# Milestone 2

Jordan Gaeta, Ian Brown, Gleb Alexeev

March 10, 2016

## 1 Introduction

In this milestone, we had the following tasks:

- Collect and analyze open field range test data
- Develop operating constraints for wireless connectivity

However, there were some bugs to be fixed and extra capabilities that needed to be implemented before we could accomplish these tasks. These included:

- Sending and Receiving from the STM32 to the BlueFruit Module **properly** through SPI
- Allowing payload length to be greater than 16 bytes at a time
- Developing a way to send a constant stream of data to a device connected via bluetooth

## 2 Proper SPI Communication

Milestone 1 was to be able to communicate serially with the BlueFruit Module. We were successful, but the results that we were achieving were inconsistent. Often, we would send an AT command over the SPI channel, and the BlueFruit module would not behave or respond in the expected way. This was due to the way that we were initially sending data through the SPI channel. After Milestone 1, sending messages over SPI happened as follows:

1. Package the entire message and header according to SDEP protocol into a buffer
2. Send the entire buffer over the SPI channel to the BlueFruit module
3. Using a SPI exchange, receive the response from the BlueFruit module

We decided to debug by comparing what we saw on the logic analyzer from the Arduino with what we were doing on the STM32. We saw the following being done via SPI with the Arduino for the write and then the read, respectively.

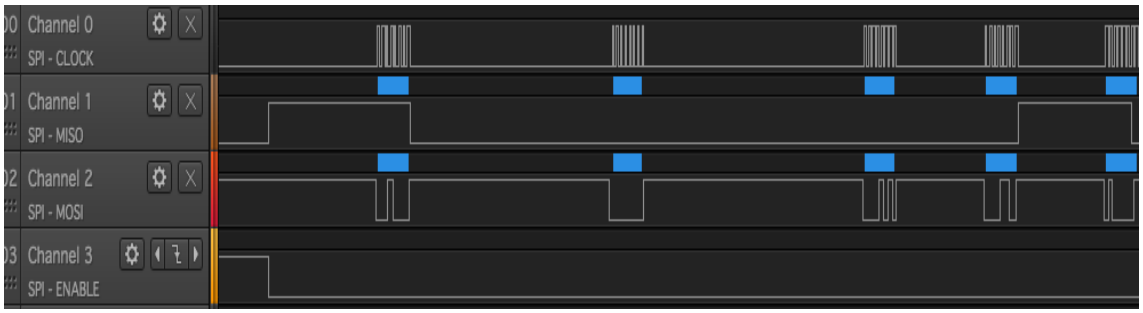


Figure 1: SPI send Arduino


Figure 2: SPI receive Arduino

You'll notice that it first sends the message, and then it pulls Chip Select high. After some time, it then pulls Chip Select low again to receive the message. Another thing to note that is not shown in these figures is that there is also a polling that takes place to check that the SPI device is ready. If the device is not ready, then it returns '0xFE' as a response. We can send '0x10' (command type message indication) to the device until we see something other than '0xFE'. Once we see something else, we can send the message. Our approach to the code became to emulate what was done in the c++ Arduino library. This meant changing the way that we sent messages over the SPI channel. As opposed to before, sending data over SPI now takes place as follows:

1. Package the entire message and header according to SDEP protocol into a buffer
2. Poll the device with '0x10' until we see something other than '0xFE'
3. Send the message one byte at time
4. Poll the device for a message type of '0x10'/'0x20'/'0x40'/'0x80'
5. Receive the message one byte at a time

# 3 Large Payloads (>16 bytes)

The maximum size of a packet not including the header is 16 bytes. Thus, if the amount of data (not including the header) is greater than 16 bytes, we must send the data in multiple packets. Packets are formatted in the following way:

| Name | Type | Meaning |
|---|---|---|
| Message Type | U8 | Always '0x10' |
| Command Id | U16 | Unique command identifier |
| Payload Length | U8 | [7] More data<br>[6-5] Reserved<br>[4-0] Payload length (0..16) |
| Payload | ... | Optional command payload (parameters, etc.) |

The important thing to note is the 'More data' bit that is at the 'Payload Length' position in the header. If the data being sent must be multiple packets, the more data bit should be set to 1 until the last packet. The last packet will have the more data bit set to 0. In order to account for this, our code needs to be able to handle payload lengths of less than and greater than 16 bytes. The sending is handled with the following algorithm:

**Data:** Payload (Message): $[x_0, ..., x_n]$, Size of the message: $i$
**while** $i > 0$ **do**
  **if** $i > 16$ **then**
    $i = i - 16$;
    poll until something other than 0xFE is seen;
    package 16 bytes of data (third bit of header is (16|0x80));
    send the data one byte at a time;
  **else**
    poll until something other than 0xFE is seen;
    package remaining data (i is the amount remaining);
    send the data one byte at a time;
    $i = 0$;
  **end**
**end**

An example of doing this and capturing it via Salae logic is at the top of the next page.

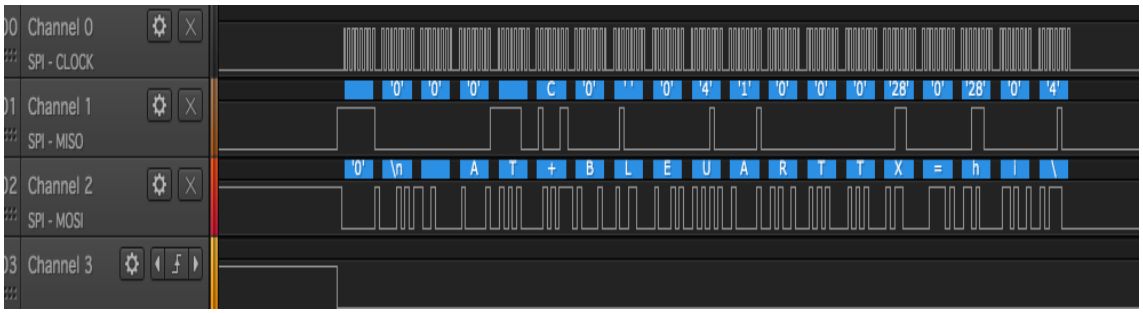


Figure 3: First part of message

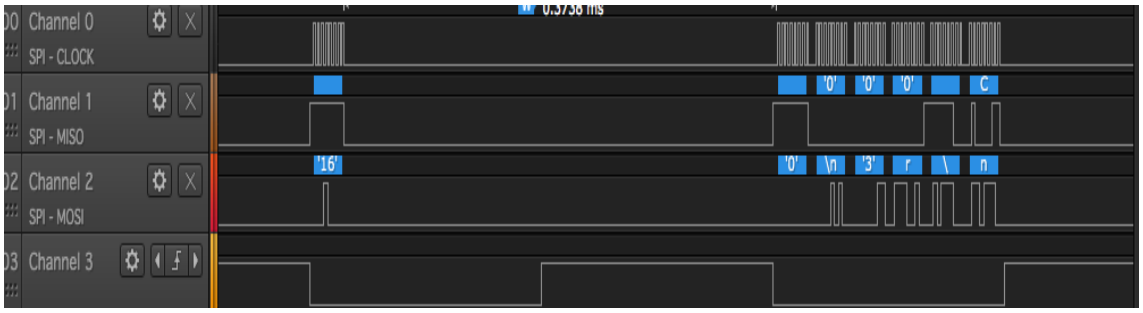


Figure 4: Second Part of Message

# 4 Data Stream

In order to test the bluetooth range, we wanted to take the bluetooth module outdoors and test its transmission to a phone in a variety of conditions. However, we did not want it to be necessary to be plugged in to a serial port to send messages. So, we essentially created a thread that always runs that sends an AT command with a message attached every 1.5 seconds. We decided that we wanted to enclose the board so that we could mimic actual operating conditions. However, the discovery board is too large to enclose. Since we just wanted to test the BlueFruit module range, we decided to do this using the much more portable Arduino board. The logic for the code was easily translated into Arduino code. The user of the phone sees the following via the Bluefruit Application on our sending of the test messages.

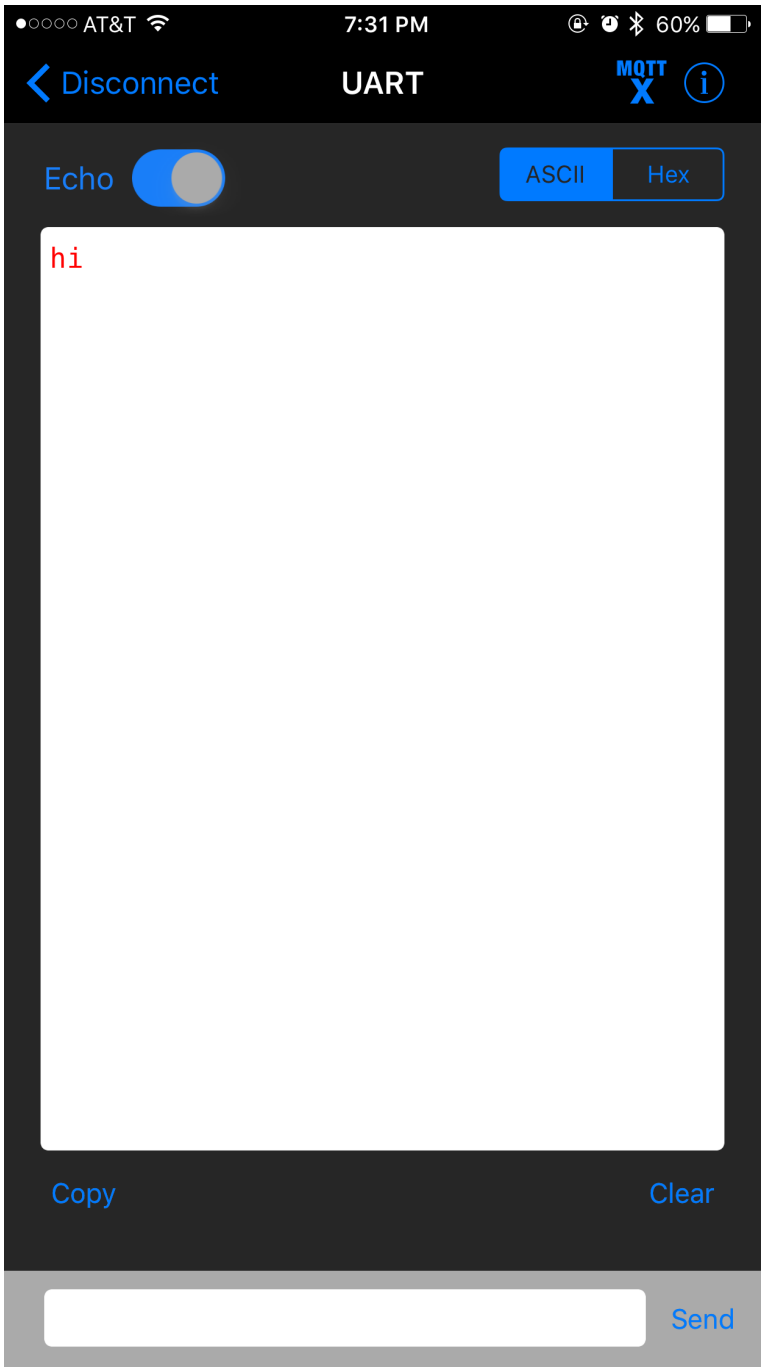


Figure 5: Ian's Phone

# 5 Field Testing

The goal of our testing was to simulate conditions that our system may encounter in real use. We wanted to measure the range with the following conditions:

1. No obstacles
2. Obstacle in way (bridge)
3. Submersion in water

The experiment was done by simply using a tape measure to calculate the distance between the cellphone connected via UART to the BlueFruit module and the BlueFruit module. We achieved the following results at various power levels which are indicated in each table:

| **Scenario (Power Level 0)** | **Range** |
|---|---|
| Open Range | >30 ft. |
| Obstacle | >30 ft. |
| Submersion >3 inches | N/A |
| 3 inches >Submersion >0 inches | ≈17.67 ft |

| **Scenario (Power Level -8)** | **Range** |
|---|---|
| Open Range | >30 ft. |
| Obstacle | >30 ft. |
| Submersion >3 inches | N/A |
| 3 inches >Submersion >0 inches | N/A |

| **Scenario (Power Level -16)** | **Range** |
|---|---|
| Open Range | ≈30 ft. |
| Obstacle | ≈30 ft. |
| Submersion >3 inches | N/A |
| 3 inches >Submersion >0 inches | N/A |

| **Scenario (Power Level -40)** | **Range** |
|---|---|
| Open Range | ≈5 ft. |
| Obstacle | ≈4 ft. |
| Submersion >3 inches | N/A |
| 3 inches >Submersion >0 inches | N/A |

The power levels tested are defined as follows:

| **Power Level (dbM)** | **Watts ($\mu w$)** | **Current ($\mu a$)** |
|---|---|---|
| 0 | 1000 | 303.0303 |
| -8 | 158.5 | 48.0303 |
| -16 | 25.1 | 7.60606 |
| -40 | 0.1 | 0.0303030303 |

# 6 Operating Constraints

Using the data that we found, we can create some basic operating constraints for the bluetooth module. The bluetooth module is only drawing these levels of current and power when it is transmitting. This will only be done when data needs to be pulled from the system using a phone. So, we should be free to use one of the higher power levels so that we may get good range. Assuming that we decide to use the power level which draws 303 $\mu a$, then we have very good results. Obstacles such as a bridge don't seem to affect the range enough to be significant. Submersion in water greatly affects the signal, and the signal will likely be lost if fully submerged in water so data should be collected when it is not submerged.