2019 MASTERs Conference

23075 IoT6

Simplifying TCP/IP Applications with MPLAB® Harmony

Hands-On

Lab Manual

Instructors:
Martin Ruppert
Raji Shanmugasundaram
Niklas Larsson

Microchip Technology Inc.

MICROCHIP

MPLAB® Harmony TCP/IP Stack

UDP Module API Function List

Socket Management Functions

TCPIP_UDP_ServerOpen	Opens a UDP socket as a server.
TCPIP_UDP_ClientOpen	Opens a UDP socket as a client.
TCPIP_UDP_IsOpened	Determines if a socket was opened.
TCPIP_UDP_IsConnected	Determines if a socket has an established connection.
TCPIP_UDP_Bind	Bind a socket to a local address and port. This function is meant for client sockets. It assigns a specific source address and port for a socket.
TCPIP_UDP_RemoteBind	Bind a socket to a remote address This function is meant for server sockets.
TCPIP_UDP_Close	Closes a UDP socket and frees the handle.
TCPIP_UDP_OptionsGet	Allows getting the options for a socket such as current RX/TX buffer size, etc.
TCPIP_UDP_OptionsSet	Allows setting options to a socket like adjust RX/TX buffer size, etc
TCPIP_UDP_SocketInfoGet	Returns information about a selected UDP socket.
TCPIP_UDP_SocketNetGet	Gets the network interface of an UDP socket
TCPIP_UDP_SocketNetSet	Sets the network interface for an UDP socket
TCPIP_UDP_TxOffsetSet	Moves the pointer within the TX buffer.
TCPIP_UDP_SourceIPAddressSet	Sets the source IP address of a socket
TCPIP_UDP_BcastIPV4AddressSet	Sets the broadcast IP address of a socket Allows an UDP socket to send broadcasts.
TCPIP_UDP_DestinationIPAddressSet	Sets the destination IP address of a socket
TCPIP_UDP_DestinationPortSet	Sets the destination port of a socket
TCPIP_UDP_Disconnect	Disconnects a UDP socket and re-initializes it.
TCPIP_UDP_SignalHandlerDeregister	Deregisters a previously registered UDP socket signal handler.
TCPIP_UDP_SignalHandlerRegister	Registers a UDP socket signal handler.
TCPIP_UDP_Task Standard	TCP/IP stack module task function.

Transmit Data Functions

TCPIP_UDP_PutIsReady	Determines how many bytes can be written to the UDP socket.
TCPIP_UDP_TxPutIsReady	Determines how many bytes can be written to the UDP socket.
TCPIP_UDP_ArrayPut	Writes an array of bytes to the UDP socket.
TCPIP_UDP_StringPut	Writes a null-terminated string to the UDP socket.
TCPIP_UDP_Put	Writes a byte to the UDP socket.
TCPIP_UDP_TxCountGet	Returns the amount of bytes written into the UDP socket.
TCPIP_UDP_Flush	Transmits all pending data in a UDP socket.

Receive Data Transfer Functions

TCPIP_UDP_GetIsReady	Determines how many bytes can be read from the UDP socket.
TCPIP_UDP_ArrayGet	Reads an array of bytes from the UDP socket.
TCPIP_UDP_Get	Reads a byte from the UDP socket.
TCPIP_UDP_RxOffsetSet	Moves the read pointer within the socket RX buffer.
TCPIP_UDP_Discard	Discards any remaining RX data from a UDP socket.

Table of Contents

Introduction

This Lab Manual provides the step by step procedure to complete two labs in the MASTERs 21070 NET1 Class. In Lab 1 you will configure the MPLAB Harmony TCPIP/IP Stack and test basic network connectivity and in Lab 2 you will learn how to implement and test a network application which will utilise a TCP Client and UDP Server in a real world application. Both Labs have specific hardware and software requirements.

Hardware Requirements

The following hardware is required:

- **SAM E70 Xpained Ultra** (Microchip Part Number: DM320113)
    - http://www.microchip.com/DevelopmentTools/ProductDetails.aspx?PartNO=dm320113
- OLED1 Xplained Pro extension kit (Microchip Part Number: ATOLED1-XPRO)
- **Cat 5 Ethernet Patch Cable**
- **USB Male A to USB Male B Micro Cable**

The following hardware is optional:

- **Multimedia Expansion Board II** (Microchip Part Number: DM320005-2)
    - http://www.microchip.com/DevelopmentTools/ProductDetails.aspx?PartNO=dm320005-2
- **MPLAB ICD3 In-Circuit Debugger** (Microchip Part Number: DV164035)
    - http://www.microchip.com/DevelopmentTools/ProductDetails.aspx?PartNO=dv164035
- **USB Male A to USB Male B Cable** (supplied with DV164035)
- **6 Core Modular Cable with RJ11 Connectors** (supplied with DV164035)
- **9V, 500mA Power Supply with 2.5mm Plug**

Software Requirements

The following software is required:

- **Microchip MPLAB X IDE v5.20.04**
    - http://www.microchip.com/mplab
- **Microchip MPLAB XC32 Compiler v2.20**
    - http://www.microchip.com/mplab/compilers
- **Microchip MPLAB Harmony v2.03B**
    - http://www.microchip.com/mplab/mplab-harmony
- **Microchip MPLAB Harmony Configuration (MHC) Tool Plugin v2.0.3.5**
    - MPLAB X Plugin *"com-microchip-mplab-modules-mhc.nbm"* is bundled with MPLAB Harmony under the `microchip\v2_03b\utilities\mhc` folder
- **Tera Term v4.95**
    - https://ttssh2.osdn.jp/index.html.en
- **Packet Sender v5.1 (lab 2 only)**
    - https://packetsender.com
- **JSMN JSON Parser (lab 2 only)**
    - http://www.zserge.com/jsmn.html

Lab 3

Introduction

In many IoT applications, JSON is commonly used as a format in order to transport high-level data in an effective way. It is generally an alternative to XML. Consider the following example of describing a person named Raji-Niklas Ruppert in JSON-format:

```
{
    “firstName” : “Raji-Niklas”,
    “lastName” : “Ruppert”,
    “age” : 30,
    “address” : {
        “streetAddress” : “2355 W Chandler Blvd”,
        “city” : “Chandler”,
        “state” : “AZ”,
        “postalCode” : “85224”,
    }
}
```

Using this format makes it very easy to communicate between applications requiring information about Raji-Niklas Ruppert.

The advantage of using JSON in embedded applications is that because it is easy to read for humans, it is simple to parse and make use of. Due to this, it is commonly used to transmit data between a server and a web application. In this lab we are going to implement an embedded application fetching weather data from a web server. When the application accesses a specific URL specifying a command with a geographic location, the web server will respond by sending the current weather in JSON-format to the web application. The application will be running on our SAME70-boards.

In this lab we will only do very simple parsing (which is one of the strengths using JSON), using standard string operations. There are however more sophisticated parsers which can be used for more robust and complex applications, while still only consuming a very limited footprint.

The weather service used in this lab is https://openweathermap.org/. With OpenWeatherMap, there are several services such as hourly forecast, UV Index, Air pollution and more, all outputting in JSON. With the free account there are limited option to only use the “Current Weather Data” service. With this service you can request the current weather from different geographic locations. Depending on by which method (City ID, ZIP Code, Coordinates etc.) the URL call will be slightly different. A full description of the API can be found here: https://openweathermap.org/current. For this lab we will fetch current weather by city. The following URL for this is:

http://api.openweathermap.org/data/2.5/weather?q=**{CITY}**&APPID=**{API Key}**

The API Key is unique to each user. This is also how OpenWeatherMap tracks how many requests you attempt. The API Key is a 15-byte long hexadecimal string. It can look like this:

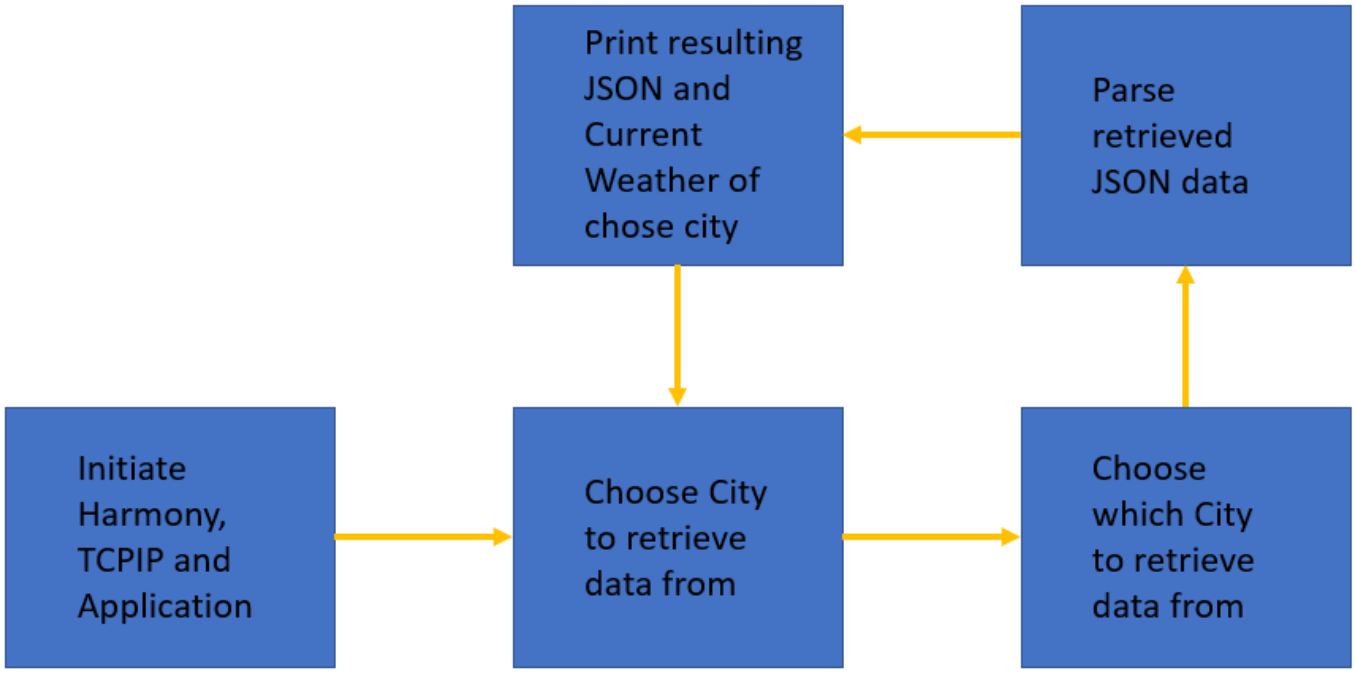
```
ed3da58111974261002c2af4f8e8e81f
```

In most JSON API:s there is also a well defined format specified, which tells you where the different objects and strings are located in the JSON-message. From OpenWeatherMap:

```
{"coord":{"lon":-122.09,"lat":37.39},
```

```
"sys":{"type":3,"id":168940,"message":0.0297,"country":"US","sunrise":1427723751,"sunset":14
27768967},

"weather":[{"id":800,"main":"Clear","description":"Sky is Clear","icon":"01n"}],

"base":"stations",

"main":{"temp":285.68,"humidity":74,"pressure":1016.8,"temp_min":284.82,"temp_max":286.48},

"wind":{"speed":0.96,"deg":285.001},

"clouds":{"all":0},

"dt":1427700245,

"id":0,

"name":"Mountain View",

"cod":200}
```

Application Flow



Lab Outline

- The pre-made template is built from the Harmony example project, tcpip_tcp_client.
- First, we will need to declare the APPID_KEY.
- We will then set the host & port of the remote connection static as we will only connect to OpenWeatherMap.
- After this, we will redirect the user input from the command console to a char* buffer to be used in the application.
- Now we have all information required to build the URL from the introduction.
- When we have connected and requested the data, we need to parse the resulting JSON- string (the whole JSON containing the current weather will be in one string).
- Typically, good practice when you debug JSON-strings is to print the resulting string for you to view with your own eyes that it looks correct.
- At last, redirect the application to go back to accepting user input.

Lab Procedure

1. Start by closing any open projects in MPLAB X IDE.
2. Open a new project and choose lab3 -> Firmware -> sam_e70_xult_freertos.X.
3. Open the file app.c located under source files.
4. Go to (CTRL+F) “TODO A”. Enter the correct APPID_KEY. Either you create your own account on OpenWeatherMap or you take the one written I the class.

```
37  // ************************************************************************************
    //TODO A: Enter the correct APPID_KEY
39  static const char* APPID_KEY = "";
40  char jsonBuffer[1024];
41  char cityBuffer[128];
42
43  // ************************************************************************************
```

5. Now scroll down to “TODO B”, the function APP_Initialize.
6. Set the application to connect to the host api.openweathermap.org and the port to 80. This is set to 80 because this call will be over HTTP.

```
105
106     memset(jsonBuffer, 0, sizeof (jsonBuffer));
107     memset(cityBuffer, 0, sizeof (cityBuffer));
108     //TODO B: Set the application to connect to api.openweasthermap.org and port 80
109     appData.host = "";
110     appData.port =;
111
112 }
113
```

7. Re-direct the user input from APP_URL_BUFFER to the cityBuffer array. This can be done in several ways, but one is to use the built-in C function **snprintf**(char* dest, size_t size, const char *format, …). The first argument is the destination buffer (cityBuffer), the second one is the max size to be copied (128, because that is specified in the declaration) and the formatted input in this scenario is APP_URL_BUFFER. This can be found in “TODO C”.

```
200
    //TODO C: Re-direct the user input to cityBuffer from APP_URL_BUFFER
202 snprintf(,,);
203 SYS_CONSOLE_PRINT("cityBuffer: %s\r\n", cityBuffer);
204
```

8. Scroll down to “TODO D”, the state APP_TCPIP_WAIT_FOR_CONNECTION. In this state we will wait for a connection to be established. Once established we will send a GET command with the full URL in the format specified in the introduction: http://api.openweathermap.org/data/2.5/weather?q=**{CITY}**&APPID=**{API Key}**.

```
277     //TODO D: Build the full URL in pathBuffer.
278     char pathBuffer[128];
279     snprintf(, 128, "data/2.5/weather?q=%s&APPID=%s", , );
280     appData.path = pathBuffer;
281
```

9. Once the request is sent to the server, the application will go into the APP_TCIPIP_WAIT_FOR_RESPONSE state. Once the connection is closed, set the next state to be APP_STATE_JSON_PARSE_RETRIEVED_DATA.

```
298 if (!TCPIP_TCP_IsConnected(appData.socket)) {
299     SYS_CONSOLE_MESSAGE("\r\nConnection Closed\r\n");
300     //TODO E: Set the next state to be APP_STATE_JSON_PARSE_RETRIEVED_DATA
301     appData.state = ;
302     break;
```

10. Now go down in the state APP_STATE_JSON_PARSE_RETRIEVED_DATA. One of the first things we want to do after we have sorted out the JSON-part of the retrieved data is to print the raw JSON-string. This helps us debug & analyse.

```
313             char* resultingJson;
314             char* pos;
315
316             pos = strstr(jsonBuffer, "{\"");
317             *(&resultingJson) = pos;
318
319             //TODO F: Print the resultingJson string
320             SYS_CONSOLE_PRINT("resultingJson: \r\n %s \r\n", );
```

11. In a real application, we would need to first know the format of the JSON message in order to be able to parse it correctly. To make this lab more efficiently, we will do this backwards. If you look on this example piece of API response from OpenWeatherMap found in the introduction section to this lab. Looking at the format from the API, we need to calculate in what position the value of humidity start. The function strstr will cut the resultingJson string at the first occurrence of "humidity". A hint is to look at the other blocks where you parse the temperature, pressure and main weather.

```
322             //Find Humidity
323             char* mainHumidityJson;
324             char* mainHumidtyBuffer;
325
326             //TODO G: Find the correct number of positions to move to the right after humidity
327             pos = strstr(resultingJson, "humidity");
328             *(&mainHumidityJson) = pos + ;
329             mainHumidtyBuffer = strtok(mainHumidityJson, ",");
330
```

12. Once the parsing is done, we wish to print the values of the main weather, pressure, temperature and humidity.

```
335         mainMainWeatherBuffer = strtok(mainMainWeatherJson, "\"");
336
337
338         SYS_CONSOLE_PRINT("\r\nCurrent Weather in %s \r\nHumidity: %s\r\nPressure: %s\r\nTemperature: %2.2f\r\nMain Weather: %s \r\n\r\n",
339                 , , , , );
340
```

13. Now to complete the loop, we want to go back to the APP_TCPIP_WAITING_FOR_COMMAND state once the JSON-parsing and printing is done.

```
359
360             //TODO I: Go back to the APP_TCPIP_WAITING_FOR_COMMAND state to continue application operation
361
362         }
```