Milestone 1: The Beginnings of a Beautiful Relationship
Gleb Alexeev, Ian Brown, Jordan Gaeta

Contribution to TrailCounter: Bluetooth wireless collection of data and firmware updates.

Milestone: Send/Receive serial data stream across bluetooth channel usng SPI Bluefruit module and provided app.

Background for the work:
Initially, what was thought to be an impossible task was soon realized through teamwork, understanding, and the professor giving us a 5 minute explanation setting everything straight. Initially we believed that we have to work with the NRF51xxx chip that is on the given bluetooth (images below and graphs below). However, the chip already comes with a higher level command set downloaded onto it (as it has it's own M0 cortex processor). The command set is based on the Hayes Command set, with a little bit of a twist (message sending format).

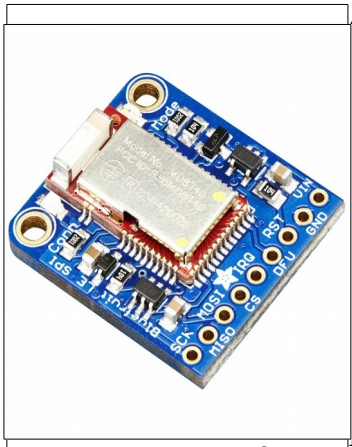

Figure 1: The Bluefruit Bluetooth Chip in question

So realizing that all we have to do is write a SPI protocol for the Adafruite Bluefruite LE SPI Friend, our lives became easier. The SPI protocol works in the way of Master and Slave, and coincidentally, we had two other chips to go off of: the pressure sensor, and the wireless chip.

We decided to wire using SPI-1 pins (A5 – SCK, A6 – MISO, A7 – MOSI, A8 – CS, E3 – Does nothing but that's CS for the gyro, C4 C5 – RX/TX). (see image below) Then going off of the gyro's read/write commands, we create a write/read that takes in two char arrays, and a size (for the tx array). As well, we hooked up Salae logic to see the signals sent and received.

From this point on, we create a command (cmd_bluefruit) that creates s_data and r_data (send buffer and receive buffer). Then, we send the send buffer and receive the receive buffer and print out the values. Simple as pie. Except not...

Difficulties: There were several big problems that arose and they all started with the fact that we were not sending data over correctly. Having looked up the Hayes Command set and the way a message, we found the following information:
(from the official github page, lots of info)
“The Simple Data Exchange Protocol (SDEP) can be used to send and receive binary messages between two connected devices using any binary serial bus (USB HID, USB Bulk, SPI, I2C, Wireless, etc.), exchanging data using one of four distinct message types (Command, Response, Alert and Error messages).

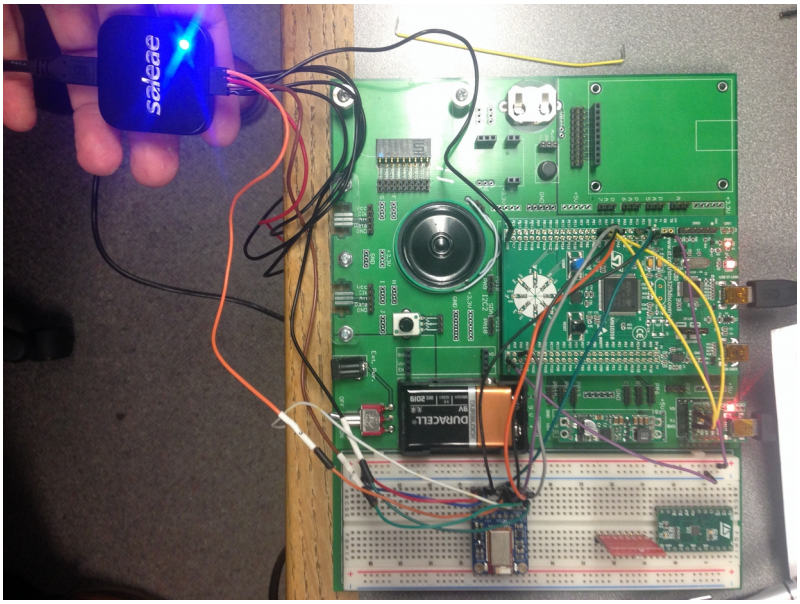

Figure 2: Wiring is similar to previous chips. (save for the Salae)

The first byte of every message is an 8-bit identifier called the **Message Type Indicator**. This value indicates the type of message being sent, and allows us to determine the format for the remainder of the message.

Message Type	ID (U8)	Description
Command	0x10	
Response	0x20	
Alert	0x40	
Error	0x80	

Command Messages

Command messages (Message Type = 0x10) have the following structure:

Name	Type	Meaning
Message Type	U8	Always '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.)

Command ID (bytes 1-2) and **Payload Length** (byte 3) are mandatory in any command message. The message payload is optional, and will be ignored if Payload Length is set to 0 bytes. When a message payload is present, it's length can be anywhere from 1..16 bytes, to stay within the 20-byte maximum message length.

A long command (>16 bytes payload) must be divided into multiple packets. To facilitate this, the **More data** field (bit 7 of byte 3) is used to indicate whether additional packets are available for the same command. The SDEP receiver must continue to reads packets until it finds a packet with **More data == 0**, then assemble all sub-packets into one command if necessary.

The contents of the payload are user defined, and can change from one command to another.

A sample command message would be:

0: Message Type (U8)	1+2: Command ID (U16)	3: Payload Len (U8)	4: Payload (...)
10	34 12	01	FF

- The first byte is the Message Type (0x10), which identifies this as a command message.
- The second and third bytes are 0x1234 (34 12 in little-endian notation), which is the unique command ID. This value will be compared against the command lookup table and redirected to an appropriate command handler function if a matching entry was found.
- The fourth byte indicates that we have a message payload of 1 byte
- The fifth byte is the 1 byte payload: 0xFF

Response Messages

Response messages (Message Type = 0x20) are generated in response to an incoming command, and have the following structure:

Name	Type	Meaning
Message Type	U8	Always '0x20'
Command ID	U16	Command ID of the command this message is a response to, to correlated responses and commands
Payload Length	U8	[7] More data [6-5] Reserved [4-0] Payload length (0..16)
Payload	...	Optional response payload (parameters, etc.)

By including the **Command ID** that this response message is related to, the recipient can more easily correlate responses and commands. This is useful in situations where multiple commands are sent, and some commands may take a longer period of time to execute than subsequent commands with a different command ID.

Response messages can only be generate in response to a command message, so the Command ID field should always be present.

A long response (>16 bytes payload) must be divided into multiple packets. Similar to long commands, the **More data** field (bit 7 of byte 3) is used to indicate whether additional packets are available for the same response. On responses that span more than one packet, the **More data** bit on the final packet will be set to `0` to indicate that this is the last packet in the sequence. The SDEP receiver must re-assemble all sub-packets in into one payload when necessary.

If more precise command/response correlation is required a custom protocol should be developed, where a unique message identifier is included in the payload of each command/response, but this is beyond the scope of this high-level protocol definition.

A sample response message would be:

0: Message Type (U8)	1+2: Command ID (U16)	3: Payload Len (U8)	4: Payload (...)
20	34 12	01	FF

- The first byte is the Message Type (0x20), which identifies this as a response message.
- The second and third bytes are 0x1234, which is the unique command ID that this response is related to.
- The fourth byte indicates that we have a message payload of 1 byte.
- The fifth byte is the 1 byte payload: 0xFF”

So the whole time we tried to send and receive messages, we were sending and receiving them the wrong way. The data we got didn't make any sense. It was garbage. We sifted through pages of code from Arduino and Adafruit Bluefruit Libraries trying to figure out the encoding of the messages. SDEP is a custom command set that followed the rules that had no sort of documentation but what was listed above, and following those rules didn't help.

But then, a ray of light shone upon us. God guided us through the fields of valor and the mountains of Valhalla, and as we soared through the skies of heaven, we found what we were looking for… A guy that was doing literally the exact same thing but with a UART! Alas! Our prayers have been answered!

```
0x10 : 0x00 0x0A : 0x02   : 0x41 0x54
Type : ID        : Length :  'A'  'T'
```

Figure 3: The actual way to do and test things

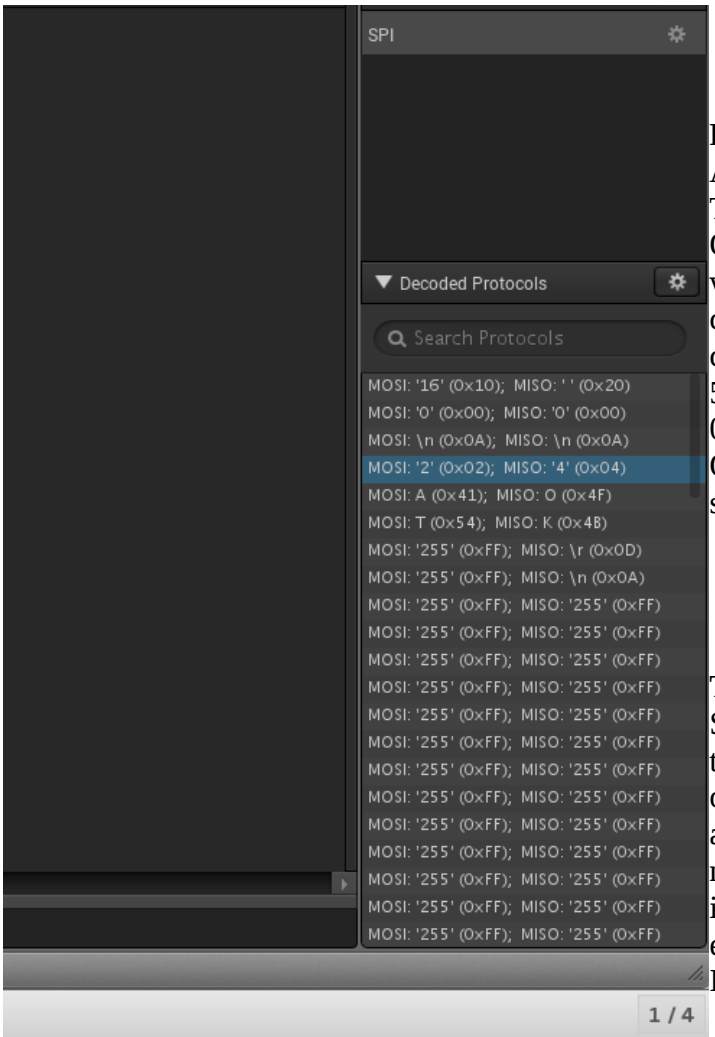

Figure 4: IT'S ALIVE!!! (see bottom right decoding: O,K,\r,\n. Just as expected)

Results:
Above is the encoding for the Command 'AT'. Through this command we should just receive OK\r\n or Error\r\n if the chip is ready or not. And we have! (see figure 4). We had to hardcode the command as screen doesn't decode the messages correctly, possibly due to the baud rate (see Figure 5) (uint8_t testAT[6] = {0x10, 0x00, 0x0A, 0x02, 0x41, 0x54};), and then ran it through Salae Logic. On the next pages are the results for the signals sent through.

The Next Step:
So now that we know the encoding and know how to sent the message correctly and what to expect, our next step is to write a list of AT commands, and interpretations. i.e. type message, length of message is all depended on what is passed through in Screen and the ChibiOS. At that point, we can easily utilize the accelerometer through internal I2C wiring and go do field testing.

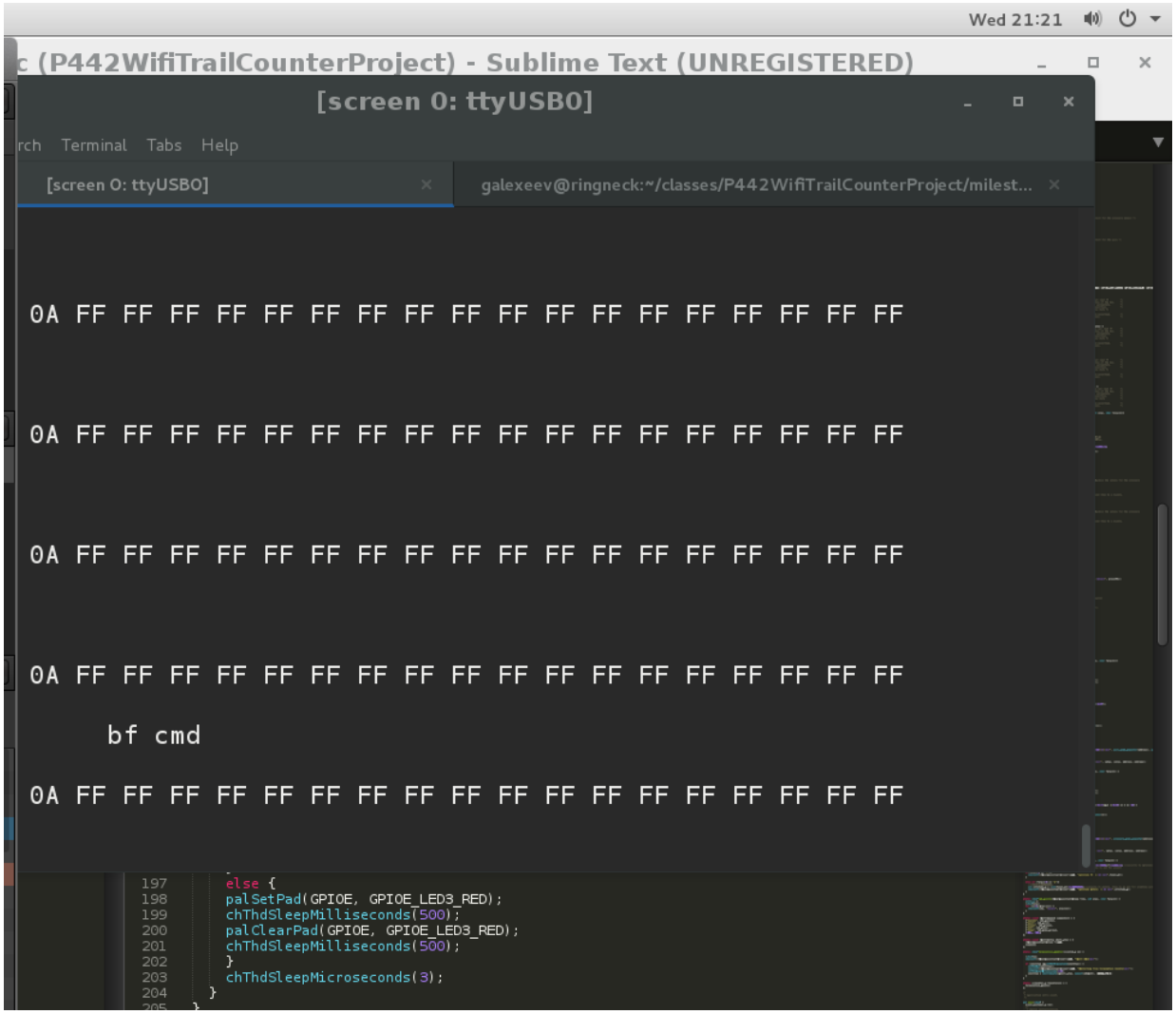

Figure 5: Screen does not read the values in correctly (screen is set at 38400 baud and the chip is at 115200).

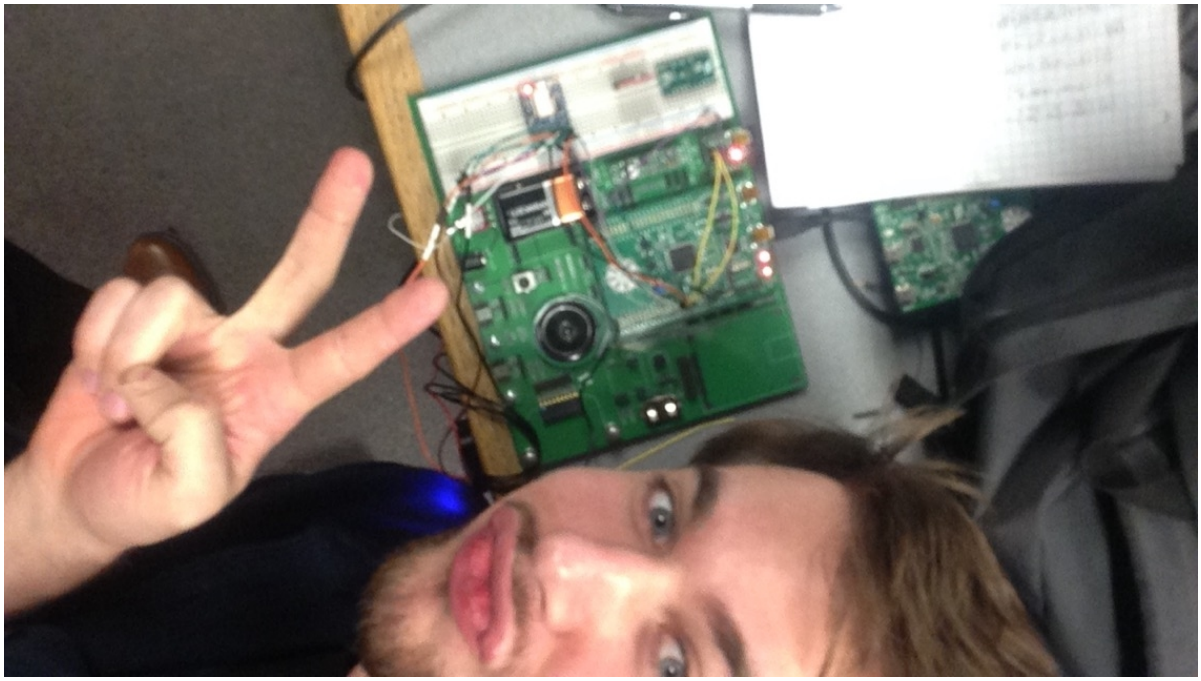

Figure 6: The elusive Gleb, hard at work (it works! Thank god!!!)