Freescale MQX™ RTOS RTCS User's Guide (IPv4 and IPv6)

Document Number: MQXRTCSUG
Rev. 3, 09/2015

freescale™

Contents

Section number | **Title** | **Page**

Chapter 1
Before You Begin

Chapter 2
Setting up the RTCS

Section number	Title	Page

Chapter 3 Using sockets

Section number	Title	Page

Chapter 5
RTCS application protocols

Section number	Title	Page

Section number	Title	Page

Chapter 6
Rebuilding

Chapter 7
Function Reference

Section number	Title	Page

Section number	Title	Page

Section number	Title	Page

Section number	Title	Page

Section number	Title	Page

Section number	Title	Page

Chapter 8
Compile-time Options

Section number	Title	Page

Section number	Title	Page

Chapter 9
Data Types

Section number	Title	Page

Section number	Title	Page

Chapter 1
Before You Begin

1.1 About This Book

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

This document 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 RTOS 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 about commands or procedures that could have unexpected or undesirable side effects, or could be dangerous to files or hardware.

CAUTION

If you modify MQX RTOS 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 it is ready with sockets.

For information about	See
Data types mentioned in this chapter	Chapter 8, "Data Types"
PPP Driver	Chapter 4, "Point-to-point drivers"
Protocols	Appendix A, "Protocols and Policies"
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 discussed in this manual. For more information about protocols, see the table and Appendix A, "Protocols and Policies" .

2.3 RTCS Included with Freescale MQX RTOS

The RTCS stack included in Freescale MQX RTOS distribution is based on the ARC RTCS version 2.97. See the Release Notes document accompanying the Freescale MQX RTOS for any new RTCS features supported.

Major changes in the RTCS introduced in Freescale MQX RTOS distribution:

- RTCS is now distributed within the Freescale MQX RTOS package. The RTCS adopts version numbering of the Freescale MQX RTOS distribution, starting with 3.0.
- RTCS build process and compile-time configuration follow the same principles as other MQX RTOS 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 RTOS distribution.
- A new HTTP server functionality is added in the Freescale MQX RTOS release.

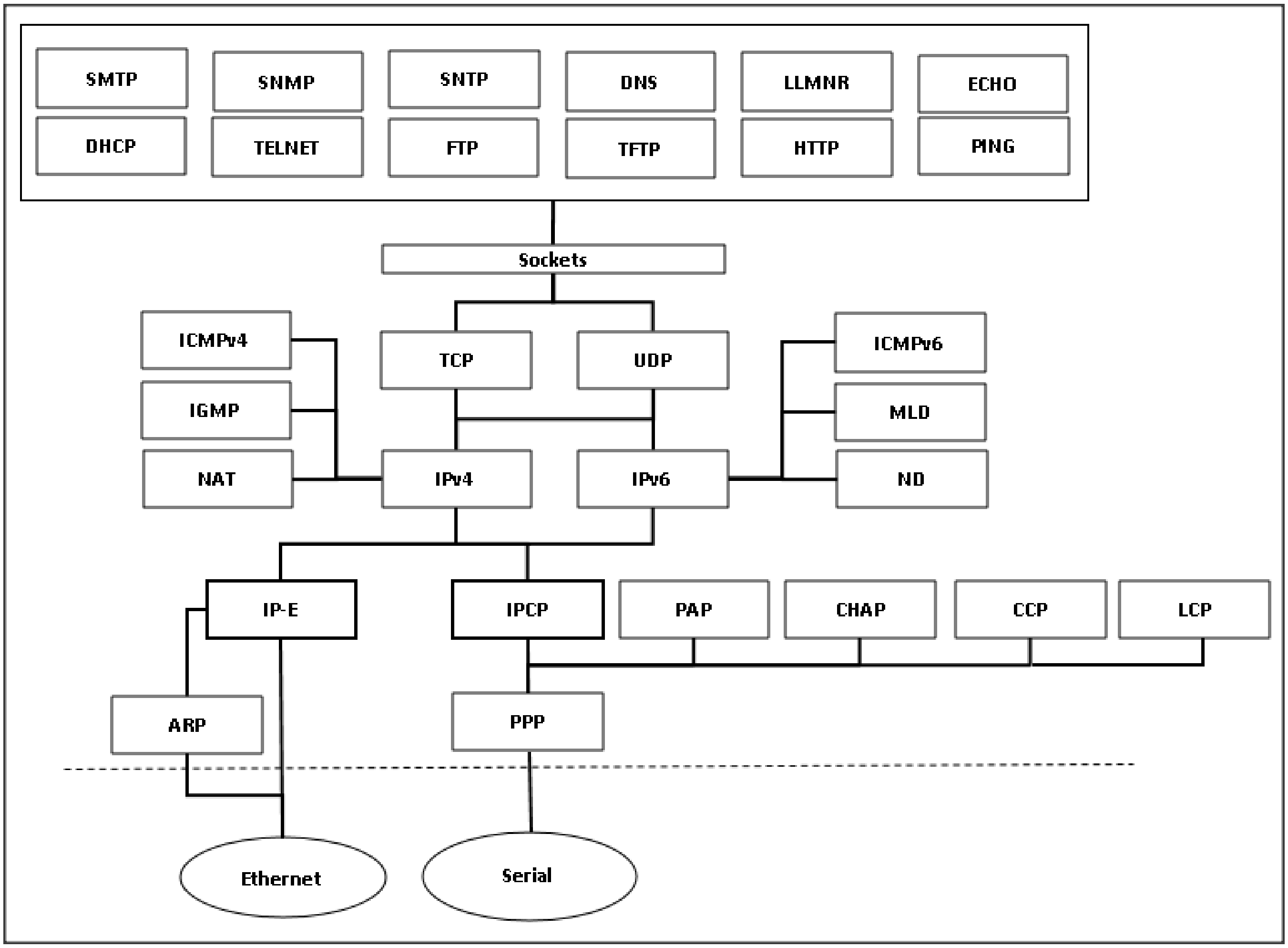

Figure 2-1. Protocols

Table 2-1. RTCS features

Protocol or policy	Description	RFC
ARP	Address Resolution Protocol for Ethernet	826
BootP	Bootstrap Protocol	951, 1542

Table continues on the next page...

Table 2-1. RTCS features (continued)

Protocol or policy	Description	RFC
CCP	Compression Control Protocol (used by PPP)	1692
CHAP	Challenge Handshake Authentication Protocol (used by PPP)	1334
CIDR	Classless Inter-Domain Routing	1519
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
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
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
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
SMI	Structure of Management Information	1155

Table continues on the next page...

Table 2-1. RTCS features (continued)

Protocol or policy	Description	RFC
SNMPv1 MIB	SNMPv1 Management Information Base	1213
SNMPv2	SNMP version 2	1902 — 1907
SNMPv2 MIB	SNMPv2 Management Information Base	1902, 1907
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

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.

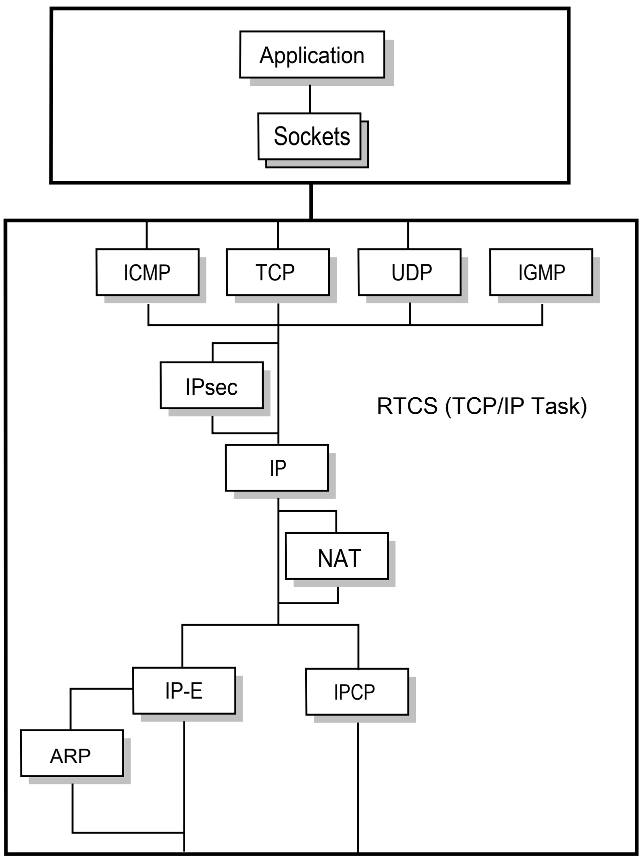

Figure 2-2. Protocol stack architecture

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.

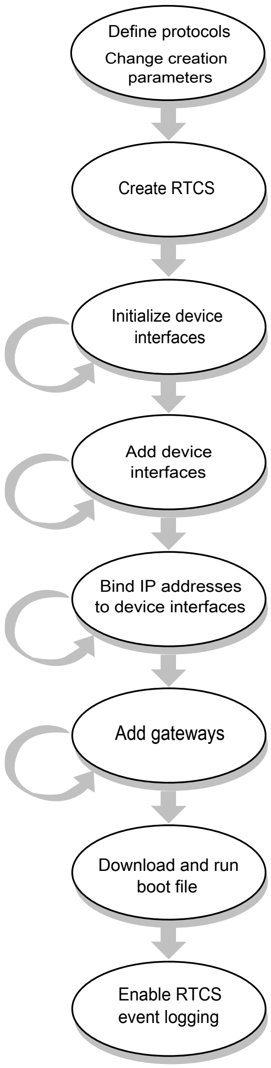

Figure 2-3. Steps to set up the RTCS

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 "RTCS_protocol_table" in Chapter 8, "Data Types" for the list of available protocols. You can add or remove protocols using the instructions provided there, 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 or before it calls RTCS_create().

To change:	From this default value:	Change this creation variable:
Priority of RTCS task. If the priority of RTCS task is too low, RTCS might miss received packets or violate the timing specifications for a protocol.	6	_RTCSTASK_priority
RTCS task (TCP/IP) stack size	3000	_RTCSTASK_stacksize
Maximum number of packet control blocks (PCBs) that RTCS uses	10	_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 or 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

You may want to bypass the generation and verification of TCP checksums in isolated networks, if the performance of data transfer is an issue.

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.

The initialization function returns a handle to the interface when the application initializes an interface to a device. 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

Call **ENET_get_stats()** to the device handle to the interface o get statistics about ethernet interfaces.

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. Call **RTCS_gate_add()** with the IP address of the gateway and a network mask to add a gateway.

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

Call **RTCS_gate_remove()** to remove a gateway.

2.13 Enabling RTCS logging

You can enable RTCS event logging in the MQX RTOS 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, a 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*. To enable RTCS event logging calling **RTCSLOG_enable()** with a required event mask. Call **RTCSLOG_disable()** to disable RTCS event logging.

2.14 Starting network address translation

NAT allows sites using private addresses to initiate unidirectional, 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.

The border router translates the source IP address to an address from the reserved pool when a packet leaves the private network 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.

A space for an internal configuration structure is allocated at initialization time. 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.

2.14.1 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 SOL_NAT_setsockopt() function.

When the SOL_NAT_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.

2.14.2 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 SOL_NAT_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. The maximum port number is 20000.

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

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

2.14.3 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 this:

```
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 like this at compile time:

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

2.14.4 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() .

2.14.5 Supported protocols

The Freescale MQX RTOS 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 Appendix A, "Protocols and Policies" .

2.14.5.1 Limitations

Freescale MQX RTOS 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 RTOS 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_gate_add	Adds a gateway to RTCS.
RTCS_gate_remove	Removes a gateway from RTCS.
RTCS_if_add	Adds a device interface to RTCS.

Table continues on the next page...

Table 2-2. Summary: Setup Functions (continued)

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.
SOL_NAT_setsockopt()	Sets the NAT options.

2.14.6 Example: setting up RTCS

Set up RTCS with one Ethernet device like this:

```
_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);
```

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	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:

- 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. If desired, an application can prespecify a remote endpoint for a datagram socket.

3.7 Unreliable transfer

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

3.8 Block-oriented

A datagram socket is block-oriented, which means when an application sends a block of data, the bytes of data remain together. If an application writes a block of data of, for example, 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.

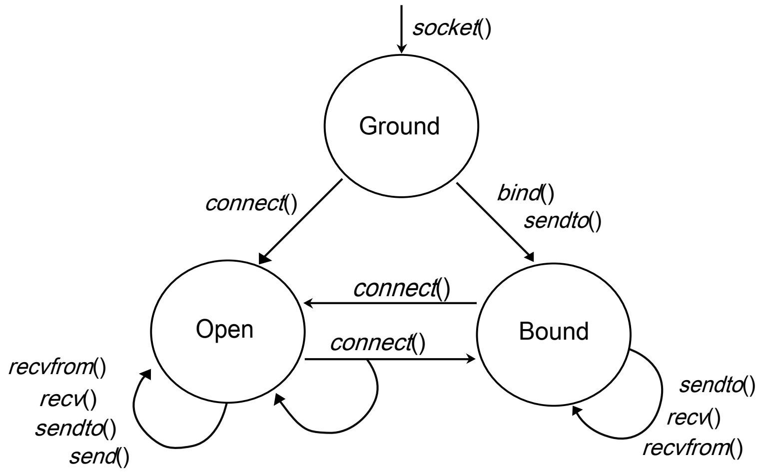

Figure 3-1. Creating and using datagram sockets (UDP)

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

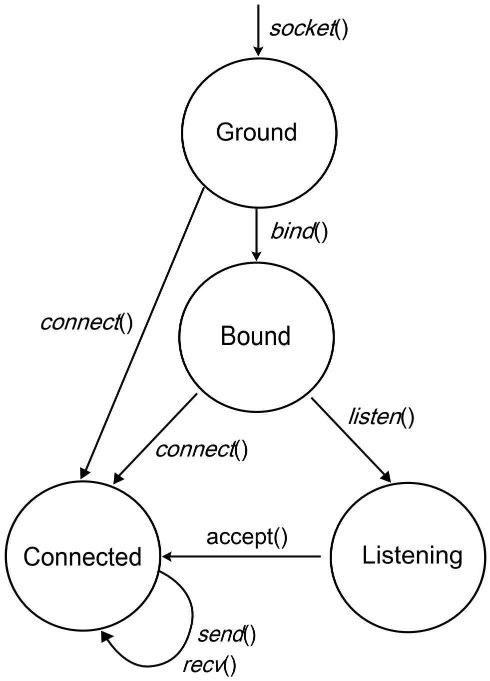

Figure 3-2. Creating and using stream sockets (TCP)

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.

By default, send() and sendto() return when the data is copied to the socket layer and scheduled for sending (non-blocking behavior).

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()** buffer outgoing data. This behavior can be changed by using either the OPT_SEND_NOWAIT socket option, or the RTCS_MSG_BLOCK 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 **closesocket()**. 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 **closesocket()** 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 these 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 calling **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 persocket 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()** block until the remote endpoint acknowledges some or all of the data.

The size of the RTCS persocket 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, which 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 **closesocket()**. The SO_LINGER socket option indicates how the socket is to be shut down: either gracefully or with an abort operation (TCP reset). By default, the function returns immediately. The SO_LINGER socket option can be used if blocking behavior is required.

Outstanding calls to **send()** and **recv()** return immediately and RTCS discards any data that is in its receive buffer for the socket before **closesocket()** returns. When the closesocket() function returns, the socket handle is invalid and the application can no longer use 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. By default, RTCS maintains the socket connection for two seconds after the remote endpoint disconnects. Use SO_LINGER socket option if longer time is required.

3.29.2 Shutting down with an abort operation

These actions occur if the socket is shut down with an abort operation:

- 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.
closesocket()	Shuts down the connection and discards the socket.
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_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.
select()	Waits for activity on any socket in given socket sets. Can distinguish between read and write activity.
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.
shutdownsocket()	Disallows further send or receive requests for 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) {
    uint32_t rtcserrno = RTCS_get_errno();
    if(rtcserrno == RTCSERR_SOCK_CLOSED) {
        /* Other task must have called closesocket() on the listensock handle.
         * The listensock does not exists, we must not access it's handle anymore.
         */
        handle_closed_listensock();
        _task_block();
    }
    else {
        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	Chapter 7, "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 multiprotocol datagrams over point-to-point links. The 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, and the most significant bit 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.

We define bit zero to be the least significant bit since all processors do not number bits in the same way.

The driver sends escaped characters as two bytes in this order:

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

For example, if bit zero of the ACCM is one, every 0x00 byte 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

ACFC is FALSE by default. 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 the "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

PPP Driver does not negotiate the MRU, but is prepared to advertise any MRU that is up to 1500 bytes by default. 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

PFC is FALSE by default. 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 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, 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, 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 these:

- PAP
- CHAP

Each protocol uses ID-password pairs (PPP_SECRET structure). For details of the structure, see the listing for PPP_SECRET in Chapter 8, "Data Types" .

4.2.3.1 PAP

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

- _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 these 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".

For PAP authentication on the client side, initialize these global variables.

```
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, as it lets you have a different password on each host that you might want to connect to. User "arc" has two accounts, using these:

- 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

For PAP authentication on the server side, initialize these global variables

```
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

For CHAP authentication on the server side, 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 Chapter 2, "Setting up the 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 application protocols

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	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	Chapter 7, "Function Reference"
Setting up the RTCS	Chapter 2, "Setting up the RTCS"
Using RTCS and sockets	Chapter 3, "Using Sockets"

5.2 DHCP client

The Dynamic Host Configuration Protocol (DHCP) is a binding protocol, as described in RFC 2131. Freescale MQX RTOS 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.

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. The client does not accept the server's offer if a host on the network answers the ARP. 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
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 Chapter 7, "Function Reference" .

5.3 DHCPv6 Client

The Dynamic Host Configuration Protocol for IPv6 (DHCP) enables DHCP servers to pass configuration parameters such as IPv6 network addresses to IPv6 nodes. It offers the capability of automatic allocation of reusable network addresses and additional configuration flexibility. For further information see RFC3315 (http://tools.ietf.org/html/rfc3315).

5.3.1 Supported features

Following network configuration options are supported:

- Client IPv6 addresses. Up to IP6_IF_ADDRESSES_MAX addresses are supported.
- DNS servers.
- Domain Search List (default domain).

Link checking feature:

- Client has an integrated link status check, which allows address confirmation/rebind whenever device is reconnected to same or different network. This feature can be enabled by setting flag DHCPCLN6_FLAG_CHECK_LINK in client initialization parameters. See section Obtaining addresses/other configuration for information about this flag.

Stateless configuration:

- Client can also be run in so-called "stateless" mode. In this mode, IP address is obtained from stateless address configuration and only additional information like DNS server addresses is acquired from DHCP server. This feature can be enabled by setting DHCPCLN6_FLAG_STATELESS flag in client initialization parameters. See section Obtaining addresses/other configuration for information about this flag.

5.3.2 Obtaining addresses/other configuration

To obtain the IP address from DHCPv6 server, start the client with DHCPCLN6_init() function with the DHCPCLN6_PARAM_STRUCT parameter. After the client starts, it automatically obtains the network configuration from the server. These are the relevant members for the DHCPCLN6_PARAM_STRUCT:

The following flags are supported:

flags member

DHCPCLN6_FLAG_STATELESS - When this flag is set, DHCPv6 client requests only additional information from server (DNS prefix, DNS server IP address etc.) but no IP address.

DHCPCLN6_FLAG_CHECK_LINK - If this flag is set, client checks a link status on interface it is running on. If the link is lost and then regained, CONFIRM/REPLY message exchange is performed.

interface member

This variable is handle to RTCS interface on which DHCP client is started. Setting this variable is mandatory. If invalid handle is passed to DHCPCLN6_init() function it fails.

5.3.3 Releasing obtained addresses

To release any of addresses obtained by DHCPv6 client, call the RTCS6_if_unbind_addr() function. If there are no addresses bound by DHCP, client stops automatically.

5.3.4 Stopping the client

To stop DHCPv6 client and release all addresses obtained, call function DHCPCLN6_release() with client handle (return value of DHCPCLN6_init()) as a parameter.

5.4 DHCP server

DHCP server allocates network addresses and delivers initialization parameters to client hosts that request them. Freescale RTCS DHCP Server is based on RFC2131.

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. Pass the DHCPSVR_FLAG_DO_PROBE flag to DHCPSRV_set_config_flag_off() to disable probing.

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

5.4.1 Example: setting up and modifying DHCP Server

See DHCPSRV_init() in Chapter 7, "Function Reference".

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()**.

Echo Server communicates with a client. The RTCS contains the Echo client application that can be used to communicate with the Echo server.

5.6 Echo client

The Echo client implements a client for the RFC 862 Echo server. The echo client sends a data to the server, receives data back from the server, and compares the outgoing data with the received data. To connect to a server, the application calls the ECHOCLN_connect() function. After the connection is established, the ECHOCLN_process() is used to exchange data.

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

These commands are supported by 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 FTP server creates its main and session task, and can be overridden by setting server_prio member of the server initialization structure to a nonzero value. The value of this macro is set to TCP/IP task priority decreased by 1 by default.

- 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 the selected address family, in the server initialization structure. The default value of this macro is INADDR_ANY.
- FTPSRVCFG_DEF_SES_CNT: Default maximum number of sessions. This value limits maximum number of sessions, or connections, to the server. A new session is created each time a new connection is established from the client. The value of this parameter can be overridden by setting the max_ses member of the server initialization structure. The default value is 2 sessions.
- FTPSRVCFG_TX_BUFFER_SIZE: Size of the socket transmit buffer in bytes. This option cannot be overridden in runtime. The default value is 1460 bytes.
- FTPSRVCFG_RX_BUFFER_SIZE: Size of the socket receive buffer in bytes. This option cannot be overridden in runtime. The default value is 1460 bytes.
- FTPSRVCFG_TIMEWAIT_TIMEOUT: Timeout value for send/receive operations on the sockets in milliseconds. This option cannot be overridden in runtime. The default value is 1000 ms.
- FTPSRVCFG_SEND_TIMEOUT: Timeout value for server sockets in milliseconds, and cannot be changed during runtime. The default value is 500ms.
- FTPSRVCFG_CONNECT_TIMEOUT: Hard timeout for connection establishment in milliseconds for FTP server sockets, and cannot be changed during runtime. The default value is 5000ms.
- FTPSRVCFG_RECEIVE_TIMEOUT: Timeout for the recv() function. After this timeout recv() returns with whatever data it has, and cannot be changed during runtime. The default value is 50ms.
- FTPSRVCFG_IS_ANONYMOUS: Macro defining if login/password are required to run privileged 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, except the root directory, are optional. You can set any parameter to zero and the server uses 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_ftpsrv.c. The server parameters structure description can be found in Chapter FTPSRV_PARAM_STRUCT.

5.9 LLMNR server

The Link-Local Multicast Name Resolution (LLMNR) is a network protocol that allows both IPv4 and IPv6 hosts to perform name resolution for hosts on the same local link. The LLMNR is defined in the RFC 4795. It is based on the Domain Name System (DNS) packet format, but operates on a separate port from the DNS. It is natively supported by the Microsoft Windows® operating systems. Therefore, no additional PC-tool installation is required. The RTCS supports the LLMNR server.

There are only two steps the user must follow to start the LLMNR server:

1. Create and fill the LLMNRSRV_PARAM_STRUCT structure with the required server settings, such as a host-name table and an interface handle on which the LLMNR server is listening. The user may set any other parameter to zero and the server uses a default value.
2. Start the server using the LLMNRSRV_init() function with a parameter structure created in the previous step. Both of these steps are demonstrated by an example which the user can find in the %MQX_PATH%\shell\source\rtcs\sh_llmnrsrv.c. location.

The sever can be stopped using the LLMNRSRV_release() function.

5.10 HTTP server

Hypertext Transfer Protocol (HTTP) server is a simple web server that handles, evaluates, and responds to HTTP requests. Depending on the configuration and incoming client requests, it returns static file system content, such as web pages, style sheets, images, or content dynamically generated by callback routines. The HTTPSRV application supports HTTP protocol in version 1.0 defined by RFC1945. Additionally, these HTTP 1.1 features are implemented:

- GET, POST, and HEAD requests
- CGI scripts RFC3875
- ASP-like Server Side Includes (commands with parameters enclosed by '<%' and '%>')
- Basic authentication
- HTTP keep-alive
- Percent encoded URI
- Cache control

- Multiple root directories (aliases)
- Chunked transfer encoding

The server creates a separate task and an internal data structure for every incoming connection from the client. This is the called session, described further intext. When the session processing is done (a response is sent 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.

5.10.1 Cache control

The server implements a simple HTTP cache control directives, which means that static files are cached in a web browser and need not to be updated when the webpage is reloaded. Below is the list of cached extensions (directive Cache-Control: max-age=3600):

- 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 3600 s. See RFC2616 section 14.9 for more details about the cache control mechanism.

5.10.2 Supported MIME types

These MIME types are supported:

- text/plain
- text/html
- text/css
- image/gif
- image/jpeg
- image/png

- image/svg+xml
- application/javascript
- application/xml
- application/zip
- application/pdf
- application/octet-stream

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

5.10.3 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. This 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");
}
```

5.10.4 Compile time configuration

A few macros are used for setting HTTP server default configuration during compile time. Default values for 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_ADDR: Default server IPv4/IPv6 address. The server is listening on this address if different value is not set by 'ipv4_address' or 'ipv6_address' member, depending on selected address family, in the server initialization structure. The default value of this macro is INADDR_ANY.

- 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 nonzero value. The value of this macro is set to priority of RTCS TCP/IP task decreased by 1 by default.
- HTTPSRVCFG_DEF_PORT: Default port to listen on. It can be overridden by setting a nonzero value of the port member in the server initialization structure. The 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. The default index page is "index.htm".
- HTTPSRVCFG_DEF_SES_CNT: Default maximum number of sessions. This value limits maximum number of sessions, or connections, created by 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. The 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. The 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. The 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 sent to the client. The 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. The default value of this macro is 32.
- HTTPSRVCFG_KEEPALIVE_ENABLED: Macro determining if HTTP keep-alive is enabled or disabled. The 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 waits for a next request after the previous request was successfully processed. This value cannot be overridden during runtime. The 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. The 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. The default value is 1460 bytes.

- HTTPSRVCFG_RX_BUFFER_SIZE: Size of the socket receive buffer in bytes. This option cannot be overridden in runtime. The default value is 1460 bytes.
- HTTPSRVCFG_TIMEWAIT_TIMEOUT: Timeout value for send/receive operations on the sockets in milliseconds. This option cannot be overridden in runtime. The default value is 1000 ms.
- HTTPSRVCFG_RECEIVE_TIMEOUT: Timeout for the recv() function. After this timeout recv() returns with whatever data it has, and cannot be changed during runtime. The default value is 50 ms.
- HTTPSRVCFG_CONNECT_TIMEOUT: Hard timeout for connection establishment in miliseconds for HTTP server sockets. This cannot be changed during runtime. The default value is 5000 ms.
- HTTPSRVCFG_SEND_TIMEOUT: Timeout value for server sockets in milliseconds. This option cannot be changed during runtime. The default value is 500 ms.

5.10.5 Basic usage

These are the 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 and the server uses 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.

5.10.6 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 the 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):One task is created to handle all user scripts on server startup. This task has a stack size determined by the script_stack variable in the server parameters structure. A message is sent from a session to this task and runs the script when a script is to be executed. 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 the previous case, a task is created on the server startup to handle scripts, but this task has a stack of minimal size. When the script is encountered during the session processing, a message is sent to this task. Instead of running a user callback, a new detached task is created with stack size set to value from the CGI/SSI link structure. 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, such as 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 the 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 nonzero stack in script table, it is processed in the separate task.

Example:

```
/* First the table of CGI callbacks is created*/
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, these steps must be taken:

1. Check the method of request (GET or POST).

2. Create a variable of type HTTPSRV_CGI_RES_STRUCT, called "response" further in the text.
3. Read the data from the client using httpsrv_cgi_read() function. All 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 the address of data you want to send, and store the length of data in bytes to the response.data_length variable. Whenever you call httpsrv_cgi_write(), the data is stored in the session buffer and then sent to the client.

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

5.10.7 Using server side include (SSI) callbacks

Server side includes functions that are called every time aspecial 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 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);
```

When writing something from server side, include the response sent to the client, and use the httpsrv_ssi_write() function.

5.10.8 Secure HTTP using CyaSSL

HTTPSRV supports the HTTPS protocol. To enable SSL in HTTPSRV you must pass valid pointer to HTTPSRV_SSL_STRUCT structure as a parameter in HTTPSRV_PARAMS_STRUCT. See project located in %MQX_ROOT%\rtcs\examples\httpsrv for HTTPSRV+SSL for a code example.

5.10.9 Chunked transfer coding

Since MQX RTOS version 4.1.2 there is support for chunked transfer coding in HTTPSRV. This feature allows sending data of unknown overall size (without content-length) when HTTP keep-alive is enabled. To activate it simply call HTTPSRV_cgi_write() with response parameter with content_length set to -1. Each subsequent call of HTTPSRV_cgi_write() then leads to creation of one chunk of data for client with data_length size.

To terminate the transfer and thus signalizing to client that all data is send call HTTPSRV_cgi_write() with data_length set to zero and data set to NULL. For further details about chunked transfer coding see RFC2616, section 3.6.

Example:

```
static _mqx_int cgi_test(HTTPSRV_CGI_REQ_STRUCT* param)
{
   HTTPSRV_CGI_RES_STRUCT response = {0};

   response.ses_handle = param->ses_handle;

   response.data = "Th";
   response.data_length = strlen(response.data);
   response.content_length = -1;
   HTTPSRV_cgi_write(&response);

   response.data = "is is";
   response.data_length = strlen(response.data);
   HTTPSRV_cgi_write(&response);

   response.data = " test\r\n\r\nstring.";
   response.data_length = strlen(response.data);
   HTTPSRV_cgi_write(&response);

   response.data = NULL;
   response.data_length = 0;
   HTTPSRV_cgi_write(&response);
}
```

5.10.10 HTTP server memory requirements

Various components of HTTP server application have different memory requirements. All items marked with asterisk (*) are user configurable and thus depending on your application. Default values are listed in tables.

Table 5-3. Server task

Memory used (bytes)	
Stack	1500
Context	148
Total	1648

Server task in mandatory component of HTTPSRV and is created automatically once the server is started. No user available configuration can affect its memory usage. One or two listening sockets are created depending on address families enabled in HTTPSRV_PARAM_STRUCT supplied by user (one for IPv4 and one for IPv6).

Table 5-4. Session task

Memory used (bytes)	
Data buffer*	1360
Socket buffers*	2920
Context	140
Stack**	3000
URL	129
Total	7549

At least one session is required for the server to function properly. Session is created whenever there is incoming HTTP request. Data buffer size is defined by macro HTTPSRVCFG_SES_BUFFER_SIZE. Socket buffer are affected by HTTPSRVCFG_TX_BUFFER_SIZE and HTTPSRVCFG_RX_BUFFER_SIZE macros. Maximum length of URL is defined by HTTPSRVCFG_DEF_URL_LEN. One socket per session is created for communication with client.

Tip

Session stack size is not user configurable, but can be changed in file httpsrv_prv.h (HTTPSRV_SESSION_STACK_SIZE). You might want to change it because default value is selected for worst case (debug configuration with worst compiler).

DANGER

Be aware that if session stack size is too small, it is highly probable that session stack overflows and crash/HardFault your application!

Table 5-5. Script handler task

Memory used (bytes)	
Stack	850
Message queue	192
Total	1042

Script handler task is only created is CGI/SSI table pointer is present in HTTPSRV_PARAM_STRUCT supplied by user. Its stack size can be specified by script_stack variable in server parameters structure. Also it is possible to create multiple script handlers by specifying stack sizes in HTTPSRV_CGI_LINK_STRUCT. See chapter about CGI/SSI processing for further details. There is also message queue created for API call receiving.

Table 5-6. WebSocket task

Memory used (bytes)	
Data buffer*	1360
Socket buffers*	2920
Context	148
Stack**	3000
Total	7428

WebSocket is optional HTTPSRV component. It can be enabled by setting macro HTTPSRVCFG_WEBSOCKET_ENABLED to 1. Data buffer size is defined by macro HTTPSRVCFG_SES_BUFFER_SIZE and socket buffers sizes are defined by macro HTTPSRVCFG_TX_BUFFER_SIZE and HTTPSRVCFG_RX_BUFFER_SIZE. One socket per WebSocket connection is created.

Tip

WebSocket task stack size is not user configurable, but can be changed in file httpsrv_prv.h (HTTPSRV_SESSION_STACK_SIZE).You might want to change it because default value is selected for worst case (debug configuration with worst compiler).

DANGER

Be aware that if WebSocket stack size is too small, it is highly probable that session stack overflows and crashes/HardFaults your application!

5.11 WebSocket Protocol

WebSocket is a protocol providing full-duplex communication channel over TCP connection. The protocol consists of an opening handshake followed by basic message framing, layered over TCP and is standardized by RFC6455. Implementation in RTCS is done as part of HTTPSRV application (a plugin). The WebSocket simplifies much of the complexity around bi-directional web communication and connection management.

For every WebSocket connection, one task is created to handle all the receive and transmit operations. The communication socket and data buffer are reused from the HTTP session, which is required for the WebSocket handshake. Full UTF-8 data validation is implemented and interoperability with all modern web browsers is provided.

5.11.1 The WebSocket API

The WebSocket API consists of two functions and four callbacks:

- Function WS_send() is used for sending data through the WebSocket.
- Function WS_close() is used to close the WebSocket.
- Callback on_message is invoked when message is received.
- Callback on_error is invoked when error occurs.
- Callback on_connect is invoked when new WebSocket connection is created.
- Callback on_disconnect is invoked when WebSocket connection is released.

5.11.2 Creating the WebSocket as a HTTPSRV plugin

To setup the plugin several steps must be completed:

1. Create a structure of type WS_PLUGIN_STRUCT and fill in user functions. This structure determines which function is called in case of a WebSocket event. Example defining simple WebSocket echo:

```
WS_PLUGIN_STRUCT ws_echo_plugin = {
    echo_connect, //callbacks
    echo_message,
    echo_error,
```

```
    echo_disconnect,
    NULL //callback parameter
};
```

2. Create a structure of type HTTPSRV_PLUGIN_STRUCT to define the web server plugin and its type. Example defining our echo callbacks as WebSocket plugin:

```
HTTPSRV_PLUGIN_STRUCT echo_plugin = {
    HTTPSRV_WS_PLUGIN, //plugin type
    (void *) &ws_echo_plugin  //pointer to callbacks
};
```

3. Create a structure of type HTTPSRV_PLUGIN_LINK_STRUCT which links a server resource to the server plugin. Example linking ws://SERVER_ADDRESS/echo to our echo plugin:

```
HTTPSRV_PLUGIN_LINK_STRUCT plugins[] = {
    {"/echo", &echo_plugin},
    {NULL, NULL}
};
```

4. Set a plugin array as initialization parameter for the HTTP server. Example setting plugins structure as server plugins:

```
HTTPSRV_PARAM_STRUCT params;
...
params.plugins = &plugins;
...
HTTPSRV_init(&params);
...
```

Now every time a client (i.e web browser) requests URI ws://SERVER_ADDRESS/echo, a WebSocket connection is created and callbacks are invoked to handle WebSocket events.

5.11.3 Sending data through WebSocket

To send data the function WS_send() is used. It has a parameter structure of type WS_USER_CONTEXT_STRUCT. Most important variables in this structure are handle, data and fin_flag.

1. The handle variable - This variable is simply a number identifying the connection. It can be first retrieved from parameter of the on_connected callback.
2. The data variable - This variable is of type WS_DATA_STRUCT. Data to be send to client are passed through this variable. It has three items:
    - data_ptr - Pointer to data.
    - length - Length of data.
    - type - Type of data (text or binary).

The fin_flag variable - This variable is used for indication of end of message. When its value is greater than zero, buffer content is flushed to the client.

CAUTION

When data are send it is important to set value of the fin_flag variable to non-zero value in case the last chunk of data is sent to client. Otherwise there is no way in which the server can determine end of user data. You can set fin_flag every time you call WS_send() but be aware that such a method may affect the WebSocket performance.

Examples describing difference in fin_flag usage (sending message "Hello World!" in two data chunks):

1. Setting the fin_flag only for last chunk of data:
   - Data sent on server:
     1. Write data "Hello "; fin_flag = 0.
     2. Write data "World!"; fin_flag = 1.
   - Messages received by client:
     1. "Hello World!"
2. Setting the fin_flag for every chunk of data:
   - Data sent on server:
     1. Write data "Hello "; fin_flag = 1.
     2. Write data "World!"; fin_flag = 1.
   - Messages received by client:
     1. "Hello"
     2. "World!"
3. Not setting the fin_flag for any chunk of data:
   - Data sent on server:
     1. Write data "Hello "; fin_flag = 0.
     2. Write data "World!"; fin_flag = 0.
   - Messages received by client:
     1. NONE - data are still in server buffer, because there was no fin_flag set.

5.11.4 Receiving data from WebSocket

Callback on_message with parameter of type WS_USER_CONTEXT_STRUCT is invoked to process them when data is received. Information about data like the number of received bytes, type of data and pointer is stored in the data substructure of they type WS_DATA_STRUCT.

5.11.5 WebSocket error handling

In case an error occurs during a communication with the client, callback on_error is invoked with a parameter of type WS_USER_CONTEXT_STRUCT. In this structure, value of error variable represents a type of the error. This is a description of error codes:

- WS_ERR_OK: No error occurred.
- WS_ERR_SOCKET: Client terminated the connection without a proper close handshake.
- WS_ERR_BAD_FRAME: Bad frame received (wrong close reason, reserved field has invalid value etc).
- WS_ERR_BAD_OPCODE: Frame is valid, but wrong (unknown) frame opcode is set.
- WS_ERR_BAD_SEQ: Wrong frame sequence received (data frame between continuation frames; continuation frame without previous data frame etc).
- WS_ERR_NO_UTF8: Received data type is set to text data, but data is not valid UTF-8 sequence.
- WS_ERR_SERVER: Server error; server ran out of memory, and the server is unable to create tasks etc.

All errors are fatal and connection is closed automatically by the server after on_error callback processing is done.

5.11.6 Closing WebSocket connection

To close WebSocket, call the WS_close() function. You do not have to close the WebSocket in case of error, such cases are handled by the server automatically.

5.12 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:

1. Create RTCS as described in previous sections (**RTCS_create()**).
2. Initialize network device using **ipcfg_init_device()**.
3. Use one of the ipcfg_bind_*xxx* functions to bind the interface to an IP address, mask and gateway. IPv6 address is 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().
4. You can start the link status monitoring task (**ipcfg_task_create()**) to automatically rebind in case of Ethernet cable is reattached. Another method to handle this monitoring is to call **ipcfg_task_poll()** periodically in an existing task.
5. You can acquire bind information using various iocfg_get_xxx functions.

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

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

IPCFG functionality is affected by these definitions:

- RTCSCFG_ENABLE_GATEWAYS: Must be set to nonzero to enable reaching devices behind gateways within the network. IPCFG ignores all gateway-related data without this feature.
- RTCSCFG_IPCFG_ENABLE_DNS: Must be set to nonzero to enable DNS name resolving in IPCFG.
- RTCSCFG_IPCFG_ENABLE_DHCP: Must be set to nonzero to enable DHCP binding in IPCFG. Note that DHCP also depends on *RTCSCFG_ENABLE_UDP*.
- RTCSCFG_IPCFG_ENABLE_BOOT: Must be set to nonzero to enable TFTP names processing and BOOT binding

5.13 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, such as 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. 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.

These are the usage procedures of IWCFG:

1. Create RTCS as described in previous sections (**RTCS_create()**).
2. Initialize network device using **ipcfg_init_device()** .
3. Initialize wifi device using these 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.

5.14 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. The MQX RTOS implementation supports both IPv4 and IPv6 protocol.

5.14.1 Sending an email

Only the SMTP_send_email function must be called to send an email. A structure of data type SMTP_PARAM_STRUCT structure must be set up and passed to the function as first parameter before calling. 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 see the reference for these functions and data types:

- SMTP_send_email()
- SMTP_EMAIL_ENVELOPE
- SMTP_PARAM_STRUCT

5.14.2 Example application

There is an 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 an email with authentication from the RTCS shell.

5.15 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 RTOS SNMPv1 Agent conforms to these RFCs:

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

The Freescale MQX RTOS SNMPv2c Agent is based on these RFCs:

- RFC 1905
- RFC 1906

5.15.1 Compile time configuration

A few macros are used for setting SNMP agent default configuration during compile time. Default values for 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 mqx_sdk_config.h of your project.

- RTCSCFG_SNMPSRV_SERVER_PRIO: Default priority of agent task. This value is used when the SNMP agent creates its background task. The value can be overridden by setting server_prio member of the server initialization structure to a nonzero value. The value of this macro is set to priority of RTCS TCP/IP task decreased by 1 by default.
- RTCSCFG_SNMPSRV_BUFFER_SIZE: Buffer size used for incoming and outgoing packets in bytes. Must be at least 494 bytes. Default value is 1KiB.

5.15.2 Defining Management Information Base (MIB)

The MIB database objects, or nodes, are defined in user application as a tree of RTCSMIB_NODE structures. The structure 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 constant and put into read only memory. MIB tree of all parent nodes for each object is required, because node IDs are used to create OID send in SNMP messages.

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 like this:

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

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

Writable objects

For each variable object which is writable, provide the MIB_set_objectname() function, where objectname is the name of the variable object.

```
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 — The length of the value in bytes.

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

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

Example:

We want to use ipForwarding (OID 1.3.6.1.2.1.4.1) in our application. This OID equals: iso(1).org(3).dod(6).internet(1).mgmt(2).mib-2(1).ip(4).ipForwarding(1) Following nodes must be present in MIB tree with correct pointers to parent and child nodes:

```
RTCSMIB_NODE MIBNODE_iso_org
RTCSMIB_NODE MIBNODE_dod
RTCSMIB_NODE MIBNODE_internet
RTCSMIB_NODE MIBNODE_mgmt
RTCSMIB_NODE MIBNODE_mib2
RTCSMIB_NODE MIBNODE_ip
RTCSMIB_NODE MIBNODE_ipForwarding
```

The leaf node (MIBNODE_ipForwarding) will then look like this:

```
const RTCSMIB_NODE MIBNODE_ipForwarding = {
    1,
    NULL,
    NULL,
    (RTCSMIB_NODE *)&MIBNODE_ip,
    RTCSMIB_ACCESS_READ,
    NULL,
    MIB_instance_zero,
    ASN1_TYPE_OCTET,
    (RTCSMIB_VALUE *)&MIBVALUE_ipForwarding,
    MIB_set_ipforwarding
};
```

content of MIBVALUE_ipForwarding:

```
const RTCSMIB_VALUE MIBVALUE_ipForwarding = {
    RTCSMIB_NODETYPE_INT_FN,
    (void *)MIB_get_ipforwarding
};
```

functions for setting the value (Valid ipForwarding values are described in RFC1213):

```
uint32_t MIB_set_ipforwarding(void *dummy, unsigned char *varptr, uint32_t varlen)
{
    int32_t varval = MIB_int_read(varptr, varlen);

    if (varval == 1)
    {
        _IP_forward = TRUE;
    }
    else if (varval == 2)
    {
        _IP_forward = FALSE;
    }
    else
    {
        return(SNMP_ERROR_wrongValue);
    }
    return(SNMP_ERROR_noError);
}

int32_t MIB_int_read(uint8_t *varptr, uint32_t varlen)
{
    int32_t varval = 0;

    if (varlen)
    {
        varval = *varptr++;
        varlen--;
    }
    while (varlen--)
    {
        varval <<= 8;
        varval += *varptr++;
    }
    return(varval);
}
```

and finally function for reading of IP forwarding value in RTCS:

```
int32_t MIB_get_ipforwarding(void *dummy)
{
    if (_IP_forward)
    {
        return(1); /* return 1 - Forwarding enabled */
    }
    else
    {
```

```
        return(2); /* return 2 - Forwarding disabled */
    }
}
```

Complete example code with whole MIB tree can be found in file sh_snmpsrv.c in folder %MQX_PATH%\nshell\source\rtcs.

5.15.3 Sending a trap message to the client application

To send a trap message to the client, use these functions:

- SNMPv2_trap_userSpec() function - use this function to create a valid node RTCSMIB_NODE in the MIB tree. Value of this node is sent to the client with an OID defined by its position in the MIB tree.
- SNMP_trap_linkDown() or SNMP_trap_linkUp - these functions are used for the communication device descriptor. This can be arbitrary value, but it must be unique for all devices present in system. The function is sent in a trap message as ifIndex (OID 1.3.6.1.2.1.2.2.1.1).
- SNMPv2_trap_coldStart(), SNMPv2_trap_warmStart(), and SNMPv2_trap_authenticationFailure functions are used to send a trap message. Before sending a trap message, create an SNMP agent by using the function SNMP_init().

The trap sending process is demonstrated in the SNMP example.

5.15.4 Basic Usage

Follow these steps to start the SNMP agent:

1. Create MIB tree and store pointer to it in RTCSMIB_NODE. See chapter Defining Management Information Base for further details.
2. Create and fill rest of params structure with required server settings. All parameters, but MIB tree pointer are optional. You can set any parameter to zero/NULL and the server will use a default value.
3. Start the server using function SNMP_init() with a parameter created in previous step.

These steps are demonstrated by an example which you can find in the %MQX_PATH%\nshell\source\rtcs\sh_snmpd.c file.

5.16 SNTP (Simple Network Time Protocol) Client

RTCS provides an SNTP Client 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 5-7. Summary: SNTP Client services

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

5.17 Telnet Client

The Telnet Client implements a client that complies with the Telnet protocol specification, RFC 854. A Telnet connection establishes a network virtual terminal between the two computers with dissimilar character sets. The server host provides a service to the user host that initiated the communication. Both IPv4 and IPv6 are supported.

5.17.1 Connecting and disconnecting to Telnet server

To connect to the remote host with the Telnet client, use the TELNETCLN_connect() function. This function takes one parameter params TELNETCLN_PARAM_STRUCT. The key parameters of the TELNETCLN_PARAM_STRUCT are:

- sa_remote_host - sockaddr structure describing remote host. The content of this structure must be initialized by the getaddrinfo() function. It contains information such as the remote IP address, port and, address family.
- fd_in - A pointer (descriptor) to a file used as input for the Telnet client. The Telnet client reads the characters and sends them to the server until the EOF is encountered.
- fd_out - A pointer (descriptor) to a file used as an output for the Telnet client. The Telnet client reads the characters from the remote host and writes them to this file until the connection is closed.
- callbacks - A TELNETCLN_CALLBACKS_STRUCT structure containing pointers to functions which are invoked for various events. See the description of this structure type for more details.

If the connection was successful, the TELNETCLN_connect() function returns a non-zero number (handle). This handle is used as a parameter for other API functions.

To check whether the client is connected, use the TELNETCLN_get_status() function with the handle created by the TELNETCLN_connect() function as a parameter. The returned value can equal either to the TELNETCLN_STATUS_STOPPED or to the TELNETCLN_STATUS_RUNNING, depending on the client state.

To disconnect the client from the server, call the TELNETCLN_disconnect() function. This function has one parameter which is the client handle. If the disconnect is successful, the returned value equals RTCS_OK. Otherwise, it is set to the RTCS_ERROR.

5.18 Telnet Server

Telnet is a network protocol used on the Internet or local area networks to provide a bidirectional interactive text-oriented communication facility using a virtual terminal connection. User data is interspersed in-band with Telnet control information in an 8-bit byte oriented data connection over the Transmission Control Protocol (TCP). Telnet protocol is standardized in RFC 854 (https://tools.ietf.org/html/rfc854).

5.18.1 Compile time configuration

A few macros are used for setting telnet 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.

- RTCSCFG_TELNETSRV_SERVER_PRIO — default priority of server tasks. This value is used when the telnet 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 priority of TCP/IP task lowered by 1.
- RTCSCFG_TELNETSRV_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. Value of this parameter can be overridden by setting the max_ses member of the server initialization structure. Default value is 2 sessions.

- RTCSCFG_TELNETSRV_TX_BUFFER_SIZE size of the socket transmit buffer in bytes. This option cannot be overridden at runtime. Default value is 1460 bytes.
- RTCSCFG_TELNETSRV_RX_BUFFER_SIZE size of the socket receive buffer in bytes. This option cannot be overridden at runtime. Default value is 1460 bytes.
- RTCSCFG_TELNETSRV_TIMEWAIT_TIMEOUT — specifies how long will server socket stay in TIME-WAIT state. Default value is 1000 ms.
- RTCSCFG_TELNETSRV_SEND_TIMEOUT - timeout value for server sockets in miliseconds. Default value is 5000ms.
- RTCSCFG_TELNETSRV_CONNECT_TIMEOUT - hard timeout for establishing a connection in miliseconds for telnet server sockets. Cannot be changed during runtime. Default value is 1000ms.
- RTCSCFG_TELNETSRV_USE_WELCOME_STRINGS - macro defining if welcome and goodbye messages should be sent to the client when he connects/ disconnects.

5.18.2 Basic Usage

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

1. Create and fill structure of type TELNETSRV_PARAM_STRUCT with required serversettings. All parameters, but shell and shell commands are optional. You can set anyparameter to zero/NULL and the server will use a default value.
2. Start the server using function TELNETSRV_init() with a parameter created in the previous step. Both of these steps are demonstrated by an example which you can find in the %MQX_PATH%\shell\source\rtcs\sh_telnetsrv.c file. The server parameters structure description can be found in TELNETSRV_PARAM_STRUCT.

5.19 TFTP Client

The RTCS implementation of the TFTP client supports both IPv4 and IPv6 remote hosts and allows two operations:

1. Downloading a file from the server to the local filesystem.
2. Uploading a local file to the TFTP server.

Compile time configuration

The TFTP client application has no compile time configuration.

Basic Usage

First, the TFTP client must be initialized. This is done by calling the TFTPCLN_connect() function. The parameter for this function is the TFTPCLN_PARAM_STRUCT which has these variables:

- sockaddr sa_remote_host - This parameter describes the TFTP server. This is the only parameter which is required. The user can obtain this structure by using the getaddrinfo function.
- TELNETCLN_DATA_CALLBACK recv_callback - This function is invoked whenever data is received from the server. It has one parameter which is the number of received bytes of data.
- TELNETCLN_DATA_CALLBACK send_callback - This function is invoked whenever data is sent to the server. It has one parameter which is the number of sent bytes of data.
- TELNETCLN_ERROR_CALLBACK error_callback - This function is invoked when an error occurs during the data transfer. It has two parameters: the error code and the string describing the error.

If the initialization is successful, a task is created to handle the GET/PUT requests and the TFTP timeouts and returned value is non-zero.

To download a file, call the TFTPCLN_get() function. This function downloads a file and then returns. If the user provides a receive callback in the TFTP initialization, this callback is invoked each time a data packet is received. This function has three parameters, a handle created by the TFTPCLN_connect() function, a local filename, and a remote filename. For details about the TFTPCLN_get() function, see the function description in chapter 7.

To upload a file to the TFTP server, use the TFTPCLN_put() function. This function uploads a file and then returns. If the user provides a sent callback in the TFTP initialization, this callback is invoked each time a data packet is sent. This function has three parameters, a handle created by the TFTPCLN_connect() function, a local filename, and a remote filename. For details about the TFTPCLN_put() function, see its chapter 7.

If the user wants to send received data from/to memory instead of the file, use either io_mem, nmem, pipe, or npipe.

5.20 TFTP server

TFTP is a simple protocol to transfer files. It is implemented on top of the Internet User Datagram protocol (UDP or Datagram). It is designed to be small and easy to implement. Therefore, it lacks most of the features of a regular FTP. The only thing it can do is read and write files from/to a remote server.

5.20.1 Compile time configuration

A few macros are used for setting TFTP 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.

- RTCSCFG_TFTPSRV_SERVER_PRIO — default priority of server tasks. This value is used when the TFTP 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 priority of TCP/IP task increased by one.
- RTCSCFG_TFTPSRV_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. Value of this parameter can be overridden by setting the max_ses member of the server initialization structure. Default value is 2 sessions.

5.20.2 Basic Usage

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

1. Create and fill structure of type TFTPSRV_PARAM_STRUCT with required serversettings. All parameters, but root directory are optional. You can set anyparameter to zero/NULL and the server will use a default value.
2. Start the server using function TFTPSRV_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%\shell\source\rtcs\sh_tftpsrv.c file. The server parameters structure description can be found in TFTPSRV_PARAM_STRUCT.

5.21 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.

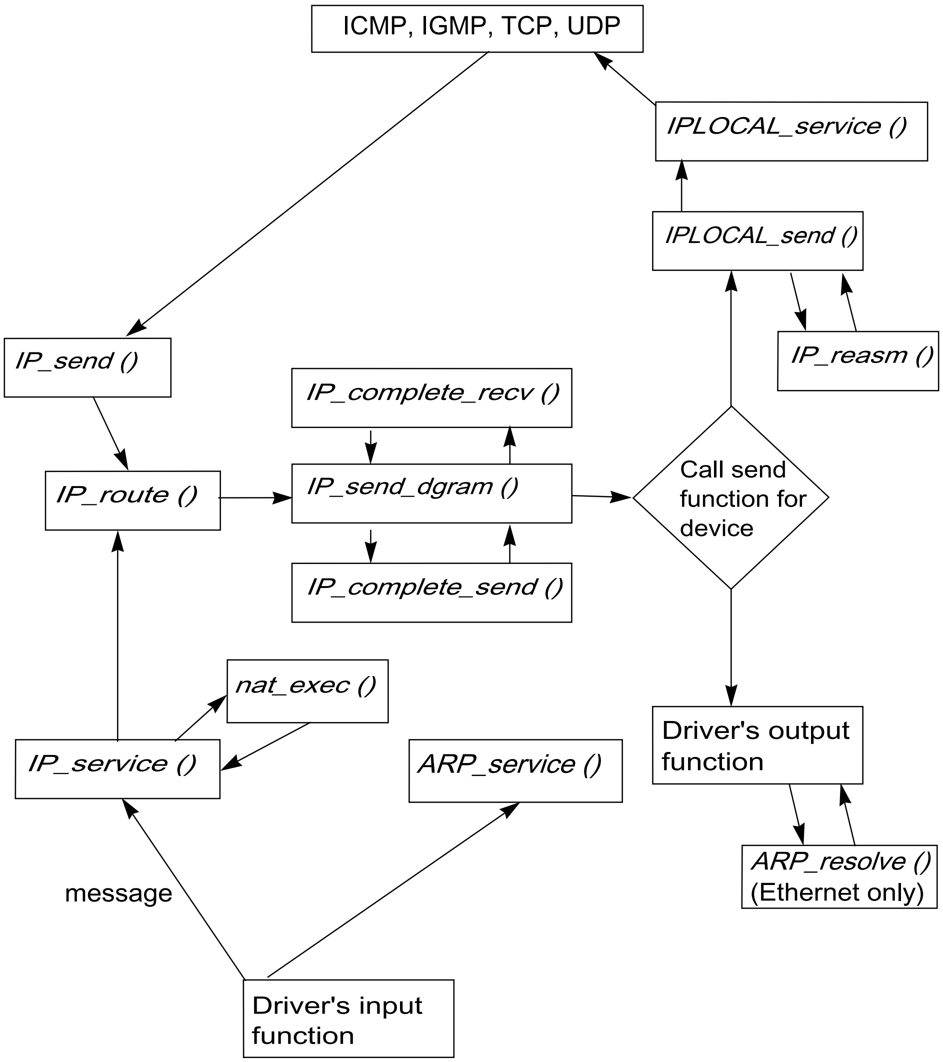

Figure 5-1. Typical RTCS packet-processing paths

Receiving IPoE packets

6 bytes	6 bytes	2 bytes	46 - 1500 bytes	4 bytes
Destination address	Source address	Type	Data	FCS

Figure 5-2. Typical RTCS IPoE code path for receiving data

Sending IPoE packets

6 bytes	6 bytes	2 bytes	46 - 1500 bytes	4 bytes
Destination address	Source address	Type	Data	FCS

Figure 5-3. Typical RTCS IPoE code path for sending data

Chapter 6
Rebuilding

6.1 Reasons to rebuild RTCS

The RTCS needs to be rebuilt if any of these are done:

- Change compiler options, such as optimization level.
- Change RTCS compile-time configuration options.
- Incorporate changes that were made to RTCS source code

6.2 Before you begin

Before rebuilding the RTCS, it is recommended:

- To see the MQX RTOS User's Guide, a document for MQX RTOS rebuild instructions. A very similar concept applies also to the RTCS.
- To see the MQX RTOS Release Notes that accompany Freescale MQX RTOS to get information \ 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 rebuild instructions, as they are described in the Release Notes document, and also the instructions provided in the following sections.

6.3 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, CodeWarrior, for example, 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.

6.3.1 Post-build processing

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 RTOS applications, which need to use the RTCS services, do not need to make any reference to the RTCS source tree.

6.3.2 Build targets

The CodeWarrior development environment allows for multiple build configurations, otherwise known as build targets. All projects in the Freescale MQX RTOS RTCS contain at least two build targets:

- Debug Target — Compiler optimizations are set low to enable easy debugging.
- 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.

6.4 Rebuilding Freescale MQX RTCS

Rebuilding the MQX RTOS RTCS library is a simple task which involves opening the proper build project in the development environment and building it. Do not forget to select the proper build target or to build all targets.

For specific information about rebuilding MQX RTOS RTCS and the example applications, see the Release Notes that accompany the Freescale MQX RTOS 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 these:

- *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.

7.2 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, and 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. If the parent socket (listening) is closed by a call to the closesocket() from another task, while accept() is blocked, accept() returns -1 (RTCS_SOCKET_ERROR) and the RTCS_errno is set to the RTCSERR_SOCK_CLOSED. If the parent socket (listening) is shut down by a call to the shutdownsocket() from another task, accept() returns -1 (RTCS_SOCKET_ERROR) and the RTCS_errno is set to the RTCSERR_SOCK_ESHUTDOWN.

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
    {
        uint32_t rtcserrno = RTCS_get_errno();
        /* The value will be RTCSERR_SOCK_CLOSED if closesocket()
         * has been called (from other task).
         * After closesocket(), the socket cannot be used.
         */
        if (rtcserrno == RTCSERR_SOCK_CLOSED)
        {
            service_closed_listensock();
        }
        else
        {
            status = RTCS_geterror(handle);
            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)
    {
        uint32_t rtcserrno = RTCS_get_errno();
        /* The value will be RTCSERR_SOCK_CLOSED if closesocket()
         * has been called (from other task).
         * After closesocket(), the socket cannot be used.
         */
        if (rtcserrno == RTCSERR_SOCK_CLOSED)
        {
            service_closed_listensock();
        }
    }
    else
    {
        status = RTCS_geterror(sock6);
        printf("Error, accept() failed with error code %lx",status);
        _task_block();
    }
}
```

7.3 ARP_stats()

Gets a pointer to the ARP statistics that RTCS collectsa 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).
- Zero (failure: rtcs_if_handl*e* 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);
}
```

7.4 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();
}
```

7.5 closesocket()

Closes the socket. After the closesocket() returns, the application can no longer use the socket.

Synopsis

```
int32_t closesocket(uint32_t sock)
```

Parameters

- *sock [in]* – socket handle

Description

Datagram socket is closed immediately, outstanding calls to recvfrom() return immediately, queued incoming packets are discarded. Unconnected stream socket is closed immediately. For a connected stream, the behavior depends on the SO_LINGER socket option:

Table 7-1. Connected stream closesocket() behavior based on SO_LINGER socket option

SO_LINGER	(Default)			
l_onoff	0	1	1	1
l_linger_ms	Don't care	0	>0 milliseconds	>0 milliseconds
4-way graceful close			Complete before l_linger_ms timeout expires.	Does not complete within l_linger_ms time.
Close type	Graceful	Abort (reset)	Graceful	Abort (reset)
Returns when	Immediately (non-blocking)	Immediately (non-blocking)	After 4-way graceful close completes (blocking)	After l_linger_ms expires (blocking).

If the closesocket() function is called on a socket, calls to send() and recv() functions return immediately.

When linger is off (l_onoff = 0), the TIMEWAIT_TIMEOUT timer starts with application call to the closesocket() function. If the graceful connection close does not complete within the TIMEWAIT_TIMEOUT, the TCB for the socket is closed and reset is sent to the remote host. When linger is on (l_onoff != 0), l_linger_ms timer starts with the application call to the closesocket() function. If the graceful connection close does not complete within the l_linger_ms, the TCB for the socket is closed and reset is sent to the remote host. If l_linger_ms is zero, the (TCB close and reset) occurs immediately. If RTCS is sending the FIN first, the TCB for the socket is maintained for the TIMEWAIT_TIMEOUT time when the connection is closed gracefully (TCP connection in the TIME_WAIT state).

By default, RTCS configures the TIMEWAIT_TIMEOUT to 2 seconds (DEFAULT_TIMEWAIT_TIMEOUT compile time macro). Use the socket option TIMEWAIT_TIMEOUT for an application that requires the TCP connections in TIME_WAIT state to be kept for a longer time period.

Mapping to the RTCS legacy API, (shutdown() /RTCS_shutdown()):
shutdown(FLAG_CLOSE_TX) or shutdown(0) correspond to the graceful close process and shutdown(FLAG_ABORT_CONNECTION) corresponds to the abort (reset) close.

Return value

- Zero (Success – RTCS_OK)
- Non-zero (Failure – specific error code)

See also

- shutdownsocket()
- setsockopt()

Example

```
closesocket(clientsock); /* default Non-blocking graceful close */
```

7.6 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, or with 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);
```

7.7 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).
- Zero (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);
```

7.8 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().

7.9 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, or 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().

7.10 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, or 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_int32()
- DHCP_option_string()
- DHCP_option_variable()

Example

See DHCPSRV_init().

7.11 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, or 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().

7.12 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, or 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().

7.13 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, or 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().

7.14 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().

7.15 DHCPCLN6_init()

This function starts DHCPv6 client.

Synopsis

```
uint32_t DHCPCLN6_init(
DHCPCLN6_PARAM_STRUCT *params);
```

Parameters

params [in] - client parameters.

Description

Use this function to start DHCPv6 client. This function blocks until startup is complete.

Return Value

- Client handle if initialization was successful, zero otherwise.

See also

- DHCPCLN6_release()
- DHCPCLN6_PARAM_STRUCT

Example

```
/*
* Start DHCPv6 client on interface no.0 with link checking enabled. Wait
* 5 seconds and then stop it.
*/
uint32_t             handle;
char                 *result_s;
DHCPCLN6_PARAM_STRUCT params = {0};
uint32_t              result;

params.flags |= DHCPCLN6_FLAG_CHECK_LINK;
params.interface = RTCS_if_get_handle(0);
handle = DHCPCLN6_init(params);
fprintf(stdout, "DHCPv6 initialization %s.\n", (handle == 0) ? "failed" : "successful");
if (handle != 0)
{
   uint32_t i;
```

```
    /* Wait 15 seconds for address. */
    for(i = 0; i < 15; i++)
    {
        if (DHCPCLN6_get_status(dhcp6_handle) == DHCPCLN6_STATUS_BOUND)
        {
           printf("Address from DHCPv6 server obtained.\n");
           break;
        }
        _time_delay(1000);
    }

    if (i == 15)
    {
        printf("Failed to obtain address from DHCPv6 server!\n");
    }
    _time_delay(5000);
    result = DHCPCLN6_release(handle);
    fprintf(stdout, "DHCPv6 release %s.\n", (result == RTCS_OK) ? "successful" : "failed");
}
```

7.16 DHCPCLN6_release()

This functions stops DHCPv6 client.

Synopsis

```
uint32_t DHCPCLN6_release(
uint32_t handle);
```

Parameters

handle [in] - handle to DHCPv6 client created by function DHCPCLN6_init().

Description

Use this function to stop DHCPv6 client. As a result of this function, all addresses acquired by client are released. This function blocks, until release is done.

Return Value

- RTCS_OK if release was successful, RTCS_ERROR otherwise.

Example

- See example for DHCPCLN6_init()

See also

- DHCPCLN6_init()

7.17 DHCPCLN6_get_status()

This is used when status of client is needed by application.

Synopsis

```
DHCPCLN6_STATUS DHCPCLN6_get_status(
uint32_t handle);
```

Parameters

handle [in] - handle to DHCPv6 client created by function DHCPCLN6_init().

Description

Use this function to read current status of DHCPv6 client. Return value indicates if there are some addresses assigned by client and if client is running.

Return Value

- Status of client from DHCPCLN6_STATUS enum.

Example

- See example for DHCPCLN6_init()

See also

- DHCPCLN6_init()
- DHCPCLN6_STATUS

7.18 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).
- Zero (no option of the specified type exists).

See Also

- DHCP_find_option()

7.19 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.

Functions of the DHCPCLNT_release():

- 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);
```

7.20 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);
```

7.21 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()
- DHCP_DATA_STRUCT

Example

See DHCPSRV_init()

7.22 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()

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);
    }
```

7.23 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

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);
       }
```

7.24 ECHOCLN_connect

Connects to the RFC 862 ECHO server.

Synopsis

```
uint32_t     ECHOCLN_connect (const struct addrinfo *addrinfo_ptr)
```

Parameters

- *addrinfo_ptr [in]* – pointer to struct addrinfo of remote host

Description

Try to connect to a server on remote host. Remote host is specified by struct addrinfo, that can be obtained for example from getaddrinfo().

Return value

- RTCS_SOCKET_ERROR (failure)
- socket handle (success).

See also

- ECHOCLN_process
- getaddrinfo()

Example

```
int32_t i_result;
uint32_t sock = RTCS_SOCKET_ERROR;
struct addrinfo *result = NULL,
 *ptr = NULL,
 hints;

memset( &hints, 0, sizeof(hints) );
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;

i_result = getaddrinfo("192.168.1.202", "7", &hints, &result);
if ( i_result != 0 )
{
    fprintf(stdout, "getaddrinfo failed with error: %d\n", i_result);
    goto exit;
}

for (ptr=result; ptr != NULL ;ptr=ptr->ai_next)
```

```
{
    sock = ECHOCLN_connect(ptr);
    if (sock == RTCS_SOCKET_ERROR)
    {
        continue;
    }
    break;
}
freeaddrinfo(result);

if (sock == RTCS_SOCKET_ERROR)
{
    fprintf(stdout, "Unable to connect to server!\n");
    goto exit;
}
```

7.25 ECHOCLN_process

Send echo data to a server and receive echo back.

Synopsis

```
int32_t ECHOCLN_process(uint32_t sock,
        char  *buffer,
        uint32_t   buflen,
        int32_t      count,
        TIME_STRUCT_PTR  time_ptr)
```

Parameters

- sock [in] – connected socket handle
- buffer [in] – pointer to data for sending
- buflen[in] – length of send data in bytes
- count [in] – number of echo loops. If negative or zero, RTCSCFG_ECHOCLN_DEFAULT_LOOPCNT applies.
- time_ptr [in/out] Non zero value on input means the function will measure time duration of echo loops, specified by count argument, and resulting time will be written to TIME_STRUCT pointed by time_ptr. If given as NULL, time is not measured.

Description

One echo loop consists of: given data buffer is sent to the given sock. Then, inbound data is received from the socket and received data is compared with the send data. This loop is repeated count times. Return value

Return values

- RTCSOK (Success).
- ECHOCLN_ERR_SOCKET (send() or recv() error)
- ECHOCLN_ERR_DATA_COMPARE_FAIL (received and send data are different)

- ECHOCLN_ERR_OUT_OF_MEMORY (failed to allocated memory for receive data)
- ECHOCLN_ERR_INVALID_PARAM (input parameters are invalid)

See also

- ECHOCLN_connect
- RTCSCFG_ECHOCLN_DEFAULT_BUFLEN
- RTCSCFG_ECHOCLN_DEFAULT_LOOPCNT
- RTCSCFG_ECHOCLN_DEBUG_MESSAGES

Example

```
int32_t i_result;
TIME_STRUCT diff_time;

i_result = ECHOCLN_process(sock, buffer, buflen, loop_cnt, diff_time);
if (RTCS_OK != i_result)
{
   fprintf(stdout, "ECHO client error %i after run time %i.%i\n", i_result,
diff_time.SECONDS, diff_time.MILLISECONDS);
}
else
{
    fprintf(stdout, "exchanged %d echo packets, total time: %i.%i s\n", loop_cnt,
diff_time.SECONDS, diff_time.MILLISECONDS);
}
closesocket(sock);
```

7.26 ECHOSRV_init()

Starts RFC 862 Echo Server. This function receives data and sends it back to the sender. One listening stream socket and one datagram socket are created per enabled IP family during service initialization. All possible combinations are supported, such as IPv4 only, IPv6 only and a dual IPv4+IPv6.

Synopsis

```
void * ECHOSRV_init(ECHOSRV_PARAM_STRUCT * params)
```

Parameters

ECHOSRV_PARAM_STRUCT * *params* — pointer to ECHOSRV_PARAM_STRUCT.

Description

ECHOSRV is fully configured with general RTCS build/runtime options, ECHOSRV specific build time options and ECHOSRV_PARAM_STRUCT configuration structure. Before calling the function, create and fill ECHOSRV_PARAM_STRUCT, which can exist on an application task's stack, or can be a global variable. After the ECHOSRV_init() function returns a nonzero value, the ECHOSRV_PARAM_STRUCT structure becomes obsolete.

Return value

When an invalid input parameter is provided, this function returns a value zero (0). Task error code is set appropriately in this case. Otherwise, a valid, nonzero, pointer to a new ECHOSRV instance is returned. This value should be stored by the application so that it can be used later in the ECHOSRV_release() function.

See also

- ECHOSRV_release()
- ECHOSRV_PARAM_STRUCT
- ECHOSRV build time options (2.16)

Example

```
#include "echosrv.h"
void * echosrv_ptr;
ECHOSRV_PARAM_STRUCT params = {
     AF_INET | AF_INET6, /* for IPv4+IPv6 */
     7, /* service runs on port 7 by default */
   #if RTCSCFG_ENABLE_IP4
     INADDR_ANY, /* Listening IPv4 address */
   #endif
   #if RTCSCFG_ENABLE_IP6
     IN6ADDR_ANY_INIT, /* Listening IPv6 address */
     0, /* Scope ID for IPv6. 0 is for any Interface. */
   #endif
    7 /* priority of ECHOSRV task */
   };

echosrv_ptr = ECHOSRV_init(&params);
```

7.27 ECHOSRV_release()

Stops RFC 862 Echo Server.

Synopsis

uint32_t ECHOSRV_release(void * *server_h*)

Parameters

void * *server_h* - Pointer to an instance of ECHOSRV. This is the return value from ECHOSRV_init().

Description

This function shuts down all listening sockets, shuts down all client sockets, frees all memory resources and destroys the ECHOSRV_task.

Return value

This function returns RTCS_OK if successful. The return value equals

RTCSERR_SERVER_NOT_RUNNING if the task, specified by the *server_h* input parameter, does not exist in the system.

See also

ECHOSRV_init()

ECHOSRV_PARAM_STRUCT

ECHOSRV build time options (2.16)

Example

```
#include "echosrv.h"
uint32_t retval;
retval = ECHOSRV_release(echosrv_ptr);
```

7.28 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()
- RTCS_if_add()
- ENET_STATS

Example

```
ENET_STATS_PTR  enet;
_enet_handle    ehandle;
...
enet = ENET_get_stats();
printf("\n%d Ethernet packets received", enet->ST_RX_TOTAL);
```

7.29 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:

nonzero (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 Example: setting up RTCS.

7.30 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)

Example

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)

7.31 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 one of these:

- 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)

7.32 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)

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 */
```

7.33 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. See chapter "FTPSRV_PARAM_STRUCT" for further description of each server parameter.

Return Value

- Nonzero 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

7.34 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. The 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()

7.35 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, which is 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 (0).

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 zero-terminated strings or the zero 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 nonzero.

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 these 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 zero-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 zero pointer, 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.

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.

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");
    }
 }
```

7.36 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).

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");
    }
 }
```

7.37 getnameinfo()

Provides name resolution from an address to a name.

Synopsis

```
int32_t getnameinfo( const struct sockaddr *sa, unsigned int salen, char *host, unsigned int
hostlen, char *serv, unsigned int servlen, int flags)
```

Parameters

sa [in] — Pointer to a socket address structure to be translated. It holds the address and port number.

salen [in] — Length of the socket address structure pointed by sa, in bytes.

host [out] — Pointer to a string buffer to hold the return host name. It is optional (zero).

hostlen [in] — Length of the string buffer pointed by host, in bytes, including terminating the zero character.

serv [out] — Pointer to a string buffer to hold the return port number. It is optional (zero).

servlen [in] — Length of the string buffer pointed by serv, in bytes, including terminating the zero character.

flags [in] — Flag argument that modifies behavior of the getnameinfo() function.

Return Value

- Zero for success, or nonzero if error occurs.

Description

This function is used to translate a socket address to a host name and port number.

The host parameter points to a buffer able to contain up to hostlen characters that receives the host name as a zero terminated string if the host parameter is nonzero and the hostlen argument is nonzero. If the host argument is zero or the hostlen argument is zero, the host name should not be returned. The numeric form of the address contained in the socket address structure pointed to by the sa argument is returned instead of its name, if the host's name cannot be located.

The service argument points to a buffer able to contain up to servlen bytes that receives the port number as a zero-terminated string if the serv parameter is nonzero and the servlen parameter is nonzero. The port number string is not returned if the serv argument is zero or the servlen parameter is zero.

The flags parameter modifies the behavior of the function:

- NI_NOFQDN — If set, return only the hostname part of the FQDN (Fully Qualified Domain Name).
- NI_NUMERICHOST — If set, then the numeric form of the hostname is returned. When not set, this will still happen in case the node's name cannot be determined. The function allows numeric IPv6 address notation with scope identifier.
- NI_NAMEREQD — If set, then an error is returned if the hostname cannot be determined.

The getnameinfo() function finds the inverse of getaddrinfo(), and replaces the functionality of obsolete gethostbyaddr().

Example

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

   _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)
           {
                /* Print numeric form of the address.*/
             if(getnameinfo(addrinfo_result->ai_addr,
                           addrinfo_result->ai_addrlen,
                               host_str, sizeof(host_str),


                               NULL, 0, NI_NUMERICHOST) == 0)
             {
                 printf("\t%s\n", addr_str);
             }
             addrinfo_result = addrinfo_result->ai_next;
          }

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

7.38 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)

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);
}
```

7.39 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)

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);
}
```

7.40 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)

7.41 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 zero — defaults are used in that case. Any parameter set to zero 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_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);
```

7.42 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 the server. This function blocks until shutdown is finished.

Return Value

- HTTPSRV_OK if shutdown was successful, HTTPSRV_ERR otherwise.

7.43 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.

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 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_RES_STRUCT

Example

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

7.44 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] — number of bytes to read.

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

See file *%MQX_PATH%\demo\web_hvac\cgi_hvac.c* (you can copy link and paste it to the file explorer address bar) for a detailed example of how to use this function. This function should have a return value of parameter "length". If its return value is lower, an error occured during read from the socket and you should not call this function again with same session handle in the same context.

7.45 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;
    }
```

7.46 LLMNRSRV_init

Starts the Link-Local Multicast Name Resolution (LLMNR) server.

Synopsis

```
uint32_t LLMNRSRV_init(LLMNRSRV_PARAM_STRUCT *params)
```

Parameters

- *params[in]* – initialization parameters of the LLMNR server.

Description

The function starts the LLMNR server according to initialization parameter structure. Network interface and host name table are mandatory parameters in this structure, all others are optional and may be set to zero. See LLMNRSRV_PARAM_STRUCT for a description of each server parameter.

Return Value

- Server handle if initialization was successful, zero otherwise.

Example

```
LLMNRSRV_PARAM_STRUCT     llmnr_params;
LLMNRSRV_HOST_NAME_STRUCT host_name_table[] ={ {"rtcs",0},
                                               {0,0} /*end mark*/}
_rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);

_mem_zero(&llmnr_params, sizeof(llmnr_params));
llmnr_params.interface = ihandle;
llmnr_params.host_name_table = host_name_table;

/* Start LLMNR server.*/
llmnr_desc = LLMNRSRV_init(&llmnr_params;
```

See Also

- LLMNRSRV_release
- LLMNRSRV_PARAM_STRUCT

7.47 LLMNRSRV_release

Stops the LLMNR server and releases all of its resources.

Synopsis

```
uint32_t LLMNRSRV_release(uint32_t server_h)
```

Parameters

- *server_h[in]* - server handle (returned by LLMNRSRV_init).

Description

This function does the opposite of the LLMNRSRV_init(). It stops the server task and frees all resources allocated by the server. The calling task is blocked until the server stops and resources are released.

Return Value

- RTCS_OK

See Also

- LLMNRSRV_init

7.48 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

- TCP_stats()
- ICMP_STATS

7.49 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

- TCP_stats()
- IGMP_STATS

7.50 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. These 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 size of (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

IPv4 protocol.

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

IPv6 protocol.

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

7.51 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 a value of zero if a system error occurs, or it returns a pointer to the destination string.

Example

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);
.......
```

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);
.......
```

7.52 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

- TCP_stats()
- IP_STATS

7.53 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)
- Zero (failure: rtcs_if_handle is invalid)

See Also

- TCP_stats()
- IPIF_STATS

7.54 ipcfg_init_device()

Initializes the Ethernet 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

- 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;
}
```

7.55 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

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);
```

7.56 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 a blocking function, meaning it 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

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);
}
```

7.57 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 nonblocking functions related to the DHCP binding.

ipcfg_bind_dhcp() must be called first repeatedly, until 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 a result other than RTCS_OK or RTCSERR_IPCFG_BUSY. The network interface is left in unbound state in this case.

An alternative blocking method of DHCP bind is ipcfg_bind_dhcp_wait(). See this example for 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()

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;
}
```

7.58 ipcfg_bind_dhcp_wait()

Binds Ethernet device to network using a DHCP protocol, or 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 an auto IP bind if DHCP fails. It is a blocking function, which means it does not 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 zero, 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);
}
```

7.59 ipcfg_bind_staticip()

Binds Ethernet device to network using constant, or 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 an unbound state.

Return Value

- IPCFG_OK (success)
- RTCSERR_IPCFG_BUSY
- RTCSERR_IPCFG_DEVICE_NUMBER
- RTCSERR_IPCFG_INIT
- RTCSERR_IPCFG_BIND

7.60 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()

7.61 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()

7.62 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, otherwise the value is zero.

7.63 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.

7.64 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()

7.65 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 zero.

See Also

- ipcfg_get_state()
- ipcfg_get_desired_state()

7.66 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()

7.67 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()

7.68 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()

7.69 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_del_dns_ip()

7.70 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_add_dns_ip()

7.71 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.

7.72 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()

7.73 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()

7.74 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()

7.75 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()

7.76 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

The 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();
```

```
    }
}
```

7.77 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().

7.78 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().

7.79 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 does not 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().

7.80 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 a blocking function ,which means it does not 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_dhcp()

Example

```
void main(uint32_t param)
{
  setup_network();

  ...

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

7.81 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. An error is returned if the address is already used. This is a blocking function, which means it does not 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

Example

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

7.82 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

Example

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

7.83 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_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);
        }
    }
}
```

7.84 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_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);
       }
    }
}
```

7.85 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");
          }
        }
    }
```

7.86 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++)
{
```

7.87 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)

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++)
   {
```

7.88 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);
```

7.89 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);
```

7.90 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);
```

7.91 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, which means a network composed of only one cell without Access Point, or Managed, which is a node that 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);
```

7.92 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);
```

7.93 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);
```

7.94 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

7.95 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);
```

7.96 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

7.97 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.

7.98 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", and "wpa2".

Return Value

- ENET_OK (success)
- ENETERR_INVALID_DEVICE

7.99 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

7.100 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.
```

7.101 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] — Hint for the RTCS to determine the maximum number of pending established connections per listening socket.

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. The second function argument, backlog, helps the RTCS to determine the maximum number of pending established connections per listening socket. An established connection is a connection that processes the three way handshake SYN, SYN/ACK, ACK. A pending established connection is an established connection for which the application has not called accept() yet. The backlog argument of less than zero creates the zero length backlog queue effectively causing all connection requests for that socket to be dropped. For the backlog argument greater than or equal to zero, backlog + 1 specifies the maximum number of pending established connections unless it is greater than RTCSCFG_SOMAXCONN in which case the RTCSCFG_SOMAXCONN specifies the maximum number of pending established connections. If the backlog queue is full and a new connection request is received for the same listening socket, the RTCS drops the connection request. A call to the accept() function makes an empty slot in the backlog queue. The length of the backlog queue for a listening socket can be changed at runtime by calling the listen() function with the listening socket handle argument and with the new value for the backlog argument. There is a system wide configuration limit for a maximum number of connections allowed (both established and half open) RTCSCFG_TCP_MAX_CONNECTIONS. A non-zero positive value limits the number of the TCBs that all listening sockets ever create in the background in response to the connection requests. The zero (default) gives no limit. If the system-wide

RTCSCFG_TCP_MAX_HALF_OPEN is set to non-zero positive value, RTCS tries to monitor the half open connections and it drops older half open connections if a new connection request is received.

Return Value

- RTCS_OK (success)
- Specific error code (failure)

See Also

- accept()
- bind()
- socket()

Example

See accept().

7.102 MIB1213_init()

Initializes the MIB-1213.

Synopsis

```
void MIB1213_init(void)
```

Description

The function installs the standard MIBs defined in RFC 1213. SNMP Agent cannot access the MIB if the function is not called.

See Also

- SNMP_init()

Example

See SNMP_init().

7.103 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

7.104 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

7.105 NAT_close()

Stops Network Address Translation.

Synopsis

```
uint32_t  NAT_close(void)
```

Return Value

- RTCS_OK (success)

See Also

- NAT_init()

7.106 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 Disabling NAT Application-Level Gateways."

You can use this function to restart Network Address Translation after you call **NAT_close()**.

Return Value

- RTCS_OK (success)
- RTCSERR_OUT_OF_MEMORY (failure)
- RTCSERR_INVALID_PARAMETER (failure)

See Also

- NAT_close()
- NAT_stats()
- nat_ports
- nat_timeouts
- NAT_STATS

7.107 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

7.108 DNAT_add_rule()

Add a DNAT rule to the DNAT rule table.

Synopsis

```
uint32_t DNAT_add_rule
   (
      uint32_t         priority,
      uint16_t         protocol,
      uint16_t         start_port,
      uint16_t         end_port,
      _ip_address     private_ip,
      uint16_t         private_port
   )
```

Parameters

priority [in] — Priority of rule

protocol [in] — Network protocol

start_port [in] — Network start port

end_port [in] — Network end port

private_ip [in] — Private-network IP address *private_port [in]* — Private-network port

Description

This function adds a DNAT rule to the DNAT rule table. Rules in the DNAT rule are administratively ordered based on priority. The priority must be unique.

Return Value

- RTCS_OK or error code

7.109 DNAT_delete_rule()

Removes a DNAT rule from the DNAT rule table.

Synopsis

```
uint32_t  DNAT_delete_rule
    (
       uint32_t  priority
    )
```

Parameters

priority [in] — Priority of rule

Description

This function removes a DNAT rule from the DNAT rule table.

Return Value

- RTCS_OK or error code

7.110 ping()

See RTCS_ping().

7.111 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

The function PPP_initialize() fails if RTCS cannot do any one of these:

- 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.
- Zero.

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.");
            }
        }
}
```

7.112 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.

7.113 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 uses.

This typically includes sending AT commands to GPRS modem and performing handshake with the machine running the Windows operating system.

Return Value

- RTCS_OK if successful.
- Error code.

See Also

- PPP_resume()

Example

See PPP_init() as an example.

7.114 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

See PPP_init() as an example.

7.115 recv()

Provides RTCS with incoming buffer.

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 to place received data.

buflen [in] — Size of buffer in bytes.

flags [in] — Flags to underlying protocols. One of these:

RTCS_MSG_PEEK — for a UDP socket. Receives a datagram but does not consume it (ignored for stream sockets).

MSG_DONTWAIT - for a stream socket. The function call is executed as non-blocking, receive, push socket option is applied.

MSG_WAITALL - for a stream socket. The function call is executed as blocking. The received TCP push flag is ignored and the recv() function returns when enough data has been received to fill the buffer. The recv() function may still return less data than requested if a timeout, an error, or a disconnect occurs.

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 successfully 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 nowait socket option is FALSE, recv() blocks until the buffer is full or the receive push socket option is satisfied.

A received TCP push flag causes recv() to return with whatever data has been received if the receive push socket option is TRUE. RTCS ignores incoming TCP push flags, and recv() returns when enough data has been received to fill the buffer if the receive push socket option is FALSE.

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);
}
```

7.116 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 these:

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: The 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

Only datagrams from that source will be received if a remote endpoint has been specified with connect().

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);
    }
```

7.117 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 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 {
   ...
```

7.118 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 Example: setting up RTCS.

7.119 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 these:

- 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().

7.120 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);
```

7.121 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)
```

7.122 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);
```

7.123 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)
```

7.124 RTCS_get_errno

Returns and clears the RTCS_errno

Synopsis

```
uint32_t     RTCS_get_errno(void)
```

Parameters

- No parameters.

Description

Returns value of RTCS_errno, which is a task specific error value set by RTCS, when select() or accept() function return -1 (RTCS_SOCKET_ERROR). It can be used to determine, if the socket was closed (by closesocket()) or shut down (by shutdownsocket() or incoming TCP RST flag from connected remote host). It also clears the RTCS_errno to zero.

Return value

Specific error code

See also

- select()
- accept()

Example

```
uint32_t rtcserrno = RTCS_get_errno();
```

7.125 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 and RTCS_errno is different than the RTCSERR_SOCK_CLOSED 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().

7.126 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 Example: setting up RTCS.

7.127 RTCS_if_get_addr()

Returns the IPv4 address bound to the device interface.

Synopsis

```
_ip_address RTCS_if_get_addr(_rtcs_if_handle ihandle)
```

Parameters

- *ihandle [in]* — RTCS interface handle

Description

This function is used to retrieve the IPv4 address bound to the given device interface.

Return Value

- IPv4 address

See Also

- RTCS_if_bind()

Example

```
/* Print IPv4 address bound to interface. */
{
    _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
    _ip_address     addr = RTCS_if_get_addr(ihandle);
    char            prn_addr[RTCS_IP4_ADDR_STR_SIZE];

    inet_ntop(AF_INET, &addr, prn_addr, sizeof(prn_addr))
    printf("IPv4 Interface Address: %s\n", prn_addr);
}
```

7.128 RTCS_if_get_handle ()

Returns RTCS handle of the n-th interface.

Synopsis

```
_rtcs_if_handle RTCS_if_get_handle(uint32_t n)
```

Parameters

n [in] — Interface index (from zero).

Description

This function returns handle of n-th interface (from zero). It returns 0 if n-th interface is not available.

Return Value

- RTCS interace handle
- 0 (if n-th interface is not available)

See Also

- RTCS_if_add()

Example

```
/* Print number of registered interfaces.*/
{
   int i = 0;

   while(RTCS_if_get_handle(i)!= 0)
     {
       i++;
     }

     printf("There are %d registered interfaces.\n", i);
}
```

7.129 RTCS_if_get_mtu()

Synopsis

```
uint32_t RTCS_if_get_mtu(_rtcs_if_handle ihandle)
```

Parameters

rtcs_if_handle [in] — RTCS interface handle.

Description

This function returns Maximum Transmission Unit (MTU) of the device interface associated with *rtcs_if_handle*.

Return Value

- *Maximum Transmission Unit* (success)
- 0 (failure)

See Also

- RTCS_if_add()

7.130 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 Example: setting up RTCS.

7.131 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);

  return error;
}
```

7.132 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 rrtcs_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. */
```

7.133 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.

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,
  if (error) {
    printf("\nDHCP boot failed, error = %x.", error);
    return;
}
/* Use the network interface when it is bound. */
```

7.134 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. */
```

7.135 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, "Data Types".

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;
}
```

7.136 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

```
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;
   }
```

7.137 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()

7.138 RTCS_if_get_link_status ()

Returns actual link status of the interface.

Synopsis

```
bool RTCS_if_get_link_status(_rtcs_if_handle ihandle)
```

Parameters

ihandle [in] — Interface handle

Description

This function returns the actual link status of the given interface.

Return Value

- TRUE if interface link is active.
- FALSE if interface link is inactive.

See Also

- RTCS_if_get_handle ()

Example

```
/* Print link status.*/
{
    _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
    bool            link;;

   link = RTCS_if_get_link_status(ihandle);

   printf ("Link : %s\n", link ? "on" : "off");
}
```

7.139 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()

7.140 RTCS_if_get_dns_addr ()

Returns the n-th DNS IPv4 address from the DNS address list of the given device interface.

Synopsis

```
bool RTCS_if_get_dns_addr(_rtcs_if_handle ihandle, uint32_t n, _ip_address *dns_addr)
```

Parameters

ihandle [in] — RTCS interface handle

n [in] — DNS IPv4 address index (from 0)

dns_addr [out] — Pointer to DNS IPv4 address

Description

This function is used to retrieve DNS IPv4 addresses registered with the given device interface.

Return Value

- TRUE (success, dns_addr is filled)
- FALSE (failure, *n*-th DNS address is not available)

See Also

- RTCS6_if_add_dns_addr ()
- RTCS6_if_del_dns_addr ()

Example

```
/* Print all DNS IPv4 addresses.*/
{
   _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
   char            addr_str[RTCS_IP4_ADDR_STR_SIZE];
   int             i;
   _ip_address     dns_addr;

   for(i=0; (RTCS_if_get_dns_addr(ihandle, i, &dns_addr) == TRUE); i++)
   {

      printf ("[%d]: %s\n", i + 1, inet_ntop(AF_INET, &dns_addr, addr_str,
sizeof(addr_str)));
      }
   }
```

7.141 RTCS_if_add_dns_addr ()

Registers the DNS IPv4 address with the device interface.

Synopsis

```
uint32_t RTCS_if_add_dns_addr(_rtcs_if_handle ihandle, _ip_address dns_addr)
```

Parameters

ihandle [in] — RTCS interface handle

dns_addr [in] — DNS IPv4 address to add

Description

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

Return Value

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

See Also

- RTCS6_if_get_dns_addr ()
- RTCS6_if_del_dns_addr ()

Example

```
   /* Register DNS IPv4 address with the device interfae.*/
   {

      _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
      char             *addr_str = "8.8.8.8";
      _ip_address         dns_addr;

      if(inet_pton(AF_INET, addr_str, &dns_addr, sizeof(dns_addr)) == RTCS_OK)
   {

      if(RTCS_if_add_dns_addr(ihandle, dns_addr) == RTCS_OK)
      {
         printf("Adding DNS address is successful.\n");
      }
      else
      {
         printf("Adding DNS address is failed.\n");
      }
   }
}
```

7.142 RTCS_if_del_dns_addr ()

Unregisters the DNS IPv4 address from the device interface.

Synopsis

```
uint32_t RTCS_if_del_dns_addr(_rtcs_if_handle ihandle, _ip_address dns_addr)
```

Parameters

ihandle [in] — RTCS interface handle

dns_addr [in] — DNS IPv4 address to be removed

Description

This function removes the DNS IPv4 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 IPv4 address from the device interface.*/
{
   _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
   char            *addr_str = "8.8.8.8";
   _ip_address     dns_addr;

   if(inet_pton(AF_INET, addr_str, &dns_addr, sizeof(dns_addr)) == RTCS_OK)
   {

     if(RTCS_if_del_dns_addr(ihandle, dns_addr) == RTCS_OK)
     {

       printf("Deleting DNS address is successful.\n");
     }
     else
     {

       printf("Deleting DNS address is failed.\n");
     }
   }
}
```

7.143 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);
    }
}
```

7.144 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)

7.145 RTCS_selectall()

Select() function is recommended for new applications.

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

The function will block until activity is detected on any socket that the calling task owns if timeout is not -1. 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

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!");
     service_accept_error();

  } else {
    count = recv(connsock, buf, 500, 0);
    if (count <= 0)
       shutdown(connsock, FLAG_CLOSE_TX);
    else
      send(connsock, buf, count, 0);
  }
}
```

7.146 RTCS_selectset()

Select() function is recommended for new applications.

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);
}
```

7.147 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().

7.148 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);
```

7.149 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, uint32_t addr_lifetime)
```

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].

addr_lifetime [in] — IPv6 address valid lifetime (in seconds). The 0xFFFFFFFF value means infinite lifetime.

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,
IP6_ADDR_LIFETIME_INFINITE, IP6_ADDR_LIFETIME_INFINITE);
        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");
}
```

7.150 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");
              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");
}
```

7.151 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");
}
```

7.152 RTCS6_if_get_prefix_list_entry()

Returns content of the IPv6 prefix list of the device interface.

Synopsis

```
bool RTCS6_if_get_prefix_list_entry(_rtcs_if_handle ihandle, uint32_t n,
RTCS6_IF_PREFIX_LIST_ENTRY_PTR prefix_list_entry)
```

Parameters

ihandle [in] — RTCS interface handle

n [in] — IPv6 prefix index (from 0)

prefix_list_entry [in/out] — pointer to IPv6 prefix list entry

Description

This function may be used to retrieve the content of the IPv6 prefix list of the given device interface.

The function is used mainly for testing or obtaining information.

Return Value

- *TRUE* (success, *prefix_list_entry* is filled)
- *FALSE* (failure, *n*-th prefix is not available)

See Also

- RTCS6_if_get_neighbor_cache_entry()

Example

```
/* Print IPv6 Prefix List. */
{
    _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
    char            addr_str[RTCS_IP6_ADDR_STR_SIZE];
    int             i;
    RTCS6_IF_PREFIX_LIST_ENTRY     prefix_list_entry;

    printf("\nIPv6 Prefix List:\n");
    for(i=0; RTCS6_if_get_prefix_list_entry(ihandle, i, &prefix_list_entry) == TRUE; i++)
    {
        printf(" [%d] %s/%d\n", i,
            inet_ntop(AF_INET6, &prefix_list_entry.prefix, addr_str,
                sizeof(addr_str)), prefix_list_entry.prefix_length);
    }
 }
```

7.153 RTCS6_if_get_neighbor_cache_entry()

Returns content of the IPv6 neighbor cache of the device interface.

Synopsis

```
bool RTCS6_if_get_neighbor_cache_entry(_rtcs_if_handle ihandle, uint32_t n,
RTCS6_IF_NEIGHBOR_CACHE_ENTRY_PTR neighbor_cache_entry)
```

Parameters

ihandle [in] — RTCS interface handle

n [in] — IPv6 prefix index (from 0)

neighbor_cache_entry [in/out] — pointer to IPv6 neighbor cache entry

Description

This function may be used to retrieve content of IPv6 neighbor cache of the given device interface.

The function is used mainly for testing or information needs.

Return Value

- *TRUE* (success, *neighbor_cache_entry* is filled)
- *FALSE* (failure, *n*-th neighbor cache entry is not available)

See Also

- RTCS6_IF_NEIGHBOR_CACHE_ENTRY

Example

```
    /* Print IPv6 Prefix List. */
    {
       _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
       char            addr_str[RTCS_IP6_ADDR_STR_SIZE];
       int             i;
       RTCS6_IF_NEIGHBOR_CACHE_ENTRY neighbor_cache_entry;

       printf("\nIPv6 Neighbor Cache:\n");
       for(i=0; RTCS6_if_get_neighbor_cache_entry(ihandle, i, &neighbor_cache_entry) == TRUE;
i++)
      {
           printf(" [%d] %s = %02x:%02x:%02x:%02x:%02x:%02x (%s) \n", i,
                    inet_ntop(AF_INET6, &neighbor_cache_entry.ip_addr,
                    addr_str, sizeof(addr_str)),
                    neighbor_cache_entry.ll_addr[0],
                    neighbor_cache_entry.ll_addr[1],
                    neighbor_cache_entry.ll_addr[2],
                    neighbor_cache_entry.ll_addr[3],
                    neighbor_cache_entry.ll_addr[4],
                    neighbor_cache_entry.ll_addr[5],
                   (neighbor_cache_entry.is_router == TRUE)? "router" : "host" );
        }
}
```

7.154 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]* — sequence 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");
}
```

7.155 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)));
       }
         }
```

7.156 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");
          }
       }
         }
```

7.157 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_del_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");
   }
  }
}
```

7.158 RTCS6_if_is_disabled()

Detects if IPv6 is disabled for the interface.

Synopsis

```
_ip_address RTCS_if_get_addr(_rtcs_if_handle ihandle)
```

Parameters

- *ihandle [in]* — RTCS interface handle

Description

This function is used to detect if IPv6 is disabled for the interface. The IPv6 operation can be disabled if:

- Duplicate Address Detection has failed for the address that is a link-local address formed from an interface identifier based on the hardware address, which is supposed to be uniquely assigned (e.g., EUI-64 for an Ethernet interface).
- RTCSCFG_ENABLE_IP6 set to zero.
- Expired an operation timeout, if used IPv6 Evaluation library.

Return Value

- TRUE if IPv6 operation is disabled
- FALSE if IPv6 operation is enabled

7.159 select()

This function polls socket descriptors and checks if connection/data/close requests are pending. The function can block and wait until the RTCS signals the request of interest if no request is pending for any given socket.

Synopsis

```
int32_t select(int32_t nfds,
                  rtcs_fd_set *restrict readfds,
                  rtcs_fd_set *restrict writefds,
                  rtcs_fd_set *restrict exceptfds,
                  uint32_t timeout_ms);
```

Parameters

- int32_t nfds [IN]

The first nfds socket handles are checked in each set, meaning the sockets from 0 through nfds-1 in the descriptor sets are examined.

- rtcs_fd_set * readfds [IN/OUT]

  IN - Array of pointers to SOCKET_STRUCT to examine for receive activity

  OUT - Readfds contains the socket handles where activity has been detected

- rtcs_fd_set * writefds [IN/OUT]

  IN - Array of pointers to SOCKET_STRUCT to examine for transmit activity

  OUT - Writefds contains the socket handles where activity has been detected

- rtcs_fd_set * exceptfds [IN/OUT]

  IN - Array of pointers to SOCKET_STRUCT to examine for RTCS exception signal.

  OUT - exceptfds contains the socket handles where RTCS exception signal was detected.

  uint32_t timeout_ms [IN]

- If timeout_ms is zero, select() may block indefinitely. If timeout_ms is -1, select() only polls the socket descriptors and returns when the actual status is determined. Other values of timeout_ms determine the maximum time in milliseconds the select() function blocks.

Description

The rtcs_fd_sets must not overlap due to the restrict keyword. Readfds, writefds, and exceptfds should not point to the same rtcs_fd_set structure in memory.

Any of the rtcs_fd_set pointers may be given as null pointers if no descriptors are of interest. The select() function modifies the content of rtcs_fd_set arrays.

A stream socket is returned via readfds if it was in the input readfds and meets one of these conditions:

- Connection is requested and the socket is listening
- Close is requested
- Data is available for reading

A datagram socket is returned via readfds if it was in the input readfds and meets one of these conditions:

- Data is available for reading

A stream socket is returned via writefds if it was in the input writefds and meets one of these conditions:

- Send buffer is empty
- All send data is acknowledged by remote peer and send buffer is empty

If it was in the input writefds, a datagram socket is always returned via writefds. This is due to current RTCS implementation. The application should still check the return value from sendto()/send() function.

Select() function is recommended for new applications. The build time configuration parameter RTCSCFG_BACKWARD_COMPATIBILITY_RTCSSELECT is provided for backward compatibility with projects that rely on RTCS_selectset() and/or RTCS_selectall() functions.

The 3rd rtcs_fd_set argument, exceptfds, has special usage in RTCS. It can be used to unblock a select() call from other task by a call to setsockopt(). The functionality can be used in some networking applications, where it is required to monitor sockets for both data and application events, in RTCS, it is used in Websocket protocol implementation. If a socket is put into exceptfds and the exceptfds is monitored by the select(), it will return the socket through exceptfds in response to another task call of setsockopt() SO_EXCEPTION. The application should use getsockopt() SO_EXCEPTION to read the socket option value and clear it. The socket option value can be used to signal different application events between tasks.

Table 7-2. connecting/connected stream socket

connecting/connected stream socket	select()
The 1st FIN received (passive connection close). Remote host sends the FIN first for this connection and graceful connection close starts.	Return value > 0. The socket is in readfds or writefds array where it is monitored. Further send() is possible. Further recv() returns -1.
The 1st FIN sent (active connection close). RTCS sends the FIN first for this connection and graceful connection close starts.	Return value > 0. The socket is in readfds or writefds array where it is monitored. Further recv() is possible. Further send() returns -1.
FIN received (after it was sent out). Graceful connection close completes.	Return value > 0. The socket is in readfds or writefds array where it is monitored. Further send()/recv() return -1.
FIN sent (after it was received). Graceful connection close completes.	Return value > 0. The socket is in readfds or writefds array where it is monitored. Further send() or recv() return -1.
Abort request (remote host sends RST or unexpected SYN to us).	Return value = -1. RTCS_errno = RTCSERR_SOCK_ESHUTDOWN. The socket is in readfds or writefds array where it is monitored. send()/recv() return -1.

Table continues on the next page...

Table 7-2. connecting/connected stream socket (continued)

connecting/connected stream socket	select()
	The application calls the closesocket() function to release resources allocated by this socket.
Abort request (RTCS sends out RST in response to application closesocket() call, which resulted in connection abort).	Return value = -1. RTCS_errno = RTCSERR_SOCK_CLOSED. The socket handle is invalid and application must not use the socket.
Data segment available for reading	Return value > 0. Readfds has socket handle(s) with data available for reading.
Send buffer empty (remote host ACK'ed all send data)	Return value > 0. Writefds has socket handle(s) which can accept more send data.

Table 7-3. Listening stream socket

Listening stream socket	select()
Application calls shutdownsocket()	Return value = -1. RTCS_errno = RTCSERR_SOCK_ESHUTDOWN. The socket is in readfds or writefds array, where it was monitored.
Application calls closesocket()	Return value = -1. RTCS_errno = RTCSERR_SOCK_CLOSED. The socket handle is invalid and the application must not use the socket.
New connection is requested	Return value > 0. Readfds has socket handle(s) which can accept() the connection request.

Table 7-4. Datagram socket

Datagram socket	select()
Application calls shutdownsocket()	Does not cause select() to return.
Application calls closesocket()	Return value = -1. RTCS_errno = RTCSERR_SOCK_CLOSED. Datagram socket is invalid.
Datagram is received.	Return value > 0. Readfds has socket handle(s) with received datagram available for reading.

Return value

- The select() function returns the number of ready sockets contained in the descriptor sets, or RTCS_ERROR if an error occurred. RTCS_errno is set appropriately. Select() returns 0 if the time limit expires.

See also

- RTCS_FD_SET
- RTCS_FD_ZERO
- RTCS_FD_CLR
- RTCS_FD_ISSET

Example

```
    uint32_t socklist[3];
    rtcs_fd_set rfds;
    int32_t err;

    socklist[0] = socket(AF_INET, SOCK_STREAM, 0);
    socklist[1] = socket(AF_INET6, SOCK_STREAM, 0);
    socklist[2] = socket(AF_INET, SOCK_DGRAM, 0);

    …… /* call listen & bind as needed */

    while(1)
    {
      RTCS_FD_ZERO(&rfds);
      for(i=0; i<3; i++)
      {
         RTCS_FD_SET(socklist[i], &rfds);
      }

      err = select(3, &rfds, NULL, NULL, 0);
      if(RTCS_ERROR == err)
      {
         /* error occured */
      }
      else if(0 == err)
      {
        /* timeout */
      }
      else
      {
        if(FD_ISSET(socklist[0], &rfds))
        {
           …
        }
        if(FD_ISSET(socklist[1], &rfds))
        {
          …
        }
        if(FD_ISSET(socklist[2], &rfds))
        {
           …
      }
   }
}
```

7.160 RTCS_FD_SET

Add sock to the rtcs_fd_set.

Synopsis

void RTCS_FD_SET(const uint32_t sock, rtcs_fd_set * const p_fd_set);

7.161 RTCS_FD_CLR

Removes sock from the rtcs_fd_set.

Synopsis

void RTCS_FD_CLR(const uint32_t sock, rtcs_fd_set * const p_fd_set);

7.162 RTCS_FD_ZERO

Clears rtcs_fd_set.

Synopsis

void RTCS_FD_ZERO(rtcs_fd_set * const *p_fd_set*);

7.163 RTCS_FD_ISSET

Check if socket descriptor is present in rtcs_fd_set.

Synopsis

bool RTCS_FD_ISSET(const uint32_t *sock*, const rtcs_fd_set * const *p_fd_set*);

Return value

- Returns TRUE if sock is present in **p_fd_set*, FALSE otherwise.

Build time options

- RTCSCFG_BACKWARD_COMPATIBILITY_RTCSSELECT

    - Adds support for legacy RTCS_selectall() and RTCS_selectset() functions.
- RTCSCFG_SOCKET_OWNERSHIP
    - In addition to RTCS_selectall()/RTCS_selectset(), adds support for legacy functions and structures.

See also

- SOCK_Add_owner()
- SOCK_Remove_owner()
- SOCK_Is_owner()
- RTCS_attachsock()
- RTCS_detachsock()
- RTCS_transfersock()
- SOCK_OWNER_STRUCT

7.164 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: 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 RTCS6_if_add_dns_addr (), below. For stream sockets, flags can have one of the following values: 0, MSG_DONTWAIT, MSG_WAITACK..

Description

The function send() sends data on a stream socket or 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.

When the send() function returns depends on the flags parameter.

RTCS appends a push flag to all packets 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 override by the flags parameter 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 for datagram socket:

Group 1:

- RTCS_MSG_BLOCK — Overrides the OPT_SEND_NOWAIT datagram socket option and makes it behave as if it was FALSE.
- RTCS_MSG_NONBLOCK — Overrides the OPT_SEND_NOWAIT datagram socket option and makes it behave as if it was TRUE.

Group 2:

- RTCS_MSG_CHKSUM — Overrides the OPT_CHECKSUM_BYPASS checksum bypass option and makes it behave as if it was FALSE.
- RTCS_MSG_NOCHKSUM — Overrides the OPT_CHECKSUM_BYPASS checksum bypass option and makes it behave as though it is TRUE.

Group 3:

- RTCS_MSG_NOLOOP — Does not send the datagram to the loopback interface.
- Zero — Ignore.

Flags for stream socket:

- Zero — Socket option SEND_NOWAIT is applied. The default value for this option is FALSE.
- MSG_DONTWAIT — send() behaves as if SEND_NOWAIT socket option was TRUE.
- MSG_WAITACK — send() uses the application data buffer directly (it does not copy the data into the internal send buffer for the socket) and blocks.

When the send() function returns for the stream socket is shown in this table:

flags parameter	SEND_NOWAIT socket option	when the send() returns
Zero (0)	FALSE (default)	Blocking. Returns when all data is passed to the internal send buffer of the socket.
Zero (0)	TRUE	Copies all data up to maximum of buflen to the internal send buffer of the socket and returns immediately.
MSG_DONTWAIT	Don't care	Copies all data up to maximum of buflen to the internal send buffer of the socket and returns immediately.
MSG_WAITACK	Don't care	Blocking. Returns when all data is sent and acknowledged by the remote peer.

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

- listen()

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));
```

7.165 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 RTCS6_if_add_dns_addr ().

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)

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));
```

7.166 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] - Socket handle whose option is to be changed.

level [in] — Protocol levels, at which the option resides:

SOL_IGMP

SOL_LINK

SOL_SOCKET

SOL_TCP

SOL_UDP

SOL_IP

SOL_IP6

optname [in] — Option name.

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

- ip_mreq

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
- number of received bytes available for reading

- 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 sockets. For more information, see send(), sendto(), recv(), and recvfrom().

Options

This section describes the socket options.

Software exception for a socket

Option name	*SO_EXCEPTION*
Protocol level	*SOL_SOCKET*
Values	>= *0*
Default value	Zero
Change	Anytime
Socket type	Datagram or stream
Comments	RTCS specific option (non-portable). setsockopt() is used to set the option value and cause a select() function to return through 3rd rtcs_fd_set (exceptfds) argument, assuming the socket is set in the exceptfds. getsockopt() is used to read the option value from the socket and clear the option value to zero.

Receive timeout

Option name	*SO_RCVTIMEO*
Protocol level	*SOL_SOCKET*
Values	>= *zero milliseconds*
Default value	Zero
Change	Anytime
Socket type	Datagram or stream
Comments	When the timeout expires, recv() or recvfrom() return immediately with whatever data has been received.

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	≥ 0 (RTCS maintains the connection for this number of milliseconds).
Default value	180,000 (three 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 and for already established connections, it indicates how long the RTCS waits for acknowledge for sent segments before connection is considered as broken. 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. RTCS allows users to configure this timeout to any value so the applications that wish to do so can detect broken connections quickly.

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) or a receive timeout expires.
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 timefor 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). • Nonzero (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.

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 802.1Q VLAN Identifier Tag

Option name	*RTCS_SO_LINK_RX_8021Q_VID*
Protocol level	SOL_LINK
Values	• -1 (last received frame did not have an Ethernet 802.1Q VLAN Identifier tag). • 0…4094 (last received frame had an Ethernet 802.1Q tag with the specified VLAN ID).
Default value	—
Change	— (use with getsockopt() only; returns value in optval)
Socket type	Datagram or Stream (Ethernet)
Comments	Returned information is for teh last frame that the socket received.

Send Ethernet 802.1Q VLAN Identifier Tag

Option name	*RTCS_SO_LINK_RX_8021Q_VID*
Protocol level	SOL_LINK
Values	• -1 (RTCS does not include Ethernet 802.1Q priority tag). • 0…4094 (IRTCS includes Ethernet 802.1Q tag with the specified VLAN ID).

Table continues on the next page...

Default value	-1
Change	Anytime
Socket type	Datagram or Stream (Ethernet)
Comments	—

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) or a receive timeout expires.
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. This socket option is the same as SO_RCVTIMEO at SOL_SOCKET level on a stream socket.

Receive-Buffer Size - stream socket

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.

Receive Buffer Size - datagram socket

Option name	*OPT_RBSIZE*
Protocol level	*SOL_UDP*
Values	*At a minimum the size of the data in one UDP datagram.*
Default value	4096 bytes
Change	Anytime
Socket type	Datagram
Comments	UDP socket queues incoming datagrams up to this number bytes.

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)

Table continues on the next page...

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 (can be overriden)*
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

Table continues on the next page...

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*)

Bytes available for reading:

Option name	*SO_RCVNUM*
Protocol level	*SOL_SOCKET*
Values	-
Default value	-
Change	- (use with getsockopt() only)
Socket type	Datagram or stream
Comments	Returns number of received data bytes in the socket receive buffer available for reading from upper layer.

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

Table continues on the next page...

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 RTCSCF-G_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));`

Socket linger

Option name	*SO_LINGER*
Protocol level	*SOL_SOCKET*
Values	*Pointer to* `struct linger`
Default value	l_onoff = 0, l_linger_ms = 0
Change	Anytime
Socket type	Datagram or stream
Comments	Example

```
struct linger so_linger = {0}; /* l_onoff = 0; l_linger_ms = 0; */
int32_t status;

so_linger.l_onoff = 1; /* l_onoff = 1; l_linger_ms = 0; */
status = setsockopt(sock, SOL_SOCKET, SO_LINGER, (const
void*)&so_linger, sizeof(so_linger));
if(status == RTCS_OK)
{
    status = closesocket(sock);
}
```

Socket keepalive

Option name	*SO_KEEPALIVE*
Protocol level	*SOL_SOCKET*
Values	*TRUE or FALSE*
Default value	FALSE
Change	Before bound
Socket type	Stream
Comments	This option enables keep-alive probes for a socket connection. These probes are used to maintain a TCP connection and regularly test the connection to ensure that it's still available. It is only valid for connection-oriented protocols (TCP). default value is FALSE (keepalive feature disabled).

Keepalive count

Option name	*TCP_KEEPCNT*
Protocol level	*SOL_TCP*
Values	*>=0*
Default value	8
Change	Before bound
Socket type	Stream
Comments	When the SO_KEEPALIVE option is enabled, TCP sends a zero length packet to a connection that has been idle for some amount of time. If the remote host does not respond to a keepalive packet, TCP retransmits the keepalive certain number of times before a connection is considered to be

	broken. The default value for this keepalive packet retransmit limit is 8. The TCP_KEEPCNT option can be used to affect this value for a given socket, and specifies the maximum number of keepalive probes to be sent.

Keepalive idle time

Option name	*TCP_KEEPIDLE*
Protocol level	*SOL_TCP*
Values	*>=0 seconds*
Default value	7200 (2 hours)
Change	Before bound
Socket type	Stream
Comments	When the SO_KEEPALIVE option is enabled, TCP sends a zero length packet to a connection that has been idle for some amount of time. The TCP_KEEPIDLE option can be used to affect this time for a given socket, and specifies the number of seconds of idle time between keepalive probes. Default value for this idle time is 7200 seconds (2 hours).

Keepalive interval time

Option name	*TCP_KEEPINTVL*
Protocol level	*SOL_TCP*
Values	*>=0 seconds*
Default value	75 seconds
Change	Before bound
Socket type	Stream
Comments	When the SO_KEEPALIVE option is enabled, TCP sends a zero length packet to a connection that has been idle for some amount of time. If the remote host does not acknowledge this keepalive packet, TCP retransmits it after some amount of time. The default value for this retransmit time interval is 75 seconds. The TCP_KEEPINTVL option can be used to affect this value for a given socket, and specifies the number of seconds to wait before retransmitting a keepalive packet.

Examples

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);
```

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);
```

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);
```

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);
```

7.167 SOL_NAT_getsockopt

Reads NAT configuration.

Synopsis

```
  uint32_t SOL_NAT_getsockopt(
          uint32_t optname,
          void *optval,
          uint32_t *optlen)
```

Parameters

- optname [in] Option name, one of RTCS_SO_NAT_PORTS or RTCS_SO_NAT_TIMEOUTS.
- optval [in/out] Pointer where the configuration will be written to.
- optlen [in/out] Pointer to number of bytes. On function entry, the number determines the size of data structure the optval argument points to. On function return, the actual number of bytes written.

Return value

- Zero (Success - RTCS_OK)
- Non-zero (Failure - specific error code)

Example

```
#include <nat.h>

nat_ports ports;
nat_timeouts nat_touts;
uint32_t error;
uint32_t optlen;

optlen = sizeof(ports);
error = SOL_NAT_getsockopt(RTCS_SO_NAT_PORTS, &ports, &optlen);

optlen = sizeof(nat_touts);
error = SOL_NAT_getsockopt(RTCS_SO_NAT_TIMEOUTS, &nat_touts, &optlen);
```

7.168 SOL_NAT_setsockopt

Configures NAT.

Synopsis

```
uint32_t SOL_NAT_setsockopt(
           uint32_t optname,
           void *optval,
           uint32_t optlen)
```

Parameters

- optname [in] Option name, one of RTCS_SO_NAT_PORTS or RTCS_SO_NAT_TIMEOUTS.
- optval [in] Pointer to the option value.
- optlen [in] Number of bytes the optval points to.

Return value

- Zero (Success - RTCS_OK)
- Non-zero value (Failure - specific error code)

Example: Change the maximum port number used by RTCS NAT to 30000 and do not change the minimum number:

```
#include <nat.h>

nat_ports ports;
uint32_t error;

ports.port_min = 0; /* No modification */
ports.port_max = 30000;

error = SOL_NAT_setsockopt(RTCS_SO_NAT_PORTS, &ports, sizeof(ports));
```

Example: Change the TCP and UDP inactivity timeout values and do not change the FIN timeout value:

```
#include <nat.h>

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 = SOL_NAT_setsockopt(RTCS_SO_NAT_TIMEOUTS, &nat_touts, sizeof(nat_touts));
```

7.169 shutdownsocket()

Disable sends and/or receives on a socket.

Synopsis

```
int32_t shutdownsocket(uint32_t sock, int32_t how);
```

Parameters

- sock [in] – socket handle
- how [in] – type of socket shutdown

Description

Disables sens and/or receives on a datagram or stream socket. For a stream socket, shuts down all or part of the full duplex connection. The how argument specifies the type of shutdown. Possible values are:

- SHUT_RD Further receives will be disallowed.

- SHUT_WR Further sends will be disallowed. This may cause actions specific to the protocol family of the socket to happen.
- SHUT_RDWR Further sends and receives will be disallowed. SHUT_WR protocol specific action is included.

The following protocol specific actions apply to the use of SHUT_WR (and potentially also SHUT_RDWR), based on the properties of the socket.

Domain	Type	Protocol	Return value and action
AF_INET	SOCK_DGRAM	UDP	Return 0
AF_INET	SOCK_STREAM	TCP	Return 0. Send queued data, wait for ACK, then send FIN.
AF_INET6	SOCK_DGRAM	UDP	Return 0
AF_INET6	SOCK_STREAM	TCP	Return 0. Send queued data, wait for ACK, then send FIN.

When further sends are disallowed and send()/sendto() is attempted by an application, send()/sendto() return -1 and the error code in the socket will be set to RTCSERR_SOCK_ESHUTDOWN. When further receives are disallowed and recv()/recvfrom() is attempted by an applicatoin, recv()/recvfrom() return -1 and the error code in the socket will be set to RTCSERR_SOCK_ESHUTDOWN. When shutdownsocket() is called on a listening TCP socket, select() and accept() functions will return -1 and RTCS_errno will be set to RTCSERR_SOCK_ESHUTDOWN (assuming the socket handle being shutdown is monitored by select()/accept()). shutdownsocket() can be called repeatedly on a socket. The following state machine diagram shows the supported sequences of "how" argument for repeated calls of shutdownsocket() on the same socket:

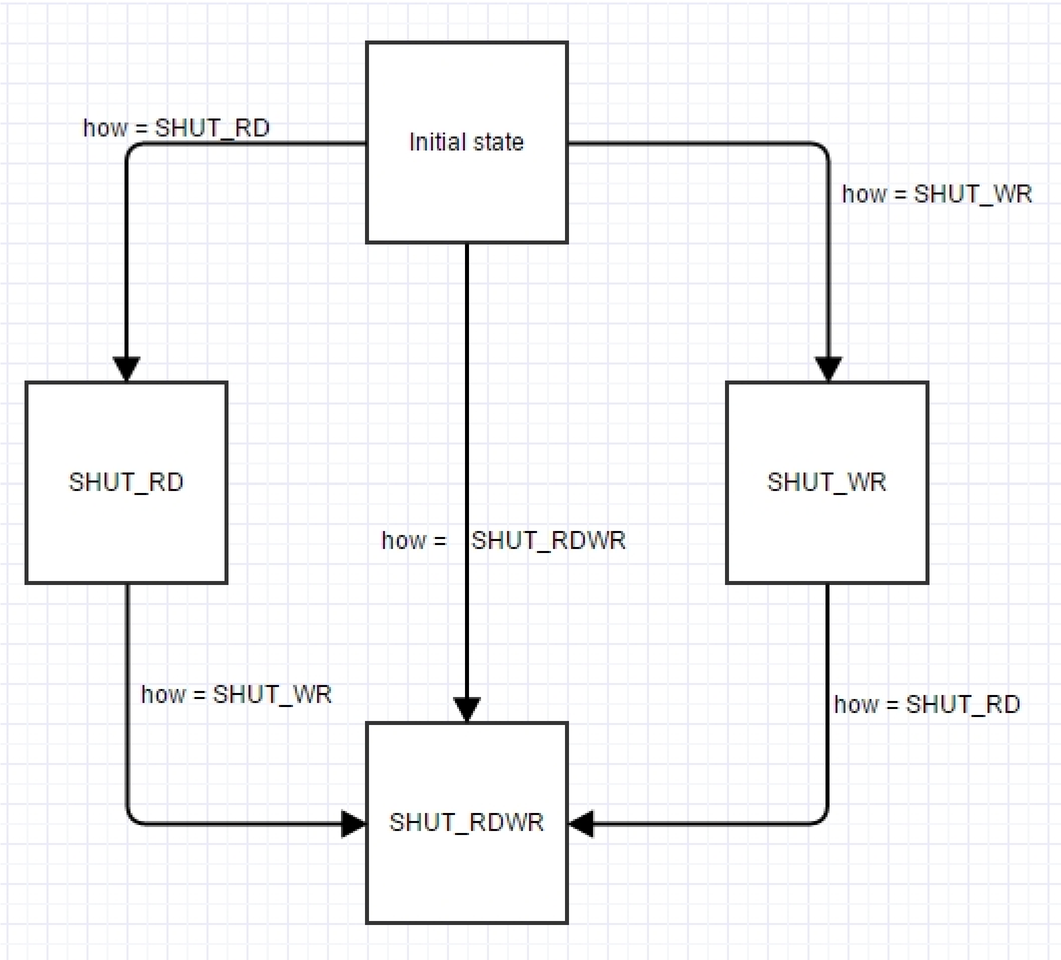

Figure 7-1. State machine diagram

When shutdown state of a socket is SHUT_RDWR, an attempted shutdownsocket() will only return RTCS_OK without changing internal socket state.

7.170 shutdown()

Shuts down the socket. This function is supported for backward compatibility. In new projects, it is recommended to use closesocket() and optionally SO_LINGER socket option.

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	• 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

Example

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

7.171 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 sends 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

See file \shell\source\rtcs\sh_smtp.c for source code demonstrating usage of function SMTP_send_email.

7.172 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");
```

7.173 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

7.174 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

7.175 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

7.176 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

7.177 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

7.178 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

7.179 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

7.180 SNMPv2_trap_warmStart()

Synopsis

```
void SNMPv2_trap_warmStart(void)
```

Description

This function sends warm start trap type 2 message.

Return Value

7.181 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()

7.182 SNMPv2_trap_authenticationFailure()

Synopsis

```
void SNMPv2_trap_authenticationFailure(void)
```

Description

This function sends authentication failure trap type 2 message.

Return Value

7.183 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

7.184 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

7.185 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

7.186 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

When the function starts the SNTP Client task that will first update the local time, 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.

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;
```

7.187 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).

7.188 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)

Example

See bind().

7.189 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

- TCP_STATS

7.190 TFTPCLN_connect()

Connect to TFTP server.

Synopsis:

```
    uint32_t TFTPCLN_connect(
        TFTPCLN_PARAM_STRUCT *params)
```

Parameters:

- params[in] - parameters of the TFTP client.

Description:

Function TFTPCLN_client() starts the TFTP client according to parameters from the _params_ structure. After successful call of this function, TFTP client is ready to transmit/receive files to/from server. As minimum only a remote host information must be set up in this structure. See chapter X.Y (link to chapter describing TFTPCLN_PARAM_STRUCT) for further description of each server parameter.

Return Value:

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

Example:

```
#include "tftpcln.h"

struct addrinfo        hints = {0};
struct addrinfo        *getadd_result;
int                    error;
uint32_t               handle;
int32_t                result;
TFTPCLN_PARAM_STRUCT   params = {0};

hints.ai_family = AF_UNSPEC;
error = getaddrinfo("192.168.1.1", "69", &hints, &getadd_result);
if (error == 0)
{
    params.sa_remote_host = *getadd_result->ai_addr;

    freeaddrinfo(getadd_result);

    handle = TFTPCLN_connect(&params);
    if (handle == 0)
    {
        printf("Error");
        _task_block();
    }
    else
    {
        printf("Downloading firmware from server...");
        result = TFTPCLN_get(handle, "a:\\firmware.bin", "firmware.bin");
        if (result == RTCS_OK)
        {
            printf("successful");
        }
        else
        {
            printf("failed");
        }
        TFTPCLN_disconnect(handle);
    }
}
else
{
    printf("Failed to resolve remote host address");
}
```

See Also:

- TFTPCLN_disconnet()
- TFTPCLN_PARAM_STRUCT

7.191 TFTPCLN_get

Download a file from TFTP server.

Synopsis:

```
int32_t TFTPCLN_get(
    uint32_t handle,
    char *local_file,
    char *remote_file)
```

Parameters:

- handle[in] - Handle of TFTP client created by function TFTPCLN_connect(). This parameter is mandatory.
- local_file[in] - Filename of file which will be stored locally.This parameter can be NULL, local filename is then same as remote filename.
- remote_file[in] - Filename of file to download from server. This parameter is mandatory (must not be NULL).

Description:

This function is used to download a remote file with TFTP protocol to local system. Calling this function blocks, until download is complete or fails.

Return Value:

- RTCS_OK (success)
- RTCS_ERROR (failure)

See Also:

- TFTPCLN_put
- TFTPCLN_connect
- TFTPCLN_disconnect

7.192 TFTPCLN_put

Upload a file to TFTP server.

Synopsis:

```
int32_t TFTPCLN_put(
    uint32_t handle,
    char *local_file,
    char *remote_file)
```

Parameters:

- handle[in] - Handle of TFTP client created by function TFTPCLN_connect(). This parameter is mandatory.

- local_file[in] - Filename of file which will be send to server from local system. This parameter is mandatory (must not be NULL).
- remote_file[in] - Filename of file to be created on server. This parameter can be NULL, remote filename is then same as local filename.

Description:

This function is used to upload a local file with TFTP protocol to remote system. Calling this function blocks, until download is complete or fails.

Return Value:

- RTCS_OK (success)
- RTCS_ERROR (failure)

See Also:

- TFTPCLN_put
- TFTPCLN_connect
- TFTPCLN_disconnect

7.193 TELNETSRV_init

Starts the Telnet server.

Synopsis

```
uint32_t TELNETSRV_init(
TELNETSRV_PARAM_STRUCT *params)
```

Parameters

params[in] - parameters of the telnet server.

Description:

Function TELNETSRV_init() starts the telnet server according to parameters from the _params_ structure.

The shell function and shell commands parameters are mandatory. If they are not provided, any connected client is immediately disconnected. See chapter TELNETSRV_PARAM_STRUCT for a description of each server parameter.

Return Value

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

Example

```
#include "shell.h"
#include "telnetsrv.h"
extern const SHELL_COMMAND_STRUCT Telnetsrv_shell_commands[];
     TELNETSRV_PARAM_STRUCT params = {0};
     uint32_t handle;

     params.shell_commands = (void *) Telnetsrv_shell_commands;
     params.shell = (TELNET_SHELL_FUNCTION) Shell;
     handle = TELNETSRV_init(params);
```

See Also

- TELNETSRV_release
- TELNETSRV_PARAM_STRUCT

7.194 TELNETSRV_release

Stops the Telnet server and releases all of its resources.

Synopsis

```
uint32_t TELNETSRV_release(
uint3_t server_h)
```

Parameters

server_h[in] - server handle (from function TELNETSRV_init).

Description

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

Return Value

- RTCS_OK - shutdown successfull.
- RTCS_ERR - shutdown failed.

See Also

- TELNETSRV_init

7.195 TFTPSRV_init

Starts the TFTP server.

Synopsis

```
uint32_t TFTPSRV_init(
TFTPSRV_PARAM_STRUCT *params)
```

Parameters

params[in] - parameters of the TFTP server.

Description

The function TFTPSRV_init() starts the TFTP server according to parameters from the _params_ structure. At a minimum, only a root directory must be set in this structure. See TFTPSRV_PARAM_STRUCT for a description of each server parameter.

Return Value

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

Example

```
#include "tftpsrv.h"
    TFTPSRV_PARAM_STRUCT params = {0};
    uint32_t handle;

    params.root_dir = "a:";
    handle = FTPSRV_init(params);
```

See Also

- TFTPSRV_release
- TFTPSRV_PARAM_STRUCT

7.196 TFTPSRV_release

Stops theTFTP server and releases all of its resources.

Synopsis

```
uint32_t TFTPSRV_release(
uint32_t server_h)
```

Parameters

server_h[in] - server handle (from function TFTPSRV_init).

Description

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

Return Value

- RTCS_OK - shutdown successfull.
- RTCS_ERR - shutdown failed.

See Also

- TFTPSRV_init

7.197 UDP_stats()

Gets a pointer to UDP statistics.

Synopsis

```
UDP_STATS_PTR  UDP_stats(void)
```

Description

- DHCPSRV_DATA_STRUCT

Function UDP_stats() gets a pointer to the UDP statistics that RTCS collects.

Return Value

Pointer to the *UDP_STATS* structure.

See Also

- ICMP_STATS
- IGMP_STATS
- TCP_STATS
- ARP_STATS

7.198 Functions Listed by Service

Service	Functions
DHCP Client	RTCS_if_bind_DHCP()
DHCP Server	DHCP* DHCPSRV*
DNS Resolver	getaddrinfo()
Echo Server	ECHOSRV_release()
Ethernet Driver	ENET_get_stats() (part of MQX RTOS) ENET_initialize() (part of MQX RTOS)
FTP Client	
FTP Server	FTPSRV_release()
HTTP Server	HTTPSRV_init() HTTPSRV_release() HTTPSRV_cgi_read() HTTPSRV_cgi_write() HTTPSRV_ssi_write()
IPCFG	ipcfg_bind_boot() ipcfg_bind_dhcp() 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_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()
IWCFG	iwcfg_set_essid()

Table continues on the next page...

Service	Functions
	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()
PPP Driver	PPP_init() PPP_release() PPP_pause() PPP_resume()
RTCS	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_ping() RTCSLOG_disable() RTCSLOG_enable()
SNMP Agent	SNMP_init() SNMP_trap_coldStart() MIB1213_init() MIB_find_objectname() MIB_set_objectname()
SNTP Client	
Sockets	listen() RTCS_selectall() RTCS_selectset()

Table continues on the next page...

Service	Functions
	select()
Statistics	IGMP_stats() NAT_stats() TCP_stats()
Telnet Client	TFTPCLN_connect()
Telnet Server	
TFTP Server	TFTPSRV_init

Chapter 8
Compile-time Options

8.1 Compile-time options

RTCS is built with certain features that can be included or excluded 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 to 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. The 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*

8.2 Recommended settings

The settings that you choose for compile-time configuration options depend on the requirements of your application. Table 8-1 llustrates some common settings you may want to use as you develop your application.

Table 8-1. Recommended compile-time settings

Option	Default	Debug	Speed	Size
RTCSCFG_IP_DISABLE_ DIRECTED_BROADCAST	0	0	0	0
RTCSCFG_ENABLE_8021Q	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

8.3 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.

8.3.1 RTCSCFG_FD_SETSIZE

Number of sockets per rtcs_fd_set. Default is 8.

8.3.2 RTCSCFG_SOMAXCONN

System wide maximum for listen backlog queue length. Default is 5.

8.3.3 RTCSCFG_ARP_CACHE_SIZE

ARP cache table size, per network interface.

8.3.4 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.

8.3.5 RTCSCFG_BACKWARD_COMPATIBILITY_RTCSSELECT

Adds support for legacy RTCS_selectall() and RTCS_selectset() functions.

8.3.6 RTCSCFG_BOOTP_RETURN_YIADDR

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

8.3.7 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.

8.3.8 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.

8.3.9 RTCSCFG_ENABLE_8021Q

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

The IEEE 802.1p priority tag is controlled by the RTCS_SO_LINK_TX_8021Q_PRIO and the RTCS_SO_LINK_RX_8021Q_PRIO socket options.

The VLAN Identifier tag is controlled by the RTCS_SO_LINK_TX_8021Q_VID and the RTCS_SO_LINK_RX_8021Q_VID socket options.

8.3.10 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.

8.3.11 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.

8.3.12 RTCSCFG_ENABLE_ICMP

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

8.3.13 RTCSCFG_ENABLE_IGMP

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

8.3.14 RTCSCFG_ENABLE_NAT

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

8.3.15 RTCSCFG_ENABLE_IPIP

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

8.3.16 RTCSCFG_ENABLE_RIP

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

8.3.17 RTCSCFG_ENABLE_SNMP

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

8.3.18 RTCSCFG_ENABLE_SSL

Default value is 0. Set to 1 to add support for SSL/TLS.

8.3.19 RTCSCFG_ENABLE_IP_REASSEMBLY

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

8.3.20 RTCSCFG_ENABLE_LOOPBACK

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

8.3.21 RTCSCFG_ENABLE_UDP

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

8.3.22 RTCSCFG_ENABLE_TCP

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

8.3.23 RTCSCFG_ENABLE_STATS

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

8.3.24 RTCSCFG_ENABLE_GATEWAYS

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

8.3.25 RTCSCFG_ENABLE_VIRTUAL_ROUTES

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

8.3.26 RTCSCFG_USE_KISS_RNG

Default 0. Must be 1 for PPP or tunneling.

8.3.27 RTCSCFG_ENABLE_ARP_STATS

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

8.3.28 RTCSCFG_PCBS_INIT

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

8.3.29 RTCSCFG_LLMNRSRV_PORT

The default listen port of the LLMNR server. According to the RFC 4795, the LLMNR server or responders must listen on the UDP port 5355. It is not recommended to change it.

8.3.30 RTCSCFG_LLMNRSRV_HOSTNAME_TTL

The default TTL value indicates for how long (in seconds) the link-local host name is valid for the LLMNR querier. The default value is 30 seconds (recommended by the RFC4795). In highly dynamic environments, such as the mobile ad-hoc networks, the TTL value may need to be reduced.

8.3.31 RTCSCFG_PCBS_GROW

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

8.3.32 RTCSCFG_PCBS_MAX

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

8.3.33 RTCSCFG_MSGPOOL_INIT

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

8.3.34 RTCSCFG_MSGPOOL_GROW

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

8.3.35 RTCSCFG_MSGPOOL_MAX

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

8.3.36 RTCSCFG_SOCKET_PART_INIT

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

8.3.37 RTCSCFG_SOCKET_PART_GROW

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

8.3.38 RTCSCFG_SOCKET_PART_MAX

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

8.3.39 RTCSCFG_UDP_RX_BUFFER_SIZE

UDP maximum per-socket queued data bytes. Override in application by using the setsockopt() OPT_RBSIZE socket option.

8.3.40 RTCSCFG_ENABLE_UDP_STATS

Set to 0 for disable UDP statistics.

8.3.41 RTCSCFG_ENABLE_TCP_STATS

Set to 0 for disable TCP statistics.

8.3.42 RTCSCFG_TCP_MAX_CONNECTIONS

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

8.3.43 RTCSCFG_TCP_MAX_HALF_OPEN

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

8.3.44 RTCSCFG_ENABLE_RIP_STATS

Default value RTCSCFG_ENABLE_STATS, enable RIP statistics.

8.3.45 RTCSCFG_QUEUE_BASE

Override in application by setting _RTCSQUEUE_base.

8.3.46 RTCSCFG_STACK_SIZE

Override in application by setting _RTCSTASK_stacksize.

8.3.47 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().

8.3.48 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.

8.3.49 RTCSCFG_ENABLE_IP4

Enable IPv4 Protocol support.

Default value 1.

8.3.50 RTCSCFG_ENABLE_IP6

Enable IPv6 Protocol support.

Default value 0.

8.3.51 RTCSCFG_ND6_NEIGHBOR_CACHE_SIZE

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

Default value 6.

8.3.52 RTCSCFG_ND6_PREFIX_LIST_SIZE

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

Default value 4.

8.3.53 RTCSCFG_ND6_ROUTER_LIST_SIZE

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

Default value 2.

8.3.54 RTCSCFG_IP6_IF_ADDRESSES_MAX

Maximum number of IPv6 addresses per interface.

Default value 5.

8.3.55 RTCSCFG_IP6_IF_DNS_MAX

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

Default value 2.

8.3.56 RTCSCFG_IP6_REASSEMBLY

Enable IPv6 packet reassembling.

Default value 1.

8.3.57 RTCSCFG_IP6_LOOPBACK_MULTICAST

Enable loopback of own IPv6 multicast packets.

Default value 0.

8.3.58 RTCSCFG_ND6_RDNSS

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

Default value 1.

8.3.59 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.

8.3.60 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.

8.3.61 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.

8.3.62 RTCSCFG_IP6_MULTICAST_SOCKET_MAX

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

Default value 1.

8.3.63 RTCSCFG_ENABLE_MLD

Enable Multicast Listener Discovery (MLDv1) Protocol support.

Default value 1.

8.3.64 FTPCCFG_SMALL_FILE_PERFORMANCE_ENANCEMENT

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

8.3.65 FTPCCFG_BUFFER_SIZE

FTP Client buffer size.

8.3.66 FTPCCFG_WINDOW_SIZE

FTP Client maximum TCP packet size.

8.3.67 RTCSCFG_ECHOSRV_DEBUG_MESSAGES

If set to TRUE at build time, the ECHOSRV_task prints information messages to stderr. Default value is TRUE.

8.3.68 RTCSCFG_ECHOSRV_DEFAULT_BUFLEN

The buffer size for the ECHOSRV service. The default value is 1500.

8.3.69 RTCSCFG_ECHOSRV_MAX_TCP_CLIENTS

Maximum number of simultaneously serviced clients connected to the ECHOSRV service with TCP protocol. Default value is 4.

8.3.70 RTCSCFG_ENABLE_SNMP_STATS

Enable SNMP statistics. Default value RTCSCFG_ENABLE_STATS.

8.3.71 RTCSCFG_IPCFG_ENABLE_DNS

Enable DNS name resolving.

8.3.72 RTCSCFG_IPCFG_ENABLE_DHCP

Enable DHCP binding (depends on RTCSCFG_ENABLE_UDP).

8.3.73 RTCSCFG_IPCFG_ENABLE_BOOT

Enable TFTP names processing and BOOT binding.

8.3.74 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.

8.3.75 BSPCFG_ENET_HW_TX_IP_CHECKSUM_NEW

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

8.3.76 BSPCFG_ENET_HW_TX_PROTOCOL_CHECKSUM_NEW

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.

8.3.77 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.

8.3.78 BSPCFG_ENET_HW_RX_PROTOCOL_CHECKSUM_NEW

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.

8.3.79 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.

8.3.80 RTCSCFG_ECHOCLN_DEFAULT_BUFLEN

The echo client application buffer size in bytes. Default is 1500.

8.3.81 RTCSCFG_ECHOCLN_DEFAULT_LOOPCNT

The echo client application loops per one call to the ECHOCLN_process(). Default is 1.

8.3.82 RTCSCFG_ECHOCLN_DEBUG_MESSAGES

When TRUE, the echo client application prints information messages to the stderr. Default is TRUE.

Chapter 9
Data Types

9.1 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*	
_task_id	**uint32_t**	*mqx.h*	In MQX source
u_char	**uchar**	*rpctypes.h*	
u_int	**uint32_t**	*rpctypes.h*	
u_long	**uint32_t**	*rpctypes.h*	
u_short	**uint16_t**	*rpctypes.h*	

9.2 Alphabetical list of RTCS data structures

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

- Function
- Definition
- Fields

9.2.1 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, be 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.

9.2.2 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.

9.2.3 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.

9.2.4 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 zero, 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.

9.2.5 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.

9.2.6 DHCPCLN6_STATUS

Enumeration type of return values for function DHCPCLN6_get_status().

```
typedef enum dhcpcln6_status
{ DHCPCLN6_STATUS_BOUND,
    DHCPCLN6_STATUS_UNBOUND,
    DHCPCLN6_STATUS_NOT_RUNNING
}DHCPCLN6_STATUS;
```

DHCPCLN6_STATUS_BOUND - Client is running, there are some addresses bound by DHCPv6 client.

DHCPCLN6_STATUS_UNBOUND - Client is running, there are no addresses bound by DHCPv6 client yet (message exchange is not done yet).

DHCPCLN6_STATUS_NOT_RUNNING - Client is not running.

9.2.7 DHCPCLN6_PARAM_STRUCT

```
typedef struct dhcpcln6_param_struct
{
    in6_addr                      *preferred;
    _rtcs_if_handle               interface;
    uint32_t                         flags;
    const DHCPCLN6_CALLBACK_TABLE *callbacks;
}DHCPCLN6_PARAM_STRUCT;
```

preferred

Preferred IPv6 address for device. Client will try to obtain this address from the server.

interface

RTCS handle to interface on which DHCPv6 client will be started.

flags

Client flags for enabling various features.

9.2.8 ECHOSRV_PARAM_STRUCT

This structure provides users with configuration parameters for ECHOSRV service. A pointer to this structure is passed as input parameter to ECHOSRV_init().

typedef struct echosrv_param_struct

```
{
  uint16_t                af;            /* Inet protocol family */
  uint16_t                port;          /* Listening port */
#if RTCSCFG_ENABLE_IP4
  in_addr                 ipv4_address;  /* Listening IPv4 address */
#endif
#if RTCSCFG_ENABLE_IP6
  in6_addr                ipv6_address;   /* Listening IPv6 address */
  uint32_t                ipv6_scope_id;  /* Scope ID for IPv6 */
#endif
  uint32_t                server_prio;    /* server task priority */
} ECHOSRV_PARAM_STRUCT;
af
AF_INET - to service only IPv4 clients
AF_INET6 - to service only IPv6 clients
AF_INET | AF_INET6 - to service IPv4 or IPv6 clients
```

port

Local port number being serviced. Port 7 should be used per RFC 862.

ipv4_address

Listening IPv4 address. All zeros mean a data from any IPv4 address will be replied to.

ipv6_address

ipv6_scope_id

Listening IPv6 address. All zeros mean a data from any IPv6 address from any interface.

server_prio

ECHOSRV service runs in a task. This parameter determines the priority of the ECHOSRV task. It should be assigned with a lower priority, or higher value, than the priority of the TCP/IP task.

9.2.9 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.

9.2.10 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.

9.2.11 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, or the server task and session tasks, will 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.

9.2.12 HTTPSRV_PARAM_STRUCT

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

```
typedef struct httpsrv_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_uri;
    _mqx_uint                   max_ses;
    bool                        use_nagle;
    HTTPSRV_CGI_LINK_STRUCT     *cgi_lnk_tbl;
    HTTPSRV_SSI_LINK_STRUCT     *ssi_lnk_tbl;
    HTTPSRV_PLUGIN_LINK_STRUCT  *plugins;
    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;
    const HTTPSRV_SSL_STRUCT    *ssl_params;
} HTTPSRV_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 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. See chapter Aliases 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).

plugins

Pointer to list of server plugins.

ssl_params

Pointer to HTTPSRV_SSL_STRUCT SSL parameter structure. It is optional and can be set to NULL.

9.2.13 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.

9.2.14 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.

9.2.15 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.

9.2.16 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.

9.2.17 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).

9.2.18 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, for example <%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.

9.2.19 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, for example the 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.

9.2.20 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.

9.2.21 HTTPSRV_PLUGIN_STRUCT

Structure defining webserver plugin:

```
typedef struct httpsrv_plugin_struct
{
    HTTPSRV_PLUGIN_TYPE type;
    void                *data;
}HTTPSRV_PLUGIN_STRUCT;
```

type

Type of plugin. Only HTTPSRV_WS_PLUGIN is supported.

data

Pointer to plugin data.

9.2.22 HTTPSRV_PLUGIN_LINK_STRUCT

Structure for linking resource (URI) to server plugin.

```
typedef struct httpsrv_plugin_link_struct
{
    char                  *resource;
    HTTPSRV_PLUGIN_STRUCT *plugin;
}HTTPSRV_PLUGIN_LINK_STRUCT;
```

resource

Path (relative to server root) of resource causing plugin invocation.

plugin

pointer to plugin structure.

9.2.23 HTTPSRV_SSL_STRUCT

The SSL parameter structure, which is used during HTTPS initialization of the HTTP server.

```
typedef struct httpsrv_ssl_struct
{
    char*               cert_file;
    char*               priv_key_file;
}HTTPSRV_SSL_STRUCT;
```

cert_file

Path to the HTTPS Server Certificate file.

priv_key_file

Path to the HTTPS Server private key file.

9.2.24 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;
```

9.2.25 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;
```

9.2.25.1 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.

9.2.26 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.

9.2.27 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.

9.2.28 in6_addr

Used as IPv6 address field in these structures:

- *ipv6_mreq*
- *sockaddr_in6*

```
typedef struct in6_addr
{
    union
    {
        uint8_t      _u6_addr8[16];
        uint16_t     _u6_addr16[8];
        uint32_t              _u6_addr32[4];
    } _u6_addr;
}in6_addr;
#define s6_addr   _u6_addr._u6_addr8
```

s6_addr

128-bit IPv6 address.

9.2.29 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.

9.2.30 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.

9.2.31 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.

9.2.32 IPCFG_IP_ADDRESS_DATA

Interface address structure.

```
typedef uint32_t _ip_address;
typedef struct ipcfg_ip_address_data
{
    _ip_address    ip;
    _ip_address mask;
    _ip_address router;
} IPCFG_IP_ADDRESS_DATA, * IPCFG_IP_ADDRESS_DATA_PTR;
```

ip

ip address

mask

mask

route

gateway

9.2.33 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.

9.2.34 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.

9.2.35 LLMNRSRV_PARAM_STRUCT

```
typedef struct llmnrsrv_param_struct
{
    _rtcs_if_handle             interface;
    LLMNRSRV_HOST_NAME_STRUCT   *host_name_table;
    uint16_t                    af;
    uint32_t                    task_prio;
} LLMNRSRV_PARAM_STRUCT;
```

Interface

RTCS handle of interface on which LLMNR server is listening.

host_name_table

Pointer to host name table. Last table entry must be zeroed.

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). This parameter is optional. By default server will use all enabled address families.

task_prio

This parameter determines the priority of the LLMNRSRV task. It should be assigned with a lower priority, or higher value, than the priority of the TCP/IP task. This parameter is optional. By default the priority is set to value defined by RTCSCFG_LLMNRSRV_SERVER_PRIO.

9.2.36 LLMNRSRV_HOST_NAME_STRUCT

Structure defining a row of the LLMNR host name table. Last table raw must be zeroed to mark the end of the table.

```
typedef struct llmnrsrv_host_name_struct
{
    char                    *host_name;
    uint32_t                host_name_ttl;
} LLMNRSRV_HOST_NAME_STRUCT;
```

host_name

Link-local host name advertised by the LLMNR server (null-terminated).

host_name_ttl

TTL value that indicates how many seconds the link-local host name is valid for the LLMNR querier in seconds (OPTIONAL). This parameter is optional. The default value is defined by the RTCSCFG_LLMNRSRV_HOSTNAME_TTL.

9.2.37 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.

9.2.38 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).

9.2.39 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.

9.2.40 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).

9.2.41 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.

9.2.42 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.

9.2.43 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.

9.2.44 rtcs_fd_set

This structure holds pointers to socket descriptors.

```
typedef struct tag_rtcs_fd_set
{
  uint32_t  fd_count;
  uint32_t  fd_array[RTCSCFG_FD_SETSIZE];
} rtcs_fd_set;
```

fd_count

The number of socket descriptors, increments with RTCS_FD_SET(), decrements with RTCS_FD_CLR().

fd_array

Pointers to socket descriptors are stored as unsigned 32-bit integer value.

9.2.45 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
```

9.2.46 RTCS_SSL_PARAMS_STRUCT

Initialization parameters for function RTCS_ssl_init()

```
typedef struct rtcs_ssl_params_struct
{
    char*              cert_file;
    char*              priv_key_file;
    char*              ca_file;
    RTCS_SSL_INIT_TYPE init_type;
}RTCS_SSL_PARAMS_STRUCT;
```

cert_file

Path to the application certificate file.

priv_key_file

Path to the application private key file.

ca_file

Path to CA (Certificate Authority) certificate file.

init_type

Type of initialization. Can have a value of either RTCS_SSL_SERVER or RTCS_SSL_CLIENT.

RTCS_SSL_SERVER means that SSL context will be initialized for server, RTCS_SSL_CLIENT

creates SSL context for client.

9.2.47 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.

9.2.48 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).

9.2.49 RTCS6_IF_PREFIX_LIST_ENTRY

Prefix List entry, returned by RTCS6_if_get_prefix_list_entry().

```
typedef struct rtc6_if_prefix_list_entry
{
    in6_addr            prefix;
    uint32_t            prefix_length;
} RTCS6_IF_PREFIX_LIST_ENTRY, *RTCS6_IF_PREFIX_LIST_ENTRY_PTR;
```

prefix

IPv6 prefix.

prefix_length

IPv6 prefix length (in bits). The number of leading bits in the Prefix that are valid.

9.2.50 RTCS6_IF_NEIGHBOR_CACHE_ENTRY

Neighbor Cache entry, returned by *RTCS6_if_get_neighbor_cache_entry()*.

```
typedef struct rtc6_if_neighbor_cache_entry
{
    in6_addr        ip_addr;
    ll_addr_t       ll_addr;
    uint32_t        ll_addr_size;
    bool            is_router;
} RTCS6_IF_NEIGHBOR_CACHE_ENTRY, *RTCS6_IF_NEIGHBOR_CACHE_ENTRY_PTR;
```

ip_addr

Neighbor's on-link unicast IPv6 address.

ll_addr

Link-layer address. Actual size is defined by *ll_addr_size*.

ll_addr_size

Size of link-layer address.

is_router

A flag indicating whether the neighbor is a router (*TRUE*) or a host (*FALSE*).

9.2.51 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.

9.2.52 RTCSMIB_VALUE

This structure is describing function for reading MIB node value and its type.

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

TYPE

Type of value. Ignored in non-leaf nodes. Can be one of following:

- RTCSMIB_NODETYPE_INT_CONST: Constant integer.
- RTCSMIB_NODETYPE_INT_PTR: Pointer to integer.
- RTCSMIB_NODETYPE_INT_FN: Pointer to function returning integer.
- RTCSMIB_NODETYPE_UINT_CONST: Constant unsigned integer.
- RTCSMIB_NODETYPE_UINT_PTR: Pointer to unsigned integer.
- RTCSMIB_NODETYPE_UINT_FN: Pointer to function returning unsigned integer.
- RTCSMIB_NODETYPE_DISPSTR_PTR: Display string (printable string).
- RTCSMIB_NODETYPE_DISPSTR_FN: Pointer to function returning string.
- RTCSMIB_NODETYPE_OCTSTR_FN: Octet string (sequence of octets).
- RTCSMIB_NODETYPE_OID_PTR: Pointer to MIB node which OID is used as a node value.
- RTCSMIB_NODETYPE_OID_FN: Pointer to function which returns node. This node OID is then used as node value.

PARAM

Pointer/value/function depending on type.

9.2.53 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.

9.2.54 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 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.

9.2.55 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.

9.2.56 sockaddr_in6

Structure for an IPv6 socket-endpoint identifier.

```
typedef struct sockaddr_in6
{
    uint16_t    sin6_family;
    uint16_t    sin6_port;
    in6_addr          sin6_addr;
    uint32_t    sin6_scope_id;
}sockaddr_in6;
```

sin6_family

Address family type. It is set to AF_INET6

sin6_port

Transport layer port number (in host byte order).

sin6_addr

128-bit IPv6 address.

sin6_scope_id

Scope zone index (interface identifier).

9.2.57 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.

9.2.58 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.

9.2.59 TELNETCLN_CALLBACK

Prototype for the Telnet client callback.

```
typedef void (TELNETCLN_CALLBACK)(void *param);
```

9.2.60 TELNETCLN_CALLBACKS_STRUCT

Structure containing telnet all client callbacks.

```
typedef struct telnetcln_callbacks_struct
{
```

```
    TELNETCLN_CALLBACK *on_connected;
    TELNETCLN_CALLBACK *on_disconnected;
    void               *param;
}TELNETCLN_CALLBACKS_STRUCT;
```

on_connected

Callback invoked when the client connects. If this callback is set to NULL, it is ignored.

on_disconnected

Callback invoked when the client disconnects from the server. If this callback is set to NULL, it is ignored.

param

Parameter for the Telnet client callbacks. It is gnored when set to NULL.

9.2.61 TELNETCLN_PARAM_STRUCT

```
typedef struct telnetcln_param_struct
{
    sockaddr                   sa_remote_host;
    bool                       use_nagle;
    MQX_FILE_PTR               fd_in;
    MQX_FILE_PTR               fd_out;
    TELNETCLN_CALLBACKS_STRUCT callbacks;
}TELNETCLN_PARAM_STRUCT;
```

sa_remote_host

Sockaddr structure filled with information about the remote Telnet host to which the client is trying to connect to.

use_nagle

Flag determining if the Nagle algorithm is used to connect to the server.

fd_in

File pointer used as the Telnet client input.

fd_out

File pointer used as the Telnet client output.

callbacks

A structure containing callbacks and the callback parameter.

9.2.62 TELNETSRV_PARAM_STRUCT

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

```
typedef struct telnetsrv_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
    uint32_t              max_ses;        /* maximal sessions count */
    bool                  use_nagle;      /* enable/disable nagle algorithm for server
sockets */
    uint32_t              server_prio;    /* server main task priority */
    TELNET_SHELL_FUNCTION shell;          /* Pointer to shell to run */
    void                  *shell_commands;/* Pointer to shell commands */
} TELNETSRV_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). By default server will use all enabled address families.

port

Port to listen on. Default value is 23 as defined by RFC.

ipv4_address

IPv4 address to listen on. This variable is present only if the IPv4 is enabled. By default server will listen on all available addresses.

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 TELNETSRVCFG_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 RTCSCFG_TELNETSRV_DEF_SERVER_PRIO.

shell

Function to be run after client connects.

shell_commands

Pointer to table of commands available to user connected to telnet server.

9.2.63 TFTPCLN_PARAM_STRUCT

his structure is used as a parameter for the TFTPCLN_connect() function.

```
typedef struct tftpcln_param_struct
{
    sockaddr                  sa_remote_host;
    TELNETCLN_DATA_CALLBACK   recv_callback;
    TELNETCLN_DATA_CALLBACK   send_callback;
    TELNETCLN_ERROR_CALLBACK  error_callback;
}TFTPCLN_PARAM_STRUCT;
```

sa_remote_host

Information about remote host client connection. It is usually created by the getaddrinfo() function. This variable is mandatory.

recv_callback

Callback invoked when the data packet is received. This variable can be NULL.

send_callback

Callback invoked when the data packet is sent. This variable can be NULL.

error_callback

Callback invoked when an error occurred during the data transfer. This variable can be NULL.

9.2.64 TELNETCLN_DATA_CALLBACK

Prototype of TFTP client send/receive callback.

```
typedef void(*TELNETCLN_DATA_CALLBACK)(uint32_t data_length);
```

data_length

Length of the sent/received data packet.

9.2.65 TELNETCLN_ERROR_CALLBACK

Prototype of TFTP client error callback.

```
typedef void(*TELNETCLN_ERROR_CALLBACK)(uint16_t error_code, char* error_string);
```

error_code

Numeric value of the error reported by the server.

error_string

A string describing the error sent by the server.

9.2.66 TFTPSRV_PARAM_STRUCT

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

```
typedef struct tftpsrv_param_struct
{
uint16_t af;
uint16_t port;
#if RTCSCFG_ENABLE_IP4
in_addr ipv4_address;
#endif
#if RTCSCFG_ENABLE_IP6
in6_addr ipv6_address;
uint32_t ipv6_scope_id;
#endif
uint32_t max_ses;
uint32_t server_prio;
char* root_dir;
}TFTPSRV_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 0 (listening on all addresses).

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 RTCSCFG_TFTPSRV_SES_CNT

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 RTCSCFG_TFTPSRV_SERVER_PRIO.

root_dir

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

9.2.67 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.

9.2.68 WS_DATA_STRUCT

WebSocket data structure.

```
typedef struct ws_data_struct
{
    uint8_t      *data_ptr;
    uint32_t     length;
    WS_DATA_TYPE type;
}WS_DATA_STRUCT;
```

data_ptr

Pointer to send/received data.

length

Length of data to be send/received.

type

Type of data (WS_DATA_INVALID, WS_DATA_TEXT, WS_DATA_BINARY).

9.2.69 WS_PLUGIN_STRUCT

Structure defining callbacks and parameter for the WebSocket plugin.

```
typedef struct ws_plugin_struct
{
    WS_CALLBACK_FN on_connect;
    WS_CALLBACK_FN on_message;
    WS_CALLBACK_FN on_error;
    WS_CALLBACK_FN on_disconnect;
    void*          cookie;
}WS_PLUGIN_STRUCT;
```

on_connect

Pointer to function called when client connects to server.

on_message

Pointer to function called when message is received from client.

on_error

Pointer to function called when error occurs.

on_disconnect

Pointer to function called when client disconnects from server.

cookie

callback parameter(s).

9.2.70 WS_USER_CONTEXT_STRUCT

Structure passed as parameter to all WebSocket callbacks.

```
typedef struct ws_user_context_struct
{
    uint32_t            handle;
    WS_ERROR_CODE       error;
    WS_DATA_STRUCT      data;
    uint32_t            fin_flag;
}WS_USER_CONTEXT_STRUCT;
```

handle

WebSocket handle.

error

Error code if error occurred.

data

Structure describing the data.

fin_flag

Flag signalizing end of message.

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

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

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Document Number MQXRTCSUG
Revision 3, 09/2015