In this second part of my IoT series, you will program the mbed with example code that I have developed, and learn how the code works. The code is fairly simple, but its real value is in its reliability. I have worked hard to try to make the wireless connection as reliable, and as fast, as possible.

1 Compiling and Programming

Install the Wifi Shield onto the mbed, and plug the mbed into your computer with a USB cable. The mbed should show up as a USB drive.

You can find the public repository of the mbed code here. Once the page loads, press the "Import into Compiler" button to import the program into your workspace. If you are prompted to select a hardware platform, choose the "NUCLEO-F401RE".

Open the "main.cpp" file, and press the "Compile" button. Once completed, a .bin file will be downloaded. When you are ready to program your mbed, copy this file onto your mbed. However, do not do it yet! We need to change some settings first.

2 Code to Change

In order for the mbed to connect to your network, you will have to tell it ahead of time your network name, and password, etc. Here are the bare minimum lines you have to modify, separated by filename:

2.1 main.cpp changes:

Line 56

```
55  // Set the IoT ID:
56  IoT_ID = 1;
57
```

This value should be unique for each device you have on your network. You can leave it a 1 for your first device

2.2 globals.h changes:

Line 49

```
48  #define WS_PORT     4444
49  #define SERVER_IP "192.168.1.99"
50
```

Change the SERVER_IP value to the IP address of the computer on which the WebSocket server will be running.

2.3 globals.cpp changes:

Lines 43 and 44

```
43  char* wifissid = "HomeNetwork";
44  char* wifipassword = "Pass";
45
```

Change the wifissid and the wifipassword to the appropriate values for your home network.

After completing all of the changes, compile and download the program to the mbed. The mbed will start connecting to the wifi network, and once successful, try to connect to your server. You can open up Tera Term to see the serial output of the microcontroller.

3 Tour of the Code

This first part of the tour will focus on the wireless communication. At the end of this section will be a few notes on other interesting aspects of the code: the messaging system, and the manual configuration of the ADC module.

3.1 Wireless Communication and the WebSocket Client

Looking at the file structure of the IoT_Ex project, you can easily identify the two libraries necessary for the wireless communication: *WiflyInterface* and *WebSocketClient*. These two libraries were developed by other mbed users, and I have modified them slightly to improve their performance. I will not be going into detail on how they work, but on how they are being used for this project.

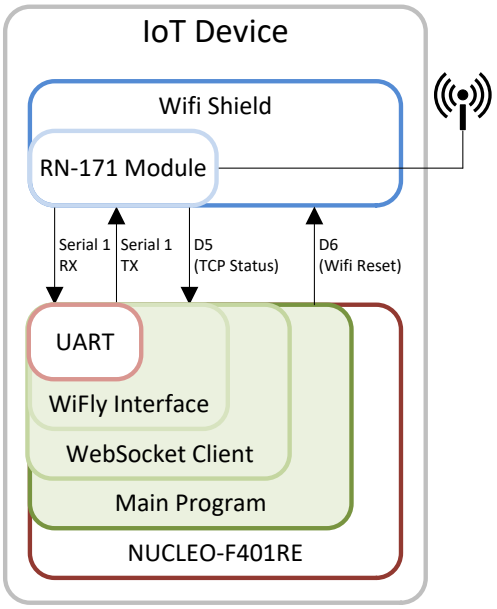

Figure 1 – Schematic diagram of the mbed software and hardware.

As shown in Figure 1, the Main Program interfaces with the "high level" WebSocket Client. Once a connection is established, the main program simply needs to call either `ws.send()` or `ws.readmsg()` to send or receive data from the server. After anyone of those function calls, the data goes through the WiFly Interface, then the UART module, then the physical wires to the RN-171 module. All of these lower level layers of code and hardware ensure everything is sent at the right time, and errors do not occur in the transmission, etc.

The next two subsections will parse through some of the important parts of main.cpp and globals.cpp that enable the wireless communication using the WebSocket Client and the WiFly Interface. I will also point out the areas of the code which you can modify to send and receive different types of data.

3.1.1 main.cpp

```
49  // Main Loop!
50  int main() {
```

The `int main(){…}` function is where most of the action takes place, so let's take some time to parse through it. The first part of the function has some standard declarations, and variable configurations. Lines 74 and 78 is some of the first communications code we find:

```
72    // Connect to the wifi network. It will basically get stuck here until it
73    // connects to the network.
74    SetupNetwork(5000);
75
```

```
76   // Configure the baud rate of the wifi shield:
77   // This will make our wireless transmissions much faster.
78   ws.setBaud(115200);
```

The function `SetupNetwork()` connects to the wireless network. It will try 5000 times before it exits. The function itself is explicity defined in globals.cpp. Once connected to the network, we know the communication between the microcontroller and the RN-171 module is working, so we increase the baud rate of the UART interface between the microcontroller and the RN-171 module to `115200` with a call to `ws.setBaud()` on line 78.

`ws` is a global instance of the WebSocketClient class, and is declared in globals.cpp.

Next, the microcontroller will try to connect to the WebSocket server on Line 87:

```
82   if(IotStatus.CheckFlag(SF_WIRELESSCONNECTED)){
...      ...
87       if(ws.connect()){
88           // Set a status flag:
89           INFO("Connected.");
90           IotStatus.SetFlag(SF_SERVERCONNECTED);
91       }else{
92           // We could not connect right now..
93           IotStatus.ClearFlag(SF_SERVERCONNECTED);
94           INFO("Could not connect to server, will try again later.");
95           ReconnectAttempts++;
96       }
97   }
```

If it cannot connect, it will try again later, during a call to `SendNetworkData()`.

Finally, we get to the infinite loop. This is where the microcontroller spends all of its processing time:

```
103  // Inifinite main loop:
104  while(1) {
105
106      // Process the wifi command:
107      if(wifi_cmd > NO_WIFI_CMD){
108          // Modify the desired variable:
109          ModifyVariable(wifi_cmd, wifi_data);
110          // Reset the command:
111          wifi_cmd = NO_WIFI_CMD;
112      }
113
114      // Check for new wifi data:
115      if((wifi_cmd == NO_WIFI_CMD)){
116          ReceiveNetworkData(&wifi_cmd, &wifi_data);
117      }
118
119      // Send the network data every 3 seconds:
120      if(DisplayTimer.read()>(3.0f)){
...           ...
133          // Send data over network:
134          SendNetworkData();
...           ...
```

```
139             // Reset the timer:
140             DisplayTimer.reset();
141
142         }
143     } // while(1)
```

On Line 116, the microcontroller checks for new data from the WebSocket server. The data is parsed in the `ReadNetworkData()` function into the two variables: `wifi_cmd` and `wifi_data`. Within this function, is the function call to `ws.readmsg()`, mentioned earlier. In effect, the microcontroller is polling the websocket for any new data. You could also implement this as an interrupt, but I leave that as an exercise for the reader. Any received data is handled by the `ModifyVariable()` function on Line 109.

The `if(){…}` statement on Line 120 ensures `SendNetworkData()` is called once every 3 seconds. `SendNetworkData()` sends a comma separated string to the WebSocket Server of the form of:

```
IoT_ID, SendCounter, TempSensor
```

All of the functions listed above are found in globals.cpp.

That's it!

```
144 } // main()
```

3.1.2 globals.cpp

globals.cpp contains global variable and function definitions. We will be looking at a few of the functions that were referenced in the previous subsection. First, `SendNetworkData()`:

```
102 void SendNetworkData(void){
103     char msg_buffer[CHARMSGBUFF];
...     ...
106     if(IotStatus.CheckFlag(SF_SERVERCONNECTED)){
107         sprintf(msg_buffer, "%d,%d,%.5f", IoT_ID, SendCounter,TempSensor);
108         INFO("Sending: %s", msg_buffer);    // When this line ...
109         intresult = ws.send(msg_buffer);
110     }else{
111         intresult = -1;
112     }
113     DBG("intresult: %d", intresult);
114
115     if(intresult < 0){
...         ...
154     }
155
156     return;
157 }
```

During execution, `SendNetworkData()` first checks to see if the microcontroller is connected to the server – that's the `if(){…}` statement on Line 106. If it is connected, data is packed into the message

buffer, and then sent using the `ws.send()` command on Line 109. The result of this function call will be an integer greater than zero if it is a success. If it returns a -1, then the connection to the server is lost. If you want to change what the microcontroller is sending to the server, simply change Line 107 to reflect your changes. Keep in mind that `msg_buffer` should be large enough to hold everything, which means changing its declaration in Line 103. If it is not, you will get unpredictable results.

The second `if(){…}` statement in the code block above on Line 115, deals with the case when the connection to the server is lost. There are several 'troubleshooting' steps the microcontroller will take to try to re-establish the lost server connection. Eventually if nothing works, it will reset the Wifi Shield and start from scratch.

The next function we will examine is ReceiveNetworkData():

```
159  void ReceiveNetworkData(unsigned int * wifi_cmd, float * value){
160      char msg_buffer[CHARMSGBUFF];
161      char msg_buffer2[CHARMSGBUFF];
162      int resp;
163      if(IotStatus.CheckFlag(SF_SERVERCONNECTED)){
164          // Check for data on the websocket:
165          resp = ws.readmsg(msg_buffer);
166          if(resp == 1){
167              INFO("Received: %s", msg_buffer);
168              sscanf(msg_buffer, "%d,%s", wifi_cmd, msg_buffer2);
169              if(*wifi_cmd == CV_LED_WIFI_CMD){
170                  // Get one more value:
171                  sscanf(msg_buffer2, "%f", value);
172              }
173          }else if(resp == -1){
174              // Connection to the server is lost:
175              IotStatus.ClearFlag(SF_SERVERCONNECTED);
176          }else{
177              //DBG("Did not receive anything :(\n\r");
178              *wifi_cmd = NO_WIFI_CMD;
179              *value = 0.0f;
180          }
181      }
182      return;
183  }
```

Just like sending data, the `ReceiveNetworkData()` function will check for a server connection, Line 163. If it is established, it will attempt to read from the WebSocket, Line 165. If you are going to change the messages that are being sent to the microcontrollers, make sure your message buffer is large enough, changing Line 160 and 161 if necessary. In this implementation, I decided to break up the parsing of the message into two stages. First, I check for a command on Line 169, putting the rest of the message 'aside' in `msg_buffer2`. Once I know what type of data I am dealing with, I look for a float value (Line 171).

If you want to send different types of data to the microcontrollers, you will need to modify Lines 167 to 171.

The last important function to look at is ModifyVariable():

```
185  void ModifyVariable(unsigned int wifi_var, float wifi_data){
186      // modifies something in the SCS Controller:
187      switch(wifi_var){
188          case CV_LED:
189              if(wifi_data > 0){
190                  Led = 1;
191              }else{
192                  Led = 0;
193              }
194              break;
195
196          default:
197              break;
198      }
199      return;
200  }
```

This function simply modifies the variable that is sent from the WebSocket Server. As you can see the only variable that can be modified right now is the `Led` digital output. Now that you see the code, you can turn on and off the led with numbers other than 0 and 1.

The easiest way to add another variable to modify would be here. Simply define another variable, like `#define CV_GAIN 2`, and add a case statement at Line 195.

3.2 Messaging System

One of the improvements I made to the WebSocketClient and WiflyInterface libraries was to their messaging systems. The messaging system is used to output a string to the serial port for mostly debugging and status purposes. These strings can be read by a terminal program like Tera Term. The same messaging system in those libraries is all over my code at the start of the .cpp files. Here is a snippet from main.cpp:

```
40
41   //#define DEBUG
42   #define INFOMESSAGES
43   #define WARNMESSAGES
44   #define ERRMESSAGES
45   #define FUNCNAME "IoT"
46   #include "messages.h"
47
```

This system allows me to have 4 levels of messages that can be turned on or off for each file very easily. In the above snippet, any debug messages are suppressed, but Info messages, warning messages, and error messages will be displayed in the terminal. Take a look at messages.h, which can be found in the WiflyInterface library, for how it works.

The declaration of FUNCNAME allows you to easily determine which file generated the message. This little feature was added my Malcolm McCulloch.

Each level of messaging can be used just like a `printf()` statement. For the four levels of messages we have:

Debug Messages:	`DBG("This is a debug message.");`
Info Messages:	`INFO("This is an info message with the number: %d", 2);`
Warning Messages:	`WARN("This is a warning message with a float: %.3f", 3.14159f);`
Error Messages:	`ERR("Lots: %d, %s, %.5f", 3, "Hello World", 4.245423423);`

3.3 Internal Temperature Sensing

This example uses the internal temperature sensor of the NUCLEO-F401RE microcontroller to output an analog signal. In order to sample this signal, you have to sample channel 16 of the ADC. Unfortunately, channel 16 is not associated with any pin, thus the mbed declaration of AnalogIn() would not work. As a consequence, I had to manually configure the ADC module, something that may be of interest to a future project. Nothing accomplished here is magic, and just requires a bit of patience as you navigate through the 800+ pages of the STM32F401RE datasheet. (The STM32F401RE is the microcontroller on the NUCLEO-F401RE mbed board.)

Manually configuring a module on a microcontroller involves "register-level programming". This is almost the lowest level programming you can go when working with microcontrollers. Registers themselves refer to specific memory locations within a microcontroller's memory which are dedicated to controlling its hardware, amongst other things. In a 32 bit microcontroller, each register has 32 bits. If you look at ADC.cpp, you will see a function called `ConfigureADC()`. This function is called at the start of the `int main()` function, and tells the microcontroller how to use the ADC hardware. As you can see, some very basic commands need to be executed first: turning on the clock to the ADC module (Line 24) and turning on the power to the ADC module (Line 28):

```
15  void ConfigureADC(void){
16
17      unsigned int value;
18
19      // ensure power is turned on
20      // Grabbed from lines 54-57 of analogin_api.c
21      // This turns on the clock to Ports A, B, and C
22      RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN | RCC_AHB1ENR_GPIOBEN |...
23      // This turns on the clock to the ADC:
24      RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;
25
26
27      // Turn on the ADC:
28      value = ADC_CR2_ADON;
29      ADC1->CR2 = value;
30      wait_us(100);
31
32      // Set the EOC flag at the end of every regular conversion:
33      ADC1->CR2 |= ADC_CR2_EOCS;
34
35      // Turn on the internal temperature sensor:
36      ADC->CCR |= ADC_CCR_TSVREFE;
37
38      // Set the first (and only channel) to convert to CH16, the internal...
39      ADC1->SQR3 |= ADC_SQR3_SQ1_4;
40
```

```
41        // Set the sample numbers (making this bigger samples more slowly):
42        ADC1->SMPR2 = ADC_SMPR1_SMP16_1 | ADC_SMPR1_SMP16_2; // Set for 144 ADC...
43
44
45        INFO("ADC configuration complete!");
46        DBG("ADC Registers:");
47        DBG("The SR Register reads: %d", ADC1->SR);
48        DBG("The CR1 Register reads: %d", ADC1->CR1);
49        DBG("The CR2 Register reads: %d", ADC1->CR2);
50        DBG("The JSQR Register reads: %d", ADC1->JSQR);
51
52        return;
53    }
```

In this implementation I am not using any interrupts to drive the sampling, which is very basic. The sampling occurs in the `while(1)` loop of main.cpp:

```
121       // Sample the internal temperature sensor:
122       STARTADCCONVERSION;
123       // Wait for the conversion to complete:
124       while(!ADCCONVERSIONCOMPLETE);
125       // Save the raw value from the ADC:
126       ADCRaw = ADC1->DR;
127       // Calculate the temperature using information from the datasheet:
128       TempSensor = ((((float)ADCRaw)/ADC_MAX)*IT_VMAX - IT_V25)/IT_AVG_SLOPE + 25.0f;
129       // Output the result:
130       DBG("TempSensor = %.5f", TempSensor);
131       DBG("ADC1->DR = %d", ADCRaw);
```

If you look at the macro definition of `STARTADCCONVERSION`, you will see it is setting a single bit in one of the ADC registers. By setting this bit, this triggers the ADC to start a conversion. Likewise, there is another bit, in another register, which is set when a conversion is complete. This is why the `while()` statement on Line 124 exists. The code will wait here until the conversion is complete. After the ADC conversion is complete, the new value is ready to be read from another one of the ADC registers: `ADC1->DR`.

All of the register 'names' (`ADC1->DR`, `ADC1->CR1`, etc…) are macros defined in mbed.h.

Since ADC conversions produce a number between 0 and 4095, you need to convert that number into something more representative of the data you are measuring. In this case we are measuring temperature, and the conversion is done on Line 128. The values in the formula in Line 128 were obtained from the device's datasheet.

Finally, it should be noted that the internal temperature sensor in the STM32F401RE is absolutely useless for absolute temperature measurement. (Something they clearly state in the datasheet as well). Therefore, don't be surprised if you see some random temperatures between microcontrollers in the same room.