 Basic Usage

There are only two steps you must follow to successfully start the FTP server:

1. Create and fill structure of type FTPSRV_PARAM_STRUCT with required server settings. All parameters, but root directory are optional. You can set any parameter to zero/NULL and the server will use a default value.
2. Start the server using function FTPSRV_init() with a parameter created in previous step. Both of these steps are demonstrated by an example which you can find in %MQX_PATH%\shell\source\rtcs\sh_ftpd.c. The server parameters structure description can be found in Chapter "FTPSRV_PARAM_STRUCT."

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. The MQX HTTP support HTTP protocol in version 1.0 defined by RFC 1945 (http://tools.ietf.org/html/rfc1945).

Server creates a separate task and an internal data structure for every incoming connection from the client (this is called session in further text). When the session processing is done (a response is send to the client) and keep-alive option is disabled, the connection from the client is closed and the session is destroyed. In case keep-alive is enabled the connection remains open and the server waits for another request from the client. This can speed up transfers of multiple small files because the connection does not need to be reestablished.

Following features are supported:

- GET and POST requests.
- CGI scripts (http://tools.ietf.org/html/rfc3875).
- ASP-like Server Side Includes (commands with parameters enclosed by ‘<%’ and ‘%>’).
- Basic authentication.
- HTTP keep-alive.
- Percent encoded URI.
- Cache control
- Aliases

5.9.1 Cache control

Server implements a simple HTTP cache control directives. This means that static files are cached in a web browser and need not to be updated when the webpage is reloaded. List of cached extensions (directive *Cache-Control: max-age=3600*) is as follows:

- js
- css
- gif
- htm
- jpg
- png
- html

Files protected by an authentication are not cached (Cache-Control: no-store directive is used). Time for which the file is stored in a cache is determined by the value of the HTTPSRVCFG_CACHE_MAXAGE macro. The default is 3600ms. See RFC2616 section 14.9 for more details about the cache control mechanism.

5.9.2 Supported MIME types

Following MIME types are supported:

- text/plain
- text/html
- text/css
- image/gif
- image/jpeg
- image/png
- application/javascript
- application/zip
- application/pdf
- application/octet-stream

Type application/octet-stream is default when no other MIME type is applicable.

5.9.3 Aliases

An alias mechanism enables you to access filesystems and folders which are not subfolders of the server root directory. Each aliased directory has a user defined name under which it can be accessed by client. The following example demonstrates how to access files from USB mass storage mounted as c: drive in the MQX RTOS. The selected name is „usb“ and all files are available on the link: http://SERVER_IP_ADDRESS/usb/.

Example code:

```
HTTPSRV_ALIAS http_aliases[] = {
    {"/usb/", "c:\\"},
    {NULL, NULL}
    };

//Initialization code for RTCS

HTTPSRV_PARAM_STRUCT params;
_mem_zero(&params, sizeof(HTTPSRV_PARAM_STRUCT));

params.root_dir = "tfs:";
params.alias_tbl = (HTTPSRV_ALIAS*)http_aliases;

server = HTTPSRV_init(&params);
if(!server)
{
printf("Error: HTTP server init error.\n");
}
```

5.9.4 Compile time configuration

A few macros are used for setting HTTP server default configuration during compile time. Default values of all of them can be found in file *%MQX_PATH%\rtcs\source\include\rtcscfg.h*. If you need to change any option, add required define directive to file *user_config.h* of your project.

- HTTPSRVCFG_DEF_SERVER_PRIO – default priority of server tasks. This value is used when the HTTP server creates its main, session and script handler task. The value can be overridden by setting ‘server_prio’ member of the server initialization structure to a non-zero value. By default, value of this macro is set to 8.
- HTTPSRVCFG_DEF_ADDR – default server IPv4/IPv6 address. The server is listening on this address if different value is not set by ‘ipv4_addres’ or ‘ipv6_address’ member (depending on selected address family) in the server initialization structure. Default value of this macro is INADDR_ANY.
- HTTPSRVCFG_DEF_PORT – default port to listen on; can be overridden by setting a non-zero value of the ‘port’ member in the server initialization structure. Default value of this macro is 80.
- HTTPSRVCFG_DEF_INDEX_PAGE – default index page. This macro specifies a name of a webpage to be send as the response when the client requests the root directory (‘/’). It can be overridden by setting the ‘index_page’ member of the server initialization structure. Default index page is "index.htm".
- HTTPSRVCFG_DEF_SES_CNT – default maximum number of sessions. This value limits maximum number of sessions (connections) created by the server. Each time a new connection is established from the client a new session is created. Value of this parameter can be overridden by setting the ‘max_ses’ member of the server initialization structure. Default value is 2 sessions.
- HTTPSRVCFG_SES_BUFFER_SIZE – default size of session buffer in bytes. This buffer is used to store all data required by the session. This setting cannot be overridden in runtime. Default value of this macro is set to 1360 bytes and is limited to 512 bytes as minimum.
- HTTPSRVCFG_DEF_URL_LEN – default maximal length of the URL in characters. Value of this parameter can be set up using the ‘max_uri’ member of the server initialization structure. When

the URL exceeds this length, a response with a code 414 (Request-URI Too Long) is send to the client. Default value of this macro is 128 characters.

- HTTPSRVCFG_MAX_SCRIPT_LN – maximal length of script (CGI and SSI) name in characters. All scripts with a name longer then this value are ignored. Default value of this macro is 16.
- HTTPSRVCFG_KEEPALIVE_ENABLED – macro determining if HTTP keep-alive is enabled or disabled. Default value of this macro is 0 (disabled). This option cannot be changed during runtime.
- HTTPSRVCFG_KEEPALIVE_TO – session timeout when using keep-alive. This value determines time in milliseconds for which the server will wait for a next request after the previous request was successfully processed. This value cannot be overridden during runtime. Default value of this macro is 200 ms.
- HTTPSRVCFG_SES_TO – session timeout in milliseconds. This value determines maximum time for which the session can be inactive until it is aborted. This option cannot be changed in runtime. Default value is 20000 ms (20 s).
- HTTPSRVCFG_TX_BUFFER_SIZE – size of the socket transmit buffer in bytes. This option cannot be overridden in runtime. Default value is 4380 bytes.
- HTTPSRVCFG_RX_BUFFER_SIZE – size of the socket receive buffer in bytes. This option cannot be overridden in runtime. Default value is 1460 bytes.
- HTTPSRVCFG_TIMEWAIT_TIMEOUT – timeout value for send/receive operations on the sockets in miliseconds. This option cannot be overridden in runtime. Default value is 1000 ms.
- HTTPSRVCFG_RECEIVE_TIMEOUT - timeout for the recv() function. After this timeout recv() returns with whatever data it has. Cannot be changed during runtime. Default value is 50ms.
- HTTPSRVCFG_CONNECT_TIMEOUT - hard timeout for connection establishment in miliseconds for HTTP server sockets. Cannot be changed during runtime. Default value is 5000ms.
- HTTPSRVCFG_SEND_TIMEOUT - timeout value for server sockets in miliseconds. This option cannot be changed during runtime. Default value is 500ms.

5.9.5 Basic Usage

There are basically only two steps you must follow to successfully start the HTTP server:

1. Create and fill structure of type HTTPSRV_PARAM_STRUCT with required server settings. All parameters are optional. You can set any parameter to zero/NULL and the server will use a default value.
2. Start the server using function HTTPSRV_init() with a parameter created in previous step.

Both of these steps are demonstrated by an example which you can find in the *%MQX_PATH%\rtcs\examples\httpsrv* folder. The server parameters structure description can be found in Chapter HTTPSRV_PARAM_STRUCT.”

5.9.6 Using CGI Callbacks

If you want to use a CGI in your application you have to create a function for each “script”. This function is then called every time the client requests a CGI file with same name as the function label. Pointers to all these functions must be saved in array of type HTTPSRV_CGI_LINK_STRUCT and this structure must be passed to server in pointer cgi_lnk_tbl as part of the server parameters structure.

There are two ways in which either SSI or CGI can be processed:

- One task: Scripts are processed one after another in one task.
- Multiple tasks: Each script is processed in separate task.

Processing in single task (serial processing):

There is one task created to handle all user scripts on server startup. This task have a stack size determined by the _script_stack_ variable in the server parameters structure. When a script is to be executed, a message is sent from a session to this task and it will run the script. The session is blocked until the script finishes. This approach is used when the size of a stack for the script is set to zero in either HTTPSRV_SSI_LINK_STRUCT or HTTPSRV_CGI_LINK_STRUCT.

Processing in multiple tasks (parallel processing):

As in previous case, there is a task created on server startup to handle scripts, but this task have stack of minimal size. When the script is encountered during the session processing, a message is sent to this task, but instead of running a user callback, a new detached task is created with stack size set to value from the CGI/SSI link structure. Finally in this new task, the user callback is run. This allows the script handling task to immediately read another message without waiting.

Thanks to parallel processing, some more complicated applications can be easily implemented (for example, uploading big files through CGI). This approach is used when the size of stack for script is set to value other than zero in the script table.

You can also combine both methods. Callbacks with stack size set to zero are processed in script handler task with stack size set by _script_stack_variable. If there is some callback with non-zero stack in script table, it will be processed in the separate task.

Example:

```
/* First we create table of CGI callbacks */
static _mqx_int cgi_ipstat(HTTPSRV_CGI_REQ_STRUCT* param);
static _mqx_int cgi_icmpstat(HTTPSRV_CGI_REQ_STRUCT* param);
static _mqx_int cgi_udpstat(HTTPSRV_CGI_REQ_STRUCT* param);
static _mqx_int cgi_tcpstat(HTTPSRV_CGI_REQ_STRUCT* param);
static _mqx_int cgi_rtc_data(HTTPSRV_CGI_REQ_STRUCT* param);


const HTTPSRV_CGI_LINK_STRUCT cgi_lnk_tbl[] = {
    { "ipstat",         cgi_ipstat,   1500},
    { "icmpstat",       cgi_icmpstat, 1500},
    { "udpstat",        cgi_udpstat,  1500},
    { "tcpstat",        cgi_tcpstat,  1500},
    { "rtcdata",        cgi_rtc_data, 0},
    { 0, 0 }    // DO NOT REMOVE - last item - end of table
};


/* Then table is saved in parameters structure and server is started */
HTTPSRV_PARAM_STRUCT params;
```

```
uint32_t server;

_mem_zero(&params, sizeof(params));

/* Every time client request i.e. file rtcdata.cgi function cgi_rtc_data is called.*/
params.cgi_lnk_tbl = (HTTPSRV_CGI_LINK_STRUCT*) cgi_lnk_tbl;
server = HTTPSRV_init(&params);
```

In the user CGI function following must be done:

1. Check the method of request (GET or POST).
2. Create a variable of type HTTPSRV_CGI_RES_STRUCT (called “response” in further text).
3. Read the data from the client using **httpsrv_cgi_read()** function. All the data must be read before sending response back to the client.
4. Fill in variables in the response structure (this is needed so you can send the data to the client). All members are mandatory.
5. Write the data using the function **httpsrv_cgi_write()**.
6. Return `content_length` of response.

After the first call of the function **httpsrv_cgi_write()** the HTTP header is formed automatically by the HTTP server. If you want to send more data, set the response.data variable to address of data you want to send and store length of data in bytes to the response.data_length variable. Then whenever you call **httpsrv_cgi_write()** data are stored in the session buffer and then send to the client.

Basic information about client and connection can be read from parameter of type HTTPSRV_CGI_REQ_STRUCT passed to every CGI callback. For detailed information about this structure see chapter HTTPSRV_CGI_REQ_STRUCT”.

5.9.7 Using server side include (SSI) callbacks

Server side includes are functions called every time special sequence of characters is encountered during parsing of files with “.shtm” or “.shtml” extension. This special sequence consists of an entry tag, function name (optionally with parameter) and an exit tag:

```
<%function_name:parameter%>
```

Similarly to CGI, functions for each SSI must be declared and pointers to these functions together with their names/labels must be stored in array of HTTPSRV_SSI_LINK_STRUCT types. This array is then passed to server as ssi_lnk_tbl variable within the parameters structure. Example:

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

uint32_t server;
HTTPSRV_PARAM_STRUCT params;

_mem_zero(&params, sizeof(params));
params.ssi_lnk_tbl = (HTTPSRV_SSI_LINK_STRUCT*)fn_lnk_tbl;
server = HTTPSRV_init(&params);
```

Whenever you wish to write something from server side include to the response send to the client, use the **httpsrv_ssi_write()** function.

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 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 on its own, running a task independently, 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:

7. Create RTCS as described in previous sections (**RTCS_create()**)
8. Initialize network device using **ipcfg_init_device()**.
9. Use one of the **ipcfg_bind_***xxx* functions to bind the interface to an IP address, mask and gateway. IPv6 address will be assigned automatically using the IPv6 stateless auto configuration. To add IPv6 address manually use **ipcfg6_bind_addr()** (see example in *shell/source/rtcs/sh_ipconfig.c: Shell_ipconfig_staticip()*).
10. 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.
11. 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 definitions:

- RTCSCFG_ENABLE_GATEWAYS - must be set to 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 to 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 to non-zero to enable DHCP binding in IPCFG. Note that DHCP also depends on RTCSCFG_ENABLE_UDP.
- RTCSCFG_IPCFG_ENABLE_BOOT - must be set to 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 some of them depending on the hardware support, and the range of values may change. Please see 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 commnads:
   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 SMTP client

Simple Mail Transfer Protocol is an internet standard designed for electronic mail transmission across IP networks. The RTCS SMTP client is based on RFC 5321. MQX implementation supports both IPv4 and IPv6 protocol.

5.12.1 Sending an email

To send an email only one function must be called - SMTP_send_email. Before calling, a structure of data type SMTP_PARAM_STRUCT must be set up and passed to the function as first parameter. Also if a detailed error/delivery message is required, the user must create a buffer for such message and pass it and its size as a second respectively third parameter to the function. For further reference of SMTP client functionality please see reference of following functions and data types:

- **SMTP_send_email()**
- **SMTP_EMAIL_ENVELOPE**
- **SMTP_PARAM_STRUCT**

5.12.2 Example application

There is example demonstrating functionality of SMTP client in RTCS. You can find this sample code in file *%MQX_PATH%\shell\source\rtcs\sh_smtp.c*. This file contains code that implements an *email* shell command that can be used for sending email with authentication from RTCS shell.

5.13 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

5.13.1 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.	**SNMPCFG_COMMUNITY_LIST**	"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).	**SNMPCFG_BUFFER_SIZE**	512
Value of the variable *system.sysDescr.*	**SNMPCFG_SYSDESCR**	"RTCS version 3.0"
Value of the variable *system.sysServices.*	**SNMPCFG_SYSSERVICES**	8

5.13.2 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 function) must be called before **SNMP_init()**.
- **SNMP_init()** along 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 recepients.

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 Changing RTCS Creation Parameters.

5.13.3 Communicating with SNMP Clients

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

5.13.4 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 defintion 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.

5.13.4.1 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 the type of 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 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, 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 reffer the 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.

5.13.4.2 MIB Definition File: Verbatim C Code

The C code, generated by the script, references other variables and functions that must be provided by the user. This kind of 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 which 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 RTCSMIB_mib_add(&MIBNODE_*objectname*) registers the object with the SNMP agent.
No-leaf object node.	No user code required. The generated RTCSMIB_NODE structure only contains pointers to other node structures.
Leaf object node (variable object).	The instance of RTCSMIB_VALUE structure named as MIBVALUE_*objectname*.
Table object	A function to map table indices to instances. The function name should be MIB_find_*objectname*(),
Writable variable object	A function to perform a set operation. The function name should be MIB_set_*objectname*(),

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 follows:

```
typedef struct rtcsmib_value
{
         uint32_t TYPE;
         void *PARAM;
} RTCSMIB_VALUE, * 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 conjuction 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	int32_t	Constant signed integer supplied directly as the PARAM value.
	RTCSMIB_NODETYPE_ INT_PTR	int32_t *	Pointer to signed integer value.
	RTCSMIB_NODETYPE_ INT_FN	RTCSMIB_INT_FN_PTR function pointer: int32_t function(void*)	Pointer to function that takes an instance pointer (void *), returning the signed int32_t value.
	RTCSMIB_NODETYPE_ UINT_CONST	uint32_t	Constant unsigned integer supplied directly as the PARAM value.
	RTCSMIB_NODETYPE_ UINT_PTR	uint32_t *	Pointer to unsigned integer value.
	RTCSMIB_NODETYPE_ UINT_FN	RTCSMIB_UINT_FN_PTR function pointer uint32_t function(void*)	Pointer to function that takes an instance pointer (void *), returning the unsigned uint32_t value.
NULL-terminated OCTET STRING, whose value SNMP agent computes when SNMP manager performs GET	RTCSMIB_NODETYPE_ DISPSTR_FN	unsigned char*	PARAM points to C string directly.
	RTCSMIB_NODETYPE_ DISPSTR_FN	RTCSMIB_UINT_FN_PTR function pointer unsigned char* function(void*)	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 unsigned char* function(void*, uint32_t*);	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(void*)	Pointer to function that takes an instance pointer (void *), returning address of an initialized RTCSMIB_NODE structure.

Table-Row Objects

For each variable object which 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.

```
bool MIB_find_objectname
(
        uint32_t op, /* IN */
        void *index, /* IN */
        void * *instance /* OUT */
)
```

Writable Objects

For each variable object which 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.

```
uint32_t MIB_set_objectname
(
        void *instance, /* IN */
        unsigned char *value_ptr, /* OUT */
        uint32_t 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.

If 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.

5.13.5 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
```

5.13.6 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*).

5.14 SNTP (Simple Network Time Protocol) 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

SNTP_init()	Starts the SNTP Client task.
SNTP_oneshot()	Sets the time using the SNTP protocol.

5.15 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()**.

5.16 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.

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 Changing RTCS Creation Parameters.

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.

5.17 TFTP Client

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

TFTP Client sends a request message to port 69.

5.18 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.

5.18.1 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 both read and write access. You can change its behavior to suit your needs.

5.18.2 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 Changing RTCS Creation Parameters.

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 an illustration for general purposes only 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.

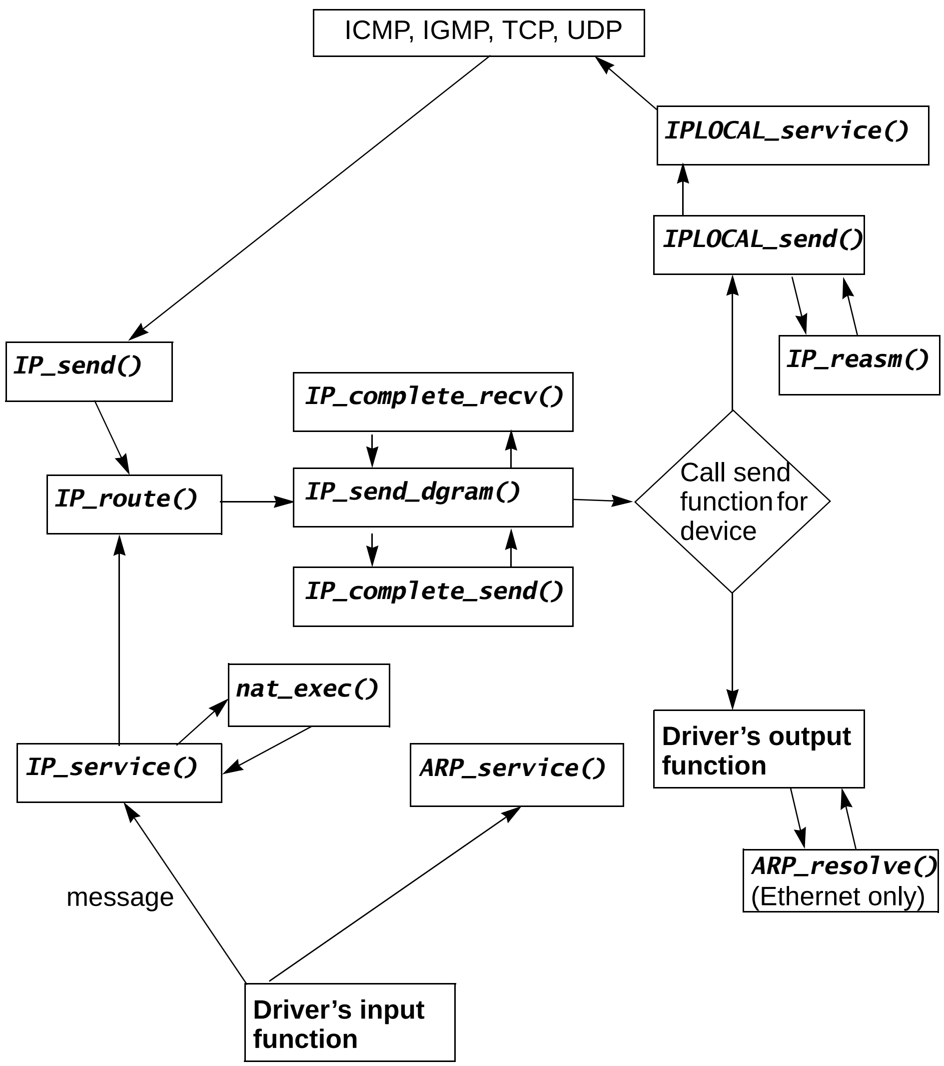

Figure 5-1. Typical RTCS Packet-Processing Paths

Chapter 6 Rebuilding

6.1 Reasons 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.

CAUTION	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:

- See the *MQX™ RTOS User's Guide*, a document for MQX RTOS rebuild instructions. A very similar concept applies also to the RTCS.
- See the *MQX™ RTOS Release Notes* that accompany Freescale MQX RTOS 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.

config		The main configuration directory.
	<board>	Board-specific directory, which contains the main configuration file (*user_config.h*).
rtcs		Root directory for RTCS within the Freescale MQX distribution.

	\build		
		\codewarror	CodeWarrior-specific build files (project files).
	\examples		
		\example	Source files (*.c*) for the example and the example's build project.
	\source		All RTCS source code files.
\lib			
	\<board>.<comp>\rtcs		RTCS library files built for your hardware and environment.

6.4 RTCS Build Projects in Freescale MQX RTOS

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 or 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 for the board-independent code being built into the board-specific output directory, is so that 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™ RTOS User's 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 in 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 which is 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.

6.4.2 Build Targets

CodeWarrior development environment allows for 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 opening the proper build project in the development environment and building it. Don't forget to select the proper build target or to 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 accept()

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

Synopsis

```
uint32_t accept(
        uint32_t              socket,
        sockaddr     *      peeraddr,
        uint16_t     *      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

a) Socket accepts IPv4 connection.

```
uint32_t     handle;
uint32_t     child_handle;
sockaddr    remote_sin;
uint16_t     remote_addrlen;
uint32_t     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);
      }
   }
}
```

b) Socket accepts IPv6 connection on port 7007.

```
uint32_t     sock, sock6;
sockaddr_in6 laddr6, raddr6;
uint16_t rlen;

memset(&laddr6,0x0,sizeof(laddr6));
laddr6.sin6_port = 7007;
laddr6.sin6_family = AF_INET6;
laddr6.sin6_addr = in6addr_any;
laddr6.sin6_scope_id = 0;

sock6 = socket(AF_INET6, SOCK_STREAM, 0);
if(RTCS_SOCKET_ERROR == sock6)
{
  printf("Error, socket() failed\n");
  _task_block();
}

error = bind(sock6, &laddr6, sizeof(laddr6));
if(RTCS_OK != error)
{
  printf("bind() failed, error 0x%lx\n", error);
  _task_block();
}

error = listen(sock6, 0);
if(RTCS_OK != error)
{
  printf("listen() failed - 0x%lx\n", error);
  _task_block();
}

sock = RTCS_selectset(&sock6,1,0);
if(RTCS_SOCKET_ERROR == sock)
{
  printf("selectset() failed - 0x%lx\n", RTCS_geterror(sock6));
  _task_block();
```

```
}

if(sock == sock6)
{
  rlen = sizeof(raddr6);
  sock = accept(sock6, &raddr6, &rlen);
  if(RTCS_SOCKET_ERROR == sock)
  {
    printf("accept() failed - 0x%lx\n", RTCS_geterror(sock6));
    _task_block();
  }
}
```

7.1.3 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**
- **inet_pton()**
- **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);
  }
```

7.1.4 bind()

Binds the local address to the socket.

Synopsis

```
uint32_t  bind(
          uint32_t              socket,
          sockaddr     *        localaddr,
          uint16_t              addrlen)
```

Parameters

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

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

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

Description

The following *localaddr* input values are required:

sockaddr field	Required input value
sin_family	AF_INET
sin_port	One of: • Local port number for the socket. • Zero (to determine the port number that RTCS chooses, call **getsockname()**).
sin_addr	One of: • IP address that was previously bound with a call to one of the **RTCS_if_bind** functions. • *INADDR_ANY.*

sockaddr field	Required input value
sin6_family	AF_INET6
sin6_port	One of: • Local port number for the socket. • Zero (to determine the port number that RTCS chooses, call **getsockname()**).
sin6_addr	IPv6 address.
sin6_scope_id	Scope zone index.

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*
- *sockaddr*

Examples

a) Binds a socket to port number 2010.

```
uint32_t      sock;
sockaddr_in                 local_sin;
uint32_t      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, (struct sockaddr *)&local_sin, sizeof (sockaddr_in));
if (status != RTCS_OK)
  printf("\nError, bind() failed with error code %lx", result);
```

b) Binds a socket to port number 7007 using IPv6 protocol.

```
uint32_t     sock, sock6;
sockaddr_in6 laddr6, raddr6;
uint16_t rlen;

memset(&laddr6,0x0,sizeof(laddr6));
laddr6.sin6_port = 7007;
laddr6.sin6_family = AF_INET6;
laddr6.sin6_addr = in6addr_any;
laddr6.sin6_scope_id = 0;

sock6 = socket(AF_INET6, SOCK_STREAM, 0);
if(RTCS_SOCKET_ERROR == sock6)
{
  printf("Error, socket() failed\n");
  _task_block();
}
```

```
error = bind(sock6, &laddr6, sizeof(laddr6));
if(RTCS_OK != error)
{
  printf("bind() failed, error 0x%lx\n", error);
  _task_block();
}
```

7.1.5 connect()

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

Synopsis

```
uint32_t connect(
          uint32_t              socket,
          sockaddr      *     destaddr,
          uint16_t              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.

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()**

Examples: Stream Socket

a) The connection use IPv4 protocol.

```
uint32_t            sock;
uint32_t            child_handle;
sockaddr_in        remote_sin;
uint16_t            remote_addrlen = sizeof(sockaddr_in);
uint32_t            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, (struct sockaddr *)&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);
}
```

b) The connection use IPv6 protocol.

```
struct addrinfo     hints;          /* Used for getaddrinfo()*/
struct addrinfo     *addrinfo_res;  /* Used for getaddrinfo()*/
uint32_t sock;
uint32_t error;

/* Extract IP address and detect family, here we will get scope_id too. */
memset(&hints,0,sizeof(hints));
hints.ai_family     = AF_UNSPEC; /* Allow IPv4 or IPv6 */
hints.ai_socktype   = SOCK_STREAM;
if (getaddrinfo("fe80::e5ec:43fc:4aca:bf13","7007", &hints, &addrinfo_res) != 0)
{
  printf("GETADDRINFO error\n");
  /* We can return right here and do not need free freeaddrinfo(addrinfo_res)*/
  return SHELL_EXIT_ERROR;
}

sock = socket(addrinfo_res->ai_family, SOCK_STREAM, 0);
if(RTCS_SOCKET_ERROR == sock)
{
  printf("Socket create failed\n");
  freeaddrinfo(addrinfo_res);
  return;
}

error = connect(sock, addrinfo_res->ai_addr, addrinfo_res->ai_addrlen);
if(RTCS_OK != error)
{
  printf("Connect failed, return code 0x%lx\n", error);
  freeaddrinfo(addrinfo_res);
```

```
    return;
}

freeaddrinfo(addrinfo_res);
```

7.1.6 DHCP_find_option()

Searches a DHCP message for a specific option type.

Synopsis

```
unsigned char     *DHCP_find_option(
  unsigned char    *msgptr,
  uint32_t         msglen,
  uchar             option)
```

Parameters

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

msglen [in] — Number of bytes in the message.

option [in] — 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 * buffer_ptr; /* This is a DHCP packet recieved
                       from a server */
uint32_t buffer_size;
uchar * optptr;

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

7.1.7 DHCP_option_addr()

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

Synopsis

```
bool  DHCP_option_addr(
         unsigned char  *   *optptr,
         uint32_t       *    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()**.

7.1.8 DHCP_option_addrlist()

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

Synopsis

```
bool  DHCP_option_addrlist(
         unsigned char    *      *optptr,
         uint32_t        *        optlen,
         uchar                   opttype,
         _ip_address     *        optval,
         uint32_t                 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()**.

7.1.9 DHCP_option_int16()

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

Synopsis

```
bool  DHCP_option_int16(
  unsigned char   *  *optptr,
  uint32_t       *  optlen,
uchar                opttype,
  uint16_t           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()**.

7.1.10 DHCP_option_int32()

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

Synopsis

```
bool  DHCP_option_int32(
  unsigned char  *  *optptr,
  uint32_t       *   optlen,
  uchar             opttype,
  uint32_t           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()**.

7.1.11 DHCP_option_int8()

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

Synopsis

```
bool  DHCP_option_int8(
  unsigned char  *   *optptr,
  uint32_t       *   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()**.

7.1.12 DHCP_option_string()

Adds a string to the list of DHCP options for DHCP Server.

Synopsis

```
uint32_t  DHCP_option_string(
         unsigned char  *   *optptr,
         uint32_t       *    optlen,
         uchar               opttype,
         char               *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()**.

7.1.13 DHCP_option_variable()

Adds a variable-length option to a list of DHCP options for DHCP Server.

Synopsis

```
uint32_t DHCP_option_variable(
         unsigned char  *   *optptr,
         uint32_t       *   optlen,
         uchar              opttype,
         uchar          *   optdata,
         uint32_t           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()**.

7.1.14 DHCPCLNT_find_option()

Searches a DHCP message for a specific option type.

Synopsis

```
unsigned char *DHCPCLNT_find_option(
          unsigned char  *msgptr,
          uint32_t    msglen,
          uchar       option)
```

Parameters

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

msglen [in] — Number of bytes in the message.

option [in] — 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()**

7.1.15 DHCPCLNT_release()

Releases a DHCP Client no longer needed.

Synopsis

```
unsigned char  *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);
```

7.1.16 DHCPSRV_init()

Starts DHCP Server.

Synopsis

```
uint32_t  DHCPSRV_init(
          char   *name,
          uint32_t    priority,
          uint32_t    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 **DHCPSRV_init()** starts the DHCP server and creates *DHCPSRV_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:

```
DHCPSRV_DATA_STRUCT      dhcpsrv_data;
uchar                    dhcpsrv_options[200];
_ip_address             routers[3];
unsigned char            *optptr;
uint32_t                optlen;
uint32_t                error;

/* Start DHCP Server: */
error = DHCPSRV_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 = 0xFFFFFF00;
/* 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: */
DHCPSRV_option_int8(&optptr, &optlen, 23, 64);
/* MTU: */
DHCPSRV_option_int16(&optptr, &optlen, 26, 1500);
/* Renewal time: */
DHCPSRV_option_int32(&optptr, &optlen, 58, 3600);
/* Rebinding time: */
DHCPSRV_option_int32(&optptr, &optlen, 59, 5400);
/* Domain name: */
DHCPSRV_option_string(&optptr, &optlen, 15, "arc.com");
/* Broadcast address: */
DHCPSRV_option_addr(&optptr, &optlen, 28, 0xC0A800FF);
/* Router list: */
routers[0] = 0xC0A80004;
routers[1] = 0xC0A80005;
routers[2] = 0xC0A80006;
DHCPSRV_option_addrlist( &optptr, &optlen, 3, routers, 3);

/* Serve addresses 192.168.0.129 to 192.168.0.135 inclusive: */
DHCPSRV_ippool_add(0xC0A80081, 7, &dhcpsrv_data, dhcpsrv_options,
                   optptr – dhcpsrv_options);
```

7.1.17 DHCPSRV_ippool_add()

Gives DHCP Server the block of IP addresses to serve.

Synopsis

```
uint32_t  DHCPSRV_ippool_add(
          _ip_address                ipstart,
          uint32_t                   ipnum,
          DHCPSRV_DATA_STRUCT_PTR    params_ptr,
          unsigned char              *optptr,
          uint32_t                   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 **DHCPSRV_ippool_add()** gives the DHCP server the block of IP addresses it serves. The DHCP Server task must be created (by calling **DHCPSRV_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()**
- **DHCPSRV_init()**
- *DHCPSRV_DATA_STRUCT*

Example

See **DHCPSRV_init()**.

7.1.18 DHCPSRV_set_config_flag_off()

Disables address probing.

Synopsis

```
uint32_t DHCPSRV_set_config_flag_off (
        uint32_t                        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 DHCPSVR_FLAG_DO_PROBE flag to **DHCPSRV_set_config_flag_off()**.

This function may be called any time after **DHCPSRV_init()**.

Return Value

- *RTCS_OK* (success)
- Error code (failure)

See Also

- **DHCPSRV_set_config_flag_on()**
- DHCPSRV_init()

Example

```
#define DHCP_DO_PROBING 1
int dhcp_do_probing = DHCP_DO_PROBING;
/*init*/
/*setup*/
if (dhcp_do_probing) {
   DHCPSRV_set_config_flag_on(DHCPSVR_FLAG_DO_PROBE);
   }
else {
   DHCPSRV_set_config_flag_off(DHCPSVR_FLAG_DO_PROBE);
   }
```

7.1.19 DHCPSRV_set_config_flag_on()

Re-enables address probing.

Synopsis

```
uint32_t DHCPSRV_set_config_flag_on (
        uint32_t                         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 *DHCPSVR_FLAG_DO_PROBE* flag to **DHCPSRV_set_config_flag_on()** to reenable probing.

Return Value

- **RTCS_OK** (success)
- Error code (failure)

See Also

- **DHCPSRV_set_config_flag_off()**
- **DHCPSRV_init()**

Example

```
#define DHCP_DO_PROBING 1
int dhcp_do_probing = DHCP_DO_PROBING;
/*init*/
/*setup*/
if (dhcp_do_probing) {
   DHCPSRV_set_config_flag_on(DHCPSVR_FLAG_DO_PROBE);
   }
else {
   DHCPSRV_set_config_flag_off(DHCPSVR_FLAG_DO_PROBE);
   }
```

7.1.20 DNS_init()

Starts a DNS client in order to use DNS services.

Synopsis

```
uint32_t  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()**

7.1.21 ECHOSRV_init()

Starts RFC 862 Echo Server.

Synopsis

```
uint32_t  ECHOSRV_init(
          char  *name,
          uint32_t   priority
          uint32_t   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 by making it a higher number.

Return Value

- *RTCS_OK* (success)
- Error code (failure)

Example

```
error = ECHOSRV_init("Echo server", 7, 1000);
```

7.1.22 EDS_init()

Starts Embedded Debug Server (EDS server).

Synopsis

```
uint32_t  EDS_init(
          char        *name,
          uint32_t        priority,
          uint32_t        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 by making it a higher number.

Return Value

- *RTCS_OK* (success)
- Error code (failure)

7.1.23 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  *   handle)
```

Parameters

handle [in] — Pointer to the Ethernet handle

Description

The function is not a part of RTCS. If you are using MQX RTOS, 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
- **inet_pton()**
- **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);
```

7.1.24 ENET_initialize()

Initializes the interface to the ethernet device.

Synopsis

```
uint32_t  ENET_initialize(
                uint32_t device_num,
                _enet_address address,
                 uint32_t flags,
                _enet_handle * 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 RTOS, 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.

NOTE

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, ?$paratext>."

7.1.25 FTP_close()

Terminates an FTP session.

Synopsis

```
int32_t  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

- **FTP_open()**

Example

See **FTP_open()**.

7.1.26 FTP_command()

Issues a command to the FTP server.

Synopsis

```
int32_t  FTP_command(
         void      *handle,
         char     *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 **FTP_open()**.

7.1.27 FTP_command_data()

Issues a command to the FTP server that requires a data connection.

Synopsis

```
int32_t  FTP_command(
         void     *handle,
         char *command,
         FILE_PTR    ctrl_fd,
         FILE_PTR    data_fd,
         uint32_t     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()

7.1.28 FTP_open()

Starts an FTP session.

Synopsis

```
int32_t  FTP_open(
         void *    *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
   (
      uint32_t  dummy
   )
{ /* Body */
   void *ftphandle;
   int32_t  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 */
```

7.1.29 FTPSRV_init()

Starts the FTP Server.

Synopsis

```
uint32_t FTPSRV_init(
         FTPSRV_PARAM_STRUCT *params)
```

Parameters

params[in] — Parameters of the FTP server.

Description

Function FTPSRV_init() starts the FTP server according to parameters from the _params_ structure. At least one root directory must be set in this structure. If the server is not anonymous (by default it is not), the authentication table must be set; otherwise, you will be unable to use the priviledged server commands. Please see chapter “FTPSRV_PARAM_STRUCT” for further description of each server parameter.

Return Value

- Non—zero value (success)
- Zero (failure)

Example

```
#include "ftpsrv.h"

static const FTPSRV_AUTH_STRUCT ftpsrv_users[] =
    {
        {"developer", "freescale", NULL},
        {NULL,  NULL,        NULL}
    };
FTPSRV_PARAM_STRUCT params;
    uint32_t handle;

    _mem_zero(&params, sizeof(params));
    params.auth_table = (FTPSRV_AUTH_STRUCT*) ftpsrv_users;
    params.root_dir = "a:";
    handle = FTPSRV_init(params);
```

See Also

- FTPSRV_release()
- FTPSRV_PARAM_STRUCT

Caution

This function does NOT copy any of parameters passed to it by pointer. It is not safe to change these parameters in runtime.

7.1.30 FTPSRV_release

Stops the FTP server and releases all of its resources.

Synopsis

```
uint32_t FTPSRV_init(
        FTPSRV_PARAM_STRUCT *params)
```

Parameters

params[in] — Parameters of the FTP server.

Description

This function does opposite of FTPSRV_init(). It shuts down all listening sockets, stops all server tasks and frees all memory used by server. Calling task is blocked until server is stopped and resources are released.

Return Value

- RTCS_OK—shutdown successful.
- RTCS_ERR—shutdown failed.

See Also

- FTPSRV_init()

7.1.31 getaddrinfo()

Gets list of IP addresses for a human-readable host name or address.

Synopsis

```
int32_t getaddrinfo(const char *hostname, const char *servname, const struct addrinfo
*hints, struct addrinfo **res)
```

Parameters

hostname [in] — Host name to resolve. It may be either a host name or a numeric host address string (a dotted-decimal IPv4 address or an IPv6 hex address).

servname [in] — Port number string.

hints [in] — A pointer to an addrinfo structure that provides hints about the type of socket. It is optional (NULL).

res [out] — The address of a location where the function can store a pointer to a result linked list of addrinfo structures.

Return Value

Zero for success, or nonzero if an error occurs.

Description

This function is used to get a list of IP addresses and port numbers for host hostname and service servname.

The hostname and servname arguments are either pointers to NUL-terminated strings or the NULL pointer. An acceptable value for *hostname* is either a valid host name or a numeric host address string consisting of a dotted decimal IPv4 address or an IPv6 address. The *servname* is a decimal port number. At least one of *hostname* and *servname* must be non-null.

hints is an optional pointer to a struct *addrinfo*.

```
struct addrinfo {
    uint16_t        ai_flags;      /* input flags */
    uint16_t        ai_family;     /* protocol family for socket */
    uint32_t        ai_socktype;   /* socket type */
    uint16_t        ai_protocol;   /* protocol for socket */
    unsigned int    ai_addrlen;    /* length of socket-address */
    char            *ai_canonname; /* canonical name for service location */
    struct sockaddr *ai_addr;      /* socket-address for socket */
    struct addrinfo *ai_next;      /* pointer to next in list */
 };
```

This structure can be used to provide hints concerning the type of socket that the caller supports or wishes to use. The caller can supply the following structure elements in hints:

- ai_family - The protocol family that should be used (AF_INET, AF_INET6, AF_UNSPEC). When ai_family is set to AF_UNSPEC, it means the caller will accept any protocol family supported by the TCP/IP stack.
- ai_socktype - Denotes the type of socket that is wanted: SOCK_STREAM or SOCK_DGRAM. When ai_socktype is zero the caller will accept any socket type.
- ai_protocol - Indicates which transport protocol is desired, IPPROTO_UDP or IPPROTO_TCP. If ai_protocol is zero the caller will accept any protocol.

- ai_flags - The ai_flags field to which the hints parameter points shall be set to zero or be the bitwise-inclusive OR of one or more of the values AI_CANONNAME, AI_NUMERICHOST and AI_PASSIVE:
  - AI_CANONNAME - If the AI_CANONNAME bit is set, a successful call to getaddrinfo() will return a NUL-terminated string containing the canonical name of the specified hostname in the ai_canonname element of the addrinfo structure returned.
  - AI_NUMERICHOST - If the AI_NUMERICHOST bit is set, it indicates that hostname should be treated as a numeric string defining an IPv4 or IPv6 address and no name resolution should be attempted.
  - AI_PASSIVE - If the AI_PASSIVE bit is set it indicates that the returned socket address structure is intended for use in a call to bind(2). In this case, if the hostname argument is the null pointer, then the IP address portion of the socket address structure will be set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address. If the AI_PASSIVE bit is not set, the returned socket address structure will be ready for use in a call to connect() for a connection-oriented protocol or connect(), sendto(), or sendmsg() if a connectionless protocol was chosen. The IP address portion of the socket address structure will be set to the loopback address if hostname is the null pointer and AI_PASSIVE is not set.

All other elements of the addrinfo structure passed via hints must be zero or the null pointer.

If hints is the null pointer, getaddrinfo() behaves as if the caller provided a struct addrinfo with ai_family set to AF_UNSPEC and all other elements set to zero or NULL.

After a successful call to getaddrinfo(), *res is a pointer to a linked list of one or more addrinfo structures. The list can be traversed by following the ai_next pointer in each addrinfo structure until a NULL pointer is encountered. The three members ai_family, ai_socktype, and ai_protocol in each returned addrinfo structure are suitable for a call to socket(). For each addrinfo structure in the list, the ai_addr member points to a filled-in socket address structure of length ai_addrlen.

This implementation of getaddrinfo() allows numeric IPv6 address notation with scope identifier, in the form <address>%<zone-id>. By appending the percent character and scope identifier to addresses, one can fill the sin6_scope_id field for addresses.

All of the information returned by getaddrinfo() is dynamically allocated: the addrinfo structures themselves as well as the socket address structures and the canonical host name strings included in the addrinfo structures.

Memory allocated for the dynamically allocated structures created by a successful call to getaddrinfo() is released by the freeaddrinfo() function.

See Also

freeaddrinfo()

getaddrinfo()

Example

```
{
    struct addrinfo     *addrinfo_result;
```

```
            struct addrinfo     *addrinfo_result_first;
            int32_t             retval;
            char                addr_str[RTCS_IP6_ADDR_STR_SIZE];

            _mem_zero(&addrinfo_hints, sizeof(addrinfo_hints));
            addrinfo_hints.ai_flags = AI_CANONNAME;

            retval = getaddrinfo("www.example.com", NULL, NULL, &addrinfo_result);
            if (retval == 0)
            {
                addrinfo_result_first = addrinfo_result;
                /* Print all resolved IP addresses.*/
                while(addrinfo_result)
                {
                    if(inet_ntop(addrinfo_result->ai_family,
                                &((struct sockaddr_in6
*)((*addrinfo_result).ai_addr))->sin6_addr,
                                addr_str, sizeof(addr_str)))
                    {
                        printf("\t%s\n", addr_str);
                    }
                    addrinfo_result = addrinfo_result->ai_next;
                }

                freeaddrinfo(addrinfo_result_first);
            }
            else
            {
                printf("Unable to resolve host\n");
            }
        }
```

7.1.32 freeaddrinfo()

Frees the memory that was allocated by getaddrinfo().

Synopsis

```
void freeaddrinfo(struct addrinfo *ai);
```

Parameters

ai [in] — A pointer to the linked list of addrinfo structures.

Return Value

- None.

Description

This function frees addrifo structures allocated by getaddrinfo(), including any buffers with addrinfo structure members point to (ai_canonname and ai_addr).

See Also

- gethostbyname()
- getaddrinfo()

Example

```
{
    struct addrinfo     *addrinfo_result;
    struct addrinfo     *addrinfo_result_first;
    int32_t             retval;
    char                addr_str[RTCS_IP6_ADDR_STR_SIZE];

    _mem_zero(&addrinfo_hints, sizeof(addrinfo_hints));
    addrinfo_hints.ai_flags = AI_CANONNAME;

    retval = getaddrinfo("www.example.com", NULL, NULL, &addrinfo_result);
    if (retval == 0)
    {
        addrinfo_result_first = addrinfo_result;
        /* Print all resolved IP addresses.*/
        while(addrinfo_result)
        {
            if(inet_ntop(addrinfo_result->ai_family,
                   &((struct sockaddr_in6 *)((*addrinfo_result).ai_addr))->sin6_addr,
                        addr_str, sizeof(addr_str)))
            {
                printf("\t%s\n", addr_str);
            }
            addrinfo_result = addrinfo_result->ai_next;
        }

        freeaddrinfo(addrinfo_result_first);
    }
    else
    {
```

```
            printf("Unable to resolve host\n");
        }
    }
```

7.1.33 gethostbyaddr()

Gets the *HOSTENT_STRUCT* structure for an IP address.

Synopsis

```
hostent *  gethostbyaddr(
        const char *  addr_ptr,
        uint32_t          len,
        uint32_t          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 *     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);
```

7.1.34 gethostbyname()

Gets the *HOSTENT_STRUCT* structure for a host name.

Synopsis

```
HOSTENT_STRUCT_PTR  gethostbyname(
         char   *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, ?$paratext>.”

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
h_addrtype	AF_INET
h_length	sizeof(struct in_addr)

See Also

- **DNS_init()**
- **gethostbyaddr()**
- *HOSTENT_STRUCT*

Example

```
HOSTENT_STRUCT                                                        host;
char                                                            string[30];
char                                                                 *name;
char                                                               *alias1;
char                                                               *alias2;
uint32_t                                                      type, length;
_ip_addr                                                               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     = *(uint32_t*)host->h_addr_list[0];
}
```

7.1.35 getpeername()

Gets the remote-endpoint identifier of a socket.

Synopsis

```
uint32_t  getpeername(
                uint32_t            socket,
                sockaddr     *       name,
                uint16_t     *      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

```
uint32_t      handle;
sockaddr_in  remote_sin;
uint32_t      status;
uint16_t      namelen;

...

namelen = sizeof (sockaddr_in);
status  = getpeername(handle, (struct sockaddr *)&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);
```

```
    printf("\nRemote IP address is %lx",
       remote_sin.sin_addr.s_addr);
  }
```

7.1.36 getsockname()

Gets the local-endpoint identifier of the socket.

Synopsis

```
uint32_t  getsockname(
          uint32_t        socket,
          sockaddr     *  name,
          uint16_t     *  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

```
uint32_t                                                                  handle;
sockaddr_in  local_sin;
uint32_t      status;
uint16_t      namelen;

...

namelen = sizeof (sockaddr_in);
status = getsockname(handle, (struct sockaddr *)&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);
}
```

7.1.37 getsockopt()

Gets the value of the socket option.

Synopsis

```
uint32_t  getsockopt(
          uint32_t         socket,
          int32_t          level,
          uint32_t         optname,
          void            *optval,
          uint32_t  *  o   ptlen)
```

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()**

7.1.38 HTTPSRV_init()

This function initializes and starts the HTTP server.

Synopsis

```
uint32_t HTTPSRV_init(
HTTPSRV_PARAM_STRUCT *params);
```

Parameters

params [in] – pointer to the parameter structure to be used by the HTTP server. Can be null – defaults are used in that case. Any parameter set to zero or NULL is ignored and default value is used instead.

Description

This is the main HTTP function used for initializing and starting the server. It uses information from the parameter to allocate internal memory buffers, set up sockets and sessions.

Any of parameters passed to the server as a pointer must not be changed during runtime, as this may cause memory corruption and other unforeseen consequences. To change server settings the server must be stopped first by using the function HTTPSRV_release() and then started with new parameters.

Return Value

- HTTP server handle if successful, zero if failed.

See Also

- **HTTPSRV_release()**
- **HTTPSRV_PARAM_STRUCT**

Example

```
#include "httpsrv.h"

HTTPSRV_PARAM_STRUCT params;

_mem_zero(&params, sizeof(params));
params.root_dir = "tfs:";
params.index_page = "\\index.html";
server = HTTPSRV_init(&params);
…
HTTPSRV_release(server);
```

CAUTION	This function does NOT copy any of parameters passed to it by pointer. It is not safe to change these parameters in runtime.

7.1.39 HTTPSRV_release()

This function stops the server and releases all its allocated resources.

Synopsis

```
uint32_t HTTPSRV_release(
uint32_t server_h);
```

Parameters

server_h [in] – server handle created by HTTPSRV_init().

Description

When user application needs to stop the server it should call this function. It does opposite of HTTPSRV_init() - it shutdowns all listening sockets, stops all server tasks and frees all memory used by server. This function blocks until shutdown is finished.

Return Value

- HTTPSRV_OK if shutdown was successful, HTTPSRV_ERR otherwise.

See Also

- **HTTPSRV_init()**

7.1.40 HTTPSRV_cgi_write()

This function is used for writing data to the client from the CGI callback.

Synopsis

```
uint32_t httpsrv_cgi_write(
HTTPSRV_CGI_RES_STRUCT* response)
```

Parameters

response [in] – CGI response filled with data. All variables in this structure must be set.

Description

If the user wants to send a response to the client from inside of a CGI callback this function needs to be used. The response structure must be created and set before calling HTTPSRV_cgi_write(). After the first call the HTTP server forms a header according to values in the response and saves it to the session buffer or sends it to the client depending on the buffer state. Also any data in the response are processed (sent/stored). Each subsequent call then writes only data pointed on by *data* variable in the response structure.

Please note that if you have keep-alive functionality enabled and set *content_length* variable of response structure to zero, keep-alive is automatically disabled for active session. For reasoning behind this functionality please see RFC2616 section 4.4 (http://tools.ietf.org/html/rfc2616#section-4.4).

Return Value

Number of bytes successfully processed by the server.

See Also

- **HTTPSRV_cgi_read()**
- **HTTPSRV_CGI_RES_STRUCT**

Example

Please see file *%MQX_PATH%\rtcs\examples\httpsrv\cgi.c* (you can copy link and paste it to the file explorer address bar) for detailed example of how to use this function.

7.1.41 HTTPSRV_cgi_read()

This function is used for reading data provided by the client as the entity body from the CGI callback function.

Synopsis

```
uint32_t httpsrv_cgi_read(
uint32_t ses_handle,
char* buffer,
uint32_t length);
```

Parameters

ses_handle [in] – session handle copied from CGI request structure.

buffer [in] – pointer to buffer in which data from the server will be read.

length [in] – length of buffer in bytes.

Description

This function is to be called whenever user CGI script needs to read data from the client.

Return Value

Number of bytes read.

Example

Please see file *%MQX_PATH%\demo\web_hvac\cgi_hvac.c* (you can copy link and paste it to the file explorer address bar) for detailed example of how to use this function.

7.1.42 HTTPSRV_ssi_write()

This function is used for writing data to the client from the server side include function.

Synopsis

```
HTTPSRV_ssi_write(
uint32_t ses_handle,
char* data,
uint32_t length)
```

Parameters

ses_handle [in] – session handle. This handle is value copied from SSI parameter structure.

data [in] – pointer to data to send to client.

length [in] – length of data in bytes.

Description

All data passed to this function are sent as a part of the HTTP response to the client.

Return Value

Number of bytes written.

Example

```
#include "httpsrv.h"
static _mqx_int usb_status_fn(HTTPSRV_SSI_PARAM_STRUCT* param)
{
    char* str;

    if (usbstick_attached())
    {
        str = "visible";
    }
    else
    {
        str = "hidden";
    }
    HTTPSRV_ssi_write(param->ses_handle, str, strlen(str));
    return 0;
}
```

7.1.43 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()**
- **inet_pton()**
- **IPIF_stats()**
- **TCP_stats()**
- **UDP_stats()**
- **ICMP_STATS**

Example

See **ARP_stats()**

7.1.44 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()**
- **inet_pton()**
- **IPIF_stats()**
- **TCP_stats()**
- **UDP_stats()**
- **IGMP_STATS**

Example

See **ARP_stats()**

7.1.45 inet_pton()

This function converts the character string src into a network address structure.

Synopsis

```
uint32_t inet_pton (
         int32_t af,
         const char *src,
         void *dst,
         unsigned int sizeof_dst)
```

Parameters

af [in] — Family name.
**src[in]* — Pointer to prn form of address.
**dst[out]* — Pointer to bin form of address.
sizeof_dst [in] — Size of dst buffer.

Description

This function converts the character string src into a network address structure in the af address family, then copies the network address structure to dst. The af argument must be either AF_INET or AF_INET6. The following address families are currently supported:

AF_INET

src points to a character string containing an IPv4 network address in dotted-decimal format, "ddd.ddd.ddd.ddd", where ddd is a decimal number of up to three digits in the range 0 to 255. The address is converted to a struct in_addr and copied to dst, which must be sizeof (struct in_addr) (4) bytes (32 bits) long.

AF_INET6

src points to a character string containing an IPv6 network address. The address is converted to a struct in6_addr and copied to dst which must be sizeof (struct in6_addr) (16) bytes (128 bits) long.

The allowed formats for IPv6 addresses follow these rules:

The format is x:x:x:x:x:x:x:x. This form consists of eight hexadecimal numbers, each of which expresses a 16-bit value (i.e., each x can be up to 4 hex digits). A series of contiguous zero values in the preferred format can be abbreviated to ::. Only one instance of :: can occur in an address. For example, the loopback address 0:0:0:0:0:0:0:1 can be abbreviated as ::1. The wildcard address, consisting of all zeroes, can be written as ::.

Return Value

- RTCS_OK (success)

- RTCS_ERROR (failure)

Example

a) IPv4 protocol.

```
uint32_t temp;
inet_pton (AF_INET, prn_addr, &temp, sizeof(temp));
```

b) IPv6 protocol.

```
in6_addr addr6;
inet_pton (AF_INET6,"abcd:ef12:3456:789a:bcde:f012:192.168.24.252",&addr6);
```

7.1.46 inet_ntop()

Converts an address *src from network format (usually a struct eiter in_addr or in6addr, in network byte order) to presentation format (suitable for external display purposes).

Synopsis

```
char *inet_ntop(
         int32_t af,
         const void *src,
         char *dst,
         socklen_t size)
```

Parameters

af [in] — Family name.

**src[in]* — Pointer to an address in network format.

**dst[out]* — Pointer to address in presentation format.

sizeof_dst [in] — Size of dst buffer.

Description

Converts an address *src from network format (usually a struct either in_addr or in6addr in network byte order) to presentation format (suitable for external display purposes). This function is presently valid for AF_INET and AF_INET6.

Return Value

This function returns NULL if a system error occurs, or it returns a pointer to the destination string.

Example

a) IPv4 protocol.

```
in_addr addr;
char prn_addr[RTCS_IP4_ADDR_STR_SIZE];
.......
inet_ntop(AF_INET, &addr, prn_addr, sizeof(prn_addr));
printf("IP addr = %s\n", prn_addr);
.......
```

b) IPv6 protocol.

```
in6_addr addr6;

char prn_addr6[RTCS_IP6_ADDR_STR_SIZE];

.......

inet_ntop(AF_INET6,&addr6, prn_addr6, sizeof(prn_addr6));

printf("IP addr = %s\n",prn_addr6);

.......
```

7.1.47 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()**

7.1.48 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()**
- **inet_pton()**
- **TCP_stats()**
- **UDP_stats()**
- *IPIF_STATS*

Example

See **ARP_stats()**.

7.1.49 ipcfg_init_device()

Initializes the Ehternet device, adds network interface, and sets up the IPCFG context for it.

Synopsis

```
uint32_t ipcfg_init_device(
         uint32_t 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)

uint32_t setup_network(void)
{
  uint32_t               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 Ehternet 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;
}
```

7.1.50 ipcfg_init_interface()

Setups IPCFG context for already initialized device and its interface.

Synopsis

```
uint32_t ipcfg_init_interface(
          uint32_t 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 intialized 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)


uint32_t setup_network(void)
{
  uint32_t               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);
}
```

7.1.51 ipcfg_bind_boot()

Binds Ethernet device to network using the BOOT procotol.

Synopsis

```
uint32_t ipcfg_bind_boot(
         uint32_t 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)

uint32_t setup_network(void)
{
    uint32_t               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;

    TFTTIP = 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);
}
```

7.1.52 ipcfg_bind_dhcp()

Binds Ethernet device to network using DHCP protocol (polling mode).

Synopsis

```
uint32_t ipcfg_bind_dhcp(
        uint32_t device,
        bool 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. If it 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

```
uint32_t ipcfg_bind_dhcp_wait(uint32_t device, bool try_auto_ip,
         IPCFG_IP_ADDRESS_DATA_PTR auto_ip_data)
{
  uint32_t result = IPCFG_OK;

  do
  {
    if (result == RTCSERR_IPCFG_BUSY) _time_delay(200);
    result = ipcfg_bind_dhcp(device, try_auto_ip);
  } while (result == RTCSERR_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 == RTCSERR_IPCFG_BUSY);
  return result;
}
```

7.1.53 ipcfg_bind_dhcp_wait()

Binds Ethernet device to network using DHCP protocol (blocking mode).

Synopsis

```
uint32_t ipcfg_bind_dhcp_wait(
          uint32_t device,
          bool 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)

uint32_t setup_network(void)
{
    uint32_t               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;

    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);
}
```

7.1.54 ipcfg_bind_staticip()

Binds Ethernet device to network using constant (static) IPv4 address information.

Synopsis

```
uint32_t ipcfg_bind_staticip(
        uint32_t 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 IPv4 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()**

7.1.55 ipcfg_get_device_number()

Returns the Ethernet device number for given RTCS interface.

Synopsis

```
uint32_t 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()**

7.1.56 ipcfg_add_interface()

Add new interface and returns corresponding device number.

Synopsis

```
uint32_t ipcfg_add_interface(
        uint32_t 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()**

7.1.57 ipcfg_get_ihandle()

Returns the RTCS interface handle for given Ethernet device number.

Synopsis

```
_rtcs_if_handle ipcfg_get_ihandle(
        uint32_t 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()**

7.1.58 ipcfg_get_mac()

Returns the Ethernet MAC address.

Synopsis

```
bool ipcfg_get_mac(
        uint32_t 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.

7.1.59 ipcfg_get_state()

Returns the IPCFG state for a given Etherent device.

Synopsis

```
IPCFG_STATE ipcfg_get_state(
        uint32_t 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

7.1.60 ipcfg_get_state_string()

Converts IPCFG status value to string.

Synopsis

```
const char *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()**

7.1.61 ipcfg_get_desired_state()

Returns the target IPCFG state for a given Etherent device.

Synopsis

```
IPCFG_STATE ipcfg_get_desired_state(
        uint32_t 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()**

7.1.62 ipcfg_get_link_active()

Returns immediate Ethernet link state.

Synopsis

```
bool ipcfg_get_link_active
          uint32_t device                                                    )
```

Parameters

device [in] — device identification

Description

This function returns the immediate Etherenet 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()**

7.1.63 ipcfg_get_dns_ip()

Returns the n-th DNS IPv4 address from the registered DNS list.

Synopsis

```
_ip_address ipcfg_get_dns_ip(
        uint32_t device,
        uint32_t n)
```

Parameters

device [in] — device identification

n [in] — DNS IP address index

Description

This function may be used to retrieve all DNS IPv4 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()**

7.1.64 ipcfg_add_dns_ip()

Registers the DNS IPv4 address with the Ethernet device.

Synopsis

```
bool ipcfg_add_dns_ip (
        uint32_t device,
        _ip_address address)
```

Parameters

device [in] — device identification

address [in] — DNS IPv4 address to add

Description

This function adds the DNS IPv4 address to the list assigned to given Ethernet device.

Return Value

TRUE if successful, *FALSE* otherwise

See Also

- **ipcfg_get_dns_ip()**
- **ipcfg_del_dns_ip()**

7.1.65 ipcfg_del_dns_ip()

Unregisters the DNS IPv4 address.

Synopsis

```
bool ipcfg_del_dns_ip (
        uint32_t device,
        _ip_address address)
```

Parameters

device [in] — device identification

address [in] — DNS IPv4 address to be removed

Description

This function removes the DNS IPv4 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()**

7.1.66 ipcfg_get_ip()

Returns an immediate IPv4 address information bound to Ethernet device.

Synopsis

```
bool ipcfg_get_ip(
         uint32_t device,
         IPCFG_IP_ADDRESS_DATA_PTR data)
```

Parameters

device [in] — device identification

data [in] — pointer to IPv4 address information (IP address, mask and gateway)

Description

This function returns the immediate IPv4 address information bound to given Ethernet device.

Return Value

TRUE if successful and *data* structure filled. FALSE if there is an error.

See Also

- **ipcfg_get_dns_ip()**

7.1.67 ipcfg_get_tftp_serveraddress()

Returns TFTP server address, if any.

Synopsis

```
_ip_address ipcfg_get_tftp_serveraddress(
        uint32_t 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()**

7.1.68 ipcfg_get_tftp_servername()

Returns TFTP servername, if any.

Synopsis

```
unsigned char *ipcfg_get_tftp_serveraddress(uint32_t 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()**

7.1.69 ipcfg_get_boot_filename()

Returns the TFTP boot filename, if any.

Synopsis

```
unsigned char *ipcfg_get_boot_filename(uint32_t 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()**

7.1.70 ipcfg_poll_dhcp()

Polls (finishes) the Ethernet device DHCP binding process.

Synopsis

```
uint32_t ipcfg_poll_dhcp(
         uint32_t device,
         bool 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()**

7.1.71 ipcfg_task_create()

Creates and starts the IPCFG Ethernet link status-monitoring task.

Synopsis

```
uint32_t ipcfg_task_create(
        uint32_t priority,
        uint32_t 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(uint32_t 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();
  }
}
```

7.1.72 ipcfg_task_destroy()

Signals the exit request to the IPCFG task.

Synopsis

```
void ipcfg_task_destroy(
        bool 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()**.

7.1.73 ipcfg_task_status()

Checks whether the IPCFG Ethernet link status monitorin task is running.

Synopsis

```
bool 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().**

7.1.74 ipcfg_task_poll()

One step of the IPCFG Ethernet link status monitoring task.

Synopsis

```
bool 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()**.

7.1.75 ipcfg_unbind()

Unbinds the Ethernet device from network.

Synopsis

```
uint32_t ipcfg_unbind(
         uint32_t device)
```

Parameters

device [in] — device identification

Description

This function releases the IPv4 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(uint32_t param)
{
  setup_network();

  ...

  ipcfg_unbind();
  while (1) {};
}
```

7.1.76 ipcfg6_bind_addr()

Binds IPv6 address information to the Ethernet device.

Synopsis

```
uint32_t  ipcfg6_bind_addr(
    uint32_t                     device,
    IPCFG6_BIND_ADDR_DATA_PTR    ip_data)
```

Parameters

device [in] — device identification

ip_data [in] — pointer to bind ip data structure

Description

This function tries to bind device to network using given IPv6 address data 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

- **ipcfg6_unbind_addr()**

Example

See example in *shell/source/rtcs/sh_ipconfig.c, Shell_ipconfig_staticip()*.

7.1.77 ipcfg6_unbind_addr()

Unbinds the IPv6 address from the Ethernet device.

Synopsis

```
uint32_t ipcfg6_unbind_addr(
    uint32_t                    device,
    IPCFG6_UNBIND_ADDR_DATA_PTR ip_data)
```

Parameters

device [in] — device identification

ip_data[in] — pointer to unbind ip data structure

Description

This function releases the IPv6 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

- **ipcfg6_bind_addr()**

Example

See example in *shell/source/rtcs/sh_ipconfig.c, Shell_ipconfig_unbind6()*.

7.1.78 ipcfg6_get_addr()

Returns an IPv6 address information bound to the Ethernet device.

Synopsis

```
uint32_t ipcfg6_get_addr(uint32_t device, uint32_t n, IPCFG6_GET_ADDR_DATA_PTR data)
```

Parameters

device [in] — device identification

n [in] — sequence number of IPv6 address to retrieve (from 0).

data [in/out] — pointer to IPv6 address information structure (IPv6 address, address state and type).

Description

This function returns the IPv6 address information bound (manually or by IPv6 Stateless Autoconfiguration process) to the given Ethernet device.

One interface may have several bound IPv6 addresses.

Return Value

- *RTCS_OK* (success, *data* is filled)
- *RTCS_ERROR* (failure, *n*-th address is not available)

See Also

- **ipcfg6_bind_addr()**
- **ipcfg6_unbind_addr()**

Example

```
/* Print all bound IPv6 addresses.*/
{
    IPCFG6_GET_ADDR_DATA    addr_data;
    char                    addr_str[RTCS_IP6_ADDR_STR_SIZE];
    int                     n;

    for(n=0;(ipcfg6_get_addr(BSP_DEFAULT_ENET_DEVICE, n, &addr_data) == RTCS_OK); n++)
    {
        /* Convert IPv6 address to string presentation and print it.*/
        if(inet_ntop(AF_INET6, &addr_data.ip_addr, addr_str, sizeof(addr_str)))
        {
            printf("IP6[%d] : %s\n", n, addr_str);
        }
    }

}
```

7.1.79 ipcfg6_get_dns_ip()

Returns the n-th DNS IPv6 address from the registered DNS list of the Ethernet device.

Synopsis

```
uint32_t ipcfg6_get_addr(uint32_t device, uint32_t n, IPCFG6_GET_ADDR_DATA_PTR data)
```

Parameters

device [in] — device identification

n [in] — sequence number of IPv6 address to retrieve (from 0).

data [in/out] — pointer to IPv6 address information structure (IPv6 address, address state and type).

Description

This function returns the IPv6 address information bound (manually or by IPv6 Stateless Autoconfiguration process) to the given Ethernet device.

One interface may have several bound IPv6 addresses.

Return Value

- *RTCS_OK* (success, *data* is filled)
- *RTCS_ERROR* (failure, *n*-th address is not available)

See Also

- **ipcfg6_bind_addr()**
- **ipcfg6_unbind_addr()**

Example

```
/* Print all bound IPv6 addresses.*/
{
    IPCFG6_GET_ADDR_DATA    addr_data;
    char                    addr_str[RTCS_IP6_ADDR_STR_SIZE];
    int                     n;

    for(n=0;(ipcfg6_get_addr(BSP_DEFAULT_ENET_DEVICE, n, &addr_data) == RTCS_OK); n++)
    {
        /* Convert IPv6 address to string presentation and print it.*/
        if(inet_ntop(AF_INET6, &addr_data.ip_addr, addr_str, sizeof(addr_str)))
        {
            printf("IP6[%d] : %s\n", n, addr_str);
        }
    }
}
```

7.1.80 ipcfg6_add_dns_ip()

Registers the DNS IPv6 address with the Ethernet device.

Synopsis

```
uint32_t ipcfg6_get_addr(uint32_t device, uint32_t n, IPCFG6_GET_ADDR_DATA_PTR data)
```

Parameters

device [in] — device identification

n [in] — sequence number of the IPv6 address to retrieve (from 0).

Description

This function adds the DNS IPv6 address to the list assigned to given Ethernet device.

Return Value

TRUE if successful, *FALSE* otherwise

See Also

- ipcfg6_get_dns_ip()
- ipcfg6_del_dns_ip()

Example

```
/* Register DNS IPv6 address with the Ethernet device.*/
    {
        char            *addr_str = "2001:470:1234:567:4c39:64fa:1caa:44c8";
        in6_addr        dns6_addr;

        if(inet_pton(AF_INET6, addr_str, &dns6_addr, sizeof(dns6_addr)) == RTCS_OK)
        {
            if(ipcfg6_add_dns_ip(BSP_DEFAULT_ENET_DEVICE, &dns6_addr) == TRUE)
            {
                printf("Adding DNS address is successful.\n");
            }
            else
            {
                printf("Adding DNS address is failed.\n");
            }
        }

    }
```

7.1.81 ipcfg6_del_dns_ip()

Returns an IPv6 address information bound to the Ethernet device.

Synopsis

```
uint32_t ipcfg6_get_addr(uint32_t device, uint32_t n, IPCFG6_GET_ADDR_DATA_PTR data)
```

Parameters

device [in] — device identification

dns_addr [in] — DNS IPv6 address to be removed.

Description

This function removes the DNS IPv6 address from the list assigned to given Ethernet device.

Return Value

TRUE if successful, *FALSE* otherwise

See Also

- ipcfg6_get_dns_ip()
- ipcfg6_add_dns_ip()

Example

```
/* Print all bound IPv6 addresses.*/
{
    IPCFG6_GET_ADDR_DATA    addr_data;
    char                    addr_str[RTCS_IP6_ADDR_STR_SIZE];
    int                     n;

    for(n=0;(ipcfg6_get_addr(BSP_DEFAULT_ENET_DEVICE, n, &addr_data) == RTCS_OK); n++)
    {
```

7.1.82 ipcfg6_get_scope_id()

Returns an IPv6 address information bound to the Ethernet device.

Synopsis

```
uint32_t ipcfg6_get_scope_id (uint32_t device)
```

Parameters

device [in] — device identification

Description

This function returns Scope ID (interface identifier) assigned to the Ethernet device.

The Scope ID is used to indicate the network interface over which traffic is sent and received.

Return Value

- Scope ID (success)
- 0 (failure)

See Also

- **ipcfg6_bind_addr()**

Example

```
/* Print all bound IPv6 addresses.*/
{
    IPCFG6_GET_ADDR_DATA    addr_data;
    char                    addr_str[RTCS_IP6_ADDR_STR_SIZE];
    int                     n;

    for(n=0;(ipcfg6_get_addr(BSP_DEFAULT_ENET_DEVICE, n, &addr_data) == RTCS_OK); n++)
    {
```

7.1.83 iwcfg_set_essid()

Synopsis

```
uint32_t iwcfg_set_essid
    (
        uint32_t  dev_num,
        char *essid
    )
```

Parameters

dev_num [in] — Device identification (index).

essid [in] — Pointer to ESSID (Extended Service Set Identifier) string.

Description

This function sets to device identificated 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
int32_t                  error;

/* IP configuration */
error = RTCS_create();
ENET_get_mac_address (DEFAULT_DEVICE, ENET_IPADDR, enet_address);
error = ipcfg_init_device (DEFAULT_DEVICE, enet_address);
/* Set SSID */
iwcfg_set_essid (DEFAULT_DEVICE, SSID);
iwcfg_commit( DEFAULT_DEVICE );
/* end of IP configuration */
error = ipcfg_bind_staticip (DEFAULT_DEVICE, &ip_data);
```

7.1.84 iwcfg_get_essid()

Synopsis

```
uint32_t iwcfg_get_essid
    (
        uint32_t  dev_num,
        char *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_ssid (DEFAULT_DEVICE, &ssid_name);
```

7.1.85 iwcfg_commit()

Synopsis

```
uint32_t iwcfg_commit
    (
        uint32_t 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);
```

7.1.86 iwcfg_set_mode()

Synopsis

```
uint32_t iwcfg_set_mode
    (
        uint32_t  dev_num,
        char *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);
```

7.1.87 iwcfg_get_mode()

Synopsis

```
uint32_t iwcfg_get_mode
    (
        uint32_t  dev_num
        char *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);
```

7.1.88 iwcfg_set_wep_key()

Synopsis

```
uint32_t iwcfg_set_wep_key
    (
        uint32_t  dev_num,
        char      *wep_key,
        uint32_t  key_len,
        uint32_t  key_index
    )
```

Parameters

dev_num [in] — Device identification (index).

wep_key [in] — Wep_key.

key_len [in] — Lenght of the key.

key_index [in] — Aditional 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);
```

7.1.89 iwcfg_get_wep_key()

Synopsis

```
uint32_t iwcfg_get_wep_key
    (
        uint32_t  dev_num,
        char      *wep_key,
        uint32_t  key_index
    )
```

Parameters

dev_num [in] — Device identification (index).

wep_key [in] — Wep_key.

key_index [in] — Aditional optional device specific parameters. Index must be lower than 256.

Description

Get the wep key.

Return Value

- ENET_OK (success)
- ENETERR_INVALID_DEVICE

7.1.90 iwcfg_set_passphrase()

Synopsis

```
uint32_t iwcfg_set_passphrase
    (
        uint32_t  dev_num,
        char     *passphrase
    )
```

Parameters

dev_num [in] — Device identification (index).

passphrase [in] — SSID passpharse.

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);
```

7.1.91 iwcfg_get_passphrase()

Synopsis

```
uint32_t iwcfg_get_passphrase
    (
        uint32_t  dev_num,
        char    *passphrase
    )
```

Parameters

dev_num [in] — Device identification (index).

passphrase [out] — SSID passpharse (string).

Description

Get the wpa passpharse from initialized wifi device.

Return Value

- ENET_OK (success)
- ENETERR_INVALID_DEVICE

7.1.92 iwcfg_set_sec_type()

Synopsis

```
uint32_t iwcfg_set_sec_type
    (
        uint32_t  dev_num,
        char     *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.

7.1.93 iwcfg_get_sectype()

Synopsis

```
uint32_t iwcfg_get_sec_type
    (
        uint32_t  dev_num,
        char      *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

7.1.94 iwcfg_set_power()

Synopsis

```
uint32_t iwcfg_set_power
    (
        uint32_t dev_num,
        uint32_t pow_val,
        uint32_t flags
    )
```

Parameters

dev_num [in] — Device identification (index).

pow_val [in] — Power in dBm.

flags [in] — Device specific options.

Description

Sets the transmit power in dBm for cards supporting multiple transmit powers. If W is the power in Watt, the power in dBm is P = 30 + 10.log(W).

Return Value

- ENET_OK (success)
- ENETERR_INVALID_DEVICE

7.1.95 iwcfg_set_scan()

Synopsis

```
uint32_t iwcfg_set_scan
    (
        uint32_t dev_num,
        char  *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 Wi-Fi 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"
int32_t                  error;

/* IP configuration */
error = RTCS_create();
ENET_get_mac_address (DEFAULT_DEVICE, ENET_IPADDR, enet_address);
error = ipcfg_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
```

```
ssid = Faz
bssid = 0:21:91:12:da:cc
channel = 1
strength = ####.
indicator = 172
---

scan done.
```

7.1.96 listen()

Puts the stream socket into the listening state.

Synopsis

```
uint32_t  listen(
          uint32_t  socket,
          uint16_t  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()**.

7.1.97 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()**.

7.1.98 MIB_find_objectname()

Find object in table.

Synopsis

```
bool MIB_find_objectname(uint32_t op, void *index, void * *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

7.1.99 MIB_set_objectname()

Set name for writable object in table.

Synopsis

```
uint32_t MIB_set_objectname(void *instance, unsigned char *value_ptr, uint32_t 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

7.1.100 NAT_close()

Stops Network Address Translation.

Synopsis

```
uint32_t  NAT_close(void)
```

Return Value

- *RTCS_OK* (success)

See Also

- **NAT_init()**

7.1.101 NAT_init()

Starts Network Address Translation.

Synopsis

```
uint32_t  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

Freescale 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, ?$paratext>.”

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*

7.1.102 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*

7.1.103 ping()

See **RTCS_ping()**.

7.1.104 PPP_init()

Initializes PPP Driver for the PPP link.

Synopsis

```
_ppp_handle PPP_init(
        PPP_PARAM_STRUCT* params
)
```

Parameters

params[in/out] — parameters for PPP initialization, IPCP handle created by PPP is stored here.

Description

Function **PPP_initialize()** fails, if RTCS cannot do any one of the following:

- Open low level device (i.e "ittyd:").
- Initialize HDLC layer.
- Initialize LCP layer.
- Allocate message pool.
- Create receive and transmit tasks.
- Open HDLC layer.
- Add PPP interface.
- Bind IP address on IPCP layer.

Return Value

- PPP device handle.
- Null.

See Also

- *PPP_release*
- *PPP_pause*
- *PPP_resume*
- *PPP_PARAM_STRUCT*

Example

```
/* Start PPP in listen mode */
{
        PPP_PARAM_STRUCT  params;
        uint32_t          handle;

        mem_zero(&params, sizeof(params));
        params.device = "ittyd:";
        /* Set local IP address to 192.168.1.201 */
        params.local_addr = 0xC0A801C9;
        /* Set remote IP address to 192.168.1.202 */
        params.remote_addr = 0xC0A801CA;
        params.listen_flag = 1;
```

```
        /* Init PPP */
        handle = PPP_init(&params);
        if (handle == NULL)
        {
                fprintf(stderr,"PPP initialization failed.");
        }
        else
        {
                PPP_pause(handle);
                /* Do something on ittyd: device here */
                PPP_resume(handle);
                if (PPP_release(ppp_conn->PPP_HANDLE) != RTCS_OK)
        {
                fprintf(stderr,"Failed to release PPP connection.");
                }
        }
}
```

7.1.105 PPP_release()

Deinitializes PPP driver and releases low-level device.

Synopsis

```
uint32_t PPP_release(
_ppp_handle  handle
)
```

Parameters

handle[in]— handle to PPP device.

Description

This function is used to release all resources used by PPP device. It does following steps:

- Unbind IP address on IPCP layer.
- Terminate PPP internal RX and TX tasks.
- Close HDLC layer.
- Shutdown LCP layer.
- Deallocate message pool.
- Close low level device.
- Remove PPP interface.
- Free memory.

Return Value

- RTCS_OK if release was successful.
- Error code.

See Also

- *PPP_init*

Example

Please see PPP_init() as an example.

7.1.106 PPP_pause()

Pauses the PPP state machine, so low-level device can be used for other communication.

Synopsis

```
uint32_t PPP_pause(
_ppp_handle handle
```

Parameters

handle[in] — handle to PPP device to be paused.

Description

When PPP is paused, all communication with remote peer is stopped and low level device is available for other use.

This typically includes sending AT commands to GPRS modem and performing handshake with windows machine.

Return Value

- RTCS_OK if successful.
- Error code.

See Also

- *PPP_resume*

Example

Please see PPP_init() as an example.

7.1.107 PPP_resume()

Resumes the PPP state machine.

Synopsis

```
uint32_t PPP_resume(
_ppp_handle handle
)
```

Parameters

handle[in] — handle to PPP device to be resumed.

Description

This function is used to restore communication over PPP link and works as counterpart of PPP_pause function.

Return Value

- RTCS_OK if successful.
- Error code.

See Also

- *PPP_pause*

Example

Please see PPP_init() as an example.

7.1.108 recv()

Provides RTCS with incoming buffer.

7.1.108.1 Synopsis

```
int32_t  recv(
         uint32_t      socket,
         char  *       buffer,
         uint32_t      buflen,
         uint32_t      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, **recv()** returns *RTCS_ERROR*, rather than zero as BSD 4.4 specifies. A subsequent call to **RTCS_geterror()** returns *RTCSERR_TCP_CONN_CLOSING.*

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

```
uint32_t  handle;
char     buffer[20000];
uint32_t  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);
}
```

7.1.109 recvfrom()

Provides RTCS with the buffer in which to place data that is incoming on the datagram socket.

Synopsis

```
int32_t recvfrom(
        uint32_t              socket,
        char          *       buffer,
        uint32_t              buflen,
        uint32_t              flags,
        sockaddr      *       fromaddr,
        uint16_t             *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.

```
uint32_t      handle;
sockaddr_in  remote_sin;
uint32_t      count;
char         my_buffer[500];
uint16_t      remote_len = sizeof(remote_sin);

...

count = recvfrom(handle, my_buffer, 500, 0, (struct sockaddr *) &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);
}
```

7.1.110 RTCS_attachsock()

Takes ownership of the socket.

Synopsis

```
uint32_t RTCS_attachsock(
         uint32_t   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
(
  uint32_t  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 {
...
```

7.1.111 RTCS_create()

Creates RTCS.

Synopsis

```
uint32_t 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, ?$paratext>.”

7.1.112 RTCS_detachsock()

Relinquishes ownership of the socket.

Synopsis

```
uint32_t  RTCS_detachsock(
          uint32_t   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()**.

7.1.113 RTCS_exec_TFTP_BIN()

Download and run the binary boot file.

Synopsis

```
uint32_t RTCS_exec_TFTP_BIN(
          _ip_address  server,
          char        *filename,
          unsigned char    *download_address,
          unsigned char    *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	**RTCS_if_bind_ BOOTP()**	• SADDR • BOOTFILE	*BOOTP_DATA_STRUCT*
DHCP	**RTCS_if_bind_ DHCP()**	• SADDR • FILE	*DHCPSRV_DATA_STRUCT*

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.

```
uint32_t boot_function(void) {
  BOOTP_DATA_STRUCT  boot_data;
  _enet_handle       ehandle;
  _rtcs_if_handle    ihandle;
  uint32_t            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*)boot_data.BOOTFILE,
                             (unsigned char*)DOWNLOAD_ADDR,
                             (unsigned char*)RUN_ADDR);
  return error;
}
```

7.1.114 RTCS_exec_TFTP_COFF()

Downloads and runs the COFF boot file.

Synopsis

```
uint32_t  RTCS_exec_TFTP_COFF(
          _ip_address  server,
          char      *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*

7.1.115 RTCS_exec_TFTP_SREC()

Downloads and runs the S-Record boot file.

Synopsis

```
uint32_t  RTCS_exec_TFTP_SREC(
         _ip_address  server,
         char     *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.

```
uint32_t boot_function(void)
{
  BOOTP_DATA_STRUCT  boot_data;
  _enet_handle       ehandle;
  _rtcs_if_handle    ihandle;
  uint32_t            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*)boot_data.BOOTFILE);
    return error;
  }
```

7.1.116 RTCS_gate_add()

Adds the gateway to RTCS.

Synopsis

```
uint32_t  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);
```

7.1.117 RTCS_gate_add_metric()

Adds a gateway to the RTCS routing table and assign it's metric.

Synopsis

```
uint32_t  RTCS_gate_add_metric(
          _ip_address  gateway,
          _ip_address  network,
          _ip_address  netmask
          _uint16_t     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)
```

7.1.118 RTCS_gate_remove()

Removes a gateway from the routing table.

Synopsis

```
uint32_t  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);
```

7.1.119 RTCS_gate_remove_metric()

Removes a specific gateway from the routing table.

Synopsis

```
uint32_t  RTCS_gate_remove_metric(
          _ip_address   gateway,
          _ip_address   network,
          _ip_address   netmask
          _uint16_t      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)
```

7.1.120 RTCS_geterror()

Gets the reason why the RTCS function returned an error for the socket.

Synopsis

```
uint32_t  RTCS_geterror(
          uint32_t  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()**.

7.1.121 RTCS_if_add()

Adds device interface to RTCS.

Synopsis

```
uint32_t  RTCS_if_add(
          void                 *dev_handle,
          RTCS_IF_STRUCT_PTR    callback_ptr,
          _rtcs_if_handle *   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_init()**
- **RTCS_create()**
- **RTCS_if_bind()**
- *RTCS_IF_STRUCT*

Example

See Section 2.15.6, ?$paratext>.”

7.1.122 RTCS_if_bind()

Binds the IP address and network mask to the device interface.

Synopsis

```
uint32_t  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, ?$paratext>.”

7.1.123 RTCS_if_bind_BOOTP()

Gets an IP address using BootP and binds it to the device interface.

Synopsis

```
uint32_t  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;

uint32_t boot_function(void)
{
  BOOTP_DATA_STRUCT  boot_data;
  _enet_handle       ehandle;
  _rtcs_if_handle    ihandle;
  uint32_t            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*)boot_data.BOOTFILE);

    return error;
}
```

7.1.124 RTCS_if_bind_DHCP()

Gets an IP address using DHCP and binds it to the device interface.

Synopsis

```
uint32_t  RTCS_if_bind_DHCP(
          _rtcs_if_handle       rtcs_if_handle,
          DHCP_DATA_STRUCT_PTR  callback_ptr,
          char                 *optptr,
          uint32_t              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 r*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;
uint32_t           error;
uint32_t           optlen = 100; /* Use the size that you need for
                                  the number of params that you
                                  are using with DHCP */
uchar             option_array[100];
uchar *       optptr;
```

```
DHCP_DATA_STRUCT  params;
uchar             parm_options[3] =  {DHCPOPT_SERVERNAME,
                                      DHCPOPT_FILENAME,
                                      DHCPOPT_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, DHCPOPT_LEASE, 180);
/* Request a TFTP Server, FILENAME, and Finger Server: */
DHCP_option_variable(&optptr, &optlen, DHCPOPT_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. */
```

7.1.125 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.*

7.1.125.1 Synopsis

```
uint32_t  RTCS_if_bind_DHCP_flagged(
          _rtcs_if_handle        rtcs_if_handle,
          DHCP_DATA_STRUCT_PTR   params,
          char                  *optptr,
          uint32_t                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;
uint32_t           error;
uint32_t           optlen = 100; /* Use the size that you need for
                                the number of params that you
                                are using with DHCP */
uchar             option_array[100];
uchar *       optptr;
DHCP_DATA_STRUCT  params;
uchar             parm_options[3] =  {DHCPOPT_SERVERNAME,
                                      DHCPOPT_FILENAME,
                                      DHCPOPT_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, DHCPOPT_LEASE, 180);
/* Request a TFTP Server, FILENAME, and Finger Server: */
DHCP_option_variable(&optptr, &optlen, DHCPOPT_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_timed()

Gets an IP address using DHCP and binds it to the device interface within a timeout.

Synopsis

```
uint32_t  RTCS_if_bind_DHCP_timed(
          _rtcs_if_handle       rtcs_if_handle,
          DHCP_DATA_STRUCT_PTR  params,
          char                 *optptr,
          uint32_t              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;
uint32_t           error;
uint32_t           optlen = 100; /* Use the size that you need for
                                    the number of params that you
                                    are using with DHCP */
uchar             option_array[100];
uchar *       optptr;
DHCP_DATA_STRUCT  params;
uchar             parm_options[3] =  {DHCPOPT_SERVERNAME,
                                      DHCPOPT_FILENAME,
                                      DHCPOPT_FINGER_SRV};
uint32_t           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, DHCPOPT_LEASE, 180);
/* Request a TFTP Server, FILENAME, and Finger Server: */
DHCP_option_variable(&optptr, &optlen, DHCPOPT_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. */
```

7.1.127 RTCS_if_bind_IPCP()

Binds an IP address to the PPP device interface.

Synopsis

```
uint32_t  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, ?$paratext>.”

Return Value

- *RTCS_OK* (success)
- Error code (failure)

See Also

- **PPP_init()**
- **RTCS_if_add()**
- **RTCS_if_bind()**
- *IPCP_DATA_STRUCT*

Example

Initialize PPP and bind to the interface.

```
void boot_done(void *sem) {
  _lwsem_post(sem);
}

int32_t 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;
}
```

7.1.128 RTCS_if_rebind_DHCP()

Binds a previously used IP address to the device interface.

Synopsis

```
uint32_t  RTCS_if_rebind_DHCP(
          _rtcs_if_handle        rtcs_if_handle,
          _ip_address            address,
          _ip_address            netmask,
          uint32_t                lease,
          _ip_address            server,
          DHCP_DATA_STRUCT_PTR   params,
          unsigned char              *optptr,
          uint32_t                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;
uint32_t           error;
uint32_t           optlen = 100; /* Make large enough for the number
                                    of your DHCP options */
uchar             option_array[100];
uchar *       optptr;
DHCP_DATA_STRUCT  params;
uchar             parm_options[3] =  {DHCPOPT_SERVERNAME,
                                      DHCPOPT_FILENAME,
                                      DHCPOPT_FINGER_SRV};
in_addr            rebind_address, rebind_mask, rebind_server;
uint32_t           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, DHCPOPT_LEASE, 180);
/* Request a TFTP Server, FILENAME, and Finger Server: */
DHCP_option_variable(&optptr, &optlen, DHCPOPT_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;
 }
```

7.1.129 RTCS_if_remove()

Removes the device interface from RTCS.

Synopsis

```
uint32_t  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()**

7.1.130 RTCS_if_unbind()

Unbinds the IP address from the device interface.

Synopsis

```
uint32_t  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()**

7.1.131 RTCS_load_TFTP_BIN()

Downloads the binary file.

Synopsis

```
uint32_t  RTCS_load_TFTP_BIN(
          _ip_address  server,
          char     *filename,
          unsigned char    *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*

7.1.132 RTCS_load_TFTP_COFF()

Downloads the COFF boot file.

Synopsis

```
uint32_t  RTCS_load_TFTP_COFF(
          _ip_address  server,
          char       *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*

7.1.133 RTCS_load_TFTP_SREC()

Downloads the S-Record file.

Synopsis

```
uint32_t  RTCS_load_TFTP_SREC(
          _ip_address   server,
          char         *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*

7.1.134 RTCS_ping()

Sends an ICMP echo-request packet to an IP address and waits for a reply.

Synopsis

```
uint32_t RTCS_ping(PING_PARAM_STRUCT *params)
```

Parameters

params [in] — pointer to the PING_PARAM_STRUCT parameter structure, to be used by the PING function. This should not be NULL.

Description

Function **RTCS_ping()** is the RTCS implementation of **ping**. It sends an ICMPv4 or ICMPv6 echo-request packet to the specified IPv4 or IPv6 address and waits for a reply.

Return Value

- *RTCS_OK* (success)
- Error code (failure)

See Also

- PING_PARAM_STRUCT

Example

```
/* Send ICMPv4 echo request to the IPv4 192.168.0.5 address.*/
{
    uint32_t           error;
    PING_PARAM_STRUCT ping_params;

    /* Set ping parameters.*/
    _mem_zero(&ping_params, sizeof(ping_params)); /* Zero input parameters.*/
    ping_params.addr.sa_family = AF_INET; /* Set IPv4 addr. family */
    /* IPv4 192.168.0.5 address.*/
    ((sockaddr_in *)(&ping_params.addr))->sin_addr.s_addr = IPADDR(192,168,0,5);
    /* Wait interval in milliseconds */
    ping_params.timeout = 1000;

    /* Send PING - ICMP request.
     * It will block the application while await ICMP echo reply.*/
    error = RTCS_ping(&ping_params);

    if (error)
    {
        if (error == RTCSERR_ICMP_ECHO_TIMEOUT)
            printf("Request timed out\n");
        else
            printf("Error 0x%04lX \n", error);
    }
    else
    {
```

```
            if(ping_params.round_trip_time < 1)
                printf("Reply time<1ms\n");
            else
                printf("Reply time=%ldms\n", ping_params.round_trip_time);
        }
    }
```

7.1.135 RTCS_request_DHCP_inform()

Requests a DHCP information message.

Synopsis

```
uint32_t RTCS_request_DHCP_inform(
        _rtcs_if_handle        handle,
        unsigned char              *optptr,
        uint32_t                optlen,
        _ip_address            client_addr,
        _ip_address            server_addr,
        void                   (_CODE_PTR_ inform_func)(uchar _PTR_,
                          uint32_t, _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)

7.1.136 RTCS_selectall()

If option RTCSCFG_SOCKET_OWNERSHIP is enabled then this function waits for activity on any socket that caller owns. Otherwise, it waits for activity on any socket.

Synopsis

```
uint32_t  RTCS_selectall(
          uint32_t  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 request is initiated by remote endpoint or all sent data are acknowledged.

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.

```
int32_t                servsock;
int32_t                connsock;
int32_t                status;
SOCKET_ADDRESS_STRUCT  addrpeer;
uint16_t               addrlen;
char                   buf[500];
int32_t                count;
uint32_t               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);
  }
}
```

7.1.137 RTCS_selectset()

Waits for activity on any socket in the set of sockets.

Synopsis

```
uint32_t  RTCS_selectset(
          void   *sockset,
          uint32_t  count,
          uint32_t  timeout)
```

Parameters

sockset [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.

```
int32_t       socklist[3];
sockaddr_in  local_sin;
uint32_t      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], (struct sockaddr *)&local_sin, sizeof (sockaddr_in));

local_sin.sin_port = 2011;
socklist[1] = socket(AF_INET, SOCK_DGRAM, 0);
```

```
result = bind(socklist[1], (struct sockaddr *)&local_sin, sizeof (sockaddr_in));

local_sin.sin_port = 2012;
socklist[2] = socket(AF_INET, SOCK_DGRAM, 0);
result = bind(socklist[2], (struct sockaddr *)&local_sin, sizeof (sockaddr_in));

while (TRUE) {
  sock = RTCS_selectset(socklist, 3, 0);

  rlen = sizeof(raddr);
  length = recvfrom(sock, buffer, BUFFER_SIZE, 0, (struct sockaddr *)&raddr, &rlen);
  sendto(sock, buffer, length, 0, (struct sockaddr *)&raddr, rlen);
}
```

7.1.138 RTCSLOG_disable()

Disables RTCS logging.

Synopsis

```
void RTCSLOG_disable(
        uint32_t   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()**.

7.1.139 RTCSLOG_enable()

Enables RTCS logging.

Synopsis

```
void RTCSLOG_enable(
         uint32_t   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™ RTOS Reference Manual*
- **_klog_control()** in *MQX™ RTOS 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);
```

7.1.140 RTCS6_if_bind_addr()

Binds the IPv6 address to the device interface.

Synopsis

```
uint32_t RTCS6_if_bind_addr (_rtcs_if_handle rtcs_if_handle, in6_addr *address,
rtcs6_if_addr_type address_type)
```

Parameters

rtcs_if_handle [in] — RTCS interface handle.

address [in] — IPv6 address for the device interface.

address_type [in] — IPv6 address type. It defines the way the IPv6 address to be assigned to the interface:

- IP6_ADDR_TYPE_MANUAL – the value of the *address* parameter defines the whole IPv6 address to be bind to the interface.
- IP6_ADDR_TYPE_AUTOCONFIGURABLE – the value of the *address* parameter defines the first 64bits of the bind IPv6 address. The last 64bits of the IPv6 address are overwritten with the Interface Identifier. In case of Ethernet interface, the Interface Identifier is formed from 48-bit MAC address, according to [RFC2464].

Description

Function **RTCS6_if_bind_addr()** binds IPv6 address *address* to the device interface associated with handle *rtcs_if_handle*. Parameter *rtcs_if_handle* is returned by **RTCS_if_add()**.

One interface may have several bound IPv6 addresses.

Return Value

- *RTCS_OK* (success)
- Error code (failure)

See Also

- **RTCS6_if_unbind_addr()**
- **ip6_if_addr_type**

Example

```
/* Bind 1:203:405:607:809:a0b:c0d:e0f IPv6 address.*/
{
    /* Before, interface was initialized by ipcfg_init_device().*/
    _rtcs_if_handle ihandle = ipcfg_get_ihandle(0);
    /* Bind 1:203:405:607:809:a0b:c0d:e0f IPv6 address.*/
    in6_addr        address = IN6ADDR(0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7,
                                  0x8,0x9,0xa,0xb,0xc,0xd,0xe,0xf);
    uint32_t         error;

    if(ihandle)
    {
        error = RTCS6_if_bind_addr(ihandle, &address, IP6_ADDR_TYPE_MANUAL);
        if (error == RTCS_OK)
            printf("The interface is bound.\n");
```

```
            else
                printf("Failed to bind interface, error = %x\n", error);
        }
        else
            printf("Not initialized by ipcfg_init_device().\n");
    }
```

7.1.141 RTCS6_if_unbind_addr()

Unbinds the IPv6 address from the device interface.

Synopsis

```
uint32_t RTCS6_if_unbind_addr (_rtcs_if_handle  rtcs_if_handle, in6_addr *address)
```

Parameters

- *rtcs_if_handle [in]* — RTCS interface handle.
- *address [in]* — IPv6 address to unbind.

Description

Function **RTCS6_if_unbind_addr()** unbinds IPv6 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

- RTCS6_if_bind_addr()

Example

```
    /* Unbind 1:203:405:607:809:a0b:c0d:e0f IPv6 address.*/
    {
        uint32_t         error;
        /* Before, interface was initialized by ipcfg_init_device().*/
        _rtcs_if_handle ihandle = ipcfg_get_ihandle(0);
        /* 1:203:405:607:809:a0b:c0d:e0f IPv6 address.*/
        in6_addr        address = IN6ADDR(0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7,
                                          0x8,0x9,0xa,0xb,0xc,0xd,0xe,0xf);

        if(ihandle)
        {
            error = RTCS6_if_bind_addr(ihandle, &address, IP6_ADDR_TYPE_MANUAL);
            if (error == RTCS_OK)
            {
                printf("The interface is bound.\n");

                error =  RTCS6_if_unbind_addr (ihandle, &address);

                if (error == RTCS_OK)
                    printf("The interface is unbound.\n");
                else
                    printf("Failed to unbind interface, error = %x\n", error);
```

```
        }
        else
            printf("Failed to bind interface, error = %x\n", error);
    }
    else
        printf("Not initialized by ipcfg_init_device().\n");
}
```

7.1.142 RTCS6_if_get_scope_id()

Returns the Scope ID assigned to the device interface.

Synopsis

```
uint32_t RTCS6_if_get_scope_id (_rtcs_if_handle rtcs_if_handle)
```

Parameters

- *rtcs_if_handle [in]* — RTCS interface handle.

Description

This function returns Scope ID (interface identifier) assigned to the device interface associated with *rtcs_if_handle*. The Scope ID is used to indicate the network interface over which traffic is sent and received.

Return Value

- *Scope ID* (success)
- 0 (failure)

See Also

- **RTCS6_if_bind_addr()**

Example

```
/* Get Scope ID assigned to the interface.*/
{
    uint32_t         scope_id;
    /* Before, interface was initialized by ipcfg_init_device().*/
    _rtcs_if_handle ihandle = ipcfg_get_ihandle(0);

    if(ihandle)
    {
        scope_id = RTCS6_if_get_scope_id(ihandle);
        if(scope_id == 0)
            printf("Scope ID is not assigned to the interface.\n");
        else
            printf("Scope ID = %x\n", scope_id);
    }
    else
        printf("Not initialized by ipcfg_init_device().\n");
}
```

7.1.143 RTCS6_if_get_addr()

Returns an IPv6 address information bound to the device interface.

Synopsis

```
uint32_t RTCS6_if_get_addr(_rtcs_if_handle ihandle, uint32_t n, RTCS6_IF_ADDR_INFO *addr_info)
```

Parameters

- *rtcs_if_handle [in]* — RTCS interface handle.
- *n [in]* — *s*equence number of IPv6 address to retrieve (from 0).
- *addr_info [in/out]* — pointer to IPv6 address information (IPv6 address, address state and type).

Description

This function returns the IPv6 address information bound to the given device interface.

One interface may have several bound IPv6 addresses.

Return Value

- *RTCS_OK* (success, *addr_info* is filled)
- *RTCS_ERROR* (failure, *n*-th address is not available)

See Also

- RTCS6_if_bind_addr()
- RTCS6_IF_ADDR_INFO

Example

```
/* Print all bound IPv6 addresses.*/
{
    /* Before, interface was initialized by ipcfg_init_device().*/
    _rtcs_if_handle ihandle = ipcfg_get_ihandle(0);
    char prn_addr6[RTCS_IP6_ADDR_STR_SIZE];

    if(ihandle)
    {
        RTCS6_IF_ADDR_INFO  addr_info;
        int                 n=0;

        /* Print all bound IPv6 addresses.*/
        while(RTCS6_if_get_addr(ihandle, n, &addr_info) == RTCS_OK)
        {
            /* Convert IPv6 address to string presentation and print it.*/
            if(inet_ntop(AF_INET6, &addr_info.ip_addr, prn_addr6, sizeof(prn_addr6)))
             {
                 printf("IP6[%d] : %s\n", n, prn_addr6);
            }
            n++;
        }
    }
    else
        printf("Not initialized by ipcfg_init_device().\n");
}
```

7.1.144 RTCS6_if_get_dns_addr ()

Returns the n-th DNS IPv6 address from the registered DNS list of the device interface.

Synopsis

```
bool RTCS6_if_get_dns_addr(_rtcs_if_handle ihandle, uint32_t n, in6_addr *dns_addr)
```

Parameters

- *ihandle [in]* — RTCS interface handle
- *n [in]* — DNS IPv6 address index (from 0)
- *dns_addr [in/out]* — pointer to DNS IPv6 address

Description

This function may be used to retrieve all DNS IPv6 addresses registered (manually or by IPv6 router discovery process) with the given device interface.

Return Value

- *RTCS_OK* (success, *addr_info* is filled)
- *RTCS_ERROR* (failure, *n*-th address is not available)

See Also

- RTCS6_if_add_dns_addr ()
- RTCS6_if_del_dns_addr ()

Example

```
/* Print all DNS IPv6 addresses.*/
    {
        _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
        char            addr_str[RTCS_IP6_ADDR_STR_SIZE];
        int             i;
        in6_addr        dns6_addr;

        for(i=0; (RTCS6_if_get_dns_addr(ihandle, i, &dns6_addr) == TRUE); i++)
        {
            printf ("%d: %s\n", i + 1, inet_ntop(AF_INET6, &dns6_addr, addr_str,
sizeof(addr_str)));
        }
          }
```

7.1.145 RTCS6_if_add_dns_addr ()

Registers the DNS IPv6 address with the device interface.

Synopsis

```
uint32_t RTCS6_if_add_dns_addr(_rtcs_if_handle ihandle, in6_addr *dns_addr)
```

Parameters

- *ihandle [in]* — RTCS interface handle.

- *dns_addr [in]* — pointer to the DNS IPv6 address to add.

Description

This function adds the DNS IPv6 address to the list assigned to given device interface.

Return Value

- *RTCS_OK* (success, *addr_info* is filled)
- *RTCS_ERROR* (failure, *n*-th address is not available)

See Also

- RTCS6_if_get_dns_addr ()
- RTCS6_if_del_dns_addr ()

Example

```
/* Register DNS IPv6 address with the device interfae.*/
    {
        _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
        char            *addr_str = "2001:470:1234:567:4c39:64fa:1caa:44c8";
        in6_addr        dns6_addr;

        if(inet_pton(AF_INET6, addr_str, &dns6_addr, sizeof(dns6_addr)) == RTCS_OK)
        {
            if(RTCS6_if_add_dns_addr(ihandle, &dns6_addr) == RTCS_OK)
            {
                printf("Adding DNS address is successful.\n");
            }
            else
            {
                printf("Adding DNS address is failed.\n");
            }
        }
          }
```

7.1.146 RTCS6_if_del_dns_addr ()

Unregisters the DNS IPv6 address from the device interface.

Synopsis

```
uint32_t RTCS6_if_del_dns_addr(_rtcs_if_handle ihandle, in6_addr *dns_addr)
```

Parameters

- *ihandle [in]* — RTCS interface handle.
- *dns_addr [in]* — DNS IPv6 address to be removed.

Description

This function removes the DNS IPv6 address from the list assigned to given device interface.

Return Value

- *RTCS_OK* (success)
- Error code (failure)

See Also

- RTCS6_if_get_dns_addr ()
- RTCS6_if_add_dns_addr ()

Example

```
/* Unregister DNS IPv6 address from the device interface.*/
    {
        _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
        char            *addr_str = "2001:470:1234:567:4c39:64fa:1caa:44c8";
        in6_addr        dns6_addr;

        if(inet_pton(AF_INET6, addr_str, &dns6_addr, sizeof(dns6_addr)) == RTCS_OK)
        {
            if(RTCS6_if_del_dns_addr(ihandle, &dns6_addr) == RTCS_OK)
            {
                printf("Deleting DNS address is successful.\n");
            }
            else
            {
                printf("Deleting DNS address is failed.\n");
            }
        }
    }
```

7.1.147 send()

Sends data on the stream socket, or on a datagram socket, for which a remote endpoint has been specified.

Synopsis

```
int32_t  send(
         uint32_t      socket,
         char * buffer,
         uint32_t      buflen,
         uint32_t      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 , ?$paratext>,” below. For stream sockets, flags should be zero.

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

```
uint32_t  handle;
char     buffer[20000];
uint32_t  count;

...

count = send(handle, buffer, 20000, 0);
if (count == RTCS_ERROR)
  printf("\nError, send() failed with error code %lx",
```

```
            RTCS_geterror(handle));
```

7.1.148 sendto()

Sends data on the datagram socket.

Synopsis

```
int32_t sendto(
          uint32_t              socket,
          char        *        buffer,
          uint32_t              buflen,
          uint16_t                flags,
          sockaddr     *      destaddr,
          uint16_t              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 , ?$paratext>."

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()

Examples

a) Send 500 bytes of data to IP address 192.203.0.54, port number 678.

```
uint32_t      handle;
sockaddr_in  remote_sin;
uint32_t      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, (struct sockaddr *)&remote_sin,
         sizeof(sockaddr_in));
if (count != 500)
   printf("\nsendto() failed with count %ld and error %lx",
          count, RTCS_geterror(handle));
```

b) Send "Hello, world!" to FE80::2e0:4cFF:FE68:2343 , port 7007 using IPv6 UDP protocol.

```
uint32_t socket_udp;
struct addrinfo     *foreign_addrv6_res /* pointer to PC IPv6 address */
struct addrinfo     *local_addrv6_res;  /* pointer to Board IPv6 address */
struct addrinfo      hints;             /* hints used for getaddrinfo() */

hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_DGRAM;
hints.ai_flags = AI_NUMERICHOST|AI_CANONNAME;
getaddrinfo ( "FE80::0200:5EFF:FEA8:0016%2", "7007", &hints, &local_addrv6_res);
hints.ai_family   = AF_UNSPEC;
hints.ai_socktype = SOCK_DGRAM;
hints.ai_flags    = AI_NUMERICHOST|AI_CANONNAME;
getaddrinfo ( "FE80::2e0:4cFF:FE68:2343","7007", &hints,&foreign_addrv6_res);
socket_udp = socket(AF_INET6, SOCK_DGRAM, 0);
error = bind(socket_udp, (sockaddr *)(local_addrv6_res->ai_addr), sizeof(struct
sockaddr_in6));
sendto(socket_udp,"Hello,world!", 13, 0, (sockaddr*)(foreign_addrv6_res->ai_addr),
sizeof(sockaddr_in6));
```

7.1.149 setsockopt()

Sets the value of the socket option.

Synopsis

```
uint32_t  setsockopt(
          uint32_t   socket,
          uint32_t   level,
          uint32_t   optname,
          void      *optval,
          uint32_t   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

SOL_IP6

optname [in] — Option name (see Section , ?$paratext>”).

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.

NOTE	Some options can be temporarily overridden for datagram sockets. For more information, see **send()** and **sendto()**.

Options

This section describes the socket options.

Checksum Bypass

Option name	*OPT_CHECKSUM_BYPASS* (can be overridden)
Protocol level	*SOL_UDP*
Values	• TRUE (RTCS sets the checksum field of sent datagram packets to zero, and the generation of checksums is bypassed). • FALSE (RTCS generates checksums for sent datagram packets).
Default value	FALSE
Change	Before bound
Socket type	Datagram
Comments	—

Connect Timeout

Option name	*OPT_CONNECT_TIMEOUT*
Protocol level	*SOL_TCP*
Values	≥ 180,000 (RTCS maintains the connection for this number of milliseconds).
Default value	480,000 (eight minutes).

Change	Before bound
Socket type	Stream
Comments	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

Option name	*OPT_RECEIVE_NOWAIT*
Protocol level	*SOL_UDP*
Values	• TRUE (**recv()** and **recvfrom()** return immediately, regardless of whether data to be received is present). • FALSE (**recv()** and **recvfrom()** wait until data to be received is present).
Default value	FALSE
Change	Anytime
Socket type	Datagram
Comments	—

IGMP Add Membership

Option name	*RTCS_SO_IGMP_ADD_MEMBERSHIP*
Protocol level	*SOL_IGMP*
Values	—
Default value	Not in a group
Change	Anytime
Socket type	Datagram

Comments	IGMP must be in the RTCS protocol table.
	To join a multicast group:
	`uint32_t          sock;`
	`struct  ip_mreq  group;`
	
	`group.imr_multiaddr.s_addr =  multicast_ip_address;`
	`group.imr_interface.s_addr =  local_ip_address;`
	`error = setsockopt(sock, SOL_IGMP,`
	`   RTCS_SO_IGMP_ADD_MEMBERSHIP, &group,`
	`   sizeof(group));`

IGMP Drop Membership

Option name	*RTCS_SO_IGMP_DROP_MEMBERSHIP*
Protocol level	*SOL_IGMP*
Values	—
Default value	Not in a group
Change	After the socket is created
Socket type	Datagram

Comments	IGMP must be in the RTCS protocol table.
	To leave a multicast group:
	`uint32_t          sock;`
	`struct  ip_mreq  group;`
	
	`group.imr_multiaddr.s_addr = multicast_ip_address;`
	`group.imr_interface.s_addr = local_ip_address;`
	`error = setsockopt(sock, SOL_IGMP,`
	`  RTCS_SO_IGMP_DROP_MEMBERSHIP, &group,`
	`  sizeof(group));`

IGMP Get Membership

Option name	*RTCS_SO_IGMP_GET_MEMBERSHIP*
Protocol level	SOL_IGMP
Values	—
Default value	Not in a group
Change	— (use with **getsockopt()** only; returns value in *optval*).
Socket type	Datagram
Comments	—

Initial Retransmission Timeout

Option name	*OPT_RETRANSMISSION_TIMEOUT*
Protocol level	*SOL_TCP*
Values	≥ 15 ms (see comments)
Default value	3000 (three seconds)
Change	Before bound
Socket type	Stream
Comments	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

Option name	*OPT_KEEPALIVE*
Protocol level	*SOL_TCP*
Values	• Zero (RTCS does not probe the remote endpoint). • Non-zero (if the connection is idle, RTCS periodically probes the remote endpoint, an action that detects, whether the remote endpoint is still present).
Default value	Zero minutes
Change	Before bound
Socket type	Stream
Comments	The option is not a standard feature of the TCP/IP specification and generates unnecessary periodic network traffic.

Maximum Retransmission Timeout

Option name	*OPT_MAXRTO*
Protocol level	*SOL_TCP*
Values	• Non-zero (maximum value for the retransmission timer's exponential backoff). • 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)
Default value	Zero milliseconds
Change	Before bound
Socket type	Stream
Comments	The retransmission timer is used for multiple retransmissions of a segment.

NAT Inactivity Timeout

Option name	*RTCS_SO_NAT_TIMEOUTS*
Protocol level	*SOL_NAT*
Values	See comments
Default value	See comments
Change	After the socket is created
Socket type	Datagram or stream
Comments	An application-supplied *nat_timeouts* structure defines inactivity timeout values.

NAT Port Numbers

Option name	*RTCS_SO_NAT_PORTS*
Protocol level	*SOL_NAT*
Values	See comments
Default value	See comments
Change	After the socket is created
Socket type	Datagram or stream
Comments	An application-supplied *nat_ports* structure defines port numbers.

No Nagle Algorithm

Option name	*OPT_NO_NAGLE_ALGORITHM*
Protocol level	*SOL_TCP*
Values	• TRUE (RTCS does not use the Nagle algorithm to coalesce short segments). • FALSE (to reduce network congestion, RTCS uses the Nagle algorithm [defined in RFC 896] to coalesce short segments).
Default value	FALSE
Change	Before bound
Socket type	Stream
Comments	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	*RTCS_SO_LINK_RX_8021Q_PRIO*
Protocol level	*SOL_LINK*
Values	• –1 (last received frame did not have an Ethernet 802.1Q priority tag). • 0..7 (last received frame had an Ethernet 802.1Q priority tag with the specified priority).
Default value	—
Change	— (use with **getsockopt()** only; returns value in *optval*).
Socket type	Stream (Ethernet)
Comments	Returned information is for the last frame that the socket received.

Receive Ethernet 802.3 Frames

Option name	*RTCS_SO_LINK_RX_8023*
Protocol level	*SOL_LINK*
Values	• TRUE (last received frame was an 802.3 frame). • FALSE (last received frame was an Ethernet II frame).
Default value	—
Change	— (use with **getsockopt()** only; returns value in *optval*)
Socket type	Stream (Ethernet)
Comments	Returned information is for the last frame that the socket received.

Receive Nowait

Option name	*OPT_RECEIVE_NOWAIT*
Protocol level	*SOL_TCP*
Values	• TRUE (**recv()** returns immediately, regardless of whether there is data to be received). • FALSE (**recv()** waits until there is data to be received).
Default value	FALSE
Change	Anytime
Socket type	Stream
Comments	—

Receive Push

Option name	*OPT_RECEIVE_PUSH*
Protocol level	*SOL_TCP*
Values	• TRUE (**recv()** returns immediately if it receives a push flag from the remote endpoint, even if the specified receive buffer is not full). • FALSE (**recv()** ignores push flags and returns only when its buffer is full, or if the receive timeout expires).
Default value	TRUE
Change	Anytime
Socket type	Stream
Comments	—

Receive Timeout

Option name	*OPT_RECEIVE_TIMEOUT*
Protocol level	*SOL_TCP*
Values	• Zero (RTCS waits indefinitely for incoming data during a call to **recv()**). • Non-zero (RTCS waits for this number of milliseconds for incoming data during a call to **recv()**).
Default value	Zero milliseconds
Change	Anytime
Socket type	Stream
Comments	When the timeout expires, **recv()** returns with whatever data that has been received.

Receive-Buffer Size

Option name	*OPT_RBSIZE*
Protocol level	*SOL_TCP*
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

Option name	*RTCS_SO_LINK_TX_8021Q_PRIO*
Protocol level	*SOL_LINK*
Values	• –1 (RTCS does not include Ethernet 802.1Q priority tags) • 0..7 (RTCS includes Ethernet 802.1Q priority tags with the specified priority)
Default value	–1
Change	Anytime
Socket type	Stream (Ethernet)
Comments	—

Send Ethernet 802.3 Frames

Option name	*RTCS_SO_LINK_TX_8023*
Protocol level	*SOL_LINK*
Values	• TRUE (RTCS sends 802.3 frames). • FALSE (RTCS sends Ethernet II frames).
Default value	FALSE
Change	Anytime
Socket type	Stream (Ethernet)
Comments	Returns information for the last frame that the socket received.

Send Nowait (Datagram Socket)

Option name	*OPT_SEND_NOWAIT* (can be overridden)
Protocol level	*SOL_UDP*
Values	• TRUE (RTCS buffers every datagram and **send()** or **sendto()** returns immediately). • FALSE (task that calls **send()** or **sendto()** blocks until the datagram has been transmitted; datagrams are not copied).
Default value	FALSE
Change	Anytime
Socket type	Datagram
Comments	—

Send Nowait (Stream Socket)

Option name	*OPT_SEND_NOWAIT*
Protocol level	*SOL_TCP*
Values	• TRUE (task that calls **send()** does not wait if data is waiting to be sent; RTCS buffers the outgoing data, and **send()** returns immediately). • FALSE (task that calls **send()** waits if data is waiting to be sent).
Default value	FALSE
Change	Anytime
Socket type	Stream
Comments	—

Send Push

Option name	*OPT_SEND_PUSH*
Protocol level	*SOL_TCP*
Values	• TRUE (if possible, RTCS appends a send-push flag to the last packet in the segment of the data that is associated with **send()** and immediately sends the data. A call to **send()** might block until another task calls **send()** for that socket). • FALSE (before it sends a packet, RTCS waits until it has received enough data from the host to completely fill the packet).
Default value	TRUE
Change	Anytime
Socket type	Stream
Comments	—

Send Timeout

Option name	*OPT_SEND_TIMEOUT*
Protocol level	*SOL_TCP*
Values	• Zero (RTCS waits indefinitely for outgoing data during a call to **send()**). • Non-zero (RTCS waits for this number of milliseconds for incoming data during a call to **send()**).
Default value	Four minutes
Change	Anytime
Socket type	Stream
Comments	When the timeout expires, **send()** returns

Send-Buffer Size

Option name	*OPT_TBSIZE*
Protocol level	*SOL_TCP*
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 send buffer of the specified number of bytes, which controls how much sent data RTCS can buffer for the socket.

Socket Error

Option name	*OPT_SOCKET_ERROR*
Protocol level	*SOL_SOCKET*
Values	—
Default value	—
Change	— (use with **getsockopt()** only; returns value in *optval*)
Socket type	Datagram or stream
Comments	Returns the last error for the socket.

Socket Type

Option name	*OPT_SOCKET_TYPE*
Protocol level	*SOL_SOCKET*
Values	—
Default value	—
Change	— (use with **getsockopt()** only; returns value in *optval*)
Socket type	Datagram or stream
Comments	Returns the type of socket (*SOCK_DGRAM* or *SOCK_STREAM*).

Timewait Timeout

Option name	*OPT_TIMEWAIT_TIMEOUT*
Protocol level	*SOL_TCP*
Values	> Zero milliseconds
Default value	Two times the maximum segment lifetime (which is a constant).
Change	Before bound
Socket type	Stream
Comments	Returned information is for the last frame that the socket received.

RX Destination Address

Option name	RTCS_SO_IP_RX_DEST
Protocol level	*SOL_IP*
Values	—
Default value	—
Change	— (use with **getsockopt()** only; returns value in *optval*).
Socket type	Datagram or stream
Comments	Returns destination address of the last frame that the socket received.

Time to Live - RX

Option name	RTCS_SO_IP_RX_TTL
Protocol level	*SOL_IP*
Values	—
Default value	—
Change	— (use with **getsockopt()** only; returns value in *optval*).
Socket type	Datagram or stream
Comments	Gets the TTL (time to live) field of incoming packets. Returned information is for the last frame that the socket received.

Type of Service - RX

Option name	RTCS_SO_IP_RX_TOS
Protocol level	*SOL_IP*
Values	—
Default value	—

Change	— (use with **getsockopt()** only; returns value in *optval*).
Socket type	Datagram or stream
Comments	Returns the TOS (type of service) field of incoming packets. Returned information is for the last frame that the socket received.

Type of Service - TX

Option name	RTCS_SO_IP_TX_TOS
Protocol level	*SOL_IP*
Values	uchar
Default value	0
Change	Anytime
Socket type	Datagram or stream
Comments	Sets or gets the IPv4 TOS (type of service) field of outgoing packets.

Time to Live - TX

Option name	RTCS_SO_IP_TX_TTL
Protocol level	*SOL_IP*
Values	TTL field of the IP header in outgoing datagrams
Default value	64
Change	Anytime
Socket type	Datagram or stream
Comments	Sets or gets the TTL (time to live) field of outgoing packets.

Local Address

Option name	RTCS_SO_IP_LOCAL_ADDR
Protocol level	*SOL_IP*
Values	—
Default value	—
Change	— (use with **getsockopt()** only; returns value in *optval*).
Socket type	Datagram or stream
Comments	Returns local IP address.

IPv6 hop limit for outgoing unicast packets

Option name	RTCS_SO_IP6_UNICAST_HOPS
Protocol level	SOL_IP6
Values	0-255
Default value	0
Change	Anytime
Socket type	Datagram or stream
Comments	This option defines the hop limit to use for outgoing unicast IPv6 packets. By default the option value is set to zero. It means that the hop limit is suggested by a local IPv6 router, otherwise the hop limit equals to 64.

IPv6 hop limit for outgoing multicast packets

Option name	RTCS_SO_IP6_MULTICAST_HOPS
Protocol level	SOL_IP6
Values	0-255
Default value	1
Change	Anytime
Socket type	Datagram
Comments	This option defines the hop limit to use for outgoing multicast IPv6 packets. If it set to zero, the hop limit is suggested by a local IPv6 router, otherwise the hop limit equals to 64.

IPv6 Add Membership

Option name	RTCS_SO_IP6_JOIN_GROUP
Protocol level	SOL_IP6
Values	ipv6_mreq
Default value	—
Change	Anytime
Socket type	Datagram

Comments	• Multicast Listener Discovery (MLDv1) Protocol can be enabled by the RTCSCFG_ENABLE_MLD configuration parameter. Its enabling is optional for multicast traffic that takes place inside only one local network.
	• Maximum number of IPv6 multicast memberships, that may exist at the same time per one socket, is defined by the RTCSCFG_IP6_MULTICAST_SOCKET_MAX configuration parameter.
	• Maximum number of unique IPv6 multicast memberships, that may exist at the same time in the whole system, is defined by the RTCSCFG_IP6_MULTICAST_ MAX configuration parameter.
	
	To join an IPv6 multicast group:
	`int sock;`
	`struct ipv6_mreq group;`
	`...`
	`IN6_ADDR_COPY(&<multicast_ip_address>, & group.ipv6imr_multiaddr);`
	`group.ipv6imr_interface = 0; /* Chosen by stack.*/`
	
	`<error> = setsockopt(sock, SOL_IP6,`
	`RTCS_SO_IP6_JOIN_GROUP, &group,`
	`sizeof(group));`

IPv6 Drop Membership

Option name	RTCS_SO_IP6_LEAVE_GROUP
Protocol level	SOL_IP6
Values	ipv6_mreq
Default value	—
Change	Anytime
Socket type	Datagram

Comments	To leave an IPv6 multicast group: int sock; struct ipv6_mreq group; ... IN6_ADDR_COPY(&<*multicast_ip_address*>, & group.ipv6imr_multiaddr); group.ipv6imr_interface = *0*; /* Chosen by stack.*/ <error> = setsockopt(sock, SOL_IP6, RTCS_SO_IP6_LEAVE_GROUP, &group, sizeof(group));

Examples

Example 7-1. Changing the Send-Push Option to *FALSE*

```
uint32_t  handle;
uint32_t  opt_length = sizeof(uint32_t);
uint32_t  opt_value = FALSE;
uint32_t  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, (uint32_t* *)&opt_length);
if (status != RTCS_OK)
  printf("\ngetsockopt() failed with error %lx", status);
```

Example 7-2. Changing the Receive-Nowait Option to *TRUE*

```
uint32_t  handle;
uint32_t  opt_length = sizeof(uint32_t);
uint32_t  opt_value = TRUE;
uint32_t  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*

```
uint32_t  handle;
uint32_t  opt_length = sizeof(uint32_t);
uint32_t  opt_value = TRUE;
uint32_t  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;
uint32_t      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;
uint32_t         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

```
uint32_t  handle;
uint32_t  status;
uint8_t   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);
```

7.1.150 shutdown()

Shuts down the socket.

Synopsis

```
uint32_t  shutdown(
          uint32_t  socket,
          uint16_t  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 *how*	Action
Datagram	Ignored	• Shuts down *socket* immediately. • Calls to **recvfrom()** return immediately. • Discards queued incoming packets.
Unconnected stream	Ignored	Shuts down *socket* immediately.
Connected stream	FLAG_CLOSE_TX	• Gracefully shuts down *socket*, ensuring that all sent data is acknowledged. • Calls to **send()** and **recv()** return immediately. • 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.
	FLAG_ABORT_CONNECTION	• Immediately discards the internal socket context. • Sends a TCP reset packet to the remote endpoint. • Calls to **send()** and **recv()** return immediately.

Return Value

- *RTCS_OK*
- Specific error code

See Also

- socket()

Example

```
uint32_t  handle;
uint32_t  status;
...
status = shutdown(handle, 0);
if (status != RTCS_OK)
  printf("\nError, shutdown() failed with error code %lx",
         status);
```

7.1.151 SMTP_send_email

Function for sending an email.

Synopsis

```
_mqx_int SMTP_send_email(
SMTP_PARAM_STRUCT_PTR   param,
char                *err_string,
uint32_t                 buffer_size)
```

Parameters

param [IN] – Pointer to a structure with all required parameters.

err_string[OUT] – Pointer to the user buffer for delivery/error message. This parameter can be *NULL* - no message is then returned.

buffer_size[IN] – Size in bytes of the parameter err_string.

Description

The *params* structure contains all required information for the SMTP client. This includes a SMTP envelope, the text of email, the server used for sending the email, the login and the password (only if an authentication is required).

Return value

- *SMTP_OK* – Email send successfully.
- *SMTP_ERR_BAD_PARAM* – Invalid values set in param structure.
- *SMTP_ERR_CONN_FAILED* – Connection to server failed.
- *SMTP_WRONG_RESPONSE* – Server returned wrong response to SMTP command.
- *MQX_OUT_OF_MEMORY* – Memory allocation failed for a key component of SMTP client.

Example

Please see file *\shell\source\rtcs\sh_smtp.c* for source code demonstrating usage of function SMTP_send_email.

7.1.152 SNMP_init()

Starts SNMP Agent.

Synopsis

```
uint32_t  SNMP_init(
          char   *name,
          uint32_t   priority
          uint32_t   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 by making 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

```
uint32_t  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");
```

7.1.153 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()**

7.1.154 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()

7.1.155 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()**

7.1.156 SNMP_trap_linkDown()

Synopsis

```
void SNMP_trap_linkDown(void *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()**

7.1.157 SNMP_trap_myLinkDown()

Synopsis

```
void SNMP_trap_myLinkDown(void *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()**

7.1.158 SNMP_trap_linkUp()

Synopsis

```
void SNMP_trap_linkUp(void *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()**

7.1.159 SNMP_trap_userSpec()

Synopsis

```
void SNMP_trap_userSpec(
        RTCSMIB_NODE_PTR trap_node,
        uint32_t 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()**

7.1.160 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()

7.1.161 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()

7.1.162 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()

7.1.163 SNMPv2_trap_linkDown()

Synopsis

```
void SNMPv2_trap_linkDown(void *ihandle)
```

Parameters

ihandle [in] — interface index

Description

This function sends link down trap type 2 message.

Return Value

See Also

- **SNMP_trap_linkDown()**

7.1.164 SNMPv2_trap_linkUp()

Synopsis

```
void SNMPv2_trap_linkUp(void *ihandle)
```

Parameters

ihandle [in] — interface index

Description

This function sends link up trap type 2 message.

Return Value

See Also

- **SNMP_trap_linkUp()**

7.1.165 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()

7.1.166 SNTP_init()

Starts the SNTP Client task.

Synopsis

```
uint32_t SNTP_init(
         char      *name,
         uint32_t       priority,
         uint32_t       stacksize,
         _ip_address   destination,
         uint32_t       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

```
uint32_t 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;
```

7.1.167 SNTP_oneshot()

Sets the time in UTC time using the SNTP protocol.

Synopsis

```
uint32_t SNTP_oneshot(
        _ip_address   destination,
        uint32_t      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()**

7.1.168 socket()

Creates the socket.

Synopsis

```
uint32_t  socket(
          uint16_t  protocol_family,
          uint16_t  type,
          uint16_t  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()**.

7.1.169 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()**
- **inet_pton()**
- **IPIF_stats()**
- **UDP_stats()**
- *TCP_STATS*

Example

See **ARP_stats()**.

7.1.170 TELNET_connect()

Starts Telnet Client, which starts the shell that accepts a command to start a Telnet session with a Telnet server.

Synopsis

```
uint32_t  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)

7.1.171 TELNETSRV_init()

Starts the Telnet Server.

Synopsis

```
uint32_t  TELNETSRV_init(
        char       *name,
        uint32_t        priority,
        uint32_t        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 by making 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 (void *dummy)
{
   Shell(Telnetd_shell_commands,NULL);
}

void main_task( uint32_t 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*

7.1.172 TFTPSRV_access()

Decides, whether to allow access to a TFTP client.

Synopsis

```
bool TFTPSRV_access(
        char   *string_ptr,
        uint16_t   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 both read and 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)

See Also

- **TFTPSRV_init()**

7.1.173 TFTPSRV_init()

Starts TFTP Server.

Synopsis

```
uint32_t TFTPSRV_init(
         char *name,
         uint32_t     priority,
         uint32_t     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 by making 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 *TFTPSRV_MAX_TRANSACTIONS*, whose default value (20) is defined in *tftp.h*.

Return Value

- *RTCS_OK* (success)
- RTCS error code (failure)

See Also

- **TFTPSRV_access()**

Example

```
uint32_t  error;

/* Start TFTP Server: */
error = TFTPSRV_init("TFTP server", 7, 1000);
if (error) return error;
printf("\nTFTP Server is running.");
return 0;
```

7.1.174 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**
- **inet_pton()**
- **IPIF_stats()**
- **TCP_stats()**
- *ARP_STATS*

Example

See **ARP_stats()**.

7.2 Functions Listed by Service

Table 7-2.

Service	Functions
DHCP Client	**RTCS_if_bind_DHCP()** **DHCPCLNT_find_option()**
DHCP Server	**DHCP*** **DHCPSRV***
DNS Resolver	**DNS_init()** **gethostbyaddr()** **gethostbyname()**
Echo Server	**ECHOSRV_init()**
EDS Server (Winsock)	**DNS_init()**
Ethernet Driver	**ENET_get_stats()** (part of MQX RTOS) **ENET_initialize()** (part of MQX RTOS)
FTP Client	**FTP_close()** **FTP_command()** **FTP_command_data()** **FTP_open()**
FTP Server	**FTPSRV_init()** FTPSRV_release()
HTTP Server	**HTTPSRV_init()** **HTTPSRV_release()** **HTTPSRV_cgi_read()** **HTTPSRV_cgi_write()** **HTTPSRV_ssi_write()**

Table 7-2. (continued)

IPCFG	**ipcfg_init_device()** **ipcfg_init_interface()** **ipcfg_bind_boot()** **ipcfg_bind_dhcp()** **ipcfg_bind_dhcp_wait()** **ipcfg_bind_staticip()** **ipcfg_get_device_number()** **ipcfg_add_interface()** **ipcfg_get_ihandle()** **ipcfg_get_mac()** **ipcfg_get_state()** **ipcfg_get_state_string()** **ipcfg_get_desired_state()** **ipcfg_get_link_active()** **ipcfg_get_dns_ip()** **ipcfg_add_dns_ip()** **ipcfg_del_dns_ip()** **ipcfg_get_ip()** **ipcfg_get_tftp_serveraddress()** **ipcfg_get_tftp_servername()** **ipcfg_get_boot_filename()** **ipcfg_poll_dhcp()** **ipcfg_task_create()** **ipcfg_task_destroy()** **ipcfg_task_status()** **ipcfg_task_poll()** **ipcfg_unbind()**
IWCFG	**iwcfg_set_essid()** **iwcfg_get_essid()** **iwcfg_commit()** **iwcfg_set_mode()** **iwcfg_get_mode()** **iwcfg_set_wep_key()** **iwcfg_get_wep_key()** **iwcfg_set_passphrase()** **iwcfg_get_passphrase()** **iwcfg_set_sec_type()** **iwcfg_get_sectype()** **iwcfg_set_power()** **iwcfg_set_scan()**
MIB	**MIB1213_init()**
NAT	**NAT_init()** **NAT_close()** **NAT_stats()**

Table 7-2. (continued)

PPP Driver	PPP_init() **IPIF_stats()** PPP_release() PPP_pause() PPP_resume()
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	**accept()** **bind()** **connect()** **getpeername()** **getsockname()** **getsockopt()** **listen()** **recv()** **recvfrom()** **RTCS_attachsock()** **RTCS_detachsock()** **RTCS_geterror()** **RTCS_selectall()** **RTCS_selectset()** **send()** **sendto()** **setsockopt()** **shutdown()** **socket()**
Statistics	**ARP_stats()** **ENET_get_stats()**(part of MQX RTOS) **ICMP_stats()** **IGMP_stats()** **inet_pton()** **IPIF_stats()** **NAT_stats()** **TCP_stats()** **UDP_stats()**
Telnet Client	**TELNET_connect()**
Telnet Server	**TELNETSRV_init()**
TFTP Server	**TFTPSRV_access()** **TFTPSRV_init()**

Chapter 8 Data Types

8.1 RTCS Data Types

RTCS data type	MQX data type	Defined in	Notes
_enet_address	**uchar** *[6]*	*enet.h*	In MQX source
_enet_handle	**void***	*enet.h*	In MQX source
_ip_address	**uint32_t**	*rtcs.h*	
_ppp_handle	**void***	*ppp.h*	
_pppoe_srv_handle	**void***	*pppoe.h*	
_task_id	**uint32_t**	*mqx.h*	In MQX source
bool_t	**bool**	*rpctypes.h*	
caddr_t	**char***	*rpctypes.h*	
enum_t	**uint16_t** or **uint32_t** (depends on the compiler)	*rpctypes.h*	
u_char	**uchar**	*rpctypes.h*	
u_int	**uint32_t**	*rpctypes.h*	
u_long	**uint32_t**	*rpctypes.h*	
u_short	**uint16_t**	*rpctypes.h*	

8.2 Alphabetical List of RTCS Data Structures

This section provides an alphabetical list of RTCS data structures with the following information:

- Function
- Definition
- Fields

8.2.1 addrinfo

This structure is used by the **getaddrinfo()** function.

```
typedef struct addrinfo {
        uint16_t        ai_flags;
        uint16_t        ai_family;
        uint32_t        ai_socktype;
        uint16_t        ai_protocol;
        unsigned int    ai_addrlen;
        char            *ai_canonname;
        struct sockaddr *ai_addr;
        struct addrinfo *ai_next;
    } addrinfo;
```

ai_flags

Flag field that is used by the *hints* parameter of getaddrinfo(), shall be set to zero or be the bitwise-inclusive OR of one or more of the values AI_CANONNAME, AI_NUMERICHOST and AI_PASSIVE:

- AI_CANONNAME: If the AI_CANONNAME bit is set, a successful call to getaddrinfo() will return a NUL-terminated string containing the canonical name of the specified hostname in the ai_canonname element of the addrinfo structure returned.
- AI_NUMERICHOST: If the AI_NUMERICHOST bit is set, it indicates that hostname should be treated as a numeric string defining an IPv4 or IPv6 address and no name resolution should be attempted.
- AI_PASSIVE: If the AI_PASSIVE bit is set it indicates that the returned socket address structure is intended for use in a call to bind(). In this case, if the hostname argument is the null pointer, then the IP address portion of the socket address structure will be set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address. If the AI_PASSIVE bit is not set, the returned socket address structure will be ready for use in a call to connect() for a connection-oriented protocol or connect(), sendto(), or sendmsg() if a connectionless protocol was chosen. The IP address portion of the socket address structure will be set to the loopback address if hostname is the null pointer and AI_PASSIVE is not set.

ai_family

The protocol family (AF_INET or AF_INET6).

ai_socktype

Socket type (SOCK_STREAM or SOCK_DGRAM).

ai_protocol

Protocol (IPPROTO_TCP or IPPROTO_UDP).

ai_addrlen

The length of the ai_addr member.

ai_canonname

The canonical name of the host.

ai_addr

Socket address.

ai_next

A pointer to the next *addrinfo* structure in the linked list.

8.2.2 ARP_STATS

A pointer to this structure is returned by **ARP_stats()**.

```
typedef struct {
  uint32_t            ST_RX_TOTAL;
  uint32_t            ST_RX_MISSED;
  uint32_t            ST_RX_DISCARDED;
  uint32_t            ST_RX_ERRORS;

  uint32_t            ST_TX_TOTAL;
  uint32_t            ST_TX_MISSED;
  uint32_t            ST_TX_DISCARDED;
  uint32_t            ST_TX_ERRORS;

  RTCS_ERROR_STRUCT  ERR_RX;
  RTCS_ERROR_STRUCT  ERR_TX;

  uint32_t            ST_RX_REQUESTS;
  uint32_t            ST_RX_REPLIES;

  uint32_t            ST_TX_REQUESTS;
  uint32_t            ST_TX_REPLIES;

  uint32_t            ST_ALLOCS_FAILED;
  uint32_t            ST_CACHE_HITS;
  uint32_t            ST_CACHE_MISSES;
  uint32_t            ST_PKT_DISCARDS;
} ARP_STATS, * 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.

8.2.3 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, * 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.

8.2.4 DHCP_DATA_STRUCT

A pointer to this structure in a parameter to **RTCS_if_bind_DHCP()**.

```
typedef struct {
  int32_t   (_CODE_PTR_  CHOICE_FUNC)(uchar *, uint32_t);
  void     (_CODE_PTR_  BIND_FUNC)  (uchar *, uint32_t,
                                     _rtcs_if_handle);
  bool  (_CODE_PTR_  UNBIND_FUNC)(_rtcs_if_handle);
} DHCP_DATA_STRUCT,  *  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 is 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.

8.2.5 DHCPSRV_DATA_STRUCT

A pointer to this structure is an input parameter to **DHCPSRV_ippool_add()**.

```
typedef struct dhcpsrv_data_struct {
  _ip_address  SERVERID;
  uint32_t      LEASE;
  _ip_address  MASK;
  _ip_address  SADDR;
  uchar        SNAME[64];
  uchar        FILE[128];
} DHCPSRV_DATA_STRUCT, * DHCPSRV_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.

8.2.6 ENET_STATS

A pointer to this structure is returned by **ENET_get_stats()**.

```
typedef struct {
  uint32_t  ST_RX_TOTAL;
  uint32_t  ST_RX_MISSED;
  uint32_t  ST_RX_DISCARDED;
  uint32_t  ST_RX_ERRORS;

  uint32_t  ST_TX_TOTAL;
  uint32_t  ST_TX_MISSED;
  uint32_t  ST_TX_DISCARDED;
  uint32_t  ST_TX_ERRORS;
  uint32_t  ST_TX_COLLHIST[16];

  uint32_t  ST_RX_ALIGN;
  uint32_t  ST_RX_FCS;
  uint32_t  ST_RX_RUNT;
  uint32_t  ST_RX_GIANT;
  uint32_t  ST_RX_LATECOLL;
  uint32_t  ST_RX_OVERRUN;

  uint32_t  ST_TX_SQE;
  uint32_t  ST_TX_DEFERRED;
  uint32_t  ST_TX_LATECOLL;
  uint32_t  ST_TX_EXCESSCOLL;
  uint32_t  ST_TX_CARRIER;
  uint32_t  ST_TX_UNDERRUN;
} ENET_STATS, * 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 the packet was a 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.

8.2.7 FTPSRV_AUTH_STRUCT

Structure defining authentication information about FTP server user.

```
typedef struct ftpsrv_auth_struct
{
    char* uid;
    char* pass;
    char* path;
}FTPSRV_AUTH_STRUCT;
```

uid

String for used identification. Usually username.

pass

Password for user.

path

Path to be set as FTP root directory after user logs in. If it is set to NULL, server root directory is used.

8.2.8 FTPSRV_PARAM_STRUCT

This structure is used as a parameter for the FTPSRV_init() function.

```
typedef struct ftpsrv_param_struct
{
    uint16_t                  af;
    unsigned short            port;
  #if RTCSCFG_ENABLE_IP4
    in_addr                   ipv4_address;
  #endif
  #if RTCSCFG_ENABLE_IP6
    in6_addr                  ipv6_address;
    uint32_t                  ipv6_scope_id;
  #endif
    _mqx_uint                 max_ses;
    bool                      use_nagle;
    uint32_t                  server_prio;
    const char*               root_dir;
    FTPSRV_AUTH_STRUCT*       auth_table;
} FTPSRV_PARAM_STRUCT;
```

af

Address family used by the server. Possible values are: AF_INET (use IPv4), AF_INET6 (use IPv6), AF_INET | AF_INET6 (use both IPv4 and IPv6).

port

Port to listen on. Default value is 21 as defined by RFC.

ipv4_address

IPv4 address to listen on. This variable is present only if the IPv4 is enabled. Default value is defined by macro FTPSRVCFG_DEF_ADDR.

ipv6_address

IPv6 address to listen on. This variable is present only if the IPv6 is enabled. Default value is in6addr_any.

ipv6_scope_id

Scope ID (interface identification) for IPv6. Default value is 0.

max_ses

Maximum number of users connected simultaneously to server. The default value is defined by the macro FTPSRVCFG_DEF_SES_CNT (2).

use_nagle

Set to TRUE to enable NAGLE algorithm for server sockets. Default in FALSE - NAGLE disabled.

server_prio

Priority of server tasks. All tasks created by the server (server task and session tasks) run with this priority.

The default value is defined by the macro FTPSRVCFG_DEF_SERVER_PRIO.

root_dir

Server root directory. Only files in this directory and its subdirectories are accessible for FTP clients.

auth_table

Array of users. Each user is one member of array, last element must be set to all NULLs as termination.

8.2.9 HOSTENT_STRUCT

A pointer to this structure is returned by the socket functions **gethostbyaddr()** and **gethostbyname()**.

```
typedef struct hostent
{
  char               *h_name;
  char   *          *h_aliases;
  int16_t             h_addrtype;
  int16_t            h_length;
  char  *           *h_addr_list;
} HOSTENT_STRUCT, * 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.

8.2.10 HTTPSRV_PARAM_STRUCT

This structure is used as a parameter for the HTTPSRV_init() function.

```
typedef struct httpsrv_param_struct
{
    unsigned short            port;
  #if RTCSCFG_ENABLE_IP4
    in_addr                   ipv4_address;
  #endif
  #if RTCSCFG_ENABLE_IP6
    in6_addr                  ipv6_address;
    uint32_t                   ipv6_scope_id;
  #endif
    _mqx_uint                 max_uri;
    _mqx_uint                 max_ses;
                             bool use_nagle;
    HTTPSRV_CGI_LINK_STRUCT*   cgi_lnk_tbl;
    HTTPSRV_SSI_LINK_STRUCT*   ssi_lnk_tbl;
   HTTPSRV_ALIAS*            alias_tbl;
    uint32_t                   server_prio;
    uint32_t                   script_prio;
    uint32_t                   script_stack;
    char*                     root_dir;
    char*                     index_page;
    HTTPSRV_AUTH_REALM_STRUCT* auth_table;
    uint16_t                   af;
} HTTPSRV_PARAM_STRUCT;
```

port

Port to listen on. Default value is defined by macro HTTPSRVCFG_DEF_PORT.

ipv4_address

IPv4 address to listen on. This variable is present only if the IPv4 is enabled. Default value is defined by macro HTTPSRVCFG_DEF_ADDR.

ipv6_address

IPv6 address to listen on. This variable is present only if the IPv6 is enabled. Default value is in6addr_any.

ipv6_scope_id

Scope ID (interface identification) for IPv6. Default value is 0.

max_uri

Maximum length of the URI requested by client in bytes. When URL exceeds this length, a response with code 414 (Request-URI Too Long) is sent to the client. The default value is defined by the macro HTTPSRVCFG_DEF_URL_LEN.

max_ses

Maximum number of sessions (connections) created by the server. The default value is defined by the macro HTTPSRVCFG_DEF_SES_CNT.

use_nagle

Set to TRUE to enable NAGLE algorithm for server sockets. Default in FALSE - NAGLE disabled.

cgi_lnk_tbl

Table of function names and pointers to functions used as CGI callbacks. The default is an empty table (NULL pointer).

ssi_lnk_tbl

Table of function names and pointers to functions used as SSI callbacks. The default is an empty table (NULL pointer).

alias_tbl

Table of directory aliases. Please see chapter 5.9.3, ?$paratext>” for description of alias functionality.

server_prio

Priority of server tasks. All tasks created by the server (server task and session tasks) run with this priority. The default value is defined by the macro HTTPSRVCFG_DEF_SERVER_PRIO.

script_prio

Priority of script handler tasks. This value should be either lower or the same as server_prio. The default value is defined by the macro HTTPSRVCFG_DEF_SERVER_PRIO.

script_stack

Size of a stack of the script handler task in bytes. Set the value of this variable according to the memory requirements of the CGI and SSI callbacks. The default value is 750 bytes.

root_dir

Root directory of the server. All files available to clients are stored in the path defined by this variable. The default value is “tfs:” (root set to trivial file system).

index_page

Default page sent to the client when the root directory is requested. The default value is defined by the macro HTTPSRVCFG_DEF_INDEX_PAGE.

auth_table

Table of authentication realms. The default is an empty table (NULL pointer).

af

Address family used by the server. Possible values are: AF_INET (use IPv4), AF_INET6 (use IPv6), AF_INET | AF_INET6 (use both IPv4 and IPv6).

8.2.11 HTTPSRV_AUTH_USER_STRUCT

Structure defining a user. Used for authentication purposes.

```
typedef struct httpsrv_auth_user_struct
{
    char* user_id;
    char* password;
}HTTPSRV_AUTH_USER_STRUCT;
```

user_id

User identificator (username etc.)

password

User password.

8.2.12 HTTPSRV_AUTH_REALM_STRUCT

Structure defining the authentication realm.

```
typedef struct httpsrv_auth_realm_struct
{
    char*                     name;
    char*                     path;
    HTTPSRV_AUTH_TYPE         auth_type;
    HTTPSRV_AUTH_USER_STRUCT* users;
} HTTPSRV_AUTH_REALM_STRUCT;
```

name

Name of the realm. This string is sent to the client as an identificator so that the user can determine the correct username and password.

path

Relative path to file or directory to be protected by authentication.

auth_type

Type of authentication. Value can be either HTTPSRV_AUTH_INVALID, HTTPSRV_AUTH_BASIC, or HTTPSRV_AUTH_DIGEST. Only the basic authentication is supported by the current server (v2.0).

users

Table of users who belong to a realm.

8.2.13 HTTPSRV_CGI_REQ_STRUCT

This structure is passed as a parameter to the user-defined CGI callback function and contains basic information about the connection, the client, and the server.

```
typedef struct httpsrv_cgi_request_struct
{
    uint32_t              ses_handle;
    HTTPSRV_REQ_METHOD   request_method;
    HTTPSRV_CONTENT_TYPE content_type;
    uint32_t              content_length;
    uint32_t              server_port;
    char*                remote_addr;
    char*                server_name;
    char*                script_name;
    char*                server_protocol;
    char*                server_software;
    char*                query_string;
    char*                gateway_interface;
    char*                remote_user;
    HTTPSRV_AUTH_TYPE    auth_type;
}HTTPSRV_CGI_REQ_STRUCT;
```

ses_handle

Handle to a session. This value is required as a parameter to read from and write to the server (sending a response to client).

request_method

Method used by a client in request. It can have any of values defined by enum HTTPSRV_REQ_METHOD. User callback must check if the request has a correct type before it can process it.

content_type

Content type of entity sent to the server from the client in request. It can have any of values defined by enum HTTPSRV_CONTENT_TYPE.

content_length

Length of a request entity in bytes.

server_port

Local port on which a connection from a client is established.

remote_addr

Remote (client’s) IP address. It can be either IPv4 or IPv6 address.

server_name

Server IP address or a host name. It can be either IPv4 or IPv6 address.

script_name

Name of the called CGI function. It is useful for a script self-identification.

server_protocol

Protocol used by the server to communicate with a client (HTTP/1.0).

server_software

String identifying the name and the version of the server software.

query_string

Part of requested URI after the question mark.

gateway_interface

Type and version of a common gateway interface (CGI/1.1).

remote_user

Username sent by the client as a part of the authentication process.

auth_type

Type of authentication used.

8.2.14 HTTPSRV_CGI_RES_STRUCT

Response structure generated by user CGI function. This structure is required as a parameter for the function **httpsrv_cgi_write()**. The entire structure must be filled by the user CGI callback.

```
typedef struct httpsrv_cgi_response_struct
{
    uint32_t              ses_handle;
    HTTPSRV_CONTENT_TYPE content_type;
    uint32_t              content_length;
    uint32_t              status_code;
    char*                data;
    uint32_t              data_length;
}HTTPSRV_CGI_RES_STRUCT;
```

ses_handle

Handle to a session used for CGI read/write operations.

content_type

Content type of the response generated by CGI.

content_length

Length of the response entity from CGI script.

status_code

HTTP response status code. A typical value is either 200 (response OK) or 404 (Not Found).

data

Pointer to the user data written as a response to the client.

data_length

Size of the user data in bytes.

8.2.15 HTTPSRV_SSI_PARAM_STRUCT

Parameter structure passed to the user SSI (server side include) callback.

```
typedef struct httpsrv_ssi_param_struct
{
    uint32_t ses_handle;
    char*   com_param;
}HTTPSRV_SSI_PARAM_STRUCT;
```

ses_handle

Handle to a session required for write operations from within SSI callback.

com_param

Parameter for the SSI command from the webpage (everything following the first comma character).

8.2.16 HTTPSRV_SSI_LINK_STRUCT

Structure defining a row of the SSI callback table.

```
typedef struct httpsrv_ssi_link_struct
{
    char* fn_name;
    HTTPSRV_SSI_CALLBACK_FN callback;
} HTTPSRV_SSI_LINK_STRUCT;
```

fn_name

Name/label of the function. When i.e. <%usbstat:test%> string is encountered during parsing *.shtlm of the *.shtm file, the function named “usbstat” is called with a parameter string set to “test”.

callback

Pointer to the function called when the string <%fn_name%> is found in the SSI file.

stack

Stack size for SSI. If set to zero, default script handler task will be used. Otherwise new independent task is created to process script with stack set to this value.

8.2.17 HTTPSRV_CGI_LINK_STRUCT

Structure defining a row of the CGI callback table.

```
typedef struct httpsrv_ssi_link_struct
{
    char* fn_name;
    HTTPSRV_SSI_CALLBACK_FN callback;
uint32_t stack;
} HTTPSRV_SSI_LINK_STRUCT;
```

fn_name

Name/label of the function. When i.e. rtcdata.cgi file is requested by the client, a function with a label “rtcdata” is called.

callback

Pointer to the function called when the filename fn_name.cgi is requested.

stack

Stack size for CGI. If set to zero, default script handler task will be used. Otherwise new independent task is created to process script with stack set to this value.

8.2.18 HTTPSRV_ALIAS

This structure is defining one item in server alias table.

```
typedef struct httpsrv_alias
{
    char* alias;
    char* path;
}HTTPSRV_ALIAS;
```

alias

User defined name for aliased path. This name is used as part of URI when accessing files.

path

Filesystem path to be aliased.

8.2.19 PING_PARAM_STRUCT

```
typedef struct ping_param_struct
{
    sockaddr             addr;
    uint32_t             timeout;
    uint16_t             id;
    uint8_t              hop_limit;
    void                *data_buffer;
    uint32_t             data_buffer_size;
    uint32_t             round_trip_time;
}PING_PARAM_STRUCT, * PING_PARAM_STRUCT_PTR;
```

8.2.20 ICMP_STATS

A pointer to this structure is returned by **ICMP_stats()**.

```
typedef struct {
  uint32_t           ST_RX_TOTAL;
  uint32_t           ST_RX_MISSED;
  uint32_t           ST_RX_DISCARDED;
  uint32_t           ST_RX_ERRORS;

  uint32_t           ST_TX_TOTAL;
  uint32_t           ST_TX_MISSED;
  uint32_t           ST_TX_DISCARDED;
  uint32_t           ST_TX_ERRORS;

  RTCS_ERROR_STRUCT  ERR_RX;
  RTCS_ERROR_STRUCT  ERR_TX;

  uint32_t           ST_RX_BAD_CODE;
  uint32_t           ST_RX_BAD_CHECKSUM;
  uint32_t           ST_RX_SMALL_DGRAM;
  uint32_t           ST_RX_RD_NOTGATE;

  uint32_t           ST_RX_DESTUNREACH;
  uint32_t           ST_RX_TIMEEXCEED;
  uint32_t           ST_RX_PARMPROB;
  uint32_t           ST_RX_SRCQUENCH;
  uint32_t           ST_RX_REDIRECT;
  uint32_t           ST_RX_ECHO_REQ;
  uint32_t           ST_RX_ECHO_REPLY;
  uint32_t           ST_RX_TIME_REQ;
  uint32_t           ST_RX_TIME_REPLY;
  uint32_t           ST_RX_INFO_REQ;
  uint32_t           ST_RX_INFO_REPLY;
  uint32_t           ST_RX_OTHER;

  uint32_t           ST_TX_DESTUNREACH;
  uint32_t           ST_TX_TIMEEXCEED;
  uint32_t           ST_TX_PARMPROB;
  uint32_t           ST_TX_SRCQUENCH;
  uint32_t           ST_TX_REDIRECT;
  uint32_t           ST_TX_ECHO_REQ;
  uint32_t           ST_TX_ECHO_REPLY;
  uint32_t           ST_TX_TIME_REQ;
  uint32_t           ST_TX_TIME_REPLY;
  uint32_t           ST_TX_INFO_REQ;
  uint32_t           ST_TX_INFO_REPLY;
  uint32_t           ST_TX_OTHER;
} ICMP_STATS, * ICMP_STATS_PTR;
```

8.2.20.0.1 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.

8.2.21 IGMP_STATS

A pointer to this structure is returned by **IGMP_stats()**.

```
typedef struct {

   uint32_t  ST_RX_TOTAL;
   uint32_t  ST_RX_MISSED;
   uint32_t  ST_RX_DISCARDED;
   uint32_t  ST_RX_ERRORS;

   uint32_t  ST_TX_TOTAL;
   uint32_t  ST_TX_MISSED;
   uint32_t  ST_TX_DISCARDED;
   uint32_t  ST_TX_ERRORS;

   RTCS_ERROR_STRUCT ERR_RX;
   RTCS_ERROR_STRUCT ERR_TX;

   uint32_t  ST_RX_BAD_TYPE;
   uint32_t  ST_RX_BAD_CHECKSUM;
   uint32_t  ST_RX_SMALL_DGRAM;
   uint32_t  ST_RX_QUERY;
   uint32_t  ST_RX_REPORT;

   uint32_t  ST_TX_QUERY;
   uint32_t  ST_TX_REPORT;

} IGMP_STATS, * 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.

8.2.22 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.

8.2.23 ip_mreq

IPv4 multicast group.

```
typedef struct ip_mreq {
  in_addr  imr_multiaddr;
  in_addr  imr_interface;
} ip_mreq;
```

imr_multiaddr

Multicast IPv4 address.

imr_interface

Local IP address.

8.2.24 ipv6_mreq

IPv6 multicast group.

```
typedef struct ipv6_mreq
{
    in6_addr      ipv6imr_multiaddr;
    unsigned int  ipv6imr_interface;
} ipv6_mreq;
```

ipv6imr_multiaddr

IPv6 multicast address of group.

ipv6imr_interface

Interface index. It equals to the scope zone index, defining network interface.

8.2.25 IP_STATS

A pointer to this structure is returned by **inet_pton()**.

```
typedef struct {
  uint32_t            ST_RX_TOTAL;
  uint32_t            ST_RX_MISSED;
  uint32_t            ST_RX_DISCARDED;
  uint32_t            ST_RX_ERRORS;

  uint32_t            ST_TX_TOTAL;
  uint32_t            ST_TX_MISSED;
  uint32_t            ST_TX_DISCARDED;
  uint32_t            ST_TX_ERRORS;

  RTCS_ERROR_STRUCT  ERR_RX;
  RTCS_ERROR_STRUCT  ERR_TX;

  uint32_t            ST_RX_HDR_ERRORS;
  uint32_t            ST_RX_ADDR_ERRORS;
  uint32_t            ST_RX_NO_PROTO;
  uint32_t            ST_RX_DELIVERED;
  uint32_t            ST_RX_FORWARDED;

  uint32_t            ST_RX_BAD_VERSION;
  uint32_t            ST_RX_BAD_CHECKSUM;
  uint32_t            ST_RX_BAD_SOURCE;
  uint32_t            ST_RX_SMALL_HDR;
  uint32_t            ST_RX_SMALL_DGRAM;
  uint32_t            ST_RX_SMALL_PKT;
  uint32_t            ST_RX_TTL_EXCEEDED;
  uint32_t            ST_RX_FRAG_RECVD;
  uint32_t            ST_RX_FRAG_REASMD;
  uint32_t            ST_RX_FRAG_DISCARDED;

  uint32_t            ST_TX_FRAG_SENT;
  uint32_t            ST_TX_FRAG_FRAGD;
  uint32_t            ST_TX_FRAG_DISCARDED
} IP_STATS, * 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.

8.2.26 IPCFG_IP_ADDRESS_DATA

Interface address structure.

```
typedef uint32_t _ip_address;

typedef struct ipcfg_ip_address_data
{
        _ip_addressip;
        _ip_address mask;
        _ip_address router;
} IPCFG_IP_ADDRESS_DATA, * IPCFG_IP_ADDRESS_DATA_PTR;
```

ip

ip address

mask

mask

route

gateway

8.2.27 IPCP_DATA_STRUCT

A pointer to this structure is a parameter of **RTCS_if_bind_IPCP()**.

```
typedef struct {
  void (_CODE_PTR_  IP_UP) (void*);
  void (_CODE_PTR_  IP_DOWN) (void*);
  void             *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, * IPCP_DATA_STRUCT_PTR;
```

IP_UP

IP_DOWN

IP_PARAM

RTCS calls	With	When IPCP successfully
IP_UP	*IP_PARAM*	Enters the opened state.
IP_DOWN	*IP_PARAM*	Leaves the opened state.

ACCEPT_LOCAL_ADDR

LOCAL_ADDR

IPCP attempts to negotiate *LOCAL_ADDR* as its local IP address.

If ACCEPT_LOCAL_ADDR is:	IPCP does this
TRUE	Allows the peer to negotiate a different local IP address.
FALSE	Accepts only *LOCAL_ADDR* as its local IP address.

ACCEPT_REMOTE_ADDR

REMOTE_ADDR

IPCP attempts to negotiate *REMOTE_ADDR* as the peer IP address.

If ACCEPT_REMOTE_ADDR is:	IPCP does this
TRUE	Allows the peer to negotiate a different peer IP address.
FALSE	Accepts only *REMOTE_ADDR* as its peer IP address.

NETMASK

DEFAULT_NETMASK

If DEFAULT_NETMASK is:	IPCP does this
TRUE	Dynamically calculates the link's netmask based on the negotiated local and peer IP addresses.
FALSE	IPCP always uses *NETMASK* 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 NEG_LOCAL_DNS is:	IPCP does this
TRUE	Attempts to negotiate *LOCAL_DNS* 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 NEG_REMOTE_DNS is	IPCP does this
TRUE	Attempts to negotiate REMOTE_DNS 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.

8.2.28 IPIF_STATS

A pointer to this structure is returned by **IPIF_stats()**.

```
typedef struct {
  uint32_t            ST_RX_TOTAL;
  uint32_t            ST_RX_MISSED;
  uint32_t            ST_RX_DISCARDED;
  uint32_t            ST_RX_ERRORS;

  uint32_t            ST_TX_TOTAL;
  uint32_t            ST_TX_MISSED;
  uint32_t            ST_TX_DISCARDED;
  uint32_t            ST_TX_ERRORS;

  RTCS_ERROR_STRUCT  ERR_RX;
  RTCS_ERROR_STRUCT  ERR_TX;

  uint32_t            ST_RX_OCTETS;
  uint32_t            ST_RX_UNICAST;
  uint32_t            ST_RX_MULTICAST;
  uint32_t            ST_RX_BROADCAST;

  uint32_t            ST_TX_OCTETS;
  uint32_t            ST_TX_UNICAST;
  uint32_t            ST_TX_MULTICAST;
  uint32_t            ST_TX_BROADCAST;
} IPIF_STATS, * 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.

8.2.29 nat_ports

Used by Freescale MQX NAT to control the range of ports between and including the minimum and maximum ports specified.

```
typedef struct {
   uint16_t  port_min;
   uint16_t  port_max;
} nat_ports;
```

PORT_MIN

Minimum port number.

PORT_MAX

Maximum port number.

8.2.30 NAT_STATS

Network address translation statistics.

```
typedef struct {
   uint32_t  ST_SESSIONS;
   uint32_t  ST_SESSIONS_OPEN;
   uint32_t  ST_SESSIONS_OPEN_MAX;

   uint32_t  ST_PACKETS_TOTAL;
   uint32_t  ST_PACKETS_BYPASS;
   uint32_t  ST_PACKETS_PUB_PRV;
   uint32_t  ST_PACKETS_PUB_PRV_ERR;
   uint32_t  ST_PACKETS_PRV_PUB;
   uint32_t  ST_PACKETS_PRV_PUB_ERR;
} NAT_STATS, * 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).

8.2.31 nat_timeouts

Used by Freescale MQX NAT to determine inactivity timeout settings.

```
typedef struct {
   uint32_t  timeout_tcp;
   uint32_t  timeout_fin;
   uint32_t  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.

8.2.32 PPP_PARAM_STRUCT

Used as parameter for initialization of PPP device.

typedef struct ppp_param_struct

```
{
    char*                   device;
    void (_CODE_PTR_ up)    (void *);
    void (_CODE_PTR_ down)  (void *);
    void                    *callback_param;
    _rtcs_if_handle         if_handle;
    _ip_address             local_addr;
    _ip_address             remote_addr;
    int                     listen_flag;
}PPP_PARAM_STRUCT;
```

device

Low-level communication device name. All PPP data are send through this device.

up

Function to be called when PPP link goes up.

down

Function to be called when PPP link goes down.

callback_param

Parameter for UP/DOWN callbacks.

if_handle

Handle to ipcp interface. PPP stores handle to IPCP device to this variable. It can be used to read remote and local IP address of PPP link.

local_addr

Local IP address to be used on PPP. Only relevant when listen_flag is set to TRUE.

remote_addr

IP address to be set to remote peer. Only relevant when listen_flag is set to TRUE.

listen_flag

Flag for determining if PPP should be started in listen mode (true) or connect mode (false).

8.2.33 PPP_SECRET

Used by PPP Driver for PAP and CHAP authentication of peers.

```
typedef struct {
  uint16_t   PPP_ID_LENGTH;
  uint16_t   PPP_PW_LENGTH;
  char  *PPP_ID_PTR;
  char  *PPP_PW_PTR;
} PPP_SECRET, * 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.

8.2.34 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 {
  uint32_t   ERROR;
  uint32_t   PARM;
  _task_id  TASK_ID;
  uint32_t   TASKCODE;
  void   *MEMPTR;
  bool   STACK;
} RTCS_ERROR_STRUCT, * 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 RTOS has allocated.

STACK

Whether the stack for the task that set the code is past its limit.

8.2.35 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 {
   uint32_t (_CODE_PTR_  OPEN) (struct ip_if *);
   uint32_t (_CODE_PTR_  CLOSE)(struct ip_if *);
   uint32_t (_CODE_PTR_  SEND) (struct ip_if *,
                               struct rtcspcb *,
                               _ip_address,
                               _ip_address);
   uint32_t (_CODE_PTR_  JOIN) (struct ip_if *,
                               _ip_address);
   uint32_t (_CODE_PTR_  LEAVE)(struct ip_if *,
                               _ip_address);
} RTCS_IF_STRUCT, * 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.

8.2.36 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 uint32_t (_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 the RTCS provides in *if\rtcsinit.c*:

```
uint32_t (_CODE_PTR_ RTCS_protocol_table[])(void) = {
   RTCSPROT_IGMP,
   RTCSPROT_UDP,
   RTCSPROT_TCP,
   RTCSPROT_IPIP,
   NULL
};
```

8.2.37 RTCS_TASK

Definition for Telnet Server shell task.

```
typedef struct {
  char           *NAME;
  uint32_t       PRIORITY;
  uint32_t       STACKSIZE;
  void (_CODE_PTR_   START)(void*);
  void          *ARG;
} RTCS_TASK, * 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.

8.2.38 RTCS6_IF_ADDR_INFO

```
typedef struct rtcs6_if_addr_info
{
    in6_addr              ip_addr;
    rtcs6_if_addr_state ip_addr_state;
    rtcs6_if_addr_type ip_addr_type;
} RTCS6_IF_ADDR_INFO, * RTCS6_IF_ADDR_INFO_PTR;
```

ip_addr

IPv6 address.

ip_addr_state

IPv6 address state (tentative or preferred).

ip_addr_type

IPv6 address type (set manually or using auto-configuration).

8.2.39 rtcs6_if_addr_type

```
typedef enum
{
    IP6_ADDR_TYPE_MANUAL = 0,
    IP6_ADDR_TYPE_AUTOCONFIGURABLE = 1
} rtcs6_if_addr_type;
```

IP6_ADDR_TYPE_MANUAL

IPv6 address is set manually.

IP6_ADDR_TYPE_AUTOCONFIGURABLE

IPv6 address is set using auto-configuration.

8.2.40 RTCSMIB_VALUE

```
typedef struct rtcsmib_value {
        uint32_t TYPE;
        void *PARAM;
} RTCSMIB_VALUE, * RTCSMIB_VALUE_PTR;
```

TYPE

Value type.

PARAM

8.2.41 SMTP_EMAIL_ENVELOPE structure

This structure stores information required for successful email delivery . In RFC refered to as SMTP envelope. Declaration can be found in file *rtcs_smtp.h*

```
typedef struct smtp_email_envelope
{
    char    *from;
    char    *to;
}SMTP_EMAIL_ENVELOPE, * SMTP_EMAIL_ENVELOPE_PTR;
```

from

Contains string passed as parameter to MAIL FROM command.

to

Contains string passed as parameter to RCPT TO command.

8.2.42 SMTP_PARAM_STRUCT structure

```
typedef struct smtp_param_struct
{
    SMTP_EMAIL_ENVELOPE envelope;
    char *text;
    struct sockaddr* server;
    char *login;
    char *pass;
    bool auth_req;
}SMTP_PARAM_STRUCT, * SMTP_PARAM_STRUCT_PTR;
```

envelope

The SMTP envelope as described in chapter SMTP_EMAIL_ENVELOPE structure.

ext

Body of the email that will be send. Inside must be the fully formatted email message. Minimum content and format of the message is following:

```
"From: <>\r\n"
"To: <>\r\n"
"Subject: \r\n"
"Date: \r\n\r\n"
```

For detailed example of the message format and usage please see file *\shell\source\rtcs\sh_smtp.c*.

server

The SMTP server that is used for email sending. Socket on SMTP port will be created and connected for communication with this server.

login

The username for SMTP authentication. Can be NULL no authentication is then used.

pass

The password for SMTP authentication. If NULL empty password will be send to server when using authentication.

8.2.43 sockaddr_in

Structure for a socket-endpoint identifier.

```
typedef struct sockaddr_in
{
  uint16_t  sin_family;
  uint16_t  sin_port;
  in_addr  sin_addr;
} sockaddr_in;
```

sin_family

Address family type.

sin_port

Port number.

sin_addr

IP address.

8.2.44 sockaddr

Structure for a socket-endpoint identifier supported by IPv4 and IPv6.

```
#if RTCSCFG_ENABLE_IP6
    typedef struct sockaddr
    {
        uint16_t  sa_family;
        char      sa_data[22];
    } sockaddr;
#else
    #if RTCSCFG_ENABLE_IP4
        #define sockaddr   sockaddr_in
        #define sa_family   sin_family
    #endif
#endif
```

sa_family

The code for the address format. It identifies the format of the data that follows.

sa_data

The actual socket address data which is format-dependent. The length also depends on the format.

Each address format has a symbolic name which starts with “**AF_**”.

AF_INET

This determines the address format that goes with the Internet namespace.

AF_INET6

This is similar to AF_INET, but refers to the IPv6 protocol.

AF_UNSPEC

This determines no particular address format.

8.2.45 TCP_STATS

A pointer to this structure is returned by **TCP_stats()**.

```
typedef struct {
  uint32_t            ST_RX_TOTAL;
  uint32_t            ST_RX_MISSED;
  uint32_t            ST_RX_DISCARDED;
  uint32_t            ST_RX_ERRORS;

  uint32_t            ST_TX_TOTAL;
  uint32_t            ST_TX_MISSED;
  uint32_t            ST_TX_DISCARDED;
  uint32_t            ST_TX_ERRORS;

  RTCS_ERROR_STRUCT  ERR_RX;
  RTCS_ERROR_STRUCT  ERR_TX;

  uint32_t            ST_RX_BAD_PORT;
  uint32_t            ST_RX_BAD_CHECKSUM;
  uint32_t            ST_RX_BAD_OPTION;
  uint32_t            ST_RX_BAD_SOURCE;
  uint32_t            ST_RX_SMALL_HDR;
  uint32_t            ST_RX_SMALL_DGRAM;
  uint32_t            ST_RX_SMALL_PKT;
  uint32_t            ST_RX_BAD_ACK;
  uint32_t            ST_RX_BAD_DATA;
  uint32_t            ST_RX_LATE_DATA;
  uint32_t            ST_RX_OPT_MSS;
  uint32_t            ST_RX_OPT_OTHER;

  uint32_t            ST_RX_DATA;
  uint32_t            ST_RX_DATA_DUP;
  uint32_t            ST_RX_ACK;
  uint32_t            ST_RX_ACK_DUP;
  uint32_t            ST_RX_RESET;
  uint32_t            ST_RX_PROBE;
  uint32_t            ST_RX_WINDOW;

  uint32_t            ST_RX_SYN_EXPECTED;
  uint32_t            ST_RX_ACK_EXPECTED;
  uint32_t            ST_RX_SYN_NOT_EXPECTED
  uint32_t            ST_RX_MULTICASTS;

  uint32_t            ST_TX_DATA;
  uint32_t            ST_TX_DATA_DUP;
  uint32_t            ST_TX_ACK;
  uint32_t            ST_TX_ACK_DELAYED;
  uint32_t            ST_TX_RESET;
  uint32_t            ST_TX_PROBE;
  uint32_t            ST_TX_WINDOW;

  uint32_t            ST_CONN_ACTIVE;
  uint32_t            ST_CONN_PASSIVE;
  uint32_t            ST_CONN_OPEN;
  uint32_t            ST_CONN_CLOSED;
  uint32_t            ST_CONN_RESET;
  uint32_t            ST_CONN_FAILED;
```

```
  uint32_t            ST_CONN_ABORTS;
} TCP_STATS, * 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.

8.2.46 UDP_STATS

A pointer to this structure is returned by **UDP_stats()**.

```
typedef struct {
  uint32_t           ST_RX_TOTAL;
  uint32_t           ST_RX_MISSED;
  uint32_t           ST_RX_DISCARDED;
  uint32_t           ST_RX_ERRORS;

  uint32_t           ST_TX_TOTAL;
  uint32_t           ST_TX_MISSED;
  uint32_t           ST_TX_DISCARDED;
  uint32_t           ST_TX_ERRORS;

  RTCS_ERROR_STRUCT  ERR_RX;
  RTCS_ERROR_STRUCT  ERR_TX;

  uint32_t           ST_RX_BAD_PORT;
  uint32_t           ST_RX_BAD_CHECKSUM;
  uint32_t           ST_RX_SMALL_DGRAM;
  uint32_t           ST_RX_SMALL_PKT;
  uint32_t           ST_RX_NO_PORT;
} UDP_STATS, * 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 which RTCS supports. RFC 894 (a standard for the transmission of IP datagrams over ethernet networks) defines the way 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.

6 bytes	6 bytes	2 bytes	46 - 1500 bytes	4 bytes
Destination address	Source address	Type	Data	FCS

Figure A-1. Ethernet Frame

A.2 ARP (Address Resolution Protocol)

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 a request that asks all hosts on the subnet whether they are the destination IP address. Even though all hosts receive ARP request, only the destination host replies. The destination host sends an ARP reply that contains the destination host's ethernet address directly to the source host without using a broadcast message.

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.

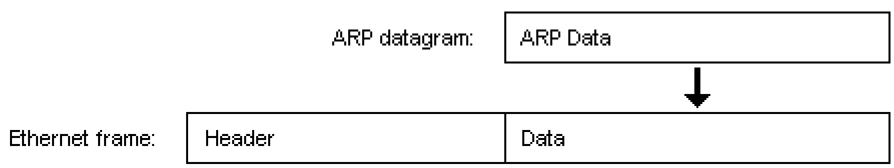

Figure A-3. ARP Datagram in an Ethernet Frame

A.3 IP (Internet Protocol)

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.

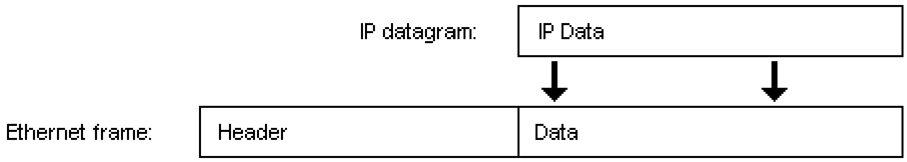

Figure A-5. IP Datagram in an Ethernet Frame

A.4 ICMP (Internet Control Message Protocol)

IP uses Internet Control Message Protocol (RFC 792) to send and receive errors and status information.

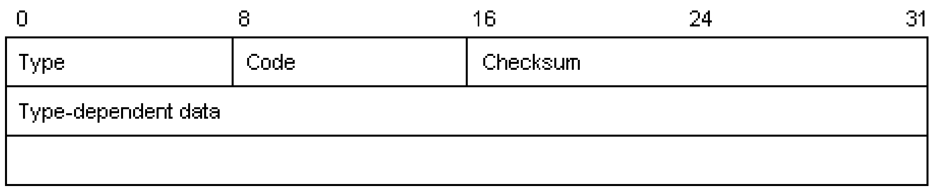

Figure A-6. ICMP Message

In an IP datagram that contains an ICMP message, the *Protocol* field contains one.

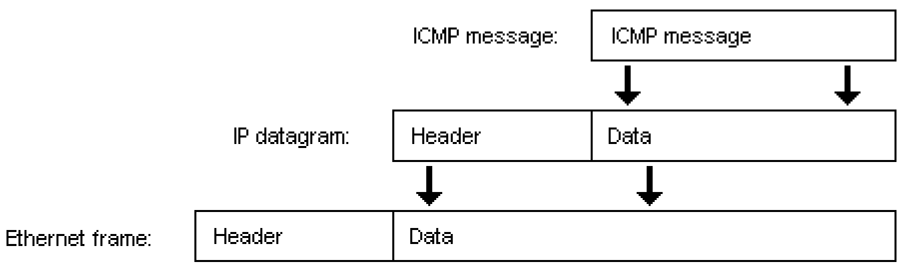

Figure A-7. ICMP Message in an Ethernet Frame

A.5 UDP (User Datagram Protocol)

User Datagram Protocol (RFC 768) provides the same unreliable, connectionless, datagram transport protocol as does the IP. In addition, UDP adds to the 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. An 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.

0 8	16 24 31
UDP source port	UDP destination port
UDP message length	UDP checksum
Data	

Figure A-8. UDP Datagram

In an IP datagram that contains a UDP datagram, the *Protocol* field contains 17.

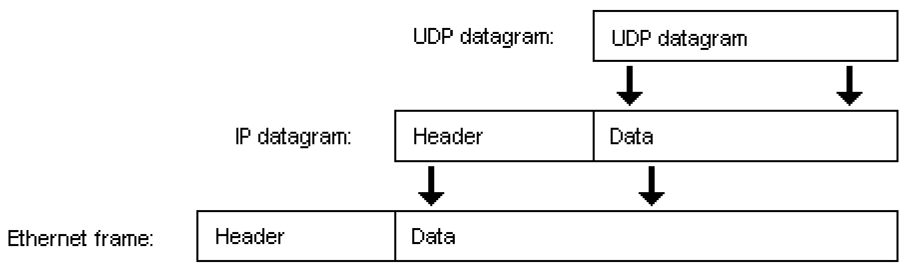

Figure A-9. UDP Datagram in an Ethernet Frame

A.6 TCP (Transmission Control Protocol)

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

0	8		16	24 31
Sequence port number			Destination port number	
Sequence number				
Acknowledgement number				
HLen	Reserved	Code bits	Window	
Checksum			Urgent pointer	
Data				

Figure A-10. TCP Segment

In an IP datagram that contains a TCP segment, the *Protocol* field contains six.

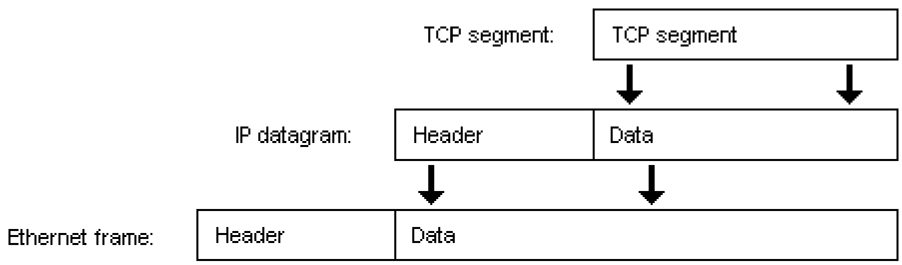

Figure A-11. TCP Segment in an Ethernet Frame

A.7 BootP (Boot Strap Protocol)

Bootstrap Protocol (RFC 951) is used to get an IP address based on an ethernet address, to load an executable boot file, and to run the loaded file.

BootP is built on top of UDP/IP and either 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.

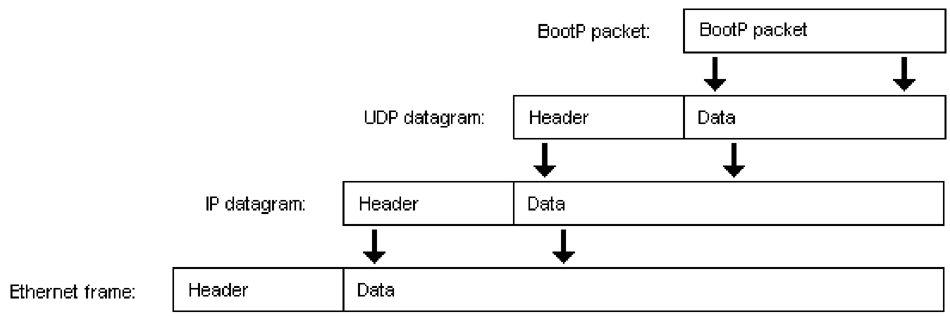

Figure A-12. BootP Packet in an Ethernet Frame

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)

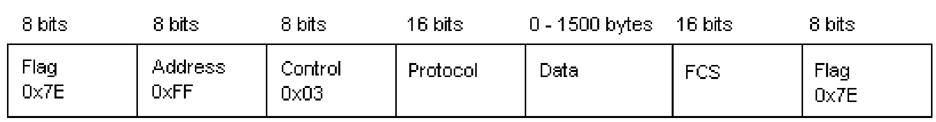

Figure A-13. PPP Frame

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 i.e 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 (Frame-Check Sequence)

The frame-check sequence by default uses CCITT-16, and is calculated over all bits of the *Address*, *Control*, *Protocol*, and *Data* fields.

8.3 LCP (Link Control Protocol)

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.

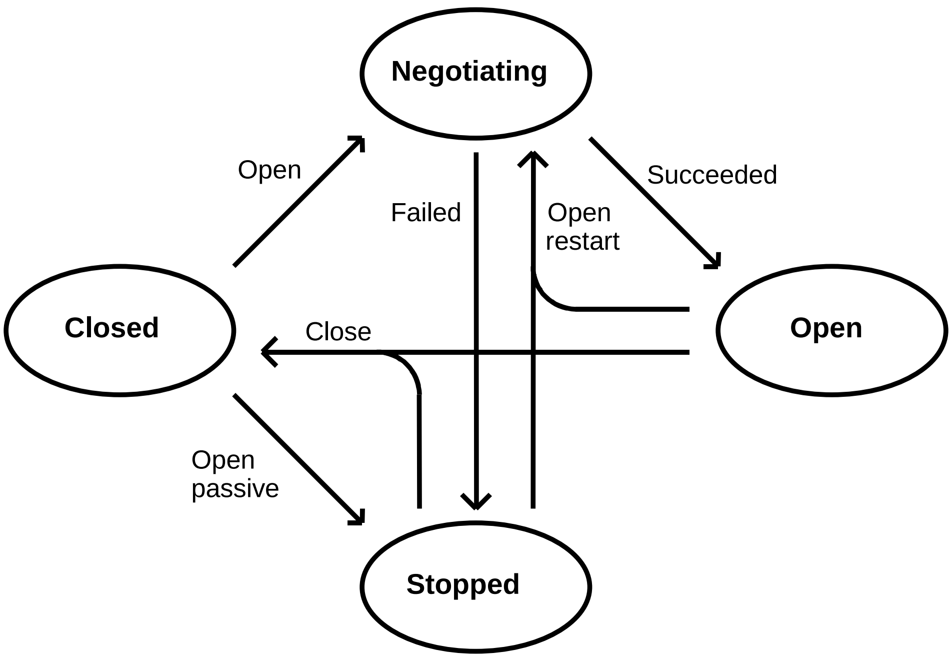

Figure A-14. PPP State Diagram

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.

8.4 SNTP (Simple Network Time Protocol)

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.

8.5 IPsec

IPsec (IP security) defines a set of protocols and cryptographic algorithms for creating secure IP traffic sessions between IPsec hosts. For more information, see 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)

8.6 NAT (Network Address Translator)

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. Therefore, they can 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 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)

NOTE	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.

Appendix B Internet Protocol Configuration

B.1 ipconfig Shell Command

Shell command ipconfig configures IP (internet protocol configuration).

Usage

```
ipconfig [<device>] [<command>]
```

Description

Shell command ipconfig displays all current TCP/IP network configuration values and refreshes Dynamic Host Configuration Protocol (DHCP) . Used without parameters, ipconfig displays MAC address, IP address, subnet mask, default gateway, and DNS for all adapters. Used with parameters, ipconfig can configure IP. See the list of available commands below.

Commands

init [<mac>]	Initialize Ethernet device (once).
task [start <priority> <period> | stop]	Manage link status checking task.
dns [add <ipv4> | del <ipv4>]	Manage DNS IP list. Support IPv4 only.
ip <ip> <mask> [<gateway>]	Bind with ip for IPv4. For IPv6 you should put ipv6 address only, for example, 'ip <ipv6>', to bind IPv6 address manually.
dhcp [<ipv4> <mask> [<gateway>]]	Bind with dhcp [use <ip> & <mask> in case dhcp fails]. Support IPv4 only.
autoip [<ipv4> <mask> [<gateway>]]	Obsolete, use 'ip' instead.
boot	Bind with boot protocol.
unbind [<ipv6>]	Unbind the network interface. Using 'unbind' without a parameter will unbind IPv4 address from the interface. In the case of IPv6, you should use the ipv6 address as a parameter to unbind it from the interface.

Parameters

<device>	Ethernet device number
<mac>	Ethernet MAC address.
<priority>	Link status task MQX RTOS priority.
<period>	Link status task check period (ms).
<ip>	IP address to use both families.
<ipv4>	IPv4 address to use.

<ipv6>	IPv6 address to use.
<mask>	Network mask to use.
<gateway>	Network gateway to use.

Example

```
shell> ipconfig
Eth#: 0
Link: off
MAC : 00:00:5e:fe:03:03
IP4 : 0.0.0.0 type: UNBOUND
IP6 : fe80::200:5eff:fefe:303 state: PREFERRED, type: AUTOCONFIGURABLE
MASK: 0.0.0.0
GATE: 0.0.0.0
DNS1: 169.254.0.1
Link status task stopped
shell> ipconfig dhcp
Bind via dhcp successful.
shell> ipconfig
Eth#: 0
Link: on
MAC : 00:00:5e:fe:03:03
IP4 : 192.168.5.101 type: DHCPNOAUTO
IP6 : fe80::200:5eff:fefe:303 state: PREFERRED, type: AUTOCONFIGURABLE
IP6 : 2001::200:5eff:fefe:303 state: PREFERRED, type: AUTOCONFIGURABLE
MASK: 255.255.255.0
GATE: 192.168.5.1
DNS1: 192.168.5.1
Link status task stopped
shell>
```