Real-Time Sensor Network Monitoring with SmartMesh IP

By
Arteom Katkov
Suhaib Abugdera
Zak Abdi
Isaac Koop

Senior Design Final Report

Submitted to:
Advisor:
Dr. Yi Zheng

_____

Signature

Committee:
[Dr. Yao Aiping, Dr. Shensheng Tang]

_____

Signature

_____

Signature

Department of Electrical and Computer Engineering
College of Science and Engineering
St. Cloud State University
St. Cloud, MN

Fall 2022

Table of Contents

List of Figures

Abstract

This project proposes a wireless sensor network based on SmartMesh IP for deployment in remote areas without Wi-Fi access. The network utilizes an LTE module for connectivity and is able to transmit sensor data in real-time to a remote database for storage and monitoring. The system is battery-powered and uses solar panels for recharging, making it self-sustaining and suitable for deployment in remote locations. The use of SmartMesh IP technology allows for efficient and reliable communication between the sensors and the central hub, even in challenging environments. This wireless sensor network is ideal for a variety of applications, including environmental monitoring, agricultural monitoring, and industrial process monitoring, and has the potential to provide valuable insights and enable informed decision-making in remote areas where traditional internet connectivity may not be available.

Chapter 1: Introduction

Background

SmartMesh IP network brings a scalable, reliable, and energy efficient wireless sensor connectivity. This technology has advanced network management and a wide range of security features which include end to end encryption, message integrity checking and device authentication. With up to 8 times less power consumption than other solutions, SmartMesh IP is leading the industry in energy efficiency when it comes to wireless mesh sensing technology. This energy efficiency is unparalleled even in harsh dynamic RF environments [3]. A SmartMesh IP network consists of highly scalable, self-forming, self-healing, multi-hop mesh of wireless nodes called motes which are autonomous and compact devices with sensor capabilities. These motes are also capable of processing, actuating as well as communicating wirelessly and are used to send and receive data with a host application.

The network architecture is based on Time Synchronized Mesh Protocol (TSMP) which is standardized by the International Society of Automation ISA100.11a. At the heart of SmartMesh motes and network managers is the Eterna IEEE 802.15.4e system-on-chip (SoC). The SoC features Analog Devices' highly integrated, low-power 2.4GHz radio design. It also features an ARM Cortex-M3 32-bit microprocessor running SmartMesh networking software [4].

Using TSMP allows the motes to only operate when they are scheduled to receive or transmit data ensuring low power consumption. This is made possible by Time Slotted Channel Hopping (TSCH) media access layer (MAC) which is included in TSMP. With TSCH, time is divided into slots and the motes follow a communication schedule which gives directions on what to do in each time slot: send data, receive data or sleep. This utilization of TSMP and TSCH removes collision on the network, increases network bandwidth by allowing multiple transmissions to occur simultaneously and increases energy efficiency of the system [3].

Here at SCSU, several senior design groups implemented the SmartMesh IP network in their projects in the past. One group used an external microcontroller to configure and set up the network manager and control the motes through serial port connected to a computer. Another group used the built-in microcontroller to control the network through serial port and connected to a computer. They then used a C# GUI and a mobile app to monitor various sensor data online.

Our group will use an external microcontroller to setup the SmartMesh IP. We will also add a LTE module to our design to make the system more compact and allow us to monitor data remotely. Finally, unlike the previous groups who used the computer for communicating with the SmartMesh IP system, we will be using the microcontroller for the same purpose instead. This means all the processing that would be done by the computer would be through the microcontroller. So, we would develop the APIs that the system requires to communicate with the SmartMesh IP as well as process data collected.

The system will be fed from a solar panel and rechargeable battery through a power supply. The solar power will be controlled by a charger controller to charge the battery and from the battery to a buck converter of two output voltages 3.3V for Network Manager and 4V for LTE module. The current of the power supply can be programmed up to 5A. For this design, only 600mA is needed as maximum current. The purpose of having our system to be powered 100% from solar is to make the system self-contained and self-sustained.

The hardware design will be implemented separately at first. The network manager, LTE module, power supply, and charger controller are each a separate design. They will be designed using Altium Designer by creating a layout PCB. The PCB will be soldered and tested to achieve the desired purpose. After getting all hardware design implemented and worked, the motherboard will be designed and implemented as one PCB board including those separate designs. Also, an enclosure will be

designed to include our system; we will be using a Solid works software to create the enclosure and print it using see through plastic if possible.

Problem Statement

Currently, SmartMesh IP requires a computer to store and relay data from the sensor network. The goal with this project is to replace the computer with microcontroller and LTE module. This would allow for collected sensor data to be sent directly to the cloud where the entire system is live and can be monitored from a remote location. The system will be fed from the solar panel and rechargeable battery to ensure the power is available to the system load at any time.

Product Description

The finalized product will be a configurable network manager which can be deployed in remote locations with access to the internet through a LTE module. The entire SmartMesh IP sensor network will be able to be monitored remotely without needing to directly access the network manager. The updated network manager would be configurable to allow for setting up uploading frequency, configuring private keys used for signing transactions sent to the blockchain, and configure network status notifications.

The system must also be low power and allow for a minimum of three days of operation without any external power source. When the system does not have a power source connected, it will be run off lithium-ion batteries or other type of a high-quality battery. A power source will be able to be connected at any point which will start the battery automatically.

Communication between the network manager, microcontroller, and LTE module will be accomplished through the UART protocol. Communication with the computer will be established either through Bluetooth or USB to conserve power. The detailed product requirements are shown below.

Detailed Product Requirements

- The system will be able to communicate with an external server without the use of a computer.
- The system will use a cortex M0+ microcontroller for reduced power consumption.
- The system will be powered by rechargeable lithium-ion batteries or other battery type and have a power port for 3.3V and other required output using buck and boost converter.
- The system will be able to be live monitored from anywhere.
- The system will implement blockchain technology for data storage.
- The system will use the USB and UART communication protocols.
- The system will use SmartMesh IP for the sensor network.

Non-Technical Constraints

- Manufacturing: Given the current state of the IC markets, there is a parts shortage and will be difficult to obtain some of the IC components.
- Environmental: The Ethereum merge to POS (proof of stake) will be 99.95% more energy efficient than POW (proof of work) [5].
- As wireless transmission is extensively used in the system, the regulation of FCC and IEEE 802.11 standards need to be followed.

Benefits and Feasibility of the Project

Currently, to monitor a SmartMesh IP system, the network manager must be connected to a computer which will collect the data from the sensor network. However, this is not always a possibility since most SmartMesh IP sensor networks are deployed in remote areas with rough conditions. The finished product would remove the necessity for a computer to be needed with the network manager. The microcontroller and LTE module would replace the computer in this system. Through this system, the ease and cost of maintaining and operating a SmartMesh IP wireless sensor network would be

reduced substantially. This would also allow for the network to be deployed in more remote areas where it would not be feasible to have a computer deployed.

This project is the culmination of multiple senior design projects that have been completed at St. Cloud State over the last several years. The design uses existing technologies and combines them to create a product that will be able to overcome current limitations. Since all technologies are currently available, the project is considered feasible.

Chapter 2: System Design Methodology and Implementation

The system design was split into two main sections: hardware and software. The software included the GUI on the computer, the phone apps, as well as the firmware designed for the SAML21 MCU. The hardware design was split into three stages which will be described later within the report.

System-Level Design

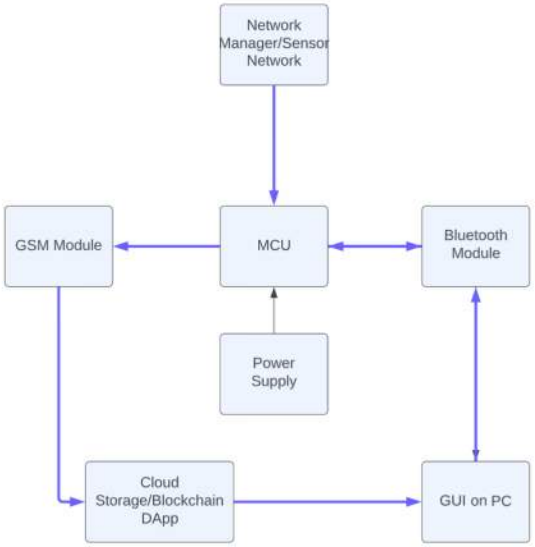

Figure 1: System Diagram

Figure 1 displays the overall system block diagram. The heart of the system will be the SAML21 MCU which will collect data from the LTC5800 network manager chip and send the data over to both local devices through Bluetooth and to a remote database through the NB-IoT module. The system will be battery powered and use a buck converter to step down the voltage to appropriate levels.

The SAML21 microcontroller is a low-power Cortex-M0+ microcontroller. This chip contains many peripherals such as UART and DMAC. This was very important due to our system requiring at least three separate UART communication channels for SmartMesh IP, LTE module, and Bluetooth. The SAML21 chip also supported the FreeRTOS distribution which is a real-time operating system(RTOS) and was necessary for this project to be accomplished.

The LTC5800 chip is the heart of the SmartMesh IP network and is used to form and maintain networks as well as communicate with the motes on this network. To communicate with this chip from the MCU, a UART port was used. A custom API developed by Dust Network was built on top of the UART communication protocol. It used packets with start and end flag to send commands between the MCU and LTC5800 chip.

The SIM7000 NB-IoT module is low power and supports UART/USB communication through AT Commands. It has many other features including GPIO, PCM, I2C, auto baud rate selection as well as a 20MB on board memory for local storage. This module was chosen over others because of its unique combination of performance, security, and flexibility as well as its support for low latency and low throughput IoT projects.

To summarize, the overall system diagram shown above will have a SAML21 uC at the heart. This MCU will be used to control the entire system. It will be used to communicate with the LTE module through UART which will be the bridge to connect the system to the internet so data can be stored to a database. It will also be used to process data from the motes which is collected through the redesigned

Network Manager. This will be done by directly interfacing with the LTC5800 MCU that is used for the SmartMesh IP Network Manager and a simple block diagram of the new Network manager is shown below in figure 2.

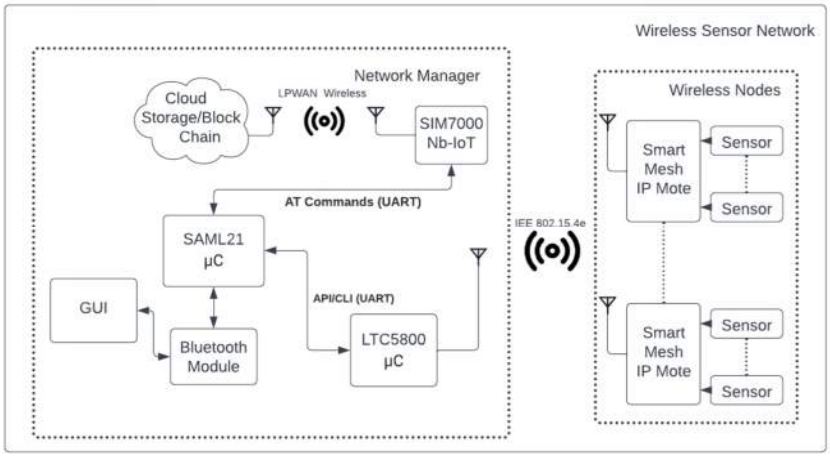

Figure 2: Redesigned Network Manager

FreeRTOS Real-time Operating System

A real-time operating system was necessary within this project for a couple of reasons. First, data that would be coming in from the network manager might not have the time to be processed before the next data packet would come in. Next, Bluetooth, LTE module, and network manager communication must occur simultaneously. Finally, certain components of the system must be processed in greater precedence before others. The overall FreeRTOS structure is shown in Figure 3.

```
FreeRTOS
└─Source
    ├─tasks.c          FreeRTOS source file - always required
    ├─list.c           FreeRTOS source file - always required
    ├─queue.c          FreeRTOS source file - nearly always required
    ├─timers.c         FreeRTOS source file - optional
    ├─event_groups.c   FreeRTOS source file - optional
    └─croutine.c       FreeRTOS source file - optional
```

Figure 3 - FreeRTOS Structure

The main components from FreeRTOS used within this project are the tasks, queue, and semaphore libraries. Tasks allow for multithreading to occur within the system, a queue is a special data structure which allows for first-in-first-out(FIFO) data storage, and finally semaphores are flags that can be used to control access to certain resources as well as control which tasks are currently active.

Whenever a task is created and starts running, there are two main states it can be in: blocked and unblocked. A more in-depth diagram is shown below in Figure 4. Whenever the task is waiting for some other task or process to be finished it is stuck within the blocked state. If a task is manually paused, it goes into the suspended state. If a task needs to be executed, however, a task with greater precedence is currently running, the task is within the ready state. This diagram was used as the basis for constructing most of the code for this project.

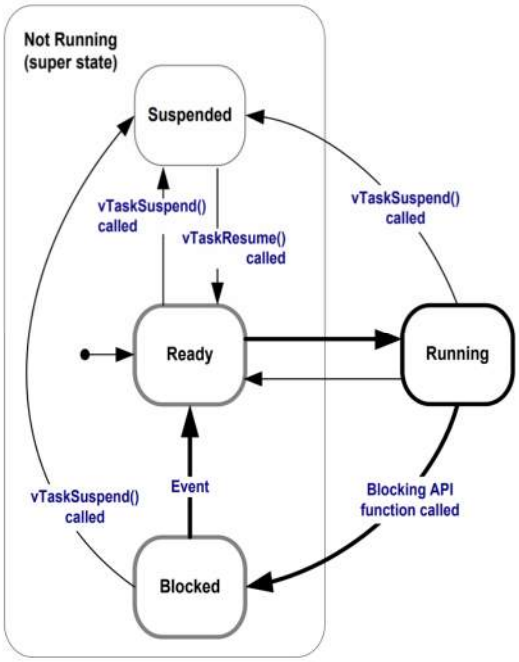

Figure 4 - States a Task May be In

To schedule the tasks, FreeRTOS uses the Fixed Priority Pre-emptive Scheduling with Time Slicing algorithm. Others exists, however, for this application this algorithm is the easiest to use and understand. Fixed priority means that the scheduler does not change the task priority assigned to the task. Pre-emptive means the scheduler will "pre-empt" a running task if a higher priority task is waiting to be executed. Time slicing is the method the scheduler uses to allocate CPU time to tasks of different priorities. One time-slice is the time it takes for the RTOS to tick once. For this project the RTOS was set to run at 1000Hz therefore setting each time slice to 1ms. Figure 5 shows the scheduling parts described above in action.

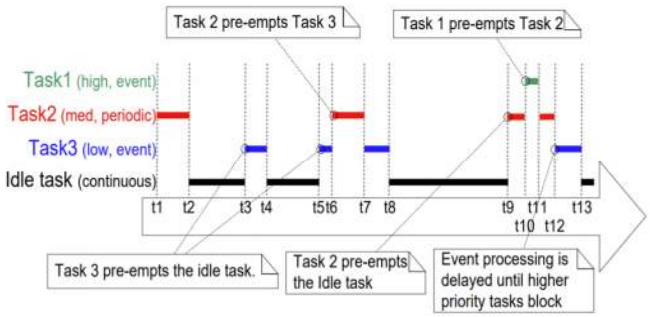

Figure 5 - Task Scheduling Example

The idle task is the default task that runs whenever the CPU has no other tasks running. Task priorities can be configured whenever a task is created or modified during task execution. The maximum task priority is setup within the RTOS configuration file before the program is compiled. For the project, priority values from 0 to 55 were used.

The other extremely important concept for using real-time operating systems is resource control. A mutex is a special type of binary semaphore which is used for access control for limited resources. Figure 6 shows how a mutex is used within a multithreaded application. Whenever a second tasks tries to take a mutex which has already been taken by the first task, it will enter a blocked state

until either its wait time elapses, or the semaphore has been given back. This process effectively allows for only one task at a time to access a limited resource.

A real-time operating system was critical for processing packets from the network manager chip as well as for simultaneously processing data coming in from different sources. This formed the backbone for the firmware which was developed.

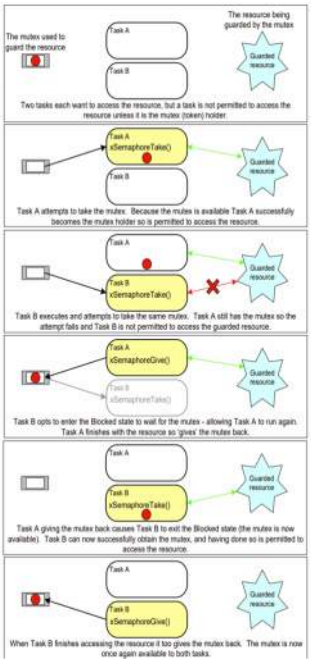

Figure 6 - Mutex Usage

DMAC Direct Memory Access Controller

Implementing DMA into this project was also extremely vital for its success. Since the MCU is busy with running the rest of the code of the project, copying buffers of data should be offloaded therefore increasing the performance of the overall system. The block diagram of this DMA is shown below in Figure 7. The DMA contains 16 different channels which can be used to setup multiple different transfer types for different situations. Only one of these channels can be active at a time with the lowest

channel number having the highest priority. A CRC engine is also included with the DMA which unfortunately did not output the correct checksum that was needed for this project and therefore was not used. The low-power bus matrix was used to transfer data between different sources and destinations with the DMA controlling where the data would be sent.

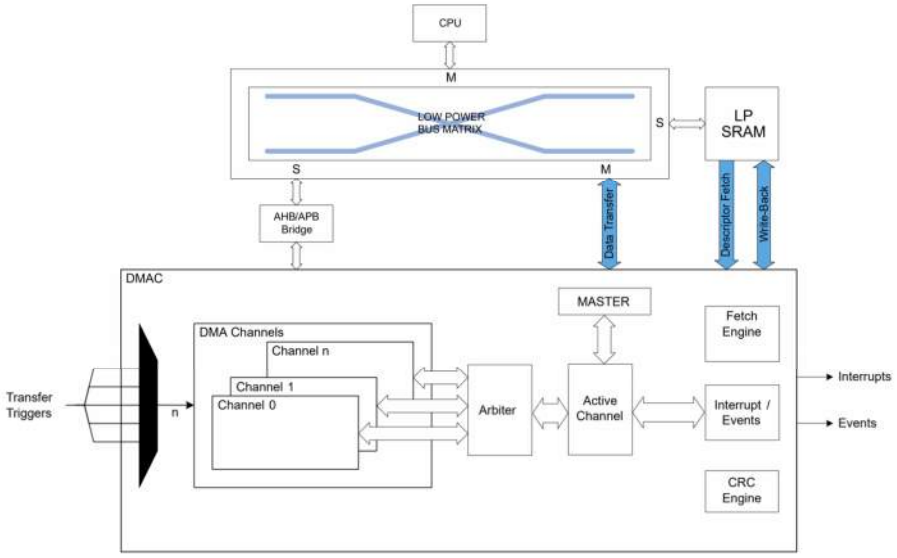

Figure 7 - DMAC Bock Diagram

To setup the DMA a transfer descriptor must be stored in low-power SRAM. The transfer descriptor begins at the base address where the channel 0 transfer descriptor is stored. The transfer descriptor is used to tell the DMA from where to get the data and where to copy the data to. Also, auto address increment options and transfer size are all configurable within this descriptor. Finally, event lines which run from different peripherals can be used to keep triggering the DMA to copy more data over until the data count goes to zero.

Multiple different transfer descriptors can be linked to run one after another as shown below in Figure 8. Each descriptor contains a pointer to the next descriptor to be run with the last descriptor

containing a NULL address. This indicates that this was the final channel descriptor and there is no more data to copy after this set. The writeback address stores the current version of the transfer descriptor which can be edited as the number of bytes that are left to copy goes down. The base and writeback memory addresses can either be the same or different.

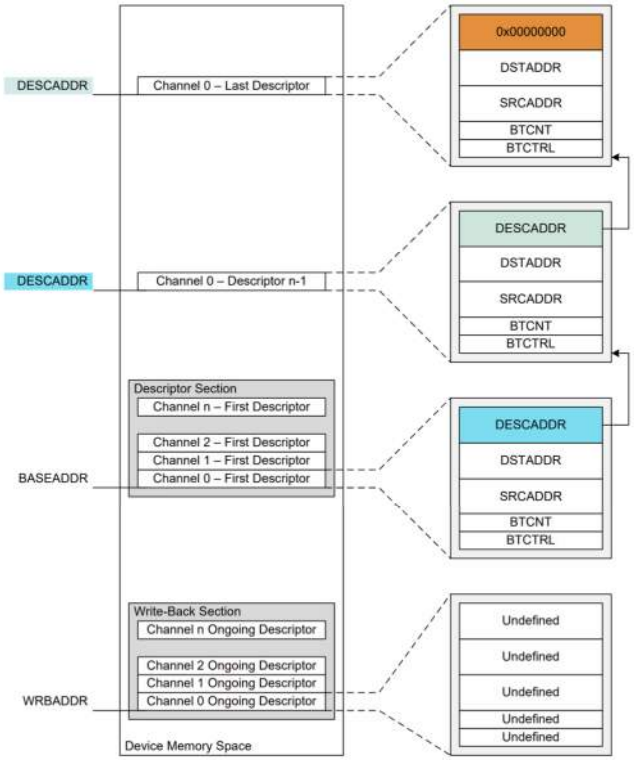

Figure 8 - DMA Transfer Descriptors

SmartMesh IP API Port Packet Structure

The SmartMesh IP API port uses a packet-based transmission protocol that is built on top of the UART communication protocol. The basic packet structure is shown below in Figure 9. The packet begins and ends with a flag which for the protocol is 0x7E. The 2-byte frame check sequence(FCS) is calculated from the data bytes that come in and uses the RFC 1662 defined FCS.

Flag	Data	Frame Check Sequence	Flag
1 byte	N bytes	2 bytes	1 byte

Figure 9 - Packet Structure

A detailed look at the data portion of the packet is shown below in Figure 10. The maximum length of this portion is 127 bytes therefore setting the maximum packet length to 131 bytes. The control byte is used to tell if the packet is a data or acknowledgement packet and if an acknowledgement should be sent by the recipient. The packet type indicates what data is being sent or received. All commands were given within the SmartMesh IP API user guide. The sequence number is the number of packets since communication was first initialized between the device and the network manager. The payload length gives the length of the payload which is the next N bytes that are sent.

Commented [ZA1]: @Katkov, Arteom what?

Commented [ASF2R1]: @Abdi, Zakaria M; @Katkov, Arteom lol

Control	Packet Type	Seq. Number	Payload Length	Payload
1 byte	1 byte	1 byte	1 byte	0-N bytes (type-specific)

Figure 10 - Data Portion of Packet

To initialize communication between a device and the network manager, a special hello packet must be sent. The payload of this packet is shown below in Figure 11. The API version used within this project is version 4. The sequence number parameter is the first number to start incrementing from. Once this packet is sent and the packet response is received from the network manager, communication between the two devices has been initiated and other commands can now be sent.

Parameter	Type	Description
version	INT8U	Version of the Protocol supported by the client. The manager checks this field to decide on compatibility with the client. If the protocol version is not supported by the manager, the *helloResponse* will contain an unsupported version error code, and the version field will contain the supported version. This document describes protocol version 4.
cliSeqNo	INT8U	The client sequence number is the unique number of this *hello* packet. Used for reliable communication, both sides use unique numbers to detect duplicate reliable messages. Once the *helloResponse* is received, the client's reliable commands must start with the "next" sequence number.
mode	INT8U	Reserved for compatibility; must be 0

Figure 11 - Hello Packet Parameters

SmartMesh IP API Packet Collection

The packets that were described in the previous section had to be collected and packaged before any parsing could be done to any of them. This process would be completed within the UART interrupt since this is where through what the data would be coming in from. The flowchart for this process is shown below in Figure 12.

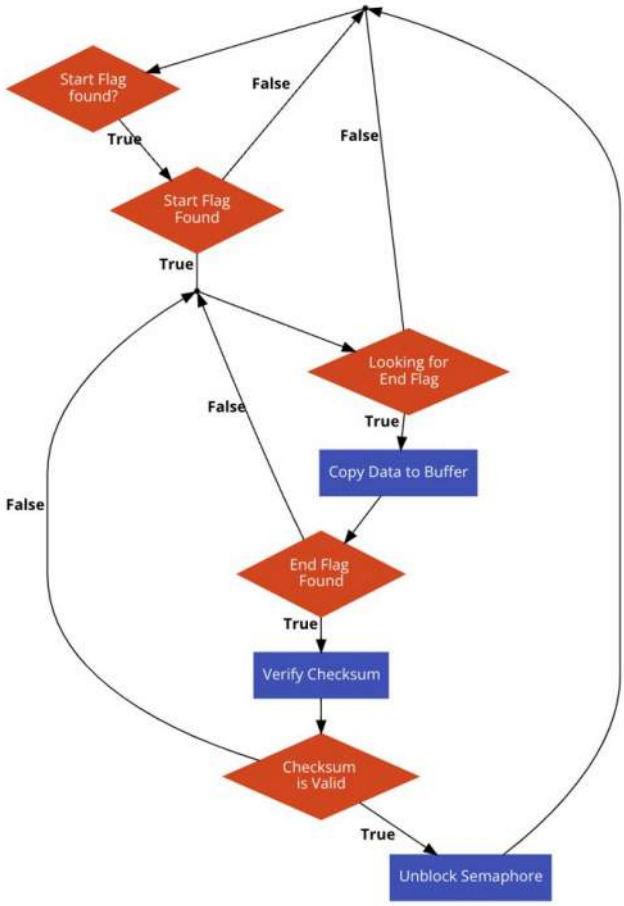

Figure 12 - Packet Parsing Flowchart

First the program looks for a start flag which in this case is 0x7E. Once it is found, packet collection can begin, and the program looks for the end flag which is the same as the start flag. Once the end flag is found, the program then runs the checksum which is compared to the checksum received through the UART port. If they match, the end of the packet has been found so a semaphore is unblocked. Otherwise, it was just a data byte that is part of the packet and not the actual end of the

packet which means that data collection should just continue. The code implementation of this flowchart is shown below in Figure 13.

```
if(!startFlag && data == 0x7E){// Start flag found
  length = 0;
  startFlag = true;
  txbuffer[length] = data;
}
else if(startFlag && data == 0x7E && length > 2){// Checksum to check if end of flag was found
  uint16_t check = verifyPacket(START_CHECKSUM, txbuffer, length - 2);
  uint16_t currCheck = txbuffer[length - 1] | (txbuffer[length] << 8);
  txbuffer[length+1] = data;
  if(check == currCheck || (txbuffer[2] == 0x40 && length == 0x32) || (txbuffer[2] == 0x3F && length == 0x1D)){/
    xSemaphoreGiveFromISR(dataRecieved, NULL);
    startFlag = false;
    xSemaphoreTakeFromISR(dma_in_use, NULL);
    DMAC_REGS->DMAC_CHID = 0;
    while(DMAC_REGS->DMAC_CHCTRLA != 0);// Check to see if DMA is still running
    uint32_t *desc = (uint32_t*)0x30000000;
    *desc++ = ((length+2) << 16)|0x0C01;
    *desc++ = (uint32_t)(txbuffer + length + 2);// Source address
    *desc = (uint32_t)(smartmeshData + length + 2);// Dest address
    DMAC_REGS->DMAC_CHCTRLA = 0x2;// Enable the channel
    DMAC_REGS->DMAC_SWTRIGCTRL |= 0x1;
    xSemaphoreGiveFromISR(dma_in_use, NULL);
  }
  else{// End of packet not found
    length++;
  }
}
else if(startFlag && data != 0x7E){// Copy to buffer
  txbuffer[length+1] = data;
  length++;
}
```

Figure 13 - Packet Parsing Code

A Boolean global variable called startFlag was used to keep track of when a start flag has been found. Next, until the end flag is found data is stored within a buffer which holds the current packet being collected. Finally, whenever the end flag is found, the checksum is run to verify if the end of the packet has been found. This is done through the verifyPacket function which is not shown within the figure. If the packet is indeed valid, the data stored within the temporary buffer is then copied over to a separate buffer so no overwrites could happen. This is done using the DMA with the descriptor being setup after the semaphore for the DMA has been taken. Finally, both the DMA semaphore and the packet received semaphore are both unblocked which allows for the SmartMesh IP packet parsing task to begin running.

Firmware Implementation

The MCU firmware had to be implemented very carefully since a real-time operating system was being used. Resource control had to be implemented for multiple tasks to not be able to access the same resource at the same exact time. Also, since each task would have its own flowchart, communication between tasks had to be accomplished with semaphores and mutexes. Also, to improve code efficiency, the C++ programming language was used in the firmware. This allows for class creation and access to many more methods which improve the code size and quality.

System Initialization

Before the main operation of the system is started, microcontroller drivers and communication with the different devices had to be established. All driver code had to be developed since no libraries were available to download online. Each component's initialization will now be described in detail.

Clock Initialization

Before any of the other drivers were initialized, the internal clock was setup to 48MHz using a phase-locked-loop(PLL). Since the system requires quick processing of data, setup the clock to the maximum possible frequency was imperative for the system to respond quickly. The flowchart for setting this clock up is shown below in Figure 14. Once this process is complete, each instruction will execute at 48MHz which improve the system performance greatly at minimal energy cost. The SystemCoreClock variable is also updated to reflect the new clock value. This must be done to ensure that the RTOS kernel ticks at 1kHz and the frequency does not change.

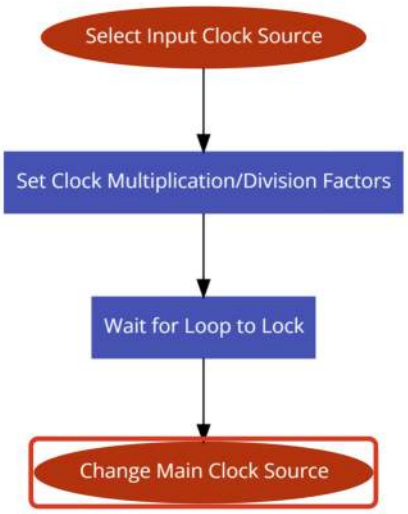

Figure 14 - PLL Flowchart

DMA Initialization

For the DMA to function properly, the writeback and base address had to be setup prior to the DMA being used. Also, the control register had to be set to receive proper notifications from the correct peripherals which for this system is from UART. The code for this setup is shown below in Figure 15. The comments on top shown which DMA channels are used for what.

```
// DMA initialization and setup
// Channel 0 => Smartmesh IP data copying
// Channels 1-3 => UART TX data transfer for SERCOM0-SERCOM2
// Channel 4 => Misc. Copying tasks
DMAC_REGS->DMAC_BASEADDR = 0x30000000;
DMAC_REGS->DMAC_WRBADDR = 0x30000100;
DMAC_REGS->DMAC_CTRL = 0xF02;
DMAC_REGS->DMAC_CHCTRLB = 0x60;// Channel 0 config
DMAC_REGS->DMAC_CHID = 0x1;// Set to channel 1
DMAC_REGS->DMAC_CHCTRLB = 0x800260;
DMAC_REGS->DMAC_CHID = 0x2;// Set to channel 2
DMAC_REGS->DMAC_CHCTRLB = 0x800460;
DMAC_REGS->DMAC_CHID = 0x3;// Set to channel 3
DMAC_REGS->DMAC_CHCTRLB = 0x800660;
DMAC_REGS->DMAC_CHID = 0x4;// Set to channel 4
DMAC_REGS->DMAC_CHCTRLB = 0x60;
```

Figure 15 - DMA Setup

Since the low-power SRAM begins at address three million hex, this is where the base and writeback addresses are stored. Each descriptor takes up 16 bytes of memory thus, for 16 different channels, it would take up 0x100 bytes of memory. Therefore, the writeback address is offset by this amount of memory. To configure each channel, the channel ID must be changed to point to the specific channel which is going to be setup. Each channel contains separate configuration and status registers.

UART Initialization

Since multiple UART port were going to be used within this project, a class was developed that would allow for the creation of different UART objects which would point to different UART ports. This class contained two different functions which would abstract the sending process of UART and make it easier to write code for the main program. The class constructor was used to setup the control registers for the specific UART port which is passed as a command to said constructor. Some of the setup code is shown below in Figure 16.

```
}else if(port == SERCOM2_REGS){
  dma_channel_id = 3;
  PORT_REGS->GROUP[0].PORT_PINCFG[12] = 0x1;
  PORT_REGS->GROUP[0].PORT_PINCFG[14] = 0x1;
  PORT_REGS->GROUP[0].PORT_PMUX[6] = 0x2;
  PORT_REGS->GROUP[0].PORT_PMUX[7] = 0x2;
  NVIC->IP[2] |= (64 << 16);
  NVIC->ISER[0] |= (1 << 10);
  NVIC->ICPR[0] |= (1 << 10);
  GCLK_REGS->GCLK_PCHCTRL[20] = 0x42;// Enable UART clock
}

// UART interface setup w/ proper baud rate
UART_port->SERCOM_CTRLA = 0x40010004;// Internal clock => SERCOM_PAD[2] TX,
UART_port->SERCOM_CTRLB |= 0x30000;// Enable tx and rx pins
UART_port->SERCOM_BAUD = 57999;// Always set to 115200 for now
UART_port->SERCOM_INTENSET |= 0x4;// Enable interrupt rx complete interrupt
UART_port->SERCOM_CTRLA |= 0x2;// Enable the UART port
```

Figure 16 - UART Setup Code

Each of the UART ports would have their own dedicated channel which will be used to copy data from the buffer over to the transmission register. Two pins also had to be setup from GPIO mode to UART TX and RX mode. The UART interrupt was also enabled, and its priority had to be changed. Within

the SAML21 microcontroller, there are only four different interrupt priorities, and they use the two most significant bits within the interrupt priority register. This means that there only are priorities of 0, 64, 128, and 192 available to choose from. On any ARM Cortex processors, the highest interrupt priority corresponds to the lowest interrupt number, thus 0 is the highest priority for interrupts in this case. When running the FreeRTOS kernel, the highest interrupt priority must be saved for the kernel to perform task switching and any other interrupts must have a lower priority, or the system will not be stable. Therefore, the interrupt priority was adjusted for the UART interrupt. To finish the UART setup, the baud rate was set to 115200 and the UART port was enabled which allowed data to be send and received.

The data sending code is shown below in Figure 17. Whenever data was ready to be sent, the DMA semaphore was taking to gain access to the control registers, Next, the channel id is to the value that was initialized in the constructor. Finally, the transfer descriptor was setup to copy from the proper location and to copy the proper number of bytes. The DMA semaphore was then given since the resource was no longer being accessed. A similar process was completed within the other UART function which implemented a printf function for data transmission.

```
73 void UART::send_array(uint8_t *data, uint8_t length){
74   xSemaphoreTake(dma_in_use, portMAX_DELAY);// Wait for dma regs to become available
75   DMAC_REGS->DMAC_CHID = dma_channel_id;// Setup the dma transfer
76   while(DMAC_REGS->DMAC_CHCTRLA != 0);// Check to see if DMA is still running
77   uint32_t *desc = (uint32_t*)(0x30000000 + 0x10*dma_channel_id);// If not send data
78   *desc++ = (length << 16)|0x0401;
79   *desc = (uint32_t)(data + length);// Source address end value
80   DMAC_REGS->DMAC_CHCTRLA = 0x2;// Enable the channel
81   xSemaphoreGive(dma_in_use);// Give back the semphore as done accessing the data
82 }
```

Figure 17 - Array Sending Code

SmartMesh IP Connection Initialization

Before any commands could be sent through the API port communication had to first be initialized with the network manager. As explained in the packet description section, a hello packet must

be sent by the MCU to initiate the communication. The flowchart for this process is shown below in Figure 18.

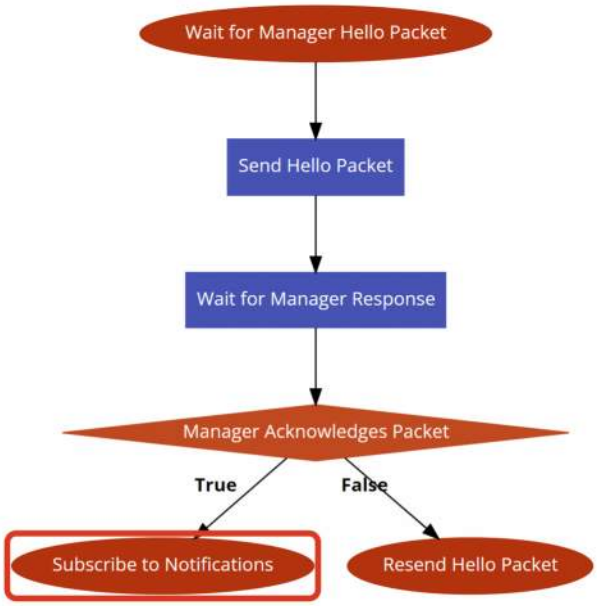

Figure 18 - Network Manager Communication Setup

The network manager will send its own hello packet roughly every three seconds which is then parsed by the MCU. Once this happens, the hello packet described above will be send to the network manager and the MCU will wait for the response. If the network manager acknowledges the packet, then the MCU will subscribe to notifications to be able to receive data packets from motes. Otherwise, the process will be repeated until successful. Some of the code for this flowchart is shown in Figure 19. The MGR_HELLO case statement will if the network manager sends its own hello packet. The Boolean flag is used to tell the GUI whether a connection with the network manager has been established or not. Next, the mgr_init function is executed which sends the network manager a hello command. A semaphore is used to control access to the SmartMesh IP API library so that multiple requests are not

sent at the same time from different functions. Whenever a manger response is received, the program then subscribes to all notifications which as stated above allows the network manager to send data that has been received from mote through the API port. Finally, the Boolean flag is set to true signaling that the network manager connection has been established successfully.

```
case MGR_HELLO_RESPONSE:
  xSemaphoreTake(apiInUse, portMAX_DELAY);
  api.subscribe(0xFFFFFFFF,0xFFFFFFFF);// Subscribe to notifications
  xSemaphoreGive(apiInUse);
  connectedToManager = true;// Set that the network manager is connected to the MCU
  break;

case MGR_HELLO:// No communication was setup
  connectedToManager = false;// No connection has been established
  xSemaphoreTake(apiInUse, portMAX_DELAY);
  api.mgr_init();// Send hello packet
  xSemaphoreGive(apiInUse);
  break;
```

Figure 19 - Manager Connection Setup Code

LTE Module Connection Initialization

The LTE module has a straightforward process in this project. First, the module is initialized by synchronizing its baud rate with the MCU. Then, a network connection is established using the hologram SIM Card and by sending a few commands to the module. Once that is done, the module is ready to send data to a database. This happens in a task in the RTOS, and it is controlled using a semaphore which is initially blocked and only gets unblocked after connection is successfully established. After sending a command to the module, its response is parsed, and this happens in a second task that is dynamically created whenever a response has been received. This task is then deleted after the response is parsed. The different parts of this process will now be discussed.

Starting with the initialization, once the system is powered on and system clocks/peripherals are setup, the next thing to do is to get the LTE module ready and establish a network connection so that data can be sent as soon as it is received. The following flowchart in Figure 20 shows the first part of this process.

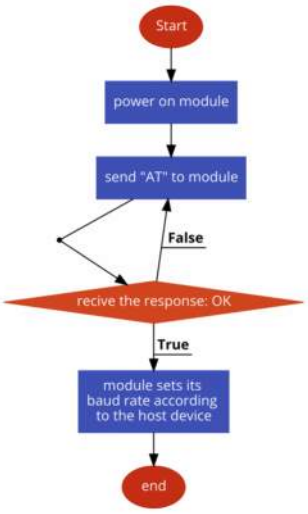

Figure 20: LTE module initialization process

As the figure shows, as soon as the module is powered on, the command “AT” is sent to it. This is done to let the module synchronize its baud rate with the MCU. The command will be sent repeatedly until an “OK” response is received. After baud rate is synchronized, it will respond with “OK” and commands can now be sent to set up network. This process can be seen below in Figure 21.

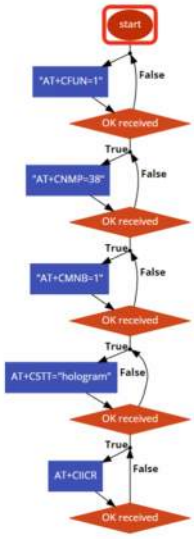

Figure 21: Setting up network connection for the LTE module

To set up a network connection, the commands shown in figure 20 above must be sent in the order shown. These commands do the following:

Commented [ASF3]: @Katkov, Arteom; @Abdi, Zakaria M; @Koop, Isaac P this should be figure 21 right?

1. "AT+CFUN=1" - Set module to full functionality mode
2. "AT+CNMP=38" - Set preferred mode to LTE only
3. "AT+CMNB=1" – Set device preference to Cat-M1 over NB-IoT
4. "AT+CSTT="hologram"" - Set the APN (access point name)
5. "AT+CIICR" - Establish a connection.

The commands must be sent in an order and therefore, if a command fails it must be resent before the next one can be sent to ensure the setup does not fail. This means the responses from the module will need to be parsed (which will be discussed in a later section.) Once a network connection is

established, data can be sent to a database and a task is used to do this (which will also be discussed in a later section). So, the semaphore corresponding to this task will be unblocked at this step.

AT Command Types and Responses

To parse the LTE module responses, it is important to know the types of commands that can be sent to the module and their corresponding responses. There are 4 types of commands, and they are as follows:

1. Write Command: i.e., “AT+CNMP=38”
    a. Sets parameters/values
    b. Response: “OK”, “ERROR”
2. Run Command: i.e., “AT+CIICR”
    a. Executes a command
    b. Response: “OK”, “ERROR”
3. Test Command: i.e., “AT+CNMP=?”
    a. Returns list of parameter/value that can be set by write command
    b. Response: URC – unsolicited response code i.e., +CNMP=2,13,38,51
4. Read Command: i.e., “AT+CNMP?”
    a. Returns currently set parameters/values
    b. Response: URC – unsolicited response code i.e., +CNMP=38

Knowing the types of commands enables us to be able to parse them accordingly. So, the responses can be categorized into 3 main types “OK”, “ERROR”, and a URC which is a response that starts with the character ‘+’.

LTE Module Response Parsing

Sending commands to the LTE module and parsing the responses happens in 2 different tasks in the RTOS. The sending task sends the command, and the parsing task is dynamically created when a

response is received. This is important because as stated above, the commands must be sent in a certain order and if one doesn't go through, the rest of them will not go through either. So, the tasks are controlled using semaphores and a command will not be sent while a response from previous command is being parsed.

The LTE module's responses always end in a new line character, so the response is stored in a buffer until a new line character is received. Once that happens, the parsing task is created to parse this response. The reason for creating it dynamically rather than creating it once and using a semaphore to control access to it is because some of the commands have multiple lines of response. So, creating a new task every time a newline character is received makes the system faster to the point where multiple lines can possibly be parsed simultaneously. A flowchart is shown below in Figure 22 to show how commands are sent to the module.

Commented [ASF4]: @Katkov, Arteom; @Abdi, Zakaria M this should be figure 22 right?

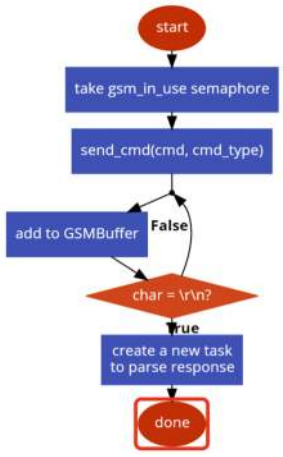

Figure 22: Sending Command to the LTE Module

The gsm_in_use semaphore is used to control access to the command sending task. So, when sending a task, the semaphore is taken, and it must be given before another command can be sent. This

happens after the response of that command is parsed. The function send_cmd() is used to send commands and it takes 2 parameters. One is the command itself and the other is to specify the command type used for parsing the response. Data is added to the GSMBuffer until a new line character is received then the response parsing task is created. A flowchart for this task can be seen below in Figure 23.

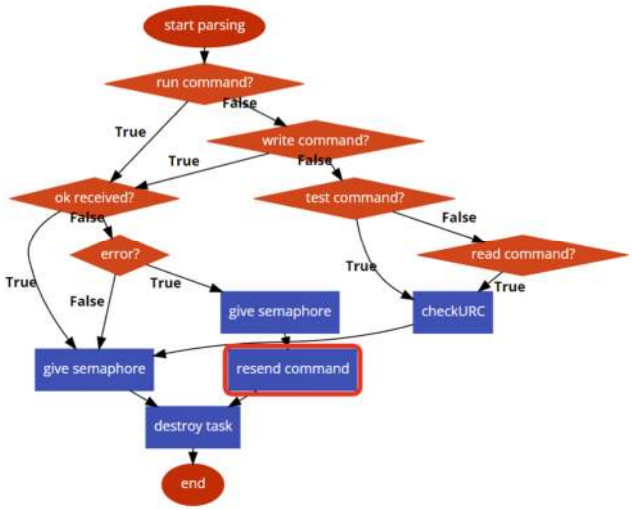

Figure 23: LTE Module Response Parsing Flowchart

As explained in the previous section, the responses are parsed according to their types. The run and write commands have OK or ERROR responses and the other 2 have a URC response. So, in the flowchart above, the response type is first determined, then the response is checked. If it is an OK response or URC response, the gsm_in_use semaphore is given. This allows for the command sending task to send another command. If ERROR is received, the command is resent using the command sending task and by calling the function send_cmd(). Of course, the gsm_in_use semaphore is given first

before resending command. Once parsing is completed, the task is destroyed as there is no use for it anymore. A small portion of the code for parsing the response can be seen below in Figure 24.

```
99  void parseGSMData(void* unused)
100 {
101   char tempBuffer[200];
102
103   xSemaphoreTake(dma_in_use, NULL);
104   DMAC_REGS->DMAC_CHID = 0;
105   while(DMAC_REGS->DMAC_CHCTRLA != 0);// Check to see if DMA is still running
106   uint32_t *desc = (uint32_t*)0x30000000;
107   *desc++ = ((responseLengthCopy + 2) << 16)|0x0C01;
108   *desc++ = (uint32_t)(responseGsmBuffer + responseLengthCopy + 2);// Source address
109   *desc = (uint32_t)(tempBuffer + responseLengthCopy + 2);// Dest address
110   DMAC_REGS->DMAC_CHCTRLA = 0x2;// Enable the channel
111   DMAC_REGS->DMAC_SWTRIGCTRL |= 0x1;
112   xSemaphoreGive(dma_in_use);
113
114   responseLengthCopy = 0;
115   if (AT_COMMAND_RUN)
116   {
117     if(strstr(tempBuffer, "OK") != NULL)
118     {
119       xSemaphoreGive(gsm_in_use); // GSM parsing done
120     }
```

Figure 24: LTE Module Response Parsing Code

The parseGSMData task is created whenever a response is received that ends with a newline character. In this task, first the data is copied over from the responseGsmBuffer (see Figure 22) which contains the response. This is done using a DMA transfer and since the length of the response is not known, it is kept track of when response is being received and added to the buffer. So, the responseLengthCopy is used to specify how much data to copy over to the local buffer tempBuffer. The response is then parsed using the flowchart in Figure 23. In the code snippet, we see, if the response is OK, the semaphore gsm_in_use is given to allow another command to be sent to the module.

SmartMesh IP API Library

To improve the readability of the main code, a library was developed for the SmartMesh IP API protocol. This would abstract sending and receiving commands and make parsing said commands much easier. A C++ class was developed which held all the methods that were part of the library and contained internal variables and buffers. To initialize the class, a UART class object would have to be passed in as a parameter. This is what the library would use to communicate with the network manager. For most command types, a function was created that could be called with the correct parameters to run the API command corresponding to the function. This way each command would not have to be constructed manually at the top level and instead a single function would just have to be called.

SmartMesh IP Packet Parsing Previous Semester

This section is not meant for reading consumption. However, the flow charts are mostly valid. SmartMesh IP packet parsing was an integral component of the entire new network manager that was to be designed. If a packet was not able to be processed properly, then data would be lost and not transmitted over to the cloud or GUI. Therefore, it was important to perfect the packet parsing part of the firmware and make sure there were no bugs. The packet parsing flowchart which was designed is shown below in Figure 25. The waiting for packet state is used to represent the blocked task while it is waiting for a semaphore to be posted. This semaphore is posted when a packet has been found which was discussed in a previous section. Once the semaphore has been unblocked, the task dynamically created a new task to parse the packet that came in. The reason this must be done is because the network manager can send up to 36 packets per second which would not all be able to be processed linearly without buffer corruption. Therefore, multithreading is used with each task having a local buffer which stores the packet it is parsing. Once the parsing is complete, the task will then delete itself since there is no more use for it. Overall, the packet parsing requires two separate tasks with one creating the other one.

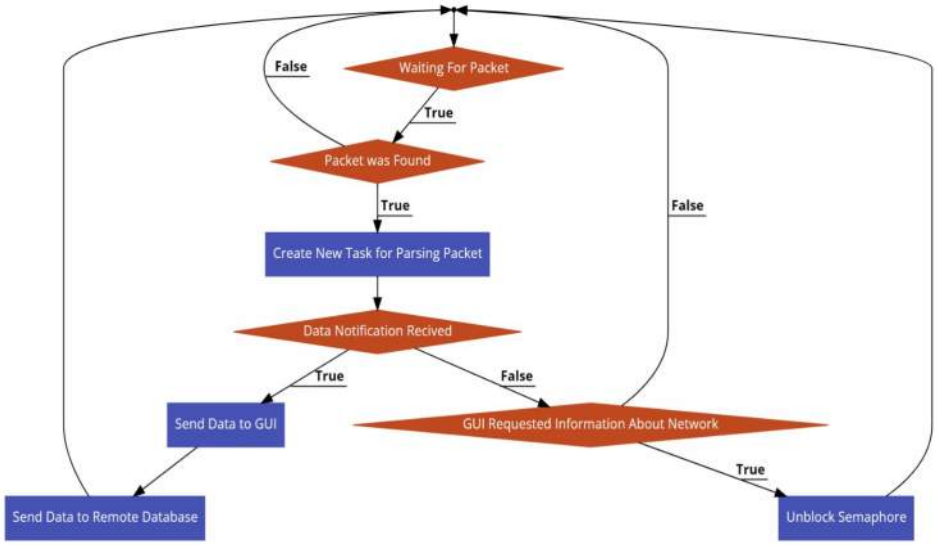

Figure 25 - Packet Parsing Flowchart

The code implementation for the first task is shown below in Figure 26. The infinite while loop is used since this task runs forever after its inception. Initially, the initialization command is sent to the network manager to start the connection process faster. Next the semaphore for a packet received is taken. However, until this semaphore is given by the packet collection interrupt handler, this task will remain in the blocked state and will not be able to continue. Once a packet is received, the task will then be unblocked and take the semaphore and create a new task which will then do the actual packet parsing. That way once another packet is received, this task can create a whole new separate task which will parse that packet potentially simultaneously as the first packet which was received.

```
157 void setupParse(void* unused){
158     api.mgr_init();// Initialize connection with the network manager
159 
160     while(1){
161       xSemaphoreTake(dataRecieved, portMAX_DELAY);
162 
163       // Create a new instance of smartmesh data parsing task
164       xTaskCreate(parseSmartmeshData, "Smart", 256, NULL, 32, NULL);
165     }
166   }
167 
```

Figure 26 - First Parsing Task

A portion of the second task code is shown below in Figure 27. When the task is first created, a local buffer is setup which stores the packet which was collected by the interrupt handler. The packet is then copied from the global buffer to the local buffer. The DMA is used for the actual copying to reduce CPU usage and improve performance. After this point, the packet can no longer be corrupted by a new packet which is coming in from the network manager.

```
53  // This task will be created spontaneously to parse a recieved packet
54  void parseSmartmeshData(void* unused){
55    uint8_t buffer[130];// Buffer for stored data
56 
57    // Copy data over from buffer in setupParse to task buffer
58    xSemaphoreTake(dma_in_use, portMAX_DELAY);
59    DMAC_REGS->DMAC_CHID = 0;
60    while(DMAC_REGS->DMAC_CHCTRLA != 0);// Check to see if DMA is still running
61    uint32_t *desc = (uint32_t*)0x30000000;
62    *desc++ = ((length+2) << 16)|0x0C01;
63    *desc++ = (uint32_t)(smartmeshData + smartmeshData[4] + 8);// Source address
64    *desc = (uint32_t)(buffer + smartmeshData[4] + 8);// Dest address
65    DMAC_REGS->DMAC_CHCTRLA = 0x2;// Enable the channel
66    DMAC_REGS->DMAC_SWTRIGCTRL |= 0x1;
67    xSemaphoreGive(dma_in_use);
68 
69    // TODO: finish parsing section
70    uint8_t packetType = buffer[2];
71    switch(packetType){
```

Figure 27 - Second Task Code

Once this process is started, the packet type is found by looking at the third byte of the data that is in the buffer. This can be seen by referencing Figure 10 which shows that data structure of a packet. Based on this packet, one of three options can happen. Either a semaphore is unblocked, a response is sent to the GUI, or a data packet is sent to the LTE module for uploading. For most of the tasks, the first

two options are done. These commands are executed because the user has requested some information or would like to change some parameters within the network manager. The final option is only running whenever a data packet is received from a mote. With this option, data is sent the GUI and to the cloud through the LTE module.

GUI Command Parsing

Whenever the user would like to either get some information from the network manager or set some parameters, a command is sent from the GUI to the MCU. Each command is a character possible followed by data if it is necessary. These commands are then sent through the Bluetooth module over to the MCU. A UART interrupt handler is used to collect the data and store it in a queue. A queue has several advantages that make it useful for storing and processing data. First, a queue is first-in-first-out which means whatever data comes in from Bluetooth will be the first to get processed. Next, since the Bluetooth commands are of least priority within the RTOS kernel, the queue allows for commands to be stored while some other tasks are executing. Finally, if the user were to spam multiple commands, the firmware would process each of them one-by-one whenever the task is active therefore not causing issues where commands do not execute and get dropped. The interrupt code is shown below in Figure 28. Once data coming in has been detected, the interrupt code is executed, and the data byte is stored within the queue.

```
234  void SERCOM0_Handler(void){// Bluetooth handler
235    uint8_t data = SERCOM0_REGS->USART_INT.SERCOM_DATA;
236    //bluetooth._printf("%c",data);
237    xQueueSendFromISR(bluetoothData, &data, NULL);// Send data to the bluetooth queue
238
239    NVIC->ICPR[0] |= (1 << 8);// Clear the interrupt
240  }
```

Figure 28 - Bluetooth Interrupt Handler

The flowchart for processing the commands that come in is shown below in Figure 29. While a character has not been received from the GUI, the task is stuck looking for a character. Getting data out

of a queue works similarly to taking a semaphore. Whenever there is nothing in the queue, the task will remain in the blocked state until some data is received, and the task is unblocked.

Once the character is received based on its value, either more data will be received from the queue, or some data will be received from the network manager and sent back to the GUI. Once a command has been executed successfully, a character is sent back to the GUI to acknowledge that the command has been executed successfully. The flowchart shows two example commands, however, there are many others which would not fit within the flowchart and therefore were omitted. All commands follow a similar procedure to the one shown for the two separate commands in the flowchart. The first one sets the network ID of the network manager and is represented by the character 'A'. Since the network ID is 2 bytes longs, extra data is then received through the queue which then runs a command to send the network ID. The final step is to respond back to the GUI with a character 'A'.

The second command is used to get network information such as the network ID, IPV6 address, and MAC address of the network manager. Since this command does not set any parameters, the task responds with the character 'C' followed by the information described above.

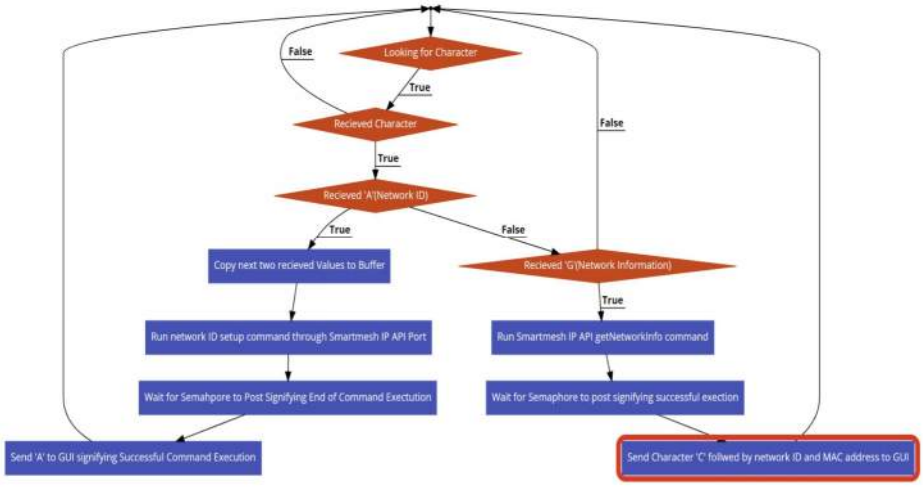

Figure 29 - GUI Command Parsing Flowchart

Part of the Bluetooth parsing task code is shown below in Figure 30. The while loop runs infinitely since the program can receive data through Bluetooth anytime during its lifetime. Next, the task is put into a blocked state until a character comes in through the Bluetooth UART port. Based on the character that was received, the firmware will execute either some SmartMesh IP command or send some other information to the GUI.

```
uint8_t recieved_data;
network_config config;
network_info info;

while(1){
  xQueueReceive(bluetoothData, &recieved_data, portMAX_DELAY);// Wait for data to come in
  uint16_t motes;
  switch(recieved_data){// Main state machine for bluetooth data
```

Figure 30 - Bluetooth Parsing Task

Storing Data to a Database

Once the SmartMesh IP packet parsing is complete, data is ready to be sent to a database for storage. The LTE module is already initialized at this point and the SIM7000 the two main internet protocols TCP and UDP. In this project TCP and HTTPS which is based on TCP were attempted. Sending

data is vastly different between HTTPS and TCP. This is because HTTPS has an added security layer and requires extra steps to successfully implement.

Initially, the goal was to store data to a blockchain via HTTPS post and get messages. This would work like how transactions are kept track of in blockchains and the data stored would be treated like one. The main reason one might choose to store data to a blockchain is for security. For one, data stored in a blockchain is immutable, meaning it cannot be altered. What's more? When storing data to the blockchain, the data is signed with a unique hash key and encrypted using public and private keys. The only way to decrypt it is to use those same keys which are only accessible to the user. Finally, blockchain is decentralized meaning not one entity has sole control of the data stored in the blockchain but rather, a group. This means the data is inaccessible to anyone other than the owner which includes the one who runs the servers like Ethereum or Bitcoin.

Due to some complications with how HTTPS method works when using an LTE module, this goal of blockchain storage was not satisfied this semester and will be further investigated next semester. The difference between HTTPS and TCP IP protocol will be discussed in the next couple sections.

Sending Data Through a TCP Port Connection

TCP (Transmission Control Protocol) is a standard for establishing and maintaining a reliable connection for transmission of data between two devices over an internet connection. It is a connection-oriented protocol, meaning that it establishes a reliable connection between two devices before transferring data. This allows for error checking and retransmission of lost packets, ensuring that the data is transmitted accurately and completely.

When sending data via TCP using the LTE module, the process is simple. All that needs to be done is to connect to a TCP port once network is established then send data to the server. Figure 31 below shows this process.

AT+CGATT?	
+CGATT: 1	//Data Service's status
OK	
	//Start task and set APN.
AT+CSTT="CMNET"	//The default APN is "CMNET", with no username or password. Check with local GSM provider to get the APN.
OK	
AT+CIICR	//Bring up wireless connection
OK	
AT+CIFSR	//Get local IP address
10.78.245.128	
AT+CIPSTART="TCP","116.228.221.51","8500"	//Start up the connection
OK	
CONNECT OK	//The TCP connection has been established successfully
AT+CIPSEND	//Send data to remote server, CTRL+Z (0x1a) to send. User should write data only after the promoting mark ">", and then use CTRL+Z to send. User can use command "AT+CIPSPRT" to set whether echo promote ">" after issuing "AT+CIPSEND".
> hello TCP serve	
SEND OK	//Remote server receives data. For TCP, "SEND OK" means data has been sent out and received successfully by the remote server, due to the TCP connection-oriented protocol.

Figure 31: Sending Data Through TCP

The first 3 commands are done in the initialization and the important thing when trying to connect to a TCP port is to first obtain an IP address. Although the module is connected to a network at this point, it does not have a local IP address assigned to it. Once the IP address is obtained, the TCP port connection can begin. After connecting to the port, data can be sent. One thing that is not shown in the figure is when sending data, the length of data needs to be specified in bytes and that includes the newline character and cartridge return. For example, below is a snippet of our code for connecting to the thingspeak server TCP port and sending data.

```
115  at_send_cmd("\r\nAT+CIPSHUT\r\n", AT_COMMAND_RUN);
116  at_send_cmd("AT+CSTT=\"hologram\"\r\n", AT_COMMAND_WRITE);
117  at_send_cmd("AT+CIICR\r\n", AT_COMMAND_RUN);
118  at_send_cmd("AT+CIFSR\r\n", AT_COMMAND_RUN);
119  at_send_cmd("AT+CIPSTART=\"TCP\",\"api.thingspeak.com\",\"80\"\r\n", AT_COMMAND_WRITE);
120  at_send_cmd("AT+CIPSEND=77\r\n", AT_COMMAND_WRITE);
121  at_send_cmd(buffer,AT_COMMAND_WRITE);
```

Figure 32: Code Snippet for Sending Data Through TCP

Looking at the second last command in line 120, we specify the length of the data we are sending, and it is 77 bytes. The actual data is about 8 bytes but the API commands we send to the ThingSpeak server are long and this will be discussed in an upcoming section.

Sending Data Via HTTPS Messages

HTTPS (Hypertext Transfer Protocol Secure) is a secure version of the HTTP used to transmit data on the internet. It is designed to protect the privacy and integrity of the data being transmitted between a client (such as a web browser) and a server (such as a website). HTTPS utilizes TCP for data transmission and works by establishing a secure, encrypted connection between the client and server using Transport Layer Security (TLS) or Secure Sockets Layer (SSL) protocols. This encryption helps to prevent third parties from intercepting and reading the data being transmitted between the two devices. The following commands need to be sent when sending an HTTPS post message.

```
AT+CSSLCFG="sslversion",1,3                    //Configure SSL/TLS version
OK

AT+SHSSL=1,"baidu_root_ca.cer"                 //Set HTTP SSL Configure
OK                                             //if you would skip certificate verify, use AT+SHSSL=1,"" instead
AT+SHCONF="URL","https://httpbin.org"          //Set connect server parameter
OK
AT+SHCONF="BODYLEN",1024                       //Set max body length
OK
AT+SHCONF="HEADERLEN",350                      //Set max header length
OK
AT+SHCONN                                      //Connect HTTPS server
OK
AT+SHSTATE?                                    //Get HTTP status
+SHSTATE: 1

OK
AT+SHCHEAD                                     //Clear HTTP header
OK
AT+SHAHEAD="Content-Type","application/js      //Add header content
on"
OK
AT+SHAHEAD="Cache-control","no-cache"          //Add header content
OK
AT+SHAHEAD="Connection","keep-alive"           //Add header content
OK
AT+SHAHEAD="Accept","*/*"                      //Add header content
OK
AT+SHBOD=" {\"title\":\"Hello http server\"}",29 //Set body content
OK
AT+SHREQ="/post",3                             //Set request type is POST
OK                                             //Get data size is 458.

+SHREQ: "POST",200,458
```

Figure 33: HTTPS Post Message AT Commands

As can be seen from Figure 33, the first thing that needs to be done when sending an HTTPS post message is to obtain an SSL certificate for the server the message is being sent to. Obtaining an SSL certificate is not a straightforward process and the easiest method we found is to use an SSL scanner. The one used in this project is a website named FairSSL and by providing a server address, it can find the certificates of said server. Now, this website gives multiple SSL certificates and the specific one needed by the LTE module is the Root CA certificate. So once that certificate is downloaded, it can be stored on the LTE module's memory. This is done using QPST, a tool used for flashing the SIM7000 module firmware and accessing the internal storage. After storing the certificate in the module's memory, it can then be used to configure the certificate for the server we're trying to connect to. For example, in figure 33, this is done in the second command and the certificate file is named baidu_root_ca.cer. The extension of this file specifies that it is a certificate file, and the certificate is a Root CA certificate. An SSL Root CA certificate looks like the one shown below in Figure 34.

```
static const char ssl_certificates[] =
    // Root CA for httpbin.org
    "-----BEGIN CERTIFICATE-----\r\n"
    "MIIENjCCAx6gAwIBAgIBATANBgkqhkiG9w0BAQUFADBvMQswCQYDVQQGEwJTRTEU\r\n"
    "MBIGA1UEChMLQWRkVHJ1c3QgQUIxJjAkBgNVBAsTHUFkZFRydXN0IEV4dGVybmFs\r\n"
    "IFRUUCBOZXR3b3JrMSIwIAYDVQQDExlBZGRUcnVzdCBFeHRlcm5hbCBDQSBSb290\r\n"
    "MB4XDTAwMDUzMDEwNDgzOFoXDTIwMDUzMDEwNDgzOFowbzELMAkGA1UEBhMCU0Ux\r\n"
    "FDASBgNVBAoTC0FkZFRydXN0IEFCMSYwJAYDVQQLEx1BZGRUcnVzdCBFeHRlcm5h\r\n"
    "bCBUVFAgTmV0d29yazEiMCAGA1UEAxMZQWRkVHJ1c3QgRXh0ZXJuYWwgQ0EgUm9v\r\n"
    "dDCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALf3GjPm8gAELTngTlvt\r\n"
    "H7xsD821+iO2zt6bETOXpClMfZOfvUq8k+0DGuOPz+VtUFrWlymUWoCwSXrbLpX9\r\n"
    "uMq/NzgtHj6RQa1wVsfwTz/oMp50ysiQVOnGXw94nZpAPA6sYapeFI+eh6FqUNzX\r\n"
    "mk6vBbOmcZ5ccbNQYArHE504B4YCqOmoaSYYkKtMsE8jqzpPhNjfzp/haW+710LX\r\n"
    "a0Tkx63ubUFfclpxCDezeWkWaCUN/cALw3CknLa0Dhy2xSoRcRdKn23tNbE7qzN\r\n"
    "E0S3ySvdQwAl+mG5aWpYIxG3pzOPVnVZ9c0p10a3CitlttNCbxWyuHv77+ldU9U0\r\n"
    "WicCAwEAAaOB3DCB2TAdBgNVHQ4EFgQUrb2YejS0Jvf6xCZU7wO94CTLVBowCwYD\r\n"
    "VR0PBAQDAgEGMA8GA1UdEwEB/wQFMAMBAf8wgZkGA1UdIwSBkTCBjoAUrb2YejS0\r\n"
    "Jvf6xCZU7wO94CTLVBqhc6RxMG8xCzAJBgNVBAYTAlNFMRQwEgYDVQQKEwtBZGRU\r\n"
    "cnVzdCBBQjEmMCQGA1UECxMdQWRkVHJ1c3QgRXh0ZXJuYWwgVFRQIE5ldHdvcmsx\r\n"
    "IjAgBgNVBAMTGUFkZFRydXN0IEV4dGVybmFsIENBIFJvb3SCAQEwDQYJKoZIhvcN\r\n"
    "AQEFBQADggEBALCb4IUlwtYj4g+WBpKdQZic2YR5gdkeWxQHIzZlj7DYd7usQWxH\r\n"
    "YINRsPkyPef89iYTx4AWpb9a/IfPeHmJIZriTAcKhjW88t5RxNKWt9x+Tu5w/Rw5\r\n"
    "6wwCURQtjr0W4MHfRnXnJK3s9EK0hZNwEGe6nQY1ShjTK3rMUUKhemPR5ruhxSvC\r\n"
    "Nr4TDea9Y355e6cJDUCrat2PisP29owaQgVR1EX1n6diIWgVIEM8med8vSTYqZEX\r\n"
    "c4g/VhsxOBi0cQ+azcgOno4uG+GMmIPLHzHxREzGBHNJdmAPx/i9F4BrLunMTA5a\r\n"
    "mnkPIAou1Z5jJh5VkpTYghdae9C8x49OhgQ=\r\n"
    "-----END CERTIFICATE-----\r\n"
    // Root CA for google.com
    "-----BEGIN CERTIFICATE-----\r\n"
```

Figure 34: SSL Certificate Example

The SSL certificate shown above contains Root CA for httpbin.org and google.com (not shown in the figure). Multiple Root CA certificates can be added to one file so that makes it easier to use one file for multiple servers and clients. Now that the certificate is configured, all that needs to be done is to

structure the message in the format we need. In the example, JSON format is used so body and header length as well as their content is specified and finally, the HTTPS message is sent. If 200 OK response is received from the server, the message is successfully sent.

The ThingSpeak Database

ThingSpeak is an IoT (Internet of Things) platform that allows users to collect, store, and analyze data from connected devices and sensors. It is a cloud-based platform that enables users to create dashboards and charts to visualize their data and to set up rules to trigger actions based on the data. The server provides APIs (Application Programming Interfaces) that allow users to send data to the platform from their devices and to retrieve data from the platform for use in their applications. It also provides integration with other IoT platforms and services, such as Google Maps and Amazon Web Services. ThingSpeak is widely used in a variety of applications, including environmental monitoring, industrial automation, and home automation. It is designed to be easy to use and is suitable for users with a range of technical backgrounds.

To use ThingSpeak, we need to sign up for an account on the platform and create a new channel to store the data. Each channel consists of a set of fields in which we can store the data and a set of associated metadata, such as the name and description of the channel. Once we have created a channel, we can send data to it from the LTE module using the ThingSpeak API. This can be done using a variety of programming languages, such as Python, C, or JavaScript. Below are some of the functions the API provides.

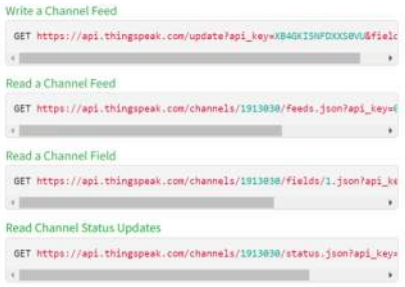

Figure 35: ThingSpeak API Requests

The API provides functions for sending data to specific fields in our channel and for updating the metadata associated with the channel. We can also retrieve data from the channel using the ThingSpeak API if we need to. In addition to using the API, we can also use the ThingSpeak web interface to visualize and analyze our data. The interface provides a range of charting and plotting tools that allow us to view the data in different ways and to set up rules to trigger actions based on the data. Below is a picture of the channel we set up for our system and some fields we made for motes that show some random data we sent to it for testing.

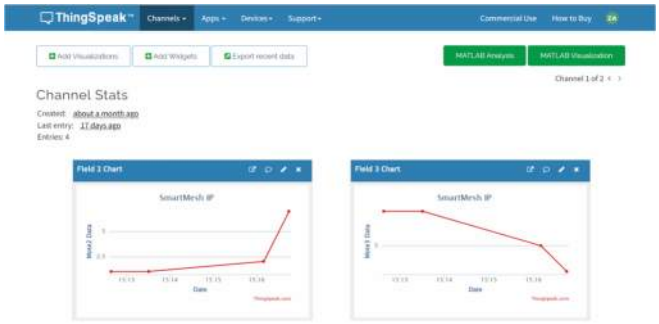

Figure 36: ThingSpeak Data Sent from LTE Module

The Firebase Database

Many developers have switched from using platforms like ThingSpeak to Firebase Realtime Database for several reasons. Firstly, ThingSpeak is primarily focused on IoT applications and lacks some of the advanced features needed for complex mobile and web applications. Firebase Realtime Database, on the other hand, is a general-purpose NoSQL database that can handle a wide range of applications.

Another key benefit of Firebase Realtime Database is its real-time synchronization feature, which automatically updates all connected clients in real-time. This is particularly useful for applications that require real-time collaboration or frequent data updates, such as multiplayer games or chat applications. In contrast, ThingSpeak requires manual refreshing of data on the client-side.

Firebase Realtime Database also provides robust security and access controls, ensuring that data is safe and only accessible by authorized users. This feature is crucial for applications that handle sensitive user data or financial information.

Firebase Realtime Database also offers a serverless architecture, meaning that developers do not need to manage a server or database infrastructure. Firebase automatically handles scaling, backups, and other administrative tasks, allowing developers to focus on building their applications.

Finally, Firebase Realtime Database has excellent integration with other Firebase services, such as Firebase Authentication, Firebase Cloud Messaging, and Firebase Hosting, making it easy to build and deploy end-to-end applications.

Firebase is a cloud-based platform for developing mobile and web applications, providing developers with a range of tools and services to build, test, and deploy applications quickly and efficiently. One of the services offered by Firebase is Realtime Database, which is a NoSQL cloud-hosted database that stores data in JSON format.

JSON, or JavaScript Object Notation, is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. JSON is widely used for transmitting data between a client and a server in web applications and mobile apps. JSON is a text format that uses a key-value pair structure, making it easy to represent complex data structures.

Firebase Realtime Database stores data in JSON format, which means that all the data is organized into a collection of JSON objects. Each object represents a piece of data, and it is identified by a unique key. The key-value pairs in each object represent the attributes and values of the data. For example, if you were storing data about a user, you might have an object that looks like this:

```
{
  "id": "1",
  "name": "John Smith",
  "email": "john.smith@example.com",
  "age": 35,
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "state": "CA",
    "zip": "12345"
  }
}
```

In this example, the object represents a user with an ID of 1, a name of John Smith, an email address of john.smith@example.com, an age of 35, and an address object that contains the user's street, city, state, and zip code.

Firebase Realtime Database stores data in a tree-like structure, where each object is a node in the tree. The root node of the tree represents the entire database, and each child node represents a subset of the data. Each node can have zero or more child nodes, and each child node can have its own child nodes. This allows for a flexible and scalable data model that can accommodate a wide range of applications.

One of the key features of Firebase Realtime Database is its real-time synchronization. When data is added, updated, or deleted in the database, all connected clients are immediately notified and updated in real-time. This makes it easy to build real-time collaborative applications such as chat applications, social networking apps, and multiplayer games.

In summary, Firebase Realtime Database stores data in JSON format, using a tree-like structure that allows for flexible and scalable data models. The real-time synchronization feature makes it easy to build real-time collaborative applications. Firebase Realtime Database is a powerful and versatile tool for building mobile and web applications.

Sending to the Firebase Database

To send data to a Firebase server from a SIM7000 module, you will need to follow the following steps:

1. Set up Firebase The first step is to set up a Firebase account and create a new Firebase project. Once you have created a project, you will need to add a Realtime Database to the project. The Realtime Database is where all your data will be stored and synced in real-time.
2. Create a SIM7000 module Next, you will need to create a SIM7000 module that will be used to send data to the Firebase server. You will need to configure the module to connect to the internet using the cellular network.
3. Set up Firebase Authentication Firebase Authentication allows you to authenticate users and secure your database. You can use Firebase Authentication to ensure that only authenticated users can access your database. You will need to configure Firebase Authentication to allow your SIM7000 module to access the database.

4. Set up Firebase Realtime Database Once you have set up Firebase Authentication, you will need to set up the Firebase Realtime Database. You will need to create a reference to the database and define the data structure that you will be sending from the SIM7000 module.

5. Send Data to Firebase To send data to Firebase from the SIM7000 module, you will need to use the Firebase REST API. The REST API allows you to send data to Firebase using HTTP requests. You will need to send a POST request to the Firebase database URL, along with the data that you want to send.

In summary, to send data to a Firebase server from a SIM7000 module, you will need to set up Firebase, create a SIM7000 module, set up Firebase Authentication, set up Firebase Realtime Database, and send data to Firebase using the REST API. By following these steps, you will be able to send data to a Firebase server in real-time using the SIM7000 module.

Firmware Changes from the Previous Semester

Throughout designing the firmware in the first semester, many flaws were found that made editing and maintaining the code much more difficult. Additionally, when running the system with many motes connected, the firmware would randomly crash or get stuck on a semaphore due to incorrect task priorities and priority inversion. Partially through this semester, we also encountered the Keil MDK 32kB code limit imposed since the free version was being used. Thus, we had to migrate to the ARM GCC compiler, which is free to use, however, must be manually setup. These three items were the main changes that were made from the last semester.

GCC Compiler Migration

Migrating from Keil MDK to GCC with Visual Studio Code (VSCode) was a significant undertaking for our team. However, we were motivated by the benefits that this transition could bring to our development process.

The Keil MDK is a widely used Integrated Development Environment (IDE) that supports the ARM architecture. While it is a reliable and well-established tool, it has some limitations. One of these limitations is that it can be inflexible, and it may not support all of the features that our team requires for our development process. Additionally, Keil MDK is a proprietary tool, which means that we are limited to the tools and libraries that are provided by the Keil ecosystem.

On the other hand, GCC is a highly configurable open-source compiler that supports multiple architectures. By migrating to GCC, we would have access to a broad range of optimizations, and we could tailor our code to specific hardware platforms. Additionally, GCC is an open-source tool, which means that we would have access to a wider range of open-source libraries and tools. These benefits, combined with the flexibility and configurability of VSCode, made the transition to GCC with VSCode an attractive proposition for our team.

The first step in implementing the transition was to install the ARM GCC toolchain, which was available from the official ARM website. We also installed and set up VSCode on our machines, along with the necessary extensions to work with the ARM GCC toolchain. This process was relatively straightforward, and we were able to complete it without any significant issues.

The next step was to configure our projects to work with the new toolchain. This involved modifying the build system, linker scripts, and other project settings to use the new toolchain. The process of configuring the projects was well documented, and we were able to complete the transition without any major issues.

After configuring the projects, we tested them to ensure that everything was working as expected. The testing process involved building the projects and running them on our target devices. We used debugging tools to troubleshoot any issues that arose during the testing process.

Refactoring the code was also necessary to ensure compatibility with the new toolchain. We made changes to the code to ensure that it was built correctly with the new compiler. While this process required some effort, the benefits of the transition made it well worth the effort.

Since implementing the transition, our development process has been smoother and more efficient. We have enjoyed greater freedom in our development process and have leveraged the benefits of open-source software to improve the quality and efficiency of our code. We have also been able to take advantage of the greater configurability of GCC to tailor our code to specific hardware platforms. Additionally, we have been able to use a wider range of open-source libraries and tools, which has improved the quality and efficiency of our code.

Migrating from Keil MDK to GCC with VSCode was a significant undertaking for us. However, the benefits of the transition, including greater flexibility, configurability, and compatibility with open-source tools and libraries, have made it well worth the effort. By following the necessary steps, we were able to successfully transition to GCC with VSCode and take advantage of the full range of features and benefits that GCC has to offer.

RTOS Task Flow Changes

With the random bugs that were occurring in the firmware, an important part of this was making each of the tasks more stable and easier to debug. To accomplish this, four tasks were designed, each using a queue-based method of flow control where data would come in and be processed in a state machine which will be discussed in the next section. This design was already being used for the Bluetooth task; however, it was also implemented within the other tasks. The four commands would be parsing Bluetooth data, parsing SmartMesh IP packets, parsing LTE commands, and sending LTE data to the cloud.

A separate LTE sending task was designed to mitigate some of the biggest RTOS issues that were encountered. Whenever a packet was received from the MCU, the sending process would take approximately a second and, in the meantime, no other packets could be processed. This caused some semaphores to expire therefore causing system crashes. With this new task, however, the sending process was done in a separate task, and a queue was used to collect the data that was to be sent to the cloud. The queue structure used is shown below in Figure 37. A pointer to 8 bytes of data was used along with a 64-bit value representing the MAC Address of the mote sending the data. The queue dataIn was used to get data from other tasks with the public function send_to_send_queue used to send a packet to the cloud. The function send_task is the task function which runs throughout the entire life cycle of the system.

```
typedef struct{
    uint8_t *data;
    uint64_t mac_address;
}packet_t;

static QueueHandle_t dataIn = xQueueCreate(10, sizeof(packet_t));

void send_task(void *unused);
void send_to_send_queue(packet_t send_data);
```

Figure 37 - Send Task Defines

The SmartMesh IP parsing task was also redesigned due to inconsistent communication with the network manager. Thus, most of the handling code was moved from within the interrupt function over to a new task which was just used for parsing data. The dynamic task creation process was removed and instead a queue-based system was implemented which would process one task at a time. The new interrupt handler is shown below in Figure 38. The queue apiPacketBuffer is used as a temporary storage place for bytes that are coming in from the handler. A redesigned state machine is then used to parse the packets that are coming in.

```
extern "C"{
    void SERCOM1_Handler(void){
        uint8_t data = SERCOM1->USART.DATA.reg;

        configASSERT(xQueueSendFromISR(apiPacketBuffer, &data, nullptr) == pdTRUE);

        NVIC_ClearPendingIRQ(SERCOM1_IRQn);// Clear the interrupt
    }
};
```

Figure 38 - SmartMesh Interrupt Handler

The state machine defines for parsing are shown below in Figure 39. The STATE_IDLE state is where the packet searching will commence. In the STATE_START_FOUND state, the data is copied into a special buffer and the task looks for the end flag. Finally, in the STATE_END_FOUND state, the end flag has been found and the packet has been verified successfully, thus the packet parsing function can now commence. Two other public functions were also provided for use in the Bluetooth task which remove many of the global variables that were needlessly defined in the header files.

```
typedef enum{
    STATE_INIT,
    STATE_IDLE,
    STATE_START_FOUND,
    STATE_END_FOUND,
    STATE_ERROR,

    STATE_AMT
}states;

typedef uint8_t packet_type_t;

void smartmesh_parse_task(void *unused);
bool mgr_connection_status(void);
BaseType_t get_packet_status(packet_type_t packet);
```

Figure 39 - SmartMesh Parsing Task Defines

State Machine Redesign

For each of the state machines within the different tasks, the state machines were redefined to make them more readable as well as robust. First an Enum was added which would contain all the

different state defines. Next, a state handler table was added which would select the next state. Finally, if necessary, callback functions could be included which would run code that should be executed within the current state. An example of the state machine setup is shown below in Figure 40. Two different Enums were created both for the normal states, as well as for the error states. Next, a custom structure was defined which holds the callback functions as well as the state that they are tied to. This setup is shown below in Figure 41. Each line contains one of the states that were shown in the previous figure. The function pointers each contains one parameter which points the UART Bluetooth instance which is used to send data to the GUI/phone app.

```
// Bluetooth states
typedef enum{
    STATE_RUNNING,
    STATE_NETID,
    STATE_JKEY,
    STATE_MOTE_LIST,
    STATE_GET_NET_INFO,
    STATE_MOTE_INFO,
    STATE_MGR_CONNECTED,
    STATE_CLEAR_STATS,
    STATE_CMD_ERROR,

    STATE_AMOUNT// Must be last
}bt_states_t;

// Error States
typedef enum{
    ERROR_NONE,
    ERROR_NO_MOTES_FOUND,
    ERROR_SEM_TAKE,
    ERROR_UNDEF,

    ERROR_AMT
}error_states_t;
```

Figure 40 - State Machine Defines

```
// Private variable definitions
// Structure for event holding event handlers
struct{
    bt_states_t state;// Optional can be removed
    void (*handler)(UART &bluetoothInstance);
}static handler_table[STATE_AMOUNT] = {
    [STATE_RUNNING]         = {STATE_RUNNING,       &runningCallback},
    [STATE_NETID]           = {STATE_NETID,         &setNetID_Handler},
    [STATE_JKEY]            = {STATE_JKEY,          &setJKeyHandler},
    [STATE_MOTE_LIST]       = {STATE_MOTE_LIST,     &getMoteListHandler},
    [STATE_GET_NET_INFO]    = {STATE_GET_NET_INFO,  &getNetInfoHandler},
    [STATE_MOTE_INFO]       = {STATE_MOTE_INFO,     &getMoteInfoHandler},
    [STATE_MGR_CONNECTED]   = {STATE_MGR_CONNECTED, &isMgrConnectedHandler},
    [STATE_CLEAR_STATS]     = {STATE_CLEAR_STATS,   &clearStatsHandler},
    [STATE_CMD_ERROR]       = {STATE_CMD_ERROR,     &errorHandler}
};
static bt_states_t curr_state = STATE_RUNNING;
static error_states_t error_state = ERROR_NONE;
```

Figure 41 - Callback Function Table

Extremely Robust Error Handling

We have implemented exception handling using C++, which has been highly effective in handling runtime errors. Exceptions are thrown when an error occurs, and they can be caught and handled by the code that caused the error. This has enabled the firmware to continue running even if an error occurs, which is critical in safety-critical systems. Additionally, we use assertions to test assumptions made by the code. We have implemented assertions using the configAssert() FreeRTOS macro which uses the C assert() macro, which is a part of the standard library. The assertions have been highly effective in catching bugs during development and testing, and they have helped us to identify the source of errors quickly. This has helped us to quickly identify and resolve any issues that arise.

Graphical User Interface

The below Figure 42, shows the main page of the graphical user interface (GUI). To the left are the different windows that the user can select. There are multiple sections of the main page GUI, broken down into group boxes. In the first group box, com port settings, this is where the user can set up the Bluetooth connection to the system. The next group box over, manager setup, the network ID and join

key are set up. For the join key, the user needs to enter in 32 characters and a counter is displayed below the textbox to notify the user. The last group box in the top row is blockchain setup, for storing data. A URL link needs to be provided as well as a private key. Now for the first group box in the middle row, Mote controls, a list of MAC addresses will be shown in the Mote list drop down. The user can select any of the MAC addresses from the list and use it to get Mote details in the next group box over. The Mote details group box will show important information stored for that specific Mote. With the MAC address selected, this single Mote can be assigned a specific plot and name of series using the single Mote setup. Finally, at the bottom is the Network Manager details group. This group box shows the necessary details of the current Network Manager connected to.

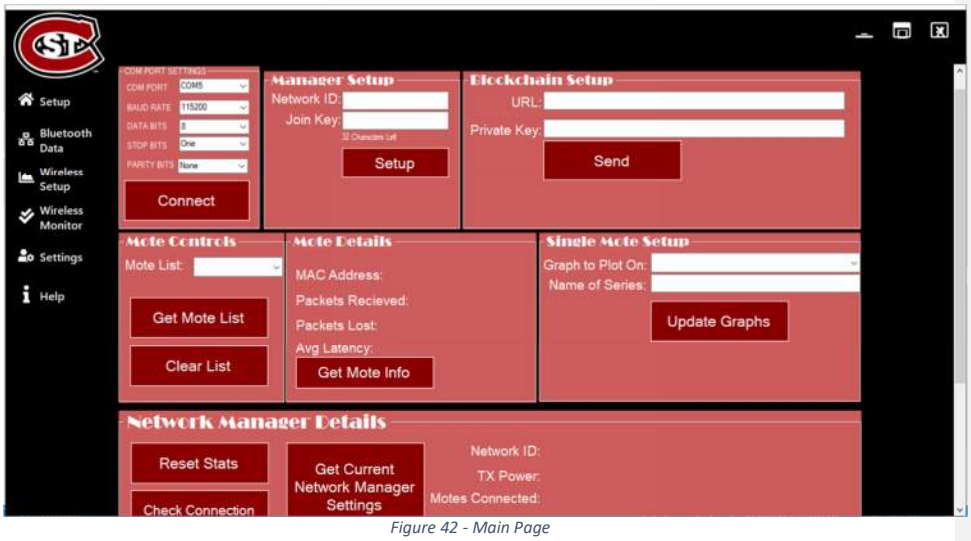

Figure 42 - Main Page

Another important page in GUI is the Bluetooth Data page. This page shows the different plots of the different Motes that are currently in the system. The below figure shows the Bluetooth Data page. At the center are displayed the plots of the different Motes. As more Motes are discovered, more plots will be created for the new Motes. To the left of the figure are different settings that the user can

specify. The first setting that the user can specify is choosing a certain plot and giving it a name. Also, there are blacklisting and whitelisting options. This setting lets the user choose which data should be displayed. Finally, the time scale can be changed for the plots. There is a list of time scale options, or the user may choose a custom time frame.

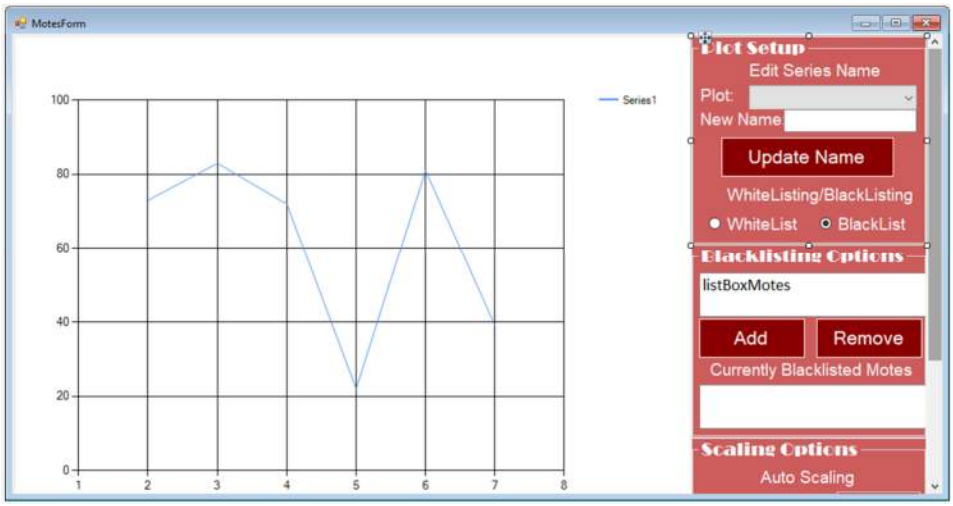

Figure 43-Bluetooth Data Page

The final page of the GUI was the cloud data page. In this page, the user is able to get the data from the cloud, export to excel and plot it out. At the bottom left, the user will click the Get List of Motes button and a list of different motes will be listed out in the above textbox. The user can then

select a mote and specify the time frame they would like to view the data. This data can then be exported to excel as well as the option of plotting out the data.

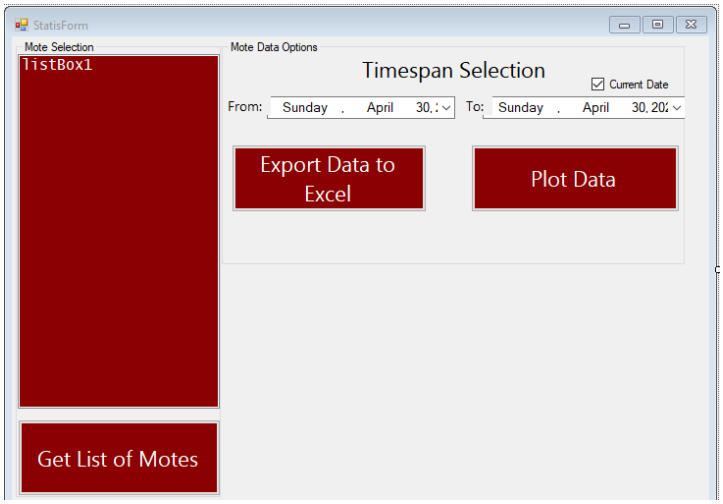

Figure 44-Cloud Data Page

The functionality of the com port settings group box can be seen in the figures below. The user selects a com port, baud rate, data bits, stop bits, parity, and clicks connect. Once the connect button is clicked the GUI will check if com port is already open and if it is not, the following code shown below is executed. This will try to establish a connection between the GUI and the system. If the connection was successful, the status label will show that the connection was successful. If the connection was unsuccessful, the status label will show that the connection was unsuccessful.

Figure 45-COM Port Connect GUI

```
public void connectButtonAutomaticClick()
{
    try
    {
        serialPort1.PortName = comPortComboBox.Text;
        Universal.portType = serialPort1.PortName.Replace("COM", "");
        Universal.portTypeInt = Convert.ToInt32(Universal.portType);

        Universal.portType = "API";
        serialPort1.BaudRate = 115200;
        serialPort1.DtrEnable = false;

        serialPort1.Open();


        connectButton.Text = "Close";
        serialPort1.DataReceived += new SerialDataReceivedEventHandler(SerialPort_DataReceived);

        toolStripStatusLabel1.Text = Universal.portType + " Port Connected";
        toolStripStatusLabel1.ForeColor = Color.Green;
        toolStripProgressBar1.Value = 100;

        //this.Invoke(new EventHandler(LiveUPDATE));
    }
    catch (Exception e)
    {
        MessageBox.Show(e.Message);
        toolStripStatusLabel1.Text = "ERROR";
        toolStripStatusLabel1.ForeColor = Color.Red;
        toolStripProgressBar1.Value = 0;
    }

}
```

Figure 46-COM Port Connect Code

Some basic setup functionality of the GUI can be seen in the below figures, Figure 47 and Figure 48. The Manager Setup group box allows the user to choose a network ID and join key. There is a character counter under the join key text box as it requires an input of a 32-character number. Figure 48 shows the code that is used for sending the entered values. First the values are extracted from the textboxes and checked if the entered values are valid. Initially a character 'A' is sent to let the Manager know that the network ID is about to be sent following by the user entered value. Next, a character 'B' is sent to let the manager know that the join key is about to be sent followed by the entered join key.

Figure 47-Manager Setup

```
1 reference
private void button1_Click(object sender, EventArgs e)
{
    try
    {
        short net = short.Parse(textBox1.Text);// Get the network ID
        byte[] netid = BitConverter.GetBytes(net);
        (netid[1], netid[0]) = (netid[0], netid[1]);// Convert to big endian
        if (textBox2.Text.Length > 32)
            throw (new Exception("Invalid Join Key"));

        byte[] jkey = new byte[16];// Setup byte array for serial port sending
        if(textBox2.Text.Length % 2 == 0)// Fix error for odd length of strings
            jkey = StringToByteArray(textBox2.Text);
        else
            jkey = StringToByteArray(textBox2.Text + "0");

        Array.Resize<byte>(ref jkey, 16);

        serialPort1.Write("A");// Send the network ID
        serialPort1.Write(netid, 0, 2);
        Thread.Sleep(50);
        serialPort1.Write("B");// Send the join key
        serialPort1.Write(jkey, 0, 16);
        //serialPort1.Write("F");// Reset the system
    }
    catch(Exception ex)
    {
        MessageBox.Show("Please Enter Valid Values!!\nError: " + ex.Message);
    }
}
```

Figure 48-Manager Setup Code

For the Mote controls group box, as seen in Figure 49, the user will be able to select MAC address, Mote, from a drop-down list. To fetch the list of Motes the Get Mote List button will need to be pressed. This can also be cleared by pressing the Clear List button. To get the list of Motes, first the ascii character 'C' will be sent to request the list from the manager as can be observed in Figure 50. After the request, the manager will send back the list and the GUI will collect the data as can be observed in Figure 51. Each MAC address will be read from the serial port and converted to string. This string value

will then be added to the drop-down list. This will continue to repeat until all the MAC addresses have been collected.

Figure 49-Mote Control

```
private void button2_Click(object sender, EventArgs e)
{
    try
    {
        serialPort1.Write("C");// Get the mote list and process in the handler
    }
    catch
    {
        MessageBox.Show("Error! Serial Port not Open");
    }

}
```

Figure 50-Send Character 'C'

```
case 'C':// Mote list was recieved
    byte[] curr_mac_addr = new byte[8];
    serialPort1.Read(buffer, 0, 1);
    while (buffer[0] == 'D')// Will send 'E' once complete
    {
        serialPort1.Read(curr_mac_addr, 0, 8);// Read the mac_addr
        string temp = Universal.ByteArrayToString(curr_mac_addr);
        this.Invoke(new MethodInvoker(delegate
        {
            CBoxMoteList.Items.Clear();// Remove previous values
            CBoxMoteList.Items.Add(temp);// Add the mac address
        }));
        serialPort1.Read(buffer, 0, 1);
    }
    break;
```

Figure 51-Receive MAC Addresses

After the user selects the MAC address, the Mote details can be viewed using the Mote Details group box as seen in Figure 52. The Mac address, packets received, packets lost, and average latency will be displayed. Once the Get Mote Details button has been pressed the GUI will send an ASCII character 'I' as well as reconfiguring the Mote MAC address to send. This process can be seen in Figure 53. To receive the data, this can be seen in Figure 54. First the MAC address is received, converted to string, and

formatted. Next, the GUI will receive several packets that have been received. The same procedure as the previous was done for get the number of packets lost and latency.

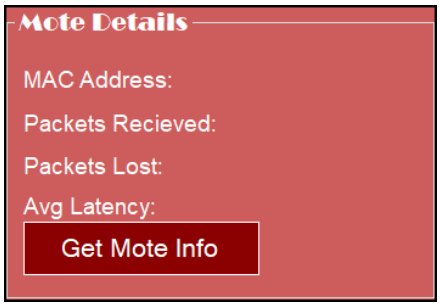

Figure 52-Mote Details

```
1 reference
private void moteInfoButton_Click(object sender, EventArgs e)
{
    serialPort1.Write("I");// Get mote information
    UInt64 mac = UInt64.Parse(CBoxMoteList.Text, System.Globalization.NumberStyles.HexNumber);
    byte[] macAddr = BitConverter.GetBytes(mac);
    (macAddr[0], macAddr[1], macAddr[2], macAddr[3], macAddr[4], macAddr[5], macAddr[6], macAddr[7]) =
        (macAddr[7], macAddr[6], macAddr[5], macAddr[4], macAddr[3], macAddr[2], macAddr[1], macAddr[0]);
    serialPort1.Write(macAddr, 0, 8);
}
```

Figure 53-Send Character 'I'

```
case 'F':// Mote Information was recived
    byte[] information = new byte[8];
    while (serialPort1.BytesToRead < 8) ;
    serialPort1.Read(information, 0, 8);// Get the mac address
    string MacAddr = BitConverter.ToString(information).Replace("-", "");
    MacAddr = Regex.Replace(MacAddr, ".{2}", "$0:");// Format the string;
    MacAddr = MacAddr.Remove(MacAddr.Length - 1);
    this.Invoke(new MethodInvoker(delegate
    {
        moteMacAddrLabel.Text = "Mac Address: " + MacAddr;
    }));
    byte[] info = new byte[4];
    while (serialPort1.BytesToRead < 4) ;
    serialPort1.Read(info, 0, 4);// Get packets
    this.Invoke(new MethodInvoker(delegate
    {
        (info[0], info[1], info[2], info[3]) = (info[3], info[2], info[1], info[0]);
        packetsRecievedLabel.Text = "Packets Recieved: " + BitConverter.ToUInt32(info, 0).ToString();
    }));


    while (serialPort1.BytesToRead < 4) ;
    serialPort1.Read(info, 0, 4);// Get lost packets
    (info[0], info[1], info[2], info[3]) = (info[3], info[2], info[1], info[0]);
    packetsLostLabel.Text = "Packets Lost: " + BitConverter.ToUInt32(info, 0).ToString();

    while (serialPort1.BytesToRead < 4) ;
    serialPort1.Read(info, 0, 4);// Get latency
    (info[0], info[1], info[2], info[3]) = (info[3], info[2], info[1], info[0]);
    avgLatencyLabel.Text = "Avg Latency: " + BitConverter.ToUInt32(info, 0).ToString();
    break;
```

Figure 54-Receive Mote Details

With the MAC address selected of the Mote, the Single Mote Setup group box can be used, as seen in Figure 55. This allows the user to specify which Mote will be plotted on different plots as well as giving a specified name. There will be a drop-down list of the different plots the Mote can be plotted on. Once the Update Graphs button is clicked, the plots will be updated.

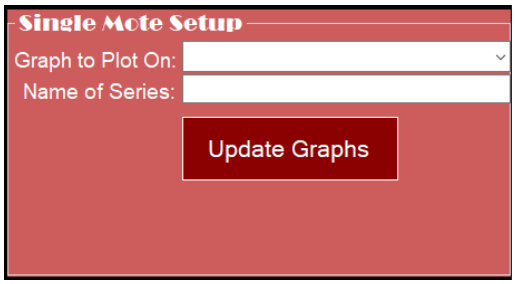

Figure 55-Single Mote Setup

For the final part of the main page of the GUI is the Network Manager Details group box, as seen in Figure 56. To the right of the figure is where the different information will be displayed. To the left of the figure is where the control buttons are for the group box. When the Reset Stats button is pressed, an ascii character 'H' will be sent to the manager to reset the statistics, as seen in Figure 57. To check the network manager connection, pressing the Check Connection button, an ascii character 'E' is initially sent. Once receiving an ascii character 'K,' the buffer is checked for a '0.' If the first character in the buffer is '0,' the network manager is not connected, otherwise it is connected. This process can be in Figure 58. If the Get Current Network Manager Settings button is clicked, the GUI will send an ascii character 'G.' In response, if an ascii character 'K' is received then the Network ID, TX Power, Motes Connected, and IPV6 will be received and displayed in the group box.

Figure 56-Network Manager Details

```
1 reference
private void button8_Click(object sender, EventArgs e)
{
    serialPort1.Write("H");// Reset manager stats
}
```

Figure 57-Send Character 'H'

```
case 'K':// Is network manager connected
    byte[] buf = new byte[1];
    serialPort1.Read(buf, 0, 1);
    if (buf[0] == '0')
        MessageBox.Show("Not Connected");
    else
        MessageBox.Show("Connected");
    break;
```

Figure 58-Manager Connection Status

```
case 'J':// Get network manager config
    try
    {
        byte[] Data = new byte[16];
        serialPort1.Read(Data, 0, 2);
        (Data[0], Data[1]) = (Data[1], Data[0]);
        UInt16 netID = BitConverter.ToUInt16(Data, 0);
        netidLabel.Text = "Network ID: " + netID.ToString();
        serialPort1.Read(Data, 0, 1);// Get the TX power
        txPowerLabel.Text = "TX Power: " + Data[0].ToString();
        serialPort1.Read(Data, 0, 2);// Get number of motes connected
        (Data[0], Data[1]) = (Data[1], Data[0]);
        motesConnLabel.Text = "Motes Connected: " + BitConverter.ToUInt16(Data, 0);
        serialPort1.Read(Data, 0, 16);// Get the IPV6 address
        string IPV6 = BitConverter.ToString(Data).Replace("-", "");
        IPV6 = Regex.Replace(IPV6, ".{4}", "$0:");// Format the string;
        IPV6 = IPV6.Remove(IPV6.Length-1);
        IPV6Label.Text = "IPV6: " + IPV6;
        MessageBox.Show("Updated");
    }
    catch
    {
        MessageBox.Show("Error!");
    }
```

Figure 59-Network Manager Statistics

Now in the Bluetooth Data page of the GUI is a Plot Setup group box as seen in Figure 60. Here the different plots can be selected and renamed. Also, there are blacklisting and whitelisting options to control what data should be displayed. Once the Update Name button has been clicked, the selected plot will be updated as can be seen in Figure 61. For blacklisting and whitelisting procedure, this can be

seen in Figure 62. If the check state is change for the blacklisting or whitelisting the list of blacklisted or whitelisted Motes will be updated as well as the different plots enabled and disabled.

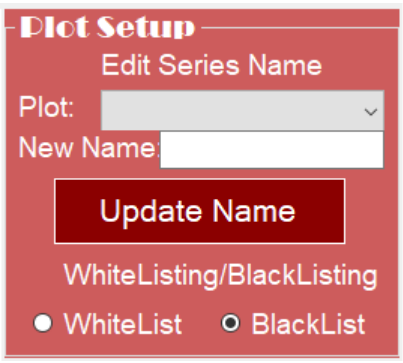

Figure 60-Plot Setup

```
// Update the name of a graphed series
1 reference
private void updateNameButton_Click(object sender, EventArgs e)
{
    string series = CBoxPlotList.Text;
    string label = textBoxNewName.Text;
    chart1.Series[series].LegendText = label;// Update the legend label
}
```

Figure 61-Update Name of Graph

```
1 reference
private void radioButtonWhitelist_CheckedChanged(object sender, EventArgs e)
{
    if (radioButtonBlacklist.Checked)
    {
        groupBoxLising.Text = "Blacklisting Options";
        labelListing.Text = "Curretly Blacklisted Motes";
        listBoxBlackWhiteListed.Items.Clear();// Clear all items

        // Enable all curves by default
        for(int i = 0;i < chart1.Series.Count; i++)
        {
            chart1.Series[i].Enabled = true;
        }
    }
    else if (radioButtonWhitelist.Checked)
    {
        groupBoxLising.Text = "Whitelisting Options";
        labelListing.Text = "Curretly Whitelisted Motes";
        listBoxBlackWhiteListed.Items.Clear();// Clear all items

        // Disable all curves by default
        for (int i = 0; i < chart1.Series.Count; i++)
        {
            chart1.Series[i].Enabled = false;
        }
    }
}
```

Figure 62-Blacklisting and Whitelisting

Underneath the plot settings are the blacklisting options as seen in the figure below, Figure 63. This is where the user can specify which Motes are to be blacklisted. For every Mote that is blacklisted, it will be added to the list of blacklisted Motes. If the user decides to Remove a Mote from the backlisted Motes, the user can select the Mote from the list and click the Remove button. Figure 64 shows the process for removing the Mote from the list. First the mode is checked and if in blacklisting mode, the Mote will then be removed from the list. Now for the add button being clicked, the code can be observed in Figure 65. Just like Figure 64, the mode will be checked, and the Mote will be added to the list.

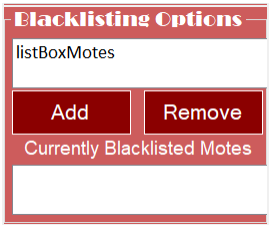

Figure 63-Blacklisting Options

```
private void removeButton_Click(object sender, EventArgs e)
{
    if (radioButtonBlacklist.Checked)// Blacklist mode
    {
        if (listBoxMotes.SelectedIndex == -1)// Error checking
            return;

        string mac = listBoxMotes.Items[listBoxMotes.SelectedIndex].ToString();
        if (listBoxBlackWhiteListed.Items.Contains(mac) == false)// Was never blacklisted
            return;

        listBoxBlackWhiteListed.Items.Remove(mac);

        // Run the actual blacklisting procedure
        chart1.Series[mac].Enabled = true;
    }
    else// Whitelisting mode
    {
        if (listBoxMotes.SelectedIndex == -1)// Error checking
            return;

        string mac = listBoxMotes.Items[listBoxMotes.SelectedIndex].ToString();
        if (listBoxBlackWhiteListed.Items.Contains(mac) == false)// Was already blacklisted
            return;

        listBoxBlackWhiteListed.Items.Remove(mac);

        // Run the actual blacklisting procedure
        chart1.Series[mac].Enabled = false;
    }
}
```

Figure 64-Remove Button Clicked

```
1 reference
private void addButton_Click(object sender, EventArgs e)
{
    if (radioButtonBlacklist.Checked)// Blacklist mode
    {
        if (listBoxMotes.SelectedIndex == -1)// Error checking
            return;

        string mac = listBoxMotes.Items[listBoxMotes.SelectedIndex].ToString();
        if (listBoxBlackWhiteListed.Items.Contains(mac) == true)// Was already blacklisted
            return;

        listBoxBlackWhiteListed.Items.Add(mac);

        // Run the actual blacklisting procedure
        chart1.Series[mac].Enabled = false;
    }
    else// Whitelisting mode
    {
        if (listBoxMotes.SelectedIndex == -1)// Error checking
            return;

        string mac = listBoxMotes.Items[listBoxMotes.SelectedIndex].ToString();
        if (listBoxBlackWhiteListed.Items.Contains(mac) == true)// Was already blacklisted
            return;

        listBoxBlackWhiteListed.Items.Add(mac);

        // Run the actual whitelisting procedure
        chart1.Series[mac].Enabled = true;
    }
}
```

Figure 65-Add Button Clicked

For the final part of the Bluetooth Data page, is the scaling options. This group box allows the user to specify the time frame of the plots. There are two options, auto scaling and manual scaling. For the Auto scaling, there will be a dropdown of different time frames that the user can specify. To do manual scaling, the start and end times need to be provided. After pressing the scale button, the time frames of the plots will be updated.

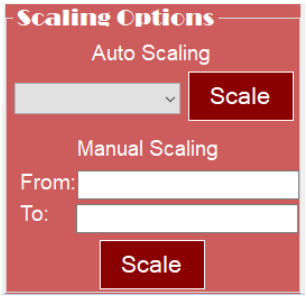

Figure 66-Scaling Options

Now switching to the cloud data page functions, the Get List of Motes button is for retrieving the different motes stored in Firebase. The list of motes will be displayed in the textbox above the

button. This code can be seen in the bellow figure. First, the GUI will try to get a response from Firebase. This response will contain the data stored for the different motes. Next, a loop is used to list out the different mote addresses into the textbox.

```
1 reference
private async void button1_Click(object sender, EventArgs e)
{   //get the mote list
    try
    {
        FirebaseResponse resp = await client.GetAsync("/");
        JObject data = resp.ResultAs<JObject>(); //The response wil

        JToken myToken = data.First;
        listBox1.Items.Clear();
        while (myToken != data.Last)    //add motes to the list
        {
            listBox1.Items.Add(myToken.Path);
            myToken = myToken.Next;
        }
        listBox1.Items.Add(myToken.Path);
    }
    catch
    {

    }
}
```

Figure 67-Get List of Motes

The next part of the code is for when the user clicks on the plot button. First, a plot is created and then the plot is shown. Next, the data for the specified mote will be retrieved from Firebase. This data includes the time of the data readings as well as the data reading at that time. Once the data has been retrieved, a loop will be used for plotting out the data. What data will be plotted will depend on the time frame specified by the user. The if elseif statement will plot out either of the two scenarios that the user specified for the time frame.

```
1 reference
private async void button3_Click(object sender, EventArgs e)
{   //plot data
    Remote_Plot plot = new Remote_Plot();
    plot.Show();

    try
    {
        FirebaseResponse resp = await client.GetAsync("/" + listBox1.SelectedItem.ToString());
        JObject data = resp.ResultAs<JObject>(); //The response will contain the data being retreive
        JToken token = data.First;
        while (token != data.Last)
        {
            //plot out data
            if (checkBox1.Checked && (DateTime.Parse(token.Path) >= dateTimeFrom.Value))
                plot.plotPoint(DateTime.Parse(token.Path), Convert.ToInt32(token.First));
            else if ((DateTime.Parse(token.Path) > dateTimeFrom.Value) && (DateTime.Parse(token.Path
                plot.plotPoint(DateTime.Parse(token.Path), Convert.ToInt32(token.First));

            token = token.Next;
        }
        plot.plotPoint(DateTime.Parse(token.Path), Convert.ToInt32(token.First));
    }
    catch
    {

    }
}
```

Figure 68-Plot Firebase Data

The final part of the cloud data page is from exporting mote data into an excel sheet. First, a check is performed to make sure that Excel exists on the computer and installed correctly. Next, the data is retrieved from Firebase and Excel variables are defined and initialized. A loop is then used to go through the different times and values and stored into the Excel sheet. Before storing all the data, the first two cells are assigned “Timestamp” and “Value” for the time value and the data value. The file is then saved and Excel is closed.

```
try
{
    saveFileDialog1.ShowDialog();
    Excel.Application xlApp = new Microsoft.Office.Interop.Excel.Application();

    if (xlApp == null)  //check if excel exsists
    {
        MessageBox.Show("Excel is not properly installed!!");
        return;
    }

    FirebaseResponse resp = await client.GetAsync("/" + listBox1.SelectedItem.ToString());

    JObject data = resp.ResultAs<JObject>(); //The response will contain the data being retreived

    Excel.Workbook xlWorkBook;
    Excel.Worksheet xlWorkSheet;
    object misValue = System.Reflection.Missing.Value;

    xlWorkBook = xlApp.Workbooks.Add(misValue);
    xlWorkSheet = (Excel.Worksheet)xlWorkBook.Worksheets.get_Item(1);

    xlWorkSheet.Cells[1, 1] = "Timestamp";  //header cells
    xlWorkSheet.Cells[1, 2] = "Value";
    int currCell = 2;
    JToken token = data.First;
    while (token != data.Last)  //store time and value
    {
        xlWorkSheet.Cells[currCell, 1] = token.Path;
        xlWorkSheet.Cells[currCell, 2] = token.First;
        token = token.Next;
        currCell++;
    }
    xlWorkSheet.Cells[currCell, 1] = token.Path;
    xlWorkSheet.Cells[currCell, 2] = token.First;

    //save file and exit
    xlWorkBook.SaveAs(saveFileDialog1.FileName, Excel.XlFileFormat.xlWorkbookNormal, misValue, misValue, misValue, misValue, Excel.XlSaveAsAccessMode.x
    xlWorkBook.Close(true, misValue, misValue);
    xlApp.Quit();

    Marshal.ReleaseComObject(xlWorkSheet);
    Marshal.ReleaseComObject(xlWorkBook);
    Marshal.ReleaseComObject(xlApp);
}
```

Figure 69-Export to Excel

Android Phone Application

The main page for the android app can be seen in the figure below, Figure 70. In between the two huskies is the Bluetooth status. This status lets the user know if the Bluetooth for the device is currently on or off. Underneath are buttons that have different functions. The first button, the turn on button, will turn on the device Bluetooth and change the Bluetooth symbol to on. For the turn off button, the device Bluetooth will be turned off and the symbol will change to off. To connect to the system, press the connect button and the disconnect button to disconnect. For the final three buttons, they open separate pages. The setup button will open the setup page where the user is able to set up the network ID and join key. The chart button will open and show a plot of the data stored in Firebase. The final button will open the statistics page, this is where the user can see the manager and specific Mote details.

Figure 70-Android App Main Page

Most of the important functions of the app were in the main activity of the app. These functions are shown in Figure 71 Through Figure 74. First, when the connect button is pressed, the app will connect to the Bluetooth device and start the background running task. This task will be continuously reading the Bluetooth data. Once data has been received, it will go through the different cases and perform certain actions. Most of the time it is for data storage, which stores to the dataHolder class and can be used to access the stored data. Finally, there is the Bluetooth class which is for connecting, disconnecting, receiving and sending data.

```
//connect button pressed
mDiscoverBtn.setOnClickListener(new View.OnClickListener() {
    @RequiresApi(api = Build.VERSION_CODES.S)
    @Override
    public void onClick(View view) {
        bluetooth.connect();
        if(connected) {
            showToast( msg: "Connected");
            new LongRunningTask().execute();    //run background tasks
        }
        else
            showToast( msg: "Failed to Connect");
    }
});
```

Figure 71-Connect Button Pressed

```
public class LongRunningTask extends AsyncTask<Void, Void, Void> {
    @Override
    protected Void doInBackground(Void... voids) {
        while(!isCancelled){
            int temp =  bluetooth.readInputStreaming();
            switch (temp) {
                case 'A':
                    showMessage( msg: "Network ID Set");
                    break;
                case 'B':
                    showMessage( msg: "Join Key Set");
                    break;
                case 'C':
                    byte[] curr_mac_addr = new byte[8];
                    int addressCounter = 0;
                    while (temp == 68 | temp == 67) {
                        for (int a = 0; a < 8; a++) {  //get Mote MAC Address
                            curr_mac_addr[a] = (byte) bluetooth.readInputStreaming();
                        }
                        System.out.println(curr_mac_addr);
                        addressCounter = addressCounter + 1;
                        String s = ByteToHex(curr_mac_addr);
                        MainActivity.dataHolder.saveMacAddressList(s, addressCounter);   //save addresses
                        temp = bluetooth.readInputStreaming();
                    }
                    break;
```

Figure 72-Background Running Task

```
//Data storage to be used between different activities/pages
public static class dataHolder{
    private static String[] MacAddresses = new String[10]; //motes MAC address
    private static String MoteInfoAddress;
    private static String packetsReceived;
    private static String packetsLost;
    private static String averageLat;
    private static Long NetworkID;
    private static byte TXPower;
    private static Long motesConnected;
    private static String IPV6;

    public static String[] getMoteAdressesList(){return MacAddresses;}
    public static String getCurrentMoteAddress(){return MoteInfoAddress;}
    public static String getPacketsReceived(){return packetsReceived;}
    public static String getPacketsLost(){return packetsLost;}
    public static String getAverageLat(){return averageLat;}
    public static Long getNetworkID(){return NetworkID;}
    public static byte getTXPower(){return TXPower;}
    public static Long getMotesConnected(){return motesConnected;}
    public static String getIPV6(){return IPV6;}

    public static void saveMacAddressList(String MAcAddress, int AddressCounter){
        MacAddresses[AddressCounter] = MAcAddress;
    }
```

Figure 73-Data Holder Function

```
public static class bluetooth{

    //connect to bluetooth device
    public static void connect(){
        try {
            btSocket = HC05.createInsecureRfcommSocketToServiceRecord(mUUID);
            btSocket.connect();
            connected = true;
        }catch (IOException e){
            System.out.println(e);
        }
    }

    //disconnect from device
    public static void disconnect(){
        try{
            btSocket.close();
            connected = false;
        }catch(IOException e){
            System.out.println("Failed to disconnect");
        }
    }
```

Figure 74-Bluetooth Function

In the below figure, Figure 75, is displayed what happens when the on and off buttons are pressed on the Android app. First, when the user presses the on button, the Bluetooth symbol will change to on, and a message will pop up. When looking at the Bluetooth icon on the device, the Bluetooth is turned on. The same can be said for the off button. The Bluetooth symbol will update to off, a message will be displayed, and the device Bluetooth will be turned off. The next two figures, Figure 76 and Figure 77, show the code for how the device Bluetooth is turned on and off. For turning on Bluetooth, Figure 76, a listener is set onto the button, waiting to be pressed. Once the button is pressed, a message will be displayed and a request to enable Bluetooth will be made. If Bluetooth is off, an additional message will pop from the device asking if it is okay for the app to access Bluetooth. To turn off Bluetooth, Figure 77, first the app will check if Bluetooth is enabled or disabled. If Bluetooth is enabled, it will proceed to turning Bluetooth off. If it is already off, a message will show saying that Bluetooth is already off.

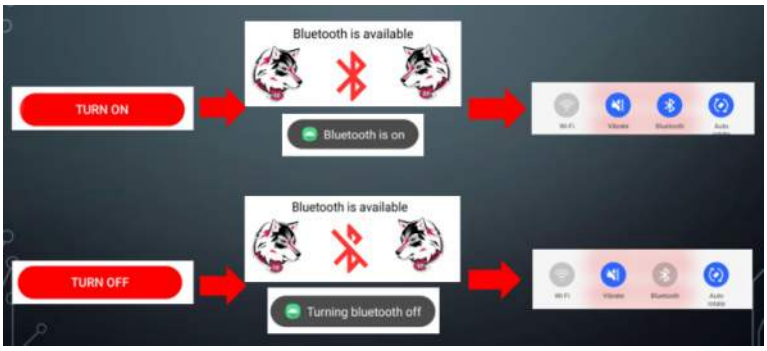

Figure 75-Turning Bluetooth On and Off

```
//on button click
mOnBtn.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        if (!mBlueAdapter.isEnabled()) {
            showToast( msg: "Turning On Bluetooth...");
            //intent to on bluetooth
            Intent Intent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
            //android.getResult.launch(Intent);
            startActivityForResult(Intent, REQUEST_ENABLE_BT);
        }

    }
});
```

Figure 76-On Button Clicked

```
    //off button click
    mOffBtn.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            if(mBlueAdapter.isEnabled()){
                mBlueAdapter.disable();
                showToast( msg: "Turning bluetooth off");
                mBlueTv.setImageResource(R.drawable.ic_action_off_red);
            }
            else{
                showToast( msg: "Bluetooth is already off");
            }
        }
    });
```

Figure 77-Off Button Clicked

Once the user has pressed the set-up button, the set-up page will be opened. The page will be like the one in the figure below, Figure 78. In this figure it is displayed that once the network ID or join key has been sent, it will wait for the system to respond. If the network ID or join key has been set successfully, a toast message will be displayed. The sending and receiving data can be seen in the next two figures, Figure 79 and Figure 80. For sending the network ID, a on key listener was set up and within the on key listener it waits for the done key to be pressed. Once the done key is pressed, the numerical value entered in the network ID text box will be extracted as well as some initialized variables. If connected to Bluetooth, the app will first send ascii character ‘A’ to let the manager know that network ID is being sent, followed by the entered network ID. The same process is followed for sending the join

key but sending an ascii character ‘B’. As a 32-character number must be provided for the join key, an extra step had to be done. After sending over the entered join key, zeros will be sent to get the character count to 32. The background task will receive either a character ‘A’ or ‘B’ for the network ID or join key being set successfully.

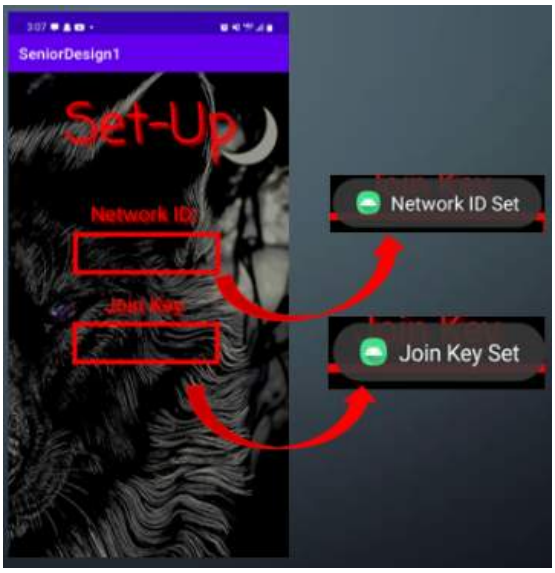

Figure 78-Setup Page

```
NetworkID.setOnKeyListener(new View.OnKeyListener(){
    public boolean onKey(View v, int keyCode, KeyEvent event) {
        if (keyCode == KeyEvent.KEYCODE_ENTER) {
            String value = mNetworkID.getText().toString();
            int NetworkIDEntered = Integer.parseInt(value);

            //set the Network ID
            MainActivity.bluetooth.writeOutputStream( character: 'A');
            MainActivity.bluetooth.writeOutputStream((char) ((NetworkIDEntered >> 8) & 0xFF));
            MainActivity.bluetooth.writeOutputStream((char) (NetworkIDEntered & 0xFF));
        }
        return false;
    }
});
```

Figure 79-Send Network ID

```
public void addKeyListener(){
    NetworkID = (EditText) findViewById(R.id.NetworkID);
    JoinKey = (EditText) findViewById(R.id.JoinKey);

    JoinKey.setOnKeyListener(new View.OnKeyListener(){
        boolean sentOnce = false;
        public boolean onKey(View v, int keyCode, KeyEvent event) {
            if (keyCode == KeyEvent.KEYCODE_ENTER&&(!sentOnce)) {
                sentOnce = true;
                String value = mJoinKey.getText().toString();
                int JoinKeyEntered = Integer.parseInt(value);
                int CharsLeftToSend;
                //write to bluetooth
                MainActivity.bluetooth.writeOutputStream( character: 'B');
                for (int i = 0; i < Integer.toHexString(JoinKeyEntered).length(); i++) {
                    MainActivity.bluetooth.writeOutputStream((char)JoinKeyEntered);
                }
                CharsLeftToSend = (32 - Integer.toHexString(JoinKeyEntered).length())/2;
                for (int i = 0; i < CharsLeftToSend; i++) {   //join key needs to be 32 characters
                    MainActivity.bluetooth.writeOutputStream((char) 0x00);
                }
                return true;
            }
            return false;
        }
    });
```

Figure 80-Send Join Key

If the user wishes to view collected data in a plot, the chart page can be used as seen in Figure 81. In this page the user will be able to specify which MAC address, Mote, that they would like plotted. There will also be a time frame option to view data at different time frames. In Figure 81-Chart Page, a time frame was selected of one week for a specific Mote. The Motes data stored in Firebase is plotted out on the plot. Now, in the next two figures, Figure 82 and Figure 83, the different variables are defined. A listener was set on the time frame drop down. This listener will check what was selected from the drop down and go through the switch case statement. Figure 84 shows the code for if the selection was one minute. First, the time and the values are retrieved and converted to float. Next, the data is filtered for the specified time frame using the filterDataByTimeFrame() function. The data will then be plotted out based on the filtered data. The same code was used for the different times frames. To get the data from Firebase, the code in Figure 85 was used. The date strings will need to be converted into date objects. Also, the values are not stored in the correct order and will need to be sorted out. Finally,

Figure 86 shows the filtering function. This function will take the date list, values, and time frame, and will return the filtered data.

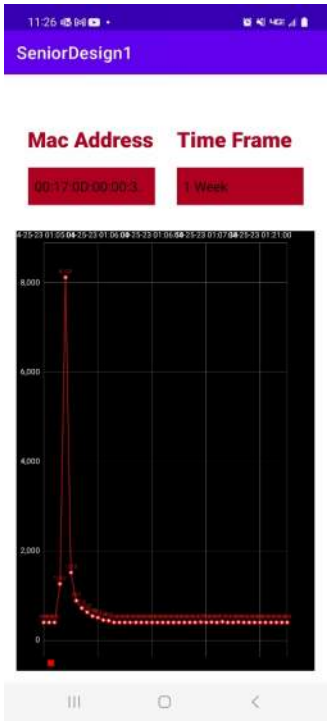

Figure 81-Chart Page

```
public class Chart extends AppCompatActivity {

    //XYPlot plot;
    LineChart plot;
    Spinner spin1, spin2;
    int counter;

    private static final String[] items = new String[]{"00:01","00:02", "00:03"};
    FirebaseDatabase firebaseDatabase;
    DatabaseReference databaseReference;
    Set<String> keys;
    Collection<String> values;
    // create an ArrayList to hold the Entry objects
    ArrayList<Entry> entries = new ArrayList<>();
    List<String> filteredLabels = new ArrayList<>();

    // create a SimpleDateFormat object to parse the date strings
    SimpleDateFormat sdf = new SimpleDateFormat( pattern: "MM-dd-yy HH:mm:ss");

    boolean starting=true;
```

Figure 82-Chart Page Variable Definitions

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_chart);

    plot = findViewById(R.id.chart);
    spin1 = findViewById(R.id.spinner); //mac address drop down
    spin2 = findViewById(R.id.spinner2);

    firebaseDatabase = FirebaseDatabase.getInstance();
    databaseReference = firebaseDatabase.getReference( path: "00:17:00:00:00:32:02:08");
    getdata();

    //for the time frame drop down
    String[] timeFrame = {"All","1 Minute", "1 Hour", "1 Day", "1 Week", "1 Month"};
    ArrayAdapter<String> aa = new ArrayAdapter<String>( context: Chart.this,android.R.layout.simple_spinner_dropdown_it
    spin2.setAdapter(aa);

    //for the address drop down
    String[] addresses = {"00:17:00:00:00:32:02:08"};
    ArrayAdapter<String> aa1 = new ArrayAdapter<String>( context: Chart.this,android.R.layout.simple_spinner_dropdown_it
    spin1.setAdapter(aa1);
```

Figure 83-Chart Page onCreate Variable Definitions

```
case "1 Minute":    // if user selects last minute                                    ⚠ 39 ⚠ 1
    //set the time frame and get the data
    TimeFrame selectedTimeFrame = TimeFrame.MINUTE;
    List<String> listKeys = new ArrayList<>(keys);
    List<Entry> listValues = new ArrayList<>();
    int i = 0;
    for (String value : values) {   //convert the stored value from string to float
        listValues.add(new Entry(i, Float.valueOf(value)));
        i++;
    }
    //filter for the specific time frame
    List<Entry> filteredData = filterDataByTimeFrame(listValues, listKeys, selectedTimeFrame);
    LineDataSet dataSet = new LineDataSet(filteredData,  label: "Data");
    dataSet.setCircleColor(Color.RED);  //plot and adjust the plot design
    dataSet.setColor(Color.RED);
    dataSet.setValueTextColor(Color.RED);
    LineData data = new LineData(dataSet);
    plot.clear();
    plot.setData(data);
    plot.setBackgroundColor(Color.BLACK);
    XAxis xAxis = plot.getXAxis();
    xAxis.setValueFormatter(new IndexAxisValueFormatter(filteredLabels));
    xAxis.setTextColor(Color.WHITE);
    YAxis yAxis = plot.getAxisLeft();
    yAxis.setTextColor(Color.WHITE);
    break;
```

Figure 84-Time Frame Listener

```
private void getdata() {
    // calling add value event listener method
    // for getting the values from database.
    databaseReference.addListenerForSingleValueEvent(new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot snapshot) {

            Map<String, String> map = (Map<String, String>) snapshot.getValue();
            keys = map.keySet(); // Retrieve all the keys
            values =  map.values(); // Retrieve all the values

            System.out.println(map);
            System.out.println(keys);
            System.out.println(values);

            // convert the date strings to Date objects
            Date date;
            List<Date> dates = new ArrayList<>();
            SimpleDateFormat format = new SimpleDateFormat( pattern: "MM-dd-yy HH:mm:ss");
            for (String key : keys) {
                try {
                    date = sdf.parse(key);
                    dates.add(date);
                } catch (ParseException e) {
                    // handle parse exception
                }
            }
```

Figure 85-Get Data from Firebase

```
private List<Entry> filterDataByTimeFrame(List<Entry> originalData, List<String> originalLabels, TimeFrame timeFrame){
    List<Entry> filteredData = new ArrayList<>();
    //List<String> filteredLabels = new ArrayList<>();

    long now = System.currentTimeMillis();
    long timeFrameMillis = timeFrame.getMillis();

    //filter out for the specific time frame
    for (int i = 0; i < originalData.size(); i++) {
        Entry entry = originalData.get(i);
        String label = originalLabels.get(i);
        long timestamp = parseTimestampFromLabel(label);

        if (now - timestamp <= timeFrameMillis) {
            // Add the data point to the filtered data
            filteredData.add(entry);
            filteredLabels.add(label);
        }
    }

    return filteredData;
}
```

Figure 86-Filter by Time Frame

The user is also able to view the manager and mote details using the stats page. On this page the user can view the network ID, TX power, motes connected, and IPV6 for the manager. The connection of the manager can be checked, and the statistics can be retrieved or reset. For the mote details, first the list of motes must be retrieved. The available motes will be listed in a drop-down list where the user will select the mote and press the get mote info button. Now the information for the mote will be displayed. This information includes the MAC address, packets received, packets lost, and average latency. Each button sends its own character/command whenever pressed. Now the background running task will handle the receiving process of the data, one example can be seen in Figure 88. If receiving a character 'C,' the MAC addresses are being received. For each MAC address being received, it will store into the dataHolder class. This class is then used by the stats page to display the information. Another example can be seen in Figure 89 and Figure 90. There is a on-click listener when the check connection button is pressed, this will send a character 'E.' In the background, a character 'K' was received with the connection status. The connection status is then displayed to the user. Yo Cucumber

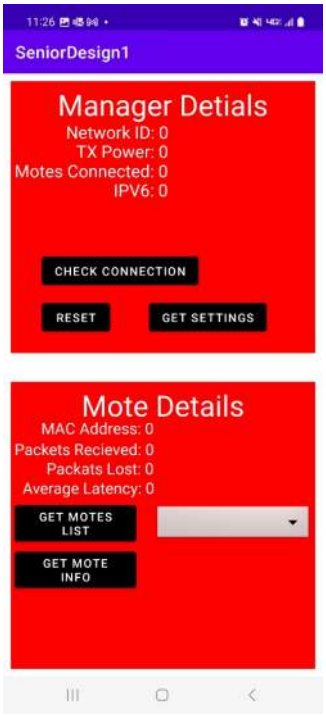

Figure 87-Stats Page

```
case 'C':   //getting the mote MAC addresses
    byte[] curr_mac_addr = new byte[8];
    int addressCounter = 0;
    while (temp == 68 | temp == 67) {
        for (int a = 0; a < 8; a++) {  //get Mote MAC Address
            curr_mac_addr[a] = (byte) bluetooth.readInputStreaming();
        }
        System.out.println(curr_mac_addr);
        addressCounter = addressCounter + 1;
        String s = ByteToHex(curr_mac_addr);
        MainActivity.dataHolder.saveMacAddressList(s, addressCounter);
        temp = bluetooth.readInputStreaming();
    }
    break;
```

Figure 88-Get Mote MAC Addresses

```
//check manager connection
mCheckManagerConnection.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        MainActivity.bluetooth.writeOutputStream( character: 'E');
    }
});
```

Figure 89-Check Manager Connection Click

```
case 'K':   //manager connection
    byte[] buf = new byte[1];
    buf[0] = (byte) bluetooth.readInputStreaming();
    if (buf[0] == '0') {
        showMessage( msg: "Manager not Connected");
    } else {
        showMessage( msg: "Manager Connected");
    }
    break;
```

Figure 90-Check Manager Connection Response

Power Management System Design

The system needs two output voltages 3.3V for Network Manager and 4V for LTE module; Also, the maximum current of 600mA is needed. The power supply is a DC-DC buck converter is fed from a rechargeable battery and straight from solar to DC-DC power supply. The battery must be charged from a solar panel, which is controlled by a charge controller. See the block diagram in Figure 91 below for overview of solar charge controller and power supply.

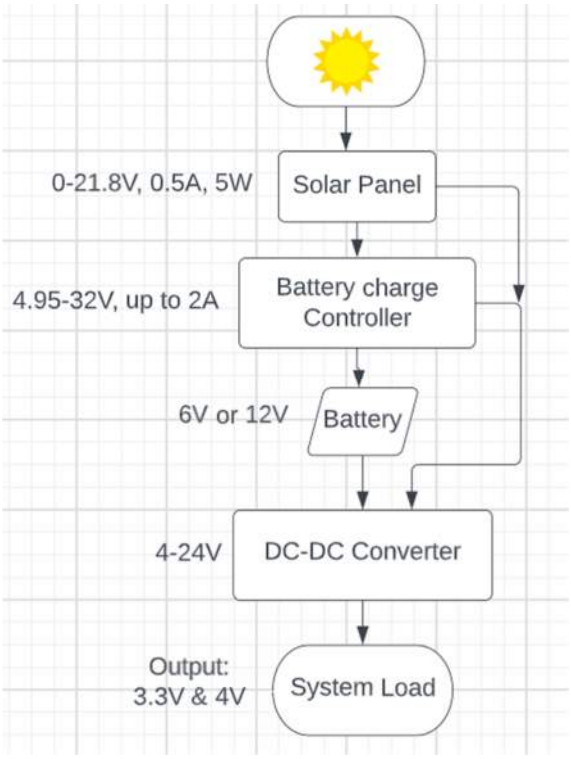

Figure 91 - Power Supply Block Diagram

The block diagram above shows the starting of the system which is the solar panel of 21.8V and 0.5A to charge the battery through charge controller; the battery voltage will be bucked to 3.3V and 4V from the DC-DC converter. The system might need more two outputs, one 5V for Bluetooth and the other one will be for future extension. The two outputs are connected to the load to feed the system. The components selection of this design will be discussed next.

Main Power Management Components selection

1- **ECO-WORTHY 5W Solar Panel:**

Selecting the Specification for solar panel to feed the system based on maximum current of 0.6A to charge a battery of 6V up to 12V; this solar panel was chosen at random, and we might need a bigger size to be compatible to charge a 2.9Ah battery. The table below shows the specification of the solar panel.

Description	Value
Rated Power	5W
The open circuit voltage V_{oc}	21.6V
Working Voltage (V_{op})	18V
Short circuit current (I_{sc})	0.305A
Working current (Iop)	0.277A
Output Tolerance	±3%
Temperature range	-40°Cto +80°C

2- Battery Charge Controller

The Power Tracking 2A Battery Charger for Solar Power LT3652 was chosen based on its specification and features. The LT3652 is a step-down battery charger that operates from a 4.95V up to 32V input voltage range. It provides a constant-current and constant-voltage charge, with maximum charge current externally adjustable up to 2A. The charger employs a 3.3V float voltage feedback reference, so any desired battery float voltage up to 14.4V can be adjustable with a resistor divider [6]. As mentioned, the selection was made because it has the input voltage range and output charging range compatible for what the solar panel, rechargeable battery, and DC-DC converter need to be operated. Besides the voltage operation range and charging range, this chip has temperature sensing to protect the battery from high temperature and overvoltage protection to protect the battery from over charging while under-voltage

protection stops the battery from being discharging beyond its capacity. Plus, the last reason for selecting this chip is for the maximum peak power tracking MPPT, which is a method to keep the output of the solar panels as high as possible regardless of environmental factors such as varying temperature or weak sun ray.

3- **Dual, 2-Phase Synchronous Step-Down Switching Controller LTC3850**

LTC3850 is a high performance dual synchronous step-down switching regulator controller that operates in an input range from 4V up to 24V. The selection of this chip is based on its features that are needed for this design; it includes ±1% 0.8V output voltage accuracy. These two outputs are adjustable by resistor divider are needed to power the Network Manager and LTE module. Also, high efficiency up to 95%, DCR current sensing, and fixed frequency from 250KHz up to 780KHz [7].

Power Supply Design

The DC-DC converter was our priority to design and implement before the battery charger. The design was started by obtaining the information of the load that needed to be powered. The system load needs to output power one is 3.3V for Network Manager and 4V for LTE module; the maximum current needed is 0.6A. The LTC3850 step-down buck converter was the chosen chip to achieve the output to feed the system. The design started with simulation and then is implemented in PCB.

LTC3850 Design Procedure

Designing a dual 2-phase synchronous step-down switching controller based on the LTC3850 involves multiple steps, including selecting components, calculating component values, and creating a printed circuit board (PCB) layout. Here's a step-by-step design procedure:

1) Study the datasheet:

Familiarize with the LTC3850 datasheet, which provides essential information about the IC, its features, and its operation. Understanding the specifications and requirements will help make informed design choices.

2) Determine the input and output voltage requirements:

Identify the input voltage range and the output voltage requirements for the application. The LTC3850 supports a wide input voltage range from 4V to 38V and provides two output channels, each capable of delivering up to 5A; it is 2A in this design and solar panel of 21.6V.

3) Set the switching frequency:

Choose an appropriate switching frequency for the application, which is 750KHzis used, considering factors such as efficiency, component size, and EMI. The LTC3850 supports switching frequencies ranging from 250 kHz to 750 kHz. Lower frequencies typically result in smaller inductor sizes, while higher frequencies allow for smaller capacitors and inductors, reducing overall solution size.

4) Calculate component values:

Calculate the values for external components, such as the inductors, input and output capacitors, feedback resistors, and compensation components. These component values will affect the controller's performance, efficiency, and stability.

5) Choose appropriate components:

Select high-quality components that can handle the voltage and current requirements of the design. Make sure to choose components with appropriate temperature ratings, as the application may experience temperature variations.

6) Create a schematic:

Using an Altium Designer, create a schematic that includes the LTC3850 IC and all required external components. Make sure to follow the application circuits provided in the datasheet as a reference.

7) Design the PCB layout:

Using a PCB Altium Designer, create a PCB layout based on the schematic. Follow best practices for PCB design, such as ensuring proper grounding, minimizing trace lengths, and using a ground plane. Additionally, follow any specific layout recommendations provided in the datasheet.

8) Verify and optimize the design:

Simulate the design using a LTSpice to ensure it meets the requirements. Make any necessary adjustments to component values or the PCB layout to optimize performance, efficiency, and stability.

9) Prototype and test:

Once the design is finalized, order a prototype PCB from JLCPCB, and assemble the LTC3850. Test the performance of the LTC3850 controller in various operating conditions to ensure it meets your requirements.

10) Iterate and finalize:

If necessary, make any adjustments to the design based on the testing results. Iterate the design process until it's a functional and reliable LTC3850 dual 2-phase synchronous step-down switching controller.

In this design, these steps were followed, and the design created an efficient and reliable step-down switching controller based on the LTC3850 IC that meets the specific application requirements.

Design Calculation and Components Selection

The instructions for the design and components selection were followed by the datasheet, which provides the formula for calculating the output voltage, current, and other parameters. The maximum current used in the calculation is 2A and can exceed this value if needed. The calculations are as follow:

1- Setting The Output Voltage (Pin V_{FB1} and Pin V_{FB2} are in pin 6 and 8)

These pins are called error amplifier feedback inputs; they receive feedback voltage from external resistive dividers; setting R_A to be 20KΩ and R_B can be found from the equation below.

V_{out} = 0.8V(1+R_B/R_A) = 3.3V = 0.8V(1+RB/20kΩ) = 63.4kΩ

2- Setting R_{sense} ($Sense^+$ and $Sense^-$ are Pin 2, 3 and Pin 11, 12)

These pins are current sense comparators which are connected to the output with a resistor to sense the current. To calculate the sense resistor values for two outputs using the equation below:

R_{sense} = $V_{sense(max)}$/$I_{load(max)}$+ΔI_L/2 =40mV/5A+1.5/2) =0.008Ω

$V_{sense(max)}$ is Maximum Current Sense Threshold can be found in the electrical characteristics in datasheet to be 40mV. $I_{load(max)}$ is the maximum load current assuming 5A. ΔI_L is peak to peak ripple current.

3- Inductors Selection

The value of the inductor is calculated and selected based on 1.75A, that is 35% max. current ripple assumption of 5A; the highest value of ripple current occurs at the maximum input voltage using the formula below to obtain the value of the inductor.

$$L = \frac{V_{out}}{f \times \Delta I_{L(max)}} \times \left(1 - \frac{V_{out}}{V_{in(nom)}}\right)$$

$$L1 = \frac{3.3}{500Khz \times 1.75} \times \left(1 - \frac{3.3}{12}\right) = 2.73\mu H \approx 2.2\mu H$$

$$L2 = \frac{4}{500Khz \times 1.75} \times \left(1 - \frac{4}{12}\right) = 3.05\mu H \approx 3.3\mu H$$

For 3.3V output, the inductor is L1= 3.3µH since the calculated value is 2.73µH and the next highest standard value is 3.3µH. The output of 4V will require 3.3µH since the calculated value is 3.05µH.

Based on the value of the inductor, the ripple at nominal input voltage 12V will be:

$$\Delta I_{L(nom)} = \frac{V_{out}}{f \times L} \times \left(1 - \frac{V_{out}}{V_{in(nom)}}\right) = \frac{3.3}{500kHz \times 3.3\mu} \times \left(1 - \frac{3.3}{12}\right) = 1.45\ A$$

So, the ripple is 1.45A which is 29% ripple and channel 2 will have the same ripple since it has the same inductor. This is the maximum peak current of the inductor adding the DC value plus 50% of 1.45A of the ripple found. The typical DCR values of the inductors are chosen based on the suggestion of the datasheet; the suggestion is based on the less power loss; the range are given to "30mΩ DCR_{MAX} at 20°C and at 100°C, the estimated maximum DCR values are 26.4mΩ and 39.6mΩ" [7].

4- MOSFET Selection

The selection of the N- Channel MOSFETs is optional but has some hints from datasheets based on the $R_{DS(on)}$. The datasheet suggests a MOSFET with low $R_{DS(on)}$. The one that was available is SI4936BDY-T1-E3 which has $R_{DS(on)}$ = 40mΩ, but the datasheet suggested 18.7mΩ which is out of stock. Their selection was based on power dissipation. The power loss is predicted from the DC resistance; since the two MOSFETs have 40mΩ each and R_{sense} is 8mΩ, the total is 48mΩ; then, the power loss for 3.3V and 1A current load will be I^2R = 1A×48mΩ = 48mW. This power loss is much less than the datasheet is calculated because the calculate for more current load.

5- Input capacitor and output capacitors selection

C_{in} was chosen for RMS current rating of at least 2A based on the maximum designed current and the datasheet suggested that. CAP ALUM POLY 22UF 20% 35V SMD is the capacitor for the input and has 2.3A RMS current rating and 50mΩ ESR (Equivalent Series Resistance).

C_{out} is chosen based on ESR; the ESR is 0.05Ω or less for low output ripple. The maximum ripple voltage will be high at maximum input voltage. The output voltage ripple due to ESR can be calculated by the formula below provided in datasheet.

$V_{OutRipple} = R_{ESR} \times \Delta I_L = 0.05 \times 1.5 = 75mV$

6- Diodes selection

The Schottky diodes selection is optional; the diodes are connected from pins INTVCC which are pins on the chip that function as internal 5V regulator output that control the circuits that are powered from this regulator. These pins INTVCC are decoupled with 4.7µF capacitor with low ESR TO PGND pin. One of the importance of these diodes are to conduct between the two MOSFETs during the dead time off and on; means to prevent the bottom diodes from turning on and storing charge from the capacitor during the time that requires a reverse recovery to increase the efficiency at high Vin and when the SW pin is low. Based on the datasheet "A 1A to 3A Schottky is generally a good compromise for both regions of operation due to the relatively small average current. Larger diodes result in additional transition losses due to their larger junction capacitance" [7]. The diode was selected to be CMDSH-3TR Diode Schottky 30V 100mA Surface Mount SOD-323. The DC reverse breakdown is greater than Vin maximum.

LTSpice Simulation for Power Supply LTC3850 Buck Converter

The simulation was done using LTSpice; as seen in Figure 92 below, the schematic of the circuit was designed based on the typical application from datasheet; the component was chosen after the required calculation according to datasheet. The output 1 and 2 is 3.3V with two capacitors of 47µF and 220µF with ESR 2mΩ and 1mΩ and 4A RMS current rating since is selected based on ESR and RMS current rating of at least 2A; the V_{FB} pins are called VOSENSE in LTSpice are connected to resistive divider to obtain the output voltages of 3.3V and 4V as mentioned in design calculation section above. The input capacitor is 22µF with ESR of 2mΩ. The input is connected to the input pin of LTCC3850 chip with 0.1µF and 10Ω resistor. Also, the input is connected to the drain of the MOSFETs; the SW1 and SW2 pins are switching pins for the MOSFETs that are connected to the source of the top MOSFETs and drain of the bottom MOSFETs to the inductors. There are two Schottky diodes connected from INTVCC pin to

Boost pins and MOSFETS through decoupling capacitor to store and discharge during the time on and off that requires diodes to conduct the MOSFETs. The selection of these diodes is explained in the design calculation and selection section above.

The $sense^+$ and $sense^-$ pins are the pins for current sense comparator are connected to Rsense with a decoupling capacitor from $sense^+$ and $sense^-$ pins. Also, the Track/SS pins are for soft-start inputs and output voltage tracking relate to decoupling capacitor for just to smooth the simulation. BG1 and BG2 pins are bottom gate driver outputs to operate the gates of the bottom N-Channel MOSFETs and swings to see the maximum and minimum difference of the voltages between PGND and INTVCC. The TG1 and TG2 are directly connected to the top MOSFETs gates for voltage output swing equal to INTVCC pin. The I_{TH1} and I_{TH2} are current control threshold and error amplifier compensation to provide the feedback mechanism and compensation. In these pins I_{TH1} and $I_{TH2,}$ a voltage divider connected to the output provides a sample of the output voltage, which is compared to a reference voltage by the error amplifier. These pins are configured by connecting resistor and capacitor in series with the output of the error amplifier and a decoupling capacitor 1000pF for I_{TH1} and 100pF for I_{TH2} before the resistor R7 and R8 10kΩ and 13kΩ respectively. FREQ/PLLFLTR (Pin 28/Pin 25/Pin 26) is the Phase-Locked Loop's Low-Pass Filter is connected to this pin to vary the frequency of the internal oscillator. This pin is connected to external resistor of 3.16KΩ and 10KΩ to PGOOD pin through 100KΩ; this PGOOD pin is to indicate all the voltages are within correct specification and that the chip may proceed to boot and operate when the output voltages are stable or not.

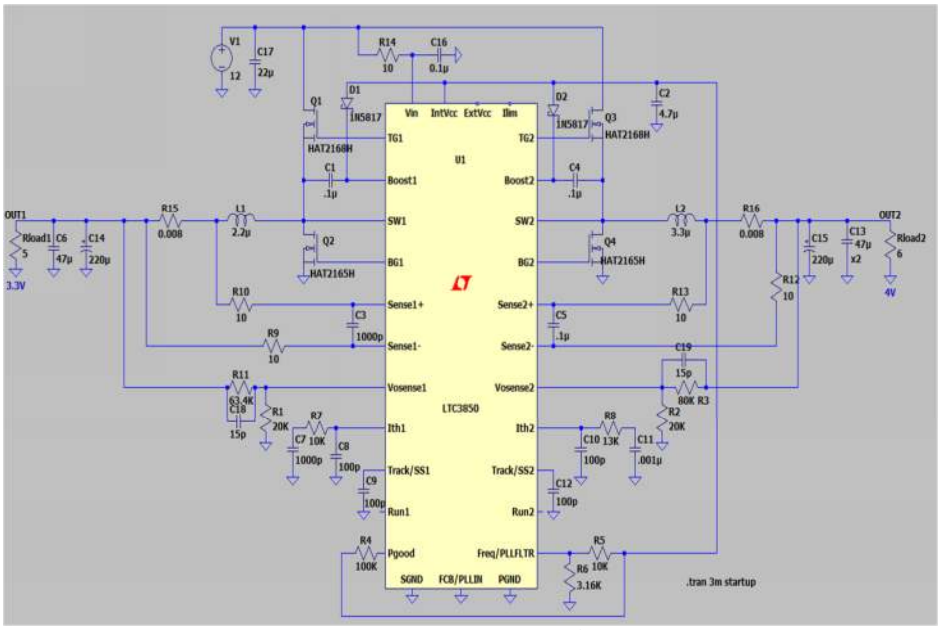

Figure 92 - LTSpice LTC3850 Circuit Schematic

The above schematic of the LTC3850 is simulated and the output voltages of 3.3V and 4V are obtained; see the plot below in Figure 93 that shows the output voltages and currents required to feed the system.

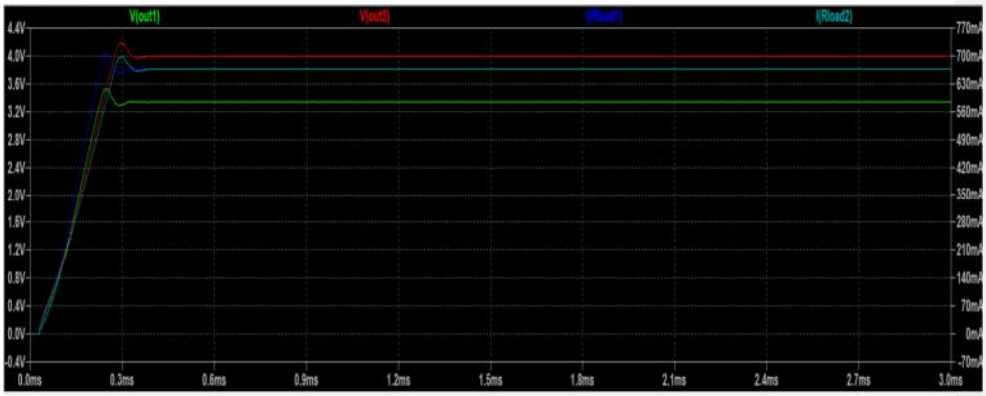

Figure 93 - LTSpice Simulation Plot for the Power Supply

LTC3850 Design Analysis

Design analysis for an LTC3850 dual 2-phase synchronous step-down switching controller:

1. Input and output voltage range:

The LTC3850 supports a wide input voltage range from 4V to 38V, making it suitable for various applications with varying input voltage requirements. The IC provides two output channels, each capable of delivering up to 5A, but 2A in this design, allowing for flexible power supply design.

2. Switching frequency:

The LTC3850 allows for adjustable switching frequencies ranging from 250 kHz to 750 kHz. This flexibility enables designers to balance efficiency, component size, and EMI considerations based on the specific requirements of the application.

3. Current mode control:

The LTC3850 utilizes current mode control, providing fast transient response, inherent cycle-by-cycle current limiting, and simplified loop compensation. This results in stable and responsive operation across a wide range of input voltages, output currents, and output capacitances.

4. Synchronous operation:

The synchronous operation of the LTC3850 enables high efficiency by eliminating the need for external Schottky diodes. The IC integrates two sets of N-channel MOSFET gate drivers, further simplifying the design and reducing external component count.

5. Phase interleaving:

The LTC3850 employs 2-phase operation, which interleaves the switching of the two output channels, reducing input ripple current and enabling the use of smaller input capacitors. This leads to a more compact design and improved transient response.

6. Power good indication:

The LTC3850 includes power good (PG) pins that provide an open-drain output indicating when the output voltages are within a specified range. This feature allows for easy monitoring of the power supply's status and can be used for sequencing other power supplies or enabling downstream loads.

7. Short-circuit and overcurrent protection:

The LTC3850 incorporates cycle-by-cycle current limiting, which provides protection against output short circuits and overcurrent conditions. The controller also includes a hiccup mode to reduce power dissipation during prolonged overcurrent events.

8. Thermal performance:

The LTC3850's thermal performance is critical to ensure reliable operation. Proper thermal management measures, such as using components with suitable temperature ratings, ensuring adequate heat dissipation, and incorporating thermal vias in the PCB design, should be considered in the design.

9. Stability and compensation:

Proper compensation of the control loop is essential for stable operation. The LTC3850 datasheet provides guidelines for selecting compensation components, which should be followed to achieve desired phase margin and gain margin for the controller.

10. PCB layout and EMI considerations:

Following best practices for PCB layout, such as proper grounding, minimizing trace lengths, and using a ground plane, is essential for optimal performance and EMI reduction. Additionally, carefully placing input capacitors, output capacitors, and power inductors can further reduce EMI and improve overall performance.

In conclusion, the LTC3850 dual 2-phase synchronous step-down switching controller offers a flexible and efficient solution for a wide range of applications. By considering factors such as input/output voltage range, switching frequency, current mode control, synchronous operation, phase interleaving,

power good indication, short-circuit and overcurrent protection, thermal performance, stability and compensation, and PCB layout and EMI considerations, a designer can create a robust and efficient power supply.

Battery Charger Controller Design

The selection of a battery management system needed consideration to ensure the battery is protected; so, the choice is Linear Technology LT3652 due to four important factors. First, temperature sensing monitors the battery temperature to ensure that the temperature doesn't go out of the range and stops charging the battery until it gets back to the proper temperature range; this will protect the battery. Second, over-voltage protection that protects the battery from over charging and over discharging. Third, power path is considered important to our system; the power path controls the power to select the source to feed the system between solar panel and battery. Finally, maximum peak power tracking (MPPT) which is an algorithm that is included in charge controller used to obtain or to extract maximum available power from PV module under certain conditions. These factors are included in the selection of the chip LT3652 which is a power tracking 2A battery charger for solar power 1A solar panel powered 1-stage 12V lead-acid fast/float charger with input up to 16V and output up to 14.4V at 1A.

LT3652 Features

1- Input Range: 4.95V to 32V Including MPPT.
2- Programable charge rate up to 2A.
3- Resistor programmable float voltage up to 14.4V.
4- 1MHz fixed frequency.
5- 0.5% float voltage reference accuracy.
6- 5% charge current accuracy.
7- Operating junction temperature range -40˚C to 125˚C.

8- Absolute maximum voltage rating 40V.

LT3652 Design Procedure

Designing an LT3652 Monolithic Multichemistry Battery Charger for Solar Power involves multiple steps, including choosing appropriate components, calculating component values, and creating a printed circuit board (PCB) layout. Here is a step-by-step design procedure for creating a charger based on the LT3652 IC.

1. Study the datasheet:

Familiarize with the LT3652 datasheet, which provides essential information about the IC, its features, and its operation. Understanding the specifications and requirements will help make informed design choices.

2. Select the battery type and capacity:

Determine the chemistry and capacity of the battery needed to charge, as this will influence the design choices. The LT3652 is compatible with various battery chemistries, such as Li-ion, Li-polymer, LiFePO4, and lead-acid batteries.

3. Determine the input voltage range:

Select an appropriate solar panel or other power source that can provide the required input voltage for the LT3652. The input voltage range should be compatible with the IC's specifications (4.95V to 32V).

4. Set the charging current and voltage:

Based on the battery chemistry and capacity, determine the charging current and voltage. The charging current can be set using an external resistor connected to the TIMER/PROG pin of the LT3652. Use the formula provided in the datasheet to calculate the resistor value.

5. Calculate component values:

Calculate the values for other external components, such as the inductor, input and output capacitors, and feedback resistors. These component values will affect the charger's performance and efficiency.

6. Choose appropriate components:

Select high-quality components that can handle the voltage and current requirements of the design. Make sure to choose components with appropriate temperature ratings, as solar applications can experience wide temperature variations.

7. Create a schematic:

Using a schematic capture tool, Altium Designer is used in this design to create a schematic that includes the LT3652 IC and all required external components. Make sure to follow the application circuits provided in the datasheet as a reference.

8. Design the PCB layout:

Using a PCB design Altium Designer, create a PCB layout based on the schematic. Follow best practices for PCB design, such as ensuring proper grounding, minimizing trace lengths, and using a ground plane. Additionally, follow any specific layout recommendations provided in the datasheet.

9. Verify and optimize the design:

Simulate the design using a circuit simulator or SPICE tool to ensure it meets the requirements. Make any necessary adjustments to component values or the PCB layout to optimize performance and efficiency.

10. Prototype and test:

Once the design is finalized, order a prototype PCB and assemble the charger. Test the performance of the charger in various operating conditions to ensure it meets your requirements.

11. Iterate and finalize:

If necessary, make any adjustments to the design based on the testing results. Iterate the design process until it's functional and reliable solar-powered battery charger based on the LT3652 IC.

LT3652 Design Calculation and LTSPice Simulation

The calculation will be done to charge an acid battery of 6V or 12V with float voltage of 14.4V and the solar panel of 0.3A.

1- Charge current programming.

$R_1 = \frac{V_{sense}}{I_{charge(max)}} = \frac{100mV}{1A} = 100mV/1A = 100m\Omega$

Where R_1 is R_{sense}, and V_{sense} is the voltage drop will be created by sense resistor.

This sense resistor is connected to the inductor, which will be discussed next, and to the output decoupling capacitors.

2- Inductor Selection

To select the inductor, the two considerations from the datasheet will be applied. First, consideration is the ripple current created in the inductor. Second, its saturation current should be equal to or exceeds the maximum peak current in the inductor. Using the following equation to calculate the inductor value.

$$L = \frac{10 \times R_{sense}}{\frac{\Delta I_L}{I_{chg(\max)}}} \times V_{BAT(FLT)} \times \left[1 - \frac{V_{BAT(FLT)}}{V_{IN(MAX)}}\right]$$

$$L = \frac{10 \times 100m\Omega}{\frac{0.3A}{1A}} \times 14.4V \times \left[1 - \frac{14.4V}{21V}\right] = 15\mu H$$

From the above calculation, the ripple current is set to 25% to 35% of the $I_{CHG(MAX)}$; by setting $0.25 < \frac{\Delta I_L}{I_{CHG(MAX)}} < 0.35$.

The inductor is connected to the SW pin which is the output of the charge switch. The SW pin is connected to BOOST pin via a capacitor of 1µF. the BOOST pin works as bootstrapped supply for operating range of 0V up to 8.5V; the capacitor voltage connected to SW pin is refreshed by rectifying diode with the cathode connected to BOOST pin and anode is connected to either the

battery output voltage or the load source. However, the diode selected to refresh the decoupling capacitor from the battery with battery float voltages higher than 8.4V, a >100mA, is Zener diode can be put in series with the rectifying diode to prevent exceeding the BOOST pin operating voltage range.[6]

3- Battery Float Voltage Programming

The output battery float voltage $V_{BAT(FLT)}$ is programmed by external resistive divider from the battery pin to V_{FB} pin. The equivalent input resistance at V_{FB} pin is 174KΩ to compensate for bias current error and for $V_{BAT(FLT)}$ = 13.5V; so, to calculate the other resistor using the following equation.

$\frac{R_8}{R_5} = \frac{3.3V}{(V_{BAT(FLT)} - 3.3V)} = \frac{R_8}{R_5} = \frac{3.3V}{(8.2 - 3.3V)} = 0.67$

With R_8 = 100kΩ, $R_5 = \frac{R_8}{0.67} = \frac{100K\Omega}{0.67}$ = 148,484Ω ≈150kΩ

The divider equivalent resistance is:

$R_5 \,||\, R_8$ = 100kΩ || 150kΩ = 60kΩ

R_7 = 250kΩ - 75kΩ = 190kΩ

Where R_8 is R_{FB1} and R_5 is R_{FB2}, and R_7 is R_{FB3}.

For this design, it is a 2-cell battery with float voltage 8.2V and current of 0.25A. the calculation is as follows:

R1 = (VBAT(FLT) • 2.5 • 10^5)/3.3 (Ω) = $(8.2 \times 2.5 \times 10^5)/3.3$ = 621,212 (619kΩ was found)

R2 = (R1 • 2.5 • 10^5)/(R1 - (2.5 • 10^5)) = 418,367 (417kΩ was found)

4- Battery specific design

The charge controller based on the LT3652 for charging two Samsung INR18650-25R cells in series at 0.25A (based on the solar panel that generates 0.3A), needs to consider the specific

characteristics of the battery cells and the charger IC. Here's a summary of the design parameters and their impact on the PCB:

a. Battery specifications:

The Samsung INR18650-25R cells have a nominal voltage of 3.6V and a capacity of 2500mAh. When connecting two cells in series, the total voltage is 7.2V, and the capacity remains the same (2500mAh).

b. Charging current:

To charge the battery pack at 0.25A. To set the charging current, needs to choose an appropriate resistor value (R_PROG) for the TIMER/PROG pin of the LT3652. Use the formula from the datasheet:

R_PROG = (0.1V / I_CHARGE)

Where I_CHARGE is the desired charging current. For a 0.25A charging current, R_PROG will be approximately 0.33Ω.

c. Charging voltage:

The LT3652 needs to be configured to charge the battery pack at the appropriate voltage. For two INR18650-25R cells in series, the charging voltage should be around 8.4V (4.2V per cell). To set the charging voltage, you'll need to choose appropriate resistor values for the voltage feedback resistors (R1 and R2) connected to the VFB pin. Use the formula from the datasheet:

V_FLOAT = 3.3V * (1 + R1/R2)

Choose R1 and R2 values that satisfy this equation for V_FLOAT = 8.4V as in part 3.

d. Solar panel input:

The solar panel provides an adequate input voltage range for the LT3652 and sufficient power to charge the battery pack at the desired charging current (0.25A). The input voltage should be compatible with the IC's specifications (4.95V to 32V).

e. Component selection and PCB layout:

Select appropriate values for the inductor, input and output capacitors, and other components according to the datasheet recommendations. Create a schematic and PCB layout following best practices and any specific layout guidelines provided in the datasheet.

As a result of designing an LT3652 charge controller for charging two Samsung INR18650-25R cells in series at 0.25A, has a compact and efficient solar-powered battery charger suitable for your application. The PCB should be optimized for performance, efficiency, and thermal management, ensuring reliable operation under various conditions.

5- LTSpice Simulation for LT3652 Battery Charge Controller

In the LTSpice schematic in Figure 94 below shows the circuit simulated without MPPT system; the photocell diode is created to simulate the input voltage as a solar panel output.

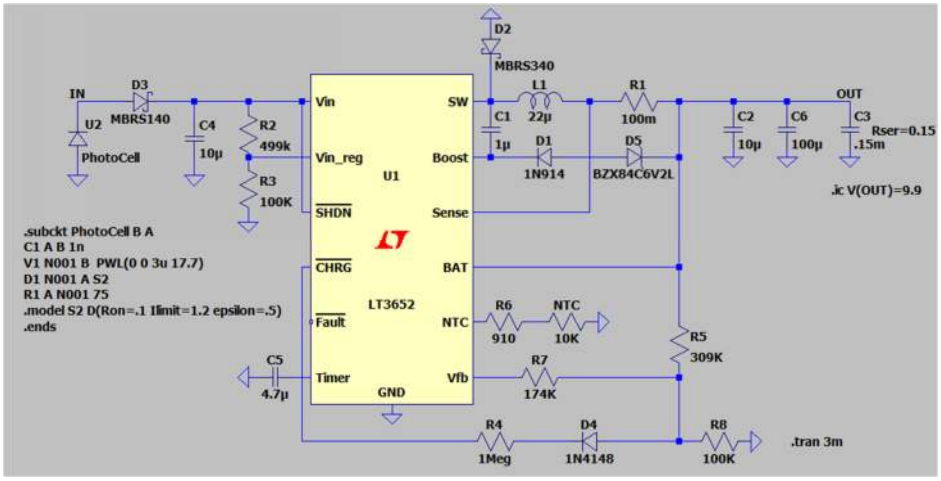

Figure 94 - LTSpice Battery Charger Controller Schematic

The simulation plot in Figure 95 below shows the charging current of 1A and float voltage of 14.4V. This simulation might be changed based on the solar panel and battery type if the current panel and battery need to be replaced with higher values.

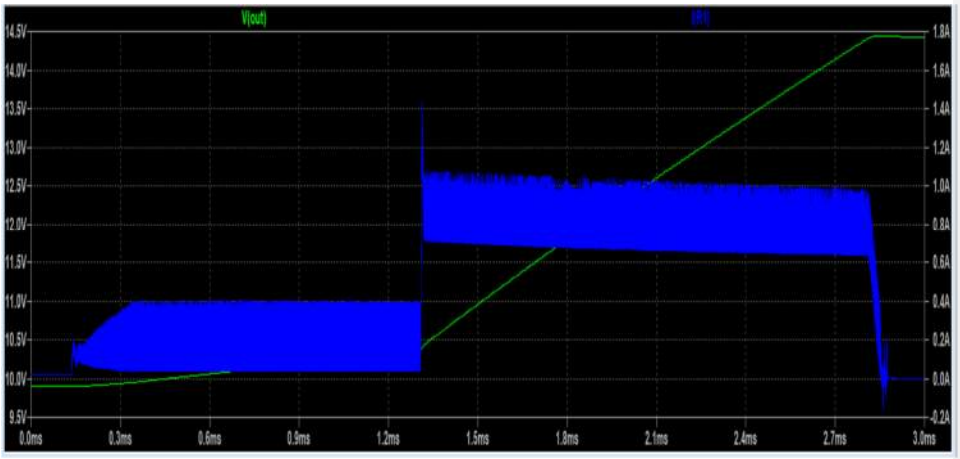

Figure 95 - LTSpice Charging Simulation Plot for Battery Charge Controller

LT3652 Design Analysis and Battery Protection

Design analysis for an LT3652 Monolithic Multichemistry Battery Charger for Solar Power with Battery Protection:

1) Charging algorithm:

The LT3652 features a three-stage charging algorithm: constant-current (CC), constant-voltage (CV), and float charge. This algorithm ensures efficient charging while maintaining battery health and longevity.

2) Input voltage regulation:

The LT3652 incorporates input voltage regulation to prevent input overvoltage conditions and protect the solar panel from damage. The voltage regulation loop maintains the input voltage at a user-defined value, preventing overloading of the solar panel.

3) Battery protection:

Battery protection is crucial for ensuring the safety and longevity of the battery. The following battery protection features should be included in the design:

a. Overvoltage protection:

Select a battery protection IC or circuit that monitors the battery voltage and disconnects the charging source if the voltage exceeds a specified limit. This prevents overcharging and potential damage to the battery.

b. Overcurrent protection:
Implement a current-sensing resistor or a dedicated overcurrent protection IC to monitor charging current. If the current exceeds the maximum allowed value, the protection circuit should disconnect the charging source, preventing damage due to excessive current.

c. Overtemperature protection:
Incorporate a temperature sensor, such as an NTC thermistor, close to the battery to monitor its temperature. If the temperature exceeds a safe limit, the protection circuit should disconnect the charging source to prevent thermal damage to the battery.

d. Reverse polarity protection:
Include a diode or a dedicated reverse polarity protection circuit in the design to prevent damage to the charger and battery if the battery is connected with reversed polarity.

4) Efficiency:

The LT3652 offers high efficiency in battery charging, typically over 90%. To maximize efficiency, choose high-quality, low ESR capacitors and low DCR inductors. Additionally, optimize the PCB layout to minimize parasitic resistances and inductances.

5) Thermal management:

Solar-powered applications may experience wide temperature variations. It's essential to design the charger with appropriate thermal management measures, such as using components with suitable temperature ratings, ensuring proper heat dissipation, and incorporating thermal vias and polygon pours in the PCB design.

6) EMI/EMC considerations:

To minimize electromagnetic interference (EMI) and ensure electromagnetic compatibility (EMC), follow best practices for PCB layout, such as proper grounding, minimizing trace lengths, using a ground plane, and placing bypass capacitors close to the IC. Additionally, consider using a shielded inductor to reduce radiated emissions.

In conclusion, designing an LT3652 2A Monolithic Multichemistry Battery Charger for Solar Power requires careful consideration of various factors, such as charging algorithm, input voltage regulation, battery protection, efficiency, thermal management, and EMI/EMC considerations. By addressing these aspects in the design, you can create a robust and efficient solar-powered battery charger.

Hardware Design

To test and verify that each component would work properly, separate breakout PCBs were designed for each of the major components in the block diagram. This was done for two main reasons. First, the LTE module PCB contained an impedance matched trace which was important to get right since this is how it communicates with the cell towers. Second, separating each of the components into separate PCBs would allow for easier debugging of the entire system so that the final PCB would work much better and more flawlessly. Each of the PCBs designed were four-layer with two signal layers, a power layer, and a ground layer. This was done for easier routing, impedance matching, and since the final PCB will also be a four-layer one.

MCU and SmartMesh IP Breakout Board

The MCU and SmartMesh IP breakout board contained the two central components for the entire system. Although testing was done to implement the network manager on the chip level, this did not work and thus a PCB module was ordered from Analog Devices which contained a prebuilt network manager chip with all the required components. The schematic for this PCB is shown in Figure 96.

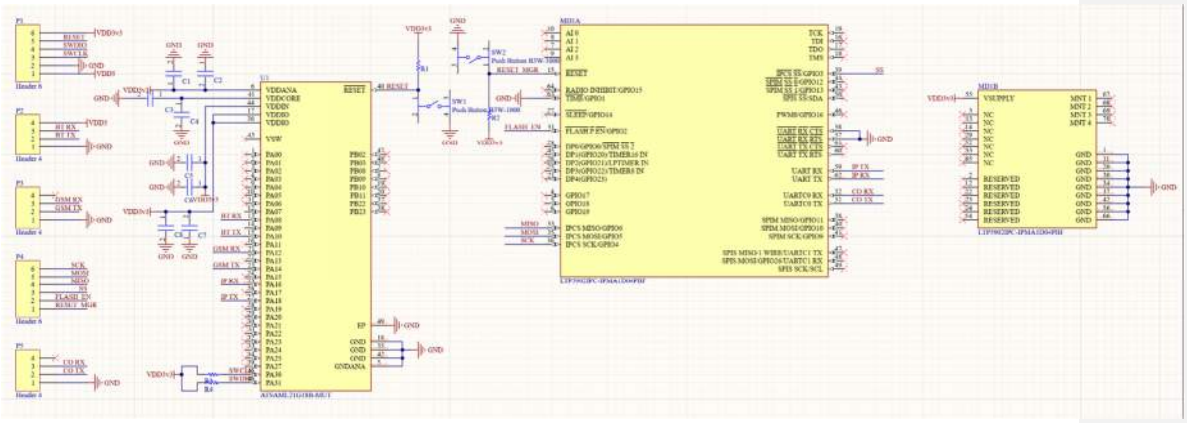

Figure 96 - MCU and Network Manager Schematic

The network manager is divided into the two chips on the left side, the MCU is shown in the middle, and the ports outputting the signals for the different UART port are shown on the left. Two reset buttons were added: one for each of the main components. Bulk and decoupling capacitors were added to the SAML21 based on the datasheet. An external power supply would be used with this module and would take an input of 3.3V. A separate input supply would also be added for the Bluetooth module so it could be easily connected to the MCU.

LTE Breakout Board

The LTE breakout board was also especially important to design and build. Since this PCB would have an impedance matched trace, the PCB layer stackup would have to be known exactly since the separation between layers and the dielectric constant of the material between them makes a significant difference on the trace impedance. JLCPCB was chosen for the manufacturer of all the PCBs and there is

a section on their website explaining the different layer stackups that could be achieved. The one selected for this PCB is shown in Figure 97.

layer	Material Type	Thickness	
Layer	Copper	0.035mm	
Prepreg	7628*1	0.2104mm	
inner Layer	Copper	0.0152mm	1.1mm (with copper c
Core	Core	1.065mm	
inner Layer	Copper	0.0152mm	
Prepreg	7628*1	0.2104mm	
Layer	Copper	0.035mm	

Figure 97 - Layer Stackup JLCPCB

This information could then be entered into Altium Designer's Layer Stack Manager. The proper dielectric constants and layer thicknesses were also selected. Once this was complete, an impedance calculator provided by Altium Designed was used to find the trace width necessary for an impedance of 50 ohms. The impedance calculator and layer stack manager are shown in Figure 98. On the left, a more in-depth layer stackup is shown which also includes the silkscreen layer as well as all the other layers. On the right, the impedance calculator is shown which uses the information on the left to calculate the trace width necessary for a certain impedance. Since the LTE module output was set at 50 ohms, this is

the impedance that was set for all four layers. The trace itself would be run on the top layer so the 13.967mil trace will be used within the PCB.

#	Name	Material	Type	Weight	Thickness	Dk
	Top Overlay		Overlay			
	Top Solder	Solder Resist	Solder Mask		0.5mil	3.8
1	Top Layer		Signal	1oz	1.378mil	
	Dielectric 2	PP-006	Prepreg		8.284mil	4.6
2	Ground	CF-004	Signal	1/2oz	0.598mil	
	Core	FR-4	Core		41.929mil	4.6
3	Power	CF-004	Signal	1/2oz	0.598mil	
	Dielectric 3	PP-006	Prepreg		8.284mil	4.6
4	Bottom Layer		Signal	1oz	1.378mil	
	Bottom Solder	Solder Resist	Solder Mask		0.5mil	3.8
	Bottom Overlay		Overlay			

Top Ref	Bottom Ref	Width (W1)	Impe...	Devia...	Delay...
	2 - Ground	13.967mil	50.01	0.03%	157.19...
1 - Top Layer	3 - Power	10.329mil	49.99	0.03%	182.88...
2 - Ground	4 - Bottom L...	10.329mil	49.99	0.03%	182.88...
3 - Power		13.967mil	50.01	0.03%	157.19...

Figure 98 - Altium Designer Layer Stack Manager

The schematic designed for this breakout board is shown in **Error! Reference source not found.**. The antenna connector selected also had an input impedance of 50 ohms so that no signal would be reflected to the source. The two LEDs were used as signals indicating that the LTE module is fully functional. A voltage-level shifter also had to be added to the PCB since the LTE module works at 1.8V. The UART pins cannot accept any voltage higher than 2.1V and therefore a voltage level shifter between 1,8 and 3.3V was used to protect the module. A SIM card slot was added and is shown on the bottom right of the figure. Each of the pins was connected to the proper one on the LTE module and a micro sim card had to be used. A header outputting a USB connection was also added to allow for firmware updates to the LTE module. Finally, bulk and decoupling capacitors were added to improve the performance of the LTE module. The bulk capacitors were especially important to add since whenever the LTE module sends data over to the cloud, it can begin using up to 300mA of current within 1 microsecond. To maintain voltage stability, three 100uF capacitors were added to the back of the PCB.

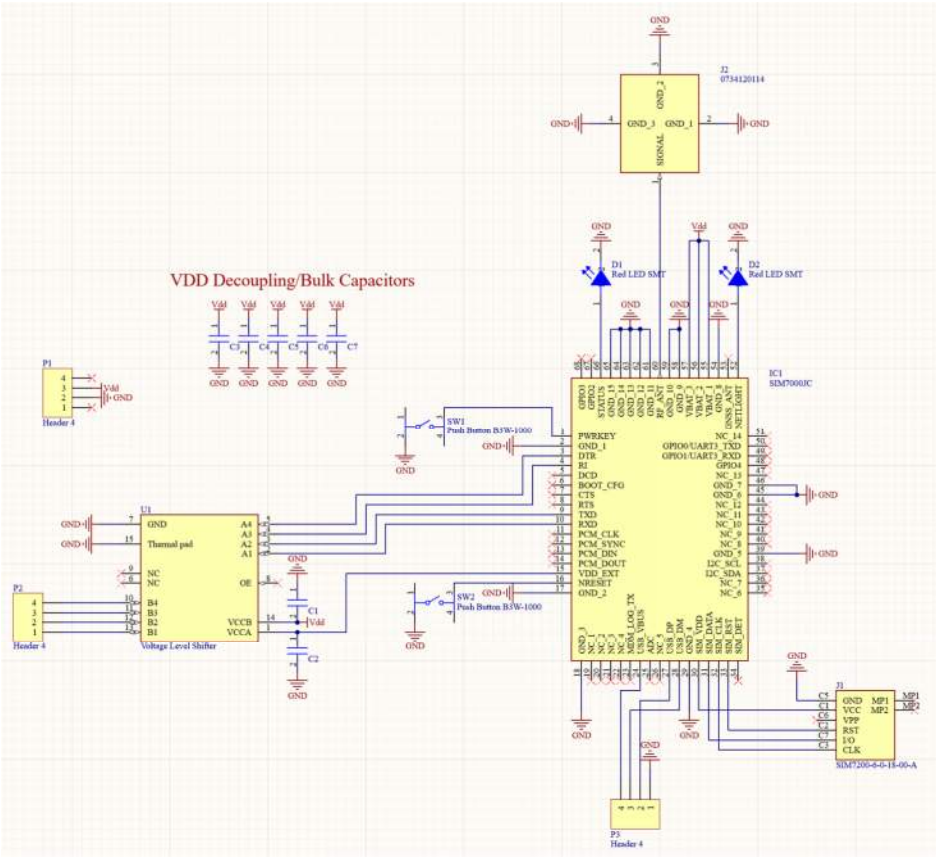

Figure 99 – LTE Module Breakout Schematic

Main final Breakout Board

The schematic of the main board is a combination of other schematics designed previously, including power supply, LTE Module, and MCU and Network Manager Schematic. The Altium Designer's Layer Stack Manager is shown in Figure below. Appropriate dielectric constants and layer thicknesses were chosen, and upon completion, Altium Designer's impedance calculator was employed to determine the required trace width for a 50-ohm impedance. Figure 100 displays the impedance calculator and layer stack manager. The left side presents a detailed layer stackup, which includes the silkscreen layer and all other layers. The right side features the impedance calculator, which utilizes the

information on the left to compute the necessary trace width for a specific impedance. As the LTE module output was set at 50 ohms, this impedance value was applied to all four layers. The trace will be placed on the top layer, and a 13.967mil trace will be incorporated within the PCB.

#	Name	Material	Type	Weight	Thickness	Dk	Copper Orientat	Top Ref	Bottom Ref	Width (W1)	Impe...	Devia...	Delay...
	Top Overlay		Overlay										
	Top Solder	Solder Resist	Solder Mask		0.5mil	3.8							
1	Top Layer		Signal	1oz	1.378mil		Above	✓	2 - GND	13.967mil	50.01	0.03%	157.19..
	Dielectric 2	PP-006	Prepreg		8.264mil	4.6							
2	GND	CF-004	Signal	1/2oz	0.598mil		Above	✓ 1 - Top Layer	3 - Power	10.329mil	49.99	0.02%	182.88..
	Core	FR-4	Core		41.929mil	4.6							
3	Power	CF-004	Signal	1/2oz	0.598mil		Below	✓ 2 - GND	4 - Bottom L...	10.329mil	49.99	0.02%	182.88..
	Dielectric 3	PP-006	Prepreg		8.264mil	4.6							
4	Bottom Layer		Signal	1oz	1.378mil		Below	✓ 3 - Power		13.967mil	50.01	0.03%	157.19..
	Bottom Solder	Solder Resist	Solder Mask		0.5mil	3.8							
	Bottom Overlay		Overlay										

Figure 101 - Altium Designer Layer Stack Manager for main board.

The layout of this main board is combined layouts of Power Supply, LTE Module, and MCU . See Figure 102 and Figure 103 below for signal trace that shows the components; the red components are placed on top layer and blue components are placed on the bottom layer. Also, the polygon pour connects the ground of the input to the ground of the output capacitors as separating a signal ground from power ground. The all-layer traces next to the signal trace contain multiple design layers such as ground, power, top, bottom, assembly, mechanical layer, and so on. The yellow overall layer shown is for setting the dimensions for manufacturers to print the PCB. The top overlay shows the component designators and texts typed. The mechanical layer is used to place information about PCB board footprints and assembly such as physical dimension of the components, vias, assembly instructions and so on.

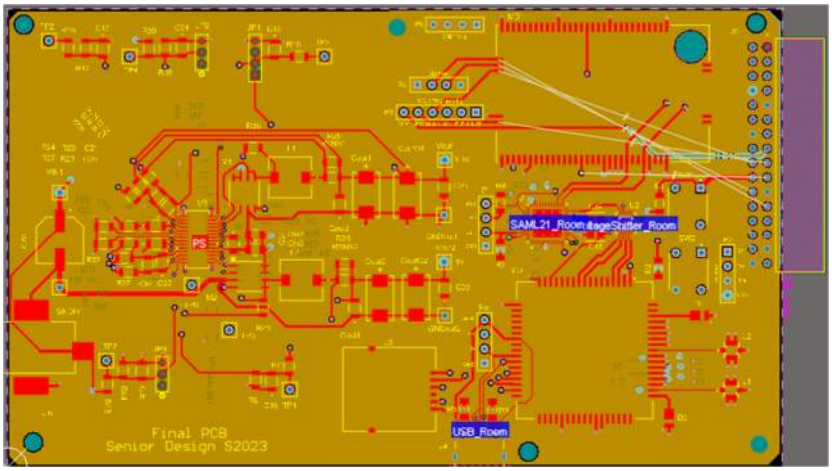

Figure 104 - Power Traces for Main Board

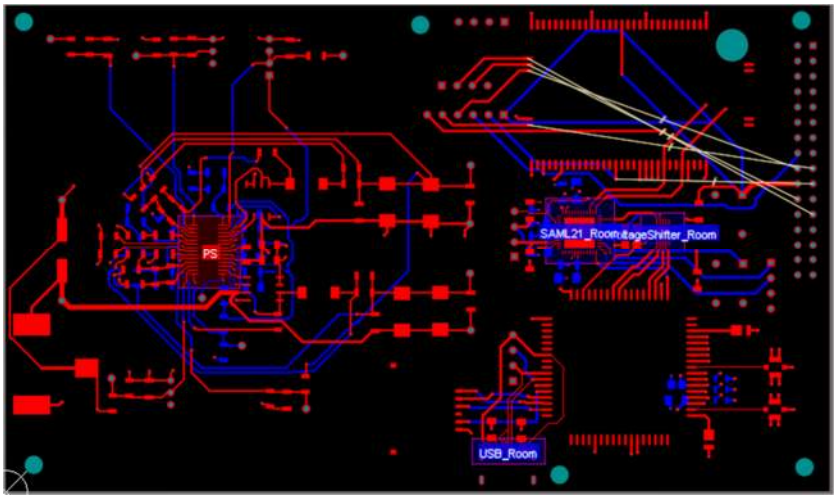

Figure 105 - Signal Traces for Main Board

Upon finishing the layout as described, examine it for potential mistakes or breaches of regulations. Next, create Gerber files and NC drill files for the manufacturer to produce the board. Import these files according to the manufacturer's guidelines to ensure all necessary requirements are met for completing the PCB printing process. Utilize the 3D view feature in Altium Designer to identify any absent footprints,

incorrectly placed components, missing designators, texts, and other issues. Refer to Figure 106 and Figure 107 below, which display both the front and back layers of the PCB board.

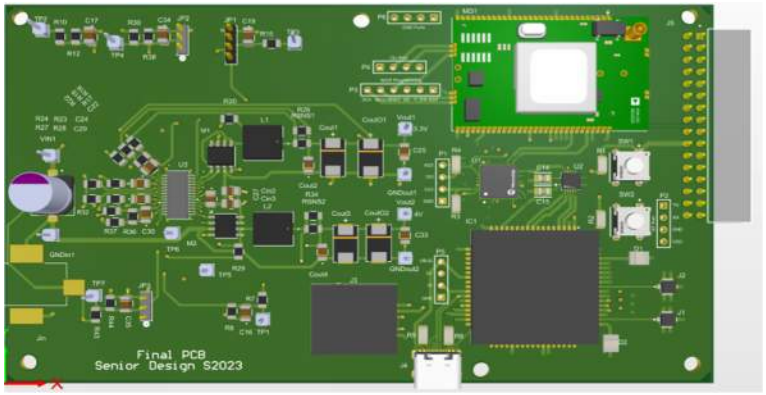

Figure 108 - 3D View of Completed Main Board PCB (Top View)

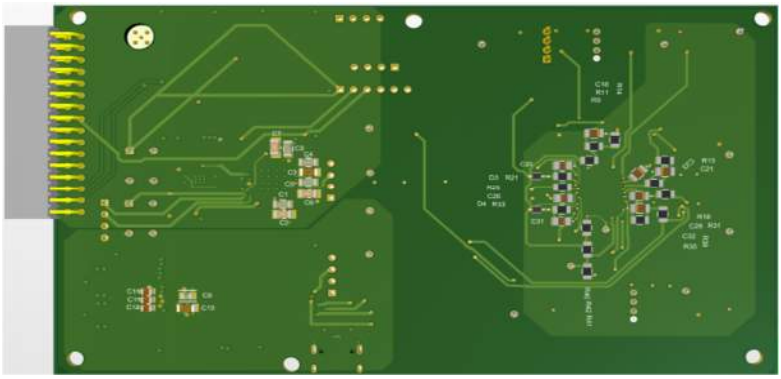

Figure 109 - 3D View of Completed Main Board PCB (Bottom View)

After obtaining the PCB from the manufacturer, solder it using solder paste and hot air, employing tools like tweezers, solder wick, cleaning flux, and digital microscope to attach all necessary components. The main board's PCB is successfully assembled and functioning as intended. Refer to Figure 110 for the finished main board PCB top view, Figure 111 for bottom view.

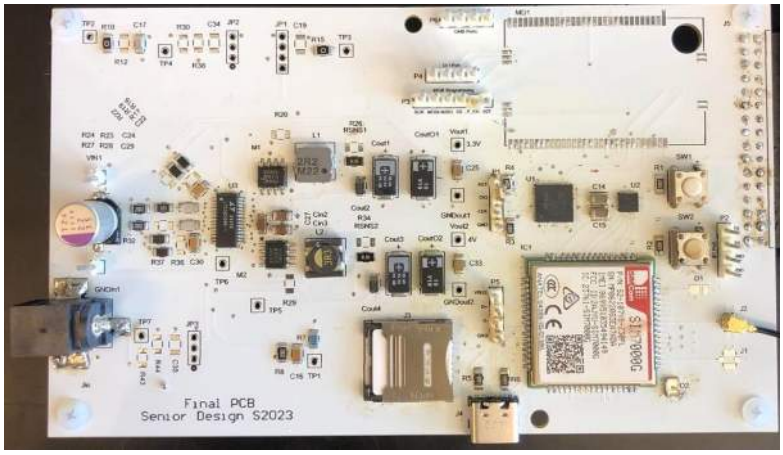

Figure 112 Soldered and Working Main Board PCB (Top View)

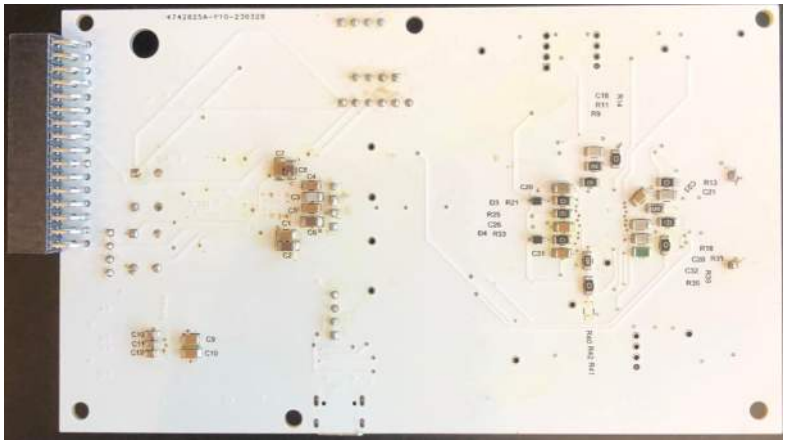

Figure 113 Soldered and Working Main Board PCB (Bottom View)

Power Supply Breakout Board

PCB Layout Implementation for Power Supply LTC3850 Buck Converter is designed using Altium Designer software. The datasheet has PCB Layout Checklist, which was followed to complete the PCB. Starting with schematic of the power supply in Figure 114 below, the design was completed according to the calculation and simulation. The schematic and footprint of the components are selected based on

the design requirements. Besides the requirements, multiple test points were created in case of any issues occur during the operation to find any problem easily.

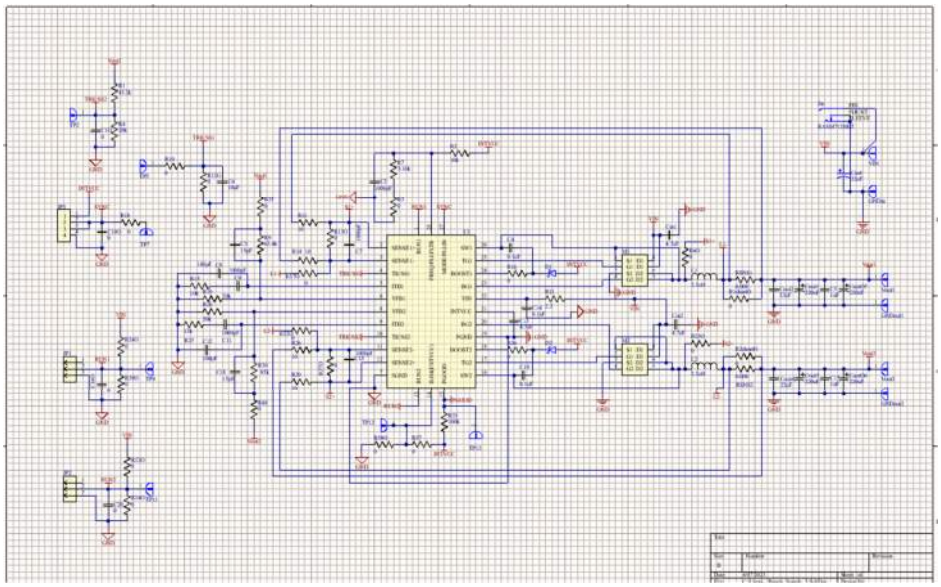

Figure 114 - Power Supply PCB Schematic

Before the PCB layout, the setting of the four layers stack up were assigned with thickness of 1.6mm (≈63mil) as seen the Figure 115 below that shows the stack of the four signal copper layers for top layer, ground, power, and bottom layer. The thickness and other parameters were selected according to JLCPCB manufacturer capabilities. Beside the stack up layers, other manufacturer's restrictions for design rules were followed such as clearance for vias, silk to silk, mask to solder, hole to hole, and components clearance; for routings, routing width, and routing and sizing vias were set in Altium Designer.

#	Name	Material	Type	Weight	Thickness	Dk	Df
	Top Overlay		Overlay				
	Top Solder	Solder Resist	Solder Mask		0.492mil	3.8	
1	Top Layer		Signal	1oz	1.378mil		
	Dielectric 2	PP-006	Prepreg		3.008mil	4.6	0.02
2	GND	CF-004	Signal	1/2oz	0.492mil		
	Dielectric 1	FR-4	Dielectric		49.803mil	4.6	
3	PWR	CF-004	Signal	1/2oz	0.492mil		
	Dielectric 3	PP-006	Prepreg		3.008mil	4.6	0.02
4	Bottom Layer		Signal	1oz	1.378mil		
	Bottom Solder	Solder Resist	Solder Mask		0.492mil	3.8	
	Bottom Overlay		Overlay				

Figure 115 - Layer Stackup for Power Supply

During the layout, some components need to be close to the IC pins according to the datasheet. The size of the capacitors, resistors are 1206 (3.2 mm × 1.6 mm) are quietly large to set them close to their required spot. So, to solve this problem, some of the components were placed in the top layer and some in the bottom layer to get enough space to meet the required distances. See Figure 116 below for signal trace that shows the components; the red components are placed on top layer and blue components are placed on the bottom layer. Also, the polygon pour connects the ground of the input to the ground of the output capacitors as separating a signal ground from power ground. The all-layer traces next to the signal trace contain multiple design layers such as ground, power, top, bottom, assembly, mechanical layer, and so on. The yellow overall layer shown is for setting the dimensions for manufacturers to print the PCB. The top overlay shows the component designators and texts typed. The mechanical layer is used to place information about PCB board footprints and assembly such as physical dimension of the components, vias, assembly instructions and so on.

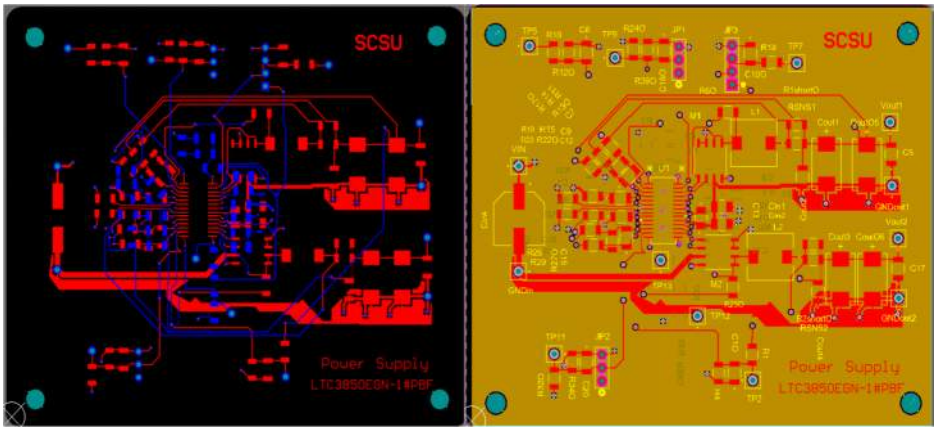

Figure 116 - Signal and Power Traces for Power Supply

After completing the layout above, check for any errors or rules violation. The last step is to generate Gerber files and NC drill files for the manufacturer to print the board. Importing these files using the manufacturer steps to avoid any missing requirement they need to finish printing the PCB board. The 3-d view in Altium Designer is useful to see any missing footprint, misplaced components, missing designators, texts, and so on. See Figure 117 below that shows front layer and back layer of the PCB board.

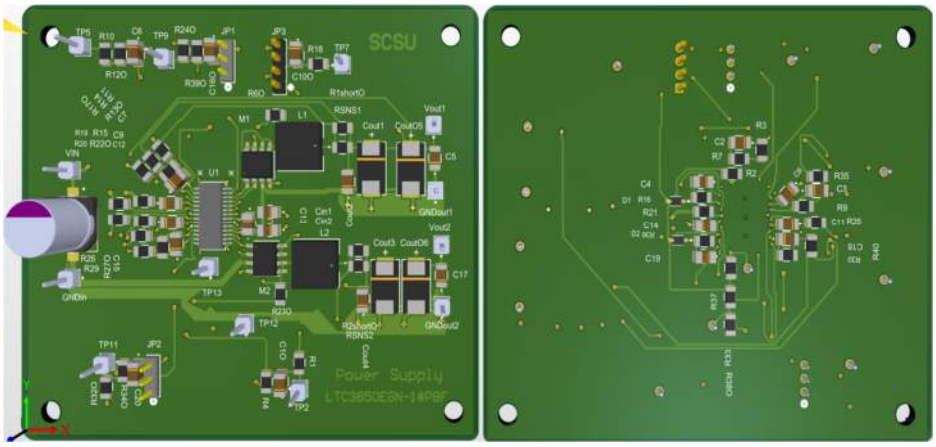

Figure 117 - 3D View of Completed PS PCB

Soldering the PCB after receiving the PCB from the manufacturer; using soldering paste, and hot air to solder all required components with soldering tools such as tweezers, solder wick, cleaning flux, and binoculars. The PCB of the power supply is completed and working as expected. See Figure 118 for completed PCB board for power supply and see Figure 119, the Altium BOM used in this power supply PCB.

Figure 118 Soldered and Working PS PCB

	Comment	Description	Designator	Footprint	LibRef	Quantity
2	Optional	CAP CER --UF 6.3V X5R 1206	C1O, C10O, C16O, C20	CAP_1206X_AVX-L	12066D226MAT2A	4
3	1000pF	CAP CER 1UF 16V X7R 1206	C2, C9, C11, C15	1206_AVX-L	1206YC105KAT2A	4
4	15pF	CAP CER 1UF 16V X7R 1206	C3, C18	1206_AVX-L	1206YC105KAT2A	2
5	0.1uF	CAP CER 4.7UF 25V X7R 1206	C4, C14, C19	CAP_MJ316_TAY-L	TMJ316BB7475MLHT	3
6	1uF	CAP CER 1UF 16V X7R 1206	C5, C17	1206_AVX-L	1206YC105KAT2A	2
7	10nF	CAP CER 22UF 6.3V X5R 1206	C6	CAP_1206X_AVX-L	12066D226MAT2A	1
8	1000pF	CAP CER 22UF 6.3V X5R 1206	C7	CAP_1206X_AVX-L	12066D226MAT2A	1
9	100pF	CAP CER 1UF 16V X7R 1206	C8, C12	1206_AVX-L	1206YC105KAT2A	2
10	4.7uF	CAP CER 4.7UF 25V X7R 1206	C13, Cin1, Cin2	CAP_MJ316_TAY-L	TMJ316BB7475MLHT	3
11	22uF	CAP ALUM POLY 22UF 20% 35V SMD	Cin4	CAP_35SVPD22M	35SVPD22M	1
12	220uF	CAP TANT POLY 220UF 4V 2917	Cout1, Cout3, CoutO5, CoutO6	PCAP_TPE_D2E_PAN	4TPE220MF	4
13	22uF	CAP CER 22UF 6.3V X5R 1206	Cout2, Cout4	CAP_1206X_AVX-L	12066D226MAT2A	2
14	CMDSH-3TR	Diode Schottky 30V 100mA Surface-Mount SOD-323	D1, D2	SOD2512X110N	CMDSH-3TR	2
15	PH1-01-UA	Connector Header Throu-Hole 1 0.100 (2.54mm)	GNDin, GNDout1, GNDout2, TP2, TP5, TP7	ADAM-TECH_PH1-01-UA	PH1-01-UA	6
16	PH1-01-UA	Connector Header Throu-Hole 1 0.100 (2.54mm)	TP9, TP11, TP12, TP13, VIN, Vout1, Vout2	ADAM-TECH_PH1-01-UA	PH1-01-UA	7
17	TMM-103-02-L-S	Connector Header Throu-Hole 3 0.079" (2.00mm)	JP1, JP2	CONN_TMM-103-XX-XX-S_SAI	TMM-103-02-L-S	2
18	TMM-104-01-L-S	Connector Header Throu-Hole 4 0.079" (2.00mm)	JP3	TMM-104-01-G-S_SAI	TMM-104-01-L-S	1
19	2.2uH	FIXED IND 2.2UH 5.4A 12 MOHM SMD	L1, L2	IND_RLF7030T-2R2M5R4_TDK-L	RLF7030T-2R2M5R4	2
20	SI4936BDY-T1-E3	MOSFET 2N-CH 30V 6.9A 8-SOIC	M1, M2	SOIC-8_SI	SI4936BDY-T1-E3	2
21	43.2k	RES 20K OHM 1% 1/4W 1206	R1	RC1206N_YAG-L	RC1206FR-0720KL	1
22	Optional	RES -- OHM 1% 1/4W 1206	R1shortO, R2shortO, R6O, R12O, R13O, R17O	RC1206N_YAG-L	RC1206FR-0720KL	6
23	Optional	RES -- OHM 1% 1/4W 1206	R22O, R23O, R24O, R27O, R32O, R34O, R38O, R39O	RC1206N_YAG-L	RC1206FR-0720KL	8
24	20k	RES 20K OHM 1% 1/4W 1206	R4, R19, R20, R40, R35, R36, R37	RC1206N_YAG-L	RC1206FR-0720KL	7
25	3.16k	RES 3.16K OHM 1% 1/4W 1206	R7, 'R2, R15, R3, R10, R16, R18	RC1206N_YAG-L	RC1206FR-0720KL	7
26	63.4k	RES 63.4 kOHM 1% 1/4W 1206	R9	RC1206N_YAG-L	RC1206FR-0720KL	1
27	10	RES 10 OHM 1% 1/4W 1206	R11, R14, R26, R29	RC1206N_YAG-L	RC1206FR-0720KL	4
28	2.2	RES 2.2 OHM 1% 1/4W 1206	R21	RC1206N_YAG-L	RC1206FR-0720KL	1
29	13k	RES 13K OHM 1% 1/4W 1206	R25	RC1206N_YAG-L	RC1206FR-0720KL	1
30	85k	RES 85K OHM 1% 1/4W 1206	R30	RC1206N_YAG-L	RC1206FR-0720KL	1
31	100k	RES 100K OHM 1% 1/4W 1206	R33	RC1206N_YAG-L	RC1206FR-0720KL	1
32	0.008	RES 0.008 OHM 1% 1/4W 1206	RSNS1, RSNS2	RC1206N_YAG-L	RC1206FR-0720KL	2
33	LTC3850EGN-1#PBF	IC REG CTRLR BUCK 28SSOP	U1	SSOP-28_GN_LIT-L	LTC3850EGN-1#PBF	1

Figure 119 Altium Designer Power Supply BOM List

Charge Controller Breakout Board

PCB Layout Implementation for Charge Controller LT3652 is designed using Altium Designer software. The datasheet has PCB Layout recommendation and typical circuit applications, which was considered to complete the PCB. Starting with schematic of the charge controller in Figure 120 below, the design was completed according to the calculation and simulation. The schematic and footprint of the components are selected based on the design requirements. Besides the requirements, multiple test points were created in case of any issues occur during the operation to find any problem easily.

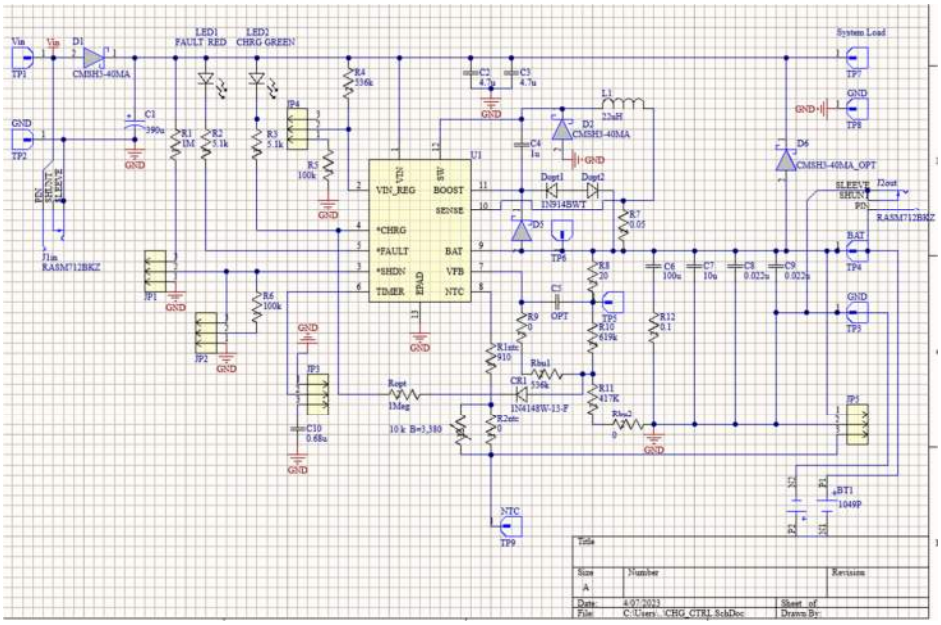

Figure 121 - Power Supply PCB Schematic

Similar to the power supply, before the PCB layout, the setting of the four layers stack up were assigned with thickness of 1.6mm (≈63mil) as seen the Figure 114 below that shows the stack of the four signal copper layers for top layer, ground, power, and bottom layer. The thickness and other parameters were selected according to JLCPCB manufacturer capabilities. Beside the stack up layers, other

manufacturer's restrictions for design rules were followed such as clearance for vias, silk to silk, mask to solder, hole to hole, and components clearance; for routings, routing width, and routing and sizing vias were set in Altium Designer. The weight of the layers was chosen to be 1 oz to add some heating protection.

#	Name	Material	Type	Weight	Thickness	Dk	Df
	Top Overlay		Overlay				
	Top Solder	Solder Resist	Solder Mask		0.492mil	3.8	
1	Top Layer		Signal	1oz	1.378mil		
	Dielectric 2	PP-006	Prepreg		3.008mil	4.6	0.02
2	GND	CF-004	Signal	1oz	1.378mil		
	Dielectric 1	FR-4	Dielectric		49.803mil	4.6	
3	PWR	CF-004	Signal	1oz	1.378mil		
	Dielectric 3	PP-006	Prepreg		3.008mil	4.6	0.02
4	Bottom Layer		Signal	1oz	1.378mil		
	Bottom Solder	Solder Resist	Solder Mask		0.492mil	3.8	
	Bottom Overlay		Overlay				

Figure 114 - Layer Stackup for charge controller

During the layout, the components were placed close to the IC chip that has significant impact like diodes, transistors, and voltage divider. See Figure 122 below for all layer's display that shows the top polygon pour and bottom polygon pour which are important for the IC Chip to dissipate heat if occurs. Also, the blue traces are the bottom traces and the red which is hardly seen due to the polygon pour are for the top component's' traces.

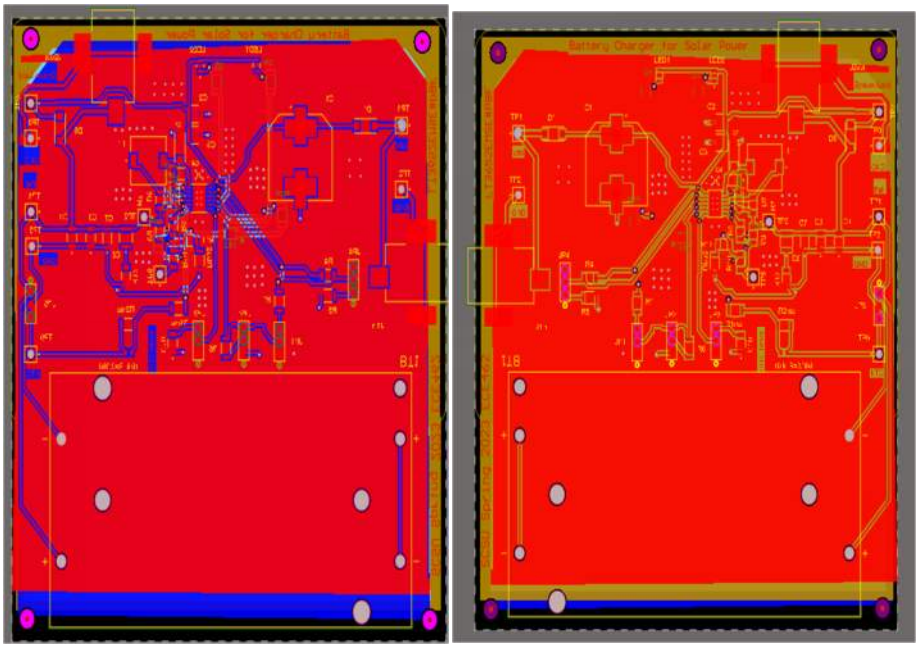

Figure 123 - Signal and Power Traces for Charge Controller

Similar to power supply procedure, after completing the layout above, check for any errors or rules violation and generate Gerber files to be sent to the manufacturers to print the PCB. See Figure 124 below that shows front layer and back layer of this charge controller PCB board.

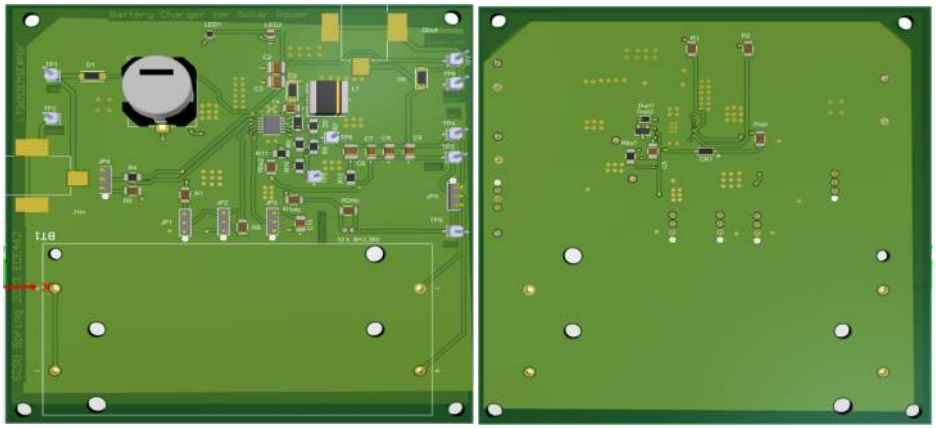

Figure 125 - 3D View of Completed Charge Controller PCB

The PCB of the charge controller is soldered and working as expected. See Figure 118 for completed PCB board for charge controller and see Figure 119, the Altium BOM used in this charge controller PCB.

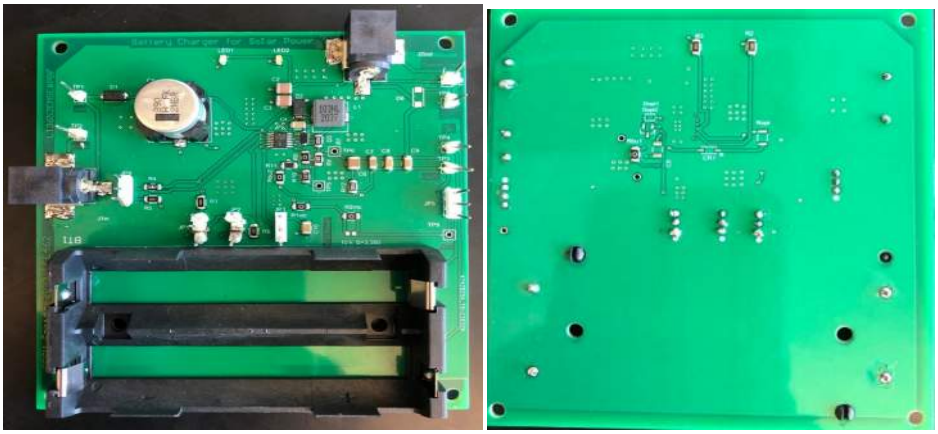

Figure 126 Soldered and Working Charge Controller PCB

	Component	Discreption	Designator	Quantity
2	Jumber	RES 0 OHM 1% 1/4W 1206	Rbu2	1
3	Jumber	RES 0 OHM JUMPER 1/4W 1206	R9	1
4	Jumber	RES SMD 0 OHM 1% 1/4W 1206	R2ntc	1
5	Capacitor	CAP CER 0.022UF 16V X7R 1206	C8, C9	2
6	Sense Resistor	RES 0.05 OHM 1% 1/2W 1206	R7	1
7	Resistor	RES 0.1 OHM 1% 1/2W 1206	R12	1
8	Capacitor	CAP CER 0.68UF 10V X7R 1206	C10	1
9	Resistor	RES SMD 1M OHM 1% 1/4W 1206	R1	1
10	Resistor	RES 1 Meg OHM 1% 1/4W 1206	Ropt	1
11	Diode	DIODE GEN PURP 75V 200MA SOD523F; 1N914BWT	Dopt1	1
12	Diode	DIODE GEN PURP 100V 300MA SOD123; 1N4148W-13-F	CR1	1
13	Capacitor	CAP CER 1UF 50V X7R 1206	C4	1
14	Capacitor	CAP CER 4.7UF 50V X7R 1210	C2, C3	2
15	Resistor	RES SMD 5.1k OHM 1% 1/4W 1206	R3	1
16	Resistor	RES SMD 5.1K OHM 1% 1/4W 1206	R2	1
17	Thermistor	THERMISTOR NTC 10KOHM 3380K 0603; 10 k B=3,380	THERMISTOR NTC	1
18	Capacitor	CAP CER 10UF 10V X5R 1206	C7	1
19	Resistor	RES 20 OHM 5% 1/4W 1206	R8	1
20	Inductor	FIXED IND 22UH 1.41A 110MOHM SMD	L1	1
21	Resistor	RES SMD 100K OHM 1% 1/4W 1206	R5, R6	2
22	Capacitor	CAP CER 100UF 10V X5R 1210	C6	1
23	Capacitor	CAP ALUM 390UF 20% 50V SMD	C1	1
24	Resistor	RES 417K OHM 0.1% 1/4W 1206	R11	1
25	Resistor	RES 536K OHM 1% 1/4W 1206	R4, Rbu1	2
26	Resistor	RES SMD 619K OHM 0.1% 1/4W 1206	R10	1
27	Resistor	RES SMD 910 OHM 1% 1/4W 1206	R1ntc	1
28	Battery Holder	BAT_1049P	BT1	1
29	Header	Connector Header Through Hole 1 position 0.100 (2.54mm); BAT	TP4	1
30	Zenor Diode	DIODE ZENER 6.2V 250MW SOT23-33; BZX84C6V2LT1G	Dopt2	1
31	CHRG GREEN	LED GREEN CLEAR SMD Green 571nm LED Indication - Discrete 2V 0805 (2012 Metric)	LED2	1
32	Schottky Diodes	SILICON SCHOTTKY RECTIFIERS 3.0 AMP, 20 THRU 100 VOLT;CMSH3-40MA	D1, D2	2
33	Schottky Diodes Optional	SILICON SCHOTTKY RECTIFIERS 3.0 AMP, 20 THRU 100 VOLTCMSH3-40MA_OPT	D6	1
34	Red LED	LED RED DIFFUSED 1608 SMD	LED1	1
35	GND	Connector Header Through Hole 1 position 0.100 (2.54mm)	TP2, TP3, TP8	3
36	LT3652EMSE#PBF	IC BATT CHG MULTI-CHEM 12MSOP; LT3652EMSE#PBF	U1	1
37	NTC	Connector Header Through Hole 1 position 0.100 (2.54mm)	TP9	1
38	Capacitor Optional	CAP CER 0.022UF 16V X7R 1206	C5	1
39	Header	Connector Header Through Hole 1 position 0.100 (2.54mm) PH1-01-UA	TP5, TP6	2
40	RASM712BKZ	Power Barrel Connector Jack 2.50mm ID (0.098"), 5.50mm OD (0.217") Surface Mount, Right Angle	J1in, J2out	2
41	Schottky Diode	DIODE SCHOTTKY 40V 2A PMDU; RBR2MM40ATFTR	D5	1
42	Header	Connector Header Through Hole 1 position 0.100 (2.54mm); System Load	TP7	1
43	Header	Connector Header Through Hole 3 position 0.079" (2.00mm); TMM-103-02-L-S	JP1, JP2, JP3, JP4, JP5	5
44	Header	Connector Header Through Hole 1 position 0.100 (2.54mm); Vin	TP1	1

Figure 127 Altium Designer Charge Controller BOM List

Network Manager Breakout Board

The PCB layout implementation for a Network Manager Eterna LTP5902 was designed using Altium Designer software, with the PCB Layout recommendation and typical circuit applications provided in the datasheet taken into consideration. The design process began with studying the datasheet and integration guide, which was completed based on its recommendation [8]. The schematic and component footprints were selected to meet the design requirements, and multiple test points were added to the design to aid in troubleshooting any potential issues during operation. Some issues were encountered during the soldering which the pin pads were under the PCB and hard to solder and it's easy to solve by removing the pads that are not needed to make a space between pads to make sure there is no short circuit; also, the time was not enough to troubleshoot. Refer to Figure 119 for the schematic.

Commented [ASF5]: @Katkov, Arteom , how can these figures be fixed?

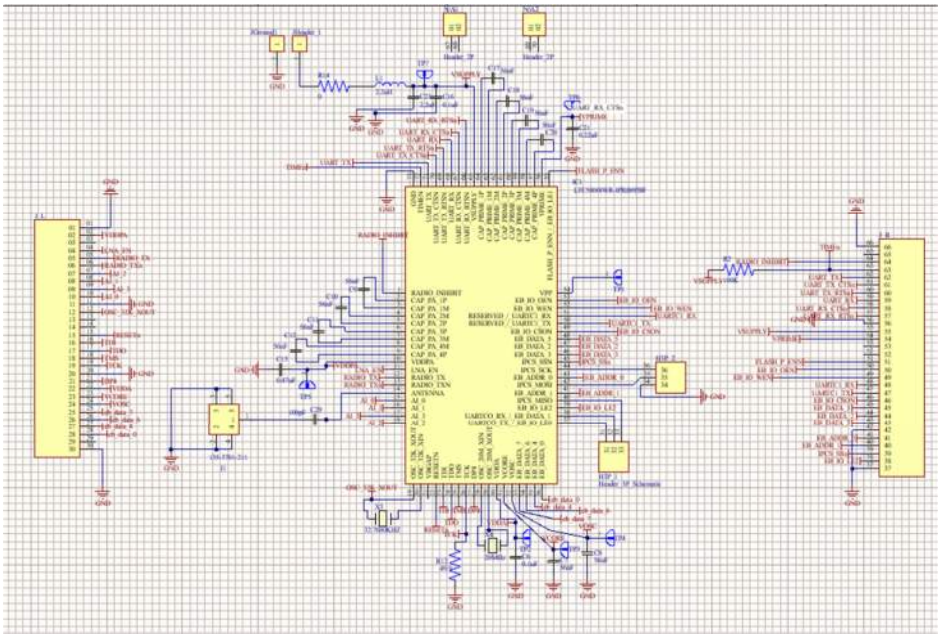

Figure 128 Altium Designer Network Manger schematic

Before beginning the PCB layout, the four-layer stack up was assigned a thickness of 1.0mm (≈39.37mil) see Figure 115 below for the stackup. The selection of the thickness and other parameters, such as the weight of the layers and design rules for clearance, routing width, and sizing vias, was based on the capabilities and restrictions of the JLCPCB manufacturer. Other considerations, including clearance for vias, silk to silk, mask to solder, hole to hole, and component clearance, were also considered. The matching impedance for the trace was designed and calculated using Altium Designer software building tool as seen the Figure 129 below.

#	Name	Material	Type	Weight	Thickness	Dk	Co	Top Ref	Bottom Ref	Width (W1)	Impedance (Z0)	Deviation	Delay...
	Top Overlay		Overlay										
	Top Solder	Solder Resist	Solder Mask		0.5mil	3.8							
1	Top Layer		Signal	1oz	1.378mil		Abc	✓	2 - GND	13.972mil	50	0%	157.19...
	Dielectric 2	PP-006	Prepreg		8.284mil	4.6							
2	GND	CF-004	Signal	1/2oz	0.598mil		Abc	✓ 1 - Top Layer	3 - Power	8.574mil	50.01	0.02%	183.01...
	Core	Core-034	Core		15.748mil	4.6							
3	Power	CF-004	Signal	1/2oz	0.598mil		Bel	✓ 2 - GND	4 - Bottom L...	8.574mil	50.01	0.02%	183.01...
	Dielectric 3	PP-006	Prepreg		8.284mil	4.6							
4	Bottom Layer		Signal	1oz	1.378mil		Bel	✓ 3 - Power		13.972mil	50	0%	157.19...
	Bottom Solder	Solder Resist	Solder Mask		0.5mil	3.8							
	Bottom Overlay		Overlay										

Figure 130 - Layer Stackup for Network Manager

While completing the layout, components were positioned in close proximity to the IC chip, specifically the decoupling capacitors, which have a significant impact on the circuit. See Figure 131 displays all layers of the design, including the top and bottom polygon pours, which are made by the manufacture to meet their restriction capabilities. The blue traces indicate the bottom traces, while the top component traces are marked in red, although they are barely visible due to the polygon pour.

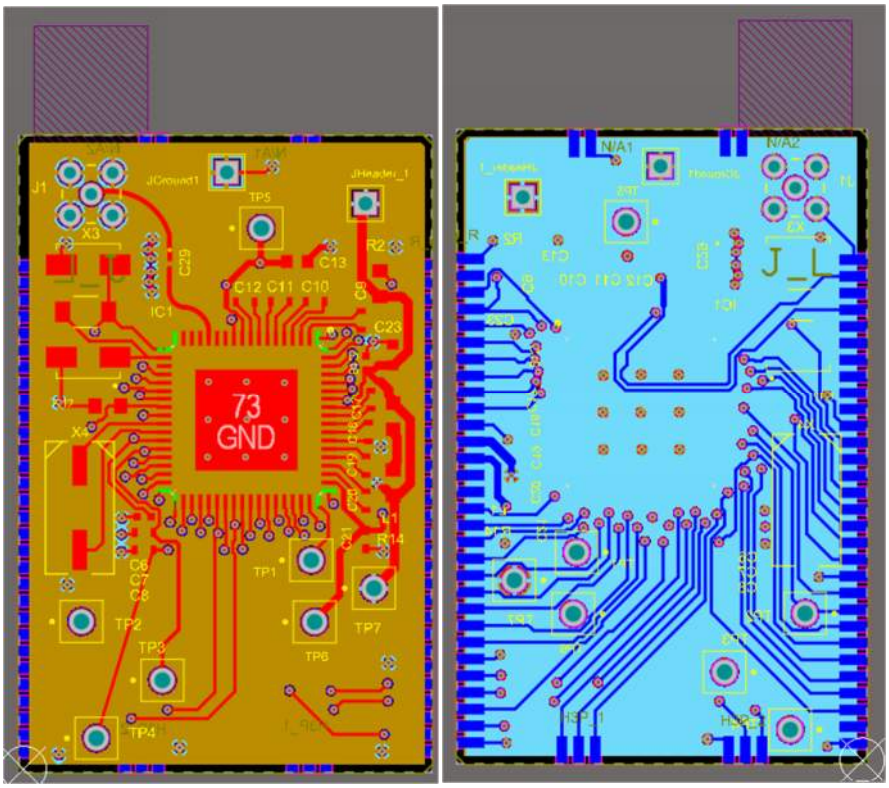

Figure 132 - Signal and Power Traces for Network Manager

Following the layout process outlined for the main board and other PCBs, the completed layout was checked for any errors or violations of design rules before generating Gerber files to be sent to the manufacturer for PCB printing. See Figure 117 provides a view of both the front and back layers of the charge controller PCB board.

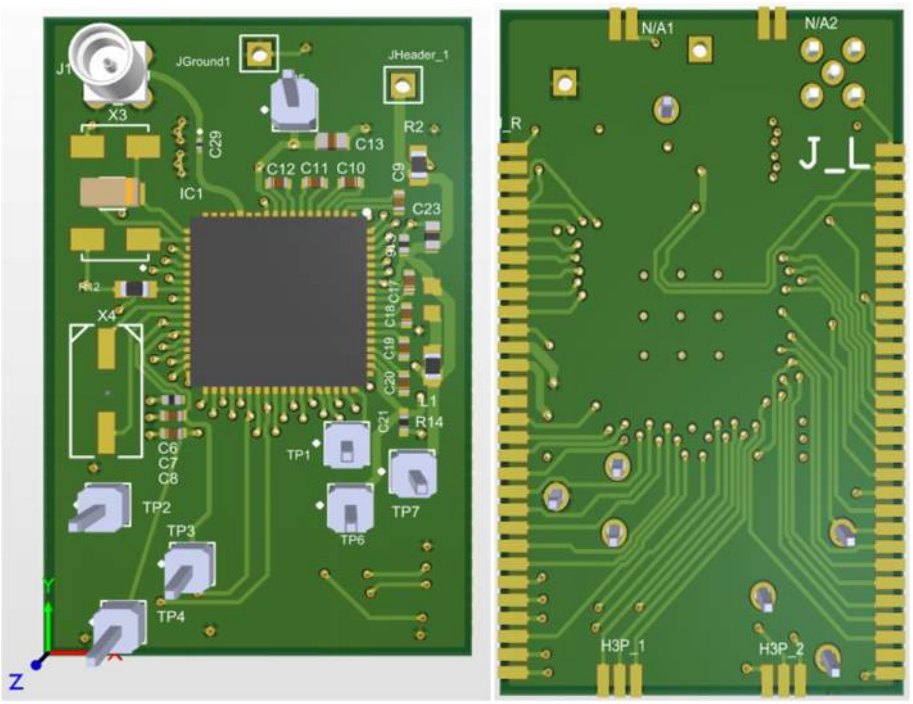

Figure 133 - 3D View of Completed Network Manager PCB

The PCB board for the Network Manager has been successfully soldered using digital microscope to assist soldering the components of sizes 0201 and 0402 which are exceedingly small and hard to be seen by human eyes. See Figure 118 displays the completed PCB board for the Network Manager, while Figure 88 shows the Altium BOM used in the design of this Network Manager PCB. The time was narrow, and the troubleshooting and testing was not complete due to the delay for the manufacture to print the PCB.

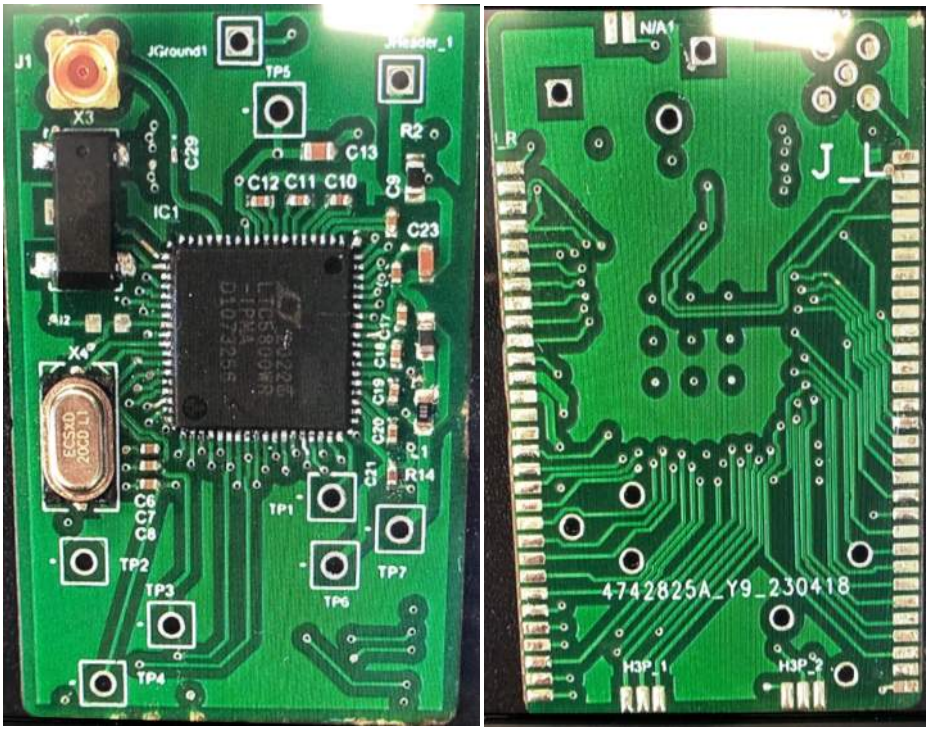

Figure 134 Soldered Network Manager PCB

Component	Description	Designator	Quantity
Capacitor	CAP CER 0.1UF 16V X7R 0402	C6, C16	2
Capacitor	CAP CER 0.056UF 6.3V X7R 0402	C10, C11, C12, C17, C18	10
Capacitor	CAP CER 0.47UF 10V X7R 0603	C13	1
Capacitor	CAP CER 0.22UF 16V X7R 0402	C21	1
Capacitor	CAP CER 2.2UF 10V X7R 0603	C23	1
Capacitor	CAP CER 100PF 25V C0G/NP0 0201	C29	1
Header_3P_Schematic	3 pad Land pattern	H3P_1	1
Header_3P_Schematic_1	3 pad Land pattern	H3P_2	1
LTC5800IWR-IPRB#PBF	Integrated Circuit	IC1	1
135-3701-211	RF Connectors / Coaxial Connectors PC ST JK .068" LEGS	J1	1
TSM-130-01-S-SH-A-P	Connector Header Surface Mount, Right Angle 30 position 0.100 (2.54mm)	J_L	1
TSM-130-01-S-SH-A-P	Connector Header Surface Mount, Right Angle 30 position 0.100 (2.54mm)	J_R	1
87224-1	AMPMODU 1 Position 2.54 mm Single Row Through Hole Straight Header	JGround1, JHeader_1	2
Inductor	FIXED IND 2.2UH 120MA 400 MOHM	L1	1
Header_2P	2 pad Land pattern	N/A1, N/A2	2
Resistor	RES SMD 0 OHM JUMPER 1/10W 0603	R2	1
Resistor	RES SMD 49.9 OHM 1% 1/4W 1206	R12	1
Resistor	RES SMD 0 OHM JUMPER 1/10W 0603	R14	1
PH1-01-UA	Connector Header Through Hole 1 position 0.100 (2.54mm)	TP2, TP3, TP4, TP5, TP6,	7
Crystal	CRYSTAL 32.7680KHZ 12.5PF SMD	X3	1
Crystal	CRYSTAL 20.0000MHZ 10PF SMD	X4	1

Figure 135 Altium Designer Network Manager BOM List

Chapter 3: Testing Process

The overall testing process was divided into three separate phases. Phase one would use commercial evaluation modules used with a breadboard version of the MCU. Basic firmware code was written and each of the components was tested to ensure they would work for the system designed specifications. In the next phase, breakout PCB modules would be designed for each of the major components in the block diagram. The PCB design would then be evaluated and each of the modules could be tested separately. Then, an entire system could be set up from all the separate PCB modules. The final phase, completed next semester, will involve combining the separate PCB modules into one big PCB board, which will be the final system designed.

Evaluation Modules

The evaluation system used is shown below in Figure 136. In the center, the SAML21 microcontroller is soldered onto a QFN to DIP converter. The network manager is shown in the top left and is connected to the microcontroller through UART port. The LTE modules are shown in the bottom left. Again, the only connection used is the UART TX and RX pins. The power supply and Bluetooth modules are shown on the right.

Until the breakout PCBs were designed, this system was used for developing firmware and testing if the system was functional or not. The Bluetooth module was also eventually changed from the RN-42 Bluetooth Standard module to the HC-05 BLE module. The BLE module consumes much less power than the original one and can still be used with higher baud rates. Another advantage of the BLE module is that it works with phone apps much better and more stable than the original RN-42 Bluetooth module.

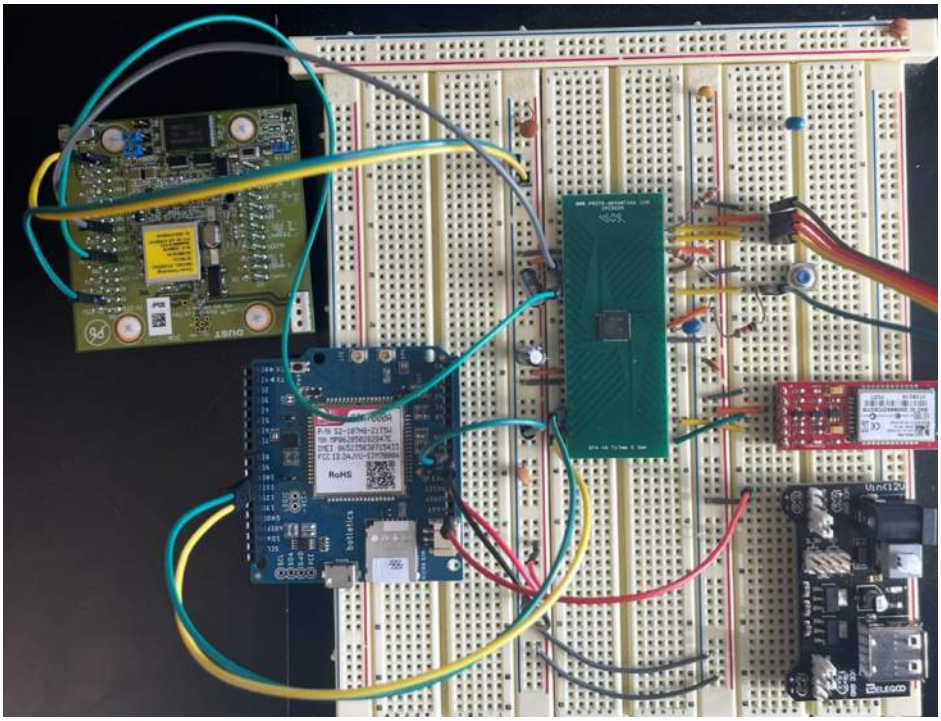

Figure 136 - Evaluation Module System Setup

Breakout PCB Modules

The breakout modules which were described in the hardware section are shown in Figure 137 and Figure 138. These boards were tested separately with of them working as was expected. Unfortunately, the MCU and network manager PCB stopped working for some reason and due to the part shortage, no new PCB modules were able to be purchased. Because of this, the final system tested at the end of the first semester did not include this module. However, the LTE module and the power supply PCB both worked perfectly and were incorporated into the system. The final system used for testing is shown in Figure 140.

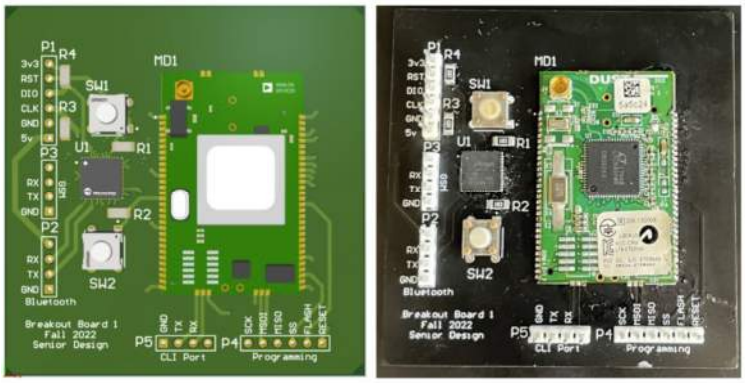

Figure 137 - MCU and Network Manager PCB

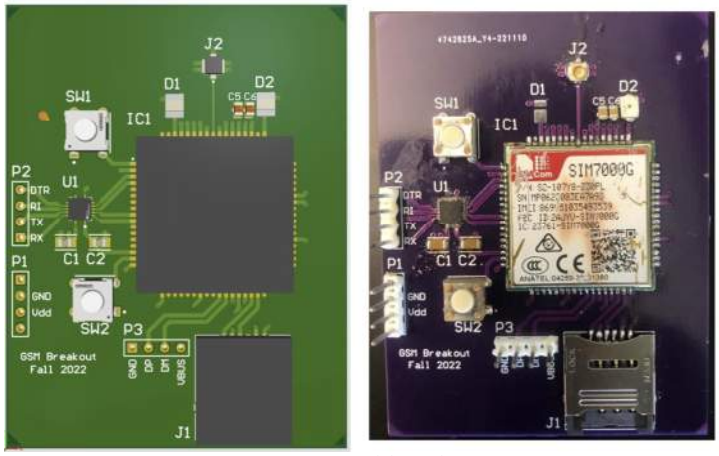

Figure 138 – LTE Module Breakout PCB

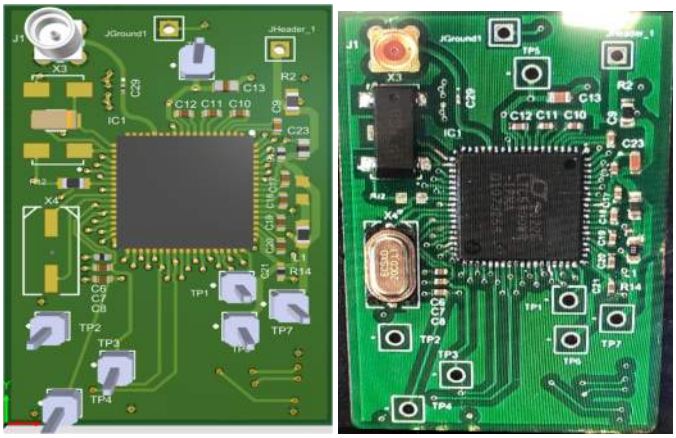

Figure 139 - Network Manager Breakout PCB

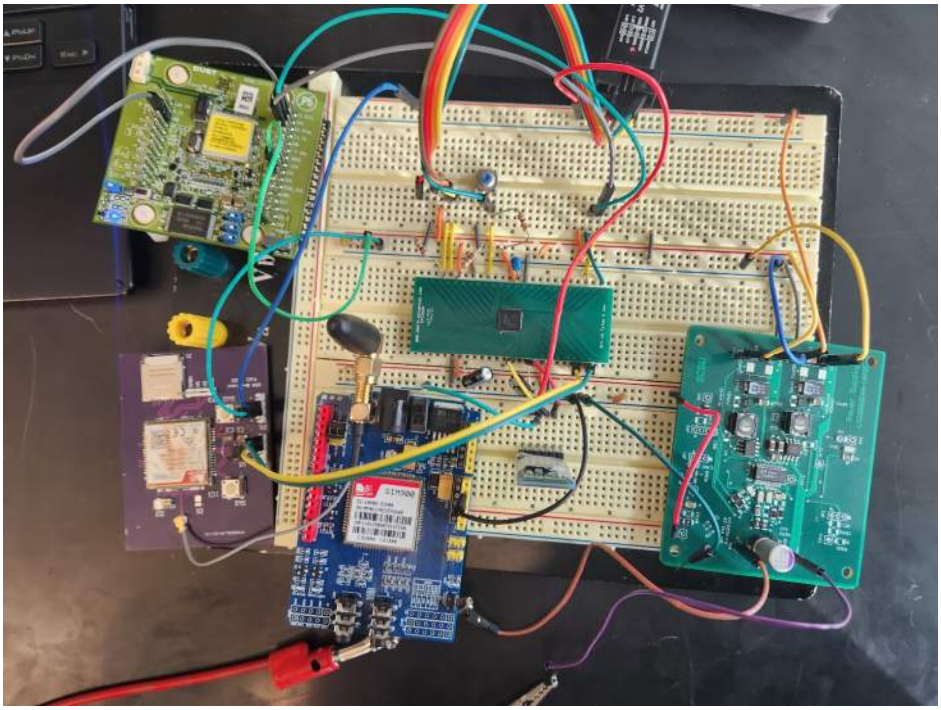

Figure 140 - Final Testing System

The network manager and MCU were the same ones used in the original testing circuit. The power supply is shown on the right and contains the two outputs. On the bottom left, the purple PCB is the LTE module. The antenna was used from a separate PCB and thus a separate module is shown with the blue PCB. The system was tested with both the C# GUI and the phone app. Both worked successfully with data transmission working as expected.

The new BLE modules are also shown in the center. The module was set up using AT commands like that of the LTE module to setup the name and baud rate of the module. It used the 4.0V supply from the power supply PCB. The entire system was also run off a 6V battery confirming that the system does not need a lab power supply to function.

Chapter 4: Results

Commented [ASF6]: @Katkov, Arteom; @Abdi, Zakaria M; @Koop, Isaac P; Do you need to put stuff here?

Firmware and Software

The goal for the end of this semester was to be able to collect data and plot it in a local location using Bluetooth. Storing data remotely was to be saved for the second semester. Thus, mote firmware was developed to collect two distinct types of data: temperature and CO2 levels. Both motes were then going to be connected to the network manager and would send a packet once a second to be plotted on the GUI.

As explained in the GUI section of the report, whenever a new mote would connect and start sending data over, a new series plot would be created which would be used to store the data for that specific MAC Address. For testing the network ID of the SmartMesh IP network was set to 1229 and the join key was set to 0. Once the motes were connected, the system was tested by checking if the data was able to be displayed on the plotting screen of the GUI. This process is shown below in Figure 141.

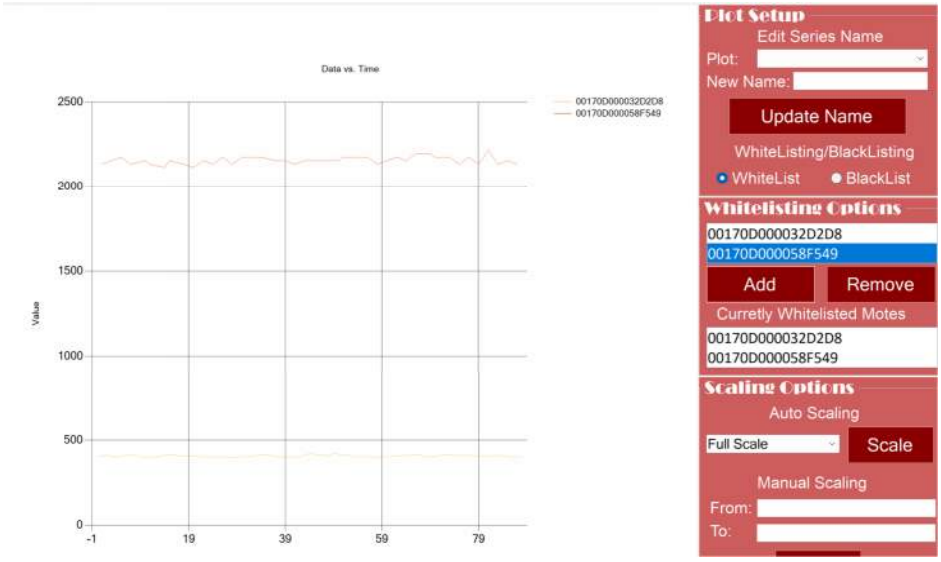

Figure 141 - Data Live Collected

Power Supply Results

In lab experiments, the output voltages required are 3.3V for Network Manager and 4V for GSM LTE Module; the current load of 0.2A is also obtained since the power supply can feed up to 2A. The results of the ripple voltage were done in shielded room (Faraday's Room) in different frequencies; the current is 0.6A and input voltage of 7V. The results were acceptable for our needs to do this project. See the plots of the ripple voltages in Figure 142 below shows a ripple voltage set to 50mV and 1ms and frequency of 210kHz. The ripple voltage peak-to-peak shown in the oscilloscope is 96mV. See the other two plots Figure 143 and Figure 144 below with DC and one in faraday's cage.

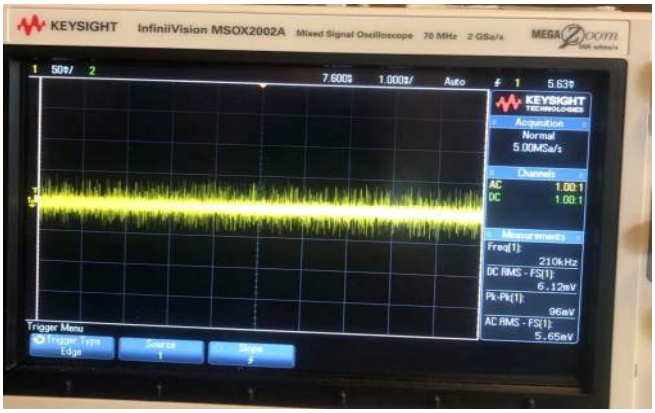

Figure 142 - Voltage Ripple

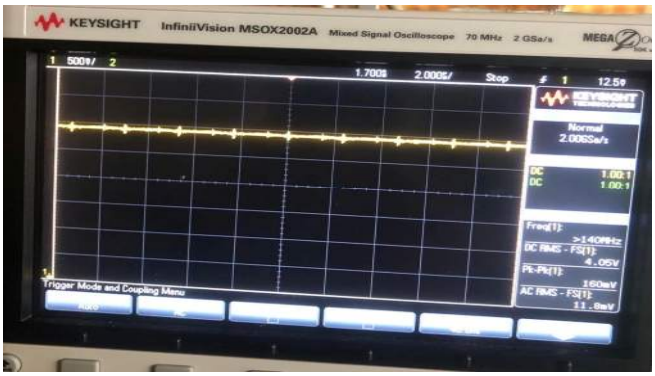

Figure 143 - Voltage Ripple with DC

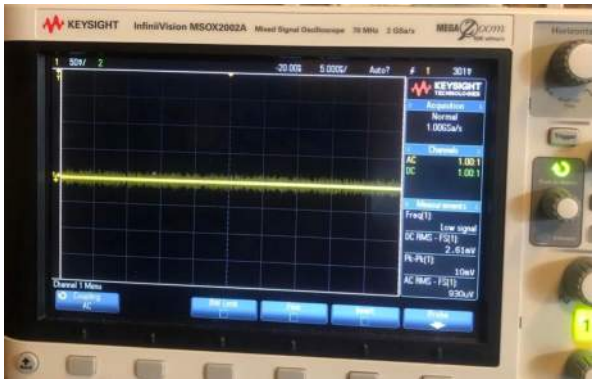

Figure 144 - Voltage Ripple in Faraday Cage

Charge Controller Results

In lab experiments, the results of the charge controller are obtained using the following test procedures based in the Figure 145below:

The procedure involves setting up the power supplies, jumpers, and loads, then monitoring charging current, battery voltage, and LED indicators to ensure proper operation of the circuit.

a) Set up jumper and power supply settings.
b) Turn on PS2 and adjust voltage to 5.4V; turn on PS1 and adjust voltage to 12V while monitoring input current. Adding 5Ω resistor as a virtual battery load.
c) Verify charging current and LED indicators. The current started to show up 0.25A as seen in the experiment setting in Figure.
d) Adjust PS2 and verify input current, battery current, and LED indicators at different battery voltages. The result of after passing 8.3V, the IC stops charging to protect the battery from overcharging.
e) Perform tests for different jumper settings to verify proper operation under various conditions. When the test points JP5 is connected to ground using a jumper, this will indicate the NTC temperature and both LEDS will blink to indicate a fault condition and the charging current will stop.
f) Test the circuit under load and different input voltages. Increasing the power supply PS1 up to 21V as the power supply capable of, the charge current will continue with no issues and the load is measured as input provides because the charge controller is providing power straight from the solar panel and from the battery.
g) The test setup used leads for power connections and multimeters for measurement equipment to ensure accurate readings.

This result shows the proper functioning of the charging system under various conditions. The system's responses to these conditions indicate the correct operation of various features of the LT3652 chip, such as its charge current regulation, voltage regulation, battery preconditioning, and fault detection features.

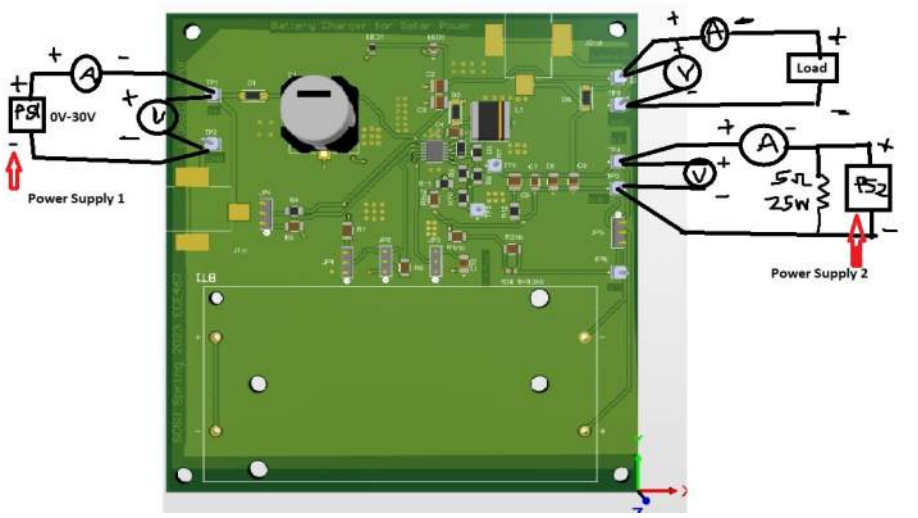

Figure 146: Experiment testing procedure for charge controller

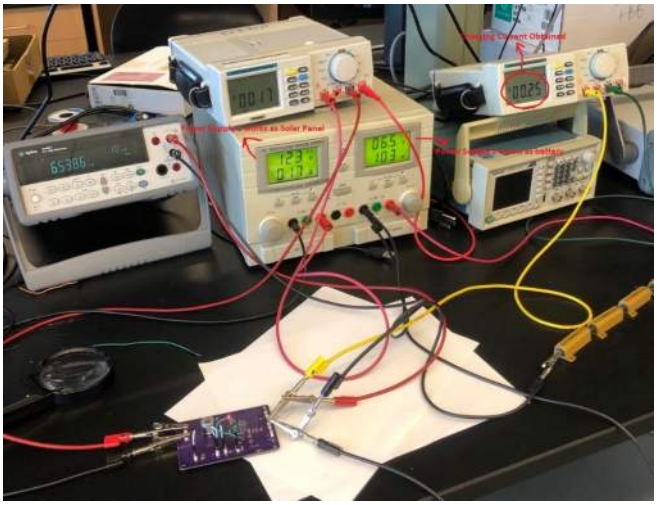

Figure 147: Experiment settings for charge controller

Chapter 5: Costs and Budget

Budget

The following parts will be required as shown in the table below, table 1. The table is split into four columns. The first column shows the required quantity for the different components. In the second column, is a description of the part. Next is the part or model number of the different components. Finally, the last column shows the price per unit with the total cost of the project at the bottom right corner.

Quantity	Description	Part/Model Number	Price Per Unit
1	Bluetooth Module	HC-05	$5.00
3	LTE Module	SIM900	$50.00
5	Microcontroller	ADUCM3027	$15.00
1	Network Manager	DC2274A	$160.00

5	Motes	DC9018B-B	$360.00
6	Printed Circuit Boards		$10.00
1	Components Kit	32BPPK	$50.00
1	Batteries (10 pack)	360821	$20.00
1	Step-Down Transformer		$80.00
1	Miscellaneous		$100
		Total	$2,500.00

Table 1: SmartMesh IP parts to be ordered

Schedule

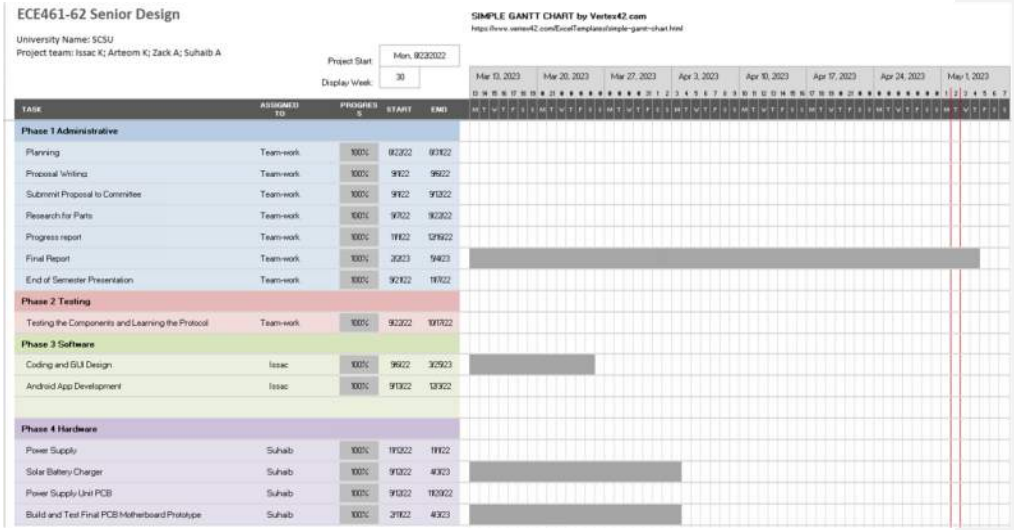

Figure 148: The Schedule for Both Semesters

Chapter 6: Conclusion

The goal of this project is to improve the functionality and flexibility of the SmartMesh IP wireless sensor network by replacing the current requirement for a computer and internet connection with a microcontroller and LTE module. This will allow for the collected sensor data to be directly sent to a database and monitored from a remote location, even in areas without access to a PC or internet. The resulting product will be a configurable network manager that can be easily deployed in remote locations and will be equipped with various features to enhance its functionality.

One of the key features of the updated network manager will be its configurability. This will allow users to set up the sensor network and customize network status notifications. This level of customization will give users greater control over the operation of their sensor network and allow them to tailor it to their specific needs. Communication between the network manager, microcontroller, and LTE module will be achieved through the UART protocol, which is a standard method for transmitting data between devices. The system will also be able to communicate with a phone or computer through Bluetooth or USB, which can be used for initial control and settings as well as conserving power.

In addition to its configurability, the updated network manager will also be designed to be low power, with a duration of operation that depends on the quality of the battery used. When a power source is not available, the system will run off lithium-ion batteries or other high-quality batteries. A rechargeable power supply using solar power will also be used to ensure the system can be used for extended periods of time in remote locations.

In conclusion, the project aims to replace the current computer and internet requirement with a microcontroller and LTE module and has the potential to significantly enhance the functionality and flexibility of the system. The resulting configurable network manager will be able to be deployed in remote locations and will offer a range of customization options to give users greater control over their

sensor network. It will also be designed to be low power and will have the ability to communicate with other devices through the UART protocol using Bluetooth or USB. Overall, this updated network manager will greatly improve the capabilities of the SmartMesh IP sensor network and make it possible to operate and monitor it from any location.

References

1. "K32 L series: Ultra-Low Power Microcontrollers (MCUs) optimized for low-leakage applications," *K32 L Series | Ultra-Low Power Microcontrollers | NXP Semiconductors*. [Online]. Available: https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/k32-l-series-cortex-m4-m0-plus:K32-L-Series. [Accessed: 06-Sep-2022].
2. "K32 L series: Ultra-Low Power Microcontrollers (MCUs) optimized for low-leakage applications," *K32 L Series | Ultra-Low Power Microcontrollers | NXP Semiconductors*. [Online]. Available: https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/k32-l-series-cortex-m4-m0-plus:K32-L-Series. [Accessed: 06-Sep-2022].
3. M. Fahrion, "What is SmartMesh IP?" *Fierce Electronics*, 22-May-2015. [Online]. Available: https://www.fierceelectronics.com/iot-wireless/what-smartmesh-ip. [Accessed: 06-Sep-2022].
4. "SmartMesh® IP™ Wireless Solutions," *Mouser Electronics*, 16-May-2018. [Online]. Available: https://www.mouser.com/new/analog-devices/adi-smartmesh-ip-wireless-solutions. [Accessed: 06-Sep-2022].
5. "Ethereum Energy Consumption" ethereum.org. (n.d.). Available: https://ethereum.org/en/energy-consumption/ [Accessed: 07-Sep-2022].
6. "LT3652 - power tracking 2A Battery Charger for solar power - analog devices." [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/3652fe.pdf. [Accessed: 09-Sep-2022].

7. “Mixed-Signal and digital signal processing ICS | Analog Devices.” [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/LTC3850-3850-1.pdf. [Accessed: 09-Oct-2022].
8. “2 ETERNA LTP5901 AND LTP5902 INTEGRATION GUIDE.” Accessed: May 01, 2023. [Online]. Available: https://www.analog.com/media/en/technical-documentation/user-guides/eterna_ltp5901_ltp5902_integration_guide.pdf