# AN1921

## Microchip TCP/IP Lite Stack

Authors:   Janaki Kuruganti,
           Alin Stoicescu,
           Marius Cristea,
           Microchip Technology Inc.

## INTRODUCTION

This application note describes the structure and the interface for the Microchip Transmission Control Protocol/Internet Protocol (TCP/IP) Lite Stack library, and includes some simple example applications using MPLAB® Code Configurator (MCC). The purpose of the implementation is to provide an optimized (low Flash and RAM footprint) TCP/IP stack for microcontrollers with ≥8 KB Flash (UDP only) and ≥16 KB Flash (TCP/IP), while still having a fully functional TCP/IPv4 stack. The stack will allow users to add wired communication and interoperability with other systems to their applications over Ethernet.

The Microchip TCP/IP Lite Stack is implemented in a configurable and modular way, allowing users to include only the intended features or functionalities to their application. The stack is written in C programming language and it is intended to be compiled with the MPLAB® XC8 compiler.

## TCP/IP STACK ARCHITECTURE

The TCP/IP Lite library implementation is based on the TCP/IP communication model, as shown in Figure 1:

**FIGURE 1:        MULTILAYER TCP/IP COMMUNICATION MODEL**

# AN1921
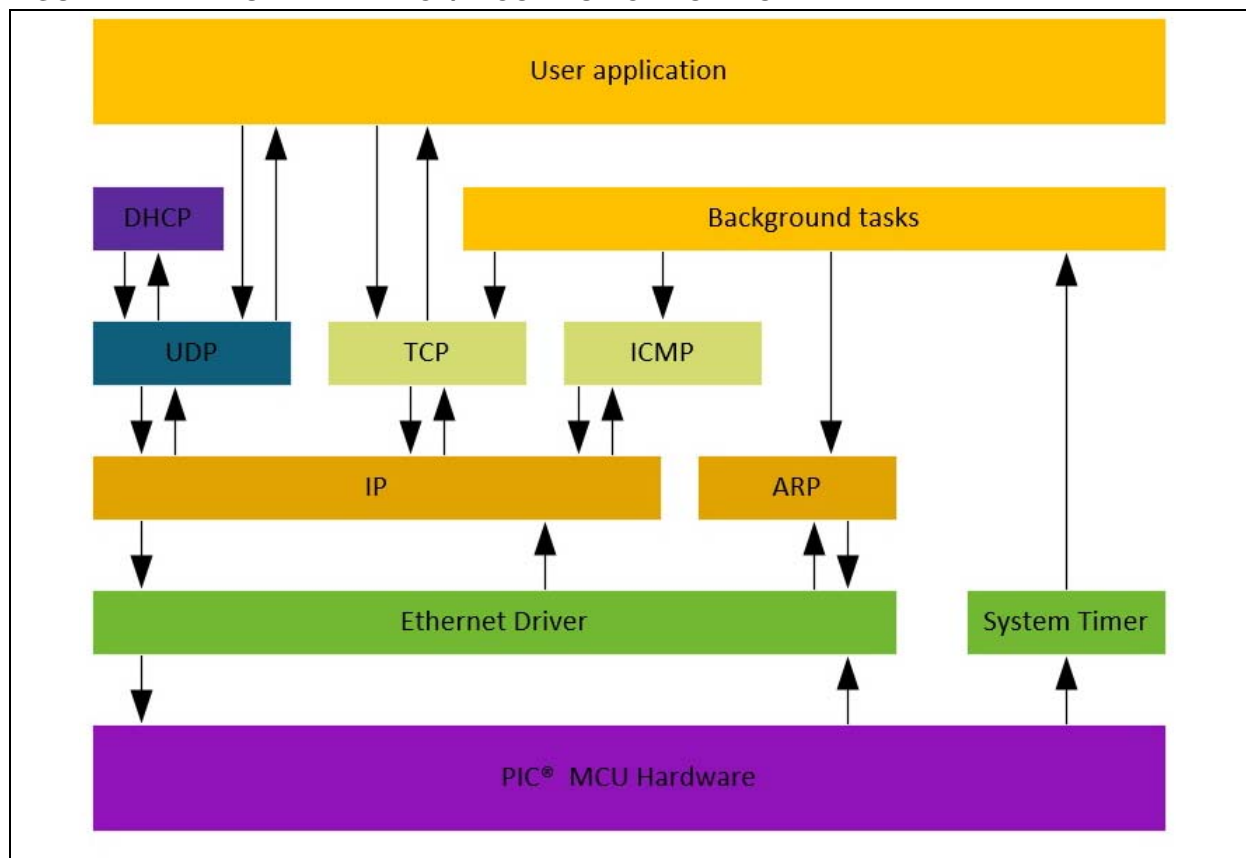
The TCP/IP stack is divided into multiple layers (Figure 1). Each layer in the Microchip TCP/IP Lite Stack can directly access one or more layers situated above or below it.

The TCP/IP stack needs a background task called periodically by the user, in order to handle asynchronous events like managing a timeout, checking the status for the Ethernet controller, and parsing the received buffers.

The code implementing each protocol resides in separate source files, while the services and the Application Programming Interfaces (APIs) are defined through header or include files.

## Stack Configuration

TCP/IP Lite stack is scalable (i.e., the user can configure the protocols as per application requirements). MCC gives the user the flexibility to choose and to generate code only for the needed protocols. Apart from the protocol selection, users also have the option of configuring the stack parameters using MCC. These stack parameters reside in the `tcpip_config.h` file as part of MCC code generation.

Some important configurations include:

1. `dhcpName:`

Used by the DHCP server to assign a human readable name to a MAC address. The default value is the device name ETHERNET, for example: "PIC16F18346 ETHERNET".

2. `ARP_MAP_SIZE:`

This refers to the maximum size of the Address Resolution Protocol (ARP) table. The default value is 8.

3. IP address, Subnet Mask, Default Gateway, Preferred and Alternate DNS Server addresses, in case of static IP address

4. For ICMP, there is an option to generate Echo Response and Port Unreachable messages.

## TCP/IP STACK BUFFER MANAGEMENT

### Overview

The TCP/IP stack uses by default the least possible memory, so that users have the maximum possible memory available to be allocated for their applications. In order to achieve this, users are responsible for providing all the buffers required for each TCP/IP protocol/connection, as described further on.

The Ethernet controller receives and stores multiple packets until the TCP/IP stack can process them. The buffer for each received packet is managed by the Ethernet controller automatically and a buffer descriptor is available for the user. The Ethernet

controller starts dropping the received packets if it does not have enough memory left to store the incoming packets.

Ethernet packets to be transmitted are also built and kept in the Ethernet controller's memory.

## Buffers Used by the UDP Protocol

The stack allows the user to directly store the payload in the Ethernet controller's RAM. The user will call the API to start a UDP packet, transfer the payload and send the packet.

When receiving data over the UDP protocol, the Ethernet controller manages the received packet buffer. If the packet was received successfully and there is a user callback registered for the incoming port, the stack calls the registered function (callback) and gives the user the opportunity to access the payload directly from the Ethernet controller. This avoids copying the payload multiple times and saves time and memory.

## Buffers Used by the TCP Protocol

In case of TCP, the user needs to allocate some memory for each TCP connection. There are a few types of buffers needed by the TCP:

- The Socket Memory
  - Memory allocation for each socket is the user's responsibility. This can be done by calling an API, especially designed for this purpose. This is where all the internal information about the TCP connection is kept.
- The Rx and Tx Buffers
  - The receive (Rx) and the transmit (Tx) buffers for each TCP connection are to be created by the user and passed to the stack via the stack API. Each socket can have only one Rx and one Tx buffer at a time. The stack always needs one Rx buffer available to receive data from the remote host. The stack functions only for a short period of time without the Rx buffer, before asking for packet retransmission.

## TCP/IP Stack Features and Limitations

The TCP/IP stack has some limitations based on the limited memory, for both RAM and Flash, available to run on constrained devices like 8-bit microcontrollers. To find information about the current features and limitations of the stack, refer to MCC's TCP/IP Lite library release notes.

## RUNNING THE TCP/IP STACK DEMOS

### Required Hardware and Software to Run the Demo

1. Curiosity Development Board (DM164137)
2. PIC16F18346 MCU
3. USB Power Supply
4. MPLAB® X v3.45 or later
5. XC8 v1.41 C compiler or later
6. Computer with Windows, Linux or MAC OS
7. TCP/IP Demo Application
8. Ethernet ENC28J60 Click Board (ETH Click)
9. Packet Sender Application (to send and receive UDP/TCP packets). Download the application at https://packetsender.com/.
10. DHCP Server (without it the board cannot retrieve an IP address and the UDP demo will not work). The TCP Demo works with static IP address configuration.
11. Ethernet cables:
   - Straight-through – if the board will be connected to a router/switch
   - Crossover – if the board will be connected to the computer directly

### Setting up the Hardware

1. Connect the ETH click board to the Curiosity Development board (connector J35).
2. Connect the ENC28J60 to an Ethernet network using an Ethernet cable (it can be connected directly to the Ethernet port of a PC). The board must be able to connect to a running DHCP server for the UDP demo.
3. Connect the USB power supply to the Curiosity Development board using the J2 connector.
4. For each application the user should follow the steps presented in the setup chapter related to each demo.

## SETUP THE SOFTWARE FOR TCP CLIENT/SERVER DEMO USING MCC

In order to create an application using TCP the user should configure the TCP/IP Lite module, available through MCC, as shown below:

1. Start MPLAB X and create a new project for the PIC16F18346 device. (This demo uses PIC16F18346, but any 8-bit MCU can be utilized instead.)
2. Open MCC.
3. From _Device Resources → Libraries_, double click on the TCP/IP Lite module, as shown in Figure 2:

**FIGURE 2:** LIBRARY SELECTION
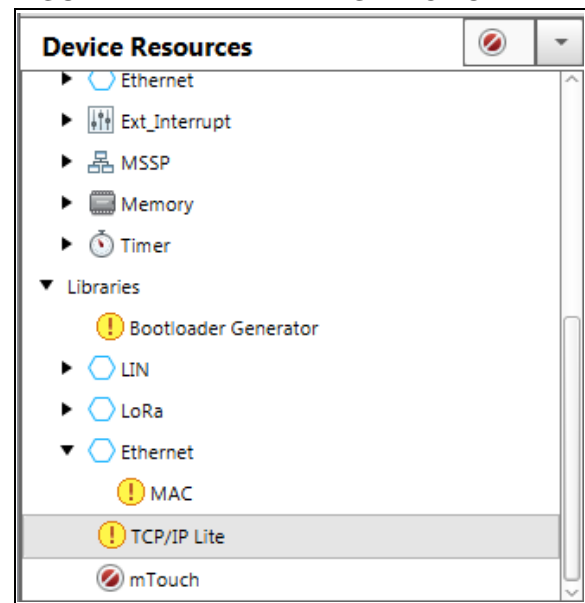
4. A TCP Application requires some changes to the TCP/IP module, as illustrated in Figure 3:
   - UDP – not checked
   - DHCP – not checked. The user must provide its own static IP configuration
   - TCP – checked

FIGURE 3: STACK CONFIGURATION



5. The **Notifications** tab shows different types of messages (Figure 4).
   - "WARNING" messages are required to be addressed to generate an error free code
   - "HINT" messages help the user follow the code generation
   - "INFO" messages allow the user to give information about the loaded modules

The **Notifications** tab shows all the dependencies of the TCP/IP stack.

FIGURE 4: NOTIFICATIONS TAB



| Category | Module Name | Type | Description |
|---|---|---|---|
| ⚠ | TCP/IP Lite | HINT | Add Network_Manage( ) in the main while loop |
| ⚠ | TCP/IP Lite | WARNING | Configure MAC under Ethernet Library for TCP/IP Lite |
| ⚠ | TCP/IP Lite | WARNING | Configure Timer1 to serve as RTCC for TCP/IP Lite |

6. TCP/IP Lite module requires the Ethernet MAC library and Timer1 module, as indicated in Figure 5.

- From *Device Resources → Libraries → Ethernet,* double click on **MAC**
- From *Device Resources → Peripherals → Timer*, double click on **TMR1**

**FIGURE 5: STACK DEPENDENCIES SELECTION**



7. The device needs to be configured with a 1s tick of system clock. The user shall configure the Timer1 as shown in Figure 6 below:

- Timer Period – 250 ms
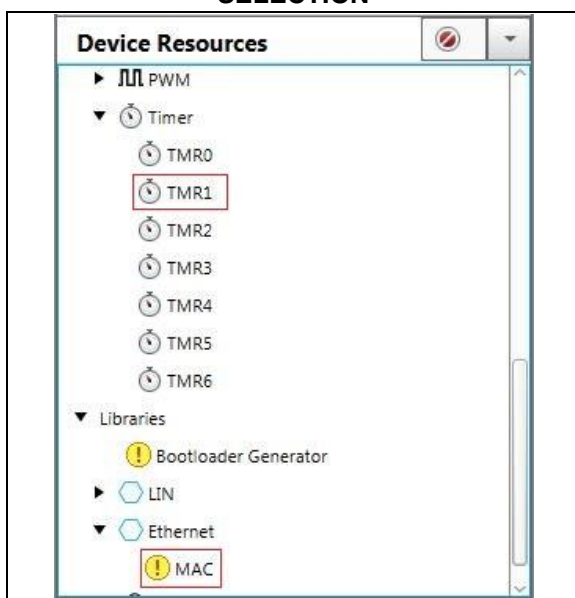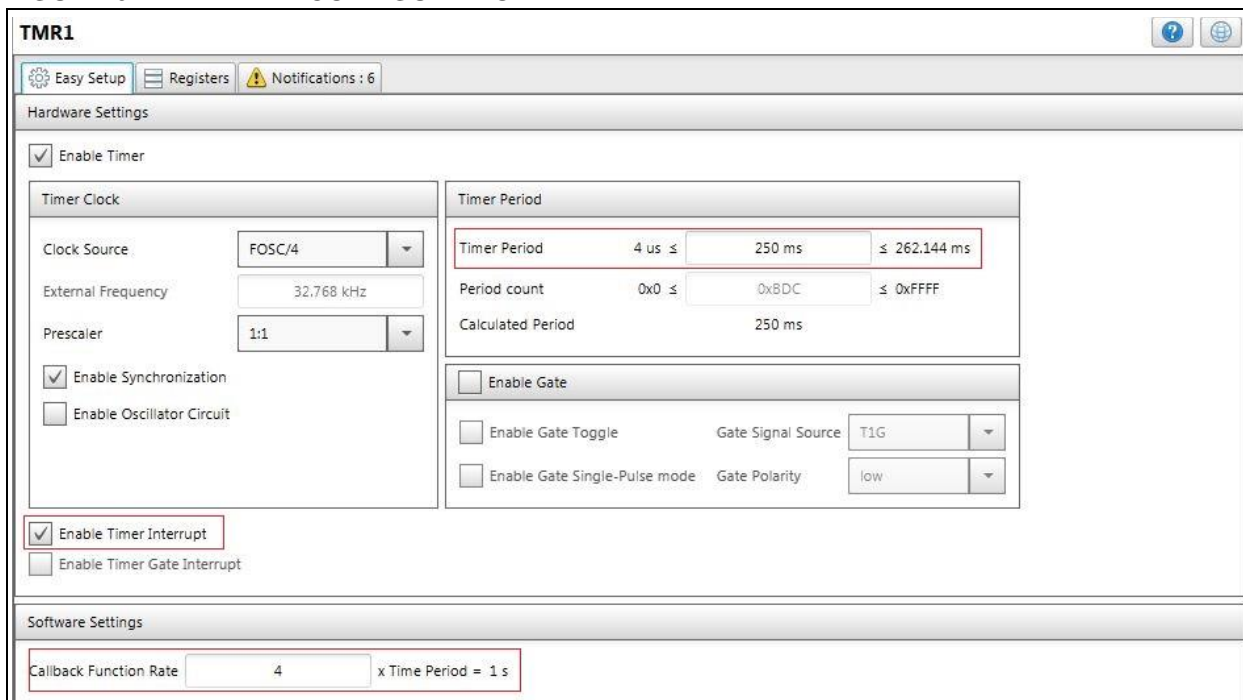- Enable Timer Interrupt – Checked
- Callback Function Rate – 4

**FIGURE 6: TIMER CONFIGURATION**

# AN1921

8. Below are the steps required to configure the Ethernet MAC module (Figure 7):
   - In the Easy Setup window, the user should select the ENC28J60 controller from the drop down selection box. The ENC28J60 controller is based on the Serial Peripheral Interface. The Curiosity Development board (DM164137) supports the MSSP1 – SPI module interface on the J35 connector.

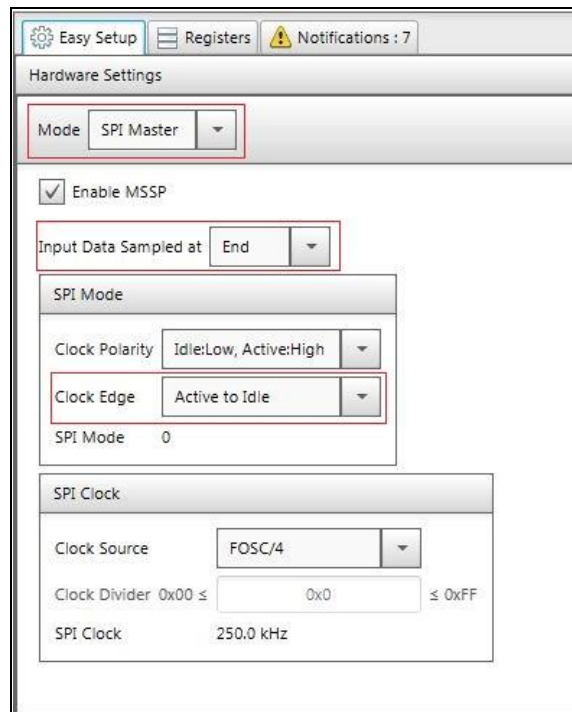**FIGURE 7:      MAC LIBRARY CONFIGURATION**



9. The user shall add the MSSP1 module, as shown in Figure 8:
   - From *Device Resources → Peripherals → MSSP*, double click on MSSP1

**FIGURE 8:      MAC LIBRARY DEPENDENCY SELECTION**
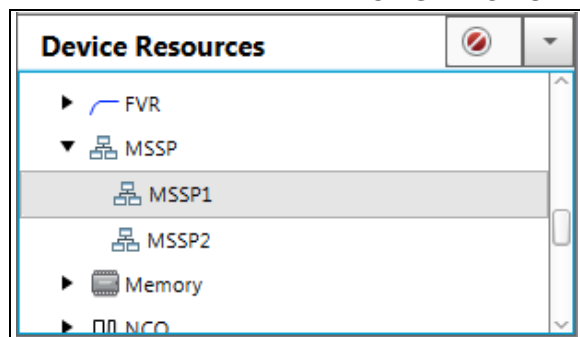


10. Steps required to configure MSSP1 module are shown in Figure 9:
   - Mode – SPI Master
   - Input Data Sampled at – End
   - Clock Edge – Active to Idle

**FIGURE 9:      SPI CONFIGURATION**



11. The user shall use the ADC module to send the potentiometer data over TCP, as shown in Figure 10.
   - From *Device Resources → Peripherals → ADC*, double click on **ADC**

**FIGURE 10:      ADC MODULE SELECTION**



© 2015-2017 Microchip Technology Inc.
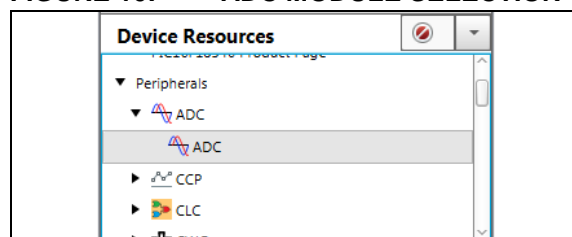
12. The ADC module should be configured (Figure 11) with:
   - Clock Source – F$_{OSC}$/4
   - Result Alignment – Right

**FIGURE 11:     ADC MODULE CONFIGURATION**

13. Steps required to configure the Pin Manager (Figure 12):
- MSSP1 module Pin configuration:
  - SCK1 – output Port RB6
  - SDI1 – input Port RB4
  - SDO1 – output - Port RC7
- MAC module Pin configuration
  - ETH_CS – for the ENC28J60 controller - output Port RC6
- ADC module Pin configuration:
  - ANx – input Port RC0
  - From *Project Resources → System → Pin Module*, the user can provide a custom name to the AN0 channel ex: Pot

- Configure Pin Manager for LED on Curiosity board
  - Pin module – output Port RA5
  - From *Project Resources → System → Pin Module*, the user can provide a custom name to the RA5 Pin ex: Toggle_Led

**FIGURE 12:    PIN FUNCTIONS AND NAMES**



14. All the required configurations were made and the user can click the **Generate** button.

## SIMPLE TCP CLIENT DEMO IMPLEMENTATION

### Overview

This is a simple TCP client implementation that will connect to a server that runs on a computer on port 60. The user needs to modify the server IP address in the firmware. Once the connection is established, the client will send status packets to the server every 2 seconds. The packets sent by the client contain potentiometer value and LED status. From the server, the user can also turn the LED on or off on the board, using the GUI push buttons. In this demo any button will trigger the same LED.

For this demo there is only one active connection implemented, but the TCP/IP stack supports multiple TCP connections on the same board. For each new connection the user needs to create a new socket, an Rx buffer, and try to connect to the server. This demo has been set up to run on the Curiosity board.

The TCP Client demo will try to connect to the server every two seconds. This was implemented so the user can easily watch and analyze the packets using Wireshark protocol analyzer.

### Setting up the Software for the TCP Client Demo

1. The user should find the IP address of the computer where the TCP/IP server Java demo application runs on.
2. Use the TCP/IP Lite library from the previously created project.
3. Add the `tcp_client_demo.c` and `tcp_client_demo.h` files to the project.
4. In `main()`, the user should call `TCP_Client_Initialize()` to add the server IP address and port number. The IP address is the one from step 1, as shown in Example 1.

### EXAMPLE 1: SETTING SERVER LOCATION

```
remoteSocket.addr.s_addr = MAKE_IPV4_AD-
DRESS(192,168,0,3);
remoteSocket.port = 60;
```

5. In `main()`, the user shall enable the Global and Peripheral Interrupts.
6. In `while(1)` loop, the `Network_Manage()` command must be called. It is an API which polls the Ethernet controller for new packets and processes them.
7. In `while(1)` loop, the `DEMO_TCP_Client()` command must be called to handle the socket states of the Client connected to the server.
8. The project must be compiled using XC8.

9. Using MPLAB X, the user can program the firmware on the PIC16F18346 on the Curiosity Development board.
10. If the IP address configured in MCC and the IP address of the computer where the TCP/IP server runs on are both unique, valid and in the same network, they must be able to exchange packets. The user can test this by sending ICMP messages (using the ping command).
11. Start the TCP/IP demo Java application on the computer, as shown in Figure 13.
12. Go to the TCP Server Demo tab.
13. Go to the S*erver → Local Port* and change the port number to 60.
14. Push the **Listen** button (The status will be changed to "Listen..." and the button should be replaced by one with a Disconnect option).
15. When the Curiosity board will connect to the computer, a message in the Sent/Received Data windows will be printed, as shown in Example 2.

### EXAMPLE 2: CONNECTION STATUS

```
192.168.0.21: Connected
```

16. In the Send field, the user can type a message to be sent to the board. The message will be sent when the **Enter** or **Send** button is pressed. Both sent data and received data will be printed in different colors in the Sent/Received Data window.
17. Messages that came from the board will be automatically printed in Sent/Received Data and contain the LED state and the raw value of the on-board potentiometer. Values are in hexadecimal format.
18. Pushing the **LED 0 – LED 7** button will initiate the sending of a command to the board. In this demo, all buttons will trigger the same LED on the board (will be turned on or off). The implementation supports only one LED turning on or off at a time.
19. Pushing the **Disconnect** button will close the Server connection and print a "Server closed" message followed by a "Client disconnected" message. This message will be printed for each "Client connected" message.
20. Repeat steps 13 to 18 to test the connection.

# AN1921

## TCP Client Demo Firmware – Buffer Creation

Before starting the Client, the user needs to create the socket and also, at least, the Rx buffer, as shown in Example 3. The Tx buffer will be created by the user and passed to the TCP stack when it is ready to be sent.

**EXAMPLE 3: SOCKET CREATION**

```
// create the socket for the TCP Client
tcpTCB_t port60TCB;

// create the TX and RX buffers
uint8_t rxdataPort60[50];
uint8_t txdataPort60[80];
```

## TCP Client Implementation

The steps required to start and implement the TCP Client are as follows:

1.  Initialize the TCP stack. The function should be called before any other TCP function is called. It is done automatically through `Network_Init()` by the MCC's `SYSTEM_Initialize()` function.

    `TCP_Init();`

2.  Set the IP address and port number. The user needs to provide the port number to connect to (this is the port number where the server listens on) and the IP address of the computer where the TCP/IP server Java demo application runs on. This function should be called before the `while(1)` loop.

    `TCP_Client_Initialize();`

3.  Check the status of the socket. This function checks if the pointer provided as parameter is registered internally to the TCP/IP stack as a socket. If the pointer is a valid socket, the function will return the state of that socket. The possible states of the socket are defined in the `tcpv4.h` file.

    `socket_state = TCP_SocketPoll(&port7TCB);`

4.  Insert and initialize the socket in order to create the connection. All the necessary information for the TCP connection is kept here.

    `TCP_SocketInit(&port60TCB);`

5.  Set the local port for the client. This step is not mandatory, as the TCP stack will use the next available port number as a local port. The user needs to use `TCP_Bind()` to make sure that a certain port number will be used when the connection is initiated. This is useful when the server accepts connections only from a certain port number.

    `TCP_Bind(&port60TCB, 1024);`

6.  Add the Rx buffer to the socket. The function will insert the buffer into the socket. It will be used for saving the received data.

    `TCP_InsertRxBuffer(&port60TCB, rxdataPort60, sizeof(rxdataPort60));`

7.  Start the Client by calling the `TCP_Connect()` function that will initiate the TCP connect procedure for connecting to the server. If the TCP handshake is successful, the user can exchange data with the remote server over the TCP connection.

    `TCP_Connect(&port60TCB, &remoteSocket);`

8.  Close a TCP connection. If the connection failed, the Client should abort the connection and close the socket. The user should initiate a new TCP connect procedure every two seconds. This function will close the TCP connection. The user needs to check the socket state periodically, until the socket is in closed state. When the socket is in closed state, the Rx and Tx buffers can be safely reused.

    `TCP_Close(&port60TCB);`

9. Check if the Tx buffer was sent correctly. This function needs to be called before trying to send anything, because the socket can handle only one buffer at a time.

```
TCP_SendDone(&port60TCB);
```

10. Read available bytes in Rx buffer and make the buffer user ready. The function below will return the number of bytes available in the buffer. After calling this function, the user can access the buffer in a safe way. Once the function is called, the stack will not save further received data into this Rx buffer. The user should provide, as quickly as possible, another Rx buffer to the stack (in order to avoid packet retransmission).

```
rxLen = TCP_GetReceivedData(&port60TCB);
```

11. Send the buffer to the remote machine. The API will allow the user to send data over an active TCP connection. The data cannot be sent if the connection is not established between the local and remote host.

```
TCP_Send(&port60TCB, txdataPort60, txLen);
```

12. Remove the socket. When the socket is closed, if the user wants to remove the socket from the internal socket list, the following API will remove the pointer.

```
TCP_SocketRemove(&port60TCB);
```

13. Background task. The function below needs to be called periodically by the application, in order to handle the timeouts from the TCP stack. The TCP background task is called once per second to handle the TCP stack timeouts.

```
TCP_Update(); // handle timeouts
```

## Source Code for the TCP Client Implementation

The TCP Client demo code in Example 4 (source code and prebuilt hex file) is available as download on www.microchip.com.

---

### *Software License Agreement*

The software supplied herewith by Microchip Technology Incorporated (the "Company") is intended and supplied to you, the Company's customer, for use solely and exclusively with products manufactured by the Company.

The software is owned by the Company and/or its supplier, and is protected under applicable copyright laws. All rights are reserved. Any use in violation of the foregoing restrictions may subject the user to criminal sanctions under applicable laws, as well as to civil liability for the breach of the terms and conditions of this license.

THIS SOFTWARE IS PROVIDED IN AN "AS IS" CONDITION. NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.

---

**EXAMPLE 4: TCP CLIENT DEMO SOURCE CODE**

```c
void DEMO_TCP_Client(void)
{
        // create the socket for the TCP Client
        static tcpTCB_t port60TCB;


        // create the TX and RX Client's buffers
        static uint8_t rxdataPort60[50];
        static uint8_t txdataPort60[80];


        static time_t t_client = 0;
        static time_t socketTimeout = 0;


        uint16_t rx_len;
        sockaddr_in_t remoteSocket;
        socketState_t socketState;
        rx_len = 0;


        socketState = TCP_SocketPoll(&port60TCB);


        time(&t_client);


        switch(socketState)
        {
                case NOT_A_SOCKET:
                        // Inserting and initializing the socket
                        TCP_SocketInit(&port60TCB);
                        break;

                case SOCKET_CLOSED:
                        // if the socket is closed we will try to connect again
                        // try to connect once at 2 seconds
                        socketTimeout = t_client + 2;
                        TCP_InsertRxBuffer(&port60TCB,rxdataPort60, sizeof(rxdataPort60));
                        TCP_Connect(&port60TCB, &remoteSocket);
                        break;

                case SOCKET_IN_PROGRESS:
                        // close the socket
                        if(t_client >= socketTimeout)
                        {
                                TCP_Close(&port60TCB);
                        }
                        break;
```

# AN1921

```
                        case SOCKET_CONNECTED:
                                // implement an echo client over TCP
                                // check if the previous buffer was sent
                                if (TCP_SendDone(&port60TCB))
                                {
                                    rx_len = TCP_GetReceivedData(&port60TCB);

                                    // handle the incoming data
                                    if(rx_len > 0)
                                    {

                                            /*
                                            ……………………………………………………………………….
                                            LED Command parsing and LCD updates was removed from this example.
                                            The full code is available in the source code.
                                            ……………………………………………………………………….
                                            */

                                             // reuse the RX buffer
                                             TCP_InsertRxBuffer(&port60TCB, rxdataPort60, sizeof(rxdataPort60));
                                    }

                                    if(t_client >= socketTimeout)
                                    {

                                            // send board status message only once at 2 seconds
                                            socketTimeout = t_client + 2;

                                            /*
                                            ……………………………………………………………………….
                                            Composing the TX message in the TX buffer was removed from this example.
                                            The full code is available in the source code.
                                            ……………………………………………………………………….
                                            */

                                            //send data back to the source
                                            TCP_Send(&port60TCB, txdataPort60, strlen(txdataPort60));
                                    }
                                }
                                break;

                        case SOCKET_CLOSING:
                                //remove the used socket form the list
                                TCP_SocketRemove(&port60TCB);
                                break;

                        default:
                                break;
            }
}
```

## SIMPLE TCP SERVER DEMO IMPLEMENTATION

### Overview

This is a simple TCP echo server implementation, listening on port 7. The Server is started on the Curiosity board and it will wait for any incoming connection. The Server will echo back all the received data once the connection with a client is established. Only one active connection will be created for this demo, but the TCP/IP stack supports multiple TCP connections on the same board. The user needs to create a new Server (create buffers, initialize and start listening) for each new connection.

### Setting up the Software for the TCP Server Demo

1. The user should create a project as it was shown in **Section "Setup the Software for TCP Client/Server Demo Using MCC"**, this time excluding steps 11 and 12. The Server does not need an ADC module or LEDs to be toggled.
2. Add the `tcp_server_demo.c` and `tcp_server_demo.h` files to the project.
3. In the `main()` function, the user will enable the Global and Peripheral Interrupts.
4. In `while(1)` loop, the `Network_Manage()` function must be called. It is an API which polls the Ethernet controller for new packets and processes them.
5. In `while(1)` loop, the `DEMO_TCP_EchoServer()` function must be called to open the socket and listen for incoming clients.
6. The project must be compiled using XC8.
7. Using MPLAB X, the user can program the firmware to the PIC16F18346 on the Curiosity Development board.
8. Start the TCP/IP demo Java application on the computer (see Figure 14).
9. Go to the TCP Client Demo tab.
10. Go to the "Server IP Address" and set the IP address and the port number to 7. The IP address is static and it was configured using MCC.
11. Click on the **Connect** button.
12. When the computer is connected to the Curiosity board, a message in the Sent/Received Data windows will be printed, as shown in Example 5.

### EXAMPLE 5: INITIAL LOG MESSAGE

```
Connected to 192.168.0.21 Port: 7
```

13. The user can type in the Send window and press

**Enter** from keyboard or push the **Send** button to send the string. Both sent and received data will be printed with different colors in the Sent/Received Data window.
14. Pushing the **Disconnect** button will close the TCP connection. A "Connection closed" message will be printed.
15. Repeat steps 9 to 13 to test the connection using different strings length.
16. To generate TCP traffic to the board the **ECHO back received message** button should be enabled. The Send text box should be filled with the desired message to be sent to the board. Pushing the **Send** button will initiate the data exchange. To stop the TCP traffic the **Echo Back received message button** should be pushed again. In this case, in the Sent/Received Data window, only the received messages will be printed.

### FIGURE 14: MICROCHIP TCP CLIENT DEMO FOR JAVA APPLICATION
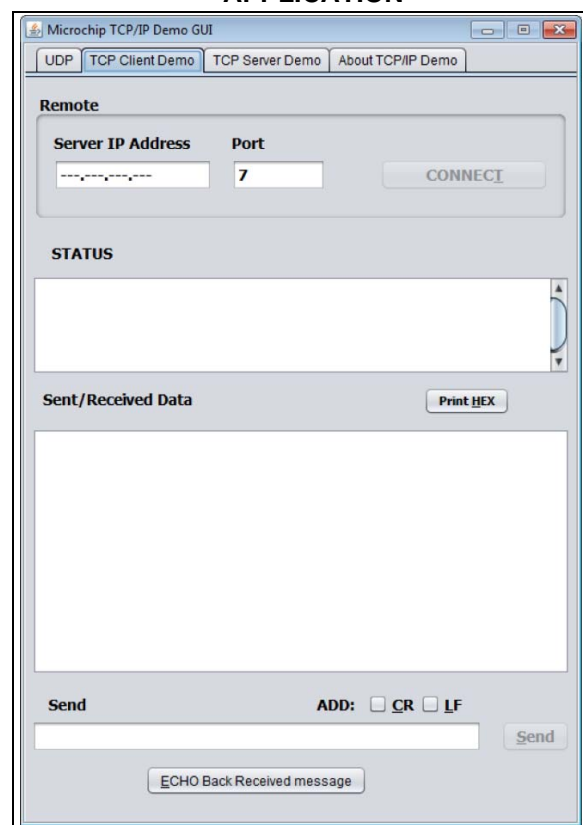
## TCP Server Demo Firmware – Buffer Creation

Before starting the Server, the user needs to create the socket and also, at least, the Rx buffer. The Tx buffer will be created by the user and passed to the TCP stack when it is ready to be sent.

### EXAMPLE 6: SOCKET CREATION

```
// create the socket for the TCP Server
tcpTCB_t port7TCB;

// create the TX and RX buffers
uint8_t rxdataPort7[20];
uint8_t txdataPort7[20];
```

## TCP Server Implementation

These are the steps required to implement the TCP Server:

1. Initialize the TCP stack. The function should be called before any other TCP function. It is done automatically through `Network_Init()` by the MCC's `SYSTEM_Initialize()` function.

   `TCP_Init();`

2. Check the status of the socket: This function checks if the pointer provided as parameter is registered internally to the TCP/IP stack as a socket. If the pointer is a valid socket, the function will return the state of that socket. The possible states of the socket are defined in the `tcpv4.h` file.

   `socket_state = TCP_SocketPoll(&port7TCB);`

3. Insert and initialize the socket in order to create the connection. All the necessary information for the TCP connection is kept here.

   `TCP_SocketInit(&port7TCB);`

4. Assign the local port for the Server. The function will assign a port for the socket to listen on. The Server will listen on this port for any incoming connections. The TCP stack will automatically allocate a port number to listen on if a port number is not supplied by the user.

   `TCP_Bind(&port7TCB, 7);`

5. Add the receive buffer to a socket. The function will insert the buffer into the socket for storing the received data.

   `TCP_InsertRxBuffer(&port7TCB,`

   `rxdataPort7, sizeof(rxdataPort7));`

6. Start the TCP Server. The function will set the TCP stack to listen to a port for a connection request. If the TCP handshake completes successfully, the user can exchange data with the remote over the TCP connection. Only one connection request is accepted at a time. The TCP stack can handle multiple connections for a particular port number. But for each new connection the user needs to create a new socket, an Rx buffer, and start a new instance of the Server for the same port.

   `TCP_Listen(&port7TCB);`

7. Check the status of the socket. This function checks if the pointer provided as parameter is already registered to the TCP/IP stack as a socket. If the pointer is a valid socket, the function will return the state of that socket. The possible states of the socket are defined in the `tcpv4.h` file.

   `socket_state = TCP_SocketPoll(&port7TCB);`

8. Check if the Tx buffer was sent correctly (this means that the remote acknowledged all the received bytes). This function needs to be called before trying to send anything, because the socket can handle only one buffer at a time.

   `TCP_SendDone(&port7TCB);`

9. Check if there is any data received in the socket. The function will return the number of bytes available in the Rx buffer.

   `rxLen = TCP_GetRxLength(&port7TCB);`

10. Read how many bytes are available in the Rx buffer and make the buffer ready to use. The function will return the number of bytes available in the buffer. After calling this function, the user can access the buffer in a safe way. Once the function is called, the stack will not save any further data received into this Rx buffer. The user should provide, as fast as possible, another Rx buffer to the stack, in order to avoid packet retransmission.

    `rxLen = TCP_GetReceivedData`
    `(&port7TCB);`

11. Send the buffer to the remote machine. The API will allow the user to send data over an active TCP connection. The data cannot be sent if the connection is not established between the local and the remote host.

    `TCP_Send(&port7TCB, txdataPort7,`
    `txLen);`

12. Close the TCP connection. Socket connection closing will happen after the TCP connection handshake is done (the connection closing is not done right away). The user needs to check the socket state periodically, until the socket is in Closed state. When the socket is in Closed state, the Rx and the Tx buffers can be safely reused.

    `TCP_Close(&port7TCB)`

13. Remove the socket. If the user wants to remove the socket from the internal socket list when the socket is closed, the following API will remove the pointer.

```
TCP_SocketRemove(&port7TCB)
```

14. Background task. This function needs to be called periodically by the application, in order to handle the timeouts from the TCP stack. The TCP background task is called once per second to handle the TCP stack timeouts.

```
TCP_Update();// handle timeouts
```

## Source Code for the TCP Server Implementation

The TCP Client demo code in Example 7 (source code and prebuilt hex file) is available as download on www.microchip.com.

### EXAMPLE 7: TCP ECHO SERVER IMPLEMENTATION

```
void DEMO_TCP_echo_server(void)
{
        // create the socket for the TCP Server
        static tcpTCB_t port7TCB;


        // create the TX and RX buffers
        static uint8_t rxdataPort7[20];
        static uint8_t txdataPort7[20];


        uint16_t rxLen, txLen, i;
        socket_state_t socket_state;
        rxLen = 0;


        // checking the status of the socket
        socket_state = TCP_SocketPoll(&port7TCB);


        switch(socket_state)
        {
                case NOT_A_SOCKET:
                        //Inserting and initializing the socket
                        TCP_SocketInit(&port7TCB);

                case SOCKET_CLOSED:
                        //configure the local port
                        TCP_Bind(&port7TCB, 7);


                        // add receive buffer
                        TCP_InsertRxBuffer(&port7TCB,rxdataPort7, sizeof(rxdataPort7));


                        // start the server
                        TCP_Listen(&port7TCB);
                        break;
```

# AN1921

```
            case SOCKET_CONNECTED:

                    // check if the buffer was sent, if yes we can reuse the //buffer

                    if(TCP_SendDone(&port7TCB))
                    {
                            // check to see if there are any received data
                            rxLen = TCP_GetRxLength(&port7TCB);

                            if(rxLen > 0)
                            {
                                    //make sure it safe to use the receive buffer
                                    rxLen = TCP_GetReceivedData(&port7TCB);

                                    //simulate some buffer processing copy from //the RX buffer to the TX buffer
                                    for(i = 0; i < rxLen; i++)
                                    {
                                        txdataPort7[i] = rxdataPort7[i];
                                    }

                                    // reuse the rx buffer
                                    TCP_InsertRxBuffer(&port7TCB,rxdataPort7, sizeof(rxdataPort7));

                                    txLen = rxLen;

                                    //send data back to the source
                                    TCP_Send(&port7TCB, txdataPort7, txLen);
                            }
                    }
                    break;


            case SOCKET_CLOSING:
                    TCP_SocketRemove(&port60TCB);
                    break;
            default:
                    // we should not end up here
                    break;
        }
}
```

## UDP DEMO

### Overview

This is a UDP Client and Server implementation. It consists of UDP Send (UDP Client) and UDP Receive (UDP Server) implementations. As UDP Send, the Curiosity Development board sends potentiometer readings as UDP packets. As UDP Receive, the Curiosity Development board starts listening to any incoming UDP packets such as toggle LEDs on port 65531. The port numbers can be anything between in 0 to 65535, but some of them are reserved or already registered. Therefore, it is recommended to choose a port between 49152 to 65535, since all of them are free.

### Setting up the Software for UDP Send/ Receive (Client/Server) Demo

1.  Start MPLAB X and create a new project for the device PIC16F18346.
2.  Load the MCC module.
3.  Go to Device Resources, and under Libraries select TCP/IP Lite module.
4.  Load the TCP/IP Lite module.
5.  The **Notifications** tab shows different types of messages:
    - "WARNING" messages are required to be implemented to generate an error free code
    - "HINT" messages help the user follow the code generation
    - "INFO" messages allow the user to give information about the modules loaded
6.  Go to Device Resources, under Libraries select *Ethernet → MAC module*.
7.  Load the MAC module.
8.  In the Easy Setup, select the ENC28J60 controller from the drop down selection box.
9.  The ENC28J60 controller is based on Serial Peripheral Interface. So, as per the Curiosity Development board schematics, the "MSSP1 – SPI module" interface is supported on the connector J35.
10. Go to Device Resources, under Peripherals select MSSP1 module.
11. Load MSSP1 module and configure to SPI Master mode and select the Clock Edge to Active to Idle mode, and set Input Data Sampled at "End".
12. Go to Device Resources, and under Peripherals select TMR1 module.
13. Update the Timer Period to 250 ms. Enable the Timer Interrupt and update the Callback Function Rate to 4 to generate a 1s period. This feature is required to configure the device with a 1s tick of system clock.

14. Configure the ADC module to send the potentiometer data over UDP. Go to Device Resources, under Peripherals select ADC module. Load the ADC module.
15. In the ADC module, configure Clock Source to F$_{OSC}$/4, and Result Alignment to right.
16. Configure the Pin Manager:
- MSSP1 module Pin Configuration
    - SCK1 – output Port RB6
    - SDI1 – input Port RB4
    - SDO1 – output Port RC7
- MAC module Pin Configuration
- ETH_CS (for the ENC28J60 controller) – output Port RC6
- ADC module Pin Configuration
    - ANx – input PORT RC0
    - From *Project Resource → System → Pin Module*, the user can provide a custom name to the AN0 channel, for example: Pot
- Configure the Pin Manager for LEDs on Curiosity board:
    - Pin module – output Port RA5
    - From *Project Resource → System → Pin Module*, the user can provide a custom name to the RA5 Pin, for example: Toggle_Led
17. Click on **Generate** button to generate the code.
18. Add the `udp_demo.c` and `udp_demo.h` files to the project.
19. In the `main()` function, the user should enable the Global and Peripheral interrupts.
20. In `main()`, call the `POT_UDP_Initialize()` API in the `udp_demo.c` file to initialize the potentiometer and UDP server packet i.e., UDP server (destination) Port Number – 65531, UDP Server Address – computer IP address (Destination address), and UDP Client (Source) Port Number – 65533.
21. In `while(1)` loop, call the `Network_Mange()` API which polls the Ethernet controller for new packets and processes them.
22. The project must be compiled using XC8 and program the firmware to the PIC16F18346 on the Curiosity Development board.
23. Start the Packet Sender application (Figure 15) on the computer.

**FIGURE 15:     PACKET SENDER INTERFACE**
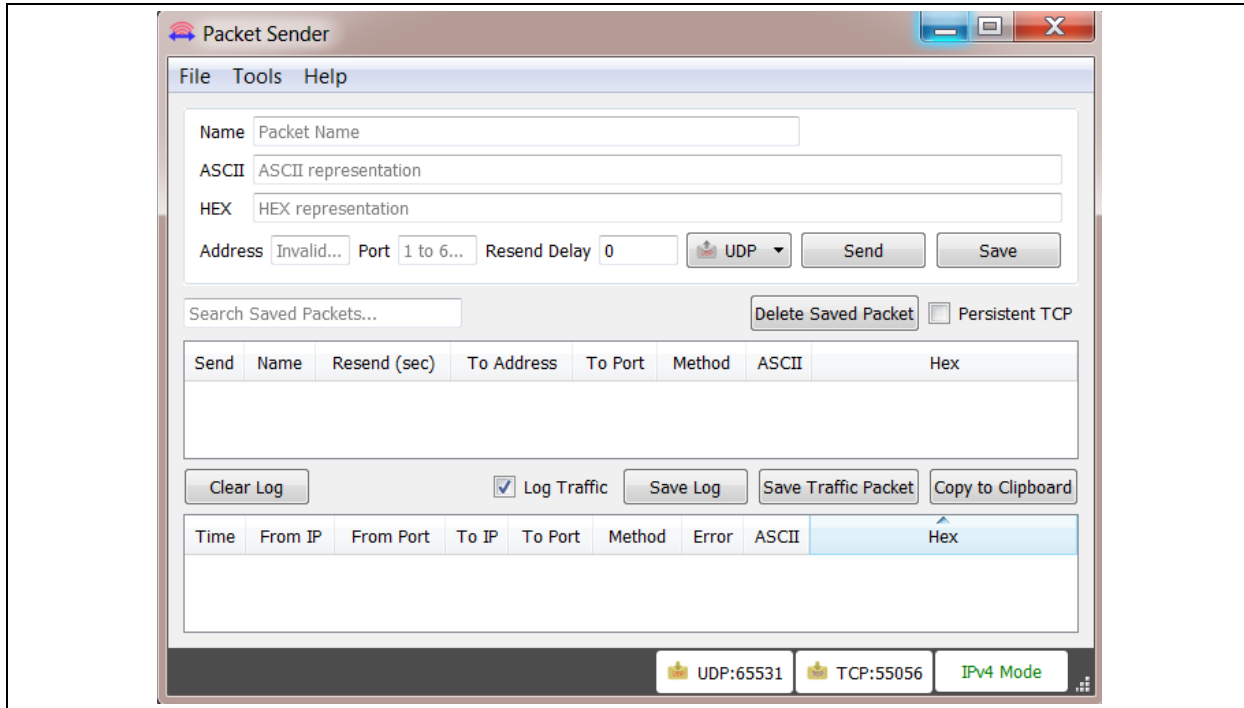


24. Go to *File →Settings →Network*, enable the UDP Server and update the port to 65531.

25. Turn the knob on the Curiosity Development board. The board sends UDP packets to display the potentiometer reading in volts.

26. In the Packet Sender application, send a UDP packet to turn the LEDs on the Curiosity Development board to toggle.

- Name – UDP Led1 Send
- ASCII – L1
- Address – board's IP address. This can be accessed from the D-Discover O-Offer R-Request A-Acknowledgment process using the Wireshark analyzer tool.
- Port – 65531
- Select the UDP protocol
- Click on **Save** to save this packet

**FIGURE 16:**      **UDP PACKET CONFIGURATION**
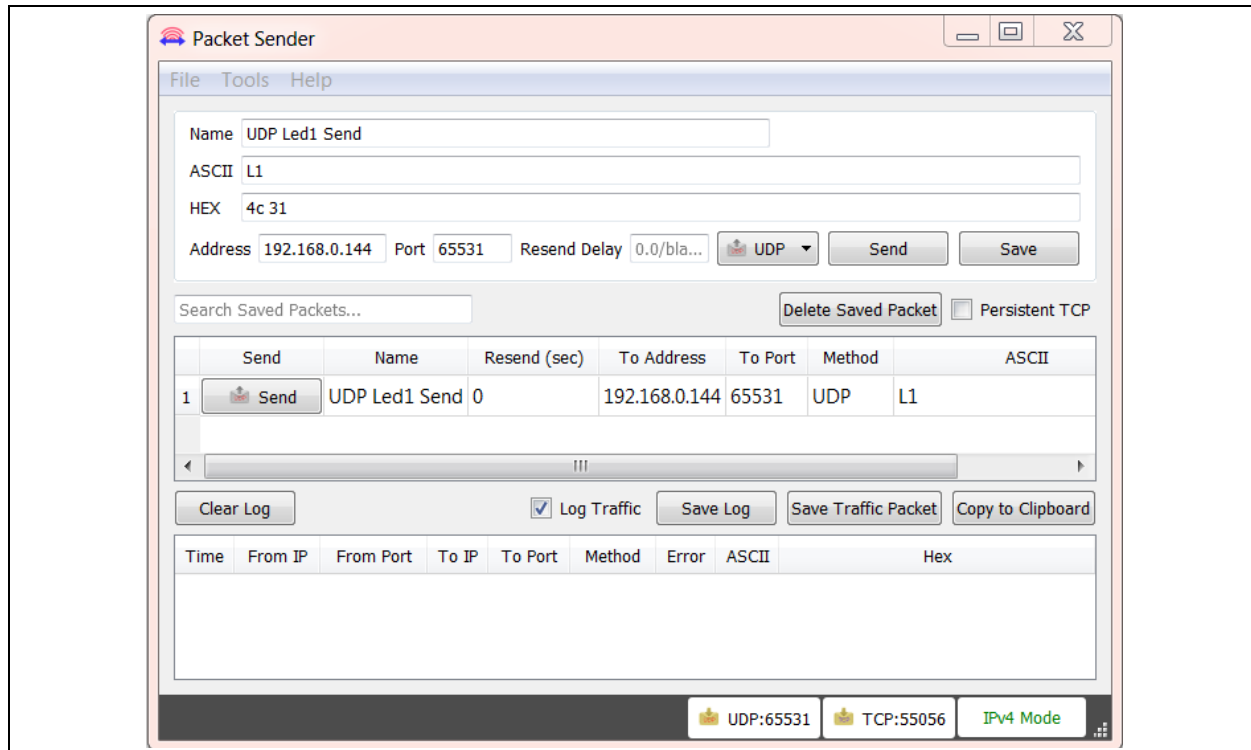


27. In Packet Sender, click on the **Send** button of the saved UDP packets to toggle the LED on the Curiosity board.

28. Repeat steps 25 and 27 to verify the UDP send and UDP receive packets on port 65531.

## UDP Send Implementation

In order to start the UDP packet, the following steps are required:

1. Start the UDP Packet

The function will start the UDPv4 Packet, which starts the IPv4 packet and writes the UDP Header. The UDP Header fields – checksum and Data length – are initially set to '0'.

```
- UDP_Start(uint32_t destIP,
  uint16_t srcPort,
  uint16_t destPort);
```

2. Write the UDP Packet

There are six methods of writing a UDP packet, depending on the size and order of data written.

```
- UDP_WriteBlock(uint8_t* data,
  uint16_t length) – Writes a block of
  data
- UDP_Write8(uint8_t data) – Writes 1
  byte of data
- UDP_Write16(uint16_t data); –
  Writes 2 bytes of data in Host Order
- UDP_Write24(uint32_t data); –
  Writes 3 bytes of data in Host Order
- UDP_Write32(uint32_t data); –
  Writes 4 bytes of data in Host Order
- UDP_String(const char *string)–
  Writes String in Network Order
```

3. Send the UDP Packet

The function will insert the total payload length into the UDP header, compute the checksum of the UDP packet and send the packet on the wire.

```
- UDP_Send();
```

## UDP Receive Implementation

In order to receive the UDP packet, the following steps are required:

1.  Port Handling

In the `udpv4_port_handler_table.c` file, the `UDP_CallBackTable[]` function needs to be updated with the receiving port number and its callback function.

**EXAMPLE 8:     PORT HANDLING**

```
typedef struct
{
    uint16_t portNumber;
    ip_receive_function_ptr callBack;
}udp_handler_t;

const udp_handler_t UDP_CallBackTable [] = \
{
      {portNumber, &callback}
};
```

2.  Receive the UDP Packet

At first, check for the valid checksum. Any UDP packets with invalid checksums are discarded. If the checksum is correct, the function will check for the port number to be matched in `UDP_CallBackTable[]`. If the port number and the corresponding function handler (callback) are valid in the table, the length of UDP payload is passed as parameter to the callback. Any UDP packet with invalid port number will respond with an "ICMP PORT UNREACHABLE" message, if "ICMP Port Unreachable" option is selected in the MCC while configuring the stack; else the UDP packet is discarded.

  - `UDP_Receive(uint16_t udpcksm);`
3.  Read the UDP Packet

There are five methods of reading a UDP packet, depending on the size and order of data.

  – `UDP_ReadBlock(uint8_t* data, uint16_t length)` – Reads a block of data
  – `UDP_Read8()` – Reads 1 byte of data
  – `UDP_Read16()` – Reads 2 bytes of data in Host Order
  – `UDP_Read24()` – Reads 3 bytes of data in Host Order
  – `UDP_Read32()` – Reads 4 bytes of data in Host Order

## Source Code for the UDP Client/Server Implementation

The UDP demo code (source code and prebuilt hex file) is available as download on www.microchip.com.

### EXAMPLE 9: UDP CLIENT IMPLEMENTATION

```
void UDP_DEMO_Send (void)
{
    error_msg ret = ERROR;

    potCurrResult = (ADCC_GetSingleConversion(Pot)/10);
    if(...)
    {
        ...
        ret        =        UDP_Start(udpPacket.destinationAddress,        udpPacket.sourcePortNumber,
udpPacket.destinationPortNumber);

        if(ret = SUCCESS)
        {
            UDP_Write16(potCurrResult);
            UDP_Send();
        }
    }
}
```

### EXAMPLE 10: UDP SERVER IMPLEMENTATION

```
const udp_handler_t UDP_CallBackTable [] = \
{
    {68, DHCP_Handler},
    {65531, UDP_DEMO_Recv}
};

void UDP_DEMO_Recv (int length)
{
    ...
    UDP_ReadBlock(&data,sizeof(data));
    /*
    ...
    Process the Receive Buffer data
    */
}
```

## CONCLUSION

This application note presents some very simple software solutions for implementing a TCP Server, a TCP Client and exchange data over UDP based on the Microchip lightweight TCP/IP stack. The TCP/IP stack provides space efficiency and modular implementation allowing to add network connectivity to embedded systems with limited resources.

## APPENDIX A: REFERENCES

1. *User Datagram Protocol, RFC 768*
2. *Internet Protocol, DARPA Internet Program Protocol Specification, RFC 791*
3. *Internet Control Message Protocol, DARPA Internet Program Protocol Specification, RFC 792*
4. *Transmission Control Protocol, DARPA Internet Program Protocol Specification, RFC 793*
5. *Requirements for Internet Hosts, Communication Layers, RFC 1122*
6. *An Ethernet Address Resolution Protocol or Converting Network Protocol Addresses to 48 bit Ethernet Address for Transmission on Ethernet Hardware, RFC 826*
7. *Domain Names – Implementation and Specification, RFC 1035*
8. *Clarifications to the DNS Specification, RFC 2181*
9. *Service Name and Transport Protocol Port Number Registry* (www.iana.org)

# AN1921

## APPENDIX B:   FLOWCHART FOR TCP CLIENT DEMO

Since the demo containing the TCP/IP stack is very large, the software flowchart shown in Figure B-1 is only for the main application with a focus on the TCP Client routine found in the main.c file.

**FIGURE B-1:    FLOWCHART FOR TCP CLIENT DEMO**
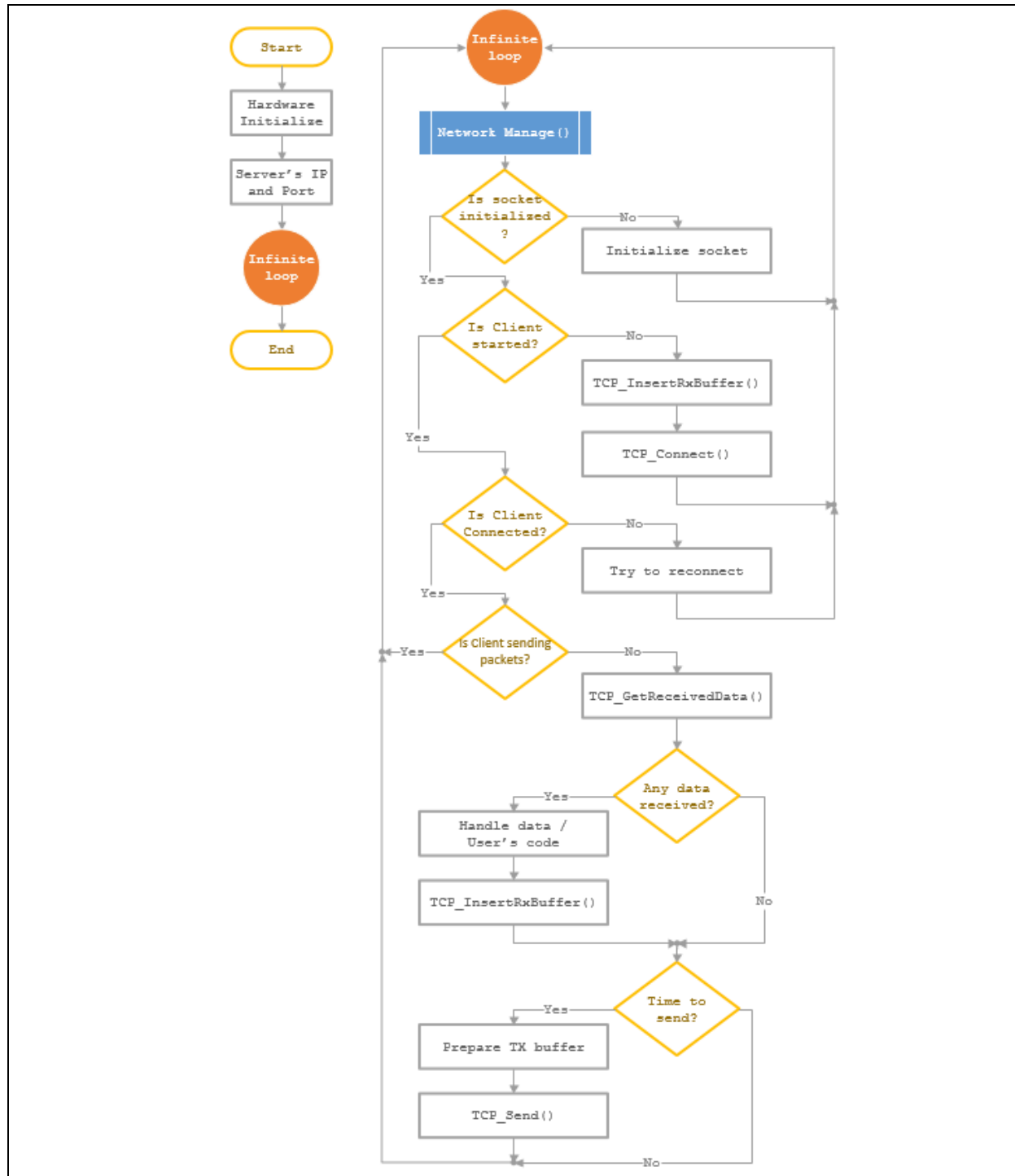
**FIGURE B-2: FLOWCHART FOR TCP SERVER DEMO**

# AN1921
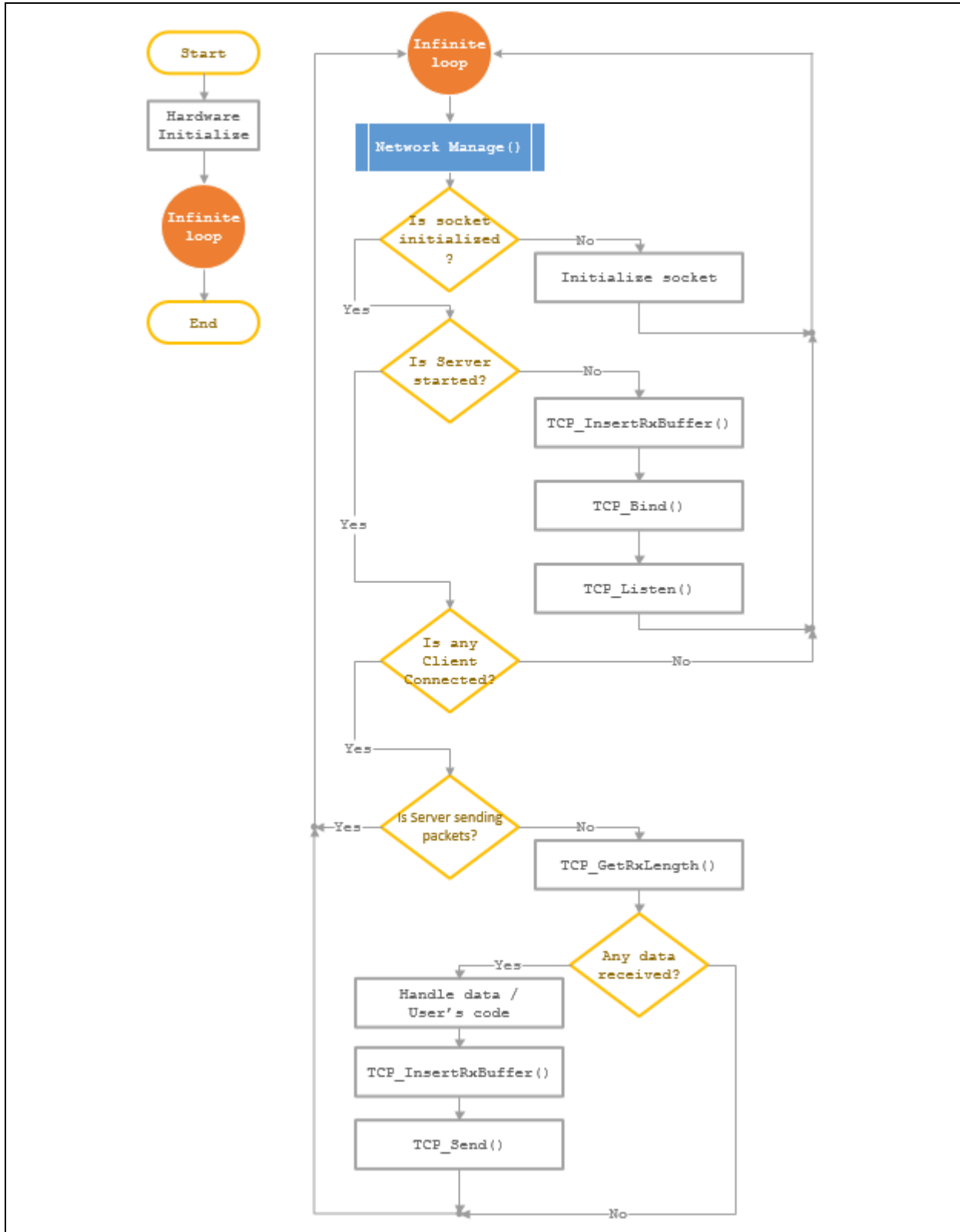
**FIGURE B-3:    FLOWCHART FOR UDP DATA EXCHANGE DEMO**
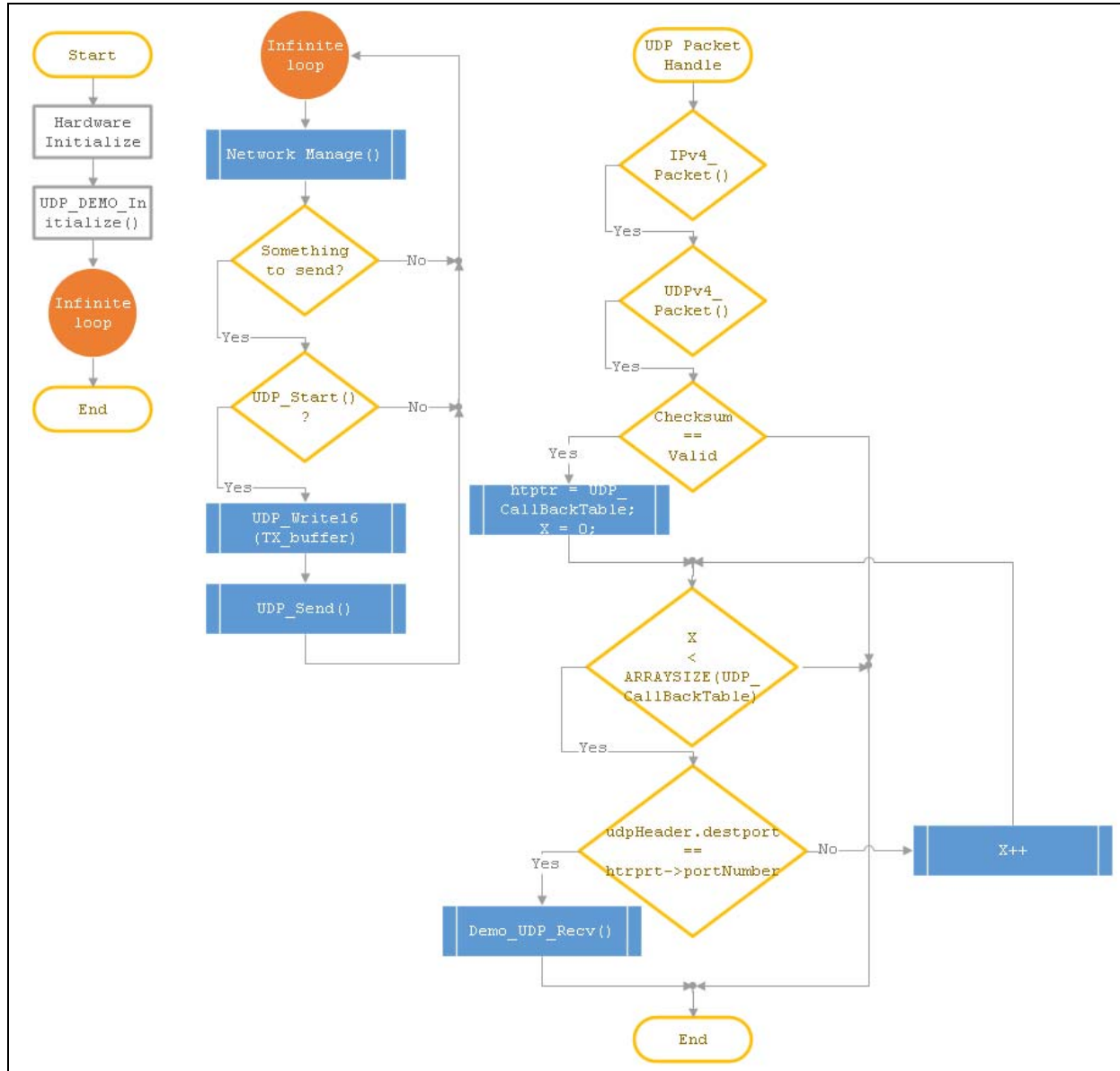
**Note the following details of the code protection feature on Microchip devices:**

• Microchip products meet the specification contained in their particular Microchip Data Sheet.

• Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.

• There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.

• Microchip is willing to work with the customer who is concerned about the integrity of their code.

• Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

**QUALITY MANAGEMENT SYSTEM**
**CERTIFIED BY DNV**
**═ ISO/TS 16949 ═**

# Worldwide Sales and Service

### AMERICAS

**Corporate Office**
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://www.microchip.com/
support
Web Address:
www.microchip.com

**Atlanta**
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

**Austin, TX**
Tel: 512-257-3370

**Boston**
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

**Chicago**
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

**Dallas**
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

**Detroit**
Novi, MI
Tel: 248-848-4000

**Houston, TX**
Tel: 281-894-5983

**Indianapolis**
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453
Tel: 317-536-2380

**Los Angeles**
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608
Tel: 951-273-7800

**Raleigh, NC**
Tel: 919-844-7510

**New York, NY**
Tel: 631-435-6000

**San Jose, CA**
Tel: 408-735-9110
Tel: 408-436-4270

**Canada - Toronto**
Tel: 905-695-1980
Fax: 905-695-2078

### ASIA/PACIFIC

**Asia Pacific Office**
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
**Hong Kong**
Tel: 852-2943-5100
Fax: 852-2401-3431

**Australia - Sydney**
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

**China - Beijing**
Tel: 86-10-8569-7000
Fax: 86-10-8528-2104

**China - Chengdu**
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

**China - Chongqing**
Tel: 86-23-8980-9588
Fax: 86-23-8980-9500

**China - Dongguan**
Tel: 86-769-8702-9880

**China - Guangzhou**
Tel: 86-20-8755-8029

**China - Hangzhou**
Tel: 86-571-8792-8115
Fax: 86-571-8792-8116

**China - Hong Kong SAR**
Tel: 852-2943-5100
Fax: 852-2401-3431

**China - Nanjing**
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

**China - Qingdao**
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

**China - Shanghai**
Tel: 86-21-3326-8000
Fax: 86-21-3326-8021

**China - Shenyang**
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

**China - Shenzhen**
Tel: 86-755-8864-2200
Fax: 86-755-8203-1760

**China - Wuhan**
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

**China - Xian**
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

### ASIA/PACIFIC

**China - Xiamen**
Tel: 86-592-2388138
Fax: 86-592-2388130

**China - Zhuhai**
Tel: 86-756-3210040
Fax: 86-756-3210049

**India - Bangalore**
Tel: 91-80-3090-4444
Fax: 91-80-3090-4123

**India - New Delhi**
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

**India - Pune**
Tel: 91-20-3019-1500

**Japan - Osaka**
Tel: 81-6-6152-7160
Fax: 81-6-6152-9310

**Japan - Tokyo**
Tel: 81-3-6880- 3770
Fax: 81-3-6880-3771

**Korea - Daegu**
Tel: 82-53-744-4301
Fax: 82-53-744-4302

**Korea - Seoul**
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

**Malaysia - Kuala Lumpur**
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

**Malaysia - Penang**
Tel: 60-4-227-8870
Fax: 60-4-227-4068

**Philippines - Manila**
Tel: 63-2-634-9065
Fax: 63-2-634-9069

**Singapore**
Tel: 65-6334-8870
Fax: 65-6334-8850

**Taiwan - Hsin Chu**
Tel: 886-3-5778-366
Fax: 886-3-5770-955

**Taiwan - Kaohsiung**
Tel: 886-7-213-7830

**Taiwan - Taipei**
Tel: 886-2-2508-8600
Fax: 886-2-2508-0102

**Thailand - Bangkok**
Tel: 66-2-694-1351
Fax: 66-2-694-1350

### EUROPE

**Austria - Wels**
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

**Denmark - Copenhagen**
Tel: 45-4450-2828
Fax: 45-4485-2829

**Finland - Espoo**
Tel: 358-9-4520-820

**France - Paris**
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

**France - Saint Cloud**
Tel: 33-1-30-60-70-00

**Germany - Garching**
Tel: 49-8931-9700

**Germany - Haan**
Tel: 49-2129-3766400

**Germany - Heilbronn**
Tel: 49-7131-67-3636

**Germany - Karlsruhe**
Tel: 49-721-625370

**Germany - Munich**
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

**Germany - Rosenheim**
Tel: 49-8031-354-560

**Israel - Ra'anana**
Tel: 972-9-744-7705

**Italy - Milan**
Tel: 39-0331-742611
Fax: 39-0331-466781

**Italy - Padova**
Tel: 39-049-7625286

**Netherlands - Drunen**
Tel: 31-416-690399
Fax: 31-416-690340

**Norway - Trondheim**
Tel: 47-7289-7561

**Poland - Warsaw**
Tel: 48-22-3325737

**Romania - Bucharest**
Tel: 40-21-407-87-50

**Spain - Madrid**
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

**Sweden - Gothenberg**
Tel: 46-31-704-60-40

**Sweden - Stockholm**
Tel: 46-8-5090-4654

**UK - Wokingham**
Tel: 44-118-921-5800
Fax: 44-118-921-5820