FREERTOS+TCP

First, in main, before calling vTaskStartScheduler(), the following call is made to setup the TCP/IP stack.

FreeRTOS\_IPInit(ucIPAddress, ucNetMask, ucGatewayAddress, ucDNSServerAddress, ucMACAddress);

# FreeRTOS\_IPInit

## - 1. The Init function calls three asserts which establish that the function has not already been called.

configASSERT( xIPIsNetworkTaskReady() == pdFALSE );

configASSERT( xNetworkEventQueue == NULL );

configASSERT( xIPTaskHandle == NULL );

## - 2. The queue is created that will be used to communicate with the IP task.

xNetworkEventQueue = xQueueCreate( ( unsigned portBASE\_TYPE ) ipconfigEVENT\_QUEUE\_LENGTH, ( unsigned portBASE\_TYPE )sizeof( IPStackEvent\_t ) );

*Note the FreeRTOS queue create function has the following signature:*

*QueueHandle\_t xQueueCreate( UBaseType\_t uxQueueLength, UBaseType\_t uxItemSize );*

ipconfigEVENT\_QUEUE\_LENGTH is defined as ipconfig\_NUM\_NETWORK\_BUFFER\_DESCRIPTORS + 5

where:

ipconfig\_NUM\_NETWORK\_BUFFER\_DESCRIPTORS is 45.

# **Each of those 50 items is the size of an IPStackEvent\_t, which is defined in FreeRTOS\_IP\_Private.h as:**

typedef struct IP\_TASK\_COMMANDS

{

eIPEvent\_t eEventType;

void \*pvData;

} IPStackEvent\_t;

# **The eIPEvent\_t enum type is as follows:**

typedef enum

{

eNoEvent = -1,

eNetworkDownEvent = 0, /\* 0: The network interface has been lost and/or needs [re]connecting. \*/

eNetworkRxEvent, /\* 1: The network interface has queued a received Ethernet frame. \*/

eARPTimerEvent, /\* 2: The ARP timer expired. \*/

eStackTxEvent, /\* 3: The software stack has queued a packet to transmit. \*/

eDHCPEvent, /\* 4: Process the DHCP state machine. \*/

eTCPTimerEvent, /\* 5: See if any TCP socket needs attention. \*/

eTCPAcceptEvent, /\* 6: Client API FreeRTOS\_accept() waiting for client connections. \*/

eTCPNetStat, /\* 7: IP-task is asked to produce a netstat listing. \*/

eSocketBindEvent, /\* 8: Send a message to the IP-task to bind a socket to a port. \*/

eSocketCloseEvent, /\* 9: Send a message to the IP-task to close a socket. \*/

eSocketSelectEvent, /\*10: Send a message to the IP-task for select(). \*/

eSocketSignalEvent, /\*11: A socket must be signalled. \*/

} eIPEvent\_t;

# **- 3. Initialise the Network Buffers:**

xNetworkBuffersInitialise()

In our case we are using BufferAllocation\_2.

xNetworkBuffersInitialise creates a counting semaphore.

xNetworkBufferSemaphore = xSemaphoreCreateCounting( ipconfigNUM\_NETWORK\_BUFFER\_DESCRIPTORS, ipconfigNUM\_NETWORK\_BUFFER\_DESCRIPTORS );

The buffers are tracked with a xList. This is the same type of list that is used to track the FreeRTOS TCBs.

**static xList xFreeBuffersList;**

The actual entries of the buffer list are of type NetworkBufferDescriptor\_t. A pool of these descriptors are declared statically in BufferAllocation\_2:

**static NetworkBufferDescriptor\_t xNetworkBufferDescriptors[ ipconfigNUM\_NETWORK\_BUFFER\_DESCRIPTORS ];**

# **Struct xLIST**

**typedef struct xLIST**

{

**unsigned portBASE\_TYPE uxNumberOfItems;**

**xListItem \* pxIndex;** /\*< Used to walk through the list. Points to the last item returned by a call to pvListGetOwnerOfNextEntry (). \*/

**xMiniListItem xListEnd;** /\*< List item that contains the maximum possible item value meaning it is always at the end of the list and is therefore used as a marker. \*/

} **xList;**

# **xListItem is defined as:**

**struct xLIST\_ITEM**

{

**portTickType xItemValue;** /\*< The value being listed. In most cases this is used to sort the list in descending order. \*/

**struct xLIST\_ITEM \* pxNext;** /\*< Pointer to the next xListItem in the list. \*/

**struct xLIST\_ITEM \* pxPrevious;**  /\*< Pointer to the previous xListItem in the list. \*/

**void \* pvOwner;** /\*< Pointer to the object (normally a TCB) that contains the list item. There is therefore a two way link between the object containing the list item and the list item itself. \*/

**void \* pvContainer;**  /\*< Pointer to the list in which this list item is placed (if any). \*/

};

**typedef struct xLIST\_ITEM xListItem;**

(Note : xMiniListItem is like xListItem without Owner or Container)

# **NETWORK BUFFER DESCRIPTOR TYPE**

**typedef struct xNETWORK\_BUFFER**

{

**xListItem xBufferListItem;** /\* Used to reference the buffer form the free buffer list or a socket. \*/

**unsigned int ulIPAddress;** /\* Source or destination IP address, depending on usage scenario. \*/

**unsigned char \*pucEthernetBuffer;** /\* Pointer to the start of the Ethernet frame. \*/

**size\_t xDataLength;**  /\* Starts by holding the total Ethernet frame length, then the UDP/TCP payload length. \*/

**unsigned short usPort;** /\* Source or destination port, depending on usage scenario. \*/

**unsigned short usBoundPort;**  /\* The port to which a transmitting socket is bound. \*/

#if( ipconfigUSE\_LINKED\_RX\_MESSAGES != 0 )

struct xNETWORK\_BUFFER \*pxNextBuffer; /\* Possible optimisation for expert users - requires network driver support. \*/

#endif

} **NetworkBufferDescriptor\_t;**

When xNetworkBuffersInitialise is completed we have a list of buffers but no memory has yet been allocated to those buffers. Memory is allocated to the pucEthernetBuffer pointer member.

Lets look a little more at how the list is established. The actual list is somewhat orthogonal to the data of the list, if that makes sense. All the list accounting is through the xBufferListItem of each NetworkBuffer. Since the NetworkBuffer descriptor array was statically allocated and uninitialised, it is placed into the bss section and zero-initialized at startup. (See the list file and startup.s).

This means the xItemValue will be zero for all of the descriptors.

The overall list, **xFreeBuffersList,**  is declared statically in FreeRTOS\_IP.c. Note that the list has a member xListEnd; The list is initialized by doing several things:

1. Setting pxIndex to point to xListEnd.

2. Setting uxNumberOfItems to zero.

3. Setting xListEnd.xItemValue to portMAX\_DELAY, which effectively sets it to the largest word value of 0xffffffff; Recall that all the other list items have xItemValue = 0; No other list item should ever have the max value;

4. Setting the xListEnd next and previous pointers to point back to the xListEnd.

Initially, the list starts out like this:

uxNumberOfItems = 0

pxIndex = xListEnd;

xMiniListItem xListEnd

This is an xList, the list meta data

This is xListEnd. Just after list is initialized, its next and previous

xItemValue = 0xffffffff;

pxNext = xListEnd;

PxPrevious = xListEnd;

pointers just point to itself. Note that it is a MiniListItem, so it does not have owner or container members. It’s owner and its container are implicitly the xList.

Network Buffers are added to the list during initialization as follows:

pxNewListItem->pxNext = pxIterator->pxNext;

pxNewListItem->pxNext->pxPrevious = ( xListItem \* ) pxNewListItem;

pxNewListItem->pxPrevious = pxIterator;

pxIterator->pxNext = ( xListItem \* ) pxNewListItem;

/\* Remember which list the item is in. This allows fast removal of the item later. \*/

pxNewListItem->pvContainer = ( void \* ) pxList;

( pxList->uxNumberOfItems )++;

where pxIterator will always be xListEnd, since xItemValues are always zero for network descriptors.

The list item owner is the corresponding network descriptor that contains the xBufferListItem member.

# **- 4. Setup Default Addressing**

Now the xNetworkEventQueue has been successfully created and the network buffers have been successfully initialised, set the default addressing. FreeRTOS\_IP.c contains two global variables as follows:

NetworkAddressingParameters\_t xNetworkAddressing = { 0, 0, 0, 0, 0 };

NetworkAddressingParameters\_t xDefaultAddressing = { 0, 0, 0, 0, 0 };

DefaultAddressing would be the default if DHCP did not lead to setting an IP address. In the static case, these will be identical. First, the arguments to the init function are set in NetworkAddressing and then copied into DefaultAddressing.

Then for the static IP case:

\*ipLOCAL\_IP\_ADDRESS\_POINTER = xNetworkAddressing.ulDefaultIPAddress;

This variable is actually set on a structure (xDefaultPartUDPPacketHeader) that is defined in FreeRTOS\_UDP\_IP.c

ipLOCAL\_MAC\_ADDRESS is also set in the same structure.

# **- 5. Init Network Sockets**

A call is made to vNetworkSocketsInit(). This is defined in FreeRTOS\_Sockets.c

This initializes a list for each of UDP and TCP. The list of bound sockets.

For each of UDP and TCP, a randomly chosen port for clients is set as the next port to choose. These would be what are referred to as ephemeral ports.

# **- 6. Create the IP Task**

xReturn = xTaskCreate( prvIPTask, "IP-task", ( unsigned short ) ipconfigIP\_TASK\_STACK\_SIZE\_WORDS, NULL, ( unsigned portBASE\_TYPE ) ipconfigIP\_TASK\_PRIORITY, &xIPTaskHandle );

pconfigIP\_TASK\_STACK\_SIZE\_WORDS is 1024

pconfigIP\_TASK\_STACK\_SIZE\_WORDS is 2

# **prvIPTask**

The last thing done by IPInit is to start the IP Task. When the call to vTaskStartScheduler() is made from main.c, the IP task will start running.

Before entering its main processing loop, it calls FreeRTOS\_NetworkDown(); and prvIPTimerReload( &xTCPTimer, pdMS\_TO\_TICKS( ipTCP\_TIMER\_PERIOD\_MS ) );

# **-1a. FreeRTOS\_NetworkDown**

This function, in FreeRTOS\_IP.c, makes the following call:

xSendEventStructToIPTask (&xNetworkDownEvent, xDontBlock )

That function will return either pdPASS or pdFAIL. If it fails, (not the usual case), it will mark the network down event as still pending.

Otherwise, the IPTask will receive this event. Note that the IP task sent the event to itself.

Sending an event means queueing it on the xNetworkEventQueue, which was created in IPInit.

xReturn = xQueueSendToBack( xNetworkEventQueue, pxEvent, xTimeout );

Note here that the xTimeout is set to zero. The function contains a check to see of the event is being queued by the IP Task. If so, the xTimeout value is explicitly set to zero.

# **-1b. Reload TCP Timer**

prvIPTimerReload( &xTCPTimer, pdMS\_TO\_TICKS( ipTCP\_TIMER\_PERIOD\_MS ) );

where ipTCP\_TIMER\_PERIOD\_MS is 1000, which is 1 second.

prvIPTimerReload will set the ulReloadTime to 1000ms and call prvIPTimerStart (xTime), which in turn will do the following (xTime is just passed along from prvIPTimerReload):

- initialize the xTimeOut struct members to zero. (see below, struct defined in tasks.h)

- set the ulRemainingTIme to 1000ms.

- set the timer as not expired (since xTime is not zero)

- set the timer as active

/\* tasks.h \*/

typedef struct xTIME\_OUT

{

portBASE\_TYPE xOverflowCount;

portTickType xTimeOnEntering;

} xTimeOutType;

Now, the IP Task goes into its processing loop.

# **-2. IP Task main loop**

# **-2a prvCheckNetworkTimers();**

Network timers are defined with the following struct.

*typedef struct xIP\_TIMER*

*{*

*unsigned int*

*bActive : 1, /\* This timer is running and must be processed. \*/*

*bExpired : 1; /\* Timer has expired and a task must be processed. \*/*

*xTimeOutType xTimeOut;*

*portTickType ulRemainingTime;*

*portTickType ulReloadTime;*

*} IPTimer\_t;*

For example, the ARP Timer.

*static IPTimer\_t xARPTimer;*

Note the typedef appears to initialize the bActive and bExpired members to 1, however, xARPTimer is declared static and uninitialized, which means it is in the bss section and initially the timer has all members initialized to zero, so it is not active. But this is ok, because we have a network down event waiting to be processed on the queue, and this will cause timers to be restarted.

In addition to the ARP Timer, there are also the following timers.

xDHCPTimer

xDNSTimer

xTCPTimer

The first time, we check network timers, only the TCP timer is active.

For each timer, the function prvIPTimerCheck is called, but this will return pdFALSE is the timer is not active.

Otherwise, for the TCP Timer, the following is done:

*portBASE\_TYPE xWillSleep;*

*/\* xStart keeps a copy of the last time this function was active,*

*and during every call it will be updated with xTaskGetTickCount()*

*'0' means: not yet initialised (although later '0' might be returned*

*by xTaskGetTickCount(), which is no problem). \*/*

*static portTickType xStart = 0;*

*portTickType xTimeNow, xNextTime;*

*portBASE\_TYPE xCheckTCPSockets;*

*extern unsigned int ulNextInitialSequenceNumber;*

*if( uxQueueMessagesWaiting( xNetworkEventQueue ) == 0 )*

*{*

*xWillSleep = pdTRUE;*

*}*

*else*

*{*

*xWillSleep = pdFALSE;*

*}*

*xTimeNow = xTaskGetTickCount();*

*if( xStart != 0 )*

*{*

*/\* It is advised to increment the Initial Sequence Number every 4*

*microseconds which makes 250 times per ms. This will make it harder*

*for a third party to 'guess' our sequence number and 'take over'*

*a TCP connection \*/*

*ulNextInitialSequenceNumber += ipINITIAL\_SEQUENCE\_NUMBER\_FACTOR \* ( ( xTimeNow - xStart ) \* portTICK\_PERIOD\_MS );*

*}*

*xStart = xTimeNow;*

*/\* Sockets need to be checked if the TCP timer has expired. \*/*

*xCheckTCPSockets = prvIPTimerCheck( &xTCPTimer );*

*/\* Sockets will also be checked if there are TCP messages but the*

*message queue is empty (indicated by xWillSleep being true). \*/*

*if( ( xProcessedTCPMessage != pdFALSE ) && ( xWillSleep != pdFALSE ) )*

*{*

*xCheckTCPSockets = pdTRUE;*

*}*

*if( xCheckTCPSockets != pdFALSE )*

*{*

*/\* Attend to the sockets, returning the period after which the*

*check must be repeated. \*/*

*xNextTime = xTCPTimerCheck( xWillSleep );*

*prvIPTimerStart( &xTCPTimer, xNextTime );*

*xProcessedTCPMessage = 0;*

*}*

The first time through this code will not do much. For one, xStart is zero, the timer has likely not expired yet, and there are no xProcessedTCPMessage, so we are just going to go back to the IP task.

# **-2b xNextIPSleep = prvCalculateSleepTime();**

This routine calculates the maximum time we can sleep in the next xQueueReceive call.

It looks at all the active timers and whichever has the shortest remaining time, then that time will be set as the sleep time. Note the first time this is called, only the TCP timer is active. (We are not using DHCP in our build.)

# **-2c xQueueReceive();**

This is done in the following two lines of code. First, xReceivedEvent.eEventType is set to eNoEvent just in case the call times out. Note that we are using the sleep time calculated in step 2b for the timeout argument (xNextIPSleep).

*xReceivedEvent.eEventType = eNoEvent;*

*xQueueReceive( xNetworkEventQueue, ( void \* ) &xReceivedEvent, xNextIPSleep );*

The first time through, this call will receive the eNetworkDownEvent that was put in the queue before the task loop was entered.

Following the receive call, there is a switch statement that switches on the type of event. If the event type is eNetworkDown then the following code is executed.

*xNetworkUp = pdFALSE;*

*prvProcessNetworkDownEvent();*

# **-2d prvProcessNetworkDownEvent();**

This is in FreeRTOS\_IP.c

First, turn off the ARP timer. XARPTimer.bActive = pdFALSE;

Then, there is come code to call an IP Event hook, but this is only executed if ipconfigUSE\_NETWORK\_EVENT\_HOOK is defined as 1. In our build, it is defined as 0.

Next, we call xNetworkInterfaceInitialise(). This is a function that must be provided by the port level code. FreeRTOS + TCP expects this to be provided by the platform port.

The port-specific network code is provided in a file named NetworkInterface.c.

# **-2e xNetworkInterfaceInitialise();**

For the Raspberry PI, the driver code is the USPi driver and library for the SMSC 9514 USB Hub and 10/100 Ethernet controller. It was made by SMSC, which was bought by Microchip.

*if (!USPiInitialize ()){*

*println("Cannot initialize USPi", 0xFFFFFFFF);*

*return pdFAIL;*

*}*

*if (!USPiEthernetAvailable ()){*

*println("Ethernet device not found", 0xFFFFFFFF);*

*return pdFAIL;*

*}*

*xTaskCreate(ethernetPollTask, "poll", 1024, NULL, 2, NULL);*

This initializes the 9514 and checks if the ethernet interface is available.

If so, then it creates the ethernetPollTask. It’s created at the same priority as the IPTask, 2. And currently, I am not sure if that is the best priority to assign, but it seems reasonable.

If all that is successful, then we call vIPNetworkUpCalls();

# **-2f vIPNetworkUpCalls();**

Because of the build options we are using (no NetworkEventHook, no NABTO, and no DNS callbacks), this routine just ends up doing:

prvIPTimerReload( &xARPTimer, pdMS\_TO\_TICKS( ipARP\_TIMER\_PERIOD\_MS ) );

Recall the ARP Timer had been disabled before.

At this point we should have everything we need for basic TCP/IP networking.

If we do nothing else at this point, the link will be up just doing ARP transactions. It will send out gratuitous ARPs to the network, and process ARPs that are sent to it.

# Create a Server Socket

# **1. Get Socket**

To create a server socket, first call:

/\* domain \*/ /\* type \*/

listen\_sock = FreeRTOS\_socket(FREERTOS\_AF\_INET, FREERTOS\_SOCK\_STREAM,

/\* protocol \*/

FREERTOS\_IPPROTO\_TCP);

This function returns a Socket\_t type to the caller. Socket\_t is an opaque pointer (typedef void \* Socket\_t) to a FreeRTOS\_Socket\_t \*. The function internally fills a FreeRTOS\_Socket\_t and returns it by doing : return ( Socket\_t ) pxSocket where pxSocket is declared as FreeRTOS\_Socket\_t \* pxSocket.

The function allocates an amount of storage (pvPortMalloc) for a socket structure based upon the domain, type, and protocol.

It then initializes elements of the structure.

/\* Socket-based initializations \*/

*pxSocket->xReceiveBlockTime = ipconfigSOCK\_DEFAULT\_RECEIVE\_BLOCK\_TIME;*

*pxSocket->xSendBlockTime = ipconfigSOCK\_DEFAULT\_SEND\_BLOCK\_TIME;*

*pxSocket->ucSocketOptions = FREERTOS\_SO\_UDPCKSUM\_OUT;*

*pxSocket->ucProtocol = (unsigned char)xProtocol; /\* protocol: UDP or TCP \*/*

/\* TCP-based initializations \*/

*/\* StreamSize is expressed in number of bytes \*/*

*/\* Round up buffer sizes to nearest multiple of MSS \*/*

*pxSocket->u.xTcp.usInitMSS = pxSocket->u.xTcp.usCurMSS = ipconfigTCP\_MSS;*

*pxSocket->u.xTcp.uxRxStreamSize = ( size\_t ) ipconfigTCP\_RX\_BUF\_LEN;*

*pxSocket->u.xTcp.uxTxStreamSize = ( size\_t ) FreeRTOS\_round\_up( ipconfigTCP\_TX\_BUF\_LEN, ipconfigTCP\_MSS );*

/\* TCP Window-based initializations \*/

*/\* Use half of the buffer size of the TCP windows \*/*

*#if ( ipconfigUSE\_TCP\_WIN == 1 )*

*{*

*pxSocket->u.xTcp.uxRxWinSize = FreeRTOS\_max\_uint32( 1UL, ( unsigned int ) ( pxSocket->u.xTcp.uxRxStreamSize / 2 ) / ipconfigTCP\_MSS );*

*pxSocket->u.xTcp.uxTxWinSize = FreeRTOS\_max\_uint32( 1UL, ( unsigned int ) ( pxSocket->u.xTcp.uxTxStreamSize / 2 ) / ipconfigTCP\_MSS );*

*}*

where the constants have the default values:

*ipconfigSOCK\_DEFAULT\_RECEIVE\_BLOCK\_TIME = portMAX\_DELAY = 0xFFFFFFFF*

*ipconfigSOCK\_DEFAULT\_SEND\_BLOCK\_TIME = portMAX\_DELAY = 0xFFFFFFFF*

*FREERTOS\_SO\_UDPCKSUM\_OUT 2*

*#define ipconfigNETWORK\_MTU 1500*

#define ipconfigTCP\_MSS ( ipconfigNETWORK\_MTU - ipSIZE\_OF\_IP\_HEADER - ipSIZE\_OF\_TCP\_HEADER )

#define ipconfigTCP\_RX\_BUF\_LEN ( 4 \* ipconfigTCP\_MSS ) /\* defaults to 5840 bytes \*/

#define ipconfigTCP\_TX\_BUF\_LEN ( 4 \* ipconfigTCP\_MSS ) /\* defaults to 5840 bytes \*/

If we expand out the values in the #if (configUSE\_TCP\_WIN == 1) block, we can see that the default TCP Rx Window Size is ( ( *ipconfigTCP\_RX\_BUF\_LEN / 2 ) / ipconfigTCP\_MSS )*

which comes out to be 5840 / 2 / 1460 = 2

The default TCP Tx window size is the same.

This socket opaque pointer is returned to the caller to be used in subsequent network calls.

# **2. Bind Socket**

After successfully obtaining a socket, we need to bind it to an address and port if it is to be a listening socket.

In our case, we just need to specify the port, we are using 4242..

struct freertos\_sockaddr server;

The way the main.c code was calling this is:

//server.sin\_port = FreeRTOS\_htons((uint16\_t)4242);

But it looks like the bind code is also doing this.

*status = FreeRTOS\_bind(listen\_sock, &server, sizeof(server));*

The pertinent parts of the bind function are shown below. TL;DR: It sends a bind event to the IP Task with the socket struct as the data. This is done because the IP Task can more naturally provide mutual exclusing to the bound sockets list. Note the API caller will block until the IP Task signal the event is completed (via the xBindEvent, see code below).

/\* Prepare a messages to the IP-task in order to perform the binding.

The desired port number will be passed in 'usLocPort \*/

xBindEvent.eEventType = eSocketBindEvent;

xBindEvent.pvData = ( void \* ) xSocket;

if( pxAddress != NULL )

{

pxSocket->usLocPort = FreeRTOS\_ntohs( pxAddress->sin\_port );

}

else

{

/\* Caller wants to bind to a random port number. \*/

pxSocket->usLocPort = 0;

}

/\* portMAX\_DELAY is used as a the time-out parameter, as binding \*must\*

succeed before the socket can be used. \*/

if( xSendEventStructToIPTask( &xBindEvent, ( portTickType ) portMAX\_DELAY ) == pdFAIL )

{

/\* Failed to wake-up the IP-task, no use to wait for it \*/

FreeRTOS\_debug\_printf( ( "FreeRTOS\_bind: send event failed\n" ) );

xReturn = -pdFREERTOS\_ERRNO\_ECANCELED;

}

else

{

/\* The IP-task will set the 'eSOCKET\_BOUND' bit when it has done its

job. \*/

xEventGroupWaitBits( pxSocket->xEventGroup, eSOCKET\_BOUND, pdTRUE /\*xClearOnExit\*/, pdFALSE /\*xWaitAllBits\*/, portMAX\_DELAY );

if( socketSOCKET\_IS\_BOUND( pxSocket ) == pdFALSE )

{

xReturn = -pdFREERTOS\_ERRNO\_EINVAL;

}

}

# **2. Bind Socket – IPTask handling**

As mentioned before, the IP Task loop basically receives events from the xNetworkEventQueue and then enters a big switch statement on the event type.

In this case, it will receive a eSocketBindEvent.

case eSocketBindEvent:

/\* FreeRTOS\_bind (a user API) wants the IP-task to bind a socket

to a port. The port number is communicated in the socket field

usLocPort. vSocketBind() will actually bind the socket and the

API will unblock as soon as the eSOCKET\_BOUND event is

triggered. \*/

*pxSocket = ( FreeRTOS\_Socket\_t \* ) ( xReceivedEvent.pvData );*

*xAddress.sin\_addr = 0; /\* For the moment. \*/*

*xAddress.sin\_port = FreeRTOS\_ntohs( pxSocket->usLocPort );*

*pxSocket->usLocPort = 0;*

*vSocketBind( pxSocket, &xAddress, sizeof( xAddress ), pdFALSE );*

*/\* Before 'eSocketBindEvent' was sent it was tested that*

*( xEventGroup != NULL ) so it can be used now to wake up the*

*user. \*/*

*pxSocket->xEventBits |= eSOCKET\_BOUND;*

*vWakeUpSocketUser( pxSocket );*

*break;*

Note this is the third time that ntohs has been called on the port :). The first time is the client code that opened the socket. (According to examples found in the online FreeRTOS pages, the client code should do this.) Then the bind call did it, and now the IP Task does it. I suppose as long as it is an odd number of times, it is ok!

Lets go into the vSocketBind call that is made here. It does three important things:

1. Make sure the port is not already bound.

2. Add the port to the socket

3. Add the port to the bound TCP sockets list.

FINDINGS SUMMARY:

- Need to build with -mno-unaligned-access in dbuild.config.mk or program will crash in AllowIPPacket. This happens because it will make a word aligned access to a half word location. And, even though unaligned access might be turned off in the control register, the Cortex A-7 can still fault per the ARM documentation. I dont believe the Data Abort handler will even be called.

- Need to #define ipconfigZERO\_COPY\_TX\_DRIVER (1) or the IP task will either release the network buffer or corrupt it before it gets over to the ethtask to send to the driver.

I first tried to just the send the frame in the NetworkInterfaceOutput routine, as that would be called as part of the IP task and return before the NetworkBuffer was messed with, but then I ran into USPi driver assert error : assert(!pThis→m\_bWaiting). DWHCIDeviceTransferStage function. I still don’t get this one. I thought this would solve the corruption problem, but, then I got the idea of using the ZERO\_COPY setting from looking at the TCP/IP code that sends the frame. The driver is not actually a ZERO\_COPY driver, but since the NetworkBuffer is passed to another task for sending, it acts like a ZERO\_COPY driver.

- Need to set SO\_REUSE\_LISTEN\_SOCKET or the WaitforEvent in FreeRTOS\_accept will block forever. Without this setting, the conditional logic will not execute the code to set the ACCEPT condition. It needs to set parent socket = pSocket for this to happen, but it wont unless bReuse is set in the socket.