

# 1. PROJECT REQUIREMENTS

## 1.1 Description of the Problem

There is a need to remotely monitor environmental data for a delicate asset located in storage. A sensor system will be attached to the asset. The sensor system will be able to monitor the ambient temperature of the room as well as the orientation of the asset. The sensor system will send out temperature and orientation data to a remote monitoring system at regular intervals. In turn, this data will be displayed and logged by the remote monitoring system.

## 1.2 Description of the Input(s)

### 1.2.1 I2C Temperature Sensor (TMP006)

The temperature sensor will measure the ambient temperature of the room at regular intervals.

### 1.2.2 I2C Accelerometer Sensor (BMA222)

The 3-axis accelerometer will measure the orientation of the asset. Its data can reveal whether the asset was tilted or mishandled while in storage.

### 1.2.3 UART

The UART will be used to display activity from the CC3220SF board. It will also provide a simple command interface and a menu for testing/debugging individual program features or potentially to even override the default WIFI configuration settings.

### 1.2.4 WIFI / Time Server

A WIFI connection will be used to connect to an external Time Server in order to initialize the internal clock of the sensor system. This will provide accurate timestamps corresponding to the temperature and orientation data.

The WIFI connection will require the following to be designated:

- SSID (i.e. WIFI network name)
- WIFI security configuration (i.e. SL\_WLAN\_SEC\_TYPE\_WPA\_WPA2 or unsecured SL\_WLAN\_SEC\_TYPE\_OPEN)
- Password (i.e. none if unsecured)

In the event that it is necessary to manually designate a Network Time Protocol (NTP) server:

- Host name (i.e. us.pool.ntp.org)
- Port (i.e. 123)

## 1.3 Description of the Output(s)

### 1.3.1 Red LED (D10)

The Red LED will serve as an indication of whether or not the sensor system is connected to WIFI, and if it is sending data to the remote monitoring system.

Sensor System Status	Red LED State
WIFI not connected	Not illuminated
WIFI connected	Illuminated
WIFI connected + remote monitoring	Blinking

### 1.3.2 UART

The UART will provide an optional interface that displays information such as sensor system initialization, WIFI connection status, and task activity.

### 1.3.3 WIFI / TCP Client

A WIFI connection will allow for the creation of a TCP Client. This TCP Client will connect to a remote server with the following to be designated:

- Server IP Address (i.e. 192.168.2.250)
- Port (i.e. 5001)

## 1.4 Processing to Convert the Input to the Output

### 1.4.1 Temperature Sensor

Measurements are received over the I2C bus, and are characterized by two bytes which require interpretation for Celsius or Fahrenheit.

### 1.4.2 Accelerometer Data

Measurements are received over the I2C bus, and are characterized by three signed bytes, and require interpretation for device orientation. Depending on how the sensor system is attached to the asset, 1 g of acceleration may be observed along one axis, while the other perpendicular axes may each report nearly 0 g.

### 1.4.3 Time Server Data

NTP time uses an epoch of 1900, which may need to be adjusted for a Unix time system with an epoch of 1970. NTP time is characterized by 4 bytes corresponding to the number of seconds since 1900, and then another 4 bytes for a fractional value of seconds. If necessary, a timer can be manually configured increment the time after it has been configured.

### 1.4.4 TCP Client Data

The timestamp and each measurement will be converted into ASCII values and then periodically transmitted using an (approximately) 80 byte message with Big Endian. The remote monitoring system running a TCP Server will simply need to display the ASCII data. Care will be taken to limit the precision of each measurement.

TCP Client ASCII Message Format					
Timestamp	Temp °C	Temp °F	Acc X	Acc Y	Acc Z

It would require fewer bytes to transmit this information if the data retained its original numerical representation. However, for this exercise the remote monitoring system will be designed to be as simple as possible.

## 1.5 Performance Measurement(s)

- 1.5.1 **Sensor system must display message data with UART command if remote monitoring system is not available**
- 1.5.2 **Sensor system must display correct Red LED state**
- 1.5.3 **Sensor system must attempt to setup automatically without user input.**
- 1.5.4 **Sensor system must not transmit corrupted message to the remote monitoring system**

## 2. PROJECT DESIGN

### 2.1 Description of the program structure

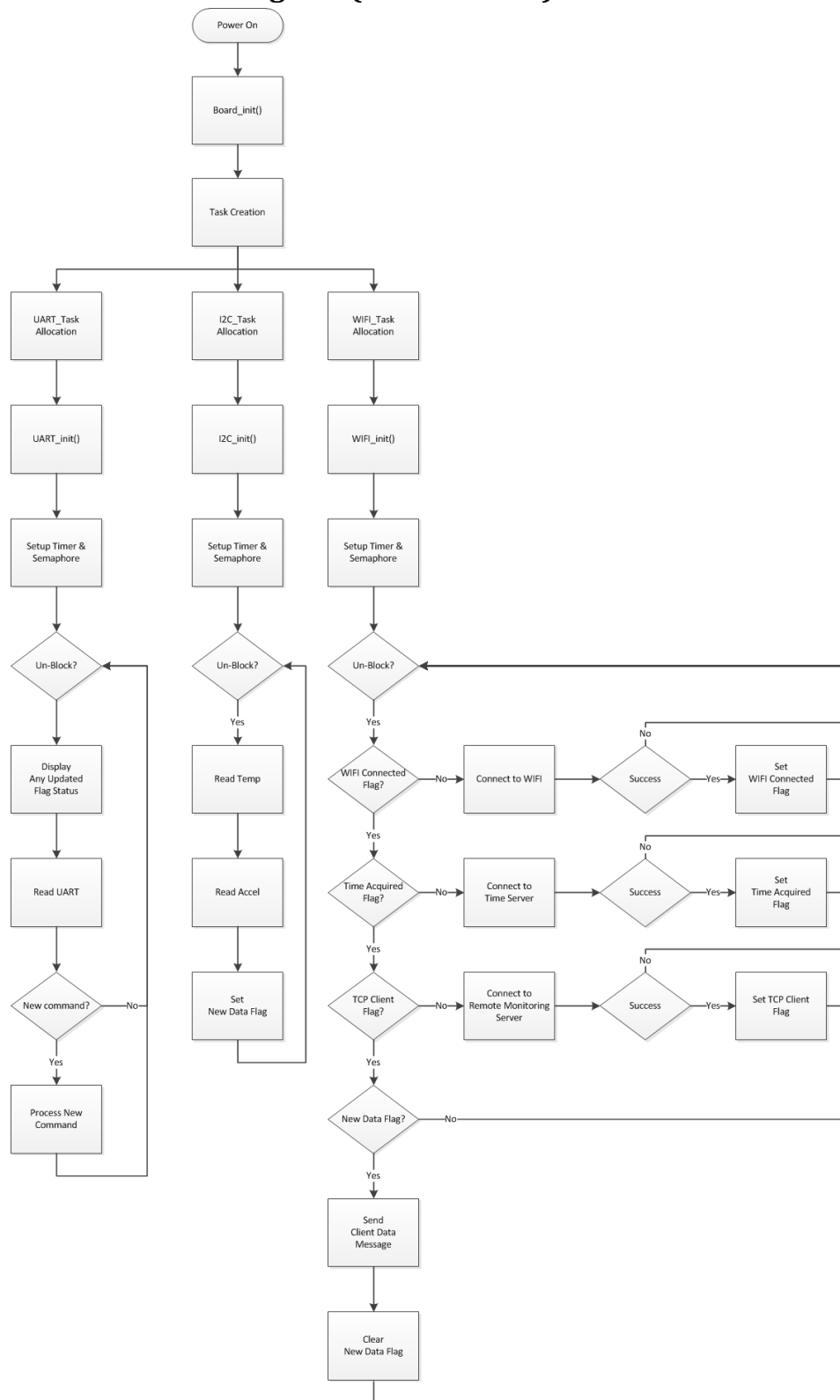
#### 2.1.1 FreeRTOS Implementation

The FreeRTOS implementation of this project will rely upon lessons learned while examining the different example projects. In general, several tasks to carry out hardware initialization and perform actions at prescribed intervals. Timers and semaphores will be used to control when tasks block or un-block. Flags will be used to signal when one task has completed an action that another task may be waiting upon in order to carry out further actions.

#### 2.1.2 Non-RTOS Implementation

The Non-RTOS implementation of this project will use a traditional state-machine architecture. A switch case structure will define individual states. It is expected that the initialization of the UART, I2C bus, and the WIFI connection will all have to be sequentially completed before the process of reading sensors begins. If UART or TCP communication are traditional blocking processes (like getchar), then appropriate timeouts must be implemented.

## 2.2 State transition diagram (or flow chart)



## **2.3 Work to be performed in each state (or flowchart step)**

### **2.3.1 Board\_init()**

Standard CC3220 device initialization

### **2.3.2 Task Creation**

Calls to xTaskCreate and vTaskStartScheduler()

### **2.3.3 UART/I2C/WIFI Task Allocation**

Memory allocation and declaration of variables

### **2.3.4 UART/I2C/WIFI\_init()**

Calls to hardware initialization APIs

### **2.3.5 Setup Timer & Semaphore**

Setup timers and create (counting) semaphores. Tasks attempt to take semaphores and block until semaphores are given by timers. Fewer semaphores and timers may be used depending on task execution based upon task priority.

### **2.3.6 Un-Block?**

Tasks resume execution when un-blocked

### **2.3.7 Read UART/Temp/Accel**

Read data from hardware

### **2.3.8 Connect to WIFI sequence**

Task will check a flag if it already connected to WIFI. If not, the task will attempt to connect to the specified WIFI ID. If successful, a corresponding flag is set.

### **2.3.9 Connect to time server sequence**

Task will check a flag if it already connected to a time server. If not, the task will attempt to connect to the time server using an API. If successful, a corresponding flag is set.

### **2.3.10 Connect to remote monitoring server sequence**

Task will check a flag if it is already connected to the remote monitoring server. If not, the task will attempt to connect to the specified WIFI ID. If successful, a corresponding flag is set.

### **2.3.11 Set New Data Flag**

The I2C task will set a flag indicating that new measurements are available

### **2.3.12 Clear New Data Flag**

The WIFI task will clear the flag indicating that the new measurements were sent to the remote monitoring system

## 3. PROJECT BUILD

### 3.1 Hardware Drivers

The special CCS “syscfg” settings have been surveyed for many of the example projects, and then cross-referenced. I believe that I have identified which selections will be needed.

### 3.2 UART

```

COM7 - Tera Term VT
File Edit Setup Control Window Help

Week 6 Lab - Michael Horne
Console (h for help)
> h
Valid Commands
-----
h: help
m: display current memory heap
c: create task 3 with xTaskCreate()
d: delete task 3 with vTaskDelete()
j: create task 4 with xTaskCreate()
k: delete task 4 with vTaskDelete()
q: quit and shutdown UART
s: clear the screen
t: display current temperature
a: display accelerometer values
p: test UART_print() and vPortFree()
> m
configTOTAL_HEAP_SIZE: 32768
xPortGetFreeHeapSize: 30352
xPortGetMinimumEverFreeHeapSize: 30352
> p
0123456789ABCDEFGHIJKLMNQRSTUUVWXYZ

```

Printing to the UART will need to be standardized by selecting of one of the methods found in the different examples. Some methods print directly to the UART, while others utilize a display driver that selects the UART. WIFI is expected to be much more verbose than other efforts, and could definitely benefit from a good reliable implementation.

### 3.3 Temperature Sensor

```

COM7 - Tera Term VT
File Edit Setup Control Window Hel

j: create task 4 with xTaskCreate()
k: delete task 4 with vTaskDelete()
q: quit and shutdown UART
s: clear the screen
t: display current temperature
a: display accelerometer values
p: test UART_print() and vPortFree()
> t
Current temp = 22C (71F)
> t
Current temp = 22C (71F)
> t
Current temp = 22C (71F)
> t
Current temp = 22C (71F)
> t
Current temp = 22C (72F)
> t
Current temp = 22C (72F)
> t
Current temp = 22C (72F)
> t
Current temp = 22C (73F)
>

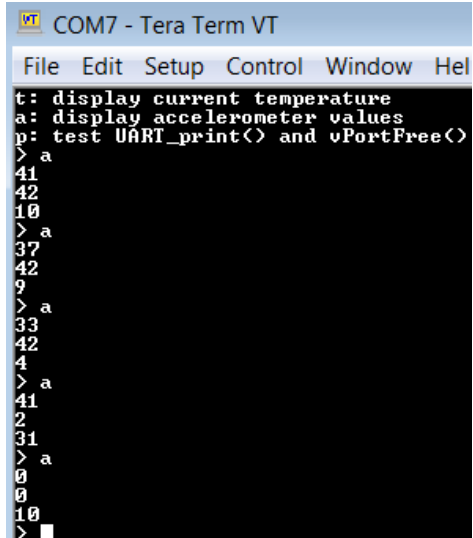
```

Validation of the temperature accuracy is pending. Currently, values are reported as integers, but floating point values may be introduced.

### 3.4 Red LED (D10)

Currently, the red LED blinks and is controlled by the I2C task. It will be controlled by the WIFI task once implemented.

### 3.5 Accelerometer



```
COM7 - Tera Term VT
File Edit Setup Control Window Hel
t: display current temperature
a: display accelerometer values
p: test UART_print() and vPortFree()
> a
41
42
10
> a
37
42
9
> a
33
42
4
> a
41
2
31
> a
0
0
10
>
```

Early results from testing from an example suggest values are reported on a range of 0 to 42. With about 10 being interpreted as 1 g, and about 30 (when the device is inverted) with -1g. Each axis appears to respond along this range. I see an operation where the received values are divided by 6. If the bytes were unsigned, and had a value of 255, dividing by 6 would yield 42.5.

### 3.6 WIFI / Time Server

It might be better to wait to initiate the WIFI connection, and rely on the UART menu to perform setup. Unless the recipient of this program changes macro definitions, WIFI would never be able to connect anyway.

## 4. PROJECT EVALUATE

### 4.1 Project Performance

Evaluate your project according to the performance measurements.

#### 4.1.1 High-Level Project Requirements

For this project, there were many high-level requirements to fulfill:

- Setup UART for basic I/O capabilities and formatted printing
- Measure ambient temperature with I2C temperature sensor
- Measure device orientation with I2C accelerometer sensor
- Connect to WIFI
- Acquire time from external time source
- Setup TCP Client on device
- Setup external TCP Server
- Use TCP Client to send measurements to external TCP server
- Setup Red LED to indicate WIFI/TCP Client Status

Overall, the FreeRTOS version of this project is able to fulfill these requirements. However, automating many of these features proved difficult. For several reasons, the implementation of the WIFI and networking functionality required much more user interaction that was originally intended.

#### 4.1.2 UART Task

The UART implementation in this project was originally derived from a SDK example project (portableNative). This example included a simple command console that would respond to a single character input. However, output capabilities were very limited, and formatted printing (i.e. `printf()` behavior) was limited. I definitely was interested in formatted printing, as I have seen how much of a benefit there is in getting detailed information in desktop environment. Granted, there are alternatives with debugging methods such as using watch expressions and looking at memory locations.

##### 4.1.2.1 Reading and Writing from the UART

This implementation of a UART task has a low priority (1) and it blocks when calling `UART_read()` while waiting for a keystroke. This permits the FreeRTOS idle task to run in the background, and of course for higher priority tasks to run as needed. This worked well, but I did notice occasional fragmented messages while running multiple tasks. Based on a recommendation from another student, calls to `UART_write()` were replaced with `UART_writePolling()` in order to ensure that complete messages were printed and to avoid confusion. Although, I do believe that using `UART_writePolling()` interferes with the response times for other tasks.

##### 4.1.2.2 Command Console

In regard to the command console, it is based upon a switch case statement with cases pertaining to individual characters. The command console was expanded to display information



from other tasks and particularly to control activities of the WIFI task. It also includes utility commands to display heap memory and task stack usage. It was often used to help develop individual functions before they were implemented elsewhere. The valid command set can be displayed by simply typing “h”.

```
Remote Monitoring System Project - FreeRTOS - Michael Horne
UART Console <h for help>
> h
Valid Commands
-----
h: help
t: Display temperature data
a: Display accelerometer data
1: Display WIFI settings
2: Change WIFI settings3: Connect to WIFI
4: Update device time
5: Display device time <UTC>
6: Display TCP Client message
7: Test TCP Client
8: Close TCP Client socket
9: Send TCP Client message
-----
m: Display current memory heap
s: Display task stack usage
p: Test UART_printf<>
>
```

#### 4.1.2.3 Formatted Printing

I had seen other lab assignments and example projects that utilized a display driver that supported formatted printing, and had syntax similar the following:

```
Display_printf(display, 0, 0, "Received, value = %d\n", var);
```

I considered migrating to this type of implementation until I located a suitable alternative. Another SDK example project (local\_time) was similar to the existing implementation I was using, but included a function to allow for formatted printing. However, this function made calls to malloc(), realloc(), and free(), which I had to replace with the pvPortMalloc() and vPortFree().

At this point, I found that I neglected to increase the task stack size to accommodate calling pvPortMalloc(), and I encountered stack overflow errors. I learned of the FreeRTOS function uxTaskGetStackHighWaterMark(), which helped me understand the stack usage for each task. Upon resolving this complication, formatted printing was possible.

Regarding the TCP Client ASCII Message Format described in the design phase, it was implemented with the formatted printing function. Essentially, the string that is formatted and then written to the UART is copied to another character buffer that is sent by the TCP Client. Technically, it would be possible to send every string written to the UART to the external TCP server once it is connected. However, a flag was created that only allows the TCP Client ASCII Message to be copied. Granted, it is possible to just have two similar formatted printing functions or to have something akin to a stdout argument.

#### 4.1.2.4 Improved User Input and WIFI Command Queue

While working on the WIFI features, it occurred to me that it would be a great benefit to the user if the WIFI name could be changed at runtime. Otherwise, the user would need to change the #define, re-compile the program, and re-load the program. To support this capability,

another function (getString) was derived from a SDK example project (local\_time). This function could be called by the UART task, and it accepts multiple character inputs comprising a string. As an example, a string can be relayed to the WIFI task for processing by using a queue.

A queue was setup to originally control activities of the WIFI task based on selections from the command console running on the UART task. The queue was later reworked to send structures containing a command, a parameter value, and a character pointer. The inclusion of the parameter value, and a character pointer allowed related activities, such as changing the WIFI name and password, to be represented by a single command. The queue can also allow for a series of commands to be issued to the WIFI task.

### 4.1.3 I2C Task

The I2C temperature sensor implementation in this project was originally derived from a SDK example project (portableNative). Similarly, the I2C accelerometer sensor implementation in this project was derived from another SDK example project (out\_of\_box). The I2C task relies on a timer in order to run at 1 second intervals. With consideration of a Consumer/Producer system architecture, the I2C task is considered to be the Producer.

#### 4.1.3.1 Temperature Measurement

The CC3220SF development board can have different temperature sensors, which themselves have different I2C bus addresses. According to the datasheet, Rev A and B use the TMP006/B temperature sensor, while Rev C uses TMP116. My specific CC3220SF development board is Rev A, and includes the TMP006/B temperature sensor. Using the supplied TI drivers and the example project, it is possible to identify the applicable temperature sensor by initially querying each possible address.

According to the datasheet for the TMP006/B temperature sensor the representation of the temperature is a signed 14-bit integer. In which the LSB represents  $(1/32)$  °C. If the temperature data is read into a single 16-bit buffer, the value would need to be right-shifted by 2, and then divided by 32.0 in order to yield the float value in Celsius. For example 25.0 °C:

DIGITAL OUTPUT Binary	DIGITAL OUTPUT Hex	SHIFTED OUTPUT Binary	SHIFTED OUTPUT Hex	SHIFTED OUTPUT (Dec)	FLOAT Output (Dec / 32.0) °C
0000 1100 1000 0000	0x0C80	0000 0011 0010 0000	0x0320	800	25.0 °C

If the temperature data is read into two 8-bit buffers, then conversion is more complicated. The MSBs would need to be left-shifted by 6, and converted to a signed 16-bit integer. The LSBs would need to right-shifted by 2. Taking the bitwise OR of the two offset values, converting the result to a float, and then dividing by 32.0 will also yield the correct value in Celsius.

```
> t
Current temp = 24.75C <76.55F>
> 
```

#### 4.1.3.2 Accelerometer Measurement

The CC3220SF development board includes the BMA222E 3-axis accelerometer sensor, with its own I2C bus address. With the supplied TI drivers, responses to measurement queries are received as 6 signed 8-bit integers, of which (starting with byte 0) byte 1, byte 3, and byte 5 are of particular interest.

Byte 0	Byte 1 (int8_t)	Byte 2	Byte 3 (int8_t)	Byte 4	Byte 5 (int8_t)
New data flag	X-Axis data	New data flag	Y-Axis data	New data flag	Z-Axis data

These signed integers are divided by 6, and then passed as signed integers (rather than floats) to the application layer. I have not observed an LSB representation corresponding to 6. However, experimentally a value of +1g corresponds to approximately 10, while a value of -1g corresponds to approximately -10. The CC3220SF development board can be rotated in order to observe changes on all three axes.

```
> a
AccX = 0
AccY = 0
AccZ = 10
> □
```

#### 4.1.4 WIFI Task

Both the basic WIFI connection and the time acquisition implementation in this project were originally derived from a SDK example project (local\_time). Similarly, the TCP Client implementation in this project was derived from another SDK example project (power\_measurement). WIFI features rely on TI drivers and especially the SimpleLink APIs.

WIFI activities are organized hierarchically, with more sophisticated activities requiring prior completion of more basic activities. Activity hierarchy is enforced using flags. WIFI commands are associated with the activities, and are organized into an enumeration which also follows the general hierarchy.

- 1 WIFI connection activities
  - 2 Time acquisition activities
    - 3 TCP Client activities

WIFI commands are accessed using the command console with selections '1-9'. Assuming that wifi\_config.h has correct WIFI settings, selections '3', '5', and '9' would connect to WIFI, acquire the time, and send messages with the TCP Client.

##### 4.1.4.1 WIFI Connection

Establishing the WIFI connection was streamlined for this project. The device simply needs to connect to an existing WIFI network, with a known password and security setting. A special header file (wifi\_config.h) was created to hold default WIFI configuration settings.

```
#define DEFAULT_WIFI_SSID_NAME          "Wifi Network Name"
#define DEFAULT_WIFI_SECURITY_TYPE      SL_WLAN_SEC_TYPE_WPA_WPA2
#define DEFAULT_WIFI_SECURITY_KEY       "password"
```

In the software, the WIFI network name (or simply the WIFI name) is often described as the SSID or the Access Point (AP). Similarly, the WIFI password is also described as a key.

Regarding the security type, WPA2 is the most common and requires a password. For public or open WIFI networks that do not use a password, SL\_WLAN\_SEC\_TYPE\_OPEN can be specified instead.

Connecting to the WIFI was originally an automatic process in the SDK example project. However, it relied on WIFI configuration settings specified at compile time. Otherwise, connecting was not possible. To provide a better user experience, I added a way for the WIFI configuration settings to be changed at runtime. Although, connecting to WIFI became a manual process involving a command console selection. Another command console selection displays the current WIFI configuration settings.

Aside from the WIFI Task, the SDK example project also uses POSIX functions to create an additional SimpleLink task (sl\_Task) that is needed for the SimpleLink APIs. This SimpleLink task has a high priority (9) and appears to require substantial heap memory allocation. At this time, I have not made progress with reducing the heap memory allocation or replacing the POSIX functions.

#### 4.1.4.2 Time Acquisition

Acquiring time from a time server was relatively simple due to the TI SimpleLink APIs for clock synchronization. Once connected to WIFI, testing showed that initially calling ClockSync\_update() was successful after a brief delay. Though, a waiting period is enforced between multiple calls to ClockSync\_update().

However, further testing showed that calling ClockSync\_update() might be unnecessary. Instead, calling ClockSync\_get() and then asctime() in order to display the current time automatically acquired the date and time as well.

```
> 5
> UTC time = Sat Dec 5 06:38:55 2020
```

In preparing a timestamp for the TCP Client ASCII Message Format, it was decided that calling asctime() was adequate. It creates a string that includes both a date and a time. Though, the location of the year term is somewhat less advantageous. Granted, there are other date and time formatting functions available, but I am not sufficiently familiar with them at this point. It is also possible to offset the time zone, but it is not implemented in this project.

Given that the temperature measurements and the accelerometer measurements occur locally, only the time is needed in order to prepare a TCP Client ASCII Message. A connection to the

external TCP Server is not needed at this point. Using the formatted printing function, the command console can display an up-to-date TCP Client ASCII Message.

The Non-RTOS implementation of time acquisition was more difficult. Apparently the TI SimpleLink APIs for clock synchronization are reliant on there being an RTOS. Instead, an online tutorial [1] for accessing an NTP time source was able to be adapted for this purpose. Other SimpleLink APIs were used to recreate the functionality of the tutorial which involves UDP client configuration. At this point, the time is not maintained internally, and the NTP time source has to be queried over and over.

#### 4.1.4.3 TCP Client

The TCP Client is based on a SDK example project (power\_measurement), but it is also influenced by a second SDK example project (tcp\_echo). In tcp\_echo, the CC3220SF development board acts as a server, and it accepts connections from external clients. It also provides a Python script to run a client. In contrast, power\_measurement provides an example of the CC3220SF development board acting as a client, but no external server. Research into other TI SDKs found different TCP example projects, and identified a simple networking tool called Iperf [2] [3]. In which case, iPerf can be setup to act as a basic server.

Focusing on the TCP Client, it relies on SimpleLink APIs to create a TCP socket. The socket becomes associated with specific characteristics:

- Destination IP Address (i.e. 192.168.2.250)
- Port (i.e. 5001)
- Socket ID (integer)

Initially, the TCP Client will open a socket and assign a positive integer for the socket ID. It will attempt to connect to the destination IP address and port (the TCP Server). Connecting will timeout if it fails. If connecting succeeds, the TCP Client will send data from a buffer to the destination IP address and port. The TCP Client will reuse the same socket to send more data until the socket is closed. When the socket is closed, a negative integer is assigned to the Socket ID. At this time, the command console is used for manually closing the socket and assigning a negative integer. This prevents more than one socket from being created.

The data that is sent is located in an 80 character buffer defined in a struct. The TCP Client ASCII Message is prepared with up-to-date data, and then copied to this buffer before it is sent. Currently, no other tasks write to this buffer.

A special header file (wifi\_config.h) contains default TCP Client settings. At this time, it is not possible to change these values with the command console.

```
#define SERVER_DEST_IP_ADDR    SL_IPV4_VAL(192,168,2,250)
#define SERVER_PORT            (5001)
#define SERVER_FRAME_LENGTH    (80)
```

The existing TCP socket can be closed (and the socket ID reset) using console command selection '8'. Afterwards, a new TCP socket will be opened for a subsequent connection attempt to the external TCP Server.

```
> 8
Closing TCP Client socket [sockID=1]
> █
```

Regarding the external TCP Server, I wanted to provide at least two options for testing the TCP Client. Overall, the relationship between TCP Client and the TCP Server has to be very basic.

- The TCP Server runs on a desktop computer
- The desktop computer is on the same network as the WIFI network
- The TCP Server accepts incoming connections on a specific port
- The TCP Client connects to the IP address of the desktop computer running the TCP server, with the specific port
- The TCP Client sends a specific amount of data to the TCP Server
- The TCP Server displays the specific amount of data, if possible

iPerf can be setup to act as a TCP server. It is as simple as running the following in the Windows Command Prompt:

```
iperf3.exe -s -p 5001 -i 1
```

```
C:\Windows\System32\cmd.exe
D:\TI\iperf-3.1.3-win64>iperf3.exe -s -p 5001 -i 1
-----
Server listening on 5001
```

Unfortunately no message data will be displayed in the Windows Command Prompt. However, the UART will display different indications when the TCP Client is and isn't connected to the TCP Server. It is suggested to start running the TCP Server before running the TCP Client.

```
9
Sat Dec 5 06:37:21 2020,23.97.75.14.0.0.10
Running TCP Client
TCP Client sent message to server [sockID=1]
> █
```

A Python script has been developed [4] in order to run a more useful TCP Server. It is necessary to run this python script on a computer with the same IP address as specified by SERVER\_DEST\_IP\_ADDR (in wifi\_config.h). An easy way to identify a computer's IP address is to enter ipconfig in the Windows Command Prompt. The Python script has a buffer size that corresponds to the SERVER\_FRAME\_LENGTH. The port number is specified to be the same as iPerf [5]. For convenience, a message limit of 10 is specified to allow the Python script to stop automatically. The Python script will display message as they arrive. It is suggested to start running the TCP Server Python script before running the TCP Client.

```
1  import socket
2
3  MAX_MESSAGES = 10           # total number of messages to receive
4  BUFSIZE = 80                # buffer length
5
6  s = None # socket id
7  buffer = bytearray(BUFSIZE)
8
9  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10 s.bind((socket.gethostname(), 5001))
11 s.listen(1)
12
13 print("Listening for client...")
14 clientsocket, clientaddress = s.accept()
15
16 print(f"Connection from {clientaddress} has been established!")
17
18 for x in range(MAX_MESSAGES):
19     data = clientsocket.recv(len(buffer))
20     print(data.decode("utf-8"), end='')
21
22 print("Closing connection...")
23 clientsocket.close()
24 s.close()
```

Listening for client...

Connection from ('192.168.2.165', 51811) has been established!

Sat Dec 5 03:10:23 2020,23.63,74.53,0,0,10

Sat Dec 5 03:10:26 2020,23.63,74.53,0,0,10

Sat Dec 5 03:10:30 2020,23.63,74.53,0,0,10

Sat Dec 5 03:10:33 2020,23.63,74.53,0,0,10

Sat Dec 5 03:10:36 2020,23.66,74.58,0,0,10

Sat Dec 5 03:10:40 2020,23.66,74.58,0,0,10

Sat Dec 5 03:10:43 2020,23.66,74.58,0,0,10

Sat Dec 5 03:10:47 2020,23.66,74.58,0,0,10

Sat Dec 5 03:10:51 2020,23.69,74.64,0,0,10

Sat Dec 5 03:10:54 2020,23.69,74.64,0,0,10

Closing connection...

## 4.2 RTOS and non-RTOS Versions Comparison

Both versions are able to deliver the TCP Client ASCII Message to an external TCP Server. The non-RTOS version of the project is less sophisticated and less flexible than the RTOS version. Essentially, the non-RTOS version has a hardware initialization phase followed by the main program loop.

In the main program loop, sensors are measured, time is acquired, and then a message is sent to the external TCP Server. The command console is not implemented, but measurements and WIFI activities occur automatically. Connection issues require resetting and re-loading for the device. From one of the example projects, there is a call to a SimpleLink task (sl\_task).

The SimpleLink host driver architecture mandates calling 'sl\_task' in a NO-RTOS application's main loop. The purpose of this call, is to handle asynchronous events and get flow control information.

The different time acquisition method was particularly difficult to implement. It required much additional study and testing in order to achieve. In contrast, the TCP Client implementation was relatively straightforward. It would be preferable to acquire time once and then maintain the time internally. The slower cycle for the main loop is related to the time acquisition.

With focus on the WIFI and networking capabilities, I would recommend using an RTOS implementation for applications that are similar to this project. It's well suited to handling asynchronous events and organizing execution for blocking function calls.

```
Remote Monitoring System Project - No RTOS - Michael Horne
Initializing I2C Bus
Identifying temperature sensor
Connecting to WIFI
Trying to connect to AP : Cannes Home 2E
[NETAPP EVENT] IP Acquired: IP=192.168.2.165 , Gateway=192.168.2.1
Starting main program loop
Sleeping 3 seconds...
Current temp = 23.47C <23.47F>
AccX = 0
AccY = 0
AccZ = 10
Send Packet...
Sat Dec 5 05:50:24 2020,23.47,74.24,0,0,10
Sleeping 3 seconds...
Current temp = 23.47C <23.47F>
AccX = 0
AccY = 0
AccZ = 10
Send Packet...
Sat Dec 5 05:50:28 2020,23.47,74.24,0,0,10
Sleeping 3 seconds...
Current temp = 23.47C <23.47F>
AccX = 0
AccY = 0
AccZ = 10
Send Packet...
Sat Dec 5 05:50:31 2020,23.47,74.24,0,0,10
Sleeping 3 seconds...
Current temp = 23.47C <23.47F>
AccX = 0
AccY = 0
AccZ = 10
Send Packet...
Sat Dec 5 05:50:34 2020,23.47,74.24,0,0,10
Sleeping 3 seconds...
Current temp = 23.50C <23.50F>
AccX = 0
AccY = 0
AccZ = 10
Send Packet...
Sat Dec 5 05:50:38 2020,23.50,74.30,0,0,10
```



## 5. References

[1] Lettier, D. (2017) Let's make a NTP Client in C.

<https://lettier.github.io/posts/2016-04-26-lets-make-a-ntp-client-in-c.html>

[2] Texas Instruments. (2017) CC3200 UDP Socket Application

[https://processors.wiki.ti.com/index.php/CC3200\\_UDP\\_Socket\\_Application](https://processors.wiki.ti.com/index.php/CC3200_UDP_Socket_Application)

[3] Texas Instruments. (2017) CC3200: SimpleLink™ WiFi CC31xx/CC32xx Forum.

<https://e2e.ti.com/support/wireless-connectivity/wifi/f/968/t/592768?CC3200-SimpleLink-WiFi-CC31xx-CC32xx-Forum>

[4] Kinsley, H. (2019) Sockets Tutorial with Python 3 part 1 - sending and receiving data

<https://www.youtube.com/watch?v=Lbfe3-v7yE0>

[5] List of TCP and UDP port numbers

[https://en.wikipedia.org/wiki/List\\_of\\_TCP\\_and\\_UDP\\_port\\_numbers](https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers)

## 6. Appendix

### 6.1 Syscfg from out\_of\_box (Accelerometer)

The image displays two screenshots of the Syscfg tool interface, showing the configuration of components for the 'out\_of\_box.syscfg' project.

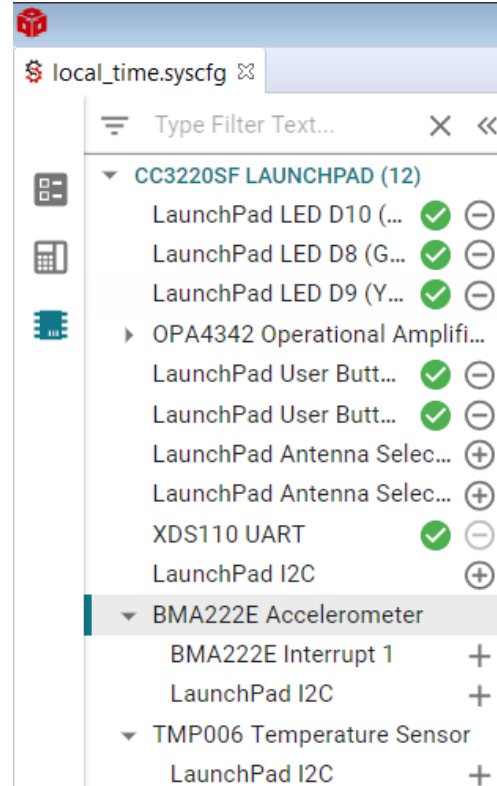
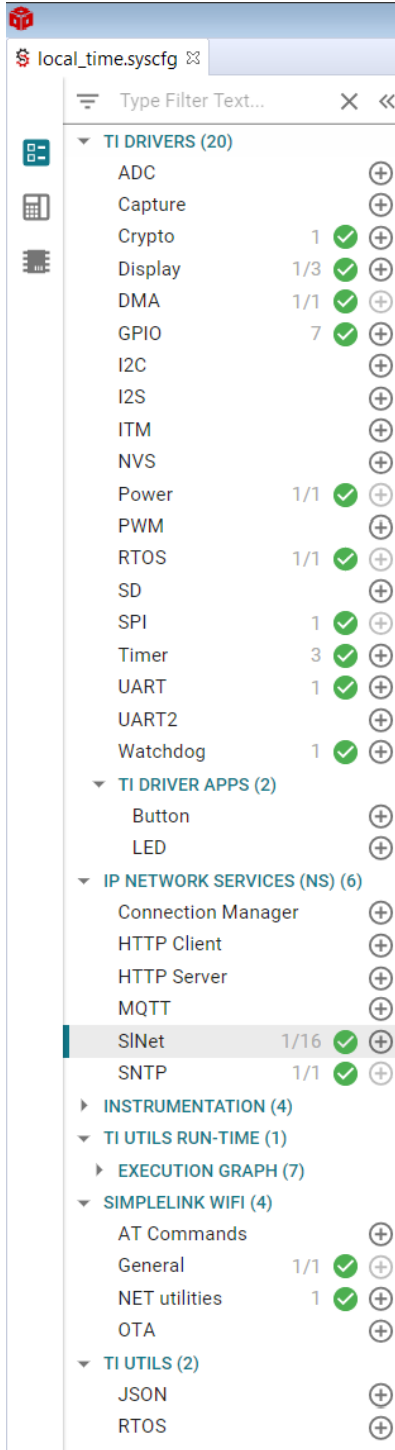
**Left Screenshot:** The 'TI DRIVERS (20)' category is expanded, showing a list of drivers with their status (checked/unchecked) and a plus icon for selection. The 'TI DRIVER APPS (2)' category is also expanded, showing 'Button' and 'LED'.

Category	Component	Status	Action
TI DRIVERS (20)	ADC	✓	+
	Capture	✓	+
	Crypto	1 ✓	+
	Display	1/3 ✓	+
	DMA	1/1 ✓	+
	GPIO	5 ✓	+
	I2C	1 ✓	+
	I2S	✓	+
	ITM	✓	+
	NVS	✓	+
	Power	1/1 ✓	+
	PWM	✓	+
	RTOS	1/1 ✓	+
	SD	✓	+
	SPI	1 ✓	+
	Timer	3 ✓	+
	UART	1 ✓	+
	UART2	✓	+
Watchdog	1 ✓	+	
TI DRIVER APPS (2)	Button	✓	+
	LED	✓	+
IP NETWORK SERVICES (NS) (6)	Connection Manager	✓	+
	HTTP Client	✓	+
	HTTP Server	✓	+
	MQTT	✓	+
	SINet	✓	+
	SNTP	✓	+
INSTRUMENTATION (4)	Bench	✓	+
	LogMain	✓	+
	LogSite	✓	+
	LoggerText	✓	+
TI UTILS RUN-TIME (1)	EXECUTION GRAPH (7)	✓	+
SIMPLELINK WIFI (4)	AT Commands	✓	+
	General	1/1 ✓	+
	NET utilities	✓	+
	OTA	✓	+
TI UTILS (2)	JSON	1/1 ✓	+
	RTOS	✓	+

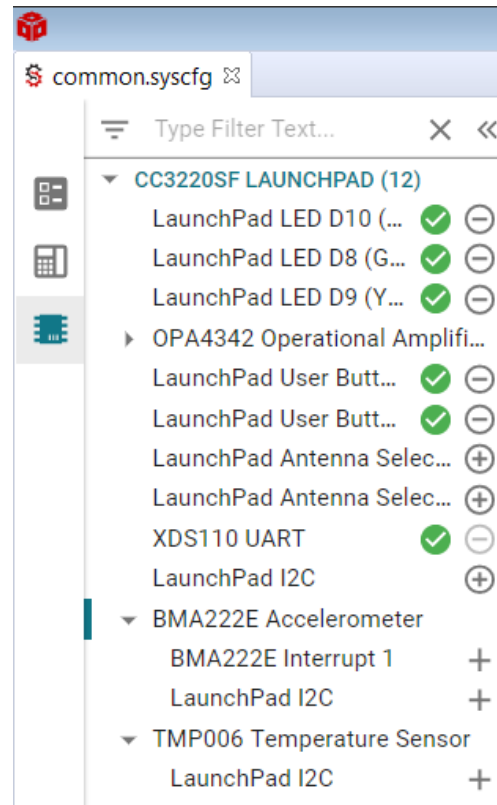
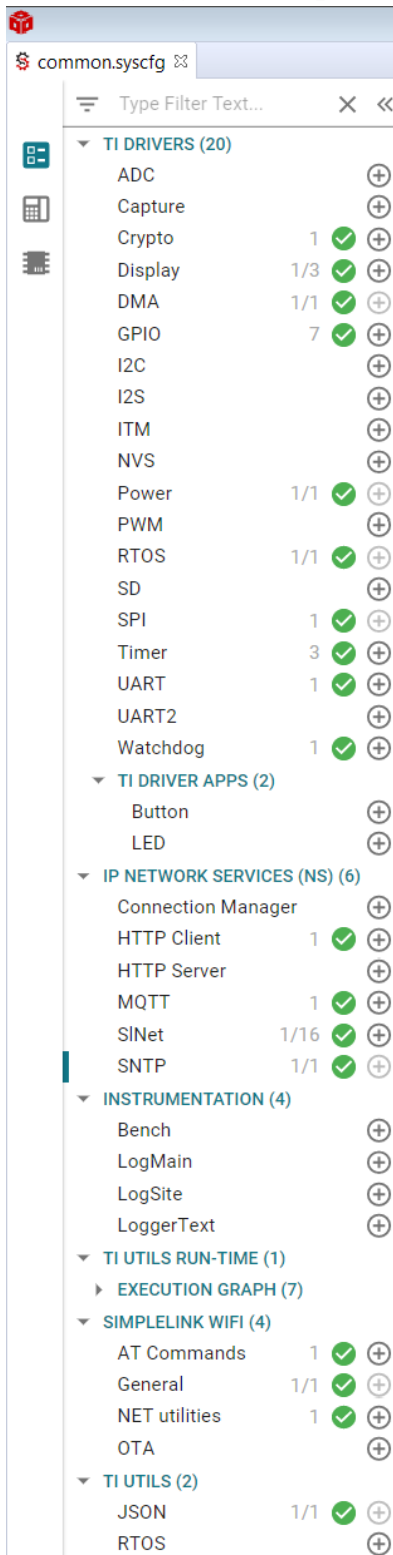
**Right Screenshot:** The 'CC3220SF LAUNCHPAD (12)' category is expanded, showing a list of launchpad components with their status (checked/unchecked) and a plus icon for selection. The 'BMA222E Accelerometer' category is also expanded, showing 'BMA222E Interrupt 1' and 'LaunchPad I2C'.

Category	Component	Status	Action
CC3220SF LAUNCHPAD (12)	LaunchPad LED D10 (...)	✓	+
	LaunchPad LED D8 (Green)	✓	+
	LaunchPad LED D9 (Yellow)	✓	+
	OPA4342 Operational Amplifi...	✓	+
	LaunchPad User Butt...	✓	+
	LaunchPad User Butt...	✓	+
	LaunchPad Antenna Selec...	✓	+
	LaunchPad Antenna Selec...	✓	+
	XDS110 UART	✓	+
	LaunchPad I2C	✓	+
	BMA222E Accelerometer	✓	+
	BMA222E Interrupt 1	✓	+
TMP006 Temperature S...	LaunchPad I2C	✓	+

## 6.2 Syscfg from local\_time (WIFI/Time Server)



### 6.3 Syscfg from tcpecho (WIFI/TCP)



## 6.4 Syscfg from portableNative (Temperature)

The image displays two screenshots of the Syscfg tool interface, showing the configuration of components for a project named 'local\_time.syscfg'.

**Left Screenshot:** The 'TI DRIVERS (20)' category is expanded, showing a list of drivers with their status (checked/unchecked) and a plus icon for selection. The 'SI' category is highlighted.

Category	Component	Status	Action
TI DRIVERS (20)	ADC		+
	Capture		+
	Crypto	1	✓ +
	Display	1/3	✓ +
	DMA	1/1	✓ +
	GPIO	7	✓ +
	I2C		+
	I2S		+
	ITM		+
	NVS		+
	Power	1/1	✓ +
	PWM		+
	RTOS	1/1	✓ +
	SD		+
	SPI	1	✓ +
	Timer	3	✓ +
	UART	1	✓ +
	UART2		+
	Watchdog	1	✓ +
	TI DRIVER APPS (2)	Button	
LED			+
IP NETWORK SERVICES (NS) (6)	Connection Manager		+
	HTTP Client		+
	HTTP Server		+
	MQTT		+
	SI	1/16	✓ +
SNTP		1/1	✓ +
INSTRUMENTATION (4)			
TI UTILS RUN-TIME (1)			
EXECUTION GRAPH (7)			
SIMPLELINK WIFI (4)	AT Commands		+
	General	1/1	✓ +
	NET utilities	1	✓ +
	OTA		+
TI UTILS (2)	JSON		+
	RTOS		+

**Right Screenshot:** The 'CC3220SF LAUNCHPAD (12)' category is expanded, showing a list of launchpad components with their status (checked/unchecked) and a plus icon for selection. The 'BMA222E Accelerometer' category is highlighted.

Category	Component	Status	Action	
CC3220SF LAUNCHPAD (12)	LaunchPad LED D10 (...)	✓	+	
	LaunchPad LED D8 (G...)	✓	+	
	LaunchPad LED D9 (Y...)	✓	+	
	OPA4342 Operational Amplifi...			
	LaunchPad User Butt...	✓	+	
	LaunchPad User Butt...	✓	+	
	LaunchPad Antenna Selec...		+	
	LaunchPad Antenna Selec...		+	
	XDS110 UART	✓	+	
	LaunchPad I2C		+	
	BMA222E Accelerometer	BMA222E Interrupt 1		+
		LaunchPad I2C		+
TMP006 Temperature Sensor	LaunchPad I2C		+	