Trail Counter Project: Final Report - Wireless Group

Gleb Alexeev, Ian Brown, Jordan Gaeta

April 22, 2016

1 Introduction and Background

Over the course of the semester, our group worked with the Adafruit BlueFruit LE SPI Friend, a custom Bluetooth chip with its own firmware installed. It uses high level commands to communicate between itself and the SPI master device. Our goals throughout the semester were the following:

- Understand the communication protocol of the BlueFruit module
- Allow communication between it and an external device connected by Bluetooth, not just the master device
- Allow for large data collection through a custom mobile application
- Test the device in the open field.

These goals were achieved over two months of work, not without their respective challenges.

2 Previous Milestones

Milestone 1

Milestone 1 was able to communicate serially with the BlueFruit module, albeit inconsistently. The command set that it used was the AT-Command set (aka the Hayes Command set), following the SDEP protocol defined in the following way:

Name	**Type**	**Meaning**
Message Type	U8	0x10
Command Id	U16	Unique command identifier
Payload Length	U8	[7] More data [6-5] Reserved [4-0] Payload length (0..16)
Payload	...	Optional command payload (parameters, etc.) Up to 256 Bytes in length

Table 1: SDEP Protocol for a command type message

Using techniques from previous labs done in P442, we wrote a command that sent whatever was typed in the shell to the bluetooth device. For example, *bf cmd AT+BLEUARTRX* would send the command AT+BLEUARTRX to the chip, which then would return everything that is on the buffer of the chip.

We couldn't get *screen* to work properly at the time only due to trying to read in too much data from the RX buffer. However, this wasn't a problem, as the final product has almost no use for *screen* other than for debugging purposes. We did need to find a way to send lots of data at once however.

Milestone 2

After Milestone 1, sending messages over SPI happened as follows:

1. Package the entire message and header according to our current (at the time) 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

This method didn't work for long messages. As such, we analyzed the Arduinos interaction with the Bluefruit Module. This produced Figure 1 and 2:

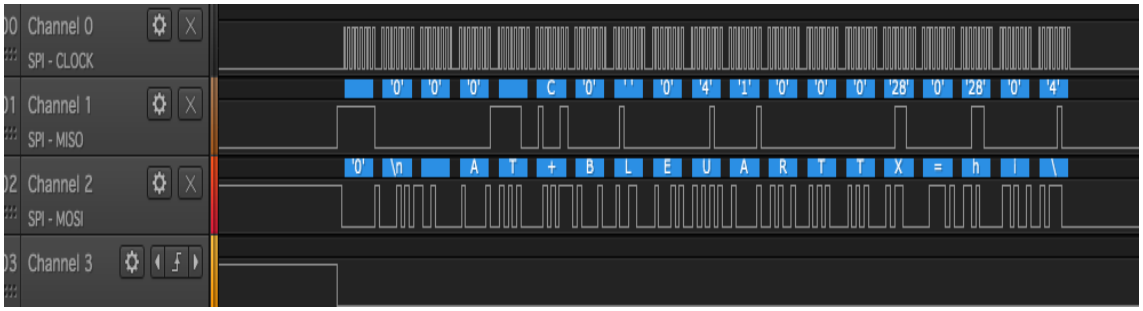

Figure 1: Using Salae Logic; First part of message

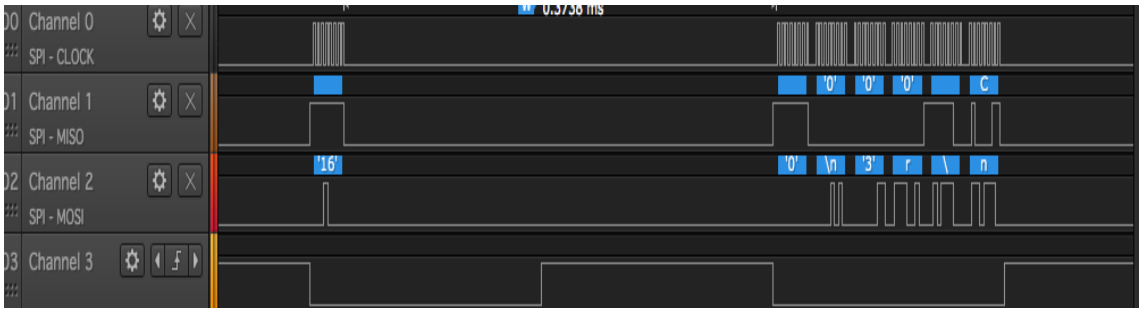

Figure 2: Second Part of Message

I'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.

To continue, If you look back at Table 1 in Milestone 1, you can see that a bit can be set for More data (in the payload length byte). By setting the seventh bit of the Payload Length, you're telling the Bluefruit module that this isn't the full message that you have sent it:

```
messagebuffer[4] = messagebuffer[4] | 0x80;
```

messagebuffer is the message you're trying to send. Note, that the payload length for each message is 16 bytes maximum. This means for messages that send a text message or a collection of data over, you'd have to accommodate for payloads that are larger than 16 bytes. (e.g. *AT+BLEUARTTX=hello world*; it sends the words *hello world* to the device currently connected to the Bluefruit module. Note that the total length of the message is 28, as it is the command + message, and the four bytes of the header (Message Type, Command ID, and Payload Length)).

As a workaround, we devised the following algorithm (which came in handy in future milestones):

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.

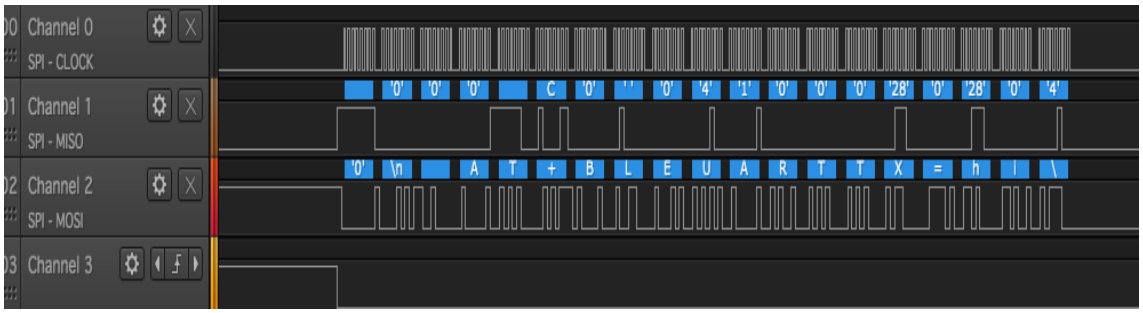

Figure 3: First part of message

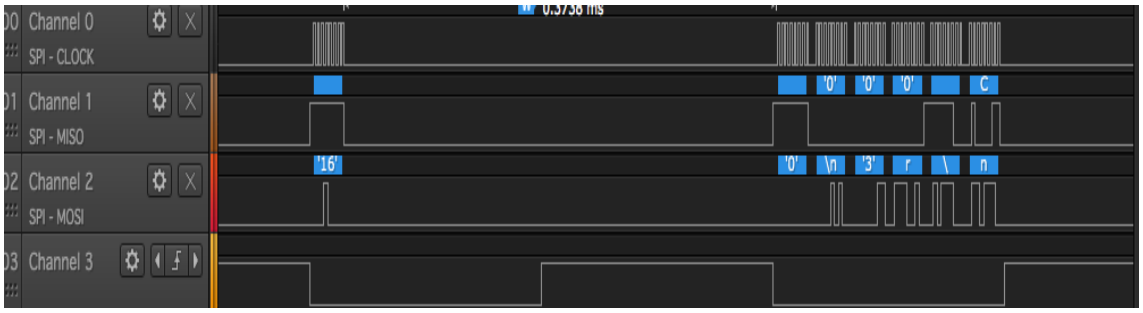

Figure 4: Second Part of Message

Field Testing

As another part of the milestone, we wanted to test out the range of the bluetooth module's activity, as well as its perceptibility in various conditions. As we decided that we wanted to enclose the board to mimic actual operating conditions, we had use the Arduino (the STM32 board was too big). In order to the conditions, we changed the arduino code to send an AT command with a message attached every 1.5 seconds. As soon as it disconnected from the iPhone, we knew that it was not in working range. The user of the phone would see the message "hi" displayed, and repeated every 1.5 seconds or so during its connection with Bluefruit Module.

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, with varvious obstacles (solid, liquid, or none at all), at various power levels which are indicated in each table on the following page. This would help us give constraints to the people installing these machines in the future.

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 (μw)	Current (μa)
0	1000	303.0303
-8	158.5	48.0303
-16	25.1	7.60606
-40	0.1	0.0303030303

Milestone 3

The purpose of milestone 3 was to develop firmware to upload counter data from the trail counter. We wanted to understand how the counter data would be taken in field. As such, we broke it down into several steps:

1. A user comes within range of the trail counter.
2. The user connects to the counter via Bluetooth.
3. Counter data is transmitted to the phone and saved.
4. The counter disconnects and returns to its normal state after transmission.

In order to satisfy these conditions, we needed to

(a) Alter the advertisement interval of the bluetooth module

(b) Check connection between the board, and an external device.

(c) Transfer data (potentially larger than 256 bytes)

(d) Disconnect once the transfer is complete

These tasks were accomplished in the following ways:

(a) We set up an advertising thread that advertised on a certain interval by turning on and off its advertising packets.

(b) Then, we checked for a connection to an external device:

```
while (!connectedFLAG) {
  WriteRead(tx_advstart, 22, tmp_data);
  chprintf((BaseSequentialStream*)&SD1, "Advertising Started\r\n");
  chThdSleepMilliseconds(10000);
  WriteRead(tx_getconn, 21, rx_data);
  if (rx_data[4] == '1') { //Pump Data if we're here
    chprintf((BaseSequentialStream*)&SD1, "Connected\r\n");
    connectedFLAG = 1;
    break;
  } else {
    chprintf((BaseSequentialStream*)&SD1, "No connection, Advertising Stopped\r\n");
    WriteRead(tx_advstop, 21, tmp_data);
    connectedFLAG = 0;
  }
  chprintf((BaseSequentialStream*)&SD1, "Shutting off\r\n");
}
```

The command to check the connection is AT+GAPGETCONN (stored as a char array in the txgetconn buffer). This command returns either a 1 or a 0 depending on whether our bluefruit module is connected to an external device. If we're not connected, we turn the advertising thread off and go to sleep. If we are, we jump out of the loop and continue to send our data.

(c) Following the code updates from milestone 2, we had to figure out how to send packets larger than 256 bytes. After all, if the trail counter stored 6 bytes of data each hour, and the organizations in charge of the land trusts only collected the data every 6 months, we'd be looking at around *25Kb* worth of data! As such, we used the same idea we workaround we did in milestone 2! We send 256 byte packets at a time. (Due to header sizes and all, we could only send 208 bytes worth of actual data at a time.) The algorithm looks something like this:

```
Data: Payload (Message): [x_0, ..., x_n], Size of the message: size
while size > 0 do
    if size > 208 then
        Send the 208 bytes worth of data via method in milestone 2;
        Increment the pointer to the Payload (message) by 208;
        size = size − 208;
    else
        Send the remaining bytes worth of data via method in milestone 2;
        Increment the pointer to the Payload (message) to the end of the
          buffer;
        size = 0;
    end
end
```

(d) Once the message has been sent, issue out a disconnect command via AT+GAPDISCONNECT, which conveniently force-disconnects us from the external device.

Unfortunately in this milestone, we were unable to send a data buffer of arbitrary size across Bluetooth to the iPhone. We found success in sending between one and twelve 256 byte packets, but more packets in immediate succession caused the Bluetooth chip to hang in a "not ready" state indefinitely. In Milestone 4, we develop a fix for this problem.

3 Milestone 4

In Milestone 4, we were tasked with building a mobile application that would collect data from the trail counter. The application had to be able to do the following tasks:

- Collect Data from counter
- Write an epoch time to the counter
- Send the data to a remote server

The application was written in Swift for use on an iPhone. We wanted to make the application easy to use so we decided to only have one view in the application. From this view, you can do any of the required tasks. Once in the application, you will see a list of the available devices.

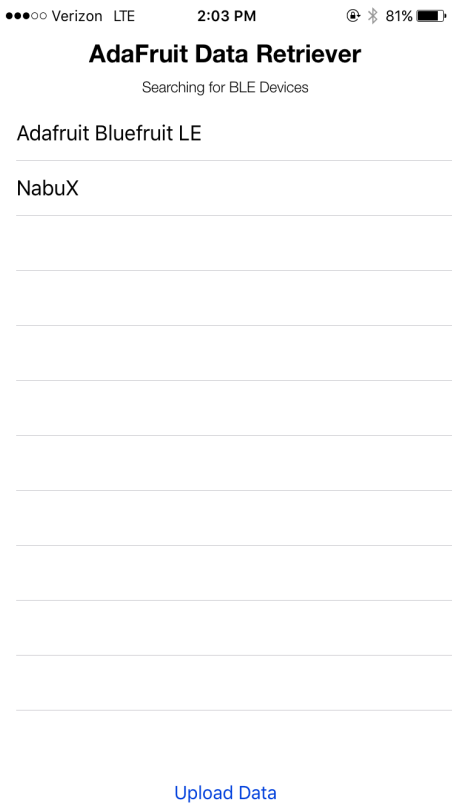

Figure 5: List of available Bluetooth devices

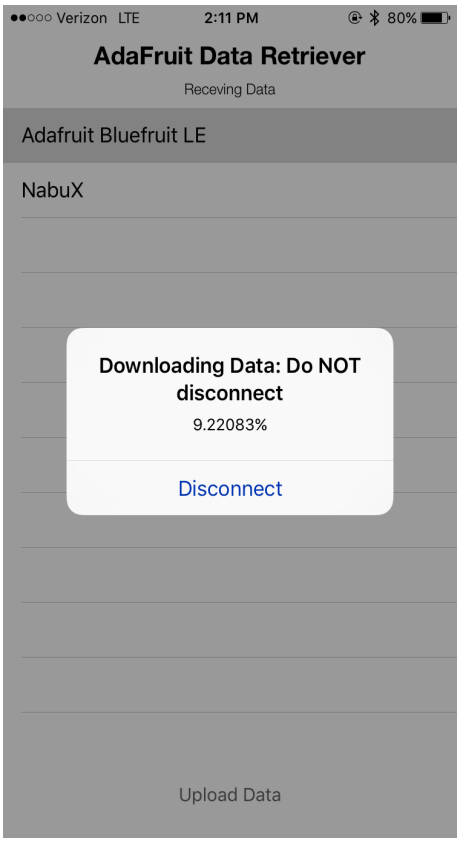

Figure 6: Downloading data from a select device

If you see a device that you would like to connect to, you click on the device name in the table. After clicking on the device name, the phone will connect to the bluetooth module. Upon connection, the phone sends an epoch time to the device. The phone is now ready to receive data from the Bluetooth module, and an alert will pop up that will indicate the percentage of data that has been received from the module. (Figure 6 above)

Upon reaching 100 percent completion, the alert message will disappear. The data is then stored in persistent storage on the user's iPhone. The user can send the data to the server by clicking the 'Upload Data' button. This will send all of the data stored to the web server, and then delete it from the iPhone's storage. The data is sent to the server is a json format that looks like the following:

```
{
    "Trail Counter Data": [0: [data], 1: [data], ... , n: [data]]
}
```

The dictionary is used as a way to separate data from different trail counters. In the future, a unique id could be assigned to each trail counter. This id could then be used as a key in the data dictionary. The server then will parse the json and display it to a web page.

As discussed earlier, we had an issue sending data buffers larger than about 2300 bytes. When attempting to send a buffer larger than 2300, the Bluetooth chip would be held indefinitely in a "not ready" state. While the exact issue is not known to us, we decided to develop a workaround that would allow us to send any size data buffer to the phone. This workaround is described below:

Data: Payload (DataBuffer): $[x_0, ..., x_n]$, Size of the message: $size$
while $size > 0$ **do**
  if $size >= 2304$ **then**
    Send DataBuffer to WriteReadWrapper with size 2304;
    Thread sleep for 7 seconds;
    $DataBuffer = DataBuffer + 2304$;
    $size = size - 2304$;
  else
    Send DataBuffer to WriteReadWrapper with size $size$;
    Thread sleep for 1 second;
    $DataBuffer = DataBuffer + size$;
    $size = 0$;
  end
end

This workaround is oddly similar to the two other workarounds mentioned earlier (for messages that are between 16 and 256 bytes, and for messages greater than 256 bytes respectively) When this workaround is used, we completely receive a data buffer of arbitrary size on the iPhone. Unfortunately, this means that sending large amounts of information through Bluetooth will require more time of the user, but this functionality is necessary if our project is to be useful. The reason for waiting, we believe, has more to do with the firmware of the Bluetooth chip handling large packet sends. We believe there is a "send buffer" within the chip that has a maximum size, and if information is loaded into the buffer quicker than it can send the information, it will be permanently locked in a "not ready" state. Therefore, by loading the maximum amount of 256 byte packets (twelve) and sleeping for 7 seconds, the chip has enough time to send each packet and clear its "send buffer" before sending another set of packets.

4 Future Work

While this project accomplished a great deal, there are still many issues left to resolve. Below, we have listed several areas that need to be improved or worked on.

- While our iPhone application can send epoch time to the Bluetooth chip, the STM board has trouble receiving the time correctly. This is because our receive function only handled information like "OK" or "ERROR" when interacting with the Bluetooth chip. When we want to obtain more important information, our receive function fails to read in anything worthwhile.
- Our iPhone application does not handle the case when a user disconnects from the Bluetooth chip when receiving data. If this occurs, we cannot guarantee that the data can be received again from the moment the disconnect occurred.
- There are obviously still bugs within the STM code, and cases such as missing data or Bluetooth chip malfunctions need to be handled. However, for now, this project works in most cases with reasonable results.
- Optimization can occur within the STM code, the iPhone code, and the Bluetooth firmware. Optimizations could include revamping the SPI protocol to make it easier to work with, and removing restrictions on the amount of data being sent across Bluetooth

We're confident that these issues can be resolved if more time was allotted and more information about the Bluetooth chip, the SPI protocol, and Bluetooth Low Energy was available. We look forward to seeing how these problem areas are addressed and to seeing how the trail counter as a whole develops.