**Ethernet Example Tutorial**

# Introduction

This document explains the operation of the Ethernet threads in the ExampleProject project. This example and tutorial is meant to provide the developer with a quick-start guide to working with Ethernet communications and using the UDP protocol with a Primary/Secondary device setup. The Primary/Secondary setup in this case includes an echo step in the communications process which guarantees packet delivery from Primary to Secondary. The purpose of this is to ensure that there is no loss of any critical data transfer. An example might be turning on an end mill in a CNC machine. If for any reason this packet is corrupted or dropped, the machine can suffer costly damage. It’s up to the developer whether they need this, or some even more complete reliability measures. The reader may also notice that this reliability scheme involves a complete packet echo, as opposed to something less costly, like an ACK packet. This was chosen in order to minimize the probability of data corruption when used with critical systems. This example should provide enough information for future developers to create their own system with more or less overhead and reliability as their application demands.

This document assumes the reader has already cloned the Tutorials repository and ExampleProject project into their e2 Studio environment, and is familiar with the process of creating their own unique project based on this. For more information on that, please refer to the Project Creation Tutorial located in the Documents folder of the Tutorials Repo. To reach the Tutorials repo, please go to “<https://github.com/NYUAD-LabOps/Tutorials>”.

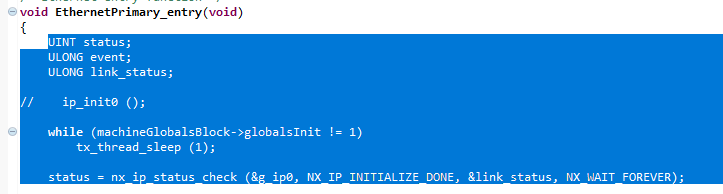
# Overview

The Ethernet example consists of two threads, EthernetPrimary and EthernetSecondary, that communicate with each other via the loopback address “127.0.0.1”. These threads simulate the actions of two real-world devices. In this case, the Primary sends a string to the Secondary every two seconds. This string contains an ASCII representation of an integer which begins at 1 and is incremented with each transmission. The Secondary receives this transmission and echoes it back to the Primary. Both threads print out statements to the console for each step in the interaction, so the developer can confirm that the application is working.

Prior to any sending operations, both threads wait for the Ethernet/IP stack to be initialized, then go through a series of API calls which initialize a socket, bind it to a port number, and tie this port to a receive callback. Initialization of the Ethernet/IP stack itself is handled automatically. This auto-init behavior is a feature enabled in the configuration.xml settings.

# **EthernetPrimary Thread**

Beginning with the “EthernetPrimary\_entry()“, we see the following lines:



The three variables declared here are available for to store values returned by API functions. After this, we see a **while** loop, and a call to nx\_ip\_status\_check. The **while** loop waits for the machineGlobalsBlock struct to be initialized. This is handled by by the MainThread thread with a call to initGlobalsBlock(), which you can see by opening the file “MainThread\_entry.c”.

Once the **while** loop recognizes the globalsInit flag has been set, it exits the loop. Next, we see a call to nx\_ip\_status\_check. This function is given the NX\_WAIT\_FOREVER option, which means it will wait indefinitely until the TCP/IP stack reports that initialization of the Ethernet hardware and software has completed. This initialization is handled by the API automatically, as explained in the Overview.

Next, the application must go through a series of steps to setup a socket and callback function for sending and receiving. These steps can be seen in the Figure 2.



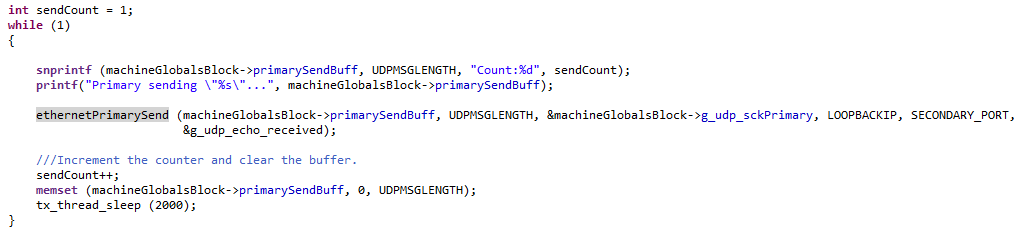
Most of the code here is just printf outputs. Each of the printf outputs is enclosed in an **if** statement that hinges on the value of the DEBUGGER macro that’s defined in “Example.h”. The DEBUGGER macro is used globally when with printf console outputs. This is necessary because printf outputs need to be disabled whenever the developer wants the controller to be capable of automatically running on powerup without a debugger attached. This is more fully explained in the Project Creation Tutorial.

At the end of this section of code, we see the call to tx\_event\_flags\_get. As the comment states, this clears event flags. This line actually does more than this, but in this case it is called only to clear the event flags associated with g\_udp\_echo\_received. This event flag is checked by ethernetPrimarySend when sending a packet, in order to determine whether an echo packet has been received. As explained earlier, in our Primary/Secondary system, every packet sent from Primary to Secondary must be echoed back in order to verify that the packet has been received by the Secondary without error. In our case, the function ethernetPrimarySend will wait until the flag has been set by the receive callback function which checks incoming packets for an echo.

A full explanation of the usage and behavior of event flags is beyond the scope of this document. However, the basic purpose of an event flag is simply to provide a means of triggering some event. The argument NX\_NO\_WAIT is given in this case, and means the function will not wait until the event has been triggered. The argument TX\_AND\_CLEAR, as the name implies, will clear the event flags associated with g\_udp\_echo\_received. This is placed here only to ensure the event flags are cleared prior to any sending operation.

Finally, we enter the code which sends packets to the Secondary. In Figure 3, we see an infinite loop.

**Figure 3**



Just before the loop, the variable sendCount is initialized. Inside the loop, a call to snprintf prepares the contents of the send buffer, machineGlobalsBlock-> primarySendBuff. Next, ethernetPrimarySend is called and supplied with the send buffer, length in bytes, UDP socket pointer, and receiver IP address and port. In the following section we’ll review the sending and receiving functions.

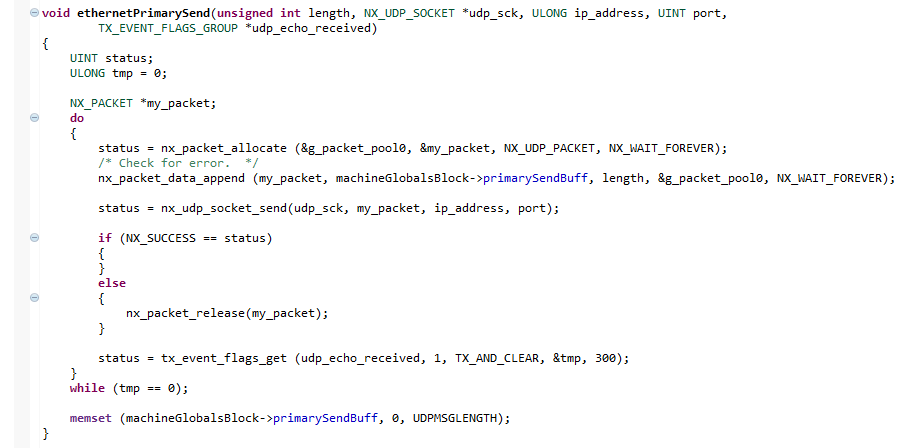
# Sending and Receiving

There are two functions dedicated to sending and receiving. These are ethernetPrimarySend and g\_udp\_sckPrimary\_receive\_cb. In the former case, this is a function that has been prepared specifically for sending data from Primary to Secondary. In the latter, this is a callback function which is called whenever the UDP socket receives an incoming packet on its associated port. This function retrieves the packet data and does any necessary processing, including checking it against outgoing data to determine if it’s an echo packet.

## EthernetPrimarySend

The function ethernetPrimarySend is used to send a packet from Primary to Secondary via UDP.

Figure 4



Looking at Figure 4, we see that the bulk of this function is found within a **do...while** loop. This loop begins with the allocation of a new packet within the packet pool g\_packet\_pool0. This packet pool and the available RAM allocated to it is configured in the configuration.xml file. After allocation, the packet is given the data to be sent, via the pointer machineGlobalsBlock->primarySendBuff. Next, an attempt at sending the packet is made via nx\_udp\_socket\_send(). If this function returns a value of 0, or NX\_SUCCESS, then the send attempt was successful. Otherwise, there was some error, and the packet must be released manually. When the attempt is successful the API will automatically perform the release operation. If the packet is not released, any further attempts at packet allocation could result in memory errors.

The last line of the loop is a call to tx\_event\_flags\_get. Looking at the supplied arguments, we see a 1, TX\_AND\_CLEAR, and 300. By supplying these two arguments together, the function will wait up to 300 timer ticks for the event flag located at bit 1 of udp\_echo\_received to be raised. If after 300 ticks the flag still has not been raised, the function will return the event flag state to tmp, and return the status of the operation to status.

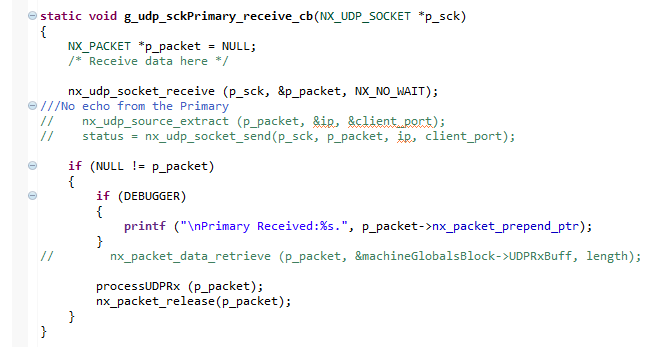
Once the application returns from tx\_event\_flags\_get, the **do...while** compares the value of tmp to 0. If the event flag has been raised, which indicates that an echo packet has been received, the value of tmp will be 1. The end result of this is that the loop will repeatedly send the packet every 300 timer ticks until an echo is received.

Finally, a call to memset is performed to clear the send buffer. It’s important to be aware of this, since any call to this function will result in the data being wiped out. This final step is included because failing to clear the buffer contents could result in the receive callback falsely identifying an incoming packet as an echo.

# g\_udp\_sckPrimary\_receive\_cb

This function is the receive callback, which is called whenever a socket receives a new packet.

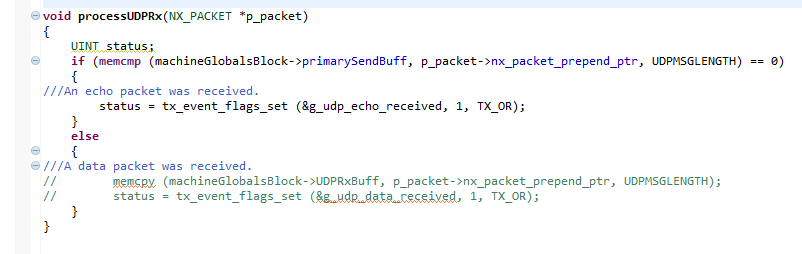
Figure 5



To begin, this function creates a pointer of type NX\_PACKET. It’s sufficient in this case to do this instead of allocating a packet because this has already been handled by the receive function. Instead, we only need to create a pointer that can be used to accept the address of the received packet, which is returned by the function nx\_udp\_socket\_receive(). Next, the function nx\_udp\_socket\_receive() is called to attempt to receive a packet. Next, if the packet pointer is not NULL, indicating that a packet has been received, then processUDPRx() is called to process the received data, and the packet is released from memory.

## processUDPRx

This function is responsible for processing the received packet data. This includes determining whether the packet is an echo from a Secondary, or some other action like toggling a GPIO pin, raising an event flag, etc… In this case, as shown in Figure 5, the function only compares the received data against the contents of the send buffer, located at machineGlobalsBlock->primarySendBuff.



Since the contents of the send buffer are cleared after every send operation, immediately following the receipt of an echo packet, the function need only compare the incoming packet with the send buffer to determine if it is an echo. If the incoming packet is the same as the send buffer, the only possibility is that the packet is an echo packet sent from a Secondary to acknowledge a packet was received. Once this has been verified, the event flag is raised, which triggers the call to tx\_event\_flags\_get() located at the end of ethernetPrimarySend().