Disclaimer

The content provided in this document is believed to be accurate, but not guaranteed to be entirely free from errors. The content of this document is also subject to change. While the information herein is assumed to be accurate, WITTENSTEIN high integrity systems accepts no liability whatsoever for errors and omissions, and assumes that all users understand the risks involved.

Copyright notice

None of the materials provided in this document may be used, reproduced or transmitted, in any form or by any means without written permission from WITTENSTEIN high integrity systems.

All rights reserved.

© 2016 WITTENSTEIN aerospace & simulation Ltd

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective companies.

Contact Address

WITTENSTEIN high integrity systems
Brown’s Court, Yanley Lane
Long Ashton Business Park
Long Ashton
Bristol
BS41 9LB
ENGLAND, UK

www.HighIntegritySystems.com
sales@HighIntegritySystems.com

Americas: +1 408 625 4712
ROTW +44 1275 395 600

WITTENSTEIN high integrity systems is a trading name of WITTENSTEIN aerospace & simulation ltd.
# Table of Contents

1. Introduction ........................................................................................................................................... 9

2. Users Guide .............................................................................................................................................. 10

   2.1 Source Code Organization .................................................................................................................. 10

   2.2 Initialising the TCP/IP Stack ................................................................................................................ 10

   2.3 Create, Configure and Bind a TCP Socket ........................................................................................... 12

   2.4 Sending Data Using a TCP Socket ........................................................................................................ 16

   2.5 Receiving Data Using a TCP Socket ...................................................................................................... 18

   2.6 Shutting Down and Closing a TCP Socket ............................................................................................ 20

   2.7 Create, Configure and Bind a UDP Socket ........................................................................................... 20

   2.8 Sending UDP Data (standard interface) ............................................................................................... 21

   2.9 Sending UDP Data (zero copy interface) ............................................................................................ 23

   2.10 Receiving UDP Data (standard interface) .......................................................................................... 26

   2.11 Receiving UDP Data (zero copy interface) ......................................................................................... 27

   2.12 TCP/IP Stack Network Buffers Allocation Schemes ........................................................................ 29

       2.12.1 Network Data Buffers .................................................................................................................. 29

       2.12.2 Buffer Allocation Schemes ........................................................................................................ 29

   2.13 OPENRTOS+TCP Configuration ......................................................................................................... 31

       2.13.1 Constants Affecting the TCP/IP Stack Task Execution Behaviour ........................................ 34

       2.13.2 Debug, Trace and Logging Settings .......................................................................................... 34

       2.13.3 Hardware and Driver Specific Settings ..................................................................................... 36

       2.13.4 TCP Specific Constants ............................................................................................................ 39
2.13.5 UDP Specific Constants ................................................................. 42
2.13.6 Other Constants Effecting Socket Behaviour ........................................ 43
2.13.7 Constants Affecting the ARP Behaviour ............................................... 45
2.13.8 Constants Affecting DHCP and Name Service Behaviour ..................... 46
2.13.9 Constants Affecting IP and ICMP Behaviour .................................... 48

2.14 OPENRTOS+TCP Configuration Examples ........................................ 48
2.14.1 TCP/IP Stack Configuration To Minimise RAM Consumption ................... 49
2.14.2 TCP/IP Stack Configuration to Maximise Throughput ........................... 49

2.15 TCP/IP Specific Trace Hook Macros .................................................. 51

3. API Reference Guide .............................................................................. 58

3.1 Primary Sockets Functions ....................................................................... 59
3.1.1 FreeRTOS_socket() ......................................................................... 59
3.1.2 FreeRTOS_bind() ............................................................................ 62
3.1.3 FreeRTOS_connect() ........................................................................ 64
3.1.4 FreeRTOS_listen() .......................................................................... 65
3.1.5 FreeRTOS_accept() .......................................................................... 67
3.1.6 FreeRTOS_send() ............................................................................ 69
3.1.7 FreeRTOS_sendto() .......................................................................... 71
3.1.8 FreeRTOS_recv() ............................................................................. 77
3.1.9 FreeRTOS_recvfrom() ....................................................................... 79
3.1.10 FreeRTOS_setsockopt() ................................................................. 86
3.1.11 Valid IOptionName Values ................................................................ 89
<table>
<thead>
<tr>
<th>Section</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.12</td>
<td>FreeRTOS_shutdown()</td>
<td>102</td>
</tr>
<tr>
<td>3.1.13</td>
<td>FreeRTOS_closesocket()</td>
<td>103</td>
</tr>
<tr>
<td>3.1.14</td>
<td>FreeRTOS_select()</td>
<td>105</td>
</tr>
<tr>
<td>3.2</td>
<td>Miscellaneous Sockets functions</td>
<td>108</td>
</tr>
<tr>
<td>3.2.1</td>
<td>FreeRTOS_createsocketset()</td>
<td>108</td>
</tr>
<tr>
<td>3.2.2</td>
<td>FreeRTOS_createsocketset()</td>
<td>109</td>
</tr>
<tr>
<td>3.2.3</td>
<td>FreeRTOS_FD_SET()</td>
<td>111</td>
</tr>
<tr>
<td>3.2.4</td>
<td>FreeRTOS_FD_CLR()</td>
<td>113</td>
</tr>
<tr>
<td>3.2.5</td>
<td>FreeRTOS_FD_ISSET()</td>
<td>116</td>
</tr>
<tr>
<td>3.2.6</td>
<td>FreeRTOS_inet_addr()</td>
<td>117</td>
</tr>
<tr>
<td>3.2.7</td>
<td>FreeRTOS_inet_ntoa()</td>
<td>119</td>
</tr>
<tr>
<td>3.2.8</td>
<td>FreeRTOS_htons(), FreeRTOS_ntohs(), FreeRTOS_hhtonl() &amp; FreeRTOS_ntohl()</td>
<td>120</td>
</tr>
<tr>
<td>3.2.9</td>
<td>FreeRTOS_outstanding()</td>
<td>121</td>
</tr>
<tr>
<td>3.2.10</td>
<td>FreeRTOS_recvcount()</td>
<td>121</td>
</tr>
<tr>
<td>3.2.11</td>
<td>FreeRTOS_issocketconnected()</td>
<td>122</td>
</tr>
<tr>
<td>3.2.12</td>
<td>FreeRTOS_getremoteaddress()</td>
<td>122</td>
</tr>
<tr>
<td>3.2.13</td>
<td>FreeRTOS_maywrite()</td>
<td>123</td>
</tr>
<tr>
<td>3.3</td>
<td>IP functions</td>
<td>124</td>
</tr>
<tr>
<td>3.3.1</td>
<td>FreeRTOS_IPInit()</td>
<td>124</td>
</tr>
<tr>
<td>3.3.2</td>
<td>FreeRTOS_GetUDPPayloadBuffer()</td>
<td>127</td>
</tr>
<tr>
<td>3.3.3</td>
<td>FreeRTOS_ReleaseUDPPayloadBuffer()</td>
<td>128</td>
</tr>
<tr>
<td>3.3.4</td>
<td>FreeRTOS_SendPingRequest()</td>
<td>129</td>
</tr>
</tbody>
</table>
3.3.5 FreeRTOS_GetAddressConfiguration() ................................................................. 133
3.3.6 FreeRTOS_GetMACAddress() .................................................................................. 135
3.3.7 FreeRTOS_GetIPAddress() ...................................................................................... 135
3.3.8 FreeRTOS_GetGatewayAddress() ........................................................................... 135
3.3.9 FreeRTOS_GetDNSServerAddress() ...................................................................... 136
3.3.10 FreeRTOS_GetNetmask() ...................................................................................... 136
3.3.11 FreeRTOS_OutputARPRequest() .......................................................................... 136
3.3.12 FreeRTOS_IsNetworkUp() .................................................................................... 137
3.3.13 FreeRTOS_SignalSocket() .................................................................................... 137
3.4 Event hook functions ............................................................................................... 138
  3.4.1 vApplicationIPNetworkEventHook() .................................................................. 138
  3.4.2 vApplicationPingReplyHook() .......................................................................... 141
4. Porting OPENRTOS+TCP ............................................................................................ 143
  4.1 Porting OPENRTOS+TCP to a New Embedded C Compiler .................................... 143
    4.1.1 Summary Bullet Points ...................................................................................... 143
    4.1.2 Detailed Explanation and Example .................................................................... 144
  4.2 Porting OPENRTOS+TCP to a Different Microcontroller ....................................... 146
    4.2.1 Introduction ....................................................................................................... 146
    4.2.2 Summary Bullet Points ...................................................................................... 148
    4.2.3 Network Buffers and Network Buffer Descriptors .......................................... 149
    4.2.4 Function That Must be Implemented by the Port Layer .................................. 150
    4.2.5 vNetworkInterfaceAllocateRAMToBuffers() only when BufferAllocation_1.c is used 151
4.2.6 Functions Provided by the TCP/IP Stack For Use By The Port Layer ........................................ 153

4.2.7 Network Interface Port Layer Examples .................................................................................. 155

4.3 FreeRTOS+TCP API .................................................................................................................. 167

4.3.1 pxGetNetworkBufferWithDescriptor() ............................................................................ 168

4.3.2 vReleaseNetworkBufferAndDescriptor() ......................................................................... 169

4.3.3 pucGetNetworkBuffer() .................................................................................................... 170

4.3.4 vReleaseNetworkBuffer() .................................................................................................. 171

4.3.5 eConsiderFrameForProcessing() ....................................................................................... 171

4.3.6 xSendEventStructToIPTask() ............................................................................................ 172

5. Running Examples ..................................................................................................................... 174

5.1 Building Example Program ..................................................................................................... 174

5.1.1 Prerequisites ....................................................................................................................... 174

5.1.2 Hardware Setup .................................................................................................................. 174

5.1.3 Opening a Project ................................................................................................................ 175

5.1.4 Software Setup #1: Setting a Static or Dynamic IP Address ............................................ 176

5.1.5 Software Setup #2: Selecting the (virtual) MAC Address .................................................. 177

5.1.6 Software Setup #3: Selecting the Network Interface ......................................................... 177

5.2 Running Examples .................................................................................................................. 179

5.2.1 Print and Logging Messages ............................................................................................... 179

5.2.2 Selecting the Examples to Run ............................................................................................ 180

5.3 Simple UDP Client and Server Examples ................................................................................. 180

5.4 A TCP Echo Client Example .................................................................................................. 180
1. Introduction

OPENRTOS+TCP is a scalable, open source and thread safe TCP/IP stack for use with FreeRTOS, OPENRTOS and SAFERTOS.

OPENRTOS+TCP provides a familiar and standards based Berkeley sockets interface, making it as simple to use and as quick to learn as possible. An alternative callback interface is also available for advanced users.

OPENRTOS+TCP’s features and RAM footprint are fully scalable, making OPENRTOS+TCP equally applicable to smaller lower throughput microcontrollers as larger higher throughput microprocessors.

<table>
<thead>
<tr>
<th>Features</th>
<th>Berkeley Sockets API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berkeley sockets API</td>
<td>FreeRTOS_socket()</td>
</tr>
<tr>
<td>Optionally supports TCP sliding windows</td>
<td>FreeRTOS_setsockopt()</td>
</tr>
<tr>
<td>Fully re-entrant and thread safe API</td>
<td>FreeRTOS_bind()</td>
</tr>
<tr>
<td>Includes ARP, DHCP, DNS, LLMNR, NBNS</td>
<td>FreeRTOS_listen()</td>
</tr>
<tr>
<td>Gratuitous ARPs</td>
<td>FreeRTOS_connect()</td>
</tr>
<tr>
<td>Static, DHCP and Auto-IP address assignment</td>
<td>FreeRTOS_accept()</td>
</tr>
<tr>
<td>Can also be used as a UDP only stack</td>
<td>FreeRTOS_send() / FreeRTOS_sendto()</td>
</tr>
<tr>
<td>Optionally callback interface</td>
<td>FreeRTOS_recv() / FreeRTOS_recvfrom()</td>
</tr>
<tr>
<td>Optionally fragment outgoing packets</td>
<td>Etc.</td>
</tr>
</tbody>
</table>
2. Users Guide

2.1 Source Code Organization

The OPENRTOS+TCP source code is distributed with the directory structure shown below.

```
FreeRTOS-Plus-TCP  [Contains the source files that implement the TCP/IP stack]
|                    |
+--include          [Contains the header files for the TCP/IP stack]
|                    |
+--portable         |
    |                  |
    +--Compiler       |
        |              |
        +--Compiler_x [Contains structure packing header files for Compiler_x]
        |              |
        +--Compiler_y [Contains structure packing header files for Compiler_y]
        |              |
        +--Compiler_z [Contains structure packing header files for Compiler_z]
|                    |
+--BufferManagement [Source files that implement various buffer allocation schemes]
|                    |
+--NetworkInterface |
    |                  |
    +--MCU_x         [Contains a network driver for the MCU_x family microcontrollers]
    |                  |
    +--MCU_y         [Contains a network driver for the MCU_y family of microcontrollers]
    |                  |
    +--MCU_z         [Contains a network driver for the MCU_z family of microcontrollers]
```

Figure 2-1 The OPENRTOS+TCP Directory Structure

2.2 Initialising the TCP/IP Stack

This section describes FreeRTOS_IPInit() and the callback function that gets invoked when 'network up' and 'network down' events occur.

FreeRTOS_IPInit() must be the first OPENRTOS+TCP function called. FreeRTOS_IPInit() can be called before or after the RTOS scheduler is started.

FreeRTOS_IPInit() creates the OPENRTOS+TCP RTOS task. The OPENRTOS+TCP task configures and initialises the network interface. If ipconfigUSE_NETWORK_EVENT_HOOK is set to
in FreeRTOSIPConfig.h then the TCP/IP stack will call
the vApplicationIPNetworkEventHook() callback function when the network is ready for use.

Two examples are provided below. The first demonstrates FreeRTOS_IPInit(). The second demonstrates vApplicationIPNetworkEventHook().

/* The MAC address array is not declared const as the MAC address will
normally be read from an EEPROM and not hard coded (in real deployed
applications). */
static uint8_t ucMACAddress[6] = { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55 };

/* Define the network addressing. These parameters will be used if either
ipconfigUDE_DHCP is 0 or if ipconfigUSE_DHCP is 1 but DHCP auto configuration failed. */
static const uint8_t ucIPAddress[4] = { 10, 10, 10, 200 };
static const uint8_t ucNetMask[4] = { 255, 0, 0, 0 };
static const uint8_t ucGatewayAddress[4] = { 10, 10, 10, 1 };

/* The following is the address of an OpenDNS server. */
static const uint8_t ucDNSServerAddress[4] = { 208, 67, 222, 222 };

int main( void )
{
    /* Initialise the RTOS's TCP/IP stack. The tasks that use the network
are created in the vApplicationIPNetworkEventHook() hook function
below. The hook function is called when the network connects. */
    FreeRTOS_IPInit( ucIPAddress,
                    ucNetMask,
                    ucGatewayAddress,
                    ucDNSServerAddress,
                    ucMACAddress );

    /*
    * Other RTOS tasks can be created here.
    */
    /* Start the RTOS scheduler. */
    vTaskStartScheduler();
/* If all is well, the scheduler will now be running, and the following line will never be reached. If the following line does execute, then there was insufficient FreeRTOS heap memory available for the idle and/or timer tasks to be created. */
for( ;; );
}

Figure 2-2 Example use of the FreeRTOS_IPInit() API function

void vApplicationIPNetworkEventHook( eIPCallbackEvent_t eNetworkEvent )
{
    static BaseType_t xTasksAlreadyCreated = pdFALSE;

    /* Both eNetworkUp and eNetworkDown events can be processed here. */
    if( eNetworkEvent == eNetworkUp )
    {
        /* Create the tasks that use the TCP/IP stack if they have not already been created. */
        if( xTasksAlreadyCreated == pdFALSE )
        {
            /*
             * For convenience, tasks that use OPENRTOS+TCP can be created here to ensure they are not created before the network is usable.
             */
            xTasksAlreadyCreated = pdTRUE;
        }
    }
}

Figure 2-3 Example vApplicationIPNetworkEventHook() definition

2.3 Create, Configure and Bind a TCP Socket

TCP Sockets are:

- Created using the FreeRTOS_socket() API function with the xType (second) parameter set to FREERTOS_SOCK_STREAM,
- Configured using the FreeRTOS_setsockopt() function,
- and bound to a port using the FreeRTOS_bind() function.
If the socket is used to implement a server then call FreeRTOS_listen() to place the socket into the Listening state, and call FreeRTOS_accept() to accept incoming connections. Source code examples are provided in this section.

To change the size of the receive and send buffers used by the TCP socket from their defaults call FreeRTOS_setsockopt() using the FREERTOS_SO_RCVBUF and FREERTOS_SO_SNDBUF parameters respectively. This must be done immediately after the socket is created, before it is connected.

If ipconfigUSE_TCP_WIN is set to 1 in FreeRTOSIPConfig.h then the socket will use a sliding window to minimise overhead and maximise throughput. The size of the sliding window can be changed from its default using the FREERTOS_SO_WIN_PROPERTIES parameter to FreeRTOS_setsockopt(). The sliding window size is specified in units of MSS (so if the MSS is set to 200 bytes then a sliding window size of 2 is equal to 400 bytes) and must always be smaller than or equal to the size of the internal buffers in both directions.

By default a child socket is automatically created to handle any connections accepted on a listening TCP/IP socket (the default behaviour can be changed using the FREERTOS_SO_REUSE_LISTEN_SOCKET parameter in a call to FreeRTOS_setsockopt()). Child sockets inherit the buffer sizes and sliding window sizes of their parent sockets.

The first example below demonstrates how to create, configure and bind a client socket. The second example below demonstrates how to create, configure, and bind a server socket, then how to accept new connections on the server socket.

```c
void vCreateTCPClientSocket( void )
{
  xSocket_t xClientSocket;
  socklen_t xSize = sizeof( freertos_sockaddr );
  static const TickType_t xTimeOut = pdMS_TO_TICKS( 2000 );

  /* Attempt to open the socket. */
  xClientSocket = FreeRTOS_socket( PF_INET,
    SOCK_STREAM, /* SOCK_STREAM for TCP. */
    IPPROTO_TCP );

  /* Check the socket was created. */
  configASSERT( xClientSocket != FREERTOS_INVALID_SOCKET );

  /* If FREERTOS_SO_RCVBUF or FREERTOS_SO_SNDBUF are to be used with
```
FreeRTOS_setsockopt() to change the buffer sizes from their default then do it here!. (see the FreeRTOS_setsockopt() documentation. */

/* If ipconfigUSE_TCP_WIN is set to 1 and FREERTOS_SO_WIN_PROPERTIES is to be used with FreeRTOS_setsockopt() to change the sliding window size from its default then do it here! (see the FreeRTOS_setsockopt() documentation. */

/* Set send and receive time outs. */
FreeRTOS_setsockopt( xClientSocket, 0, FREERTOS_SO_RCVTIMEO, &xTimeOut, sizeof( xTimeOut ) );

FreeRTOS_setsockopt( xClientSocket, 0, FREERTOS_SO_SNDTIMEO, &xTimeOut, sizeof( xTimeOut ) );

/* Bind the socket, but pass in NULL to let OPENRTOS+TCP choose the port number. See the next source code snipped for an example of how to bind to a specific port number. */
FreeRTOS_bind( xClientSocket, NULL, xSize );

FIGURE 2-4 Creating, configuring and binding a TCP client socket

void vCreateTCPServerSocket( void )
{
struct freertos_sockaddr xClient, xBindAddress;
xSocket_t xListeningSocket, xConnectedSocket;
socklen_t xSize = sizeof( xClient );
static const TickType_t xReceiveTimeOut = portMAX_DELAY;
const BaseType_t xBacklog = 20;

/* Attempt to open the socket. */
xListeningSocket = FreeRTOS_socket( PF_INET,
SOCK_STREAM, /* SOCK_STREAM for TCP. */
IPPROTO_TCP);

/* Check the socket was created. */
configASSERT(xListeningSocket != FREERTOS_INVALID_SOCKET);

/* If FREERTOS_SO_RCVBUF or FREERTOS_SO_SNDBUF are to be used with FreeRTOS_setsockopt() to change the buffer sizes from their default then do it here!. (see the FreeRTOS_setsockopt() documentation. */

/* If ipconfigUSE_TCP_WIN is set to 1 and FREERTOS_SO_WIN_PROPERTIES is to be used with FreeRTOS_setsockopt() to change the sliding window size from its default then do it here! (see the FreeRTOS_setsockopt() documentation. */

/* Set a time out so accept() will just wait for a connection. */
FreeRTOS_setsockopt(xListeningSocket,
    0,
    FREERTOS_SO_RCVTIMEO,
    &xReceiveTimeOut,
    sizeof(xReceiveTimeOut));

/* Set the listening port to 10000. */
xBindAddress.sin_port = (uint16_t) 10000;
xBindAddress.sin_port = FreeRTOS_htons(xBindAddress.sin_port);

/* Bind the socket to the port that the client RTOS task will send to. */
FreeRTOS_bind(xListeningSocket, &xBindAddress, sizeof(xBindAddress));

/* Set the socket into a listening state so it can accept connections. 
The maximum number of simultaneous connections is limited to 20. */
FreeRTOS_listen(xListeningSocket, xBacklog);

for( ;; )
{
    /* Wait for incoming connections. */
    xConnectedSocket = FreeRTOS_accept(xListeningSocket, &xClient, &xSize);
    configASSERT(xConnectedSocket != FREERTOS_INVALID_SOCKET);}
Figure 2-5 Creating, configuring and binding a TCP server socket

2.4 Sending Data Using a TCP Socket

After a TCP socket has been created, configured, and bound it can be connected to a remote socket using the FreeRTOS_connect() API function, or it can accept connections from a remote socket. Once connected, data is sent to the remote socket using the FreeRTOS_send() API function.

The source code below shows a function that creates a socket, sends data to the socket, then gracefully shuts down and closes the socket. Note that this socket is not explicitly bound to a port number - causing it to be bound automatically inside the FreeRTOS_connect() API function.

```c
void vTCPSend( char *pcBufferToTransmit, const size_t xTotalLengthToSend ) {
    xSocket_t xSocket;
    struct freertos_sockaddr xRemoteAddress;
    BaseType_t xAlreadyTransmitted = 0, xBytesSent = 0;
    TaskHandle_t xRxTask = NULL;
    size_t xLenToSend;

    /* Set the IP address (192.168.0.50) and port (1500) of the remote socket to which this client socket will transmit. */
    xRemoteAddress.sin_port = FreeRTOS_htons( 15000 );
    xRemoteAddress.sin_addr = FreeRTOS_inet_addr_quick( 192, 168, 0 200 );

    /* Create a socket. */
    xSocket = FreeRTOS_socket( FREERTOS_AF_INET,
                               FREERTOS_SOCK_STREAM, /* FREERTOS_SOCK_STREAM for TCP.*/
                               /* FREERTOS_SOCK_DGRAM for UDP. */
                               );
```
/* Connect to the remote socket. The socket has not previously been bound to a local port number so will get automatically bound to a local port inside the FreeRTOS_connect() function. */
if( FreeRTOS_connect( xSocket, &xRemoteAddress, sizeof( xRemoteAddress ) ) == 0 )
{
    /* Keep sending until the entire buffer has been sent. */
    while( xAlreadyTransmitted < xTotalLengthToSend )
    {
        /* How many bytes are left to send? */
        xLenToSend = xTotalLengthToSend - xAlreadyTransmitted;
        xBytesSent = FreeRTOS_send( /* The socket being sent to. */
            xSocket,
            /* The data being sent. */
            &pcBufferToTransmit[ xAlreadyTransmitted ],
            /* The remaining length of data to send. */
            xLenToSend,
            /* ulFlags. */
            0 );

        if( xBytesSent >= 0 )
        {
            /* Data was sent successfully. */
            xAlreadyTransmitted += xBytesSent;
        }
        else
        {
            /* Error - break out of the loop for graceful socket close. */
            break;
        }
    }
}

/* Initiate graceful shutdown. */
FreeRTOS_shutdown( xSocket, FREERTOS_SHUT_RDWR );
/* Wait for the socket to disconnect gracefully (indicated by FreeRTOS_recv() returning a FREERTOS_EINVAL error) before closing the socket. */
while( FreeRTOS_recv( xSocket, pcBufferToTransmit, xTotalLengthToSend, 0 ) >= 0 )
{
    /* Wait for shutdown to complete. If a receive block time is used then
       this delay will not be necessary as FreeRTOS_recv() will place the RTOS task
       into the Blocked state anyway. */
    vTaskDelay( pdTICKS_TO_MS( 250 ) );

    /* Note - real applications should implement a timeout here, not just
       loop forever. */
}

/* The socket has shut down and is safe to close. */
FreeRTOS_closesocket( xSocket );

Figure 2-6 Example using FreeRTOS_send()

2.5 Receiving Data Using a TCP Socket

FreeRTOS_recv() is used to receive data from a TCP socket. FreeRTOS_recv() cannot be called
until the TCP socket has been created, configured, bound and connected to a remote socket.

The source code below demonstrates how to use FreeRTOS_recv() to place received data into a
buffer. In the example it is assumed that the socket has already been created and connected.

#define BUFFER_SIZE 512
static void prvEchoClientRxTask( void *pvParameters )
{
    xSocket_t xSocket;
    static char cRxedData[ BUFFER_SIZE ];
    BaseType_t lBytesReceived;

    /* It is assumed the socket has already been created and connected before
       being passed into this RTOS task using the RTOS task's parameter. */
    xSocket = ( xSocket_t ) pvParameters;

    for( ;; )
{ /* Receive another block of data into the cRxedData buffer. */
   lBytesReceived = FreeRTOS_recv( xSocket, &cRxedData, BUFFER_SIZE, 0 );

   if( lBytesReceived > 0 )
   {
      /* Data was received, process it here. */
      prvPorcessData( cRxedData, lBytesReceived );
   }
   else if( lBytesReceived == 0 )
   {
      /* No data was received, but FreeRTOS_recv() did not return an error.
         Timeout? */
   }
   else
   {
      /* Error (maybe the connected socket already shut down the socket?).
         Attempt graceful shutdown. */
      FreeRTOS_shutdown( xSocket, FREERTOS_SHUT_RDWR );
      break;
   }
}

/* The RTOS task will get here if an error is received on a read. Ensure the
   socket has shut down (indicated by FreeRTOS_recv() returning a FREERTOS_EINVAL
   error before closing the socket). */

while( FreeRTOS_recv( xSocket, pcBufferToTransmit, xTotalLengthToSend, 0 ) >= 0 )
{
   /* Wait for shutdown to complete. If a receive block time is used then
      this delay will not be necessary as FreeRTOS_recv() will place the RTOS task
      into the blocked state anyway. */
   vTaskDelay( pdTICKS_TO_MS( 250 ) );

   /* Note - real applications should implement a timeout here, not just
      loop forever. */
}

/* Shutdown is complete and the socket can be safely closed. */
FreeRTOS_closesocket( xSocket );

/* Must not drop off the end of the RTOS task - delete the RTOS task. */
xTaskDelete( NULL );}

Figure 2-7 Example using FreeRTOS_recv()

2.6 Shutting Down and Closing a TCP Socket

A TCP socket that is not connected can be closed using the FreeRTOS_closesocket() API function.

A TCP socket that is connected should not be closed until the connection has been shut down. To gracefully shut down a socket first call FreeRTOS_shutdown(), then wait for read attempts on the socket to return FREERTOS_EINVAL, indicating that the socket is no longer connected.

The source code examples on both the Sending TCP Data and the Receiving TCP Data sections demonstrate a connected socket being shut down then closed.

2.7 Create, Configure and Bind a UDP Socket

UDP Sockets are created using the FreeRTOS_socket() API function with the xType (second) parameter set to FREERTOS_SOCK_DGRAM, configured using the FreeRTOS_setsockopt() function, and bound to a port (if necessary) using the FreeRTOS_bind() function.

static void prvSimpleUDPServerTask( void *pvParameters )
{
  long lBytes;
  struct freertos_sockaddr xBindAddress;
  xSocket_t xListeningSocket;
  const TickType_t xSendTimeOut = 200 / portTICK_PERIOD_MS;

  /* Attempt to open the UDP socket. */
  xListeningSocket = FreeRTOS_socket( FREERTOS_AF_INET,
                                      FREERTOS_SOCK_DGRAM, /*FREERTOS_SOCK_DGRAM for UDP. */
                                      /*FREERTOS_SOCK_DGRAM for UDP. */
                                      FREERTOS_SOCK_DGRAM, /*FREERTOS_SOCK_DGRAM for UDP. */
                                      ...)
```c
FREERTOS_IPPROTO_UDP );

/* Check for errors. */
configASSERT( xListeningSocket != FREERTOS_INVALID_SOCKET );

/* Ensure calls to FreeRTOS_sendto() timeout if a network buffer cannot be obtained within 200ms. */
FreeRTOS_setsockopt( xListeningSocket, 0, FREERTOS_SO_SNDTIMEO, &xSendTimeOut, sizeof( xSendTimeOut ) );

/* Bind the socket to port 0x1234. */
xBindAddress.sin_port = FreeRTOS_htons( 0x1234 );
FreeRTOS_bind( xListeningSocket, &xBindAddress, sizeof( xBindAddress ) );

for( ;; )
{
    /*
     * The socket can now send and receive data here.
     */
}
```

Figure 2-8 Creating, configuring and binding a UDP socket

### 2.8 Sending UDP Data (standard interface)

The `FreeRTOS_sendto()` TCP/IP stack API function is used to send data to a UDP socket. Data can only be sent after the socket has been created, configured, and optionally bound to a local port number.

As detailed on the `FreeRTOS_sendto()` API reference section, `FreeRTOS_sendto()` can be used with standard calling semantics, or zero copy calling semantics. This section demonstrates the standard calling semantics.
The source code below shows a RTOS task that creates a UDP socket before entering a loop that sends a string to the socket (using the standard calling semantics) every 1 second (1000ms). The comments in the source code example provide more information.

```c
static void vUDPSendUsingStandardInterface( void *pvParameters ) {
    xSocket_t xSocket;
    struct freertos_sockaddr xDestinationAddress;
    uint8_t cString[ 50 ];
    uint32_t ulCount = 0UL;
    const TickType_t x1000ms = 1000UL / portTICK_PERIOD_MS;

    /* Send strings to port 10000 on IP address 192.168.0.50. */
    xDestinationAddress.sin_addr = FreeRTOS_inet_addr( "192.168.0.50" );
    xDestinationAddress.sin_port = FreeRTOS_htons( 10000 );
    /* Create the socket. */
    xSocket = FreeRTOS_socket( FREERTOS_AF_INET,
                              FREERTOS_SOCK_DGRAM,/*FREERTOS_SOCK_DGRAM for UDP.*/
                              FREERTOS_IPPROTO_UDP );

    /* Check the socket was created. */
    configASSERT( xSocket != FREERTOS_INVALID_SOCKET );

    /* NOTE: FreeRTOS_bind() is not called. This will only work if
    ipconfigALLOW_SOCKET_SEND_WITHOUT_BIND is set to 1 in FreeRTOSIPConfig.h. */

    for( ;; )
    {
        /* Create the string that is sent. */
        sprintf( cString,
                 "Standard send message number %lu\n",
                 ulCount );

        /* Send the string to the UDP socket. ulFlags is set to 0, so the standard
        semantics are used. That means the data from cString[] is copied
        into a network buffer inside FreeRTOS_sendto(), and cString[] can be
        reused as soon as FreeRTOS_sendto() has returned. */
        FreeRTOS_sendto( xSocket,
```
cString,
strlen( cString ),
0,
&xDestinationAddress,
sizeof( xDestinationAddress ) );

ulCount++;

/* Wait until it is time to send again. */
vTaskDelay( x1000ms );
}
}

Figure 2-9 Example using FreeRTOS_sendto() with the standard (as opposed to zero copy) calling semantics

2.9 Sending UDP Data (zero copy interface)

The FreeRTOS_sendto() TCP/IP stack API function is used to send data to a UDP socket. Data can only be sent after the socket has been created, configured, and optionally bound to a local port number.

As detailed on the FreeRTOS_sendto() API reference section, FreeRTOS_sendto() can be used with standard calling semantics, or zero copy calling semantics. This section demonstrates the zero copy calling semantics.

The source code below shows a RTOS task that creates a UDP socket before entering a loop that sends a string to the socket (using the standard calling semantics) every 1 second (1000ms). The comments in the source code example provide important information on how to use network buffers when the zero copy interface is used.

static void vUDPSendingUsingZeroCopyInterface( void *pvParameters )
{
  xSocket_t xSocket;
  uint8_t *pucBuffer;
  struct freertos_sockaddr xDestinationAddress;
  BaseType_t lReturned;
  uint32_t ulCount = 0UL;
  const uint8_t *pucStringToSend = "Zero copy send message number ";
const TickType_t x1000ms = 1000UL / portTICK_PERIOD_MS;
/* 15 is added to ensure the number, \r\n and terminating zero fit. */
const size_t xStringLength = strlen( ( char * ) pucStringToSend ) + 15;

    /* Send strings to port 10000 on IP address 192.168.0.50. */
xDestinationAddress.sin_addr = FreeRTOS_inet_addr( "192.168.0.50" );
xDestinationAddress.sin_port = FreeRTOS_htons( 10000 );

    /* Create the socket. */
xSocket = FreeRTOS_socket( FREERTOS_AF_INET,
                   FREERTOS_SOCK_DGRAM,/*FREERTOS_SOCK_DGRAM for UDP.*/
                   FREERTOS_IPPROTO_UDP );

    /* Check the socket was created. */
configASSERT( xSocket != FREERTOS_INVALID_SOCKET );

    /* NOTE: FreeRTOS_bind() is not called. This will only work if
ipconfigALLOW_SOCKET_SEND_WITHOUT_BIND is set to 1 in FreeRTOSIPConfig.h. */

for( ;; )
{
    /* This RTOS task is going to send using the zero copy interface. The
data being sent is therefore written directly into a buffer that is
passed into, rather than copied into, the FreeRTOS_sendto() function.

First obtain a buffer of adequate length from the TCP/IP stack into which
the string will be written. */
pucBuffer = FreeRTOS_GetUDPPayloadBuffer( xStringLength, portMAX_DELAY );

    /* Check a buffer was obtained. */
configASSERT( pucBuffer );

    /* Create the string that is sent. */
memset( pucBuffer, 0x00, xStringLength );
sprintf( pucBuffer, "%s%lu\r\n", ucStringToSend, ulCount );

    /* Pass the buffer into the send function. ulFlags has the
FREERTOS_ZERO_COPY bit set so the TCP/IP stack will take control of the buffer rather than copy data out of the buffer. */

lReturned = FreeRTOS_sendto( xSocket,
    ( void * ) pucBuffer,
    strlen( ( const char * ) pucBuffer ) + 1,
    FREERTOS_ZERO_COPY,
    &xDestinationAddress,
    sizeof( xDestinationAddress ) );

if( lReturned == 0 )
{
    /* The send operation failed, so this RTOS task is still responsible for the buffer obtained from the TCP/IP stack. To ensure the buffer is not lost it must either be used again, or, as in this case, returned to the TCP/IP stack using FreeRTOS_ReleaseUDPPayloadBuffer(). pucBuffer can be safely re-used after this call. */
    FreeRTOS_ReleaseUDPPayloadBuffer( ( void * ) pucBuffer );
}
else
{
    /* The send was successful so the TCP/IP stack is now managing the buffer pointed to by pucBuffer, and the TCP/IP stack will return the buffer once it has been sent. pucBuffer can be safely re-used. */
}

ulCount++;

/* Wait until it is time to send again. */
vTaskDelay( x1000ms );
2.10 Receiving UDP Data (standard interface)

The FreeRTOS_recvfrom() TCP/IP stack API function is used to receive from a UDP socket. Data can only be received after the socket has been created, configured, and bound to a local port number.

As detailed on the FreeRTOS_recvfrom() API reference section, FreeRTOS_recvfrom() can be used with standard calling semantics, or zero copy calling semantics. This section demonstrates the standard calling semantics.

The source code below shows a RTOS task that creates a socket before entering a loop that receives data using the standard (as opposed to zero copy) calling semantics.

```c
static void vUDPReceivingUsingStandardInterface( void *pvParameters )
{
  long lBytes;
  uint8_t cReceivedString[ 60 ];
  struct freertos_sockaddr xClient, xBindAddress;
  uint32_t xClientLength = sizeof( xClient );
  xSocket_t xListeningSocket;

  /* Attempt to open the socket. */
  xListeningSocket = FreeRTOS_socket( FREERTOS_AF_INET,
      FREERTOS_SOCK_DGRAM,/*FREERTOS_SOCK_DGRAM for UDP.*/
      FREERTOS_IPPROTO_UDP );

  /* Check the socket was created. */
  configASSERT( xListeningSocket != FREERTOS_INVALID_SOCKET );

  /* Bind to port 10000. */
  xBindAddress.sin_port = FreeRTOS_htons( 10000 );
  FreeRTOS_bind( xListeningSocket, &xBindAddress, sizeof( xBindAddress ) );

  for( ;; )
  {
    /* Receive data from the socket. ulFlags is zero, so the standard
     interface is used. By default the block time is portMAX_DELAY, but it
     can be changed using FreeRTOS_setsockopt(). */
    lBytes = FreeRTOS_recvfrom( xListeningSocket,
```
ReceivedString, sizeof( cReceivedString ), 0, &xClient, &xClientLength );

if( lBytes > 0 ) {
    /* Data was received and can be process here. */
}
}
}

Figure 2-11 Example using FreeRTOS_recvfrom() with the standard (as opposed to zero copy) calling semantics

2.11 Receiving UDP Data (zero copy interface)

The FreeRTOS_recvfrom() TCP/IP stack API function is used to receive from a UDP socket. Data can only be received after the socket has been created, configured, and bound to a local port number.

As detailed on the FreeRTOS_recvfrom() API reference section, FreeRTOS_recvfrom() can be used with standard calling semantics, or zero copy calling semantics. This section demonstrates the zero copy calling semantics.

The source code below shows a RTOS task that creates a socket before entering a loop that receives data using the zero copy calling semantics. The comments in the source code provide important information on how to use network buffers when the zero copy option is used.

```
static void vUDPReceivingUsingZeroCopyInterface( void *pvParameters )
{
    int32_t lBytes;
    uint8_t *pucUDPPayloadBuffer;
    struct freertos_sockaddr xClient, xBindAddress;
    uint32_t xClientLength = sizeof( xClient ), ulIPAddress;
    xSocket_t xListeningSocket;

    /* Attempt to open the socket. */
    xListeningSocket = FreeRTOS_socket( FREERTOS_AF_INET,
```
FREERTOS_SOCK_DGRAM, /*FREERTOS_SOCK_DGRAM for UDP.
   */
FREERTOS_IPPROTO_UDP);

/* Check the socket was created. */
configASSERT(xListeningSocket != FREERTOS_INVALID_SOCKET);

/* Bind to port 10000. */
xBindAddress.sin_port = FreeRTOS_htons(10000);
FreeRTOS_bind(xListeningSocket, &xBindAddress, sizeof(xBindAddress));

for( ;; )
{
   /* Receive data from the socket. ulFlags has the zero copy bit set
      (FREERTOS_ZERO_COPY) indicating to the stack that a reference to the
      received data should be passed out to this RTOS task using the second
      parameter to the FreeRTOS_recvfrom() call. When this is done the
      IP stack is no longer responsible for releasing the buffer, and
      the RTOS task must return the buffer to the stack when it is no longer
      needed. By default the block time is portMAX_DELAY but it can be
      changed using FreeRTOS_setsockopt(). */
   lBytes = FreeRTOS_recvfrom(xListeningSocket,
                                &pucUDPPayloadBuffer,
                                0,
                                FREERTOS_ZERO_COPY,
                                &xClient,
                                &xClientLength);

   if( lBytes > 0 )
   {
      /* Data was received and can be processed here. */
   }
   if( lBytes >= 0 )
   {
      /* The receive was successful so this RTOS task is now responsible for
         the buffer. The buffer must be freed once it is no longer
         needed. */

      /*
         * The data can be processed here.
2.12 TCP/IP Stack Network Buffers Allocation Schemes

2.12.1 Network Data Buffers

Data being sent to the network or received from the network is placed in network buffers. Network buffer descriptors hold information about network buffers. The descriptors are pre-allocated, whereas the network buffers themselves are allocated as they are needed.

The total number of descriptors is set by the `ipconfigNUM_NETWORK_BUFFER_DESCRIPTORS` constant in `FreeRTOSIPConfig.h`. Pre-allocating the descriptors allows the application writer to limit the maximum number of network buffers that can exist at any one time in order to prevent memory exhaustion.

Different buffer allocation schemes suite different embedded applications, so OPENRTOS+TCP keeps the buffer allocation schemes as part of the TCP/IP stack’s portable layer. At the time of writing, two example buffer allocation schemes are provided - each with different trade-offs between simplicity, RAM usage efficiency, and performance. The two schemes are described in this section.

2.12.2 Buffer Allocation Schemes

2.12.2.1 Scheme 1: Implemented by `BufferAllocation_1.c`

**Description**

- Ethernet buffers are statically allocated by the embedded Ethernet peripheral driver (at compile time). This ensures the buffers can be aligned as required by the specific Ethernet hardware.

  `BufferAllocation_1.c` calls `vNetworkInterfaceAllocateRAMToBuffers()`, which must be provided by the peripheral driver. Information detailing the requirements of this function are provided in the functions that must be
provided by the port layer section of the Embedded Ethernet Driver Porting documentation section.

**Attributes**

- Fast run time performance.
- Ethernet buffers can be allocated and freed from interrupts, allowing for more efficient embedded Ethernet peripheral drivers.
- Inefficient use of RAM - all the buffers are the same size making BufferAllocation_1.c unsuitable for some RAM constrained embedded systems.
- More complex to configure and tune than the scheme implemented by BufferAllocation_2.c.
- Simpler to achieve any special buffer alignment requirements imposed by the embedded Ethernet peripheral DMA.
- Requires support from the network interface driver (see the description bullet points above).

**Usage**

- The ipconfigNUM_NETWORK_BUFFER_DESCRPTORS constant in FreeRTOSIPConfig.h defines both the total number of descriptors and the total number of buffers.
- The ipconfigNETWORK_MTU constant (defined in FreeRTOSIPConfig.h) defines the size of each Ethernet buffer (the total size being the defined MTU size plus the number of bytes needed by the Ethernet header).

### 2.12.2.2 Scheme 2: Implemented by BufferAllocation_2.c

**Description**

- Ethernet buffers of exactly the required size are dynamically allocated and freed as required. This requires a fast memory allocation scheme that does not suffer from fragmentation - at the time or writing it is recommended that heap_4.c or heap_5.c is used.

**Attributes**

- Extremely easy to use.
- Dynamic allocation results in slower run time performance when compared with the scheme implemented by BufferAllocation_1.c.
- Ethernet buffers cannot be allocated and freed from interrupts, necessitating the use of deferred interrupt handling tasks in embedded Ethernet peripheral drivers.
• Very efficient RAM usage - only the exact amount of RAM required is allocated making BufferAllocation_2.c particularly suited for RAM constrained small embedded systems.

Usage

• Ethernet buffers are allocated from the OPENRTOS heap. To avoid memory fragmentation problems, BufferAllocation_2.c can only be used reliably with a memory allocation scheme that combines adjacent free blocks of heap memory (a coalescence algorithm). The OPENRTOS memory allocation schemes implemented in heap_4.c and heap_5.c are suitable.
• The TCP/IP stack will recover from a failed attempt to allocate a network buffer, however, as the standard heap implementation is used such a failure will result in the malloc failed hook being called (if configUSE_MALLOC_FAILED_HOOK is set to 1 in FreeRTOSConfig.h).

2.13 OPENRTOS+TCP Configuration

OPENRTOS+TCP applications must provide a FreeRTOSIPConfig.h header file - in which the parameters described in this section can be defined.

The Configuration Examples section demonstrates how to set key configuration parameters for systems that need to minimise RAM consumption and systems that need to maximise throughput.

• Constants Affecting the TCP/IP Stack Task Execution Behaviour
  o ipconfigIP_TASK_PRIORITY
  o ipconfigIP_TASK_STACK_SIZE_WORDS
  o ipconfigUSE_NETWORK_EVENT_HOOK
  o ipconfigEVENT_QUEUE_LENGTH

• Debug, Trace and Logging Settings
  See also TCP/IP Trace Macros.
  o ipconfigHAS_DEBUG_PRINTF and FreeRTOS_debug_printf
  o ipconfigHAS_PRINTF and FreeRTOS_printf
  o ipconfigTCP_MAY_LOG_PORT

• Hardware and Driver Specific Settings
  o ipconfigBYTE_ORDER
  o ipconfigDRIVER_INCLUDED_TX_IP_CHECKSUM
  o ipconfigDRIVER_INCLUDED_RX_IP_CHECKSUM
  o ipconfigETHERNET_DRIVER_FILTERS_FRAME_TYPES
- TCP Specific Constants
  - ipconfigUSE_TCP
  - ipconfigTCP_TIME_TO_LIVE
  - ipconfigTCP_RX_BUFFER_LENGTH and ipconfigTCP_TX_BUFFER_LENGTH
  - ipconfigUSE_TCP_WIN
  - ipconfigTCP_WIN_SEG_COUNT
  - ipconfigUSE_TCP_TIMESTAMPS
  - ipconfigTCP_MSS
  - ipconfigTCP_KEEP_ALIVE
  - ipconfigTCP_KEEP_ALIVE_INTERVAL
  - ipconfigTCP_HANG_PROTECTION
  - ipconfigTCP_HANG_PROTECTION_TIME
- UDP Specific Constants
  - ipconfigUDP_TIME_TO_LIVE
  - ipconfigUDP_MAX_SEND_BLOCK_TIME_TICKS
  - ipconfigUDP_MAX_RX_PACKETS
- Other Constants Effecting Socket Behaviour
  - ipconfigINCLUDE_FULL_INET_ADDR
  - ipconfigALLOW_SOCKET_SEND_WITHOUT_BIND
  - ipconfigSOCK_DEFAULT_RECEIVE_BLOCK_TIME
  - ipconfigSOCK_DEFAULT_SEND_BLOCK_TIME
  - ipconfigSOCKET_HAS_USER_SEMAPHORE
  - ipconfigSUPPORT_SIGNALS
- Constants Affecting the ARP Behaviour
  - ipconfigARP_CACHE_ENTRIES
  - ipconfigMAX_ARP_RETRANSMISSIONS
  - ipconfigMAX_ARP_AGE
  - ipconfigARP_REVERSED_LOOKUP
  - ipconfigARPSTORES_REMOTE_ADDRESSES
- Constants Affecting DHCP and Name Service Behaviour
  - ipconfigUSE_DNS
  - ipconfigUSE_DNS_CACHE
  - ipconfigDNS_CACHE_ENTRIES
- ipconfigDNS_CACHE_NAME_LENGTH
- ipconfigUSE_LLMNR
- ipconfigUSE_NBNS
- ipconfigDNS_REQUEST_ATTEMPTS
- ipconfigUSE_DHCP
- ipconfigMAXIMUM_DISCOVER_TX_PERIOD
- ipconfigRAND32

- Constants Affecting IP and ICMP Behaviour
  - ipconfigCAN_FRAGMENT_OUTGOING_PACKETS
  - ipconfigREPLY_TO_INCOMING_PINGS
  - ipconfigSUPPORT_OUTGOING_PINGS
2.13.1 Constants Affecting the TCP/IP Stack Task Execution Behaviour

2.13.1.1 ipconfigIP_TASK_PRIORITY

The TCP/IP stack executes its own RTOS task (although any application RTOS task can make use of its services through the published sockets API). ipconfigIP_TASK_PRIORITY sets the priority of the RTOS task that executes the TCP/IP stack.

The priority is a standard OPENRTOS task priority so can take any value from 0 (the lowest priority) to (configMAX_PRIORITIES - 1) (the highest priority). configMAX_PRIORITIES is a standard OPENRTOS configuration parameter defined in FreeRTOSConfig.h, not FreeRTOSIPConfig.h.

Consideration needs to be given as to the priority assigned to the RTOS task executing the TCP/IP stack relative to the priority assigned to tasks that use the TCP/IP stack.

2.13.1.2 ipconfigIP_TASK_STACK_SIZE_WORDS

The size, in words (not bytes), of the stack allocated to the OPENRTOS+TCP RTOS task. OPENRTOS includes optional stack overflow detection.

2.13.1.3 ipconfigUSE_NETWORK_EVENT_HOOK

If ipconfigUSE_NETWORK_EVENT_HOOK is set to 1 then OPENRTOS+TCP will call the network event hook at the appropriate times. If ipconfigUSE_NETWORK_EVENT_HOOK is not set to 1 then the network event hook will never be called.

2.13.1.4 ipconfigEVENT_QUEUE_LENGTH

An OPENRTOS queue is used to send events from application tasks to the IP stack. ipconfigEVENT_QUEUE_LENGTH sets the maximum number of events that can be queued for processing at any one time. The event queue must be a minimum of 5 greater than the total number of network buffers.

2.13.2 Debug, Trace and Logging Settings

2.13.2.1 Trace Macros

Information on the available TCP/IP stack trace macros is provided in a separate section.

2.13.2.2 ipconfigHAS_DEBUG_PRINTF and FreeRTOS_debug_printf

The TCP/IP stack outputs debugging messages by calling the FreeRTOS_debug_printf macro. To obtain debugging messages set ipconfigHAS_DEBUG_PRINTF to 1, then define
FreeRTOS_debug_printf() to a function that takes a printf() style format string and variable number of inputs, and sends the formatted messages to an output of your choice.

Do not define FreeRTOS_debug_printf if ipconfigHAS_DEBUGPRINTF is set to 0.

The following code is taken from the OPENRTOS+TCP example for the RTOS's Win32 simulator, which has the ability to output debugging messages to a UDP port, standard out, and to a disk file:

```c
/* Prototype for the function function that actually performs the output. */
extern void vLoggingPrintf( const char *pcFormatString, ... );

/* Set to 1 to print out debug messages. If ipconfigHAS_DEBUGPRINTF is set to 1 then FreeRTOS_debug_printf should be defined to the function used to print out the debugging messages. */
#define ipconfigHAS_DEBUGPRINTF 0
#if( ipconfigHAS_DEBUGPRINTF == 1 )
    #define FreeRTOS_debug_printf(X) vLoggingPrintf X
#endif
```

Figure 2-13 Defining ipconfigHAS_DEBUGPRINTF and FreeRTOS_debug_printf in FreeRTOSIPConfig.h

The function that performs the output (vLoggingPrintf() in the code above) must be reentrant.

2.13.2.3 ipconfigHAS_PRINTF and FreeRTOS_printf

Some of the TCP/IP stack demo applications generate output messages. The TCP/IP stack outputs these messages by calling the FreeRTOS_printf macro. To obtain the demo application messages set ipconfigHAS_PRINTF to 1, then define FreeRTOS_printf() to a function that takes a printf() style format string and variable number of inputs, and sends the formatted messages to an output of your choice.

Do not define FreeRTOS_printf if ipconfigHAS_PRINTF is set to 0.

The following code is taken from the OPENRTOS+TCP example for the RTOS's Win32 simulator, which has the ability to output application messages to a UDP port, standard out, and to a disk file:
/* Prototype for the function function that actually performs the output. */
extern void vLoggingPrintf( const char *pcFormatString, ... );

/* Set to 1 to print out application messages. If ipconfigHAS_PRINTF is set to 1 then FreeRTOS_printf should be defined to the function used to print out the application messages. */
#define ipconfigHAS_PRINTF 0
#if( ipconfigHAS_PRINTF == 1 )
  #define FreeRTOS_printf(X) vLoggingPrintf X
#endif

Figure 2-14 Defining ipconfigHAS_PRINTF and FreeRTOS_printf in FreeRTOSIPConfig.h

The function that performs the output (vLoggingPrintf() in the code above) must be reentrant.

2.13.2.4 ipconfigTCP_MAY_LOG_PORT( x )

ipconfigTCP_MAY_LOG_PORT( x ) can be defined to specify which port numbers should or should not be logged by FreeRTOS_printf(). For example, the following definition will not generate log messages for ports 23 or 2402:

#define ipconfigTCP_MAY_LOG_PORT(xPort) ( ( ( xPort ) != 23 ) && ( ( xPort ) != 2402 ) )

Figure 2-15 Filtering Log Messages

2.13.3 Hardware and Driver Specific Settings

2.13.3.1 ipconfigBYTE_ORDER

If the microcontroller on which OPENRTOS+TCP is running is big endian then ipconfigBYTE_ORDER must be set to pdFREERTOS_BIG_ENDIAN. If the microcontroller is little endian then ipconfigBYTE_ORDER must be set to pdFREERTOS_LITTLE_ENDIAN.
2.13.3.2 ipconfigDRIVER_INCLUDED_TX_IP_CHECKSUM

If the network driver or network hardware is calculating the IP, TCP and UDP checksums of outgoing packets then set ipconfigDRIVER_INCLUDED_TX_IP_CHECKSUM to 1, otherwise set ipconfigDRIVER_INCLUDED_TX_IP_CHECKSUM to 0.

Throughput and processor load are greatly improved by implementing drivers that make use of hardware checksum calculations.

2.13.3.3 ipconfigDRIVER_INCLUDED_RX_IP_CHECKSUM

If the network driver or network hardware is calculating the IP, TCP and UDP checksums of incoming packets, and discarding packets that are found to contain invalid checksums, then set ipconfigDRIVER_INCLUDED_RX_IP_CHECKSUM to 1, otherwise set ipconfigDRIVER_INCLUDED_RX_IP_CHECKSUM to 0.

Throughput and processor load are greatly improved by implementing drivers that make use of hardware checksum calculations.

2.13.3.4 ipconfigETHERNET_DRIVER_FILTERS_FRAME_TYPES

Ethernet/hardware MAC addresses are used to address Ethernet frames. If the network driver or hardware is discarding packets that do not contain a MAC address of interest then set ipconfigETHERNET_DRIVER_FILTERS_FRAME_TYPES to 1. Otherwise set ipconfigETHERNET_DRIVER_FILTERS_FRAME_TYPES to 0.

Throughput and processor load are greatly improved by implementing network address filtering in hardware. Most network interfaces allow multiple MAC addresses to be defined so filtering can allow through the unique hardware address of the node, the broadcast address, and various multicast addresses.

2.13.3.5 ipconfigETHERNET_DRIVER_FILTERS_PACKETS

Whereas ipconfigETHERNET_DRIVER_FILTERS_FRAME_TYPES is used to specify whether or not the network driver or hardware filters Ethernet frames, ipconfigETHERNET_DRIVER_FILTERS_PACKETS is used to specify whether or not the network driver filters the IP, UDP or TCP data within the Ethernet frame.

The TCP/IP stack is only interested in receiving data that is either addresses to a socket (IP address and port number) on the local node, or is a broadcast or multicast packet. Throughput and process load can be greatly improved by preventing packets that do not meet these criteria from being sent...
to the TCP/IP stack. **OPENRTOS** provides some features that allow such filtering to take place in the network driver. For example, xPortHasUDPSo() can be used as follows:

```c
if( ( xPortHasUdpSocket( xUDPHeader->usDestinationPort ) )
    #if( ipconfigUSE_DNS == 1 ) /* DNS is also UDP. */
        || ( xUDPHeader->usSourcePort == FreeRTOS_ntohs( ipDNS_PORT ) )
    #endif
    #if( ipconfigUSE_LLMNR == 1 ) /* LLMNR is also UDP. */
        || ( xUDPHeader->usDestinationPort == FreeRTOS_ntohs( ipLLMNR_PORT ) )
    #endif
    #if( ipconfigUSE_NBNS == 1 ) /* NBNS is also UDP. */
        || ( xUDPHeader->usDestinationPort == FreeRTOS_ntohs( ipNBNS_PORT ) )
    #endif
){
    /* Forward packet to the IP-stack. */
}
else
{
    /* Discard the UDP packet. */
}
```

**Figure 2-16 Example of filtering UDP packets**

### 2.13.3.6 ipconfigNETWORK_MTU

The MTU is the maximum number of bytes the payload of a network frame can contain. For normal Ethernet V2 frames the maximum MTU is 1500 (although a lower number may be required for Internet routing). Setting a lower value can save RAM, depending on the buffer management scheme used. If ipconfigCAN_FRAGMENT_OUTGOING_PACKETS is 1 then (ipconfigNETWORK_MTU - 28) must be divisible by 8.

If ipconfigNETWORK_MTU is not defined then the following defaults will be applied:

```c
#ifndef ipconfigNETWORK_MTU
```
2.13.3.7 ipconfigNUM_NETWORK_BUFFER_DESCRPTORS

ipconfigNUM_NETWORK_BUFFER_DESCRPTORS defines the total number of network buffer that are available to the TCP/IP stack. The total number of network buffers is limited to ensure the total amount of RAM that can be consumed by the TCP/IP stack is capped to a pre-determinable value. How the storage area is actually allocated to the network buffer structures is not fixed, but part of the portable layer. The simplest scheme simply allocates the exact amount of storage as it is required.

More information on network buffers and network buffer descriptors is provided in the sections that describe porting OPENRTOS+TCP to other hardware and the pxGetNetworkBufferWithDescriptor() porting specific API function.

2.13.3.8 ipconfigFILTER_OUT_NON_ETHERNET_II_FRAMES

If ipconfigFILTER_OUT_NON_ETHERNET_II_FRAMES is set to 1 then Ethernet frames that are not in Ethernet II format will be dropped. This option is included for potential future IP stack developments.

2.13.3.9 ipconfigUSE_LINKED_RX_MESSAGES

When ipconfigUSE_LINKED_RX_MESSAGES is set to 1 it is possible to reduce CPU load during periods of heavy network traffic by linking multiple received packets together, then passing all the linked packets to the IP RTOS task in one go.

2.13.4 TCP Specific Constants

2.13.4.1 ipconfigUSE_TCP

Set ipconfigUSE_TCP to 1 to enable TCP. If ipconfigUSE_TCP is set to 0 then only UDP is available.
2.13.4.2 ipconfigTCP_TIME_TO_LIVE

Defines the Time To Live (TTL) values used in outgoing TCP packets.

2.13.4.3 ipconfigTCP_RX_BUFFER_LENGTH and ipconfigTCP_TX_BUFFER_LENGTH

Each TCP socket has a buffer for reception and a separate buffer for transmission.

The default buffer size is (4 * ipconfigTCP_MSS).

FreeRTOS_setsockopt() can be used with the FREERTOS_SO_RCVBUF and FREERTOS_SO_SNDBUF parameters to set the receive and send buffer sizes respectively - but this must be done between the socket being created and the buffers used by the socket being created. The receive buffer is not created until data is actually received, and the transmit buffer is not created until data is actually sent to the socket for transmission. Once the buffers have been created their size cannot be changed.

If a listening socket creates a new socket in response to an incoming connect request then the new socket will inherit the buffers sizes of the listening socket.

2.13.4.4 ipconfigUSE_TCP_WIN

Sliding Windows allows messages to arrive out-of-order.

Set ipconfigUSE_TCP_WIN to 1 to include sliding window behaviour in TCP sockets. Set ipconfigUSE_TCP_WIN to 0 to exclude sliding window behaviour in TCP sockets.

Sliding windows can increase throughput while minimising network traffic at the expense of consuming more RAM.

The size of the sliding window can be changed from its default using the FREERTOS_SO_WIN_PROPERTIES parameter to FreeRTOS_setsockopt(). The sliding window size is specified in units of MSS (so if the MSS is set to 200 bytes then a sliding window size of 2 is equal to 400 bytes) and must always be smaller than or equal to the size of the internal buffers in both directions.

If a listening socket creates a new socket in response to an incoming connect request then the new socket will inherit the sliding window sizes of the listening socket.
2.13.4.5 ipconfigTCP_WIN_SEG_COUNT

If ipconfigUSE_TCP_WIN is set to 1 then each socket will use a sliding window. Sliding windows allow messages to arrive out-of-order, and OPENRTOS+TCP uses window descriptors to track information about the packets in a window.

A pool of descriptors is allocated when the first TCP connection is made. The descriptors are shared between all the sockets. ipconfigTCP_WIN_SEG_COUNT set the number of descriptors in the pool, and each descriptor is approximately 64 bytes.

As an example: If a system will have at most 16 simultaneous TCP connections, and each connection will have an Rx and Tx window of at most 8 segments, then the worst case maximum number of descriptors that will be required is 256 ( 16 * 2 * 8 ). However, the practical worst case is normally much lower than this as most packets will arrive in order.

2.13.4.6 ipconfigUSE_TCP_TIMESTAMPS

TCP time stamp functionality is available, but its usage is quite limited. Time-stamps can only be used if the initial SYN packet contains the time-stamp option. In most cases, incoming connection won't have the time-stamp option set.

Set ipconfigUSE_TCP_TIMESTAMPS to 1 to include TCP time stamp functionality. Set ipconfigUSE_TCP_TIMESTAMPS to 0 to exclude TCP time stamp functionality.

2.13.4.7 ipconfigTCP_MSS

Sets the MSS value (in bytes) for all TCP packets.

Note that OPENRTOS+TCP contains checks that the defined ipconfigNETWORK_MTU and ipconfigTCP_MSS values are consistent with each other.

2.13.4.8 ipconfigTCP_KEEP_ALIVE

Sockets that are connected but do not transmit any data for an extended period can be disconnected by routers or firewalls that time out. This can be avoided at the application level by ensuring the application periodically sends a packet. Alternatively OPENRTOS+TCP can be configured to automatically send keep alive messages when it detects that a connection is dormant. Note that, while having OPENRTOS+TCP automatically send keep alive messages is the more convenient method, it is also the least reliable method because some routers will discard keep alive messages.
Set ipconfigTCP_KEEP_ALIVE to 1 to have OPENRTOS+TCP periodically send keep alive messages on connected but dormant sockets. Set ipconfigTCP_KEEP_ALIVE to 0 to prevent the automatic transmission of keep alive messages.

If OPENRTOS+TCP does not receive a reply to a keep alive message then the connection will be broken and the socket will be marked as closed. Subsequent FreeRTOS_recv() calls on the socket will return -pdFREERTOS_ERRNO_ENOTCONN.

2.13.4.9 ipconfigTCP_KEEP_ALIVE_INTERVAL

If ipconfigTCP_KEEP_ALIVE is set to 1 then ipconfigTCP_KEEP_ALIVE_INTERVAL sets the interval in seconds between successive keep alive messages. Keep alive messages are not sent at all unless ipconfigTCP_KEEP_ALIVE_INTERVAL seconds have passed since the last packet was sent or received.

2.13.4.10 ipconfigTCP_HANG_PROTECTION

If ipconfigTCP_HANG_PROTECTION is set to 1 then OPENRTOS+TCP will mark a socket as closed if there is no status change on the socket within the period of time specified by ipconfigTCP_HANG_PROTECTION_TIME.

2.13.4.11 ipconfigTCP_HANG_PROTECTION_TIME

If ipconfigTCP_HANG_PROTECTION is set to 1 then ipconfigTCP_HANG_PROTECTION_TIME sets the interval in seconds between the status of a socket last changing and the anti-hang mechanism marking the socket as closed.

2.13.5 UDP Specific Constants

2.13.5.1 ipconfigUDP_TIME_TO_LIVE

Defines the Time To Live (TTL) values used in outgoing UDP packets.

2.13.5.2 ipconfigUDP_MAX_SEND_BLOCK_TIME_TICKS

Sockets have a send block time attribute. If FreeRTOS_sendto() is called but a network buffer cannot be obtained then the calling RTOS task is held in the Blocked state (so other tasks can continue to executed) until either a network buffer becomes available or the send block time expires. If the send block time expires then the send operation is aborted.

The maximum allowable send block time is capped to the value set by ipconfigUDP_MAX_SEND_BLOCK_TIME_TICKS. Capping the maximum allowable send block time prevents prevents a deadlock occurring when all the network buffers are in use and the tasks that
process (and subsequently free) the network buffers are themselves blocked waiting for a network buffer.

ipconfigUDP_MAX_SEND_BLOCK_TIME_TICKS is specified in RTOS ticks. A time in milliseconds can be converted to a time in ticks by dividing the time in milliseconds by portTICK_PERIOD_MS.

2.13.5.3 ipconfigUDP_MAX_RX_PACKETS

ipconfigUDP_MAX_RX_PACKETS defines the maximum number of packets that can exist in the Rx queue of a UDP socket. For example, if ipconfigUDP_MAX_RX_PACKETS is set to 5 and there are already 5 packets queued on the UDP socket then subsequent packets received on that socket will be dropped until the queue length is less than 5 again.

2.13.6 Other Constants Effecting Socket Behaviour

2.13.6.1 ipconfigINCLUDE_FULL_INET_ADDR

Implementing FreeRTOS_inet_addr() necessitates the use of string handling routines, which are relatively large. To save code space the full FreeRTOS_inet_addr() implementation is made optional, and a smaller and faster alternative called FreeRTOS_inet_addr_quick() is provided. FreeRTOS_inet_addr() takes an IP in decimal dot format (for example, "192.168.0.1") as its parameter. FreeRTOS_inet_addr_quick() takes an IP address as four separate numerical octets (for example, 192, 168, 0, 1) as its parameters. If ipconfigINCLUDE_FULL_INET_ADDR is set to 1 then both FreeRTOS_inet_addr() and FreeRTOS_inet_addr_quick() are available. If ipconfigINCLUDE_FULL_INET_ADDR is not set to 1 then only FreeRTOS_inet_addr_quick() is available.

2.13.6.2 ipconfigALLOW_SOCKET_SEND_WITHOUT_BIND

The address of a socket is the combination of its IP address and its port number. FreeRTOS_bind() is used to manually allocate a port number to a socket (to "bind" the socket to a port), but manual binding is not normally necessary for client sockets (those sockets that initiate outgoing connections rather than wait for incoming connections on a known port number). If ipconfigALLOW_SOCKET_SEND_WITHOUT_BIND is set to 1 then calling FreeRTOS_sendto() on a socket that has not yet been bound will result in the IP stack automatically binding the socket to a port number from the range socketAUTO_PORT_ALLOCATION_START_NUMBER to 0xffff. If ipconfigALLOW_SOCKET_SEND_WITHOUT_BIND is set to 0 then calling FreeRTOS_sendto() on a socket that has not yet been bound will result in the send operation being aborted.

2.13.6.3 ipconfigSOCK_DEFAULT_RECEIVE_BLOCK_TIME

API functions used to read data from a socket can block to wait for data to become available. ipconfigSOCK_DEFAULT_RECEIVE_BLOCK_TIME sets the default block time defined in RTOS
ticks. If \texttt{ipconfigSOCK\_DEFAULT\_RECEIVE\_BLOCK\_TIME} is not defined then the default block time will be set to \texttt{portMAX\_DELAY} - meaning an RTOS task that is blocked on a socket read will not leave the Blocked state until data is available. Note tasks in the Blocked state do not consume any CPU time.

\texttt{ipconfigSOCK\_DEFAULT\_RECEIVE\_BLOCK\_TIME} is specified in ticks. The macros \texttt{pdMS\_TO\_TICKS()} and \texttt{portTICK\_PERIOD\_MS} can both be used to convert a time specified in milliseconds to a time specified in ticks.

The time out time can be changed at any time using the \texttt{FREERTOS\_SO\_RCVTIMEO} parameter with \texttt{FreeRTOS\_setsockopt()}. \textbf{Note:} Infinite block times should be used with extreme care in order to avoid a situation where all tasks are blocked indefinitely to wait for another RTOS task (which is also blocked indefinitely) to free a network buffer.

A socket can be set to non-blocking mode by setting both the send and receive block time to 0. This might be desirable when an RTOS task is using more than one socket - in which case blocking can instead be performed on all the sockets at once using \texttt{FreeRTOS\_select()}, or the RTOS task can set \texttt{ipconfigSOCKET\_HAS\_USER\_SEMAPHORE} to one then block on its own semaphore.

\textbf{2.13.6.4 ipconfigSOCK\_DEFAULT\_SEND\_BLOCK\_TIME}

When writing to a socket the write may not be able to proceed immediately. For example, depending on the configuration, a write might have to wait for a network buffer to become available. API functions used to write data to a socket can block to wait for the write to succeed. \texttt{ipconfigSOCK\_DEFAULT\_SEND\_BLOCK\_TIME} sets the default block time defined in RTOS ticks. If \texttt{ipconfigSOCK\_DEFAULT\_SEND\_BLOCK\_TIME} is not defined then the default block time will be set to \texttt{portMAX\_DELAY} - meaning an RTOS task that is blocked on a socket read will not leave the Blocked state until data is available. Note tasks in the Blocked state do not consume any CPU time.

\texttt{ipconfigSOCK\_DEFAULT\_RECEIVE\_BLOCK\_TIME} is specified in ticks. The macros \texttt{pdMS\_TO\_TICKS()} and \texttt{portTICK\_PERIOD\_MS} can both be used to convert a time specified in milliseconds to a time specified in ticks.

The time out time can be changed at any time using the \texttt{FREERTOS\_SO\_SNDTIMEO} parameter with \texttt{FreeRTOS\_setsockopt()}. \textbf{Note:} Infinite block times should be used with extreme care in order to avoid a situation where all tasks are blocked indefinitely to wait for another RTOS task (which is also blocked indefinitely) to free a network buffer.

A socket can be set to non-blocking mode by setting both the send and receive block time to 0. This might be desirable when an RTOS task is using more than one socket - in which case blocking can
instead by performed on all the sockets at once using FreeRTOS_select(), or the RTOS task can set ipconfigSOCKET_HAS_USER_SEMAPHORE to one then block on its own semaphore.

A socket can be set to non-blocking mode by setting both the send and receive block time to 0.

2.13.6.5 ipconfigSOCKET_HAS_USER_SEMAPHORE

By default sockets will block on a send or receive that cannot complete immediately. See the description of the ipconfigSOCK_DEFAULT_RECEIVE_BLOCK_TIME and ipconfigSOCK_DEFAULT_SEND_BLOCK_TIME parameters.

If an RTOS task is using multiple sockets and cannot block on one socket at a time then the sockets can be set into non-blocking mode, and the RTOS task can block on all the sockets at once by either using the FreeRTOS_select() function or by setting ipconfigSOCKET_HAS_USER_SEMAPHORE to 1, using the FREERTOS_SO_SET_SEMAPHORE parameter with FreeRTOS_setsockopt() to provide a semaphore to the socket, and then blocking on the semaphore. The semaphore will be given when any of the sockets are able to proceed - at which time the RTOS task can inspect all the sockets individually using non blocking API calls to determine which socket caused it to unblock.

2.13.6.6 ipconfigSUPPORT_SIGNALS

If ipconfigSUPPORT_SIGNALS is set to 1 then the FreeRTOS_SignalSocket() API function is included in the build. FreeRTOS_SignalSocket() can be used to send a signal to a socket, the result of which is that any task blocked on a read from the socket will leave the Blocked state (abort the blocking read operation).

2.13.7 Constants Affecting the ARP Behaviour

2.13.7.1 ipconfigARP_CACHE_ENTRIES

The ARP cache is a table that maps IP addresses to MAC addresses.

The IP stack can only send a UDP message to a remove IP address if it knows the MAC address associated with the IP address, or the MAC address of the router used to contact the remote IP address. When a UDP message is received from a remote IP address the MAC address and IP address are added to the ARP cache. When a UDP message is sent to a remote IP address that does not already appear in the ARP cache then the UDP message is replaced by an ARP message that solicits the required MAC address information.

ipconfigARP_CACHE_ENTRIES defines the maximum number of entries that can exist in the ARP table at any one time.
2.13.7.2 ipconfigMAX_ARP_RETRANSMISSIONS

ARP requests that do not result in an ARP response will be re-transmitted a maximum of ipconfigMAX_ARP_RETRANSMISSIONS times before the ARP request is aborted.

2.13.7.3 ipconfigMAX_ARP_AGE

ipconfigMAX_ARP_AGE defines the maximum time between an entry in the ARP table being created or refreshed and the entry being removed because it is stale. New ARP requests are sent for ARP cache entries that are nearing their maximum age.

ipconfigMAX_ARP_AGE is specified in tens of seconds, so a value of 150 is equal to 1500 seconds (or 25 minutes).

2.13.7.4 ipconfigARP_REVERSED_LOOKUP

If ipconfigARP_REVERSED_LOOKUP is set to 1 then the xGetARPCacheEntryByMac() function is available for use. xGetARPCacheEntryByMac() performs an IP address look up from a MAC address.

2.13.7.5 ipconfigARPSTORES_REMOTE_ADDRESSES

ipconfigARPSTORES_REMOTE_ADDRESSES is provided for the case when a message that requires a reply arrives from the Internet, but from a computer attached to a LAN rather than via the defined gateway. Before replying to the message the TCP/IP stack RTOS task will loop up the message's IP address in the ARP table - but if ipconfigARPSTORES_REMOTE_ADDRESSES is set to 0 then ARP will return the MAC address of the defined gateway because the destination address is outside of the netmask. That might prevent the reply reaching its intended destination.

If ipconfigARPSTORES_REMOTE_ADDRESSES is set to 1 then remote addresses will also be stored in the ARP table, along with the MAC address from which the message was received. This can allow the message in the scenario above to be routed and delivered correctly.

2.13.8 Constants Affecting DHCP and Name Service Behaviour

2.13.8.1 ipconfigUSE_DNS

Set ipconfigUSE_DNS to 1 to include a basic DNS client/resolver. DNS is used through the FreeRTOS_gethostbyname() API function.
2.13.8.2 ipconfigDNS_REQUEST_ATTEMPTS

ipconfigDNS_REQUEST_ATTEMPTS sets the number of times ARP will attempt to obtain an ARP response before giving up.

2.13.8.3 ipconfigUSE_DNS_CACHE

If ipconfigUSE_DNS_CACHE is set to 1 then the DNS cache will be enabled. If ipconfigUSE_DNS_CACHE is set to 0 then the DNS cache will be disabled.

2.13.8.4 ipconfigDNS_CACHE_ENTRIES

If ipconfigUSE_DNS_CACHE is set to 1 then ipconfigDNS_CACHE_ENTRIES defines the number of entries in the DNS cache.

2.13.8.5 ipconfigDNS_CACHE_NAME_LENGTH

The maximum number of characters a DNS host name can take, including the NULL terminator.

2.13.8.6 ipconfigUSE_LLMNR

Set ipconfigUSE_LLMNR to 1 to include LLMNR.

2.13.8.7 ipconfigUSE_NBNS

Set ipconfigUSE_NBNS to 1 to include NBNS.

2.13.8.8 ipconfigUSE_DHCP

If ipconfigUSE_DHCP is 1 then OPENRTOS+TCP will attempt to retrieve an IP address, netmask, DNS server address and gateway address from a DHCP server - and revert to using the defined static address if an IP address cannot be obtained.

If ipconfigUSE_DHCP is 0 then OPENRTOS+TCP will not attempt to obtain its address information from a DHCP server, and instead immediately use the defined static address information.

2.13.8.9 ipconfigMAXIMUM_DISCOVER_TX_PERIOD

When ipconfigUSE_DHCP is set to 1, DHCP requests will be sent out at increasing time intervals until either a reply is received from a DHCP server and accepted, or the interval between transmissions reaches ipconfigMAXIMUM_DISCOVER_TX_PERIOD. The TCP/IP stack will revert to using the static IP address passed as a parameter to FreeRTOS_IPInit() if the re-transmission time interval reaches ipconfigMAXIMUM_DISCOVER_TX_PERIOD without a DHCP reply being received.
2.13.8.10 ipconfigRAND32()

ipconfigRAND32() is called by the TCP/IP stack to generate a random number that is then used as a DHCP transaction number. Random number generation is performed via this macro to allow applications to use their own random number generation method. For example, it might be possible to generate a random number by sampling noise on an analogue input.

2.13.9 Constants Affecting IP and ICMP Behaviour

2.13.9.1 ipconfigCAN_FRAGMENT_OUTGOING_PACKETS

If ipconfigCAN_FRAGMENT_OUTGOING_PACKETS is set to 1 then UDP packets that contain more data than will fit in a single network frame will be fragmented across multiple IP packets. Also see the ipconfigNETWORK_MTU setting. If ipconfigCAN_FRAGMENT_OUTGOING_PACKETS is 1 then (ipconfigNETWORK_MTU - 28) must be divisible by 8. Setting ipconfigCAN_FRAGMENT_OUTGOING_PACKETS to 1 will increase both the code size and execution time.

2.13.9.2 ipconfigREPLY_TO_INCOMING_PINGS

If ipconfigREPLY_TO_INCOMING_PINGS is set to 1 then the TCP/IP stack will generate replies to incoming ICMP echo (ping) requests.

2.13.9.3 ipconfigSUPPORT_OUTGOING_PINGS

If ipconfigSUPPORT_OUTGOING_PINGS is set to 1 then the FreeRTOS_SendPingRequest() API function is available.

2.14 OPENRTOS+TCP Configuration Examples

The OPENRTOS+TCP configuration file section documents each TCP/IP stack configuration option. This section provides suggestions on how to set key TCP parameters to tailor the TCP/IP stack to minimise its RAM consumption, and then to maximise its throughput. Note that minimising RAM consumption and maximising throughput are somewhat mutually exclusive objectives - up to a point the more RAM that is allocated to the TCP/IP stack the higher the throughput will be.

In all cases RAM consumption and CPU load can be minimised by using network drivers that make full use of any hardware features available, such as checksum offloading and MAC address filtering. If the hardware does not offer these facilities then the network driver can still improve RAM consumption and CPU load by performing any filtering it can before passing packets to the TCP/IP stack.
2.14.1 TCP/IP Stack Configuration To Minimise RAM Consumption

If you have a tiny CPU with less than 64KB of RAM, do not use sliding Windows:

```
#define ipconfigUSE_TCP_WIN 0
```

The window size will be fixed to 1 MSS. The buffer size can be declared as 1 or 2 MSS:

```
#define ipconfigTCP_TX_BUFFER_LENGTH (2 * ipconfigTCP_MSS)
#define ipconfigTCP_RX_BUFFER_LENGTH (2 * ipconfigTCP_MSS)
```

If RAM is really constrained then use smaller segments:

```
#define ipconfigNETWORK_MTU 576
#define ipconfigTCP_MSS 52
```

All peers will understand this and only send small packets.

Only allocate the minimum number of network buffer descriptors you can get away with. This also has the effect of preventing high network traffic resulting in memory exhaustion as network buffers will not be allocated if no descriptors are available:

```
#define ipconfigNUM_NETWORK_BUFFER_DESCRIPTORS [a small number]
```

Finally, ensure only the amount of RAM that is actually required is allocated at any given time by using the BufferAllocation_2.c.

2.14.2 TCP/IP Stack Configuration to Maximise Throughput

First - note that network driver implementation is crucial in the throughput that can be obtained. Network drivers should copy as little data as possible (none!), use DMAs, calculate checksums in hardware, make full use of hardware filtering, and make use of software filtering where hardware filtering is not possible (to ensure only packets that actually require processing are passed to the TCP/IP stack).

Second the advanced features of the TCP/IP stack, including the functionality mentioned in the description of the ipconfigUSE_LINKED_RX_MESSAGES parameter and the callback API are provided with the aim of maximising throughput - however these features are considered to be for advanced users only.

If you have enough RAM then the following declarations will help performance:
#define ipconfigNETWORK_MTU 1526
#define ipconfigTCP_MSS 1460
#define ipconfigTCP_TX_BUFFER_LENGTH ( 16 * ipconfigTCP_MSS )
#define ipconfigTCP_RX_BUFFER_LENGTH ( 16 * ipconfigTCP_MSS )

On a LAN, the sliding windows will get a size of ( 8 * ipconfigTCP_MSS ), meaning that only one out of 8 packets will receive an ACK.

For more flexibility FreeRTOS_setsockopt() can be used to set sizes between a socket being created and the same socket getting used.

```c
/* Declare the variable used to set the configuration parameters. */
xWinProperties_t xWinProps;

/* Start with everything set to 0. */
memset( &xWinProps, '\0', sizeof( xWinProps ) );

/* Configure as required. */
xWinProps.ulTxBufSize = 24 * ipconfigTCP_MSS;
xWinProps.ulTxWinSize = 8;
xWinProps.ulRxBufSize = 24 * ipconfigTCP_MSS;
xWinProps.ulRxWinSize = 8;

/* Set the socket options. */
FreeRTOS_setsockopt( sock,
    0,
    FREERTOS_SO_WIN_PROPERTIES,
    ( void * ) &xWinProps,
    sizeof( xWinProps ) );
```

Figure 2-17 Using FreeRTOS_setsockopt() to set TCP/IP options

Usually nothing is to be gained by setting the windows larger than ( 8 * MSS ), unless the CPU and MAC are very fast and connected to a 1 Gbit LAN. Using larger buffers for reception does make sense in case the end-point is slow, for instance if all the received data must be written to an SD-card.
Finally, ensure fast and deterministic buffer allocation that can also be used directly from within the MAC interrupt by using BufferAllocation_1.c.

### 2.15 TCP/IP Specific Trace Hook Macros

Trace hook macros allow you to collect data while your OPENRTOS+TCP application is running. The data can be used for both debugging and optimisation purposes.

Key points of interest within the RTOS’s TCP source code contain macros that an application can define for the purpose of providing application specific trace functionality. The application need only implement the macros of interest - unimplemented macros will remain empty (not generate any code) by default.

It is recommended to implement trace macros in a header file, then include the header file at the bottom of FreeRTOSIPConfig.h.

The OPENRTOS+TCP example that runs in the Windows simulator (available for download from this website) uses the trace macros to collect TCP/IP stack run time information that can then be viewed using the TCP/IP CLI interface.

The table below details the macros that can be defined.

<table>
<thead>
<tr>
<th>Macro definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iptraceNETWORK_DOWN()</td>
<td>Called when the network driver indicates that the network connection has been lost (not implemented by all network drivers).</td>
</tr>
<tr>
<td>iptraceNETWORK_BUFFER_RELEASED(pxBufferAddress)</td>
<td>Called when the network buffer at address pxBufferAddress is released back to the TCP/IP stack.</td>
</tr>
<tr>
<td>iptraceNETWORK_BUFFER_OBTAINED(pxBufferAddress)</td>
<td>Called when the network buffer at address pxBufferAddress is obtained from the TCP/IP stack by an RTOS task.</td>
</tr>
<tr>
<td>iptraceNETWORK_BUFFER_OBTAINED_FROM_ISR()</td>
<td>Called when the network buffer at</td>
</tr>
<tr>
<td>Macro definition</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>pxBufferAddress )</td>
<td>address pxBufferAddress is obtained from the TCP/IP stack by an interrupt service routine.</td>
</tr>
<tr>
<td>iptraceFAILED_TO_OBTAIN_NETWORK_BUFFER()</td>
<td>Called when a task attempts to obtain a network buffer, but a buffer was not available even after any defined block period.</td>
</tr>
<tr>
<td>iptraceFAILED_TO_OBTAIN_NETWORK_BUFFER_FROM_ISR()</td>
<td>Called when an interrupt service routine attempts to obtain a network buffer, but a buffer was not available.</td>
</tr>
<tr>
<td>iptraceCREATING_ARP_REQUEST( ulIPAddress )</td>
<td>Called when the IP generates an ARP request packet.</td>
</tr>
<tr>
<td>iptraceARP_TABLE_ENTRY_WILL_EXPIRE( ulIPAddress )</td>
<td>Called when an ARP request is about to be sent because the entry for the IP address ulIPAddress in the ARP cache has become stale. ulIPAddress is expressed as a 32-bit number in network byte order.</td>
</tr>
<tr>
<td>iptraceARP_TABLE_ENTRY_EXPIRED( ulIPAddress )</td>
<td>Called when the entry for the IP address ulIPAddress in the ARP cache is removed. ulIPAddress is expressed as a 32-bit number in network byte order.</td>
</tr>
<tr>
<td>iptraceARP_TABLE_ENTRY_CREATED( ulIPAddress, ucMACAddress )</td>
<td>Called when a new entry in the ARP table is created to map the IP address ulIPAddress to the MAC address ucMACAddress. ulIPAddress is expressed as a 32-bit number in network byte order. ucMACAddress is a pointer to</td>
</tr>
<tr>
<td>Macro definition</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>iptraceSENDING_UDP_PACKET( ulIPAddress )</td>
<td>Called when a UDP packet is sent to the IP address ulIPAddress. ulIPAddress is expressed as a 32-bit number in network byte order.</td>
</tr>
<tr>
<td>iptracePACKET_DROPPED_TO_GENERATE_ARP( ulIPAddress )</td>
<td>Called when a packet destined for the IP address ulIPAddress is dropped because the ARP cache does not contain an entry for the IP address. The packet is automatically replaced by an ARP packet. ulIPAddress is expressed as a 32-bit number in network byte order.</td>
</tr>
<tr>
<td>iptraceICMP_PACKET_RECEIVED()</td>
<td>Called when an ICMP packet is received.</td>
</tr>
<tr>
<td>iptraceSENDING_PING_REPLY( ulIPAddress )</td>
<td>Called when an ICMP echo reply (ping reply) is sent to the IP address ulIPAddress in response to an ICMP echo request (ping request) originating from the same address. ulIPAddress is expressed as a 32-bit number in network byte order.</td>
</tr>
<tr>
<td>traceARP_PACKET_RECEIVED()</td>
<td>Called when an ARP packet is received, even if the local network node is not involved in the ARP transaction.</td>
</tr>
<tr>
<td>iptracePROCESSING_RECEIVED_ARP_REPLY( ulIPAddress )</td>
<td>Called when the ARP cache is about to be updated in response to the reception of an ARP reply. ulIPAddress holds the ARP message's target IP address (as a 32-bit number in network byte order), which may not be the local network node</td>
</tr>
<tr>
<td>Macro definition</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>iptraceSENDING_ARP_REPLY( ulIPAddress )</td>
<td>An ARP reply is being sent in response to an ARP request from the IP address ulIPAddress. ulIPAddress is expressed as a 32-bit number in network byte order.</td>
</tr>
<tr>
<td>iptraceFAILED_TO_CREATE_SOCKET()</td>
<td>A call to FreeRTOS_socket() failed because there was insufficient FreeRTOS heap memory available for the socket structure to be created.</td>
</tr>
<tr>
<td>iptraceRECVFROM_DISCARDING_BYTES( xNumberOfBytesDiscarded )</td>
<td>FreeRTOS_recvfrom() is discarding xNumberOfBytesDiscarded bytes because the number of bytes received is greater than the number of bytes that will fit in the user supplied buffer (the buffer passed in as a FreeRTOS_recvfrom() function parameter).</td>
</tr>
<tr>
<td>iptraceETHERNET_RX_EVENT_LOST()</td>
<td>Called when a packet received by the network driver is dropped for one of the following reasons: There is insufficient space in the network event queue (see the ipconfigEVENT_QUEUE_LENGTH setting in FreeRTOSIPConfig.h), the received packet has an invalid data length, or there are no network buffers available (see the ipconfigNUM_NETWORK_BUFFER_DESCRIPTRORS setting in FreeRTOSIPConfig.h). Note this macro is called by the network driver rather than</td>
</tr>
<tr>
<td>Macro definition</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>iptraceSTACK_TX_EVENT_LOST( xEvent )</td>
<td>Called when a packet generated by the TCP/IP stack is dropped because there is insufficient space in the network event queue (see the ipconfigEVENT_QUEUE_LENGTH setting in FreeRTOSIPConfig.h).</td>
</tr>
</tbody>
</table>
| iptraceNETWORK_EVENT_RECEIVED( eEvent ) | Called when the TCP/IP stack processes an event previously posted to the network event queue. eEvent will be one of the following values:  
  - eNetworkDownEvent - The network interface has been lost and/or needs [re]connecting.  
  - eNetworkRxEvent - The network interface has queued a received Ethernet frame.  
  - eARPTimerEvent - The ARP timer expired.  
  - eStackTxEvent - The software stack has queued a packet to transmit.  
  - eDHCPEvent - Process the DHCP state machine.  
Note the events are defined by the private eIPEvent_t type which is not generally accessible. |
<p>| iptraceBIND_FAILED( xSocket, usPort ) | A call to FreeRTOS_bind() failed. usPort is the port number the socket xSocket was bound to. |</p>
<table>
<thead>
<tr>
<th>Macro definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iptraceDHCP_REQUESTS_FAILED_USING_DEFAULT_IP_ADDRESS(ulIPAddress)</td>
<td>Called when the default IP address is used because an IP address could not be obtained from a DHCP. ulIPAddress is expressed as a 32-bit number in network byte order.</td>
</tr>
<tr>
<td>iptraceSENDING_DHCP_DISCOVER()</td>
<td>Called when a DHCP discover packet is sent.</td>
</tr>
<tr>
<td>iptraceSENDING_DHCP_REQUEST()</td>
<td>Called when a DHCP request packet is sent.</td>
</tr>
<tr>
<td>iptraceNETWORK_INTERFACE_TRANSMIT()</td>
<td>Called when a packet is sent to the network by the network driver. Note this macro is called by the network driver rather than the TCP/IP stack and may not be called at all by drivers provided by third parties.</td>
</tr>
<tr>
<td>iptraceNETWORK_INTERFACE_RECEIVE()</td>
<td>Called when a packet is received from the network by the network driver. Note this macro is called by the network driver rather than the TCP/IP stack and may not be called at all by drivers provided by third parties.</td>
</tr>
<tr>
<td>iptraceSENDING_DNS_REQUEST()</td>
<td>Called when a DNS request is sent.</td>
</tr>
<tr>
<td>iptraceWAITING_FOR_TX_DMA_DESCRIPTOR()</td>
<td>Called when a transmission at the network driver level cannot complete immediately because the driver is having</td>
</tr>
<tr>
<td>Macro definition</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>to wait for a DMA descriptor to become free. Try increasing the configNUM_TXETHERNET_DMA_DESCRIPTORs setting in FreeRTOSConfig.h (if it exists for the network driver being used).</td>
</tr>
</tbody>
</table>
3. API Reference Guide

Primary Sockets Functions

- FreeRTOS_socket()
- FreeRTOS_bind()
- FreeRTOS_connect()
- FreeRTOS_listen()
- FreeRTOS_accept()
- FreeRTOS_send()
- FreeRTOS_sendto()
- FreeRTOS_recv()
- FreeRTOS_recvfrom()
- FreeRTOS_setsockopt()
- FreeRTOS_shutdown()
- FreeRTOS_closesocket()
- FreeRTOS_select()

Miscellaneous Sockets functions

Propitiatory socket functions are lower case to match Berkeley convention.

- FreeRTOS_gethostbyname()
- FreeRTOS_createsocketset()
- FreeRTOS_FD_SET()
- FreeRTOS_FD_CLR()
- FreeRTOS_ISSET()
- FreeRTOS_inet_addr()
- FreeRTOS_inet_addr_quick()
- FreeRTOS_inet_ntoa()
- FreeRTOS_htons()
- FreeRTOS_ntohs()
- FreeRTOS_htonl()
- FreeRTOS_ntohl()
- FreeRTOS_outstanding()
- FreeRTOS_recvcount()
• FreeRTOS_issocketconnected()
• FreeRTOS_getremoteaddress()
• FreeRTOS_maywrite()

IP functions

Propitiatory IP functions are mixed case to match FreeRTOS convention.

• FreeRTOS_IPInit()
• FreeRTOS_GetUDPPayloadBuffer()
• FreeRTOS_ReleaseUDPPayloadBuffer()
• FreeRTOS_SendPingRequest()
• FreeRTOS_GetAddressConfiguration()
• FreeRTOS_GetMACAddress()
• FreeRTOS_GetIPAddress()
• FreeRTOS_GetIPAddress()
• FreeRTOS_GetGatewayAddress()
• FreeRTOS_GetDNSServerAddress()
• FreeRTOS_GetNetmask()
• FreeRTOS_OutputARPRequest()
• FreeRTOS_IsNetworkUp()
• FreeRTOS_SignalSocket()

Event hook functions

• vApplicationIPNetworkEventHook()
• vApplicationPingReplyHook()

3.1 Primary Sockets Functions

3.1.1 FreeRTOS_socket()

FreeRTOS.Sockets.h

xSocket_t FreeRTOS_socket( BaseType_t xDomain, BaseType_t xtype, BaseType_t xProtocol );
Create a TCP or UDP socket.

See the OPENRTOS+TCP users guide for more information on using both TCP and UDP sockets.

**Parameters:**

**xDomain**
Must be set to FREERTOS_AF_INET.

**xType**
Set to FREERTOS_SOCK_STREAM to create a TCP socket.
Set to FREERTOS_SOCK_DGRAM to create a UDP socket.
No other values are valid.

**xProtocol**
Set to FREERTOS_IPPROTO_TCP to create a TCP socket.
Set to FREERTOS_IPPROTO_UDP to create a UDP socket.
No other values are valid.

**Returns:**

If a socket is created successfully, then the socket handle is returned. If there is insufficient OPENRTOS heap memory available for the socket to be created then FREERTOS_INVALID_SOCKET is returned.

**Example usage:**

The following code snippets show how to create a UDP and TCP socket respectively.

```c
/* OPENRTOS+TCP sockets include. */
#define "FreeRTOS_sockets.h"

void aFunction( void )
{
    /* Variable to hold the created socket. */
    xSocket_t xSocket;
    struct freertos_sockaddr xBindAddress;
```
/* Create a UDP socket. */

xSocket = FreeRTOS_socket( FREERTOS_AF_INET,
                        FREERTOS.SOCK_DGRAM,
                        FREERTOS_IPPROTO_UDP );

/* Check the socket was created successfully. */
if( xSocket != FREERTOS_INVALID_SOCKET )
{
    /* The socket was created successfully and can now be used to send data
        using the FreeRTOS_sendto() API function. Sending to a socket that has
        not first been bound will result in the socket being automatically bound
        to a port number. Use FreeRTOS_bind() to bind the socket to a
        specific port number. This example binds the socket to port 9999. The
        port number is specified in network byte order, so FreeRTOS_htons() is
        used. */
    xBindAddress.sin_port = FreeRTOS_htons( 9999 );
    if( FreeRTOS_bind( xSocket, &xBindAddress, sizeof( &xBindAddress ) ) == 0 )
    {
        /* The bind was successful. */
    }
    else
    {
        /* There was insufficient FreeRTOS heap memory available for the socket
            to be created. */
    }
}

Figure 3-1 Example use of the FreeRTOS_socket() API function to create a UDP socket

/* OPENRTOS+TCP sockets include. */

#define "FreeRTOS.Sockets.h"
void aFunction( void )
{
    /* Variable to hold the created socket. */
    xSocket_t xSocket;
    struct freertos_sockaddr xBindAddress;

    /* Create a TCP socket. */
    xSocket = FreeRTOS_socket( FREERTOS_AF_INET,
                              FREERTOS.SOCK_STREAM,
                              FREERTOS IPPROTO_TCP );

    /* Check the socket was created successfully. */
    if( xSocket != FREERTOS_INVALID_SOCKET )
    {
        /* The socket was created successfully and can now be used to connect to
        a remote socket using FreeRTOS_connect(), before sending data using
        FreeRTOS_send(). Alternatively the socket can be bound to a port using
        FreeRTOS_bind(), before listening for incoming connections using
        FreeRTOS_listen(). */
    }
    else
    {
        /* There was insufficient OPENRTOS heap memory available for the socket
        to be created. */
    }
}

Figure 3-2 Example use of the FreeRTOS_socket() API function to create a TCP socket

3.1.2 FreeRTOS_bind()

FreeRTOSsockets.h
BaseType_t FreeRTOS_bind( xSocket_t xSocket,
                          struct freertos_sockaddr *pxAddress,
Binds a socket to a local port number. Binding a socket associates a socket with a port number on the local IP address, resulting in the socket receiving all the data that is sent to that IP address and port number combination.

The Network Addressing and Binding sections of the Embedded Networking Basics and Glossary section provide an introduction to the topic of socket binding.

Specifying a port number of 0 will result in the socket being bound to a port number from the private range, between 49408 and 65280.

Port numbers above 49408 (0xC100) to 65280 (0xff00) are considered private numbers available to the IP stack for dynamic allocation, and should therefore be avoided. Specifying a port number of 0 or passing pxAddress as NULL will result in the socket being bound to a port number from the private range.

For convenience, if ipconfigALLOW_SOCKET_SEND_WITHOUT_BIND is set to 1 in FreeRTOSIPConfig.h, then calling FreeRTOS_send() on a socket that has not first been bound to a port number will also result in the socket being bound to a port number from the private range.

OPENRTOS+TCP does not [currently] use all the function parameters. The parameters that are not used are retained in the function's prototype to ensure consistency with the expected standard Berkeley sockets API, and to ensure compatibility with future versions of OPENRTOS+TCP.

Parameters:

**xSocket**

The handle of the socket that is being bound to an address. The socket must have previously been created by a successful call to FreeRTOS_socket().

**pxAddress**

A pointer to a freertos_sockaddr structure that contains the details of the port number being bound to. See the provided example.

**xAddressLength**

Not currently used, but should be set to sizeof( struct freertos_sockaddr ) to ensure future compatibility.

Returns:
If the bind was successful then 0 is returned (0 is the standard Berkeley sockets success return value, contrary to the OPENRTOS standard where 0 means fail!).

-FREERTOS EINVAL is returned if the socket did not get bound, probably because the specified port number was already in use.

-FREERTOS ECANCELED is returned if the calling RTOS task did not get a response from the IP RTOS task to the bind request.

Example usage:

See the examples on the OPENRTOS+TCP users guide section, and on the FreeRTOS_socket() API documentation section.

3.1.3 FreeRTOS_connect()

FreeRTOS_sockets.h

BaseType_t FreeRTOS_connect( xSocket_t xClientSocket,
                               struct freertos_sockaddr *pxAddress,
                               socklen_t xAddressLength );

Connect a TCP socket to a remote socket.

The socket must first have been successfully created by a call to FreeRTOS_socket(), and optionally bound to a port using a call to FreeRTOS_bind().

If FreeRTOS_connect() is called on a socket that is not bound to a port number, and the value of ipconfigALLOW_SOCKET_SEND WITHOUT BIND is set to 1 in FreeRTOSIPConfig.h, then the TCP/IP stack will automatically bind the socket to a port number from the private address range.

FreeRTOS_connect() has an optional timeout. The timeout defaults to ipconfigSOCK_DEFAULT_RECEIVE BLOCK TIME, and is modified using the FREERTOS SO RCVTIMEO parameter in a call to FreeRTOS_setsockopt(). If the connect operation does not succeed immediately then the calling RTOS task will be held in the Blocked state (so that other tasks can execute) until either the connect request is successful, or the timeout expires.

Parameters:
**xSocket**

The handle of the socket being bound. The socket must have already been created (see FreeRTOS_socket()).

**pxAddress**

A pointer to a freertos_sockaddr structure that contains the destination IP address and port number (the remote socket the local socket is attempting to connect to).

**xAddressLength**

Not currently used, but should be set to sizeof( struct freertos_sockaddr ) to ensure future compatibility.

**Returns:**

If the connect operation succeeded then 0 is returned.

If xSocket is not a valid TCP socket then -pdFREERTOS_ERRNO_EBADF is returned.

If xSocket was already connected before FreeRTOS_connect() was called then -pdFREERTOS_ERRNO_EISCONN is returned.

If xSocket is not in a state that allows a connect operation then either -pdFREERTOS_ERRNO_EINPROGRESS or -pdFREERTOS_ERRNO_EAGAIN is returned.

If the socket has a read block time of zero and the connect operation cannot succeed immediately then -pdFREERTOS_ERRNO_EWOULDBLOCK is returned.

If the connect attempt times out then -pdFREERTOS_ERRNO_ETIMEDOUT is returned.

Note that, because OPENRTOS does not implement errno, the behaviour in the presence of an error is necessarily different to that of connect() functions that are fully compliant with the expected Berkeley sockets behaviour.

**Example usage:**

See the Sending TCP Data section of the OPENRTOS+TCP users guide sections.

### 3.1.4 FreeRTOS_listen()

FreeRTOS_sockets.h
BaseType_t FreeRTOS_listen( xSocket_t xSocket, BaseType_t xBacklog );

Places a TCP socket into a state where it is listening for and can accept incoming connection requests from remote sockets.

The socket must first have been successfully created by a call to FreeRTOS_socket(), and bound to a port using a call to FreeRTOS_bind().

By default a new socket (a child socket) will be created to handle any accepted connections. The new socket will be returned by FreeRTOS_accept(), and can be used immediately. The child socket inherits all the properties from the parent socket.

Optionally the FREERTOS_SO_REUSE_LISTEN_SOCKET parameter can be used with a call to FreeRTOS_setsockopt() to configure the parent socket to handle any accepted connections itself - without creating a child socket for this purpose. This is a useful way to save resources when the socket will only handle a single connection at a time. For example, if the socket is used to implement a telnet server that only permits one simultaneous connection.

Parameters:

xSocket The handle of the socket being placed into the Listening state. The socket must have already been created using a call to FreeRTOS_socket() and bound to a port number using a call to FreeRTOS_bind()

xBacklog In the default case where a new socket is created for each new connection the backlog value puts a limit on the number of simultaneously connected clients.

Returns:

If the the socket was successfully placed into the listening state then 0 is returned.

If xSocket is not a valid TCP socket then then -pdFREERTOS_ERRNO_EOPNOTSUPP is returned.

If xSocket is not in bound but closed state then -pdFREERTOS_ERRNO_EOPNOTSUPP is returned.
Note that, because OPENRTOS does not implement errno, the behaviour in the presence of an error is necessarily different to that of connect() functions that are fully compliant with the expected Berkeley sockets behaviour.

**Example usage:**

See the "Creating, configuring and binding a TCP server socket" source code example in the "Creating Configuring and Binding TCP Client and Server Sockets" section of the OPENRTOS+TCP users guide sections.

### 3.1.5 FreeRTOS_accept()

FreeRTOS accepts.h

```c
xSocket_t FreeRTOS_accept( xSocket_t xServerSocket,
                        struct freertos_sockaddr *pxAddress,
                        socklen_t *pxAddressLength );
```

Accept a connection on a TCP socket.

The socket must first have been successfully created by a call to FreeRTOS_socket(), bound to a port using a call to FreeRTOS_bind(), and placed into the Listening state using a call to FreeRTOS_listen().

By default a new socket (a child socket) will be created to handle any accepted connections. The new socket will be returned by FreeRTOS_accept(), and can be used immediately. The child socket inherits all the properties from the parent socket.

Optionally the FREERTOS_SO_REUSE_LISTEN_SOCKET parameter can be used with a call to FreeRTOS_setsockopt() to configure the parent socket to handle any accepted connections itself - without creating a child socket for this purpose. This is a useful way to save resources when the socket will only handle a single connection at a time. For example, if the socket is used to implement a telnet server that only permits one simultaneous connection.

FreeRTOS_accept() has an optional timeout. The timeout defaults to ipconfigSOCK_DEFAULT_RECEIVE_BLOCK_TIME, and is modified using the FREERTOS_SO_RCVTIMEO parameter in a call to FreeRTOS_setsockopt(). If the accept operation does not succeed immediately then the calling RTOS task will be held in the Blocked state (so that
other RTOS tasks can execute) until either a connection is accepted, or the timeout expires. **Parameters:**

**xServerSocket**  
The handle of the listening socket on which new connections are to be accepted.

**pxAddress**  
A pointer to a freertos_sockaddr structure that will be filled (by FreeRTOS_accept()) with the IP address and port number of the socket from which a connection was accepted.

**pxAddressLength**  
Not currently used, but should be set to sizeof( struct freertos_sockaddr ) to ensure future compatibility.

**Returns:**

If a connection from a remote socket is accepted and a new local socket is created to handle the accepted connection then a handle to the new socket is returned.

If xServerSocket is not a valid TCP socket then then FREERTOS_INVALID_SOCKET is returned.

If xServerSocket is not in the Listening state (see FreeRTOS_listen()) then FREERTOS_INVALID_SOCKET is returned.

If a timeout occurs before a connection from a remote socket is accepted then NULL is returned.

Note that, because OPENRTOS does not implement errno, the behaviour in the presence of an error is necessarily different to that of connect() functions that are fully compliant with the expected Berkeley sockets behaviour.

**Example usage:**

See the "Creating, configuring and binding a TCP server socket" source code example in the "Creating Configuring and Binding TCP Client and Server Sockets" section of the OPENRTOS+TCP users guide.
3.1.6 FreeRTOS_send()

FreeRTOS_sockets.h
BaseType_t FreeRTOS_send( xSocket_t xSocket,
   const void *pvBuffer,
   size_t xDataLength,
   BaseType_t xFlags );

Send data to a TCP socket (see FreeRTOS_sendto() for the UDP equivalent).

The socket must have already been created using a call to FreeRTOS_socket(), bound to a port number, and connected to a remote socket.

The socket can be explicitly bound to a port number by calling FreeRTOS_bind().

The socket can actively connect to a remote socket using FreeRTOS_connect(). If FreeRTOS_connect() is called on a socket that is not bound to a port number, and the value of ipconfigALLOW_SOCKET_SEND_WITHOUT_BIND is set to 1 in FreeRTOSIPConfig.h, then the TCP/IP stack will automatically bind the socket to a port number from the private address range.

Alternatively the socket can wait for incoming connections using FreeRTOS_accept().

FreeRTOS_send() has an optional timeout. The timeout defaults to ipconfigSOCK_DEFAULT_SEND_BLOCK_TIME, and is modified using the FREERTOS_SO_SNDTIMEO parameter in a call to FreeRTOS_setsockopt(). If the send operation cannot queue the bytes for transmission immediately then the calling RTOS task will be held in the Blocked state (so that other tasks can execute) until either the bytes can be queued for sending, or the timeout expires.

OPENRTOS+TCP does not [currently] use all the function parameters. The parameters that are not used are retained in the function's prototype to ensure consistency with the expected standard Berkeley sockets API, and to ensure compatibility with future versions of OPENRTOS+TCP.

Parameters:

\textit{xSocket} \hspace{1cm} The handle of the socket to which data is being sent. The socket must have already been created and bound to a port number
pvBuffer Points to the source of the data being transmitted.

xTotalDataLength The number of bytes to send.

ulFlags Not currently used. Future OPENRTOS+TCP versions may implement send options using the ulFlags parameter.

Returns:

If the send was successful then the number of bytes queued for sending is returned (note this may be fewer bytes than the number requested by the xTotalDataLength parameter).

If no data could be sent because the socket was closed or got closed then -pdFREERTOS_ERRNO_ENOTCONN is returned.

If no data could be sent because there was insufficient memory then -pdFREERTOS_ERRNO_ENOMEM is returned.

If no data could be sent because xSocket was not a valid TCP socket then -pdFREERTOS_ERRNO_EINVAL is returned.

If a timeout occurred before any data could be sent then -pdFREERTOS_ERRNO_ENOSPC is returned.

Note that, because OPENRTOS does not implement errno, the behaviour in the presence of an error is necessarily different to that of send() functions that are fully compliant with the expected Berkeley sockets behaviour.

Example usage:

See the "Creating, Configuring and Binding TCP Client and Server Sockets" section of the OPENRTOS+TCP users guide sections for examples of how to prepare a TCP socket for sending data.

See the "Sending TCP Data" section of the OPENRTOS+TCP users guide sections for examples of sending data to a TCP socket.
3.1.7 FreeRTOS_sendto()

FreeRTOS_sockets.h

int32_t FreeRTOS_sendto( xSocket_t xSocket,
                       const void *pvBuffer,
                       size_t xTotalDataLength,
                       uint32_t ulFlags,
                       const struct freertos_sockaddr *pxDestinationAddress,
                       socklen_t xDestinationAddressLength );

Send data to a UDP socket (see FreeRTOS_send() for the TCP equivalent). The socket must have already been created by a successful call to FreeRTOS_socket().

This function can be used with standard calling semantics, or zero copy calling semantics:

- Standard sendto() semantics

  Data is copied from the address pointed to by the pvBuffer parameter into a network buffer allocated internally by the TCP/IP stack.

  The standard sendto() semantics are used when the ulFlags parameter does not have the FREERTOS_ZERO_COPY bit set. See the example at the bottom of this section, and other application examples provided on this website.

- Zero copy sendto() semantics

  The application writer:

  1. Obtains a buffer from the TCP/IP stack.
  2. Writes the data to be sent into the buffer obtained from the TCP/IP stack.
  3. Uses a pointer to the (already complete) buffer as the pvBuffer parameter.

  The TCP/IP stack then passes a reference to the same buffer through the TCP/IP stack to the Ethernet driver, where it is transmitted (normally by DMA where the hardware permits).

  The zero copy sendto() semantics are used when the ulFlags parameter has the FREERTOS_ZERO_COPY bit set. See the examples at the bottom of this section, and other application examples provided on this website.
FreeRTOS_sendto() has an optional timeout. The timeout defaults to `ipconfigSOCK_DEFAULT_SEND_BLOCK_TIME`, and is modified using `FreeRTOS_setsockopt()`. If the send operation cannot queue the bytes for transmission immediately then the calling RTOS task will be held in the Blocked state (so that other tasks can execute) until either the bytes can be queued for sending, or the timeout expires. A timeout will occur if:

- The standard sendto() semantics are used, and the TCP/IP stack was not able to obtain a network buffer in time. Or,
- No space became available on the queue used to send messages to the IP RTOS task (see the `ipconfigEVENT_QUEUE_LENGTH` setting in the `FreeRTOSIPConfig.h` header file).

If `FreeRTOS_sendto()` is called on a socket that is not bound to a port number, and the value of `ipconfigALLOW_SOCKET_SEND_WITHOUT_BIND` is set to 1 in `FreeRTOSIPConfig.h`, then the TCP/IP stack will automatically bind the socket to a port number from the private address range.

`OPENRTOS+TCP` does not [currently] use all the function parameters. The parameters that are not used are retained in the function's prototype to ensure consistency with the expected standard Berkeley sockets API, and to ensure compatibility with future versions of `OPENRTOS+TCP`.

**Parameters:**

- **xSocket**
  The handle of the socket to which data is being sent. The socket must have already been created (see `FreeRTOS_socket()`).

- **pvBuffer**
  If the standard calling semantics are being used (the `ulFlags` parameter does not have the `FREERTOS_ZERO_COPY` bit set) then `pvBuffer` points to the source of the data being transmitted. `FreeRTOS_sendto()` will copy data from `pvBuffer` into a network buffer inside the TCP/IP stack.

  If the zero copy calling semantics are being sued (the `ulFlags` parameter does have the `FREERTOS_ZERO_COPY` bit set) then `pvBuffer` points to a buffer that was previously obtained from the TCP/IP stack and already contains the data being sent. the TCP/IP stack will take control of the buffer rather than copy data out of the buffer.
See the example usage section below, and the application examples provided on this website.

**xTotalDataLength**

The number of bytes to send.

**ulFlags**

A bitwise set of options that affect the send operation.

If ulFlags has the FREERTOS_ZERO_COPY bit set, then the function will use the zero copy semantics, otherwise the function will use the standard copy mode semantics. See the description of the pvBuffer parameter above.

Future OPENRTOS+TCP versions may implement other bits.

**pxDestinationAddress**

A pointer to a freertos_sockaddr structure that contains the destination IP address and port number (the socket the data is being sent to). See the example below.

**xDestinationAddressLength**

Not currently used, but should be set to sizeof(struct freertos_sockaddr) to ensure future compatibility.

**Returns:**

The number of bytes that were actually queue for sending, which will be 0 if an error or timeout occurred.

Note that, because OPENRTOS does not implement errno, the behaviour in the presence of an error is necessarily different to that of sendto() functions that are fully compliant with the expected Berkeley sockets behaviour.

**Example usage:**

The first example sends to a socket using the standard calling semantics (see below for another example that uses the zero copy calling semantics). The socket is passed in as the function parameter, and is assumed to have already been created using a call to FreeRTOS_socket(). If ipconfigALLOW_SOCKET_SEND_WITHOUT_BIND is not set to 1 in FreeRTOSIPConfig.h, then the socket is also assumed to have been bound to a port number using FreeRTOS_bind().
/ * OPENRTOS+TCP sockets include. */
#define "FreeRTOS_sockets.h"

void vStandardSendExample( xSocket_t xSocket )
{
  /* Note - the RTOS task stack must be big enough to hold this array!. */
  uint8_t ucBuffer[ 128 ];
  struct freertos_sockaddr xDestinationAddress;
  int32_t iReturned;

  /* Fill in the destination address and port number, which in this case is 
  port 1024 on IP address 192.168.0.100. */
  xDestinationAddress.sin_addr = FreeRTOS_inet_addr_quick( 192, 168, 0, 100 );
  xDestinationAddress.sin_port = FreeRTOS_htons( 1024 );

  /* The local buffer is filled with the data to be sent, in this case it is 
  just filled with 0xff. */
  memset( ucBuffer, 0xff, 128 );

  /* Send the buffer with ulFlags set to 0, so the FREERTOS_ZERO_COPY bit 
  is clear. */
  iReturned = FreeRTOS_sendto( 
      /* The socket being send to. */
      xSocket,
      /* The data being sent. */
      ucBuffer,
      /* The length of the data being sent. */
      128,
      /* ulFlags with the FREERTOS_ZERO_COPY bit clear. */
      0,
      /* Where the data is being sent. */
  );
&xDestinationAddress,
/* Not used but should be set as shown. */
sizeof( xDestinationAddress )
);

if( iReturned == 128 )
{
    /* The data was successfully queued for sending. 128 bytes will have
been copied out of ucBuffer and into a buffer inside the TCP/IP stack.
ucBuffer can be re-used now. */
}
}

Figure 3-3 Example using FreeRTOS_sendto() with the standard (as opposed to zero copy) calling semantics

This second example sends to a socket using the zero copy calling semantics (see above for an example that uses the standard calling semantics). The socket is passed in as the function parameter, and is assumed to have already been created using a call to FreeRTOS_socket(). If ipconfigALLOW_SOCKET_SEND_WITHOUT_BIND is not set to 1 in FreeRTOSIPConfig.h, then the socket is also assumed to have been bound to a port number using FreeRTOS_bind().

/* OPENRTOS+TCP sockets include. */
#define "FreeRTOS_socket.h"

void vZeroCopySendExample( xSocket_t xSocket )
{
    struct freertos_sockaddr xDestinationAddress;
    uint8_t *pucUDPPayloadBuffer;
    int32_t iReturned;

    /* Fill in the destination address and port number, which in this case is
port 1024 on IP address 192.168.0.100. */
xDestinationAddress.sin_addr = FreeRTOS_inet_addr_quick( 192, 168, 0, 100 );
xDestinationAddress.sin_port = FreeRTOS_htons( 1024 );

/* Obtain a buffer from the TCP/IP stack that is large enough to hold the data being sent. Although the maximum amount of time to wait for a buffer is passed into FreeRTOS_GetUDPPayloadBuffer() as portMAX_DELAY, the actual maximum time will be capped to ipconfigMAX_SEND_BLOCK_TIME_TICKS (defined in FreeRTOSIPConfig.h) */
pucUDPPayloadBuffer = ( uint8_t * ) FreeRTOS_GetUDPPayloadBuffer( 128, portMAX_DELAY );

if( pucUDPPayloadBuffer != NULL )
{
    /* Write the data being sent directly into the buffer obtained from the IP stack. In this case the data is just set to 0xff. */
    memset( pucUDPPayloadBuffer, 0xff, 128 );

    /* Pass the buffer (by reference) into the TCP/IP stack using the zero copy semantics. Ensure to read the remaining source code comments for information on managing the pucUDPPayloadBuffer pointer after this call to FreeRTOS_sendto(). */
    iReturned = FreeRTOS_sendto(
        /* The socket being sent to. */
        xSocket,
        /* The buffer that already contains the data being sent. */
        &xBufferDescriptor,
        /* The length of the data being send. */
        128,
        /* ulFlags with the FREERTOS_ZERO_COPY bit set. */
        FREERTOS_ZERO_COPY,
        /* Where the data is being sent. */
        &xDestinationAddress,
/* Not used but should be set as shown. */

sizeof( xDestinationAddress )

if( iReturned != 0 )
{
    /* The buffer pointed to by pucUDPPayloadBuffer was successfully
        passed (by reference) into the TCP/IP stack and is now queued for sending.
        the TCP/IP stack is responsible for returning the buffer after it has been
        sent, and pucUDPPayloadBuffer can be used safely in another call to
        FreeRTOS_GetUDPPayloadBuffer(). */
}
else
{
    /* The buffer pointed to by pucUDPPayloadBuffer was not successfully
        passed (by reference) to the TCP/IP stack. To prevent memory and network
        buffer leaks the buffer must be either reused or, as in this case,
        released back to the TCP/IP stack. */
    FreeRTOS_ReleaseUDPPayloadBuffer( ( void * ) pucUDPPayloadBuffer );
}
}

Figure 3-4 Example using FreeRTOS_sendto() with the zero copy calling semantics

3.1.8 FreeRTOS_recv()

FreeRTOS_recv()
FreeRTOS_recv( xSocket_t xSocket,
        void *pvBuffer,
        size_t xBufferLength,
        BaseType_t xFlags );

Receive data from a TCP socket (see FreeRTOS_recvfrom() for the UDP equivalent).
The socket must have already been created using a call to FreeRTOS_socket(), bound to a port number, and connected to a remote socket.

The socket can be explicitly bound to a port number by calling FreeRTOS_bind().

The socket can actively connect to a remote socket using FreeRTOS_connect(). If FreeRTOS_connect() is called on a socket that is not bound to a port number, and the value of ipconfigALLOW_SOCKET_SEND_WITHOUT_BIND is set to 1 in FreeRTOSIPConfig.h, then the TCP/IP stack will automatically bind the socket to a port number from the private address range.

Alternatively the socket can wait for incoming connections using FreeRTOS_accept().

FreeRTOS_recv() has an optional timeout. The timeout defaults to ipconfigSOCK_DEFAULT_RECEIVE_BLOCK_TIME, and is modified using the FREERTOS_SO_RCVTIMEO parameter in a call to FreeRTOS_setsockopt(). If the receive operation cannot return received bytes immediately then the calling RTOS task will be held in the Blocked state (so that other tasks can execute) until either bytes are received, or the timeout expires.

OPENRTOS+TCP does not [currently] use all the function parameters. The parameters that are not used are retained in the function's prototype to ensure consistency with the expected standard Berkeley sockets API, and to ensure compatibility with future versions of OPENRTOS+TCP.

**Parameters:**

- **xSocket**
  The handle of the socket from which data is being read.

- **pvBuffer**
  The buffer into which received data will be placed.

- **xBufferLength**
  The size of the buffer (in bytes) pointed to by the pvBuffer parameter - and therefore also the maximum number of bytes that will be read.

- **ulFlags**
  Not currently used. Future OPENRTOS+TCP versions may implement receive options using the ulFlags parameter.

**Returns:**
If the receive was successful then the number of bytes received (placed in the buffer pointed to by pvBuffer) is returned.

If a time out occurred before data could be received then 0 is returned.

If there was not enough memory for the socket to be able to create either an Rx or Tx stream then -pdFREERTOS_ERRNO_ENOMEM is returned.

If the socket was closed or got closed then -pdFREERTOS_ERRNO_ENOTCONN is returned.

If the socket is not valid, is not a TCP socket, or is not bound then -pdFREERTOS_ERRNO_EINVAL is returned;

Note that, because OPENRTOS does not implement errno, the behaviour in the presence of an error is necessarily different to that of recv() functions that are fully compliant with the expected Berkeley sockets behaviour.

**Example usage:**

See the "Creating, Configuring and Binding TCP Client and Server Sockets" section of the OPENRTOS+TCP users guide sections for examples of how to prepare a TCP socket for receiving data.

See the "Receiving TCP Data" section of the OPENRTOS+TCP users guide sections for examples of receiving data from a TCP socket.

### 3.1.9 FreeRTOS_recvfrom()

FreeRTOS_sockets.h

```c
int32_t FreeRTOS_recvfrom( xSocket_t xSocket,
                          void *pvBuffer,
                          size_t xBufferLength,
                          uint32_t ulFlags,
                          struct freertos_sockaddr *pxSourceAddress,
                          socklen_t *pxSourceAddressLength );
```

Receive data from a UDP socket (see FreeRTOS_recv() for the TCP equivalent). The socket must have already been created by a successful call to FreeRTOS_socket().
This function can be used with standard calling semantics, or zero copy calling semantics:

- **Standard recvfrom() semantics**

  Data is copied from a network buffer inside the TCP/IP stack into the buffer pointed to by the pvBuffer parameter.

  The standard recvfrom() semantics are used when the ulFlags parameter does not have the FREERTOS_ZERO_COPY bit set. See the example at the bottom of this section, and other application examples provided on this website.

- **Zero copy recvfrom() semantics**

  The application writer receives from the TCP/IP stack a reference to the buffer that already contains the received data. No data is copied.

  The zero copy recvfrom() semantics are used when the ulFlags parameter has the FREERTOS_ZERO_COPY bit set. See the examples at the bottom of this section, and other application examples provided on this website.

FreeRTOS_recvfrom() has an optional timeout. The timeout defaults to portMAX_DELAY and is modified using FreeRTOS_setsockopt(). If the receive operation cannot complete immediately because there is no data queued on the socket to receive then the calling RTOS task will be held in the Blocked state (so that other tasks can execute) until either data has been received, or the timeout expires.

OPENRTOS+TCP does not [currently] use all the function parameters. The parameters that are not used are retained in the function's prototype to ensure consistency with the expected standard Berkeley sockets API, and to ensure compatibility with future versions of OPENRTOS+TCP.

**Parameters:**

- **xSocket**

  The handle of the socket from which data is being read. The socket must have already been created (see FreeRTOS_socket()).

- **pvBuffer**

  If the standard calling semantics are used (the ulFlags parameter does not have the FREERTOS_ZERO_COPY bit set) then pvBuffer points to the buffer into which received data will be copied.
If the zero copy calling semantics are used (the ulFlags parameter does have the FREERTOS_ZERO_COPY bit set) then *pvBuffer will be set (by FreeRTOS_recvfrom()) to point to the buffer that already holds the received data. pvBuffer is used to pass a reference to the received data out of FreeRTOS_recvfrom() without any data being copied.

The example at the bottom of this section, and other application examples provided on this website, demonstrate FreeRTOS_recvfrom() being used with both the standard and zero copy calling semantics.

**xBufferLength**

If the standard calling semantics are used (the ulFlags parameter does not have the FREERTOS_ZERO_COPY bit set) then xBufferLength must be set to the size in bytes of the buffer pointed to by the pvBuffer parameter.

If the zero copy calling semantics are used (the ulFlag parameter does not have the FREERTOS_ZERO_COPY bit set) then pvBuffer does not point to a buffer and xBufferLength is not used.

**ulFlags**

A bitwise set of options that affect the receive operation.

If ulFlags has the FREERTOS_ZERO_COPY bit set, then the function will use the zero copy semantics, otherwise the function will use the traditional copy mode semantics. See the description of the pvBuffer parameter above.

Future OPENRTOS+TCP versions may implement other bits.

**pxSourceAddress**

A pointer to a freertos_sockaddr structure that will be set (by FreeRTOS_recvrom()) to contain the IP address and port number of the socket that sent the data just received. See the example below.

**pxSourceAddressLength**

Not currently used, but should be set to sizeof( struct freertos_sockaddr ) to ensure future compatibility.
Returns:

If no bytes are received before the configured block time expires then FREERTOS_EWOULDBLOCK is returned.

If the socket is not bound to a port number then FREERTOS_EINVAL is returned.

If data is successfully received then the number of bytes received is returned.

Example usage:

The first example receives from a socket using the standard calling semantics (see below for another example that uses the zero copy calling semantics). The socket is passed in as the function parameter, and is assumed to have already been created using a call to FreeRTOS_socket(), and bound to an address using a call to FreeRTOS_bind().

```c
/* OPENRTOS+TCP sockets include. */
#define "FreeRTOS_sockets.h"

void vStandardReceiveExample( xSocket_t xSocket )
{
    /* Note - the RTOS task stack must be big enough to hold this array!. */
    uint8_t ucBuffer[ 128 ];
    int8_t cIPAddressString[ 16 ];
    struct freertos_sockaddr xSourceAddress;
    int32_t iReturned;

    /* Receive into the buffer with ulFlags set to 0, so the FREERTOS_ZERO_COPY bit is clear. */
    iReturned = FreeRTOS_recvfrom(
        /* The socket data is being received on. */
        xSocket,
        /* The buffer into which received data will be copied. */
        ucBuffer,
```
/* The length of the buffer into which data will be copied. */
128,
/* ulFlags with the FREERTOS_ZERO_COPY bit clear. */
0,
/* Will get set to the source of the received data. */
&_xSourceAddress,
/* Not used but should be set as shown. */
sizeof( xSourceAddress )
);

if( iReturned > 0 )
{
    /* Data was received from the socket. Prepare the IP address for
    printing to the console by converting it to a string. */
    FreeRTOS_inet_ntoa( xSourceAddress.sin_addr, ( char * ) cIPAddressString );

    /* Print out details of the data source. */
    printf( "Received %d bytes from IP address %s port number %d\r\n",
        iReturned, /* The number of bytes received. */
        cIPAddressString, /* The IP address that sent the data. */
        FreeRTOS_ntohs( xSourceAddress.sin_port ) ); /* The source port. */
}

Figure 3-5 Example using FreeRTOS_recvfrom() with the standard (as opposed to zero copy) calling semantics

This second example received from a socket using the zero copy calling semantics (see above for an example that uses the standard calling semantics). The socket is passed in as the function parameter, and is assumed to have already been created using a call to FreeRTOS_socket(), and bound to a port number using FreeRTOS_bind().
/* OPENRTOS+TCP sockets include. */
#define "FreeRTOS_sockets.h"

void vZeroCopyReceiveExample(xSocket_t xSocket)
{
  struct freertos_sockaddr xSourceAddress;
  uint8_t *pucReceivedUDPPayload;
  int32_t iReturned;

  /* Receive using the zero copy semantics. The address of the
   * pucReceivedUDPPayload pointer is passed in the pvBuffer parameter. */
  iReturned = FreeRTOS_recvfrom(
    /* The socket being received from. */
    xSocket,
    /* pucReceivedUDPPayload will get
    * set to points to the received data. */
    &pucReceivedUDPPayload,
    /* Ignored because the pvBuffer parameter
    * does not point to a buffer. */
    0,
    /* ulFlags with the FREERTOS_ZERO_COPY bit set. */
    FREERTOS_ZERO_COPY,
    /* Will get set to the source of the received
    * data. */
    &xSourceAddress,
    /* Not used but should be set as shown. */
    sizeof(xSourceAddress)
  );

  if( iReturned > 0 )
  {
    /* Data was received from the socket. Convert the IP address to a
    * string. */
Figure 3-6 Example using FreeRTOS_recvfrom() with the zero copy calling semantics
3.1.10 FreeRTOS_setsockopt()

FreeRTOS_setsockopt.h

BaseType_t FreeRTOS_setsockopt( xSocket_t xSocket, int32_t lLevel,
                                  int32_t lOptionName, const void
                                  *pvOptionValue,
                                  size_t xOptionLength );

Sets a socket option.

**Parameters:**

- **xSocket** The target socket (the socket being modified). The socket must have already been created by a successful call to FreeRTOS_socket().

- **lLevel** OPENRTOS+TCP does not [currently] use the lLevel parameter. The parameter is included to ensure consistency with the expected standard Berkeley sockets API, and to ensure compatibility with future versions of OPENRTOS+TCP.

- **lOptionName** The option being set or modified. See the table below for valid values.

- **pvOptionValue** The meaning of pvOptionValue is dependent on the value of lOptionName. See the description of the lOptionName parameter.

- **xOptionLength** OPENRTOS+TCP does not [currently] use the xOptionLength parameter. The parameter is included to ensure consistency with the expected standard Berkeley sockets API, and to ensure compatibility with future versions of OPENRTOS+TCP.

**Returns:**

-FREERTOS_EINVAL is returned if an invalid lOptionName value is used, otherwise 0 is returned. (0 is the standard Berkeley sockets success return value, contrary to the OPENRTOS standard where 0 means fail!)
Example usage:

This example creates a UDP socket, configures the socket's behaviour in accordance with the function's parameters, then returns the created and configured socket.

```c
/* OPENRTOS+TCP sockets include. */
#define "FreeRTOS_sockets.h"

xSocket_t xCreateASocket( TickType_t xReceiveTimeout_ms, 
                          TickType_t xSendTimeout_ms, 
                          int32_t iUseChecksum )
{
    /* Variable to hold the created socket. */
    xSocket_t xSocket;

    /* Create the socket. */
    xSocket = FreeRTOS_socket( FREERTOS_AF_INET, 
                              FREERTOS_SOCK_DGRAM, 
                              FREERTOS_IPPROTO_UDP );

    /* Check the socket was created successfully. */
    if( xSocket != FREERTOS_INVALID_SOCKET )
    {
        /* Convert the receive timeout into ticks. */
        xReceiveTimeout_ms /= portTICK_PERIOD_MS;

        /* Set the receive timeout. */
        FreeRTOS_setsockopt( xSocket, 
                             FREERTOS_SO_RCVTIMEO, /* Setting receive timeout. */
                             &xReceiveTimeout_ms, /* The timeout value. */
                             0 ); /* Not used. */
    }
```
/* Convert the send timeout into ticks. */
xSendTimeout_ms /= portTICK_PERIOD_MS;

/* Set the send timeout. */
FreeRTOS_setsockopt( xSocket, /* The socket being modified. */
0, /* Not used. */
&FREERTOS_SO_SNDTIMEO, /* Setting send timeout. */
&xSendTimeout_ms, /* The timeout value. */
0 ); /* Not used. */

if( iUseChecksum == pdFALSE )
{
   /* Turn the UDP checksum creation off for outgoing UDP packets. */
   FreeRTOS_setsockopt( xSocket, /* The socket being modified. */
   0, /* Not used. */
   &FREERTOS_SO_UDPCKSUM_OUT, /* Setting checksum on/off. */
   NULL, /* NULL means off. */
   0 ); /* Not used. */
}
else
{
   /* The checksum is used by default, so there is nothing to do here.
   If the checksum was off it could be turned on again using an option
   value other than NULL, for example { ( void * ) 1 }. */
}

return xSocket;

Figure 3-7 Example use of the FreeRTOS_setsockopt() API function
### 3.1.11 Valid `IOptionName` Values

<table>
<thead>
<tr>
<th>IOptionName Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREERTOS_SO_RCVTIMEO</td>
<td>Sets the receive timeout when <code>FreeRTOS_recvfrom()</code> is called with a UDP socket, or <code>FreeRTOS_recv()</code> is called with a TCP socket. If <code>IOptionName</code> is <code>FREERTOS_SO_RECTIMEO</code> then <code>pvOptionValue</code> must point to a variable of type <code>TickType_t</code>. Timeout values are specified in ticks. To convert a time in milliseconds to a time in ticks divide the time in milliseconds by <code>portTICK_PERIOD_MS</code> or use the <code>pdMS_TO_TICKS()</code> macro.</td>
</tr>
<tr>
<td>FREERTOS_SO_SNDTIMEO</td>
<td>Sets the transmit timeout when <code>FreeRTOS_send()</code> is used with a TCP socket, or <code>FreeRTOS_sendto()</code> is used with a UDP socket. If <code>IOptionName</code> is <code>FREERTOS_SO_SNDTIMEO</code> then <code>pvOptionValue</code> must point to a variable of type <code>TickType_t</code>. Timeout values are specified in ticks. To convert a time in milliseconds to a time in ticks divide the time in milliseconds by <code>portTICK_PERIOD_MS</code> or use the <code>pdMS_TO_TICKS()</code> macro.</td>
</tr>
<tr>
<td>FREERTOS_SO_UDPCKSUM_OUT</td>
<td>Only valid for UDP sockets.</td>
</tr>
<tr>
<td>IOptionName Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Turn on or off the generation of checksum values for outgoing UDP packets.</td>
<td></td>
</tr>
<tr>
<td>If IOptionName is FREERTOS_SO_UDPCKSUM_OUT and IOptionValue is NULL (0) then outgoing UDP packets will always have their checksum set to 0.</td>
<td></td>
</tr>
<tr>
<td>if IOptionName is FREERTOS_SO_UDPCKSUM_OUT and IOptionValue is any value other than NULL (0) then outgoing UDP packets will include a valid checksum value.</td>
<td></td>
</tr>
</tbody>
</table>

| FREERTOS_SO_SET_SEMAPHORE | Only available if ipconfigSOCKET_HAS_USER_SEMAPHORE is set to 1 in FreeRTOSIPConfig.h. |
| This option allows a reference to a semaphore to be passed to a socket. The TCP/IP RTOS task will then give to the semaphore on any of these events: |
| • Arrival of new data |
| • After delivering data, when new transmission buffer space becomes available |
| • An outgoing TCP connection has succeeded |
| • A new client has connected to a TCP socket |
| • A TCP connection was closed or reset |
Example use:

/* Declare the semaphore. */
SemaphoreHandle_t xSemaphore;
/* Create the semaphore. */
xSemaphore = xSemaphoreCreateBinary();
if( xSemaphore != NULL )
{
    /* Pass the semaphore to the socket. */
    FreeRTOS_setsockopt( xSocket,
                        0,
                        FREERTOS_SO_SET_SEMAPHORE,
                        ( void * )&xSemaphore,
                        sizeof( xSemaphore ) );

    /* The semaphore has been passed to the socket
    and will be used.

    Note: If a socket has a reference to a semaphore
    then the semaphore must not be deleted! To
    remove the semaphore call FreeRTOS_setsockopt()
    again, but this time with a NULL semaphore. */
    SemaphoreHandle_t xNoSem = NULL;
    FreeRTOS_setsockopt( xSocket,
                         0,
                         FREERTOS_SO_SET_SEMAPHORE,
                         ( void * ) &xNoSem,
                         sizeof( xNoSem ) );

    /* Now the semaphore can be deleted. */
    vSemaphoreDelete( xSemaphore );
}
<table>
<thead>
<tr>
<th><strong>IOptionName Value</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>FREERTOS_SO_RCVBUF</td>
<td>Only valid for TCP sockets. Sets the size of the receive buffer. Ideally this should be set to twice the size of the sliding window. This parameter can only be set between the socket being created and data being received on the socket because the size of the receive buffer is fixed after the buffer has been created. If <strong>IOptionName</strong> is FREERTOS_SO_RCVBUF then <strong>pvOptionValue</strong> must point to a variable of type int32_t. The receive buffer size is specified in bytes. Internally the specified size will get rounded up to the nearest multiple of the ipconfigTCP_MSS size. For example, if ipconfigTCP_MSS is 500 then setting a buffer size of 400 will result in a buffer size of 500 (1 * ipconfigTCP_MSS), setting a buffer size of 500 will result in a buffer size of 500 (1 * ipconfigTCP_MSS), and setting a buffer size of 510 will result in a buffer size of 1000 (2 * ipconfigTCP_MSS).</td>
</tr>
<tr>
<td>FREERTOS_SO_SNDBUF</td>
<td>Only valid for TCP sockets. Sets the size of the transmit buffer. This is not related to the size of the packets or the sliding window, it only sets the size of the buffer. This parameter can only be set between the socket being created and data being sent on the socket because the size of the send buffer is fixed after the</td>
</tr>
<tr>
<td>IOptionName Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>buffer has been created.</td>
<td></td>
</tr>
</tbody>
</table>

If IOptionName is FREERTOS\_SO\_SNDBUF then pvOptionValue must point to a variable of type int32\_t.

The receive buffer size is specified in bytes. Internally the specified size will get rounded up to the nearest multiple of the ipconfigTCP\_MSS size. For example, if ipconfigTCP\_MSS is 500 then setting a buffer size of 400 will result in a buffer size of 500 (1 * ipconfigTCP\_MSS), setting a buffer size of 500 will result in a buffer size of 500 (1 * ipconfigTCP\_MSS), and setting a buffer size of 510 will result in a buffer size of 1000 (2 * ipconfigTCP\_MSS).

<table>
<thead>
<tr>
<th>FREERTOS_SO_WIN_PROPERTIES</th>
<th>Advanced users only.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only valid for TCP sockets.</td>
<td></td>
</tr>
</tbody>
</table>

Sets the size of the receive buffer, receive sliding window, transmit buffer and transmit sliding window in one call.

The buffer and sliding window sizes can only be set between the socket being created and any data being sent to the socket or received from the socket.
Example use:

```c
/* Declare an xWinProperties structure. */
xWinProperties_t  xWinProps;

/* Fill in the required buffer and window sizes. */
/* Unit: bytes */
xWinProps.lTxBufSize = 4 * ipconfigTCP_MSS;
/* Unit: MSS */
xWinProps.lTxWinSize = 2;
/* Unit: bytes */
xWinProps.lRxBufSize = 4 * ipconfigTCP_MSS;
/* Unit: MSS */
xWinProps.lRxWinSize = 2;

/* Use the structure with the
FREERTOS_SO_WIN_PROPERTIES parameter in a call to
FreeRTOS_setsockopt(). */
FreeRTOS_setsockopt( xSocket,
0,
FREERTOS_SO_WIN_PROPERTIES,
( void * ) &xWinProps,
sizeof( xWinProps ) );
```

Example of setting the buffer and sliding window sizes
<table>
<thead>
<tr>
<th>lOptionName Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREERTOS_SO_REUSE_LISTEN_SOCKET</td>
<td>Only valid for TCP sockets. By default a listening socket will create a new socket to handle any accepted connections. FREERTOS_SO_REUSE_LISTEN_SOCKET can be used to change this behaviour so accepted connections are handled by the listening socket itself.</td>
</tr>
<tr>
<td></td>
<td>If lOptionName is FREERTOS_SO_REUSE_LISTEN_SOCKET then pvOptionValue must point to a variable of type BaseType_t that is set to 1 to indicate that the listening socket should be re-used for incoming connections, or 0 to indicate that the listening socket should create a new socket to handle each incoming connection.</td>
</tr>
<tr>
<td></td>
<td>Do not call FreeRTOS_accept() when using the re-use option. Instead FreeRTOS_recv() can be called right away.</td>
</tr>
<tr>
<td></td>
<td>FreeRTOS_connected() can be used to determine if a socket is connected.</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th><strong>IOptionName Value</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Example use:</td>
<td></td>
</tr>
<tr>
<td>BaseType_t xReuseSocket = pdTRUE;</td>
<td></td>
</tr>
<tr>
<td>FreeRTOS_setsockopt( xSocket,</td>
<td></td>
</tr>
<tr>
<td>0,</td>
<td></td>
</tr>
<tr>
<td>FREERTOS_SO_REUSE_LISTEN_SOCKET,</td>
<td></td>
</tr>
<tr>
<td>( void * ) &amp;xReuseSocket,</td>
<td></td>
</tr>
<tr>
<td>sizeof( xReuseSocket ) );</td>
<td></td>
</tr>
<tr>
<td><strong>Example of setting the re-use option to true</strong></td>
<td></td>
</tr>
<tr>
<td>FREERTOS_SO_CLOSE_AFTER_SEND</td>
<td>Only valid for TCP sockets.</td>
</tr>
<tr>
<td></td>
<td>FREERTOS_SO_CLOSE_AFTER_SEND TCP allows a socket to be closed immediately after the last data has been delivered. This option is useful for example in FTP where a file is being sent. Before calling FreeRTOS_send() for the last time, set this option, so the stack knows that the last packet must include the FIN flag. The stack will make sure that the connection is only closed after the last byte has been delivered, and acknowledged by the peer.</td>
</tr>
<tr>
<td></td>
<td>If IOptionName is FREERTOS_SO_CLOSE_AFTER_SEND then pvOptionValue must point to a variable of type BaseType_t that is set to 1 to indicate that the socket should be closed after the last data has been sent, or 0 to indicate that the socket should use its default behaviour of keeping the socket open until explicitly by either peer.</td>
</tr>
<tr>
<td>IOptionName Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>Example use:</strong></td>
<td></td>
</tr>
<tr>
<td>BaseType_t xCloseAfterNextSend = pdTRUE; FreeRTOS_setsockopt( xSocket, 0, FREERTOS_SO_CLOSE_AFTER_SEND, ( void * ) &amp;xCloseAfterNextSend, sizeof( xCloseAfterNextSend ) );</td>
<td></td>
</tr>
</tbody>
</table>

**Example of using the FREERTOS_SO_CLOSE_AFTER_SEND parameter**

<table>
<thead>
<tr>
<th>FREERTOS_SO_SET_FULL_SIZE</th>
<th>Only valid for TCP sockets.</th>
</tr>
</thead>
<tbody>
<tr>
<td>The FREERTOS_SO_SET_FULL_SIZE option tells the TCP/IP stack not to send any data from the socket until there is at least one complete MSS size of data ready to be sent. This option can be used to improve performance, but must be used with care. This option does not expire, so make sure the option is switched off on the last send, so that the last bytes (less than MSS) will also be delivered.</td>
<td></td>
</tr>
<tr>
<td>If IOptionName is FREERTOS_SO_SET_FULL_SIZE then pvOptionValue must point to a variable of type BaseType_t that is set to 1 to indicate that the socket should only send when there is at least MSS bytes waiting to be delivered, or 0 to indicate that the socket should use its default behaviour.</td>
<td></td>
</tr>
<tr>
<td>OptionName Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FREERTOS_SO_STOP_RX</td>
<td>Only valid for TCP sockets.</td>
</tr>
<tr>
<td></td>
<td>A TCP socket will constantly advertise a window size to its peer, so the peer knows how many bytes it may send until it has to wait for an acknowledge. This all happens automatically with a low and a high water mark.</td>
</tr>
<tr>
<td></td>
<td>FREERTOS_SO_STOP_RX forces the socket to advertise a window of zero, enabling the socket to temporarily stop receiving data.</td>
</tr>
<tr>
<td></td>
<td>If OptionName is FREERTOS_SO_STOP_RX then pvOptionValue must point to a variable of type BaseType_t that is set to 1 to indicate that the socket should advertise a window size of 0, or 0 to indicate that the socket should use its default behaviour.</td>
</tr>
<tr>
<td>Option Name</td>
<td>Value</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------</td>
</tr>
</tbody>
</table>

**Example Use:**

```c
BaseType_t xValue = pdTRUE;

/* Temporarily advertise a window size of 0 to stop reception of data */
FreeRTOS_setsockopt( xSocket,
    0,
    FREERTOS_SO_STOP_RX,
    ( void * ) &xValue,
    sizeof( xValue ) );

/* Do what ever you need to do. */
```

```c
xValue = pdFALSE;

/* Allow further reception */
FreeRTOS_setsockopt( xSocket,
    0,
    FREERTOS_SO_STOP_RX,
    ( void * ) &xValue,
    sizeof( xValue ) );
```

*Example of using the FREERTOS_SO_STOP_RX parameter*
<table>
<thead>
<tr>
<th>IOptionName Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREERTOS_SO_UDP_MAX_RX_PACKETS</td>
<td>Only valid for UDP sockets and if ipconfigUDP_MAX_RX_PACKETS is set to 1 in FreeRTOSIPConfig.h. The parameter ipconfigUDP_MAX_RX_PACKETS makes it possible to limit the maximum number of packets stored in one UDP socket. This option can change this limitation for an individual socket. If IOptionName is FREERTOS_SO_UDP_MAX_RX_PACKETS then pvOptionValue must point to a variable of type BaseType_t that holds the maximum number of RX packets that can be queued on the UDP socket.</td>
</tr>
</tbody>
</table>

**Example Use:**

```c
/* Allow a maximum of ten packets. */
BaseType_t xValue = 10;

FreeRTOS_setsockopt( xSocket,
    0,
    FREERTOS_SO_UDP_MAX_RX_PACKETS,
    ( void * ) &xValue,
    sizeof( xValue ) );
```

Example of using the FREERTOS_SO_UDP_MAX_RX_PACKETS parameter

<table>
<thead>
<tr>
<th>FREERTOS_SO_TCP_CONNECT_HANDLER</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only valid for TCP sockets. Stores the address of a function to call on connect and disconnect events on the TCP socket.</td>
<td></td>
</tr>
<tr>
<td><strong>IOptionName Value</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FREERTOS_SO_TCP_RECV_HANDLER</td>
<td>Only valid for TCP sockets. Stores the address of a function to call when data is received on the TCP socket.</td>
</tr>
<tr>
<td>FREERTOS_SO_TCP_SENT_HANDLER</td>
<td>Only valid for TCP sockets. Stores the address of a function to call when data sent to the TCP socket has been delivered and confirmed by the peer.</td>
</tr>
<tr>
<td>FREERTOS_SO_UDP_RECV_HANDLER</td>
<td>Only valid for UDP sockets. Stores the address of a function to call immediately upon reception of data on the UDP socket.</td>
</tr>
<tr>
<td>FREERTOS_SO_UDP_SENT_HANDLER</td>
<td>Only valid for UDP sockets. Stores the address of a function to call immediately that data has been sent to the UDP socket.</td>
</tr>
</tbody>
</table>
3.1.12 FreeRTOS_shutdown()

FreeRTOS_sockets.h
BaseType_t FreeRTOS_shutdown( xSocket_t xSocket, BaseType_t xHow );

Disable reads and writes on a connected TCP socket. A connected TCP socket must be gracefully shut down before it can be closed.

Parameters:

xSocket  The socket being shut down.

xHow     Must be set to FREERTOS_SHUT_RDWR.

OPENRTOS+TCP does not currently use the xHow parameter as it always shuts down both reads and writes. xHow is included to ensure the function prototype conforms to the expected Berkeley sockets standard, and for compatibility with future OPENRTOS+TCP versions which may accept other parameter values.

Returns:

If the shutdown request was successful then 0 is returned. The shutdown being complete is indicated by FreeRTOS_revc() calls on the socket resulting in FREERTOS_EINVAL being returned.

If xSocket is not a valid TCP socket then -pdFREERTOS_ERRNO_EOPNOTSUPP is returned.

If xSocket is a valid TCP socket but the socket is not connected to a remote socket then -pdFREERTOS_ERRNO_EOPNOTSUPP is returned.

Note that, because OPENRTOS does not implement errno, the behaviour in the presence of an error is necessarily different to that of connect() functions that are fully compliant with the expected Berkeley sockets behaviour.

Example usage:
The source code examples on both the Sending TCP Data and the Receiving TCP Data sections demonstrate a connected socket being shut down then closed.

### 3.1.13 FreeRTOS_closesocket()

FreeRTOS_sockets.h

BaseType_t FreeRTOS_closesocket( xSocket_t xSocket );

Close a socket.

The function is named FreeRTOS_closesocket() rather than simply FreeRTOS_close() to avoid potential name space collisions with functions in FreeRTOS+IO.

A socket should be shutdown gracefully before it is closed, and cannot be used after it has been closed.

**Parameters:**

**xSocket**  The handle of the socket being closed. The socket must have already been created (see FreeRTOS_socket()), and cannot be used after it has been closed.

**Returns:**

0 is always returned.

Although OPENRTOS+TCP does not [currently] use the return value in a meaningful way, the return value is included in the function prototype to ensure consistency with the expected standard Berkeley sockets API, and to ensure compatibility with future versions of OPENRTOS+TCP.

**Example usage:**

```c
/* OPENRTOS+TCP sockets include. */
#define "FreeRTOS_sockets.h"

void aFunction( void )
```

{  
xSocket_t xSocket;

     /* Create a socket. */
     xSocket = FreeRTOS_socket( FREERTOS_AF_INET, FREERTOS.SOCK_DGRAM,
     FREERTOS_IPPROTO_UDP );

     if( xSocket != FREERTOS_INVALID_SOCKET )
     {
         /*
          * The socket can now be used...
          */

         /* . . . */

     } /* . . . */

     /* Initialise a shutdown before closing the socket. */
     FreeRTOS_shutdown( xSocket );

     /* Wait for the socket to disconnect gracefully (indicated by FreeRTOS_recv() returning a FREERTOS_EINVAL error) before closing the socket. */
     while( FreeRTOS_recv( xSocket, pcBufferToTransmit, xTotalLengthToSend, 0 ) >= 0 )
     {
         /* Wait for shutdown to complete. If a receive block time is used then this delay will not be necessary as FreeRTOS_recv() will place the RTOS task into the Blocked state anyway. */
         vTaskDelay( pdTICKS_TO_MS( 250 ) );

         /* Note - real applications should implement a timeout here, not just loop forever. */
     }

     /* Close the socket again. */
3.1.14 FreeRTOS_select()

FreeRTOS_select() function is used in the following format:

```c
BaseType_t FreeRTOS_select( xSocketSet_t xSocketSet, TickType_t xBlockTimeTicks );
```

This function blocks on a "socket set" until an event occurs for at least one of its members.

**Socket Sets** allow an application RTOS task to block when reading from or writing to multiple sockets at the same time. Instead of blocking on an individual socket, a RTOS task instead blocks on the set.

To use a socket set:

1. Create a socket set by calling FreeRTOS_createsocketset(). A socket set is equivalent to the Berkeley sockets fd_set type.
2. Add one or more sockets to the set using calls to FreeRTOS_FD_SET(). FreeRTOS_FD_SET() is equivalent to the Berkeley sockets FD_SET() macro.
3. Call FreeRTOS_Select() to test the sockets in the set to see if any of the event types has occurred. This can be either a READ, WRITE of an EXCEPTION event.
4. Test each individual socket with FreeRTOS_FD_ISSET() to see if it has an event the user is interested in.

A socket can only be a member of one set at any time.

FreeRTOS_FD_CLR() removes a socket from a set.

**Parameters:**

- `xSocketSet` The socket set being tested.
**xBlockTimeTicks**  The maximum time, in ticks, that the calling RTOS task will remain in the Blocked state (with other tasks executing) to wait for a member of the socket set to get an event.

**Returns:**

If xBlockTimeTicks expired before a socket in the socket set had an event, then zero is returned. Otherwise a non-zero value is returned. All sockets which belong to the socket set must be checked by calling FreeRTOS_FD_ISSET()

**Example usage:**

```c
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

/* OPENRTOS+TCP includes. */
#include "FreeRTOS_IP.h"
#include "FreeRTOS_Sockets.h"

static void prvMultipleSocketRxTask( void *pvParameters )
{
  xSocketSet_t xFD_Set;
  BaseType_t xResult;
  xSocket_t xSockets[2];
  struct freertos_sockaddr xAddress;
  uint32_t xClientLength = sizeof( struct freertos_sockaddr ), x;
  uint32_t ulReceivedValue = 0;
  int32_t lBytes;
  const TickType_t xRxBlockTime = 0;

  /* Create the set of sockets that will be passed into FreeRTOS_select(). */
```
xFD_Set = FreeRTOS_createsocketset();

/* Create two sockets to add to the set. */
for( x = 0; x < 2; x++ )
{
    /* Create the socket. */
    xSockets[x] = FreeRTOS_socket( FREERTOS_AF_INET,
                                  FREERTOS.SOCK_DGRAM,
                                  FREERTOS_IPPROTO_UDP );

    /* Bind the socket to a port number 1000, 1001... */
    xAddress.sin_port = FreeRTOS_htons( 1000 + x );
    FreeRTOS_bind( xSockets[x], &xAddress, sizeof( struct freertos_sockaddr ) );

    /* Once it has been tested that a socket has a eSELECT_READ
    event, blocking on a read call is not necessary any more. Set the
    Rx block time to 0. */
    FreeRTOS_setsockopt( xSockets[x], 0, FREERTOS_SO_RCVTIMEO,
                          &xRxBlockTime, sizeof( xRxBlockTime ) );

    /* Add the created socket to the set for the READ event only. */
    FreeRTOS_FD_SET( xSockets[x], xFD_Set, eSELECT_READ );
}

for(;; )
{
    /* Wait for any event within the socket set. */
    xResult = FreeRTOS_select( xFD_Set, portMAX_DELAY );
    if( xResult != 0 )
    {
        /* The return value should never be zero because FreeRTOS_select() was
called
        with an indefinite delay (assuming INCLUDE_vTaskSuspend is set to 1)."
Now check each socket which belongs to the set if it had an event */

for( x = 0; x < 2; x++ )
{
    if( FreeRTOS_FD_ISSET ( xSockets[x], xFD_Set ) )
    {
        /* Read from the socket. */
        lBytes = FreeRTOS_recvfrom( xSockets[x], &( ulReceivedValue ),
                                   sizeof( uint32_t ), 0, &xAddress,
                                   &xClientLength );
        /* Process the received data here. */
    }
}

Figure 3-9 Example use of the FreeRTOS_select() API function

3.2 Miscellaneous Sockets functions

Propitiatory socket functions are lower case to match Berkeley convention.

3.2.1 FreeRTOS_createsocketset()

FreeRTOS_sockets.h

xSocketSet_t FreeRTOS_createsocketset() ( unsigned BaseType_t
ulEventQueueLength );

Create a socket set for use with the FreeRTOS_select() function.

Socket Sets allow an application RTOS task to block when reading from multiple sockets at the same time. Instead of blocking on an individual socket, a RTOS task instead blocks on the set.

To use a socket set:
1. Create a socket set by calling FreeRTOS_createsocketset(). A socket set is equivalent to the Berkeley sockets fd_set type.

2. Add one or more sockets to the set using calls to FreeRTOS_FD_SET(). FreeRTOS_FD_SET() is equivalent to the Berkeley sockets FD_SET() macro.

3. Call FreeRTOS_Select() to test the sockets in the set to see if any contain data that is waiting to be read.

4. Read from the socket returned by FreeRTOS_select() (if any) using a call to FreeRTOS_recvfrom() as normal.

A socket can only be a member of one set at any time.

FreeRTOS_FD_CLR() removes a socket from a set.

**Parameters:**

**uxEventQueueLength**  
A receive event is generated each time a socket in the socket set receives data. uxEventQueueLength sets the maximum number of receive events that can be stored by the socket set at any one time.

If a packet is received by a socket that is a member of a socket set, and the socket set's event queue is full, then the packet will be dropped.

**Returns:**

If the socket set was created then a handle to the created socket set is returned. If the socket set was not created (because there was insufficient OPENRTOS heap memory available) then NULL is returned.

**Example usage:**

See the example on the FreeRTOS_select() documentation section.

### 3.2.2 FreeRTOS_createsocketset()

FreeRTOS_sockets.h

```c
xSocketSet_t FreeRTOS_createsocketset() ( unsigned BaseType_t uxEventQueueLength );
```
Create a socket set for use with the FreeRTOS_select() function.

*Socket Sets* allow an application RTOS task to block when reading from multiple sockets at the same time. Instead of blocking on an individual socket, a RTOS task instead blocks on the set.

To use a socket set:

1. Create a socket set by calling FreeRTOS_createsocketset(). A socket set is equivalent to the Berkeley sockets fd_set type.
2. Add one or more sockets to the set using calls to FreeRTOS_FD_SET(). FreeRTOS_FD_SET() is equivalent to the Berkeley sockets FD_SET() macro.
3. Call FreeRTOS_Select() to test the sockets in the set to see if any contain data that is waiting to be read.
4. Read from the socket returned by FreeRTOS_select() (if any) using a call to FreeRTOS_recvfrom() as normal.

A socket can only be a member of one set at any time.

FreeRTOS_FD_CLR() removes a socket from a set.

**Parameters:**

uxEventQueueLength  A receive event is generated each time a socket in the socket set receives data. uxEventQueueLength sets the maximum number of receive events that can be stored by the socket set at any one time.

If a packet is received by a socket that is a member of a socket set, and the socket set's event queue is full, then the packet will be dropped.

**Returns:**

If the socket set was created then a handle to the created socket set is returned. If the socket set was not created (because there was insufficient OPENRTOS heap memory available) then NULL is returned.

**Example usage:**

See the example on the FreeRTOS_select() documentation section.
3.2.3 FreeRTOS_FD_SET()

FreeRTOS_sockets.h

```c
void FreeRTOS_FD_SET(xSocket_t xSocket, xSocketSet_t xSocketSet, BaseType_t xSelectBits);
```

Add a socket to a socket set and set event bits for this socket. Possible values for 'xSelectBits':

```
eSELECT_READ = 0x0001

eSELECT_WRITE = 0x0002

eSELECT_EXCEPT = 0x0004
```

In most cases, only the eSELECT_READ event will be used. A eSELECT_READ event occurs as long as a socket has data to be read. For a TCP socket in listening mode, a eSELECT_READ event means that it has received a new connection.

If the caller includes eSELECT_WRITE for some socket, a call to FreeRTOS_select() will return immediately as long as the sockets involved have space for writing. While a TCP socket is actively connecting to a peer, the 'eSELECT_WRITE' event will be triggered as soon as the connection is established.

In other words, once a eSELECT_WRITE event has fired, it should either be disabled before the next call to FreeRTOS_select(), or the caller should write enough data to the socket so that its transmission buffer is totally filled.

A eSELECT_EXCEPT event occurs when a TCP socket gets disconnected.

Socket Sets allow an application RTOS task to block when reading from multiple sockets from a single RTOS task. Instead of blocking on an individual socket, a RTOS task instead blocks on the set.

To use a socket set:

1. Create a socket set by calling FreeRTOS_createsocketset(). A socket set is equivalent to the Berkeley sockets fd_set type.
2. Add one or more sockets to the set using calls to FreeRTOS_FD_SET(). FreeRTOS_FD_SET() is equivalent to the Berkeley sockets FD_SET() macro.

3. Call FreeRTOS_Select() to test the sockets in the set to see if an event has happened to any of the sockets.

4. After FreeRTOS_select() has returned a non-zero value, check all sockets belonging to the socket set if they had an event by calling FreeRTOS_FD_ISSET().

A socket can only be a member of one set at any time.

FreeRTOS_FD_CLR() clears event bits or it removes a socket from a set.
Parameters:

\[
\begin{align*}
\texttt{xSocket} & \quad \text{The socket being added to the set.} \\
\texttt{xSocketSet} & \quad \text{The socket set to which the socket is being added.} \\
\texttt{xSelectBits} & \quad \text{The types of events which will unblock a call to FreeRTOS\_select().}
\end{align*}
\]

Returns:

\[\text{void.}\]

Example usage:

See the example on the FreeRTOS\_select() documentation section.

### 3.2.4 FreeRTOS\_FD\_CLR()

FreeRTOS\_sockets.h

```c
void FreeRTOS\_FD\_CLR( xSocket\_t xSocket, xSocketSet\_t xSocketSet, BaseType\_t xBitsToClear );
```

Clears event bits and/or removes a socket from a socket set.

Each socket member has its own set of event bits, which can be a combination of:

\[
\begin{align*}
\texttt{eSELECT\_READ} & = 0x0001 \\
\texttt{eSELECT\_WRITE} & = 0x0002 \\
\texttt{eSELECT\_EXCEPT} & = 0x0004
\end{align*}
\]

FreeRTOS\_FD\_CLR() clears event bits. Only when all event bits for a socket are cleared, the socket is removed from the socket set.
Parameters:

\textbf{xSocket} \hspace{1cm} \text{The socket being removed from the socket set.}

\textbf{xSocketSet} \hspace{1cm} \text{The socket set the socket is a member of.}

\textbf{xBitsToClear} \hspace{1cm} \text{The bits which should be cleared, use 'eSELECT_ALL' to clear all bits and remove the socket from the set.}

Returns:

\texttt{void}

```c
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

/* OPENRTOS+TCP includes. */
#include "FreeRTOS_IP.h"
#include "FreeRTOS_Sockets.h"

void vConnectExample( )
{
  xSocket_t xSocket;
  struct freertos_sockaddr xEchoServerAddress;
  const TickType_t xZeroTimeOut = 0;
  xSocketSet_t xSocketSet;

  /* Create a TCP socket. */
  xSocket = FreeRTOS_socket( FREERTOS_AF_INET, FREERTOS_SOCK_STREAM,
                              FREERTOS_IPPROTO_TCP );
```
/* Create a socket set. */
xSocketSet = FreeRTOS_createsocketset();

/*/ Make the socket a member of the set.
Only the WRITE event can unblock a call to select() */
FreeRTOS_FD_SET( xSocket, xSocketSet, eSELECT_WRITE );

/*/ When working with select(), time-outs on API's aren't necessary */
FreeRTOS_setsockopt( xSocket, 0, FREERTOS_SO_RCVTIMEO, &xZeroTimeOut, sizeof( xZeroTimeOut ) );
FreeRTOS_setsockopt( xSocket, 0, FREERTOS_SO_SNDTIMEO, &xZeroTimeOut, sizeof( xZeroTimeOut ) );

/*/ Fill in the peer's address */
xEchoServerAddress.sin_port = FreeRTOS_htons( echoECHO_PORT );
xEchoServerAddress.sin_addr = FreeRTOS_inet_addr_quick( configECHO_SERVER_ADDR0, configECHO_SERVER_ADDR1, configECHO_SERVER_ADDR2, configECHO_SERVER_ADDR3 );

/*/ Now initiate an active connect procedure to a peer. This call is non-blocking */
FreeRTOS_connect( xSocket, &xEchoServerAddress, sizeof( xEchoServerAddress ) );

/*/ Now block for at most 30 seconds. A successful connection will unblock select() with a eSELECT_WRITE event */
if( FreeRTOS_select( xSocketSet, 30000 ) != 0 )
{
    BaseType_t xMask = FreeRTOS_FD_ISSET( xSocket, xSocketSet );
    if( xMask != 0 )
    {
        /* Clear the WRITE event bit, it is not interesting any more */
        FreeRTOS_FD_CLR( xSocket, xSocketSet, eSELECT_WRITE );
    }
}
/* Set the READ event bit */
    FreeRTOS_FD_SET( xSocket, xSocketSet, eSELECT_READ );
}
}

Figure 3-10 Example use of the FreeRTOS_FD_SET / FD_CLR / FD_ISSET() API functions

3.2.5 FreeRTOS_FD_ISSET()

FreeRTOS_sockets.h

BaseType_t FreeRTOS_FD_ISSET( xSocket_t xSocket, xSocketSet_t xSocketSet );

Check if a socket has an event bit set.

Socket Sets allow an application RTOS task to block when reading from or writing to multiple sockets. Instead of blocking on an individual socket, a RTOS task instead blocks on the set.

To use a socket set:

1. Create a socket set by calling FreeRTOS_createsocketset(). A socket set is equivalent to the Berkeley sockets fd_set type.
2. Add one or more sockets to the set using calls to FreeRTOS_FD_SET(). FreeRTOS_FD_SET() is equivalent to the Berkeley sockets FD_SET() macro.
3. Call FreeRTOS_Select() to test the sockets in the set to see if any contain data that is waiting to be read.
4. After FreeRTOS_select() has returned non-zero, test each individual socket with FreeRTOS_FD_ISSET() to see if it needs attention.

A socket can only be a member of one set at any time.

FreeRTOS_FD_CLR() clears event bits and removes a socket from a set.

Parameters:

xSocket The socket being added to the set.
**xSocketSet**   The socket set to which the socket is being added.

**Returns:**

The function returns a bit-mask of the values eSELECT_READ (1), eSELECT_WRITE (2) and eSELECT_EXCEPT (4). If the return value equals zero, the socket has currently no event of a type indicated in a call to FreeRTOS_FD_SET().

**Example usage:**

See the example on the FreeRTOS_select() documentation section.

### 3.2.6 FreeRTOS_inet_addr()

FreeRTOS_inet_addr() is a function that converts an IP address expressed in decimal dot notation (for example "192.168.0.100") into a 32-bit IP address in network byte order.

FreeRTOS_inet_addr_quick() is a macro that converts an IP address expressed as four separate numeric octets (for example 192, 168, 0, 100) into an IP address expressed as a 32-bit number in network byte order.

FreeRTOS_inet_addr_quick() is the preferred method because of its smaller size and faster execution. FreeRTOS_inet_addr() is provided because it conforms to the expected Berkeley sockets function prototype.

ipconfigINCLUDE_FULL_INET_ADDR must be set to 1 in FreeRTOSIPConfig.h for FreeRTOS_inet_addr() to be available. FreeRTOS_inet_addr_quick() is always available.

**Parameters:**

**pucIPAddress**   A pointer to a string that contains the IP address being converted in decimal dot format.
Returns:

If the format of the string pointed to by the puclIPAddress parameter is valid then the same IP address expressed as a 32-bit number in network byte order is returned. In all other cases 0 is returned.

Example usage:

This example sends a string to port 5000 of IP address 192.168.0.100, using FreeRTOS_inet_addr() to convert the IP address from a string to the necessary 32-bit format. The socket is passed in as the function parameter, and is assumed to have already been created using a call to FreeRTOS_socket(). If ipconfigALLOW_SOCKET_SEND_WITHOUT_BIND is not set to 1 in FreeRTOSIPConfig.h, then the socket is also assumed to have been bound to a port number using FreeRTOS_bind().

```c
/* OPENRTOS+TCP sockets include. */
#define "FreeRTOSsockets.h"

void aFunction( xSocket_t xSocket )
{
    struct freertos_sockaddr xDestinationAddress;
    const int8_t * pcMessageToSend = "String being sent";

    /* Generate the destination address. */
    xDestinationAddress.sin_addr = FreeRTOS_inet_addr( "192.168.0.100" );
    xDestinationAddress.sin_port = FreeRTOS_htons( 5000 );

    /* Send the message. */
    iReturned = FreeRTOS_sendto( /* The socket being send to. */
        xSocket,
        /* The data being sent. */
        pcMessageToSend,
```
/* The length of the data being sent. */
strlen( pcMessageToSend ),
/* ulFlags with the FREERTOS_ZERO_COPY bit clear. */
0,
/* Where the data is being sent. */
&xDestinationAddress,
/* Not used but should be set as shown. */
sizeof( xDestinationAddress )
);    

Figure 3-11 Example use of the FreeRTOS_inet_addr_quick() API function

3.2.7 FreeRTOS_inet_ntoa()

FreeRTOS_inet_ntoa( uint32_t ulIPAddress, uint8_t *pucBuffer )

A macro that converts an IP address expressed as a 32-bit number in network byte order to a string in decimal dot notation (for example "192.168.0.200").

The standard Berkeley sockets inet_ntoa() function returns a pointer to a string that is normally stored in a global buffer. FreeRTOS_inet_ntoa() deviates from the normal semantics by instead taking the buffer into which the string is written as a parameter. The deviation is to ensure the macro is re-entrant and thread aware.

Parameters:

ulIPAddress An IP address expressed as a 32-bit value in network byte order.

pucBuffer A pointer to a buffer into which the IP address will be written in decimal dot notation.

Example usage:
The example on the FreeRTOS_recvfrom() documentation section demonstrates FreeRTOS_inet_ntoa() being used to print the IP address from which a message was received.

The example on the FreeRTOS_GetAddressConfiguration() documentation section demonstrates FreeRTOS_inet_ntoa() being used to print out the network configuration - including the IP address and net mask of the node, and the IP addresses of the gateway and DNS server respectively.

3.2.8 FreeRTOS_htons(), FreeRTOS_ntohs(), FreeRTOS_htonl() & FreeRTOS_ntohl()

FreeRTOS_sockets.h

```c
uint16_t FreeRTOS_htons( uint16_t usValueToSwap );
uint16_t FreeRTOS_ntohs( uint16_t usValueToSwap );

uint32_t FreeRTOS_htonl( uint32_t ulValueToSwap );
uint32_t FreeRTOS_ntohl( uint32_t ulValueToSwap );
```

The Byte Order and Endian section of the Embedded Networking Basics and Glossary section provides an explanation of byte order considerations in IP networks.

The definition of ipconfigBYTE_ORDER in FreeRTOSIPConfig.h must be correct for the microcontroller on which OPENRTOS+TCP will run. If the microcontroller is big endian then ipconfigBYTE_ORDER must be set to pdFREERTOS_BIG_ENDIAN. If the microcontroller is little endian then ipconfigBYTE_ORDER must be set to pdFREERTOS_LITTLE_ENDIAN.

When ipconfigBYTE_ORDER is set to pdFREERTOS_LITTLE_ENDIAN:

- FreeRTOS_htons and FreeRTOS_ntohs() return the value of their 16-bit parameter with the high and low bytes swapped. For example, if the usValueToSwap parameter is 0x1122, then both macros return 0x2211.
- FreeRTOShtonl and FreeRTOS_ntohl() return the value of their 32-bit parameter with the byte order reversed. For example, if the ulValueToSwap parameter is 0x11223344, then both macros return 0x44332211.

If the microcontroller is big endian (and therefore ipconfigBYTE_ORDER set to pdFREERTOS_BIG_ENDIAN) then the byte order of the microcontroller and the byte order of the network already match, and all four byte swapping macros are defined to have no effect.
Byte swapping macros are primarily used when specifying the IP address and port number that make up a socket address.

**Example usage:**

The examples on the FreeRTOS_socket(), FreeRTOS_inet_addr(), FreeRTOS_sendto() documentation sections demonstrate the use of FreeRTOS_htons().

The example on the FreeRTOS_recvfrom() documentation section demonstrates the use of FreeRTOS_ntohs().

### 3.2.9 FreeRTOS_outstanding()

*FreeRTOS.Sockets.h*

```c
BaseType_t FreeRTOS_outstanding( xSocket_t xSocket );
```

Returns the number of bytes in a TCP socket's Tx stream that are yet to be transmitted.

**Parameters:**

- **xSocket** The socket being queried.

**Returns:**

If the socket referenced by the xSocket parameter is not a TCP socket then `- pdFREERTOS_ERRNO_EINVAL` is returned.

If the socket referenced by the xSocket parameter does not yet have a Tx stream then 0 is returned (the Tx stream is not created until it is required).

In all other cases the returned value is the number of bytes that remain in the socket's Tx stream.

### 3.2.10 FreeRTOS_recvcount()

*FreeRTOS.Sockets.h*

```c
BaseType_t FreeRTOS_recvcount( xSocket_t xSocket );
```

Returns the number of bytes in a TCP socket's Rx stream that are yet to be read.
Parameters:

\( x\text{Socket} \) The socket being queried.

Returns:

If the socket referenced by the \( x\text{Socket} \) parameter is not a TCP socket then -pdFREERTOS_ERRNO_EINVAL is returned.

If the socket referenced by the \( x\text{Socket} \) parameter does not yet have an Rx stream then 0 is returned (the Rx stream is not created until it is required).

In all other cases the returned value is the number of bytes that remain in the socket's Rx stream.

3.2.11 FreeRTOS_issocketconnected()
FreeRTOS_sockets.h
BaseType_t FreeRTOS_issocketconnected( xSocket_t xSocket );

Tests to see if a socket is connected.

Parameters:

\( x\text{Socket} \) The socket being queried.

Returns:

If the socket referenced by the \( x\text{Socket} \) parameter is not a TCP socket then -pdFREERTOS_ERRNO_EINVAL is returned.

If the socket is in the Established or a FIN wait state then pdTRUE is returned. Otherwise pdFALSE is returned;

3.2.12 FreeRTOS_getremoteaddress()
FreeRTOS_sockets.h
BaseType_t FreeRTOS_getremoteaddress( xSocket_t xSocket, struct freertos_sockaddr *pxAddress );

Returns the remote IP address and port of a connected TCP socket.
3.2.13 FreeRTOS_maywrite()

FreeRTOS_maywrite( xSocket_t xSocket );

Returns the number of bytes that can be added to a TCP socket's Tx stream before the Tx stream is full.

Parameters:

xSocket  The socket being queried.

Returns:

If the socket referenced by the xSocket parameter is not a TCP socket then - pdFREERTOS_ERRNO_EINVAL is returned.

If the socket is not in a state that allows data to be sent (for example it is in the Listening state or is in the process of being shut down) then -1 is returned.

If the socket is in the Established the returned value is the number of bytes that can be added to the socket's Tx stream.
3.3 IP functions

Propitiatory IP functions are mixed case to match OPENRTOS convention.

3.3.1 FreeRTOS_IPInit()

FreeRTOS_sockets.h

BaseType_t FreeRTOS_IPInit( const uint8_t ucIPAddress[ ipIP_ADDRESS_LENGTH_BYTES ],
                          const uint8_t ucNetMask[ ipIP_ADDRESS_LENGTH_BYTES ],
                          const uint8_t ucGatewayAddress[ ipIP_ADDRESS_LENGTH_BYTES ],
                          const uint8_t ucDNSServerAddress[ ipIP_ADDRESS_LENGTH_BYTES ],
                          const uint8_t ucMACAddress[ ipMAC_ADDRESS_LENGTH_BYTES ] );

Initialises the OPENRTOS+TCP stack. FreeRTOS_IPInit() must be called before any other OPENRTOS+TCP function.

ipIP_ADDRESS_LENGTH_BYTES is defined as 4. ipMAC_ADDRESS_LENGTH_BYTES is defined as 6.

Parameters:

ucIPAddress

If ipconfigUSE_DHCP is set to 0 (in FreeRTOSIPConfig.h) then the IP address of the network node is static and configured by the value of ucIPAddress.

If ipconfigUSE_DHCP is set to 1 then OPENRTOS+TCP will attempt to obtain an IP address from a DHCP sever. If an IP address cannot be obtained then the network node will revert to using the static IP address configured by the value of ucIPAddress.

The IP address is specified as a four byte array, where index 0 holds the first octet of the IP address and index 3 holds the last octet of the IP address. See the example below.

ucNetmask

If ipconfigUSE_DHCP is set to 0 (in FreeRTOSIPConfig.h) then the net mask of the network node is static and configured by the value of ucNetmask.

If ipconfigUSE_DHCP is set to 1 then OPENRTOS+TCP will attempt to
obtain a net mask from a DHCP sever. If a net mask cannot be obtained then the network node will revert to using the static net mask configured by the value of ucNetMask.

The net mask is specified as a four byte array, where index 0 holds the first octet of the mask and index 3 holds the last octet of the mask. See the example below.

**ucGatewayAddress**

If ipconfigUSE_DHCP is set to 0 (in FreeRTOSIPConfig.h) then the IP address of the network gateway is static and configured by the value of ucGatewayAddress.

If ipconfigUSE_DHCP is set to 1 then OPENRTOS+TCP will attempt to obtain the IP address of the network gateway from a DHCP sever. If a gateway IP address cannot be obtained then the network node will revert to using the static IP address configured by the value of ucGatewayAddress as the gateway address.

The IP address is specified as a four byte array, where index 0 holds the first octet of the IP address and index 3 holds the last octet of the IP address. See the example below.

**ucDNSServerAddress**

If ipconfigUSE_DHCP is set to 0 (in FreeRTOSIPConfig.h) then the IP address of the DNS server is static and configured by the value of ucDNSServerAddress.

If ipconfigUSE_DHCP is set to 1 then OPENRTOS+TCP will attempt to obtain the IP address of the DNS server from a DHCP sever. If a DNS server IP address cannot be obtained then the network node will revert to using the static IP address configured by the value of ucDNSServerAddress as the DNS server address.

The IP address is specified as a four byte array, where index 0 holds the first octet of the IP address and index 3 holds the last octet of the IP address. See the example below.

**ucMACAddress**

The MAC address of the network node.

The MAC IP address is specified as a six byte array, where index 0 holds the first octet of the MAC address and index 5 holds the last octet of the
MAC address. See the example below.

**Returns:**

`pdPASS` is returned if the TCP/IP stack was initialised successfully. `pdFAIL` is returned if the TCP/IP stack was not initialised - either because `FreeRTOS_IPInit()` has already been called previously or because either the network buffers or the IP RTOS task could not be created.

**Example usage:**
/* OPENRTOS+TCP sockets include. */
#include "FreeRTOS_sockets.h"

/* Define the network addressing. These parameters will be used if either
ipconfigUDE_DHCP is 0 or if ipconfigUSE_DHCP is 1 but DHCP auto configuration
failed. */
static const uint8_t ucIPAddress[ 4 ] = { 192, 168, 0, 200 ;
static const uint8_t ucGatewayAddress[ 4 ] = { 192, 168, 0, 1 ;

/* The following is the address of an OpenDNS server. */
static const uint8_t ucDNSServerAddress[ 4 ] = { 208, 67, 222, 222 ;

/* The MAC address array is not declared const as the MAC address will normally
be read from an EEPROM and not hard coded (in real deployed applications).*/
static uint8_t ucMACAddress[ 6 ] = { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55 ;

void aFunction( void )
{
    /* Initialise the TCP/IP stack. */
    FreeRTOS_IPInit( ucIPAddress,
        ucNetMask,
        ucGatewayAddress,
        ucDNSServerAddress,
        ucMACAddress );
}

Figure 3-12 Example use of the FreeRTOS_IPInit() API function

3.3.2 FreeRTOS_GetUDPPayloadBuffer()

FreeRTOS_sockets.h

void *FreeRTOS_GetUDPPayloadBuffer( size_t xRequestedSizeBytes, TickType_t xBlockTimeTicks );
Obtains a buffer from the TCP/IP stack for use with the zero copy interface.

The zero copy interface for transmitting data is described on the FreeRTOS_sendto() documentation section.

**Parameters:**

- **xRequestedSizeBytes** The size of the buffer being requested. The size is specified in bytes.

  - The maximum time the calling RTOS task is prepared to wait for a buffer if one is not immediately available.

  - If a buffer is not available then the calling RTOS task will be held in the Blocked state (so other tasks can execute) until either a buffer becomes available or the block time expires.

  - The block time is specified in ticks. Milliseconds can be converted to ticks by dividing the time in milliseconds by portTICK_PERIOD_MS.

  - To prevent deadlocks the maximum block time is capped to ipconfigMAX_SEND_BLOCK_TIME_TICKS.

  - ipconfigMAX_SEND_BLOCK_TIME_TICKS is defined in FreeRTOSIPConfig.h

**Returns:**

- If a buffer was obtained then a pointer to the obtained buffer is returned.

- If a buffer could not be obtained then NULL is returned.

**Example usage:**

The FreeRTOS_sendto() documentation section contains an example zero copy send operation that includes a call to FreeRTOS_GetUDPPayloadBuffer().
void FreeRTOS_ReleaseUDPPayloadBuffer( void *pvBuffer );

FreeRTOS_ReleaseUDPPayloadBuffer() is used to return to the TCP/IP stack a buffer that was used with the zero copy interface.

The zero copy interface for transmitting data is described on the FreeRTOS_sendto() documentation section.

The zero copy interface for receiving data is described on the FreeRTOS_recvfrom() documentation section.

A buffer needs to be returned to the stack if:

1. It is obtained from a call to FreeRTOS_recvfrom() and the data it contains is no longer required, or
2. It was obtained from a call to FreeRTOS_GetUDPPayloadBuffer() but the buffer could not be passed into the TCP/IP stack (the call to FreeRTOS_sendto() in which the buffer was used failed).

A buffer can also be re-used rather than returned to the TCP/IP stack.

**Parameters:**

*pvBuffer*  The buffer that is being returned to the TCP/IP stack.

**Example usage:**

The FreeRTOS_sendto() documentation section includes an example zero copy send operation that demonstrates how to use FreeRTOS_ReleaseUDPPayloadBuffer() when the send operation fails.

The FreeRTOS_recvfrom() documentation section includes an example that demonstrates how to use FreeRTOS_ReleaseUDPPayloadBuffer() to release a buffer obtained from a call to FreeRTOS_recvfrom().

**3.3.4 FreeRTOS_SendPingRequest()**

FreeRTOS.Sockets.h
BaseType_t FreeRTOS_SendPingRequest( uint32_t ulIPAddress,
         size_t xNumberOfBytesToSend,
         TickType_t xBlockTimeTicks );

Send a ping (ICMP echo) request to a remote computer.

ipconfigSUPPORT_OUTGOING_PINGS must be set to 1 in FreeRTOSIPConfig.h for
FreeRTOS_SendPingRequest() to be available.

The TCP/IP stack calls the application defined vApplicationPingReplyHook() hook (or callback)
function when it receives a reply to an outgoing ping request.

Parameters:

ulIPAddress The IP address to which the ping request is sent.

The IP address is expressed as a 32-bit number in network byte
order.

xNumberOfBytesToSend The number of data bytes to send in the ping request.

xBlockTimeTicks The maximum time the calling RTOS task is prepared to wait for a
network buffer if one is not immediately available.

If a network buffer is not available then the calling RTOS task will be
held in the Blocked state (so other tasks can execute) until either a
buffer becomes available and therefore the ping request transmitted,
or the block time expires.

The block time is specified in ticks. Milliseconds can be converted to
ticks by dividing the time in milliseconds by portTICK_PERIOD_MS.

Returns:
If a ping request is successfully sent then the sequence number sent in the ping message is returned to allow the application writer to match ping requests transmitted with ping replies received. See the example below.

If a ping request could not be sent then pdFAIL is returned.

**Example usage:**

This example defines two functions. vSendPing() transmits 8 bytes to a remote IP address. vApplicationPingReplyHook() is the standard OPENRTOS+TCP ping reply callback function. vApplicationPingReplyHook() receives the ping reply, then sends the received sequence number to vSendPing() where it is compared to the sequence number from the ping request.

```c
/* OPENRTOS+TCP sockets include. */
#define "FreeRTOS_sockets.h"

/* This example code snippet assumes the queue has already been created! */
QueueHandle_t xPingReplyQueue;

/* If ipconfigSUPPORT_OUTGOING_PINGS is set to 1 in FreeRTOSIPConfig.h then
vApplicationPingReplyHook() is called by the TCP/IP stack when the stack receives a
ping reply.

void vApplicationPingReplyHook( ePingReplyStatus_t eStatus, uint16_t usIdentifier )
{
    switch( eStatus )
    {
        case eSuccess :
        /* A valid ping reply has been received. Post the sequence number
           on the queue that is read by the vSendPing() function below. Do
           not wait more than 10ms trying to send the message if it cannot be
           sent immediately because this function is called from the TCP/IP
           RTOS task - blocking in this function will block the TCP/IP RTOS task. */
           xQueueSend( xPingReplyQueue, &usIdentifier, 10 / portTICK_PERIOD_MS );
           break;
```
case eInvalidChecksum :
case eInvalidData :
    /* A reply was received but it was not valid. */
    break;
}
}

BaseType_t vSendPing( const int8_t *pcIPAddress )
{
    uint16_t usRequestSequenceNumber, usReplySequenceNumber;
    uint32_t ulIPAddress;

    /* The pcIPAddress parameter holds the destination IP address as a string in
decimal dot notation (for example, "192.168.0.200"). Convert the string into
the required 32-bit format. */
    ulIPAddress = FreeRTOS_inet_addr( pcIPAddress );

    /* Send a ping containing 8 data bytes. Wait (in the Blocked state) a
maximum of 100ms for a network buffer into which the generated ping request
can be written and sent. */
    usRequestSequenceNumber = FreeRTOS_SendPingRequest( ulIPAddress, 8, 100 /
portTICK_PERIOD_MS );

    if( usRequestSequenceNumber == pdFAIL )
    {
        /* The ping could not be sent because a network buffer could not be
obtained within 100ms of FreeRTOS_SendPingRequest() being called. */
    }
    else
    {
        /* The ping was sent. Wait 200ms for a reply. The sequence number from
each reply is sent from the vApplicationPingReplyHook() on the xPingReplyQueue queue (this is not standard behaviour, but implemented in the example function above). It is assumed the queue was created before this function was called! */
if( xQueueReceive( xPingReplyQueue,
               &usReplySequenceNumber,
               200 / portTICK_PERIOD_MS ) == pdPASS )
{
    /* A ping reply was received. Was it a reply to the ping just sent? */
    if( usRequestSequenceNumber == usReplySequenceNumber )
    {
        /* This was a reply to the request just sent. */
    }
}

Figure 3-13 Example use of the FreeRTOS_SendPingRequest() API function

3.3.5 FreeRTOS_GetAddressConfiguration()
FreeRTOS.Sockets.h

void FreeRTOS_GetAddressConfiguration( uint32_t *pulIPAddress,
                                        uint32_t *pulNetMask,
                                        uint32_t *pulGatewayAddress,
                                        uint32_t *pulDNSServerAddress );

Obtains the network address configuration from the TCP/IP stack.

Parameters:

*pulIPAddress Used to return the IP address being used by the IP stack.

The IP address is represented as a 32-bit number in network byte order.
**pulNetMask**  
Used to return the net mask being used by the IP stack.  
The net mask is represented as a 32-bit number in network byte order.

**pulGatewayAddress**  
Used to return the IP address of the gateway being used by the IP stack.  
The IP address is represented as a 32-bit number in network byte order.

**pulDNSServerAddress**  
Used to return the IP address of the DNS server being used by the IP stack.  
The IP address is represented as a 32-bit number in network byte order.

**Example usage:**

```c
/* OPENRTOS+TCP sockets include. */
#define "FreeRTOS_sockets.h"
void vApplicationIPNetworkEventHook( eIPCallbackEvent_t eNetworkEvent )
{
  uint32_t ulIPAddress, ulNetMask, ulGatewayAddress, ulDNSServerAddress;
  int8_t cBuffer[ 16 ];

  if( eNetworkEvent == eNetworkUp )
  {
    /* The network is up and configured. Print out the configuration obtained from the DHCP server. */
    FreeRTOS_GetAddressConfiguration( &ulIPAddress,
                                       &ulNetMask,
                                       &ulGatewayAddress,
                                       &ulDNSServerAddress );

    /* Convert the IP address to a string then print it out. */
    FreeRTOS_inet_ntoa( ulIPAddress, cBuffer );
    printf( "IP Address: %s\r\n", cBuffer );

    /* Convert the net mask to a string then print it out. */
    FreeRTOS_inet_ntoa( ulNetMask, cBuffer );
    printf( "Subnet Mask: %s\r\n", cBuffer );
```
/* Convert the IP address of the gateway to a string then print it out. */
FreeRTOS_inet_ntoa( ulGatewayAddress, cBuffer );
printf( "Gateway IP Address: %s\r\n", cBuffer );

/* Convert the IP address of the DNS server to a string then print it out. */
FreeRTOS_inet_ntoa( ulDNSServerAddress, cBuffer );
printf( "DNS server IP Address: %s\r\n", cBuffer );

Figure 3-14 Example use of the FreeRTOS_GetAddressConfiguration() API function

3.3.6 FreeRTOS_GetMACAddress()
FreeRTOS_IP.h
const uint8_t * FreeRTOS_GetMACAddress( void );

Returns:
Returns a pointer to the MAC address used by the NIC expressed as six separate bytes, with each byte in a consecutive memory location.

3.3.7 FreeRTOS_GetIPAddress()
FreeRTOS_IP.h
uint32_t FreeRTOS_GetIPAddress( void );

Returns:
Returns the IP address of the NIC in network byte order. FreeRTOS_inet_ntoa() can be used to convert the IP address into a more easily readable decimal dot notation ASCII string.

3.3.8 FreeRTOS_GetGatewayAddress()
FreeRTOS_IP.h
uint32_t FreeRTOS_GetGatewayAddress( void );

Returns:

Returns the IP address of the gateway in network byte order. FreeRTOS_inet_ntoa() can be used to convert the IP address into a more easily readable decimal dot notation ASCII string.

3.3.9 FreeRTOS_GetDNSServerAddress()

FreeRTOS_IP.h
uint32_t FreeRTOS_GetDNSServerAddress( void );

Returns:

Returns the IP address of the DNS server in network byte order. FreeRTOS_inet_ntoa() can be used to convert the IP address into a more easily readable decimal dot notation ASCII string.

3.3.10 FreeRTOS_GetNetmask()

FreeRTOS_IP.h
uint32_t FreeRTOS_GetNetmask( void );

Returns:

Returns the Net Mask in network byte order. FreeRTOS_inet_ntoa() can be used to convert the IP address into a more easily readable decimal dot notation ASCII string.

3.3.11 FreeRTOS_OutputARPRequest()

FreeRTOS_IP.h
void FreeRTOS_OutputARPRequest( uint32_t ulIPAddress );

Forces an ARP request to be sent for a given IP address.

Parameters:
**ullIPAddress**  The IP address for which an ARP request will be sent.

**Returns:**

void

### 3.3.12 FreeRTOS_IsNetworkUp()

**FreeRTOS_IP.h**

```c
BaseType_t FreeRTOS_IsNetworkUp( void );
```

Used to test if the network is currently up (connected) or down (disconnected). Note that disconnect events come from the network interface driver, so rely on the network interface driver for implementation.

**Returns:**

pdTRUE if the network is up (connected). Otherwise pdFALSE.

### 3.3.13 FreeRTOS_SignalSocket()

**FreeRTOS_sockets.h**

```c
BaseType_t FreeRTOS_SignalSocket( Socket_t xSocket );
```

```c
BaseType_t FreeRTOS_SignalSocketFromISR( Socket_t xSocket, BaseType_t pxHigherPriorityTaskWoken);
```

Used to send a signal to a socket, the result of which is that any task blocked on a read from the socket will leave the Blocked state (abort the blocking operation), with the task’s read operation (FreeRTOS_recv() or FreeRTOS_recvfrom()) returning -pdFREERTOS_ERRNO_EINTR.

If the socket is part of a socket set (SocketSet_t), the call to FreeRTOS_select() using that socket set, will also get interrupted and return -pdFREERTOS_ERRNO_EINTR.

FreeRTOS_SignalSocketFromISR() is a version of FreeRTOS_SignalSocket() that can be used from an interrupt service routine (ISR).

ipconfigSUPPORT_SIGNALS must be set to 1 in FreeRTOSIPConfig.h for FreeRTOS_SignalSocket() to be available.
Parameters:

xSocket

The socket to which the signal is being sent.

pxHigherPriorityTaskWoken

[FreeRTOS_SignalSocketFromISR() only.]

*pxHigherPriorityTaskWoken must be initialised to 0.

FreeRTOS_SignalSocketFromISR() will set
*pxHigherPriorityTaskWoken to pdTRUE if sending the signal to the
socket caused a task to unblock, and the unblocked task has a
priority higher than the currently running task.

If FreeRTOS_SignalSocket() sets this value to pdTRUE then a
context switch should be requested before the interrupt is exited.
The name of the macro used to request a context switch from an
ISR is dependent on the port, and will be called either
portYIELD_FROM_ISR() or portEND_SWITCHING_ISR(). Refer to
the documentation and examples provided for the port in use.

Returns:

If xSocket is not a valid socket then -pdFREERTOS_ERRNO_EINVAL is returned. Otherwise 0 is returned.

3.4  Event hook functions

3.4.1  vApplicationIPNetworkEventHook()

FreeRTOS_sockets.h

void vApplicationIPNetworkEventHook( eIPCallbackEvent_t eNetworkEvent );

vApplicationIPNetworkEventHook() is an application defined hook (or callback) function that is called
by the TCP/IP stack when the network either connects or disconnects. As the function is called by
the TCP/IP stack the TCP/IP sets sets the value of the function's parameter.
Callback functions are implemented by the application writer, but called by the TCP/IP stack. The prototype of the callback function must exactly match the prototype above (including the function name).

The value of the eNetworkEvent parameter will equal eNetworkUp if the IP stack called vApplicationIPNetworkEventHook() because the network connected:

- If ipconfigUSE_DHCP server is set to 1 in FreeRTOSIPConfig.h then vApplicationIPNetworkEventHook( eNetworkUp ) is called when an IP address is obtained from a DHCP server and when the lease for an IP address previously obtained from a DHCP is renewed.
- If ipconfigUSE_DHCP server is set to 0 in FreeRTOSIPConfig.h then vApplicationIPNetworkEventHook( eNetworkUp ) is called when the network has been initialised with a static IP address.

The value of the eNetworkEvent parameter will equal eNetworkDown if the IP stack called vApplicationIPNetworkEventHook() because the network disconnected:

- The TCP/IP stack calls vApplicationIPNetworkEventHook( eNetworkDown ) when it is informed by the network driver (the interface to the Ethernet peripheral) that network connectivity has been lost. Not all drivers will implement this functionality.

The application will only call vApplicationIPNetworkEventHook() if ipconfigUSENETWORK_EVENT_HOOK is set to 1 in FreeRTOSIPConfig.h.

The network event hook is a good place to create tasks that use the IP stack as it ensures the tasks are not created until the TCP/IP stack is ready.

**Example usage:**

```c
/* Defined by the application code, but called by OPENRTOS+TCP when the network
connects/disconnects (if ipconfigUSE_NETWORK_EVENT_HOOK is set to 1 in
FreeRTOSIPConfig.h). */
void vApplicationIPNetworkEventHook( eIPCallbackEvent_t eNetworkEvent )
```
uint32_t ulIPAddress, ulNetMask, ulGatewayAddress, ulDNSServerAddress;
static BaseType_t xTasksAlreadyCreated = pdFALSE;
int8_t cBuffer[ 16 ];

/* Check this was a network up event, as opposed to a network down event. */
if( eNetworkEvent == eNetworkUp )
{
    /* Create the tasks that use the TCP/IP stack if they have not already been created. */
    if( xTasksAlreadyCreated == pdFALSE )
    {
        /*
          * Create the tasks here.
          */

        xTasksAlreadyCreated = pdTRUE;
    }
}

/* The network is up and configured. Print out the configuration, which may have been obtained from a DHCP server. */
FreeRTOS_GetAddressConfiguration( &ulIPAddress, &ulNetMask, 
    &ulGatewayAddress, 
    &ulDNSServerAddress );

/* Convert the IP address to a string then print it out. */
FreeRTOS_inet_ntoa( ulIPAddress, cBuffer );
printf( "IP Address: %s\n", cBuffer );

/* Convert the net mask to a string then print it out. */
FreeRTOS_inet_ntoa( ulNetMask, cBuffer );
printf( "Subnet Mask: %s\n", cBuffer );
/* Convert the IP address of the gateway to a string then print it out. */
FreeRTOS_inet_ntoa( ulGatewayAddress, cBuffer );
printf( "Gateway IP Address: %s\r\n", cBuffer );

/* Convert the IP address of the DNS server to a string then print it out. */
FreeRTOS_inet_ntoa( ulDNSServerAddress, cBuffer );
printf( "DNS server IP Address: %s\r\n", cBuffer );

3.4.2 vApplicationPingReplyHook()

FreeRTOS_sockets.h
void vApplicationPingReplyHook( ePingReplyStatus_t eStatus, uint16_t usIdentifier );

vApplicationPingReplyHook() is an application defined hook (or callback) function that is called by the TCP/IP stack when the stack receives a reply to an ICMP echo (ping) request that was generated using the FreeRTOS_SendPingRequest() function.

Callback functions are implemented by the application writer, but called by the TCP/IP stack. The prototype of the callback function must exactly match the prototype above (including the function name).

Parameters:

**eStatus** eStatus will be set (by the TCP/IP stack) to one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
eSuccess The echo reply was received correctly.

eInvalidChecksum The data received in the echo reply matched that sent in the echo request, but the reply had an incorrect checksum.

eInvalidData The data received in the echo reply did not match that sent in the echo request.

usIdentifier The identifier received in the echo reply.

Each echo request has a unique identifier to allow replies to be matched to the requests. The FreeRTOS_SendPingRequest() function returns the identifier of the outgoing echo request it generated.

Example usage:

The example on the FreeRTOS_SendPingRequest() documentation section includes an example implementation of vApplicationPingReplyHook().
4. Porting OPENRTOS+TCP

The majority of the OPENRTOS+TCP source code is independent of the compiler used to build the code, and the microcontroller on which the code runs. Changing C compilers is very straightforward. There is obviously a hardware dependency in the Ethernet MAC driver, but even so, changing to a microcontroller that has a different Ethernet MAC interface is still relatively straightforward.

This section describes how to:

- Using a Different Compiler
- Creating a Simple New Embedded Ethernet Driver
- Creating a New Zero Copy Embedded Ethernet Driver

4.1 Porting OPENRTOS+TCP to a New Embedded C Compiler

4.1.1 Summary Bullet Points

- The OPENRTOS+TCP source code has two compiler dependencies:
  1. How the compiler is told to pack C structures (that is, not leave any alignment bytes between the members of the C structure). Packing structures is very important to ensure data appears correctly in the TCP, UDP, IP and Ethernet frames.
  2. How the compiler and MCU RTOS port define an interrupt service routine (ISR). The embedded IP stack only uses interrupts to manage the Ethernet driver, and so this topic is not covered here.
- The syntax used to ensure C structures are packed correctly is placed in a compiler specific header file to ensure the non-portable code is not included in the core embedded IP stack source code.
- Two header files are required for each embedded C compiler.
  1. `pack_struct_start.h` contains the keywords required to appear before the definition of a C structure that must be packed.
  2. `pack_struct_end.h` contains the keywords required to appear after the definition of a C structure that must be packed.
- Compiler specific header files that ship with the embedded TCP/IP stack are located in the FreeRTOS-Plus/FreeRTOS-Plus-TCP/portable/Compiler directory tree.
4.1.2 Detailed Explanation and Example

The OPENRTOS+TCP implementation uses C structures to map onto the TCP, IP, ARP, DNS, DHCP, etc. protocol headers within the Ethernet frames. C compilers normally lay out the memory used to hold a C structure such that access to the C structure's members is optimised for the MCU architecture on which the embedded software is running. This means empty padding bytes are placed between C structure members to achieve a byte alignment that is natural for the MCU. For example, consider the following structure defined on a 32-bit MCU:

```
struct a_struct
{
   uint8_t Member1;
   uint8_t Member2;
   uint8_t Member3;
   uint32_t Member4;
}
```

Figure 4-1 Example C structure definition

The first three members are 8 bits each, and will most likely appear in three consecutive bytes of the same 32-bit word. The forth member is 32-bits, and can be accessed most efficiently (dependent on the MCU, but in most cases) if it is placed on a 32-bit (four byte) boundary. Therefore, without the structure being packets, the compiler will most likely add a 'padding' byte between Member3 and Member4 to move the start address of Member4 up one byte and so onto a 32-bit boundary. The structure will then appear in memory as follows:

```
struct a_struct
{
   uint8_t Member1;   At address 0 (first byte of first 32-bit word)
   uint8_t Member2;   At address 1 (second byte of first 32-bit word)
```

uint8_t Member3;  // At address 2 (third byte of first 32-bit word)

Padding byte  // At address 4 (forth byte of the first 32-bit word)

uint32_t Member4;  // Now at address 4, the first byte of the second 32-bit word

---

Figure 4-2 Example of how the C structure will appear in the MCU memory

However, the TCP/IP protocol headers do not have padding bytes, so the compiler must be instructed not to add them additional bytes into structures that map onto the IP protocol headers that a written to or read from Ethernet frames. Structures that do not contain padding bytes are said to be 'packed'. The syntax required to ensure structures are packed depends on the embedded C compiler. The OPENRTOS+TCP implementation cannot use any C compiler specific syntax in the common (not MCU port specific) files, and instead allows users to define their own packing directives in two very simple header files that are then included from the C files.

Structures that require packing appear in the OPENRTOS+TCP code as follows:

```c
#include "pack_struct_start.h"

struct a_struct
{
    uint8_t Member1;
    uint8_t Member2;
    uint8_t Member3;
    uint32_t Member4;
}

#include "pack_struct_end.h"
```

Figure 4-3 A structure wrapped in pack_struct_start.h and pack_struct_end header file inclusions

A version of `pack_struct_start.h` and `pack_struct_end.h` that contains compiler specific syntax can then be provided for each compiler.
For example, the directory FreeRTOS-Plus/FreeRTOS-Plus-UDP/portable/Compiler/GCC contains definitions suitable for use with the GCC compiler. pack_struct_start.h is empty because GCC does not require any special syntax at the start of the structure. pack_struct_end.h contains the following single line of code:

```c
__attribute__((packed));
```

So, after pre-processing the C source code appears to the compiler as shown below, which is valid GCC syntax:

```c
struct a_struct {
    uint8_t Member1;
    uint8_t Member2;
    uint8_t Member3;
    uint32_t Member4;
} __attribute__((packed));
```

Figure 4-4 How GCC sees a packed structure after pack_struct_start.h and pack_struct_end.h have been included

pack_struct_end.h must, as a minimum, contain at least a semicolon (;) to mark the end of the structure definition. It is sometimes valid for pack_struct_start.h to be empty, but it is never valid for pack_struct_end.h to be completely empty.

4.2 Porting OPENRTOS+TCP to a Different Microcontroller

4.2.1 Introduction

The implementation of the network interface, and in particular the Ethernet MAC driver, are crucial to the data throughput that can be achieved when using the embedded TCP/IP stack. For high throughput the MAC driver must make efficient use of the DMA and avoid copying data where possible. End to end zero copy is possible with OPENRTOS+TCP for UDP packets, and an advanced interface exists that also allows zero copy for TCP packets. There are also advanced options available that allow packets to be filtered before they are even sent to the embedded TCP/IP stack, and packets that are received in quick successions can be sent to the embedded TCP/IP stack in one go rather than individually.
However, few applications actually require throughput to be maximised, especially on small MCUs, and the implementer may instead opt to sacrifice throughput in favour of simplicity.

This section describes how to interface OPENRTOS+TCP with a network driver, and provides an outline example of both a simple and a faster (but more complex) interface. **It is very important to refer to these examples as they demonstrate how network buffers are freed after data has been transmitted.**

The network driver port layers that ship with OPENRTOS+TCP are located in the FreeRTOS-Plus-TCP/portable/NetworkInterface directory of the OPENRTOS+TCP download. Note however that these drivers have been created in order to allow testing of the embedded TCP/IP stack, and are not intended to represent optimised examples.
4.2.2 Summary Bullet Points

- Each MCU to which OPENRTOS+TCP is ported requires an Ethernet MAC driver. It is assumed this already exists and is known to work.
- OPENRTOS+TCP is ported to new hardware by providing a 'network interface port layer' that provides the interface between the embedded TCP/IP stack and the Ethernet MAC driver. See the image on the right.
- The network interface port layer must provide a function called xNetworkInterfaceInitialise() that initialises the MAC driver.
- The network interface port layer must provide a function called xNetworkInterfaceOutput() that sends data received from the embedded TCP/IP stack to the Ethernet MAC driver for transmission.
- The network interface port layer must send packets received from the Ethernet MAC driver to the TCP/IP stack by calling xSendEventStructToIPTask().

Figure 4-5 The network interface port layer sits between the IP stack and the embedded Ethernet hardware drivers
• Only if BufferAllocation_1.c is used for buffer allocation, the network interface port layer must statically allocate network buffers and provide a function called vNetworkInterfaceAllocateRAMToBuffers() to assign the statically allocated network buffers to network buffer descriptors.
• Network buffers (the buffer in which the actual data is stored) are referenced using xNetworkBufferDescriptor_t structures.
• The embedded TCP/IP stack provides a set of porting utility functions to allow the port layer to perform actions such as obtaining and freeing network buffers.

This section provides more information on each of these steps, and provides two examples. The first example demonstrates how to implement a simple (but slower) driver. The second example demonstrates how to implement a more sophisticated (and faster) driver. It is very important to refer to these examples as they demonstrate how network buffers are freed after data has been transmitted.

4.2.3 Network Buffers and Network Buffer Descriptors

Ethernet (or other network) frames are stored in network buffers. A network buffer descriptor (a variable of type xNetworkBufferDescriptor_t) is used to describe a network buffer, and pass network buffers between the TCP/IP stack and the network drivers.

```c
typedef struct xNETWORK_BUFFER
{
    size_t xDataLength;
    uint8_t *pucEthernetBuffer;
} xNetworkBufferDescriptor_t;
```

Figure 4-6 pucEthernetBuffer points to the start of the network buffer. xDataLength holds the size of the buffer in bytes, excluding the Ethernet CRC bytes.

Only the following two members of the xNetworkBufferDescriptor_t structure should be accessed:

1. uint8_t *pucEthernetBuffer;
pucEthernetBuffer points to the start of the network buffer.

2. `size_t xDataLength`

   `xDataLength` holds the size of the network buffer pointed to by `pucEthernetBuffer`. The size is specified in bytes but the length excludes the bytes that hold the Ethernet frame's CRC byte.

   `pucGetNetworkBuffer()` is used to obtain just the network buffer itself, and is normally only used in zero copy drivers.

   `pxGetNetworkBufferWithDescriptor()` is used to obtain both a network buffer and a network buffer descriptor at the same time.

### 4.2.4 Function That Must be Implemented by the Port Layer

#### 4.2.4.1 `xNetworkInterfaceInitialise()`

`xNetworkInterfaceInitialise()` must prepare the Ethernet MAC to send and receive data. In most cases this will just involve calling whichever initialise function is provided with the Ethernet MAC peripheral drivers - which will in turn ensure the MAC hardware is enabled and clocked, as well as configure the MAC peripheral's DMA descriptors.

`xNetworkInterfaceInitialise()` does not take any parameters, returns `pdPASS` if the initialisation was successful, and returns `pdFAIL` if the initialisation fails.

```c
BaseType_t xNetworkInterfaceInitialise( void );
```

The `xNetworkInterfaceInitialise()` function prototype

#### 4.2.4.2 `xNetworkInterfaceOutput()`

The TCP/IP stack calls `xNetworkInterfaceOutput()` whenever a network buffer is ready to be transmitted.

The buffer to transmit is described by the descriptor passed into the function using the function's single parameter. If `xReleaseAfterSend` does not equal `pdFALSE` then both the buffer and the buffer's descriptor must be released (returned) back to the embedded TCP/IP stack when they are no longer required.
xNetworkInterfaceOutput() must return pdPASS if the network buffer was sent to the network successfully, or pdFAIL if the network buffer could not be transmitted.

Basic and more advanced examples are provided below, and the FreeRTOS-Plus-TCP/portable/NetworkInterface directory of the OPENRTOS+TCP download contained examples that can be referenced. Note however that the examples in the download may not be optimised.

```c
BaseType_t xNetworkInterfaceOutput( xNetworkBufferDescriptor_t * const pxDescriptor,
                                  BaseType_t xReleaseAfterSend );
```

The xNetworkInterfaceOutput() function prototype

4.2.5 vNetworkInterfaceAllocateRAMToBuffers() only when BufferAllocation_1.c is used

BufferAllocation_1.c uses pre-allocated network buffers that are normally statically allocated at compile time.

The number of network buffers that must be allocated is set by the ipconfigNUM_NETWORK_BUFFER_DESCRPTORS definition in FreeRTOSIPConfig.h, and the size of each buffer must be (ipTOTAL_ETHERNET_FRAME_SIZE + ipBUFFER_PADDING). ipTOTAL_ETHERNET_FRAME_SIZE is calculated automatically from the value of ipconfigNETWORK_MTU.

Networking hardware can impose strict alignment requirements on the allocated buffers, so it is recommended that the buffers are allocated in the embedded Ethernet driver itself - that way the buffer's alignment will always match the hardware.

The embedded TCP/IP stack allocates the network buffer descriptors, but does not know anything about the alignment of the network buffers themselves. Therefore the embedded Ethernet driver must also provide a function called vNetworkInterfaceAllocateRAMToBuffers() that allocates a statically declared buffer to each descriptor. Note that ipBUFFER_PADDING bytes at the beginning of the buffer are left unused. See the example below.

```c
void vNetworkInterfaceAllocateRAMToBuffers{
```
xNetworkBufferDescriptor_t xDescriptors[ ipconfigNUM_NETWORK_BUFFERS ];

The vNetworkInterfaceAllocateRAMToBuffers() function prototype

/* First statically allocate the buffers, ensuring an additional ipBUFFER_PADDING bytes are allocated to each buffer. This example makes no effort to align the start of the buffers, but most hardware will have an alignment requirement. If an alignment is required then the size of each buffer must be adjusted to ensure it also ends on an alignment boundary. Below shows an example assuming the buffers must also end on an 8-byte boundary. */
#define BUFFER_SIZE ( ipTOTAL_ETHERNET_FRAME_SIZE + ipBUFFER_PADDING )
#define BUFFER_SIZE_ROUNDED_UP ( ( BUFFER_SIZE + 7 ) & ~0x07UL )
static uint8_t ucBuffers[ ipconfigNUM_NETWORK_BUFFERS ][ BUFFER_SIZE_ROUNDED_UP ];

/* Next provide the vNetworkInterfaceAllocateRAMToBuffers() function, which simply fills in the pucEthernetBuffer member of each descriptor. */
void vNetworkInterfaceAllocateRAMToBuffers(
xNetworkBufferDescriptor_t pxNetworkBuffers[ ipconfigNUM_NETWORK_BUFFERS ] )
{
  BaseType_t x;

  for( x = 0; x < ipconfigNUM_NETWORK_BUFFERS; x++ )
  {
    /* pucEthernetBuffer is set to point ipBUFFER_PADDING bytes in from the beginning of the allocated buffer. */
    pxNetworkBuffers[ x ].pucEthernetBuffer = &( ucBuffers[ x ][ ipBUFFER_PADDING ] );

    /* The following line is also required, but will not be required in future versions. */
    *( ( uint32_t * ) &ucBuffers[ x ][ 0 ] ) = ( uint32_t ) &pxNetworkBuffers[ x ];
  }

4.2.6 Functions Provided by the TCP/IP Stack For Use By The Port Layer

The port layer can use the following function:

- **pxGetNetworkBufferWithDescriptor()** -
  Obtains both a network buffer and a descriptor that describes the network buffer. This function can also be used to obtain just a network buffer descriptor - which can be useful when implementing zero copy drivers.

- **vReleaseNetworkBufferAndDescriptor()** -
  Releases (returns to the embedded TCP/IP stack) a network buffer descriptor, and the network buffer referenced by the descriptor (if any).

- **pxNetworkBufferGetFromISR()**
  [not available when BufferAllocation_2.c is used]

- **vNetworkBufferReleaseFromISR()**
  [not available when BufferAllocation_2.c is used]

- **eConsiderFrameForProcessing()** -
  Used to determine if data received from the network needs to be passed to the embedded TCP/IP stack. Ideally this function would be called from the network interrupt to allow received packets to be discarded at the earliest possible opportunity.

- **xSendEventStructToIPTask()** -
  xSendEventStructToIPTask() is a function used by the embedded TCP/IP stack itself to send various events to the RTOS task that is running the embedded TCP/IP stack. The port layer uses the function with eNetworkRxEvent events to pass received data into the stack for processing.

- **pucGetNetworkBuffer()** -
  Obtains just a network buffer, without a network buffer descriptor. This function is normally only used in zero copy interfaces to allocate buffers to DMA descriptors.

- **vReleaseNetworkBuffer()** -
  Releases (returns to the embedded TCP/IP stack) a network buffer by itself - without a network buffer descriptor. This function is normally only used in zero copy interfaces where network buffers were allocated to DMA descriptors.
4.2.6.1 Receiving Data

The Ethernet MAC driver will place received Ethernet frames into a buffer. The port layer has:

1. Determine if the received data needs to be sent to the embedded TCP/IP stack. Ideally this would be done in the receive interrupt itself to allow unnecessary packets to be dropped at the earliest possible time.
2. Allocate a network buffer descriptor.
3. Set the xDataLength and pucEthernetBuffer members of the allocated descriptor accordingly (see both the basic and zero copy examples at the bottom of this section).
4. Call xSendEventStructToIPTask() to send the network buffer descriptor into the embedded TCP/IP stack for processing (see both the basic and zero copy examples at the bottom of this section).

```c
typedef struct IP_TASK_COMMANDS
{
    /* Specifies the type of event being sent to the RTOS task and must be set to eNetworkRxEvent to signify a receive event. */
    eIPEvent_t eEventType;

    /* Points to additional data about the event. In this case set pvData to the address of the network buffer descriptor. */
    void *pvData;
} xIPStackEvent_t;
```

Figure 4-8 The xIPStackEvent_t type

```c
/* The timeout is specified in RTOS ticks. Returns pdTRUE if the message was sent successfully, otherwise return pdFALSE. */
BaseType_t xSendEventStructToIPTask( const xIPStackEvent_t *pxEvent, TickType_t xTimeout )
```
Basic and more advanced examples are provided below. The network driver port layers that ship with OPENRTOS+TCP (which are not necessarily optimised) can be found in the FreeRTOS-Plus-TCP/portable/NetworkInterface directory.

4.2.7 Network Interface Port Layer Examples

4.2.7.1 Example of a Basic Network Interface Port Layer

Simple network interfaces can be created by copying Ethernet frames between buffers allocated by the Ethernet MAC driver libraries and buffers allocated by the port layer. [A more efficient zero copy alternative is provided after the simple example.

4.2.7.2 Example implementation of xNetworkInterfaceInitialise() for a basic port layer

```c
BaseType_t xNetworkInterfaceInitialise( void )
{
    BaseType_t xReturn;

    /*
     * Perform the hardware specific network initialisation here. Typically
     * that will involve using the Ethernet driver library to initialise the
     * Ethernet (or other network) hardware, initialise DMA descriptors, and
     * perform a PHY auto-negotiation to obtain a network link.
     * 
     * This example assumes InitialiseNetwork() is an Ethernet peripheral driver
     * library function that returns 0 if the initialisation fails.
     */
    if( InitialiseNetwork() == 0 )
    {
        xReturn = pdFAIL;
    }
    else
    {
        xReturn = pdPASS;
    }
}  
```
Figure 4-10 xNetworkInterfaceInitialise() is hardware specific, therefore this example describes what needs to be done without showing any detail.

4.2.7.3 Example implementation of xNetworkInterfaceOutput() for a basic port layer

```c
BaseType_t xNetworkInterfaceOutput( xNetworkBufferDescriptor_t * const pxDescriptor,
                                   BaseType_t xReleaseAfterSend )
{
    /* Simple network interfaces (as opposed to more efficient zero copy network interfaces) just use Ethernet peripheral driver library functions to copy data from the OPENRTOS+TCP buffer into the peripheral driver's own buffer. This example assumes SendData() is a peripheral driver library function that takes a pointer to the start of the data to be sent and the length of the data to be sent as two separate parameters. The start of the data is located by pxDescriptor->pucEthernetBuffer. The length of the data is located by pxDescriptor->xDataLength. */
    SendData( pxDescriptor->pucBuffer, pxDescriptor->xDataLength );

    /* Call the standard trace macro to log the send event. */
    iptraceNETWORK_INTERFACE_TRANSMIT();

    if( xReleaseAfterSend != pdFALSE )
    {
        /* It is assumed SendData() copies the data out of the OPENRTOS+TCP Ethernet buffer. The Ethernet buffer is therefore no longer needed, and must be freed for re-use. */
        vReleaseNetworkBufferAndDescriptor( pxDescriptor );
    }
}
```
Figure 4-11 Example implementation of xNetworkInterfaceOutput() suitable for a simple (rather than zero copy) network interface implementation

4.2.7.4 Example of passing received data to the TCP/IP in a basic port layer

When a packet is received from the Ethernet (or other network) driver the port layer must use xNetworkBufferDescriptor_t structure to describe the packet, then call xSendEventStructToIPTask() to send the xNetworkBufferDescriptor_t structure to the embedded TCP/IP stack.

NOTE 1: If BufferAllocation_2.c is used then network buffer descriptors and Ethernet buffers cannot be allocated from inside an interrupt service routine (ISR). In this case the Ethernet MAC receive interrupt can defer the receive processing to a task. This is demonstrated below.

NOTE 2: There are numerous advanced techniques that can be employed to minimise the amount of data sent from the port layer into the embedded TCP/IP stack. For example, eConsiderFrameForProcessing() can be called to determine if the received Ethernet frame needs to be sent to the embedded TCP/IP stack at all, and Ethernet frames that are received in quick succession can be sent to the embedded TCP/IP stack in one go.

/* The deferred interrupt handler is a standard RTOS task. FreeRTOS's centralised deferred interrupt handling capabilities can also be used. */
static void prvEMACDeferredInterruptHandlerTask( void *pvParameters )
{
    xNetworkBufferDescriptor_t *pxBufferDescriptor;
    size_t xBytesReceived;
    /* Used to indicate that xSendEventStructToIPTask() is being called because of an Ethernet receive event. */
    xIPStackEvent_t xRxEvent;

    for( ;; )
    {

return pdTRUE;
}
/* Wait for the Ethernet MAC interrupt to indicate that another packet has been received. It is assumed xEMACRxEventSemaphore is a counting semaphore (to count the Rx events) and that the semaphore has already been created (remember this is an example of a simple rather than optimised port layer). */
xSemaphoreTake( xEMACRxEventSemaphore, portMAX_DELAY );

/* See how much data was received. Here it is assumed ReceiveSize() is a peripheral driver function that returns the number of bytes in the received Ethernet frame. */
xBytesReceived = ReceiveSize();

if( xBytesReceived > 0 )
{
    /* Allocate a network buffer descriptor that points to a buffer large enough to hold the received frame. As this is the simple rather than efficient example the received data will just be copied into this buffer. */
    pxBufferDescriptor = pxGetNetworkBufferWithDescriptor( xBytesReceived, 0 );

    if( pxBufferDescriptor != NULL )
    {
        /* pxBufferDescriptor->pucEthernetBuffer now points to an Ethernet buffer large enough to hold the received data. Copy the received data into pxBufferDescriptor->pucEthernetBuffer. Here it is assumed ReceiveData() is a peripheral driver function that copies the received data into a buffer passed in as the function's parameter. Remember! While is is a simple robust technique - it is not efficient. An example that uses a zero copy technique is provided further down this section. */
        ReceiveData( pxBufferDescriptor->pucEthernetBuffer );
        pxBufferDescriptor->xDataLength = xBytesReceived;
/* See if the data contained in the received Ethernet frame needs to be processed.  NOTE! It is preferable to do this in the interrupt service routine itself, which would remove the need to unblock this task for packets that don't need processing. */
if( eConsiderFrameForProcessing( pxBufferDescriptor->pucEthernetBuffer == eProcessBuffer )
{
    /* The event about to be sent to the TCP/IP is an Rx event. */
    xRxEvent.eEventType = eNetworkRxEvent;

    /* pvData is used to point to the network buffer descriptor that now references the received data. */
    xRxEvent.pvData = ( void * ) pxBufferDescriptor;

    /* Send the data to the TCP/IP stack. */
    if( xSendEventStructToIPTask( &xRxEvent, 0 ) == pdFALSE )
    {
        /* The buffer could not be sent to the IP task so the buffer must be released. */
        vReleaseNetworkBufferAndDescriptor( pxBufferDescriptor );

        /* Make a call to the standard trace macro to log the occurrence. */
        iptraceETHERNET_RX_EVENT_LOST();
    }
    else
    {
        /* The message was successfully sent to the TCP/IP stack. Call the standard trace macro to log the occurrence. */
        iptraceNETWORK_INTERFACE_RECEIVE();
Figure 4-12 An example of a simple (rather than more efficient zero copy) receive handler

4.2.7.5 Example of a More Efficient Network Interface Port Layer

It is intended that this section is read after the section that describes how to create a simple network interface port layer.

Simple network interfaces copy Ethernet frames between buffers used and managed by the TCP/IP stack and buffers used and managed by the Ethernet (or other network) MAC drivers. Copying data between buffers makes the driver's implementation simple, but is inefficient.

Zero copy network interfaces do not copy data between buffers, but instead pass references to buffers between the TCP/IP stack and the Ethernet MAC drivers.

Zero copy interfaces are more complex, and can rarely be created without editing the Ethernet MAC drivers themselves.
Most Ethernet hardware will use DMA (Direct Memory Access) to move frames between the Ethernet hardware and pre-allocated RAM buffers. Normally the pre-allocated memory buffers are referenced using a set of DMA descriptors. DMA descriptors are normally chained - each descriptor points to the next in the chain, with the last in the chain pointing back to the first.

![Diagram of DMA Descriptors and Memory Buffers]

**Figure 4-13 Chained DMA descriptors**

**4.2.7.6 Example implementation of xNetworkInterfaceInitialise() for a zero copy port layer**

`xNetworkInterfaceInitialise()` must use `pucGetNetworkBuffer()` to obtain the pointers to which the receive DMA descriptors point. It is not necessary to allocate any buffers for the transmit DMA descriptors - the buffers will be passed in (by reference) as data becomes available to send.
The DMA Rx descriptors are initialised to point to buffers that were allocated by `pucGetNetworkBuffer()`. The DMA Tx descriptors do not point to any buffers after they have been initialised.

### 4.2.7.7 Example implementation of `xNetworkInterfaceOutput()` for a zero copy layer

`xNetworkInterfaceOutput()` does not copy the frame being transmitted to a buffer that is being managed by the MAC driver (it can't because the DMA's Tx descriptors are not pointing to any buffers) but instead updates the next DMA Tx descriptor so the descriptor points to the buffer that already contains the data.

**NOTE:** The Ethernet buffer must be released after the data it contains has been transmitted. If `BufferAllocation_2.c` is used the Ethernet buffer cannot be released from the Ethernet Transmit End interrupt, so must be released by the `xNetworkInterfaceOutput()` function the next time the same DMA descriptor is used. Often only one or two descriptors are used for transmitting data anyway, so this does not waste too much RAM.

```c
BaseType_t xNetworkInterfaceOutput( xNetworkBufferDescriptor_t * const pxDescriptor, BaseType_t xReleaseAfterSend ) {
    DMADescriptor_t *pxDMATxDescriptor;

    /* This example assumes GetNextTxDescriptor() is an Ethernet MAC driver library function that returns a pointer to a DMA descriptor of type DMA Descriptor_t. */
```
pxDMATxDescriptor = GetNextTxDescriptor();

/* Further, this example assumes the DMADescriptor_t type has a member called pucEthernetBuffer that points to the buffer the DMA will transmit, and a member called xDataLength that holds the length of the data the DMA will transmit. If BufferAllocation_2.c is being used then the DMA descriptor may still be pointing to the buffer it last transmitted. If this is the case then the old buffer must be released (returned to the TCP/IP stack) before descriptor is updated to point to the new data waiting to be transmitted. */

if( pxDMATxDescriptor->pucEthernetBuffer != NULL )
{
    /* Note this is releasing just an Ethernet buffer, not a network buffer descriptor as the descriptor has already been released. */
    vReleaseNetworkBuffer( pxDMATxDescriptor->pucEthernetBuffer );
}

/* Configure the DMA descriptor to send the data referenced by the network buffer descriptor. This example assumes SendData() is an Ethernet peripheral driver function. */

pxDMATxDescriptor->pucEthernetBuffer = pxDescriptor->pucEthernetBuffer;
pxDMATxDescriptor->xDataLength = pxDescriptor->xDataLength;
SendData( pxDMATxDescriptor );

/* Call the standard trace macro to log the send event. */

iptraceNETWORK_INTERFACE_TRANSMIT();

/* The network buffer descriptor must now be returned to the TCP/IP stack, but the Ethernet buffer referenced by the network buffer descriptor is still in use by the DMA. Remove the reference to the Ethernet buffer from the network buffer descriptor so releasing the network buffer descriptor does not result in the Ethernet buffer also being released. */

if( xReleaseAfterSend != pdFALSE )
{
pxDescriptor->pucEthernetBuffer = NULL;
vReleaseNetworkBufferAndDescriptor( pxDescriptor );
}

return pdTRUE;
}

Figure 4-15 An example zero copy implementation of xNetworkInterfaceOutput()

4.2.7.8 Receiving Data Using Zero-Copy

The receive DMA will place received frames into the buffer pointed to by the the receive DMA descriptor. The buffer was allocated using a call to pucGetNetworkBuffer(), which allows it to be referenced from a network buffer descriptor, and therefore passed by reference directly into the TCP/IP stack. A new network buffer is then allocated, and the receive DMA descriptor is updated to point to the new buffer.

All the notes regarding the implementation of the simple receive handler (including advanced features to improve efficiency) apply to the zero copy receive handler and are not repeated here.

/* The deferred interrupt handler is a standard RTOS task. FreeRTOS’s centralised deferred interrupt handling capabilities can also be used - however for additional speed use BufferAllocation_1.c to perform the entire operation in the interrupt handler. */
static void prvEMACDeferredInterruptHandlerTask( void *pvParameters )
{
    xNetworkBufferDescriptor_t *pxDescriptor;
    size_t xBytesReceived;
    DMADescriptor_t *pxDMARxDescriptor;
    uint8_t *pucTemp;
    /* Used to indicate that xSendEventStructToIPTask() is being called because of an Ethernet receive event. */
    xIPStackEvent_t xRxEvent;

    for( ;; )
    { /* The deferred interrupt handler is a standard RTOS task. FreeRTOS's centralised deferred interrupt handling capabilities can also be used - however for additional speed use BufferAllocation_1.c to perform the entire operation in the interrupt handler. */
        xIPStackEvent_t xRxEvent;

        for( ;; )
{ /* Wait for the Ethernet MAC interrupt to indicate that another packet has been received. */
xSemaphoreTake( xEMACRxEventSemaphore, portMAX_DELAY );

/* This example assumes GetNextRxDescriptor() is an Ethernet MAC driver library function that returns a pointer to the DMA descriptor (of type DMADescriptor_t again) that references the Ethernet buffer containing the received data. */
pxDMARxDescriptor = GetNextRxDescriptor();

/* Allocate a new network buffer descriptor that references an Ethernet frame large enough to hold the maximum network packet size (as defined in the FreeRTOSIPConfig.h header file. */
pxDescriptor = pxGetNetworkBufferWithDescriptor( ipTOTALETHERNETFRAME_SIZE, 0 );

/* Copy the pointer to the newly allocated Ethernet frame to a temporary variable. */
pucTemp = pxDescriptor->pucEthernetBuffer;

/* This example assumes that the DMADescriptor_t type has a member called pucEthernetBuffer that points to the Ethernet buffer containing the received data, and a member called xDataLength that holds the length of the received data. Update the newly allocated network buffer descriptor to point to the Ethernet buffer that contains the received data. */
pxDescriptor->pucEthernetBuffer = pxDMARxDescriptor->pucEthernetBuffer;
pxDescriptor->xDataLength = pxDMARxDescriptor->xDataLength;

/* Update the Ethernet Rx DMA descriptor to point to the newly allocated Ethernet buffer. */
pxDMARxDescriptor->puxEthernetBuffer = pucTemp;
/*
 * The network buffer descriptor now points to the Ethernet buffer that
 * contains the received data, and the Ethernet DMA descriptor now points
 * to a newly allocated (and empty) Ethernet buffer ready to receive more
 * data. No data was copied. Only pointers to data were swapped.
 *
 * THE REST OF THE RECEIVE HANDLER FUNCTION FOLLOWS THE EXAMPLE PROVIDED
 * FOR THE SIMPLE ETHERNET INTERFACE IMPLEMENTATION, whereby the network
 * buffer descriptor is sent to the TCP/IP on the network event queue.
 */

/* See if the data contained in the received Ethernet frame needs
 to be processed.  NOTE! It might be possible to do this in
 the interrupt service routine itself, which would remove the need
 to unblock this task for packets that don't need processing. */
if( eConsiderFrameForProcessing( pxDescriptor->pucEthernetBuffer )
    == eProcessBuffer )
{
    /* The event about to be sent to the TCP/IP is an Rx event. */
    xRxEvent.eEventType = eNetworkRxEvent;

    /* pvData is used to point to the network buffer descriptor that
     references the received data. */
    xRxEvent.pvData = ( void * ) pxDescriptor;

    /* Send the data to the TCP/IP stack. */
    if( xSendEventStructToIPTask( &xRxEvent, 0 ) == pdFALSE )
    {
        /* The buffer could not be sent to the IP task so the buffer
         must be released. */
    }
vReleaseNetworkBufferAndDescriptor( pxDescriptor );

    /* Make a call to the standard trace macro to log the
     occurrence. */
    iptraceETHERNET_RX_EVENT_LOST();
}
else
{
    /* The message was successfully sent to the TCP/IP stack.
     Call the standard trace macro to log the occurrence. */
    iptraceNETWORK_INTERFACE_RECEIVE();
}
}
else
{
    /* The Ethernet frame can be dropped, but the Ethernet buffer
     must be released. */
    vReleaseNetworkBufferAndDescriptor( pxDescriptor );
}
}

Figure 4-16 An example of a zero copy receive handler function

4.3 FreeRTOS+TCP API

The following functions are provided for use by the Ethernet interface port layer.

- pxGetNetworkBufferWithDescriptor()
- vReleaseNetworkBufferAndDescriptor()
- pucGetNetworkBuffer()
• vReleaseNetworkBuffer()
• eConsiderFrameForProcessing()
• xSendEventStructToIPTask()

4.3.1 pxGetNetworkBufferWithDescriptor()

FreeRTOS_IP_Private.h
NetworkBufferManagement.h

NetworkBufferDescriptor_t
*pxGetNetworkBufferWithDescriptor( size_t RequestedSizeBytes,
                           TickType_t xBlockTimeTicks );

Data that is sent out to the network or received from the network is stored in a network buffer. A network buffer is basically just a block of RAM (actually an array of uint8_t in the source code).

The embedded TCP/IP stack needs to first locate the network buffers, and once located know how big the network buffers are. Network buffer descriptors are used for that purpose.

typedef struct xNETWORK_BUFFER
{
  size_t xDataLength;
  uint8_t *pucEthernetBuffer;
} xNetworkBufferDescriptor_t;

pucPayloadBuffer points to the start of the network buffer.

xDataLength holds the size of the buffer in bytes, excluding the Ethernet CRC bytes.

pxGetNetworkBufferWithDescriptor() Obtains a network buffer and associated network buffer descriptor. If xRequestedSizeBytes is 0 then a network buffer descriptor is obtained by itself - without a network buffer.

pxGetNetworkBufferWithDescriptor() must not be called from an interrupt service routine (ISR).

The total number of network buffer descriptors is set by ipconfigNUM_NETWORK_BUFFER_DESCRIPTORS.
**Parameters:**

- **xRequestedSizeBytes**  
  The size of the Ethernet buffer to obtain and reference from the returned network buffer descriptor. The size is specified in bytes.
  
  If `xRequestedSizeBytes` is zero then the returned network buffer descriptor will not reference an Ethernet buffer (the reference is set to NULL).

- **xBlockTimeTicks**  
  If a network buffer is not available then the calling RTOS task will be held in the Blocked state (so other tasks can execute) until either a network buffer becomes available or the specified block time expires.
  
  The block time is specified in RTOS ticks. To convert a time specified in milliseconds to a time specified in RTOS ticks divide the time specified in milliseconds by `portTICK_PERIOD_MS`.

**Returns:**

Successful calls return a pointer to the obtained network buffer descriptor. Unsuccessful calls return NULL.

### 4.3.2 vReleaseNetworkBufferAndDescriptor()

FreeRTOS_IP_Private.h  
NetworkBufferManagement.h  

```c
void vReleaseNetworkBufferAndDescriptor( NetworkBufferDescriptor_t * const pxNetworkBuffer );
```

Returns to the TCP/IP stack a network buffer descriptor that was previously obtained from the TCP/IP stack by a call to `pxGetNetworkBufferWithDescriptor()`.

If the network buffer descriptor references an Ethernet buffer then the Ethernet buffer is also returned.

`pxGetNetworkBufferWithDescriptor()` must not be called from an interrupt service routine (ISR).
4.3.3 pucGetNetworkBuffer()

FreeRTOS_IP_Private.h
NetworkBufferManagement.h

uint8_t *pucGetNetworkBuffer( size_t *pxRequestedSizeBytes );

Data that is sent out to the network or received from the network is stored in a network buffer. A network buffer is basically just a block of RAM (actually an array of uint8_t in the source code).

The embedded TCP/IP stack needs to first locate the network buffers, and once located know how big the network buffers are. Network buffer descriptors are used for that purpose.

Whereas pxGetNetworkBufferWithDescriptor() obtains a network buffer descriptor that can (optionally) reference an Ethernet buffer, pucGetNetworkBuffer() just obtains the Ethernet buffer itself and would normally only be used to allocate buffers to DMA descriptors in zero copy drivers.

pucGetNetworkBuffer() must not be called from an interrupt service routine (ISR).

Parameters:

pxRequestedSizeBytes The size of the Ethernet buffer to obtain. The size is specified in bytes.

Returns:

Successful calls return a pointer to the obtained Ethernet buffer. Unsuccessful calls return NULL.
4.3.4  vReleaseNetworkBuffer()
FreeRTOS_IP_Private.h
NetworkBufferManagement.h

void vReleaseNetworkBuffer( uint8_t *pucPayloadBuffer );

Returns to the TCP/IP stack an Ethernet buffer that was previously obtained from the TCP/IP stack.

vReleaseNetworkBuffer() would normally only be used by a zero copy driver to release buffers that were previously allocated to DMA descriptors. Normally network buffers are allocated and released along with descriptors using pxGetNetworkBufferWithDescriptor() and vReleaseNetworkBufferAndDescriptor() respectively.

vReleaseNetworkBuffer() must not be called from an interrupt service routine (ISR).

Parameters:

*pucPayloadBuffer  A pointer to the Ethernet buffer being released (returned to the TCP/IP stack).

4.3.5  eConsiderFrameForProcessing()
FreeRTOS_IP_Private.h

eFrameProcessingResult_t eConsiderFrameForProcessing( uint8_t *pucPayloadBuffer );

Examines a received Ethernet frame to determine if, taking into account the current state of the TCP/IP stack, the Ethernet frame should be processed or dropped.

If ipconfigETHERNET_DRIVER_FILTERS_FRAME_TYPES is set to 1 in FreeRTOSIPConfig.h then eConsiderFrameForProcessing() should be called by the network interface port layer to determine whether received Ethernet frames should be sent to the IP stack for processing. If ipconfigETHERNET_DRIVER_FILTERS_FRAME_TYPES is set to 0 in FreeRTOSIPConfig.h then the TCP/IP stack will itself call eConsiderFrameForProcessing(), but only after it has already received the Ethernet frame from the network interface port layer.
It might not be necessary to call `eConsiderFrameForProcessing()` if the embedded Ethernet peripheral hardware is itself configured to filter Ethernet frames.

**Parameters:**

`pucPayloadBuffer`  A pointer to the start of the Ethernet frame being inspected.

**Returns:**

eProcessBuffer is returned if the Ethernet frame needs processing. eReleaseBuffer is returned if the Ethernet frame can be dropped (in which case both the network buffer descriptor that references the Ethernet frame, and the Ethernet frame itself, must both be returned to the TCP/IP stack).

### 4.3.6  xSendEventStructToIPTask()

FreeRTOS_IP_Private.h

```c
BaseType_t xSendEventStructToIPTask( const IPStackEvent_t *pxEvent, TickType_t xTimeout );
```

`xSendEventStructToIPTask()` is used throughout the embedded TCP/IP stack's implementation to send various events to the RTOS task that is running the embedded TCP/IP stack. The function is made available to the network port layer so the network port layer can send receive events to the same RTOS task.

**Parameters:**

`pxEvent`  A pointer to a structure of type IPStackEvent_t.

`xTimeout`  The time, specified in RTOS ticks, to wait for the message to be sent to the RTOS task that is running the embedded TCP/IP stack if the message cannot be sent immediately. The message might not be able to be sent immediately if the network event queue is full.

**Returns:**

If the event was successfully sent to the RTOS task that is running the embedded TCP/IP stack then `pdPASS` is returned. If `xTimeout` is greater than zero then the calling task may have been held in the
Blocked state (so not consuming any CPU time) to wait for the message to be sent - but the message was sent successfully before the function returned.

If the event could not be sent to the RTOS task that is running the embedded TCP/IP stack because the network event queue was full then pdFAIL is returned. If xTimeout is greater than zero then the calling task may have been held in the Blocked state to wait for space to become available on the network event queue, but the block time expired before that happened.
5. Running Examples

5.1 Building Example Program

A project is provided that allows OPENRTOS+TCP to be built and executed using free tools and in a Windows environment, so without the need to purchase any special hardware:

The projects are preconfigured to build with the free version of Visual Studio C/C++, and use the OPENRTOS Win32 port.

5.1.1 Prerequisites

The following are required to build and run the Win32 RTOS port examples:

- A windows host computer with a connected network port (see the hardware setup section below). The projects have been tested with Windows XP and Windows 7.
- An installed version of Visual Studio C/C++. The free Express edition is adequate. The project was created with Visual Studio 2010.
- The OPENRTOS+TCP source code.

5.1.2 Hardware Setup

The Win32 example uses WinPCap to read and write raw Ethernet packets in order to create a virtual node on the network. The virtual node has its own MAC address and IP address. In the examples the host computer uses its real MAC and IP addresses to communicate with the virtual MAC and IP address as if they were two separate computers on the network - whereas in reality both nodes are running on the same PC.

For this setup to work the host PC must be physically connected to a network, even though no other nodes on the network are used, otherwise as far as Windows is concerned the port is disconnected and no communication can take place.
5.1.3 Opening a Project

It is necessary to open the project before completing the software setup. The Visual Studio workspace for the OPENRTOS+TCP starter example is call FreeRTOS_Plus_TCP_Minimal.sln, and is located in the \FreeRTOS-Plus\Demo\FreeRTOS_Plus_TCP_Minimal_Windows_Simulator directory.
5.1.4 Software Setup #1: Setting a Static or Dynamic IP Address

Figure 5-1 Network address settings in FreeRTOSConfig.h

The FreeRTOSIPConfig.h and FreeRTOSConfig.h header files are the OPENRTOS+TCP and OPENRTOS configuration files respectively. Both can be opened from within Visual Studio.
If a DHCP server is present on the network to which the host computer is connected then set `ipconfigUSE_DHCP` to 1 in `FreeRTOSIPConfig.h`, and no further IP address configuration is necessary.

If there is no DHCP server connected to the network then set `ipconfigUSE_DHCP` to 0 in `FreeRTOSIPConfig.h`, then configure the IP address and netmask manually. The IP address and netmask are set using the `configIP_ADDR0/3` and `configNET_MASK0/3` constants respectively in `FreeRTOSConfig.h`. Note that the IP address setting is in `FreeRTOSConfig.h` rather than `FreeRTOSIPConfig.h` because it is related to the application, rather than being a TCP/IP stack configuration option. If you wish to access addresses outside of the local network then it will also be necessary to set the gateway address using the `configGATEWAY_ADDR0/3` constants.

When manually setting the IP address it is necessary to ensure the chosen IP address is compatible with the netmask. In most cases a compatible IP address will be one that uses the same first three octets as the host computer. For example, if the IP address of the host computer is 192.168.0.100 then any 192.168.0.nnn address (other than when nnn is 0 or 255, and any other address already present on the network) will be compatible.

5.1.5 Software Setup #2: Selecting the (virtual) MAC Address

If only one computer that is running the example is connected to the network then it will not be necessary to modify the [virtual] MAC address.

If multiple computers that are running the example are connected to the same network then it will be necessary to ensure each computer has a unique [virtual] MAC address.

The MAC address is set using the `configMAC_ADDR0/5` constants in `FreeRTOSConfig.h`.

5.1.6 Software Setup #3: Selecting the Network Interface

Most computers have multiple network interfaces, and it is necessary to tell the application which interface to use.

Compile (press F7 in Visual Studio) then run (press F5 in Visual Studio) the application. A console screen will display the available network interfaces. Set the `configNETWORK_INTERFACE_TO_USE` constant in `FreeRTOSConfig.h` to the number that appears next to the interface being used. It will then be necessary to re-compile the program.

Trouble shooting:
• If the network interfaces are not displayed then it is likely Windows is not running the NPF service. To correct this type "sc start npf" into a command console (administrator privileges are required), then re-start the application.

• If you cannot establish communication, or if you cannot see any network traffic in Wireshark, then try using a wired network rather than a wireless network. If that is not possible try connecting to (or pinging) the project from a different computer on the same network.

![Network Interfaces Displayed](image)

**Figure 5-2** The available network interfaces displayed when the example starts running
5.2 Running Examples

5.2.1 Print and Logging Messages

FreeRTOSIPConfig.h is provided with FreeRTOS_debug_printf() disabled, and FreeRTOS_printf() set to call a Windows simulator specific utility file called vLoggingPrintf().

Log output can be sent to:

1. A UDP port:
   If xLogToUDP is set to pdTRUE in main.c then log output will be sent to the UDP port number set by configPRINT_PORT in FreeRTOSConfig.h of the remote node that is also configured to be the echo server when the echo server demo example is used.

2. A disk file:
   If xLogToFile is set to pdTRUE in main.c then log output will be written to a file called RTOSDemo.log. When the file reaches 40M bytes in size it is renamed RTOSDemo.ful, and a new log file is started.

3. Standard output:
   If xLogToStdout is set to pdTRUE in main.c then log output will be sent to stdout.
Note: Output related Windows system calls should not be made from RTOS tasks. Therefore standard out and disk file log data is passed to a standard Windows thread for output. UDP logging is sent directly from the RTOS task as it uses OPENRTOS+TCP, not the Windows TCP/IP stack.

5.2.2 Selecting the Examples to Run

Each project contains multiple examples. The examples to include in the build are selected by a set of `#define constants` in `main.c`. Instructions are provided along with the description of each example below.

The supplied demonstration application includes the following examples:

- OPENRTOS+TCP UDP sockets examples
  1. Basic UDP clients communicating with basic UDP servers (standard and zero copy)
- OPENRTOS+TCP TCP sockets examples
  1. TCP echo clients (Rx and Tx performed in the same RTOS task)

5.3 Simple UDP Client and Server Examples

When `mainCREATE_SIMPLE_UDP_CLIENT_SERVER_TASKS` is set to 1 in `main.c` two UDP client RTOS tasks and two UDP server RTOS tasks are created.

The clients talk to the servers. One set of RTOS tasks use the standard sockets interface, and the other RTOS tasks use the zero copy sockets interface.

These RTOS tasks are self checking and will trigger a configASSERT() if they detect a difference in the data that is received from that which was sent. As these RTOS tasks use UDP, which can legitimately lose packets, they will cause configASSERT() to be called when they are run in a less than perfect networking environment.

5.4 A TCP Echo Client Example

When `mainCREATE_TCP_ECHO_TASKS_SINGLE` is set to 1 in `main.c` two RTOS tasks are created that send TCP echo requests to the standard echo port (port 7), then wait to receive the reply from within the same RTOS task. (Another example uses the same TCP socket to send an echo request from one RTOS task then receive the echo reply from another RTOS task).

The IP address of the echo server must be configured using the configECHO_SERVER_ADDR0 to configECHO_SERVER_ADDR3 constants in FreeRTOSConfig.h, and the echo server must (stating the obvious) be enabled and not blocked by a firewall.