Reverse Engineering Fitbit BLE Protocol

###Intro: Since the announcement of iOS 8, I've been interested in getting integration between my Fitbit One and the HealthKit app. Unfortunately Fitbit doesn't support integration with the HealthKit app, and the APIs for accessing the data as a third party are very lackluster. This post is a brain dump of the information I've discovered about communicating with a Fitbit device (specifically a Fitbit One) over the bluetooth low energy protocol.

###Resources/Tools: * Hopper Disassembler * decrypted Fitbit app (jailbreak required) * class-dump * Bluetooth Explorer

###Results: There have been a number of attempts to document the process of syncing and pulling data from a fitbit dongle over bluetooth and USB, however there is still not much in the ways of a public implementation for bluetooth data sync outside of the official fitbit mobile app. At the bottom of this post is a list of sources i used in compiling this data as well as some code implementations of the protocols.

####Packet Format Bluetooth Low Energy packets are constrained to 20 bytes. The packets always begin with a control code, `0xC0`. This is followed by an Airlink Opcode. Listed below are the known Opcodes I was able to document from the binary:

```
enum FBAirlinkOpcode {
    FBAirlinkOpcodePollHost,
    FBAirlinkOpcodeResetLink                    = 0x01,
    FBAirlinkOpcodeAck                          = 0x02,
    FBAirlinkOpcodeNak                          = 0x03,
    FBAirlinkOpcodeDisconnect                   = 0x04,
    FBAirlinkOpcodeAlertUser                    = 0x05,
    FBAirlinkOpcodeSetBondMode                  = 0x06,

    FBAirlinkOpcodeStartAirlink                 = 0x0A,

    FBAirlinkOpcodeDump                         = 0x10,

    FBAirlinkOpcodeUserActivity                 = 0x12,
    FBAirlinkOpcodeEcho                         = 0x13,
    FBAirlinkOpcodeInitAirlink                  = 0x14,

    FBAirlinkOpcodeObsoleteRxAck              // = 0x??
    FBAirlinkOpcodeReadTrackerBlock           // = 0x??
    FBAirlinkOpcodeReadTrackerMemory          // = 0x??
    FBAirlinkOpcodeReadFirstHostBlock         // = 0x??
    FBAirlinkOpcodeReadNextHostBlock          // = 0x??
    FBAirlinkOpcodeReadAirlinkBlock           // = 0x??
    FBAirlinkOpcodeUpdateTrackerBlock         // = 0x??

    FBAirlinkOpcodeXFR2HOSTSingleBlockResponse  = 0x40,
    FBAirlinkOpcodeXFR2HOSTStreamStarting       = 0x41,
    FBAirlinkOpcodeXFR2HOSTStreamFinished       = 0x42,

    FBAirlinkOpcodeClientAuthStart              = 0x50,
    FBAirlinkOpcodeClientAuthChallenge          = 0x51,
    FBAirlinkOpcodeClientAuthChallengeResponse  = 0x52,
}
```

Each Airlink Opcode describes a different packet structure. I have been able to document some of these but without more investigation with bluetooth debugging tools I won’t be able to map the full structures.

Control Packet:

```
[_1]
[C0]
1. HEADER - Control Packet

Use: sent twice from the device once the BLE link is established.
```

Reset Link:

```
[__1__]
[C0 01]
1. HEADER - FBAirlinkOpcodeResetLink

Use: seems to be used to reset and flush any logic setup from previous
```

Ack:

```
[__1__]
[C0 02]
1. HEADER - FBAirlinkOpcodeAck
```

Nak:

```
[__1__] [__2__]
[C0 03] [15 20]
1. HEADER - FBAirlinkOpcodeNak
2. these two bytes were always "15" and "20" for me, not sure of the si
```

Disconnect:

```
[__1__]
[C0 04]
1. HEADER - FBAirlinkOpcodeDisconnect

Use: End communication stream
```

Alert User:

```
[__1__]
[C0 05]
1. HEADER - FBAirlinkOpcodeAlertUser

Use:
```

Set Bond Mode:

```
[__1__]
[C0 06]
1. HEADER - FBAirlinkOpcodeSetBondMode

Use: Displays a pairing code on the device
```

Start Airlink:

```
[__1__] [__2__] [__3__] [_____4_____] [__5__] [_6]
[C0 0A] [01 00] [08 00] [10 00 00 00] [c8 00] [01]
1. HEADER - FBAirlinkOpcodeStartAirlink
2. ?
3. Version number from iOS app ?
4. ?
5. ?
6. ?

Use: Sent to the device to start Airlink handshake
```

Dump:

```
[__1__] [_2]
[C0 10] [XX]
1. HEADER - FBAirlinkOpcodeDump
2. dump type, either "03" for micro dump or "0D" for mega dump

Use: Retrieving data from a device
```

User Activity:

```
[__1__] [____2___]
[C0 12] [XX XX XX]
1. HEADER - FBAirlinkOpcodeUserActivity
2. code hints at length of 5 bytes total

Use:
```

Echo:

```
[__1__] [____2___]
[C0 13] [XX XX XX]
1. HEADER - FBAirlinkOpcodeEcho
2. code hints at length of 5 bytes total

Use:
```

Initiate Airlink:

```
[__1__] [_2] [_3] [_4] [________5________]
[C0 14] [0C] [XX] [XX] [XX XX XX XX XX XX]
1. HEADER - FBAirlinkOpcodeInitAirlink
2. total length of packet
3. Airlink version - major
4. Airlink version - minor
5. MAC address

Use: Get device info and initiate an authentication handshake
```

Transfer to Host - Single Block Response: (incomplete)

```
[__1__]
[C0 40]
1. HEADER - FBAirlinkOpcodeXFR2HOSTSingleBlockResponse

Use:
```

Transfer to Host - Stream Starting:

```
[__1__]  [_2]
[C0 41] [XX]
1. HEADER - FBAirlinkOpcodeXFR2HOSTStreamStarting
2. dump type

Use: Signal the start of streaming data from the device
```

Transfer to Host - Stream Finished:

```
[__1__] [_2] [__3__] [_____4_____]
[C0 42] [XX] [XX XX] [XX XX XX XX]
1. HEADER - FBAirlinkOpcodeXFR2HOSTStreamFinished
2. dump type
3. unknown
4. data length

Use: Signal the end of streaming data from the device
```

Auth Start:

```
[__1__] [_____2_____] [_____3_____]
[C0 50] [XX XX XX XX] [XX XX XX XX]
1. HEADER - FBAirlinkOpcodeClientAuthStart
2. random number
3. nonce

Use: Starts the authentication handshake, sent to device
```

Auth Challenge:

```
[__1__] [___________2___________] [_____3_____]
[C0 51] [XX XX XX XX XX XX XX XX] [XX XX XX XX]
1. HEADER - FBAirlinkOpcodeClientAuthChallenge
2. authentication challenge
3. challenge number (this increments each time a challenge is made)

Use: response to Auth Start, this is sent from the device
```

Auth Challenge Response: (incomplete)

```
[__1__]
[C0 52]
1. HEADER - FBAirlinkOpcodeClientAuthChallengeResponse

Use: This should authenticate the handshake, this is sent to the device
```

####Encryption Based on some preliminary analysis of the fitbit mobile app, they seem to be using LibTomCrypt for encryption on transmission of data over bluetooth. Based on the disassembled code from the Fitbit mobile app, both AES and XTEA encryption are used depending on what type of device.

Based on how the disassembled code, I believe a key is generated when pairing and this is used to decrypt data on the device (based on MAC address of the dongle?).

####Data Dumps There are two types of “dumps” that are able to be requested from the fitbit dongle:

1. Micro
2. Mega

Presumably, “Micro” dumps are for small intervals of times, and “Mega” dumps are for transferring data from a larger time interval. To retrieve the data from the device, send an Airlink Dump Opcode packet that describes the type of dump desired. Then the device will respond with a `XFR2HOSTStreamStarting` packet tagged with the dump type. The fitbit dongle then seems to transmit multiple packets before concluding the transmission with a `XFR2HOSTStreamFinished` packet describing the data just transmitted.

Micro Dump Sample:

```
30 02 00 00 01 00 E2 19   00 00 72 7A 2C 2B 28 05   0....... ..rz....
F2 E0 18 02 0F 3B E3 6B   02 8E 3F F5 8B 8D B9 7A   .......k .......z
F1 CB 58 0F 17 2E BE EC   4D B5 E9 59 62 78 56 64   ..X..... M..YbxVd
37 FB FA 79 2B 22 90 2B   F6 F0 E0 DE DD E6 2F 2F   7..y.... ........
5A 6B 41 F7 1B 97 F9 5F   5F A3 CB 22 14 2C 2C 23   ZkA..... ........
0D A8 11 39 9F 45 F3 89   23 79 B2 63 48 04 63 8C   ...9.E.. .y.cH.c.
0D C5 0C 01 56 F2 8B 1C   D1 B1 87 F0 6E BB 91 1B   ....V... ....n...
F2 9B FA 75 0B D1 2A 7B   6E 00 00                  ...u.... n..
```

Mega Dump Sample:

```
26 02 00 00 01 00 D6 19   00 00 72 7A 2C 2B 28 05   ........ ..rz....
EB E1 BC 4F E4 29 D0 88   87 89 0B A6 83 4F 29 53   ...O.... .....O.S
63 AA 6C A0 67 D0 26 91   44 0D 45 38 20 36 82 1B   c.l.g... D.E8.6..
7F E0 B6 37 B0 E2 3E 65   0A CF 16 5A 33 15 AB ED   ...7...e ...Z3...
38 2A 70 7D 1B A9 CB 30   AF 3D 6E 80 19 3D FD ED   8.p....0 ..n.....
8C A5 4A 31 AA 9C 8C 93   D5 D8 64 58 78 FC DB 78   ..J1.... ..dXx..x
E5 1E 19 BF 2F 27 F0 6A   E0 0D E5 53 6B 6E A9 10   .......j ...Skn..
2D D4 7C 7A BB 3B DD DC   CB A3 A2 AB 4A 4B EE 91   ...z.... ....JK..
67 44 A1 AE 27 68 4A 11   3A BA 52 A9 09 1A BF 8E   gD...hJ. ..R.....
4E 7B 8A C8 A9 B3 A8 33   85 1D 40 2A CE D7 00 0C   N......3 ........
53 4D 9F FC 0A 35 D2 26   33 E9 FB B2 6D 5F 74 4C   SM...5.. 3...m.tL
16 4C 36 7B BC 05 5E 45   83 73 A2 AD CE 6E 58 3A   .L6....E .s...nX.
67 06 BD B7 D6 6C E2 4D   B6 7A 5D 00 B7 E3 92 17   g....l.M .z......
8B 73 5B D8 A3 69 6C E0   82 0C DE 60 4E 61 54 5B   .s...il. ....NaT.
B5 AB 16 BC 40 0A 88 2E   14 49 A0 BE DE 0D 54 8A   ........ .I....T.
EB 9F FF D2 E8 D0 A6 44   6F 15 9B B6 B8 A7 43 28   .......D o.....C.
DB 00 00                                           ...
```

In both types of dumps, it seems that the actual contents of the message starts after the 16th byte. The formatting of these 16 bytes seems to hold to this formatting:

```
[_1] [_2] [_3] [_4] [_5] [_6] [_____7_____] [________8________]
[XX] [XX] [XX] [XX] [XX] [XX] [XX XX XX XX] [XX XX XX XX XX XX]
1. Header ? - 26 for mega, 30 for micro - this seems specific to type o
2. 02 - always ?
3. 00 - always ?
4. 00 - always ?
5. 01 - always ?
6. 00 - always ?
7. packet counter
8. device identifier
```

Following this contains what is assumed to be encrypted data. Based on some research of previous implementations of this protocol the data used to be clear, but now it seems to be encrypted based on device type. As of this post I haven’t

worked out where the decryption happens; it could be happening in the app or it could be done on the server. In addition, the micro dumps always seem to end in the following 3 bytes `6E 00 00`.

###Continuing Research At this point I've reach the limit of what I can research without additional hardware and debugging tools. I'm not in a position to actively pursue this research at this point in time. I would really like to have an app that could directly talk with the fitbit dongle without having to rely on the data provided from the fitbit web APIs. There are a few things left to research and implement:

1. data encryption decoding
2. parsing format of data coming off the device
3. mapping the remaining device commands
4. firmware updating

I think most of the challenging aspect of this process is working out the encryption/decryption process. Without a jailbroken device to debug the real app and step through the sync process it won't be feasible to work this out from just static analysis alone.

###Additional Information Sources: * Security Analysis of Wearable Fitness Devices (Fitbit) - MIT Research Paper based on the android app * benallard/libfitbit * benallard/galileo * Testing megadump from fitbit flex * openyou/libfitbit compatible with "fitbit one"? * cmwdotme/fitbitfun - if you want to build a quick app to interact, use this as a template * mrquincle/fitbit-fatbat * hiptopjones/fitbit * galileo mailing list dump analysis

---