Ubertooth

Great Scott Gadgets

Jul 18, 2022

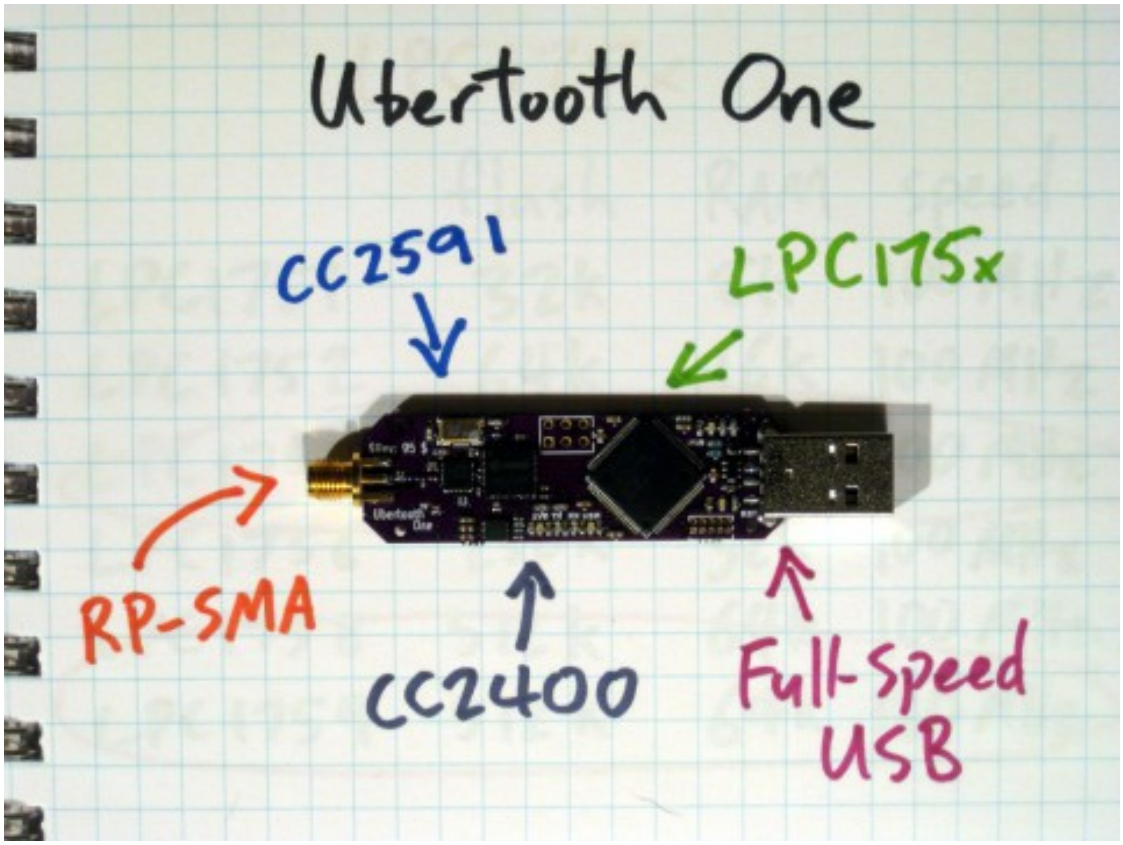
USER DOCUMENTATION

CHAPTER

ONE

UBERTOOTH ONE

Ubertooth One is the hardware platform of Project Ubertooth. It supersedes Ubertooth Zero and is currently the preferred platform.



1.1 Architecture

- RP-SMA RF connector: connects to test equipment, antenna, or dummy load.
- CC2591 RF front end.
- CC2400 wireless transceiver.
- LPC175x ARM Cortex-M3 microcontroller with Full-Speed USB 2.0.
- USB A plug: connects to host computer running Kismet or other host code.

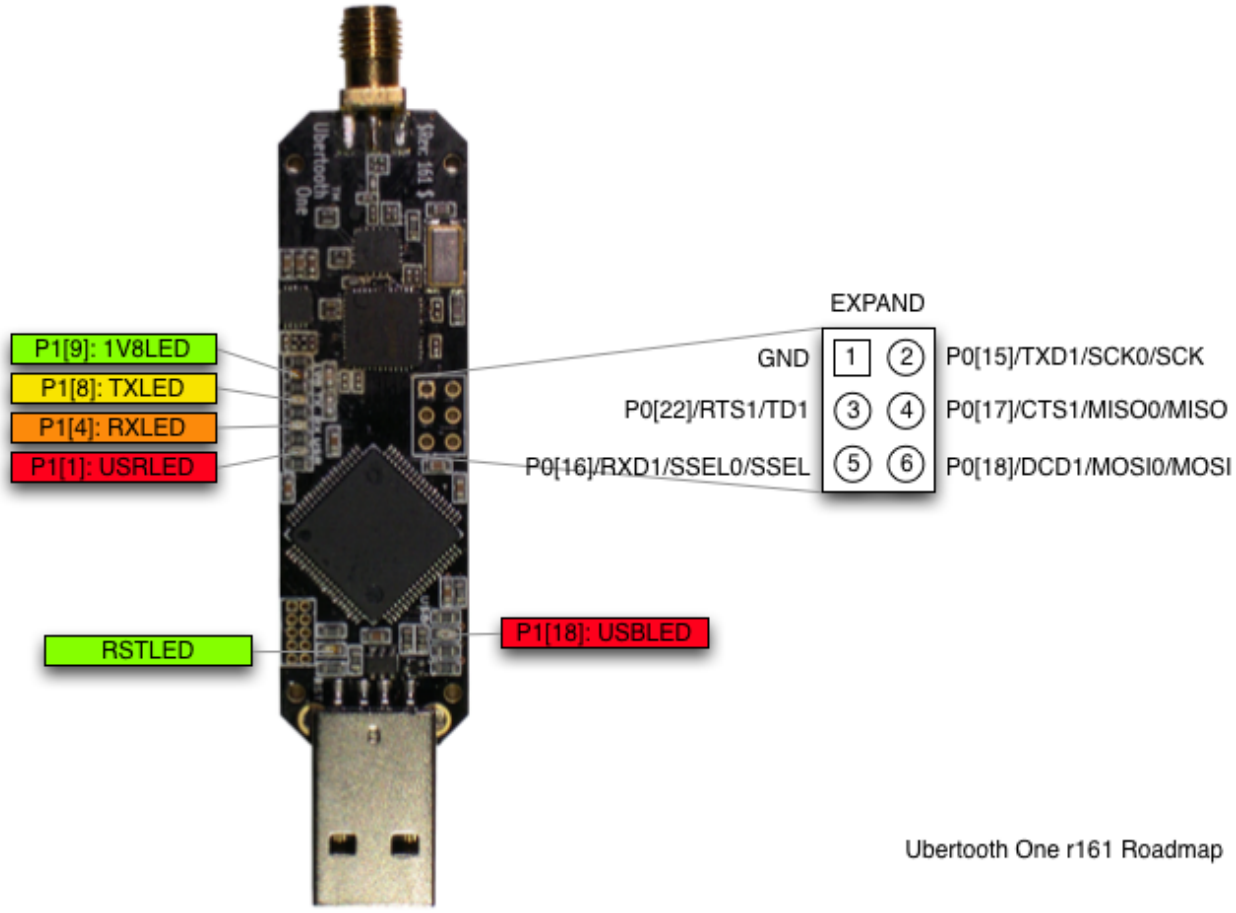
1.2 Features

- 2.4 GHz transmit and receive.
- Transmit power and receive sensitivity comparable to a Class 1 Bluetooth device.
- Standard Cortex Debug Connector (10-pin 50-mil JTAG).
- In-System Programming (ISP) serial connector.
- Expansion connector: intended for inter-Ubertooth communication or other future uses.
- Six indicator LEDs.

1.3 Design

Ubertooth One was designed in KiCad, an open source electronic design automation software package, with surface mount components suitable for reflow.

1.3.1 Pins and LEDs



This diagram shows the location of LEDs and the pins of the expansion connector.

LED guide:

- RST: indicates that the LPC175x is powered on. This should always be on during operation except during a full reset of the LPC175x (e.g., while entering ISP mode).
- 1V8: indicates that the CC2400 is being supplied with 1.8 V. Control of this supply depends on firmware. 1V8 power is required to activate the crystal oscillator which is required to activate USB.
- USB: indicates that USB has passed enumeration and configuration.
- TX: Control of this LED depends on firmware. It typically indicates radio transmission.
- RX: Control of this LED depends on firmware. It typically indicates radio reception.
- USR: Control of this LED depends on firmware.

The TX, RX, and USR LEDs blink in a distinctive chasing pattern when the bootloader is ready to accept USB DFU commands.

1.3.2 Power Usage

These measurements were taken using Ubertooth One running firmware revision 2014-02-R2. Zero_Chaos took these measurements using a Centech CT-USB-PW, available via eBay and others.

Command	Description	Power draw (amps)
	Idle	0.09A
`ubertooth-dump`	Receive	0.13A
`ubertooth-util -t`	Transmit	0.22A
`ubertooth-dfu --write`	Firmware upgrade	0.10A

1.3.3 Demonstration

Michael Ossmann presented Project Ubertooth: Building a Better Bluetooth Adapter at ShmooCon 2011.

CHAPTER

TWO

BUILD GUIDE

2.1 Release 2020-12-R1

for Release 2018-12-R1 see here

2.1.1 Prerequisites

There are some prerequisites that need to be installed before building libbtbb and the Ubertooth tools. Many of these are available from your operating system's package repositories, for example:

Debian 10 / Ubuntu 20.04 / Kali

```
sudo apt install cmake libusb-1.0-0-dev make gcc g++ libbluetooth-dev wget \
pkg-config python3-numpy python3-qtpy python3-distutils python3-setuptools
```

Fedora 33

```
sudo dnf install libusb1-devel make gcc gcc-c++ cmake wget tar bluez-libs-devel
echo /usr/local/lib | sudo tee /etc/ld.so.conf.d/libc.conf
```

RHEL 8

```
sudo subscription-manager repos --enable=codeready-builder-for-rhel-8-x86_64-
↪rpms
sudo dnf install libusb1-devel make gcc gcc-c++ wget tar bluez-libs-devel␣
↪python36 python36-devel
echo /usr/local/lib | sudo tee /etc/ld.so.conf.d/libc.conf
# Note: you will also need to install CMake 3.12.0 or greater. See https://
↪cmake.org/install/
```

CentOS 8

```
sudo dnf config-manager --enable powertools
sudo dnf install libusb1-devel make gcc gcc-c++ wget tar bluez-libs-devel␣
↪python36 python36-devel
echo /usr/local/lib | sudo tee /etc/ld.so.conf.d/libc.conf
# Note: you will also need to install CMake 3.12.0 or greater. See https://
↪cmake.org/install/
```

Mac OS X users can use either MacPorts or Homebrew to install the required packages:

```
brew install libusb wget cmake pkg-config
or
sudo port install libusb wget cmake python38 py38-numpy py38-qtpy
```

FreeBSD users can install the host tools and library directly from the ports and package system:

```
sudo pkg install ubertooth
```

2.1.2 libbtbb

Next the Bluetooth baseband library (libbtbb) needs to be built for the Ubertooth tools to decode Bluetooth packets:

```
wget https://github.com/greatscottgadgets/libbtbb/archive/2020-12-R1.tar.gz -O␣
↪libbtbb-2020-12-R1.tar.gz
tar -xf libbtbb-2020-12-R1.tar.gz
cd libbtbb-2020-12-R1
mkdir build
cd build
cmake ..
make
sudo make install
sudo ldconfig
```

2.1.3 Ubertooth Tools

The Ubertooth repository contains host code for sniffing Bluetooth packets, configuring the Ubertooth and updating firmware. All three are built and installed by default using the following method:

```
wget https://github.com/greatscottgadgets/ubertooth/releases/download/2020-12-
↪R1/ubertooth-2020-12-R1.tar.xz
tar -xf ubertooth-2020-12-R1.tar.xz
cd ubertooth-2020-12-R1/host
mkdir build
cd build
cmake ..
make
```

(continues on next page)

(continued from previous page)

```
sudo make install
sudo ldconfig
```

2.1.4 Wireshark plugins

Users of Wireshark version 2.2+ do not need to build any plugins at all and may skip this section (see this comment). This includes users of Debian 10+, Ubuntu 20.04+, Fedora 33+, RHEL 8.3+, and most other Linux distributions. You can check your version by clicking on Help –> About Wireshark.

Wireshark version 1.12 and newer includes the Ubertooth BLE plugin by default. It is also possible to capture BLE from Ubertooth directly into Wireshark with a little work.

The Wireshark BTBB and BR/EDR plugins allow Bluetooth baseband traffic that has been captured using Kismet to be analysed and disected within the Wireshark GUI. They are built separately from the rest of the Ubertooth and libbtbb software.

The directory passed to cmake as `MAKE_INSTALL_LIBDIR` varies from system to system, but it should be the location of existing Wireshark plugins, such as `asn1.so` and `ethercat.so`. On macOS this is likely `/Applications/Wireshark.app/Contents/PlugIns/wireshark/`.

```
sudo apt-get install wireshark wireshark-dev libwireshark-dev cmake
cd libbtbb-2020-12-R1/wireshark/plugins/btbb
mkdir build
cd build
cmake -DCMAKE_INSTALL_LIBDIR=/usr/lib/x86_64-linux-gnu/wireshark/libwireshark3/
↪plugins ..
make
sudo make install
```

Then repeat for the BT BR/EDR plugin:

```
sudo apt-get install wireshark wireshark-dev libwireshark-dev cmake
cd libbtbb-2020-12-R1/wireshark/plugins/btbredr
mkdir build
cd build
cmake -DCMAKE_INSTALL_LIBDIR=/usr/lib/x86_64-linux-gnu/wireshark/libwireshark3/
↪plugins ..
make
sudo make install
```

2.2 Third Party Software

There are a number of pieces of third party software that support the Ubertooth. Some support Ubertooth out of the box, while others require plugins to be built.

2.3 Firmware

This completes the install of the Ubertooth tools, the next step is to look at the getting started guide. You should always update the firmware on the Ubertooth device to match the software release version that you are using.

CHAPTER

THREE

FAQ

Question: What is the latest release?

Answer: The latest release is 2018-08-R1: the DEFCON release, but you should check this repo's releases for the most up to date information. See the Build Guide for instructions on how to download and build the software.

Question: I run Windows or Mac OS X. What's the best way to use Ubertooth?

Answer: The best way to use Ubertooth is from a native Linux install. If you don't normally run Linux, we recommend you boot Linux from USB using a distro such as Kali or Pentoo.

It **may** be possible to use Ubertooth from within a virtual machine. However, people have reported issues with USB throughput and requests timing out. If you're a newbie, booting directly into Linux is the recommended method.

Question: How do I update firmware?

Answer: Refer to the wiki page Firmware.

Question: Can I use my Ubertooth with a VM?

Answer: Yes. Many people have reported successfully using their Ubertooth with host tools installed in a virtual machine. There are, as always, some things to be aware of when attempting this.

- The Ubertooth uses USB2.0, which requires the VirtualBox extension pack to be downloaded and installed.
- The Ubertooth also uses different vendor and product identifiers when in normal operation and firmware upload mode, so rules for both of these will need to be added to the virtual machine for full operation.

Other virtual machine environments may differ and therefore present different issues. If you successfully use the Ubertooth with a different virtualisation system, please let us know so that we can help future users.

Question: What can the Ubertooth capture?

Answer: The Ubertooth is able to capture and demodulate signals in the 2.4GHz ISM band with a bandwidth of 1MHz using a modulation scheme of Frequency Shift Keying or related methods.

This includes, but is not limited to:

- Bluetooth Basic Rate packets
- Bluetooth Low Energy (Bluetooth Smart)

The following may be possible:

- 802.11 FHSS (1MBit)
- Some proprietary 2.4GHz wireless devices

Question: Can I listen to my phone calls?

Answer: Not yet, this will require full frequency hopping support and possibly require breaking encryption by sniffing the connection process. It's on the todo list, but there's a lot of hard work to go before we get there. If you're interested in making it happen, you can get involved in the project.

Question: What is the max capture size of Bluetooth Low Energy (BLE) packets?

Answer: 50 bytes. Longer BLE packets are truncated to 50 bytes by the Ubertooth firmware.

CHAPTER

FOUR

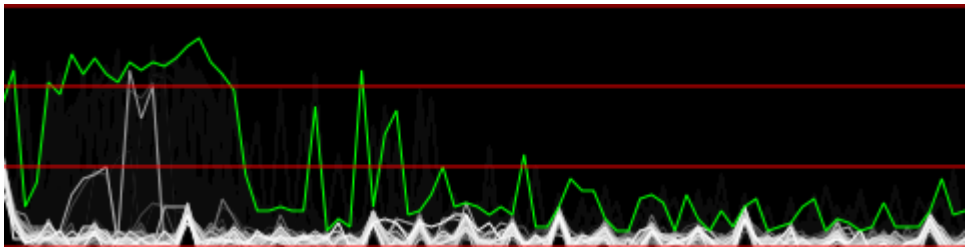
GETTING STARTED

There are three major components of Project Ubertooth:

- hardware: The hardware design of Ubertooth One is quite stable. You can build one or buy one.
- firmware: This is software that executes on the ARM processor on the Ubertooth One itself. This page assumes that you have the USB bootloader plus bluetooth_rxtx firmware installed on your board (which is typically what is installed at the time of assembly). The bluetooth_rxtx firmware is moderately stable but is likely to be enhanced as time goes on.
- host code: This is software running on a general purpose computer connected to the Ubertooth One via USB. If you have not yet built the host code, please follow the build guide.

Ubertooth One is a development platform. The true power of the device is best realized when you start writing your own software and adapting it to your needs. If you are just getting to know the board, however, it can be helpful to try out open source code that others have made available. This guide will help you get started with your Ubertooth One by introducing you to some useful host code from the Ubertooth software repository.

4.1 Spectrum Analysis



The first thing you should try with a new Ubertooth One is real-time spectrum analysis. Take a look at Jared's demonstration video for a preview.

Connect an antenna to your Ubertooth One and plug it into your computer. **Never operate your Ubertooth One without an antenna connected.** You should see the RST and 1V8 LEDs illuminate. This indicates that the LPC175x microcontroller is running (RST) and that power is being supplied to the CC2400 wireless transceiver IC (1V8). The USB LED may also light up if your computer's operating system has enumerated and configured the device (typical on Linux). Now you need some host code to tell the Ubertooth One what to do.

Download the latest Project Ubertooth file release or check out current development code from the git repository and navigate to the `host/python/specan_ui` directory. Take a look at the README file and make sure that you have installed the prerequisite software. Then execute `ubertooth-specan-ui` as described in the README and watch the 2.4 GHz activity detected by the Ubertooth One.

One possible thing that could go wrong at this point is that your operating system does not grant you permission to communicate with the USB device. Depending on your distribution and preference, this can be fixed on Linux either by adding your user account to the `usb` group or by creating a new udev rule such as:

```
$ echo 'ACTION=="add" BUS=="usb" SYSFS{idVendor}=="1d50" SYSFS{idProduct}==
↪"6002" GROUP:="plugdev" MODE:="0660"' > /etc/udev/rules.d/99-ubertooth.rules
```

A udev rules file is available in the `host/build/misc/udev/`. Copy it to `/etc/udev/rules.d/` and run the following as root:

```
# udevadm control --reload-rules
```

Make sure you are a member of the `plugdev` group or change the rule to refer to the group of your choice. After adding the udev rule, unplug the Ubertooth One, reboot or restart udevd, and plug in the Ubertooth One again.

During operation of ubertooth-specan-ui the RX LED should illuminate, and the USR LED should be dimly lit. After you finish trying out ubertooth-specan-ui reset your Ubertooth One by unplugging it and plugging it back in.

4.2 LAP Sniffing

```
mossmann@gobbo ~/ubertooth/svn/trunk/host/bluetooth_rxtx $./ubertooth-lap
rx 0 blocks of 64 bytes in 512 byte transfers
GOT PACKET on channel 39, LAP = a3101d at time stamp 821371171, clkn 131419
GOT PACKET on channel 39, LAP = 9e8b33 at time stamp 822582290, clkn 131613
GOT PACKET on channel 39, LAP = 9e8b33 at time stamp 822682288, clkn 131629
GOT PACKET on channel 39, LAP = 9e8b33 at time stamp 822782307, clkn 131645
GOT PACKET on channel 39, LAP = 9e8b33 at time stamp 823082297, clkn 131693
GOT PACKET on channel 39, LAP = 9e8b33 at time stamp 823282323, clkn 131725
GOT PACKET on channel 39, LAP = 9e8b33 at time stamp 823382318, clkn 131741
GOT PACKET on channel 39, LAP = 9e8b33 at time stamp 823482306, clkn 131757
GOT PACKET on channel 39, LAP = 9e8b33 at time stamp 823782345, clkn 131805
GOT PACKET on channel 39, LAP = 9e8b33 at time stamp 823882352, clkn 131821
GOT PACKET on channel 39, LAP = 9e8b33 at time stamp 824082345, clkn 131853
```

Bluetooth packets start with a code that is based on the Lower Address Part (LAP) of a particular Bluetooth Device Address (BD_ADDR). The BD_ADDR is a 48 bit MAC address, just like the MAC address of an Ethernet device. The LAP consists of the lower 24 bits of the BD_ADDR and is the only part of the address that is transmitted with every packet.

The most important passive Bluetooth monitoring function is simply capturing the LAP from each packet transmitted on a channel. LAP sniffing allows you to identify Bluetooth devices operating in your vicinity.

In order to sniff LAPs, you'll have to compile the tools in `host/ubertooth-tools`. These are command line programs intended to work with the bluetooth_rxtx firmware installed on your Ubertooth One. Follow the instructions in the README file in that directory to install the the prerequisite libbtbb, a library for Bluetooth baseband functions. You can install libbtbb from a file release rather than git if you prefer.

Once libbtbb is installed, just type:

```
mkdir build
cd build/
cmake ..
make
```

in the `host` directory to compile the tools there. Then make sure your Ubertooth One is plugged in and execute:

```
$ ubertooth-rx
```

You should see various random LAPs detected. Due to uncertainties in identifying Bluetooth packets without prior knowledge of an address, it is normal for this process to identify false positives. error correction should mitigate this problem, but a small number of false positives may still be seen. When you see the same LAP detected more than once, that is very likely an actual Bluetooth transmission.

Generate some Bluetooth traffic and enjoy the show. I like to use a mobile phone or other Bluetooth device to perform an inquiry (usually called "find new Bluetooth devices" or something similar) to make sure that everything is working properly. An inquiry should produce lots of packets with the LAP 0x9e8b33.

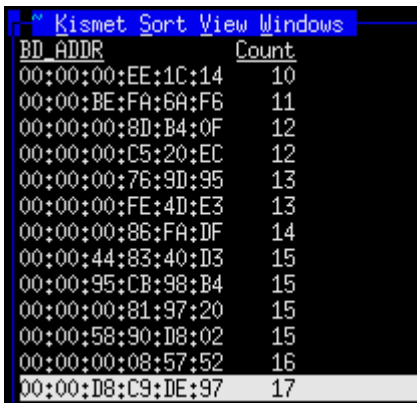
Once you have seen a LAP multiple times, you can be confident that it is a genuine Bluetooth piconet. To find the next byte of the address, the UAP, we can use:

```
$ ubertooth-rx -l [LAP]
```

In this mode ubertooth-rx only detects packets from the given piconet and uses them to determine the next byte of the address and some of the internal clock value.

For more information on this process, and the challenges involved in monitoring Bluetooth connections, please read this blog post.

4.3 Kismet



More advanced Bluetooth sniffing has been implemented in the form of a plugin for Kismet, the venerable 802.11 monitoring tool. In order to compile the Kismet-Ubertooth plugin, you will need a Kismet source tree matching the installed version. The easiest way to make this work is to uninstall any binary Kismet installation you may have installed and then download the Kismet source and follow the instructions to compile and install from the fresh source code. Once Kismet is installed, follow the instructions in `host/kismet/plugin-ubertooth/README` to install and use the plugin.

Notice that Kismet-Ubertooth identifies not only the LAP but also the 8 bit Upper Address Part (UAP) of detected devices as it is able. This is done by analyzing the timing and other characteristics of multiple packets over time. Another advantage of Kismet is that it dumps complete decoded packets to a pcapbtbb file that can be read with a Wireshark plugin that is distributed with libbtbb. Full packet decoding is only possible when the packet's UAP has been determined.

4.4 Where to Go from Here

I hope you have found this guide helpful in getting to know your Ubertooth One. If you are interested in contributing to the project, or if you need help or would just like to chat about Project Ubertooth, join the #ubertooth channel on Discord. Happy hacking!

CHAPTER

FIVE

GETTING HELP

5.1 Asking Questions

If you have questions about using Ubertooth the first place to look is the FAQ.

Many Ubertooth developers and users are available in the #ubertooth channel on Discord. Please remember that we work across many timezones and you may need to wait some time for a response to your question.

5.2 Software Bugs

We use the GitHub issue tracker to log Ubertooth bugs. If the bug has previously been reported then adding detail is often helpful. If you believe that you have found a new bug, then please create a new issue and describe the bug in as much detail as possible.

When logging issues it is helpful for us to know the following:

- Software version
- Firmware version
- Operating system and hardware (i.e. ARM, x86, etc) if known
- Which command you were using
- Expected output
- Actual output
- Any other details that may help us to reproduce the bug

CHAPTER

SIX

CAPTURING BLE IN WIRESHARK

You can capture BLE in Wireshark with standard Wireshark builds. This guide assumes Linux.

1. Run the command: `mkfifo /tmp/pipe`
2. Open Wireshark
3. Click Capture -> Options
4. Click "Manage Interfaces" button on the right side of the window
5. Click the "New" button
6. In the "Pipe" text box, type "/tmp/pipe"
7. Click Save, then click Close
8. Click "Start"

In a terminal, run `ubertooth-btle`:

```
ubertooth-btle -f -c /tmp/pipe
```

In the Wireshark window you should see packets scrolling by.

Note: If you get User encapsulation not handled: DLT=147, check your Preferences->Protocols->DLT_USER the steps you want are:

1. Click Edit -> Preferences
2. Click Protocols -> DLT_USER
3. Click Edit (Encapsulations Table)
4. Click New
5. Under DLT, select "User 0 (DLT=147)" (adjust this selection as appropriate if the error message showed a different DLT number than 147)
6. Under Payload Protocol, enter: btle
7. Click OK
8. Click OK

6.1 Capturing BLE in scapy

1. Do not use mkfifo for the filename, it will cause scapy to slow dramatically.
2. In a terminal, run ubertooth-btle:

```
ubertooth-btle -f -q /tmp/pipe
```

3. Open python and run:

```
from scapy.all import *
p = sniff(offline='/tmp/pipe')
```

p is now a list of the packets captured!

6.2 Sniffing connection data

With recent Ubertooth firmware, only advertisements are captured by default. Once you have identified the device address of the target device you would like to sniff, run:

```
ubertooth-btle -t aa:bb:cc:dd:ee:ff
```

The Ubertooth will follow connections involving this target until -t none is passed or the device is reset.

You may need to attempt connecting several times until Ubertooth is able to follow the connection successfully.

6.3 Capturing from a remote host

You can use sshdump to remotely capture packets from a Ubertooth attached to another host or virtual machine. In the Wireshark UI, this may show up as an interface named "SSH remote capture: `sshdump`" which needs to be configured first with the following "remote capture command":

```
killall ubertooth-btle; unlink /tmp/btlepipe; mkfifo /tmp/btlepipe; ubertooth-
↪btle -f -c /tmp/btlepipe &>/dev/null & cat /tmp/btlepipe
```

The "remote interface" option is ignored and can be set to any value.

6.4 Useful display filters

Only connection requests and non-zero data packets:

```
btle.data_header.length > 0 || btle.advertising_header.pdu_type == 0x05
```

Only attribute read responses, write requests, and notifications:

```
btatt.opcode in { 0x0b 0x12 0x1b }
```

CHAPTER

SEVEN

BLUETOOTH CAPTURES IN PCAP

7.1 Overview

Classic PCAP files store a sequence of packets of a single link type. Published link types are here. The pcapng format also uses these same link types and the per-packet formatting as PCAP.

Early versions of libbtbb and ubertooth saved PCAP files with the DLT_PPI format, which was expedient but is considered deprecated by the libpcap folks. Best practice is to allocate a DLT for a particular link-layer and define a pseudo-header for that DLT that precedes each packet in the file. Early versions of this article formed a place to collect such a proposal. Now that the DLTs are allocated, and this article serves to collect implementation alternatives and details.

It was possible, and somewhat consistent with DLT_BLUETOOTH_HCI_H4_WITH_PHDR, to allocate a single DLT_BLUETOOTH_LOW_LEVEL to indicate any Bluetooth capture (BR/EDR or LE). However, the preference was to have separate DLT's for BR/EDR baseband and LE link-layer (as per the terms used in the Bluetooth spec).

Since PCAP has general applicability to packet capture, it made sense to provide a vendor-neutral, generic view of Bluetooth capture. The allocated DLTs can be applied to current and future versions of both ubertooth and gr-bluetooth, and potentially other RF capture tools as well.

7.2 Aspects of Bluetooth Capture

Sometimes we want to capture and record malformed, truncated, or garbled packets along with ostensibly valid or pristine ones. We also want to capture packets when some link parameters are unknown. When operating promiscuously, the capture tool may accept noise bursts as candidate Bluetooth packets. In extreme cases, the capture tool may not even de-whiten the packets, preferring to do that as a post-processing step. It was therefore important to include some of the receiver metrics in the capture metadata so post-processing and display tools like Wireshark can easily filter out unwanted packets, or avoid redundantly checking packet integrity when it's already known.

Five areas of receiver metadata were contemplated to assist in packet classification:

1. signal and noise strength
2. flags and metrics on the validity of unprotected fields (BR sync word, LE access address)
3. error-correction on BR/EDR when subject to FEC
4. whether all or portions of the packet is de-whitened
5. packet-level error-checking already performed at capture (CRC, HEC, MIC)

Since the current ubertooth can recover BR and LE packets, but not EDR payloads, there was also a need to indicate whether the EDR data is present in the packet capture.

7.3 Capture Use Cases Summarized

The following use cases apply to both full-band and narrowband capture strategies. There are no particular restrictions imposed under full-band captures. However, for a narrowband captures (e.g. Ubertooth):

- learning implies the tool transitions from receiving a static channel, or surveying channels, to following particular hop sequence(s), and
- limited hop sequence(s) (piconets/access addresses) may be followed at a given time.

Hybrid captures are also supported by these use cases and capture formats. For example, one may configure several narrowband (e.g. Ubertooth) or partial-band (e.g. gr-bluetooth) captures, on static channels, and perform post-processing on the set of capture files (either PCAP or PCAPNG). A similar hybrid arrangement is possible to post-process several narrowband capture tools following different hop sequences.

Use cases involving encryption are TBD.

7.3.1 BR/EDR

Name	Description	Comments	Packets De-whitened?	Reference LAP	Reference UAP	HEC/CRC checking
Promiscuous capture without learning	All packets that meet the configured RF criteria (e.g. signal strength or SNR), and meet the configured criteria for access-code offenses (e.g. preamble, trailer must be valid), are decoded and stored in the capture file.	Either PCAP or PCAPNG is equally useful. Capture file may be very large. Post-processing necessary to recover UAP per LAP, and CLK per LAP/UAP, and filter packets of interest.	No	Invalid	Invalid	No
Promiscuous capture with learning	Initially operates as above, but as capture tool recovers LAPs, and the associated UAP, and CLK parameters, it is able to perform more processing per packet.	PCAP or PCAPNG may be used, but the latter is more useful since it includes a record of the capture tool's learning process. Post-processing similar to above may back-annotate, de-whiten, and integrity-check packets captured before the parameters were learned. Essentially, the post-processing mentioned above is split between capture-time and post-capture-time.	Initially no, later yes	Initially invalid, later known	Initially invalid, later known	Initially no, later optionally performed
Capture of targeted LAPs, with and without learning	Only captures packets with an access code that includes	This is simply a stricter version of the promiscuous captures,	Initially no, later yes with learning	Valid	Initially invalid, later known with learning	Initially optionally performed with learning

7.3.2 LE

Name	Description	Comments	Packets De-whitened?	Reference AA	CRC checking
Promiscuous capture without learning	All packets that meet the configured RF criteria (e.g. signal strength or SNR), and meet the configured criteria for access address offenses (e.g. preamble must be valid, access address must be well-formed), are stored in the capture file. The captured packets are optionally de-whitened.	Either PCAP or PCAPNG is equally useful. Capture file may be very large. Post-processing necessary to recover connection parameters, and filter packets of interest.	Optional	Valid for Advertising channels only	Optionally performed for Advertising channels only
Promiscuous capture with learning	Initially operates as above, except access addresses are learned by the capture tool, e.g. by accumulating candidates on Data channels or processing CONNECT_REQ PDUs on Advertising channels.	Access addresses learned by profiling Data channels cannot be CRC-checked, because the CRCInit parameter is unknown.	Yes (but optional)	Initially valid for Advertising channels only, but later valid for Data channels as access addresses are learned	Optionally performed for Advertising channels or Data channels where CRCInit is known for the access address
Targeted capture	Specific BD_ADDRs are selected, and the associated access addresses are whitelisted for capture when data from a CONNECT_REQ PDU involving those BD_ADDRs is processed	The contents of the CONNECT_REQ PDU may be found by capturing Advertising channels or configured into the capture tool directly.	Yes (but optional)	Valid	Optionally performed

7.4 Session Meta Information (Proposed)

Often there is meta-information that is recovered during the Bluetooth air capture, during post-processing, or provided out-of-band. Here we enumerate PCAPNG options for use within the interface description block for Bluetooth captures.

Some of these session-oriented data have time-windows that bound their applicability. A timestamp pair is used to define such a window. Timestamps are stored in same precision as indicated in ts_resol field of the capture interface. When both timestamps are equal (e.g. both zero), the meta-datum applies for the entire capture session.

PCAPNG options with MSB set are available for local use. We simply state that the interface options enumerated below are local to the DLTs allocated for Bluetooth RF captures. The most-significant byte for all interface option codes below is 0xd3, which was selected as unlikely to conflict with other local interface options that might be in use in PCAPNG generally. The ranges of least-significant bytes allocated below to option codes are: general Bluetooth is 0x0-0x3f, BR/EDR is 0x40-0x7f, and LE is 0x80-0xbf, with 0xc0-0xff reserved.

7.4.1 BREDR_BD_ADDR

This record provides Bluetooth device addresses (BD_ADDRs) that may be present in the packet capture. BD_ADDRs are useful in post-processing or display tools to provide unique identification of the devices involved in piconet communication.

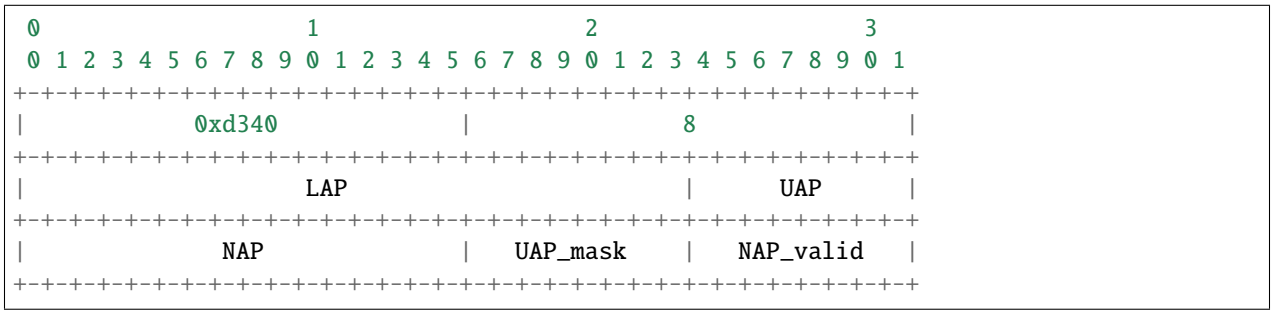
Device addresses may be recovered by the capture tool or provided by the user as a parameter to the capture session. In either case, this BREDR_BD_ADDR record may appear in the PCAPNG capture file.

When recovered by the capture tool, the UAP may be partly recovered by determining the channel hop sequence. Only the 4 least-significant bits of the UAP are used in hop-sequence determination. UAPs are also used in the BR/EDR Header Error Check, and payload Cyclic Redundancy Check generation, and may be recovered by accumulating candidates from the captured Bluetooth packets. In these cases, the UAP recorded may be masked to indicate which bits are known with certainty.

The UAP and NAP are available in the clear as fields within the FHS Packet. When captured directly, or when provided as a capture session parameter, the UAP and NAP may be recorded with certainty (all mask bits set).

The capture tool may store multiple records for the same BD_ADDR, as long as subsequent records indicate more certainty in the known UAP bits or add a known NAP. This sort of situation might occur if a capture tool starts out without knowing any UAP bits, then determines some UAP bits from hop-sequence following, more UAP bits from HEC and CRC prediction, and finally the full BD_ADDR contents after capturing an applicable FHS packet.

Option Structure



Description

The option code and length are expressed in the native endianness used by PCAPNG. All multi-octet fields defined below are expressed in little-endian format.

The **LAP** field is the Lower Address Part of the the Bluetooth device address, as per Bluetooth spec Volume 2, Part B, Section 1.2.

The **UAP** field is the Upper Address Part of the the Bluetooth device address, as per Bluetooth spec Volume 2, Part B, Section 1.2.

The **NAP** field is the Network Address Part of the the Bluetooth device address, as per Bluetooth spec Volume 2, Part B, Section 1.2.

The **UAP_mask** field has its bits set to indicate which bits of the UAP are known with certainty.

The **NAP_valid** field is a flag in the least-significant bit that indicates whether the NAP field is populated with valid data. All other bits of this field are reserved and must be zero.

C Structure

```
typedef struct _brder_bdaddr {
        uint8_t  LAP[3];
        uint8_t  UAP;
        uint16_t NAP;
        uint8_t  UAP_mask;
        uint8_t  NAP_valid;
} bredr_bdaddr;
```

7.4.2 BREDR_CLK

This record provides Bluetooth Clock alignment information. The alignment timestamp used in this record is the same precision as the PCAPNG interface header indicates.

Some capture tools estimate the master device clock by inspecting packets and building confidence in the estimate. This record provides a mask that has bits set for known master clock bits. This distinguishes known bits from unknown bits as the master clock estimate improves. Consequently, multiple BREDR_CLK records may appear in the PCAPNG capture file for the same LAP/UAP, provided that subsequent entries offer a better estimate of the device clock.

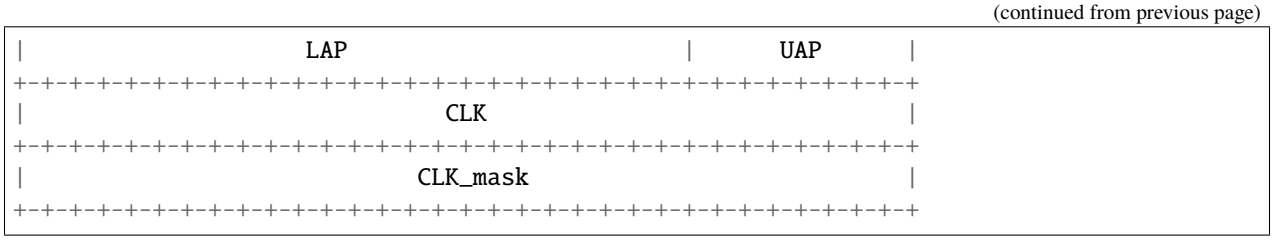
The information record may be formed as a result of capturing a Bluetooth FHS packet, in which case the CLK_mask should indicate all CLK bits are known.

Option Structure

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            0xd341             |              20               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                      alignment timestamp                      |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

(continues on next page)

(continued from previous page)



Description

The option code and length are expressed in the native endianness used by PCAPNG. All multi-octet fields defined below are expressed in little-endian format.

The **Alignment Timestamp** is a PCAPNG-resolution timestamp that serves as a reference point for **CLK** associated with the Bluetooth master device referenced by **LAP** and **UAP**.

The **LAP** field is the Lower Address Part of the the Bluetooth device address, as per Bluetooth spec Volume 2, Part B, Section 1.2.

The **UAP** field is the Upper Address Part of the the Bluetooth device address, as per Bluetooth spec Volume 2, Part B, Section 1.2.

The **CLK** field is the native clock of the Bluetooth device, with bits 0-1 and bits 28-31 always zero.

The **CLK_mask** field determines which bits of **CLK** are valid, with bits 0-1 and bits 28-31 always zero.

C Structure

```
typedef struct _brder_bdaddr {
        uint64_t ns;
        uint8_t  LAP[3];
        uint8_t  UAP;
        uint32_t CLK;
        uint32_t CLK_mask;
} bredr_bdaddr;
```

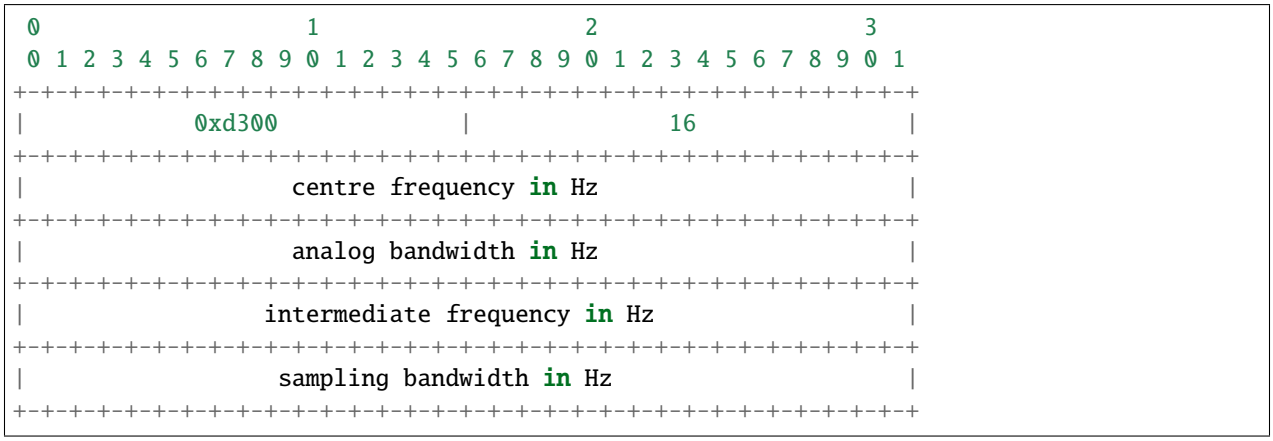
7.4.3 BT_WIDEBAND_RF_INFO

Some capture tools, e.g. gr-bluetooth, allow for intentional aliasing such that multiple Bluetooth channels appear as superimposed images within a relatively narrow baseband.

Here we define a generic wideband RF information structure so aliasing conditions may be recorded in the PCAPNG capture file.

A post-processing or display tool might use this information to indicate the set of possible RF channels ascribed to each captured packet. BT_WIDEBAND_RF_INFO only applies to packets captured under the applicable interface, where the packet's Flags field indicates the RF channel was subject to aliasing.

Option Structure



Description

The option code and length are expressed in the native endianness used by PCAPNG. All multi-octet fields defined below are expressed in little-endian format.

The **Centre Frequency** field determines the centre of the RF capture, in Hz.

The **Analog Bandwidth** field determines the passband width of the analog section employed in the capture tool. It is measured from band centre to the band edge, in Hz.

The **Intermediate Frequency** field determines the intermediate carrier frequency used in the analog receiver, relative to the **Centre Frequency**, in Hz.

The **Sampling Bandwidth** field determines the digital sampling bandwidth employed in the capture tool, in Hz.

C Structure

```
typedef struct _bt_wideband_rf_info {
        uint32_t centre_freq_hz;
        uint32_t analog_bw_hz;
        int32_t  intermediate_freq_hz;
        uint32_t sampling_bw_hz;
} bt_wideband_rf_info;
```

7.4.4 LE_LL_CONNECTION_INFO

This record provides context for a BTLE connection so that a post-processor or display tool may perform a more in-depth packet analysis. The following fields may be applied:

- InitA, the initiator's public or random device address, may be used to connect packets with a device.
- AdvA, the advertiser's public or random device address, may be used to connect packets with a device.
- AA, the access address, connects a given LE packet to the rest of the data in this record (since all LE packets contain an AA field).
- CRCInit, the 24-bit LFSR initial value, may be used to verify per-packet CRC integrity.
- WinSize, WinOffset, Interval, and Latency may be used to verify adherence to RF transmission rules.

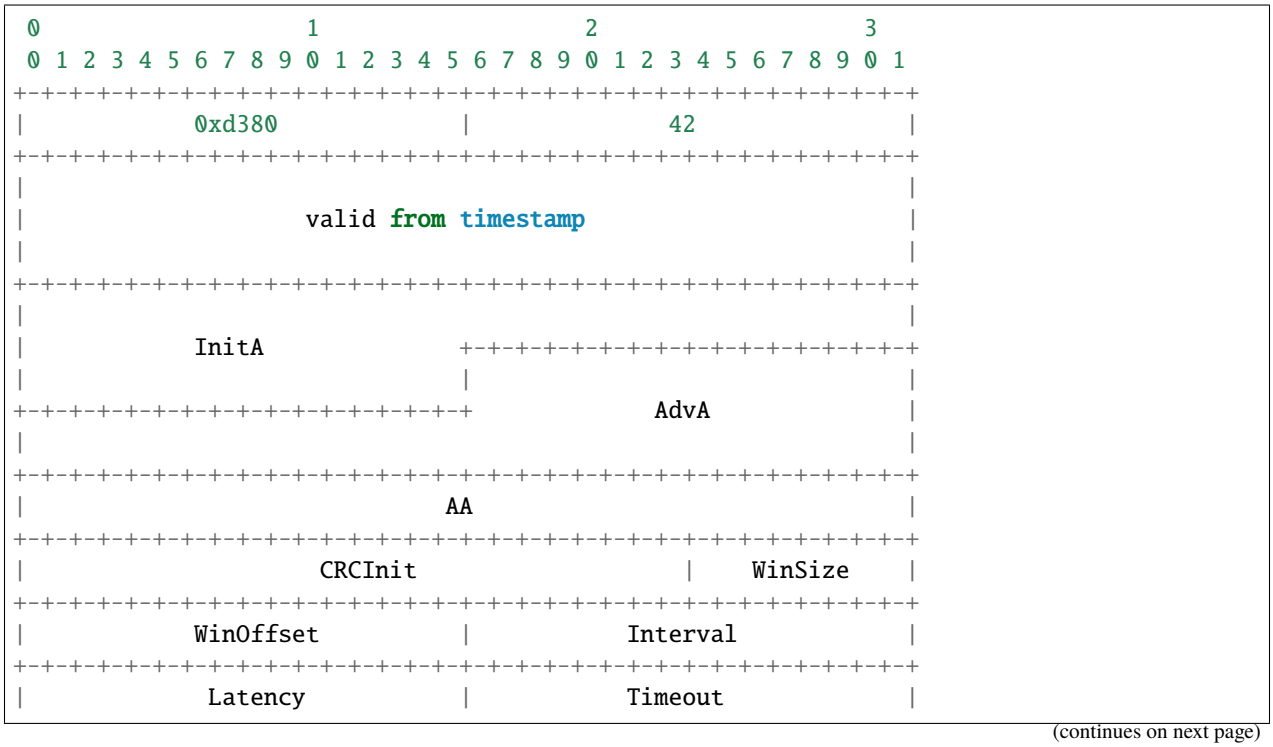
- Timeout may be used to infer connection loss when packets are absent.
- ChM, the allowable RF channel map, and Hop, may be used to verify the RF hop sequence.

The format of this record matches the CONNECT_REQ PDU used in the LE link layer. It is anticipated records of this nature would accrue in the capture file as follows:

1. when a CONNECT_REQ PDU is captured, a record is stored with the PDU contents, and the capture tool considers the values current for the indicated AA.
2. when a LL_CONNECTION_UPDATE_REQ PDU is captured after a CONNECT_REQ PDU, for the same AA:
    1. a new record is created, updating the WinSize, WinOffset, Interval, Latency, and Timeout fields.
    2. the valid-from timestamp is determined by the Instant parameter of the LL_CONNECTION_UPDATE_REQ PDU.
    3. the other parameters in this record are populated with those values already considered current.
    4. at the indicated instant, capture tool considers the updated values current for the indicated AA.
3. when a LL_CHANNEL_MAP_REQ PDU is captured after a CONNECT_REQ PDU, for the same AA:
    1. a new record is created, updating the ChM field.
    2. the valid-from timestamp is determined by the Instant parameter of the LL_CHANNEL_MAP_REQ PDU.
    3. the other parameters in this record are populated with those values already considered current.
    4. at the indicated instant, capture tool considers the updated ChM current for the indicated AA.

It is noted that an LE packet capture may contain all the information necessary to synthesize these records. Therefore, these records may be created during capture or afterwards, as a post-processing step. In the latter case, a classic PCAP file may be converted to PCAPNG.

Option Structure



(continues on next page)

(continued from previous page)

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                                       |
|    ChM         +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                |   Hop+SCA     |              pad                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Description

The definition of the fields are found in the Bluetooth specification Volume 6, Part B, Sections 2.3.3.1, 2.4.2.1, and 2.4.2.2.

C Structure

```
typedef struct _le_ll_connection_info {
        uint64_t valid_from_ts;
        uint8_t  InitA[6];
        uint8_t  AdvA[6];
        uint32_t AA;
        uint8_t  CRCInit[3];
        uint8_t  WinSize;
        uint16_t WinOffset;
        uint16_t Interval;
        uint16_t Latency;
        uint16_t Timeout;
        uint8_t  ChM[5];
        uint8_t  HopAndSCA;
} le_ll_connection_info;
```

7.4.5 Under Development

- BR/EDR
  - link-key info: link-key (16 bytes) + 2 BR_ADDR (12 bytes) + 2 timestamps (16 bytes)
  - E0 encryption info: BR_ADDR (6 bytes) + 2 timestamps (16 bytes) + keylen (1 bytes) + key (N bytes)
  - AES-CCM session info: session key (16 bytes) + nonce (13 bytes) + BR_ADDR (6 bytes) + 2 timestamps (16 bytes)
- LE
  - AES-CCM session info: session key (16 bytes) + nonce (13 bytes) + AA (4 bytes) + 2 timestamps (16 bytes)

7.5 Allocated DLTs

Common to the following pseudoheaders:

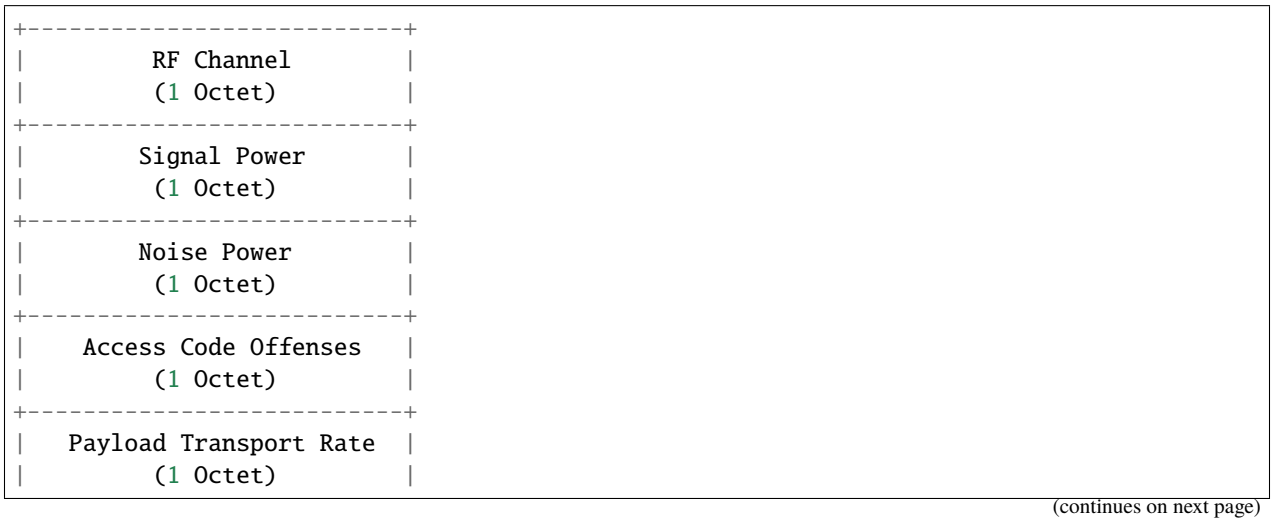
- mandatory fields:
  - a rf_channel field, although channels differ between BR/EDR and LE.
  - a flags field that indicates which optional fields are present, and other boolean metadata.
- optional fields:
  - signal power and noise power, probably used by more sophisticated capture tools.

The remaining fields are specific to the BR/EDR and LE capture process, including the packet quality indicators mentioned above.

7.5.1 LINKTYPE_BLUETOOTH_BREDR_BB

- only covers BR/EDR baseband packets, Bluetooth spec Vol.2 Part B.
- each packet includes a packed pseudoheader described below, optionally followed by the decoded BR/EDR baseband PAYLOAD.
  - here decoded is used in the same sense as Bluetooth spec Vol 2 Part B Section 7.
  - BR/EDR PAYLOAD formats are described in Bluetooth spec Vol 2 Part B Section 6.1 and 6.6.
- the packet SYNC WORD (sec 6.3) is not stored, rather the LAP is recovered and stored in the pseudoheader.
- the packet HEADER (sec 6.4) is decoded and stored in the pseudoheader.
- further downstream receiver processing may or may not be performed on the PAYLOAD.
  - refer to figure 7.2 of Bluetooth spec Vol 2 Part B.
  - de-whitening, CRC checking, decryption, and MIC checking, are optional.
  - flags in the pseudo-header indicate which of these have been performed.
  - none of these optional processing steps affect the length of the stored packet PAYLOAD.

Packet Structure



(continues on next page)

(continued from previous page)

```
+---------------------------+
|   Corrected Header Bits   |
|         (1 Octet)         |
+---------------------------+
|  Corrected Payload Bits   |
|         (2 Octets)        |
+---------------------------+
|    Lower Address Part     |
|         (4 Octets)        |
+---------------------------+
|       Reference LAP       |
|         (3 Octets)        |
+---------------------------+
|       Reference UAP       |
|          (1 Octet)        |
+---------------------------+
|     BT Packet Header      |
|         (4 Octets)        |
+---------------------------+
|           Flags           |
|         (2 Octets)        |
+---------------------------+
|    BR or EDR Payload      |
.                           .
.                           .
.                           .
```

Description

All multi-octet fields are expressed in little-endian format. Fields with a corresponding **Flags** bit are only considered valid when the bit is set.

The **RF Channel** field ranges 0 to 78. It reflects the value described in the Bluetooth specification Volume 2, Part A, Section 2.

The **Signal Power** and **Noise Power** fields are signed integers expressing values in dBm.

The **Access Code Offenses** field is an unsigned integer indicating the number of deviations from the valid access code that led to the packet capture. Access codes are interpreted as described in Bluetooth specification Volume 2, Part B, Section 6.3.

The **Payload Transport Rate** field represents a column of Bluetooth specification Volume 2, Part B, Section 6.5, Table 6.2, and is interpreted as two nibbles as follows.

- 0x.0 indicates the BT payload was BR and captured with GFSK demodulation
- 0x.1 indicates the BT payload was EDR and captured with PI/2-DQPSK demodulation
- 0x.2 indicates the BT payload was EDR and captured with 8DPSK demodulation
- 0x0. indicates the packet logical transport is any (link parameters unknown)
- 0x1. indicates the packet logical transport is SCO
- 0x2. indicates the packet logical transport is eSCO
- 0x3. indicates the packet logical transport is ACL

- 0x4. indicates the packet logical transport is CSB
- 0xff indicates this is an ID packet so BT Packet Header is ignored and there is no payload

All other values of the **Payload Transport Rate** field are reserved.

The **Corrected Header Bits** field is an unsigned integer indicating the number of corrected bits in the 18-bit **BT Packet Header**. The valid range is 0 to 18.

The **Corrected Payload Bits** field is a signed integer indicating the number of errored and corrected bits in the captured BT payload. Interpretation of this field corresponds to the **Payload Transport Rate**. The value ranges from 0 to 80 when the BT payload was captured at R=1/3 as per Bluetooth specification Volume 2, Part B, Section 7.4. The value ranges from -360 to +180 when the BT payload was captured at R=2/3 as per Bluetooth specification Volume 2, Part B, Section 7.5. A negative number indicates the field absolute value is the sum of the number of corrected and uncorrectable bits.

The **Lower Address Part** field is the 24-bit value recovered from the captured SYNC WORD as defined in Bluetooth specification Volume 2, Part B, Section 6.3.3. The most significant byte of this field is reserved and must be zero.

The **Reference LAP** field corresponds to the **Lower Address Part** configured into the capture tool that led to the capture of this packet.

The **Reference UAP** field corresponds to the **Upper Address Part** configured into the capture tool and corresponds to the **Reference LAP**.

The **BT Packet Header** field is the 18-bit value recovered from the packet capture, and is defined in Bluetooth specification Volume 2, Part B, Section 6.4. The most significant 14 bits are reserved and must be zero.

The **Flags** field represents packed bits defined as follows.

- 0x0001 indicates the **BT Packet Header** and **BR or EDR Payload** are de-whitened.
- 0x0002 indicates the **Signal Power** field is valid.
- 0x0004 indicates the **Noise Power** field is valid.
- 0x0008 indicates the **BR or EDR Payload** is decrypted
- 0x0010 indicates the **Reference LAP** is valid and led to this packet being captured
- 0x0020 indicates the **BR or EDR Payload** is present and follows this field
- 0x0040 indicates the **RF Channel** field is subject to aliasing
- 0x0080 indicates the **Reference UAP** field is valid for HEC and CRC checking
- 0x0100 indicates the HEC portion of the **BT Packet Header** was checked
- 0x0200 indicates the HEC portion of the **BT Packet Header** passed its check
- 0x0400 indicates the CRC portion of the **BR or EDR Payload** was checked
- 0x0800 indicates the CRC portion of the **BR or EDR Payload** passed its check
- 0x1000 indicates the MIC portion of the decrypted **BR or EDR Payload** was checked
- 0x2000 indicates the MIC portion of the decrypted **BR or EDR Payload** passed its check

All other bit positions of the **Flags** field are reserved and must be zero.

The decoded **BR or EDR Payload** optionally follows the previous fields, and is formatted as detailed in Bluetooth specification Volume 2, Part B, Section 6. The packet is decoded per Bluetooth specification Volume 2, Part B, Section 7. All multi-octet values in the **BR or EDR Payload** are always expressed in little-endian format, as is the normal Bluetooth practice.

C Structure

```
typedef struct _pcap_bluetooth_bredr_bb_header {
        uint8_t rf_channel;
        int8_t signal_power;
        int8_t noise_power;
        uint8_t access_code_offenses;
        uint8_t payload_transport_rate;
        uint8_t corrected_header_bits;
        int16_t corrected_payload_bits;
        uint32_t lap;
        uint32_t ref_lap_uap;
        uint32_t bt_header;
        uint16_t flags;
        uint8_t  br_edr_payload[0];
} pcap_bluetooth_bredr_bb_header;
```

7.5.2 LINKTYPE_BLUETOOTH_LE_LL_WITH_PHDR

- supplements DLT_BLUETOOTH_LE_LL which already exists but is not used for RF captures.
- only covers LE link layer packets, Bluetooth spec Vol.6 Part B.
- each packet includes a packed pseudoheader described below, followed by the LE link-layer packet consisting of ACCESS ADDRESS, PDU, and CRC, but excluding the PREAMBLE.
  - reference Bluetooth spec Vol.6 Part B sec 2 for formatting.
- not all receiver processing need be performed at capture time.
  - refer to figures 3.1 of Bluetooth spec Vol 6 Part B.
  - de-whitening, CRC checking, decryption, and MIC checking, are optional.
  - flags in the pseudo-header indicate which of these have been performed.
  - none of these optional processing steps affect the length of the stored packet data.

Packet Structure

```
+-------------------------+
|         RF Channel      |
|         (1 Octet)       |
+-------------------------+
|        Signal Power     |
|         (1 Octet)       |
+-------------------------+
|        Noise Power      |
|         (1 Octet)       |
+-------------------------+
|  Access Address Offenses  |
|         (1 Octet)       |
+-------------------------+
| Reference Access Address  |
```

(continues on next page)

(continued from previous page)

```
|        (4 Octets)         |
+---------------------------+
|          Flags            |
|        (2 Octets)         |
+---------------------------+
|  LE Packet (no preamble)  |
.                           .
.                           .
.                           .
```

Description

All multi-octet fields are expressed in little-endian format. Fields with a corresponding **Flags** bit are only considered valid when the bit is set.

The **RF Channel** field ranges 0 to 39. It reflects the value described in the Bluetooth specification Volume 6, Part A, Section 2.

The **Signal Power** and **Noise Power** fields are signed integers expressing values in dBm.

The **Access Address Offenses** field is an unsigned integer indicating the number of deviations from the valid access address that led to the packet capture. Access addresses are interpreted as described in Bluetooth specification Volume 6, Part B, Section 2.1.2.

The **Reference Access Address** field corresponds to the Access Address configured into the capture tool that led to the capture of this packet.

The **Flags** field represents packed bits defined as follows.

- 0x0001 indicates the **LE Packet** is de-whitened.
- 0x0002 indicates the **Signal Power** field is valid.
- 0x0004 indicates the **Noise Power** field is valid.
- 0x0008 indicates the **LE Packet** is decrypted.
- 0x0010 indicates the **Reference Access Address** is valid and led to this packet being captured.
- 0x0020 indicates the **Access Address Offenses** field contains valid data.
- 0x0040 indicates the **RF Channel** field is subject to aliasing.
- 0x0400 indicates the CRC portion of the **LE Packet** was checked.
- 0x0800 indicates the CRC portion of the **LE Packet** passed its check.
- 0x1000 indicates the MIC portion of the decrypted **LE Packet** was checked.
- 0x2000 indicates the MIC portion of the decrypted **LE Packet** passed its check.

All other bit positions of the **Flags** field are reserved and must be zero.

The **LE Packet** follows the previous fields, and is formatted as detailed in Bluetooth specification Volume 6, Part B, Section 2, but does not include the preamble. All multi-octet values in the **LE Packet** are always expressed in little-endian format, as is the normal Bluetooth practice.

C Structure

```
typedef struct _pcap_bluetooth_le_ll_header {
        uint8_t rf_channel;
        int8_t signal_power;
        int8_t noise_power;
        uint8_t access_address_offenses;
        uint32_t ref_access_address;
        uint16_t flags;
        uint8_t le_packet[0];
} pcap_bluetooth_le_ll_header;
```

CHAPTER

EIGHT

HISTORY

The first hardware revision is called Ubertooth Zero and was demonstrated at ToorCon 12 on October 24th, 2010. Ubertooth Zero has been superseded.

The current hardware revision is called Ubertooth One and was demonstrated at ShmooCon 7 on January 29th, 2011.

CHAPTER

NINE

BUILDING FROM GIT

9.1 Prerequisites

There are some prerequisites that need to be installed before building libbtbb and the Ubertooth tools. Many of these are available from your operating system's package repositories, for example:

9.1.1 Debian/Ubuntu

```
sudo apt-get install git cmake libusb-1.0-0-dev make gcc g++ libbluetooth-dev \
pkg-config libpcap-dev python-numpy python-pyside python-qt4
```

9.1.2 Fedora / Red Hat

```
su -c "yum install git libusb1-devel make gcc wget tar bluez-libs-devel"
```

9.1.3 Mac OS X users can use either MacPorts or Homebrew to install the required packages:

```
sudo port install git libusb wget cmake python27 py27-numpy py27-pyside
or
brew install git libusb wget cmake pkg-config homebrew/dupes/libpcap
```

9.2 libbtbb

Next the Bluetooth baseband library (libbtbb) needs to be built for the Ubertooth tools to decode Bluetooth packets:

```
git clone https://github.com/greatscottgadgets/libbtbb.git
cd libbtbb
mkdir build
cd build
cmake ..
make
sudo make install
```

Linux users: if you are installing for the first time, or you receive errors about finding the library, you should run:

```
sudo ldconfig
```

9.3 Ubertooth tools

The Ubertooth repository contains host code for sniffing Bluetooth packets, configuring the Ubertooth and updating firmware. All three are built and installed by default using the following method:

```
git clone https://github.com/greatscottgadgets/ubertooth.git
cd ubertooth/host
mkdir build
cd build
cmake ..
make
sudo make install
```

Linux users: if you are installing for the first time, or you receive errors about finding the library, you should run:

```
sudo ldconfig
```

9.4 Wireshark

Wireshark version 1.12 and newer includes the Ubertooth BLE plugin by default. It is also possible to capture BLE from Ubertooth directly into Wireshark with a little work.

The Wireshark BTBB and BR/EDR plugins allow Bluetooth baseband traffic that has been captured using Kismet to be analysed and disected within the Wireshark GUI. They are built separately from the rest of the Ubertooth and libbtbb software. The directory passed to cmake as `MAKE_INSTALL_LIBDIR` varies from system to system, but it should be the location of existing Wireshark plugins, such as `asn1.so` and `ethercat.so`.

```
sudo apt-get install wireshark wireshark-dev libwireshark-dev cmake
cd libbtbb/wireshark/plugins/btbb
mkdir build
cd build
cmake -DCMAKE_INSTALL_LIBDIR=/usr/lib/x86_64-linux-gnu/wireshark/libwireshark3/
↪plugins ..
make
sudo make install
```

Then repeat for the BT BR/EDR plugin:

```
sudo apt-get install wireshark wireshark-dev libwireshark-dev cmake
cd libbtbb/wireshark/plugins/btbredr
mkdir build
cd build
cmake -DCMAKE_INSTALL_LIBDIR=/usr/lib/x86_64-linux-gnu/wireshark/libwireshark3/
↪plugins ..
make
sudo make install
```

9.5 Third Party Software

There are a number of pieces of third party software that support the Ubertooth. Some support Ubertooth out of the box, while others require plugins to be built.

9.6 Firmware

This completes the install of the Ubertooth tools, the next step is to look at the getting started guide. You should always update the firmware on the Ubertooth device to match the software release version that you are using.

CHAPTER

TEN

SOFTWARE

10.1 Building and Running Ubertooth Software

For information on how to download, build and install the Ubertooth software and dependencies, see the build guide.

For an introduction on the capabilities of Ubertooth and how to use it, see the getting started page.

10.2 Developing Host Code

Host code is the software running on a host computer to which an Ubertooth One is attached via USB.

The Ubertooth developers use GCC on Linux. We aim to have the host code work on OSX systems, but we cannot guarantee it; if you find a bug, please let us know. Theoretically, host code could be written on or for other platforms, but it hasn't been done yet.

Although source can be found in the release downloads, you will probably want fresh code from git if you plan to develop new code.

CHAPTER

ELEVEN

THIRD PARTY SOFTWARE

The following are third party applications that support Ubertooth One, either natively or with plugins. Unless otherwise stated, the Ubertooth project does not provide support these applications.

11.1 Kismet

The version of kismet provided by Debian/Ubuntu is 2008-05-R1, which is too old to support the Ubertooth plugin. In order to use Ubertooth with Kismet it is necessary to compile Kismet from source. First make sure that you have completed the instruction in the build guide and then use the following instruction to build Kismet:

```
sudo apt-get install libpcap0.8-dev libcap-dev pkg-config build-essential␣
↪libnl-3-dev libncurses-dev libpcre3-dev libpcap-dev libcap-dev libnl-genl-3-
↪dev
wget https://kismetwireless.net/code/kismet-2013-03-R1b.tar.xz
tar xf kismet-2013-03-R1b.tar.xz
cd kismet-2013-03-R1b
ln -s ../ubertooth-2015-10-R1/host/kismet/plugin-ubertooth .
./configure
make && make plugins
sudo make suidinstall
sudo make plugins-install
Add "pcapbtbb" to the "logtypes=..." line in kismet.conf
```

Support for the Kismet is provided by the Kismet project, but the plugins are part of the Ubertooth software releases. For queries about the Ubertooth plugins please see the getting help page.

11.2 Flying Squirrel

Flying Squirrel has built in support for Ubertooth. Unfortunately Flying Squirrel is not available to the general public. "Flying Squirrel is only available to DOD and federal agencies." this was the reply when one tried to request the download to fsadmin@nrl.navy.mil.

11.3 Spectools

Spectools is a very useful 2.4GHz spectrum monitor, showing multiple views of spectrum usage. Spectools supports Ubertooth if built from git.

CHAPTER

TWELVE

FIRMWARE

12.1 How To Update Firmware

First, grab the latest Ubertooth release. Then, extract the archive and change into directory `ubertooth-one-firmware-bin`.

You may then run the `ubertooth-dfu` command like so:

```
$ ubertooth-dfu -d bluetooth_rxtx.dfu -r
Switching to DFU mode...
Checking firmware signature
.....................................
.....................................
.............
```

The device will automatically enter DFU mode and flash the firmware.

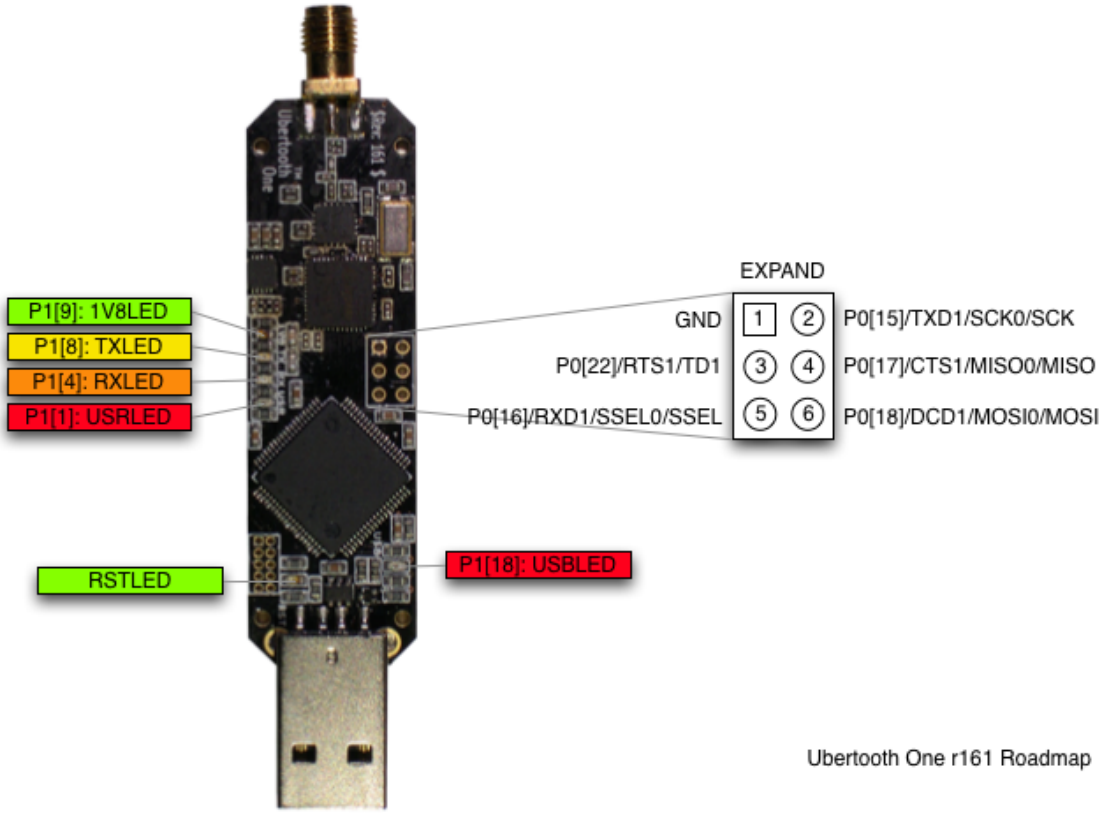
If you see `control message unsupported` at the end, this means that resetting the device failed. You can handle this by running `ubertooth-util -r` or just unplugging the USB cable from the Ubertooth and reconnecting it.

Troubleshooting:

If you run into an error such as "libUSB Error: Command Error: (-1)" or the Ubertooth's 4 LEDs next to each other perform a distinctive chasing pattern, after extracting the archive of the latest Ubertooth release, change into the `firmware` directory. Then, run:

```
make clean all && make
ubertooth-dfu -r -d bluetooth_rxtx/bluetooth_rxtx.dfu
```

If your Ubertooth Ones has no firmware or broken firmware and doesn't show up when issuing `lsusb`, you can force the Ubertooth One into DFU mode by connecting pins 1 and 3 on the EXPAND header (P4):



You should see the following appear in dmesg:

```
[ 1323.391369] usb 1-10: new full-speed USB device number 7 using xhci_hcd
[ 1323.541063] usb 1-10: New USB device found, idVendor=1d50, idProduct=6002,␣
↪bcdDevice= 1.05
[ 1323.541069] usb 1-10: New USB device strings: Mfr=1, Product=2,␣
↪SerialNumber=3
[ 1323.541073] usb 1-10: Product: Ubertooth One
[ 1323.541077] usb 1-10: Manufacturer: Great Scott Gadgets
[ 1323.541080] usb 1-10: SerialNumber: 07b00004c81435ae82624953861e00f5
[ 1341.978766] usb 1-10: USB disconnect, device number 7
```

The device will only stay in dfu mode for a few seconds, so you have to quickly issue the ubertooth-dfu command to flash.

12.1.1 What Version Am I Running?

In non-DFU mode, you can obtain firmware information with `ubertooth-util -v`. Note that the version shown should match the one you just installed:

```
$ ubertooth-util -v
Firmware version: 2018-12-R1 (API:1.05)
$ ubertooth-util -V
ubertooth 2018-12-R1 (mikeryan@steel) Tue Aug  7 15:33:06 PDT 2018
```

12.1.2 Developing Firmware

You'll need a toolchain that supports ARM Cortex-M3. The Makefiles in the firmware directory are designed for GCC and a Linux-based toolchain, specifically `arm-none-eabi-gcc and libnewlib-arm-none-eabi`. If you are running a Debian based distribution, you can run:

```
apt-get install gcc-arm-none-eabi libnewlib-arm-none-eabi
```

Otherwise it can be downloaded from https://launchpad.net/gcc-arm-embedded, just unpack the archive and add the bin directory to your PATH.

To build the firmware, start from the directory where you cloned or unpacked the source and run the following:

```
cd firmware/bluetooth_rxtx/
make
```

This will produce a file named bluetooth_rxtx.dfu which can be written to the Ubertooth using

```
ubertooth-dfu -d bluetooth_rxtx.dfu -r
```

Although firmware source and binary images can be found in the release downloads, you will probably want fresh code from git if you are planning to modify the firmware.

CHAPTER

THIRTEEN

PROGRAMMING

This page describes how to load compiled firmware onto an Ubertooth One. Other pages describe how to write firmware or host code for the platform.

There are three ways to program an Ubertooth board: the USB bootloader, the ISP bootloader, and JTAG.

Please note that it is not possible to replace the board's USB bootloader via ubertooth-dfu as there is software protection in place.

13.1 USB bootloader

This is the recommended method of loading code onto an Ubertooth Zero or Ubertooth One provided that the USB bootloader is already installed (at the time of manufacture, for example). If you need to install the bootloader itself, you will have to use either ISP or JTAG.

The bootloader executes every time the device starts up from reset or power cycle. Normally it just gets out of the way and passes control to the application firmware very quickly. Alternatively it can enter Device Firmware Upgrade (DFU) mode which permits firmware upload and download over USB. There are two ways to tell the bootloader that you want it to enter DFU mode:

1. soft bootloader entry: By setting a flag in RAM, the application firmware can instruct the bootloader to enter DFU mode following a reset (without loss of power). For example, with the bluetooth_rxtx firmware running you can trigger a reset into DFU mode using 'ubertooth-util -f' (located in host/bluetooth_rxtx/). Soft entry only works on Ubertooth One, not Ubertooth Zero.
2. hard bootloader entry (also known as pin entry): By connecting two pins (for Ubertooth Zero it's pin 1 and 3) on the expansion header with a jumper during reset (either soft or hard boot), you can force the bootloader into DFU mode. When using pin entry, the bootloader will enter DFU mode for only a few seconds and will then execute the application if no DFU activity has started during that period. This allows developers to permanently or semi-permanently jumper the pins providing a DFU opportunity on every reset. This method works on Ubertooth One but not Ubertooth Zero.

The bootloader indicates DFU mode by flashing the LEDs in a distinctive pattern. It also identifies itself as "usb_bootloader" on USB.

During DFU mode, firmware may be uploaded or downloaded using ubertooth-dfu.

It is possible that soft bootloader entry may be broken by installing a faulty application or an application that does not provide a method of triggering soft entry. In this case, pin entry must be used (e.g. by holding a paper clip in the expansion header while plugging in the device) to "unbrick" the unit.

For Ubertooth One, the pins to connect are pins 1 and 3 of the expansion header (P4). For Ubertooth Zero, the pins to connect are pins 1 and 13 of the expansion header (J1).

13.2 ISP bootloader

The LPC175x features an In-System Programming (ISP) bootloader that allows code to be loaded over a serial interface. In order to use ISP, you will need lpc21isp and a 3.3V serial programming device such as one of the following:

13.2.1 Pogoprog

The official Ubertooth ISP programmer is Pogoprog, an open source board that can be assembled using a process similar to Ubertooth One assembly.

13.2.2 An FTDI Basic Breakout - 3.3V

You can use SparkFun's FTDI Basic Breakout - 3.3V to program an Ubertooth. Adafruit's FTDI Friend is a similar board that should work as well. To allow lpc21isp to automatically activate the LPC175x's ISP bootloader, you must modify the board in one of two ways.

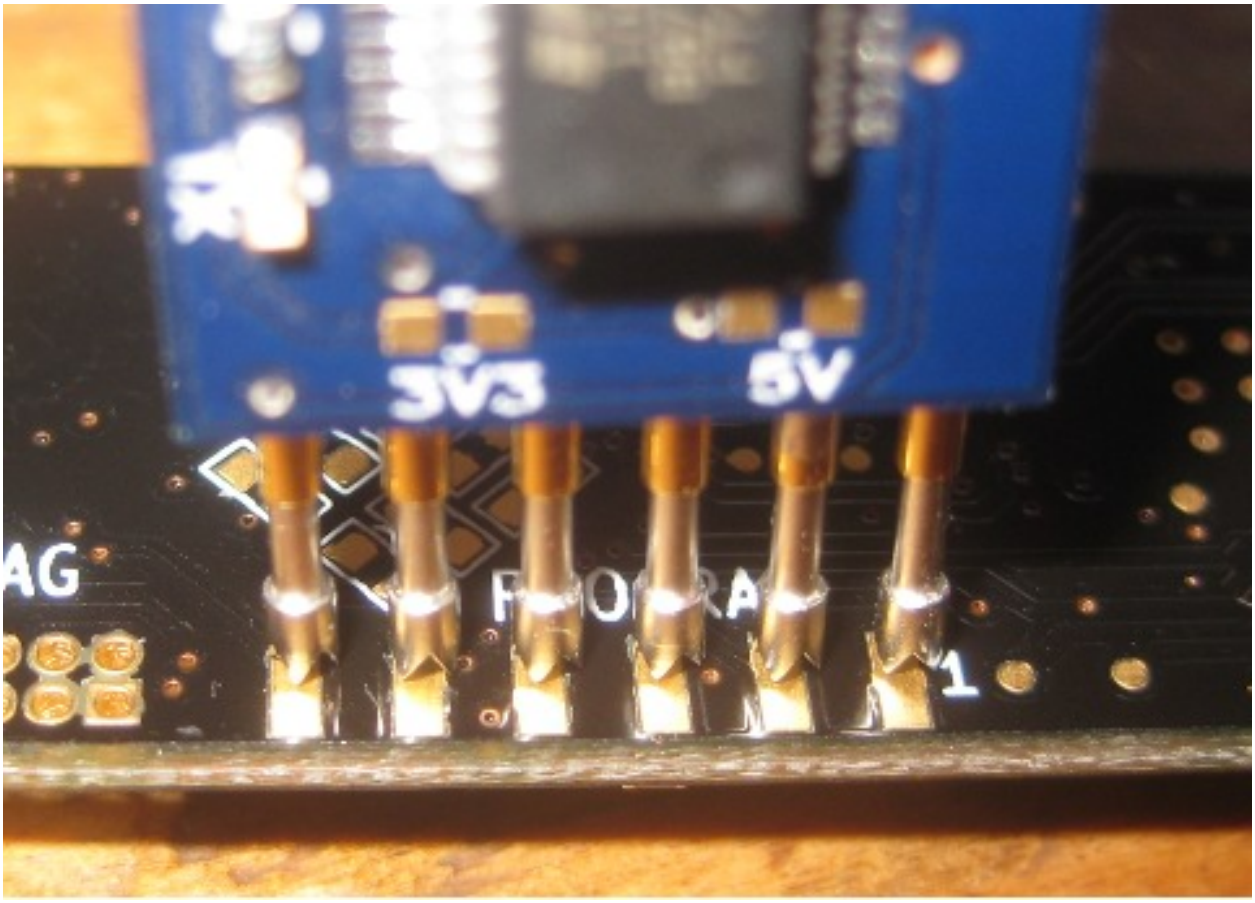
Method one: the easy way

Just short the CTS pin to GND. This will force the microcontroller into ISP mode every time it resets while the programmer is connected. In order to perform a normal reset to run the newly loaded code, you will have to first disconnect the programmer.

Method two: the better way

Connect the CTS pin to the RTS pin (pin 3) on the FT232RL using fine gauge wire such as wire wrap wire. This is trickier to solder, but it has the advantage that the pin will be fully controlled by lpc21isp. This means that you can leave the programmer connected to the Ubertooth board throughout multiple program/test cycles.

Using ISP



Warning: Code loaded via ISP will blow away the bootloader. It is generally recommended to use this method only for installing the bootloader itself.

Connect your serial programmer to the Ubertooth board and type "make program" in the firmware source code directory. If you have a precompiled binary in .hex format, you can invoke lpc21isp directly with "lpc21isp -control firmware.hex /dev/ttyUSB0 230400 4000" (replacing firmware.hex with the firmware filename and /dev/ttyUSB0 with the device file of your serial programmer.

13.2.3 JTAG

Warning: Code loaded via JTAG will blow away the bootloader. It is generally recommended to use this method only for installing the bootloader itself.

Connect your ARM Cortex-M3 JTAG debugger (such as one supported by OpenOCD) to the standard Cortex Debug Connector on the Ubertooth One or the non-standard JTAG connector on the Ubertooth Zero. You know what to do.

CHAPTER

FOURTEEN

ASSEMBLING HARDWARE

The quickest and easiest way to get Ubertooth hardware is to buy it. A list of resellers is available from the Great Scott Gadgets website. If you choose to build your own hardware, the following steps should help.

14.1 Step 0: read these instructions

Seriously. There are probably things in the later steps that you should know about before getting started.

14.2 Step 1: order a PCB

You will need a four layer printed circuit board. We recommend using OSH Park. Take the Ubertooth One Gerber files from the most recent Project Ubertooth release package (or generate them with KiCad) and send them to your chosen PCB manufacturer. The board is 1.8 square inches, so it will cost ~$18 per set of three. That's one for yourself, one for a friend, and one to screw up! Pogoprog is a two layer board.

If you are building an Ubertooth One, you should also consider building a Pogoprog unless you already have a plan for how you will program your board.

14.3 Step 2: order a stencil

Surface mount soldering is fun and easy if you use a stencil to apply solder paste to your circuit board! Send the top paste Gerber file (ubertooth-one-SoldP_Front.gtp) to a stencil manufacturer such as OHARARP, OSH Stencils, or Pololu. Alternatively you might plan to use a syringe or a toothpick to apply solder paste, but this is not recommended. You might instead just use a soldering iron, but this is strongly discouraged unless you have successfully soldered QFNs with required ground pads before (and, if you have, you probably aren't reading these instructions anyway).

14.4 Step 3: order the parts

Take the bill of materials (bom) from the most recent Project Ubertooth release (or generate it with KiCad) and order the parts. The parts should all be available from one or more online electronics suppliers such as Mouser or Digi-key. It is important to order some extra parts (especially the tiny ones which fortunately are cheap) in case you lose or damage any components.

You may want to order an antenna too. The Pulse W1030 is a nice size, but you can also find compatible antennas on many commercial Wi-Fi and Bluetooth products. Most any antenna intended for the 2.4 GHz band (such as 802.11b/g/n) is suitable as long as it has an RP-SMA connector, adapter, or pigtail. You could choose an SMA connector instead

of RP-SMA; this might especially be convenient for interfacing with benchtop test gear. RP-SMA was selected as the default choice for Project Ubertooth because there are more RP-SMA than SMA antennas floating around on consumer Wi-Fi and Bluetooth gear.

You might prefer to select alternative parts, but be careful of the 1% resistors and all of the 0402 inductors and capacitors in the RF section which have been selected for their particular characteristics. Any LPC175x microcontroller will do, but it is recommended that you choose one with at least 128 kB RAM. And, really, if you're going through all this trouble, why not go with 512 kB?

14.5 Step 4: prepare your tools and materials

Essential:

- an electric skillet, one that you don't intend to use for food ever again
- solder paste (no-clean lead-based solder paste is recommended)
- a small putty knife or razor blade
- fine tipped tweezers
- any soldering iron
- solder

Strongly recommended:

- good ventilation
- a temperature controlled soldering iron: this is more than just having a knob; it should have a temperature sensor in the iron
- an embossing tool or other high temperature heat gun (even better: a proper hot air rework station)
- a multitester with LED/diode test mode
- desoldering braid
- brass sponge
- helping hands
- magnifying glass

14.6 Step 5: apply solder paste

Using your stencil and a putty knife, apply the solder paste as described in this tutorial.

14.7 Step 6: place the parts

With fine tipped tweezers, carefully place the parts on the board. If you have to move a part, pick it up and place it again rather than sliding it around a lot. Otherwise the paste can get out of place. Most of the 0402 and 0603 parts can be placed in either direction, but the LEDs are exceptions. You must place them with ground in the direction of the arrow on the circuit board. You may have to look at the design in KiCad to see which way the arrow goes, and you'll probably have to test your LEDs with a multitester to find out which side is which. Don't populate USB connectors, RP-SMA connectors, or pogo pins (in the case of Pogoprog) at this time.

14.8 Step 7: reflow

Carefully place the board in the electric skillet, and turn the skillet on. It is best to warm up the board to a moderate temperature before turning the skillet up to full power. Then turn up the heat until you can see the solder flow. If you see parts moving around to incorrect positions, resist the temptation to correct them at this time! As soon as the solder everywhere on the board appears liquid, cut the power completely. You may want to lift the board out of the skillet with a spatula at this point to allow it to cool faster. There is a danger of overheating the components, but this is unlikely unless you left the skillet on longer than necessary or used lead-free solder paste.

14.9 Step 8: rework

Here is where the embossing tool, a good soldering iron, desoldering braid, and a magnifying glass come in very handy. If there is anything wrong with the assembly, you will have to correct individual part placement as needed.

14.10 Step 9: inspection

Once all the parts appear to be soldered in place correctly, look again, this time with a magnifying glass. You should also do some continuity tests with a multitester at this point. Watch out in particular for supply shorts; the easiest way to test for these is to verify a lack of continuity across bypass capacitors (all the caps that are close to the ICs). If there is a short that you can't see, it is probably under the pins of one of the ICs. Repeat steps 7 and 8 as necessary.

14.11 Step 10: hand soldering

There are a few parts that you should solder on by hand with an iron at this point. These are the USB and RP-SMA connectors on the Ubertooth boards and the pogo pins on Pogoprog.

14.12 Step 11: power-on test

Power on the device by plugging in the USB connection. An Ubertooth One or Zero should illuminate the RST LED. If this doesn't happen, quickly unplug USB verify that the LED is oriented correctly, and go back to step 9. A Pogoprog should flash its TX and RX LEDs during USB enumeration. If this doesn't happen, quickly unplug USB, verify the LED orientations, check your driver situation, and go back to step 8.

14.13 Step 12: further testing

If you are building a Pogoprog, you should make sure that an FTDI USB serial adapter has been detected by your host operating system. If so, you can try using it to program an Ubertooth board. If you are making an Ubertooth board, you should follow the procedure in firmware/assembly_test/README.

14.14 Step 13: boast

Tell us about your success on the Great Scott Gadgets Discord.

CHAPTER

FIFTEEN

RELEASE 2015-10-R1

15.1 Prerequisites

There are some prerequisites that need to be installed before building libbtbb and the Ubertooth tools. Many of these are available from your operating system's package repositories, for example:

15.1.1 Debian/Ubuntu

```
sudo apt-get install cmake libusb-1.0-0-dev make gcc g++ libbluetooth-dev \
pkg-config libpcap-dev python-numpy python-pyside python-qt4
```

15.1.2 Fedora / Red Hat

```
su -c "yum install libusb1-devel make gcc wget tar bluez-libs-devel"
```

15.1.3 Mac OS X users can use either MacPorts or Homebrew to install the required packages:

```
sudo port install libusb wget cmake python27 py27-numpy py27-pyside
or
brew install libusb wget cmake pkg-config homebrew/dupes/libpcap
```

15.2 libbtbb

Next the Bluetooth baseband library (libbtbb) needs to be built for the Ubertooth tools to decode Bluetooth packets:

```
wget https://github.com/greatscottgadgets/libbtbb/archive/2015-10-R1.tar.gz -O␣
↪libbtbb-2015-10-R1.tar.gz
tar xf libbtbb-2015-10-R1.tar.gz
cd libbtbb-2015-10-R1
mkdir build
cd build
cmake ..
```

(continues on next page)

(continued from previous page)

```
make
sudo make install
```

Linux users: if you are installing for the first time, or you receive errors about finding the library, you should run:

```
sudo ldconfig
```

15.3 Wireshark

Wireshark version 1.12 and newer includes the Ubertooth BLE plugin by default. It is also possible to capture BLE from Ubertooth directly into Wireshark with a little work.

The Wireshark BTBB and BR/EDR plugins allow Bluetooth baseband traffic that has been captured using Kismet to be analysed and disected within the Wireshark GUI. They are built separately from the rest of the Ubertooth and libbtbb software. The directory passed to cmake as `MAKE_INSTALL_LIBDIR` varies from system to system, but it should be the location of existing Wireshark plugins, such as `asn1.so` and `ethercat.so`.

```
sudo apt-get install wireshark wireshark-dev libwireshark-dev cmake
cd libbtbb-2015-10-R1/wireshark/plugins/btbb
mkdir build
cd build
cmake -DCMAKE_INSTALL_LIBDIR=/usr/lib/x86_64-linux-gnu/wireshark/libwireshark3/
↪plugins ..
make
sudo make install
```

Then repeat for the BT BR/EDR plugin:

```
sudo apt-get install wireshark wireshark-dev libwireshark-dev cmake
cd libbtbb-2015-10-R1/wireshark/plugins/btbredr
mkdir build
cd build
cmake -DCMAKE_INSTALL_LIBDIR=/usr/lib/x86_64-linux-gnu/wireshark/libwireshark3/
↪plugins ..
make
sudo make install
```

15.4 Third Party Software

There are a number of pieces of third party software that support the Ubertooth. Some support Ubertooth out of the box, while others require plugins to be built.

15.4.1 Firmware

This completes the install of the Ubertooth tools, the next step is to look at the getting started guide. You should always update the firmware on the Ubertooth device to match the software release version that you are using.

CHAPTER

SIXTEEN

RELEASE 2017-03-R2

for Release 2015-10-R1 see here

16.1 Prerequisites

There are some prerequisites that need to be installed before building libbtbb and the Ubertooth tools. Many of these are available from your operating system's package repositories, for example:

Debian / Ubuntu

```
sudo apt-get install cmake libusb-1.0-0-dev make gcc g++ libbluetooth-dev \
pkg-config libpcap-dev python-numpy python-pyside python-qt4
```

Fedora / Red Hat

```
su -c "yum install libusb1-devel make gcc wget tar bluez-libs-devel"
```

Mac OS X users can use either MacPorts or Homebrew to install the required packages:

```
sudo port install libusb wget cmake python27 py27-numpy py27-pyside
or
brew install libusb wget cmake pkg-config libpcap
```

FreeBSD users can install the host tools and library directly from the ports and package system:

```
sudo pkg install ubertooth
```

16.2 libbtbb

Next the Bluetooth baseband library (libbtbb) needs to be built for the Ubertooth tools to decode Bluetooth packets:

```
wget https://github.com/greatscottgadgets/libbtbb/archive/2017-03-R2.tar.gz -O␣
↪libbtbb-2017-03-R2.tar.gz
tar xf libbtbb-2017-03-R2.tar.gz
cd libbtbb-2017-03-R2
mkdir build
cd build
cmake ..
```

(continues on next page)

(continued from previous page)

```
make
sudo make install
```

Linux users: if you are installing for the first time, or you receive errors about finding the library, you should run:

```
sudo ldconfig
```

16.3 Ubertooth tools

The Ubertooth repository contains host code for sniffing Bluetooth packets, configuring the Ubertooth and updating firmware. All three are built and installed by default using the following method:

```
wget https://github.com/greatscottgadgets/ubertooth/releases/download/2017-03-
↪R2/ubertooth-2017-03-R2.tar.xz -O ubertooth-2017-03-R2.tar.xz
tar xf ubertooth-2017-03-R2.tar.xz
cd ubertooth-2017-03-R2/host
mkdir build
cd build
cmake ..
make
sudo make install
```

Linux users: if you are installing for the first time, or you receive errors about finding the library, you should run:

```
sudo ldconfig
```

16.4 Wireshark

Wireshark version 1.12 and newer includes the Ubertooth BLE plugin by default. It is also possible to capture BLE from Ubertooth directly into Wireshark with a little work.

The Wireshark BTBB and BR/EDR plugins allow Bluetooth baseband traffic that has been captured using Kismet to be analysed and disected within the Wireshark GUI. They are built separately from the rest of the Ubertooth and libbtbb software. The directory passed to cmake as `MAKE_INSTALL_LIBDIR` varies from system to system, but it should be the location of existing Wireshark plugins, such as `asn1.so` and `ethercat.so`.

```
sudo apt-get install wireshark wireshark-dev libwireshark-dev cmake
cd libbtbb-2017-03-R2/wireshark/plugins/btbb
mkdir build
cd build
cmake -DCMAKE_INSTALL_LIBDIR=/usr/lib/x86_64-linux-gnu/wireshark/libwireshark3/
↪plugins ..
make
sudo make install
```

Then repeat for the BT BR/EDR plugin:

```
sudo apt-get install wireshark wireshark-dev libwireshark-dev cmake
cd libbtbb-2017-03-R2/wireshark/plugins/btbredr
```

(continues on next page)

(continued from previous page)

```
mkdir build
cd build
cmake -DCMAKE_INSTALL_LIBDIR=/usr/lib/x86_64-linux-gnu/wireshark/libwireshark3/
↪plugins ..
make
sudo make install
```

16.5 Third Party Software

There are a number of pieces of third party software that support the Ubertooth. Some support Ubertooth out of the box, while others require plugins to be built.

16.5.1 Firmware

This completes the install of the Ubertooth tools, the next step is to look at the getting started guide. You should always update the firmware on the Ubertooth device to match the software release version that you are using.

CHAPTER

SEVENTEEN

RELEASE 2018-08-R1: THE DEFCON RELEASE

for Release 2017-03-R2 see here

17.1 Preequisites

There are some prerequisites that need to be installed before building libbtbb and the Ubertooth tools. Many of these are available from your operating system's package repositories, for example:

Debian / Ubuntu

```
sudo apt-get install cmake libusb-1.0-0-dev make gcc g++ libbluetooth-dev \
pkg-config libpcap-dev python-numpy python-pyside python-qt4
```

Fedora / Red Hat

```
su -c "yum install libusb1-devel make gcc wget tar bluez-libs-devel"s
```

Mac OS X users can use either MacPorts or Homebrew to install the required packages:

```
brew install libusb wget cmake pkg-config libpcap
or
sudo port install libusb wget cmake python27 py27-numpy py27-pyside
```

FreeBSD users can install the host tools and library directly from the ports and package system:

```
sudo pkg install ubertooth
```

17.2 libbtbb

Next the Bluetooth baseband library (libbtbb) needs to be built for the Ubertooth tools to decode Bluetooth packets:

```
wget https://github.com/greatscottgadgets/libbtbb/archive/2018-08-R1.tar.gz -O␣
↪libbtbb-2018-08-R1.tar.gz
tar -xf libbtbb-2018-08-R1.tar.gz
cd libbtbb-2018-08-R1
mkdir build
cd build
cmake ..
```

(continues on next page)

(continued from previous page)

```
make
sudo make install
```

Linux users: if you are installing for the first time, or you receive errors about finding the library, you should run:

```
sudo ldconfig
```

17.3 Ubertooth tools

The Ubertooth repository contains host code for sniffing Bluetooth packets, configuring the Ubertooth and updating firmware. All three are built and installed by default using the following method:

```
wget https://github.com/greatscottgadgets/ubertooth/releases/download/2018-08-
↪R1/ubertooth-2018-08-R1.tar.xz
tar xf ubertooth-2018-08-R1.tar.xz
cd ubertooth-2018-08-R1/host
mkdir build
cd build
cmake ..
make
sudo make install
```

Linux users: if you are installing for the first time, or you receive errors about finding the library, you should run:

```
sudo ldconfig
```

17.4 Wireshark

Wireshark version 1.12 and newer includes the Ubertooth BLE plugin by default. It is also possible to capture BLE from Ubertooth directly into Wireshark with a little work.

The Wireshark BTBB and BR/EDR plugins allow Bluetooth baseband traffic that has been captured using Kismet to be analysed and disected within the Wireshark GUI. They are built separately from the rest of the Ubertooth and libbtbb software.

The directory passed to cmake as `MAKE_INSTALL_LIBDIR` varies from system to system, but it should be the location of existing Wireshark plugins, such as `asn1.so` and `ethercat.so`. On macOS this is likely `/Applications/Wireshark.app/Contents/PlugIns/wireshark/`.

```
sudo apt-get install wireshark wireshark-dev libwireshark-dev cmake
cd libbtbb-2018-08-R1/wireshark/plugins/btbb
mkdir build
cd build
cmake -DCMAKE_INSTALL_LIBDIR=/usr/lib/x86_64-linux-gnu/wireshark/libwireshark3/
↪plugins ..
make
sudo make install
```

Then repeat for the BT BR/EDR plugin:

```
sudo apt-get install wireshark wireshark-dev libwireshark-dev cmake
cd libbtbb-2018-08-R1/wireshark/plugins/btbredr
mkdir build
cd build
cmake -DCMAKE_INSTALL_LIBDIR=/usr/lib/x86_64-linux-gnu/wireshark/libwireshark3/
↪plugins ..
make
sudo make install
```

17.5 Third Party Software

There are a number of pieces of third party software that support the Ubertooth. Some support Ubertooth out of the box, while others require plugins to be built.

17.5.1 Firmware

This completes the install of the Ubertooth tools, the next step is to look at the getting started guide. You should always update the firmware on the Ubertooth device to match the software release version that you are using.

CHAPTER

EIGHTEEN

RELEASE 2018-12-R1

18.1 Prerequisites

There are some prerequisites that need to be installed before building libbtbb and the Ubertooth tools. Many of these are available from your operating system's package repositories, for example:

Debian / Ubuntu / Kali

```
sudo apt-get install cmake libusb-1.0-0-dev make gcc g++ libbluetooth-dev \
  pkg-config python3-numpy python3-qtpy
```

Fedora / Red Hat

```
su -c "yum install libusb1-devel make gcc wget tar bluez-libs-devel"
```

Mac OS X users can use either MacPorts or Homebrew to install the required packages:

```
brew install libusb wget cmake pkg-config
or
sudo port install libusb wget cmake python38 py38-numpy py38-qtpy
```

FreeBSD users can install the host tools and library directly from the ports and package system:

```
sudo pkg install ubertooth
```

18.2 libbtbb

Next the Bluetooth baseband library (libbtbb) needs to be built for the Ubertooth tools to decode Bluetooth packets:

```
wget https://github.com/greatscottgadgets/libbtbb/archive/2018-12-R1.tar.gz -O␣
↪libbtbb-2018-12-R1.tar.gz
tar -xf libbtbb-2018-12-R1.tar.gz
cd libbtbb-2018-12-R1
mkdir build
cd build
cmake ..
make
sudo make install
```

Linux users: if you are installing for the first time, or you receive errors about finding the library, you should run:

```
sudo ldconfig
```

18.3 Ubertooth tools

The Ubertooth repository contains host code for sniffing Bluetooth packets, configuring the Ubertooth and updating firmware. All three are built and installed by default using the following method:

```
wget https://github.com/greatscottgadgets/ubertooth/releases/download/2018-12-
↪R1/ubertooth-2018-12-R1.tar.xz
tar xf ubertooth-2018-12-R1.tar.xz
cd ubertooth-2018-12-R1/host
mkdir build
cd build
cmake ..
make
sudo make install
```

Linux users: if you are installing for the first time, or you receive errors about finding the library, you should run:

```
sudo ldconfig
```

18.4 Wireshark

Wireshark version 1.12 and newer includes the Ubertooth BLE plugin by default. It is also possible to capture BLE from Ubertooth directly into Wireshark with a little work.

The Wireshark BTBB and BR/EDR plugins allow Bluetooth baseband traffic that has been captured using Kismet to be analysed and disected within the Wireshark GUI. They are built separately from the rest of the Ubertooth and libbtbb software.

The directory passed to cmake as `MAKE_INSTALL_LIBDIR` varies from system to system, but it should be the location of existing Wireshark plugins, such as `asn1.so` and `ethercat.so`. On macOS this is likely `/Applications/Wireshark.app/Contents/PlugIns/wireshark/`.

```
sudo apt-get install wireshark wireshark-dev libwireshark-dev cmake
cd libbtbb-2018-12-R1/wireshark/plugins/btbb
mkdir build
cd build
cmake -DCMAKE_INSTALL_LIBDIR=/usr/lib/x86_64-linux-gnu/wireshark/libwireshark3/
↪plugins ..
make
sudo make install
```

Then repeat for the BT BR/EDR plugin:

```
sudo apt-get install wireshark wireshark-dev libwireshark-dev cmake
cd libbtbb-2018-12-R1/wireshark/plugins/btbredr
mkdir build
cd build
cmake -DCMAKE_INSTALL_LIBDIR=/usr/lib/x86_64-linux-gnu/wireshark/libwireshark3/
↪plugins ..
```

(continues on next page)

(continued from previous page)

```
make
sudo make install
```

18.5 Third Party Software

There are a number of pieces of third party software that support the Ubertooth. Some support Ubertooth out of the box, while others require plugins to be built.

18.5.1 Firmware

This completes the install of the Ubertooth tools, the next step is to look at the getting started guide. You should always update the firmware on the Ubertooth device to match the software release version that you are using.

CHAPTER

NINETEEN

TODO LIST

19.1 PCAP

- Obtain DLT/linktype from libPCAP devs for BTBB, see here.
- Define format for including meta-data (using PPI), see here.
- Move Wireshark plugins to Wireshark repo.
- Add pcap support to ubertooth tools (currently only in kismet) by moving it to libbtbb.

19.2 Basic rate / libbtbb

- Better handling of AFH maps when trying clock values. The code currently makes the assumption that AFH is enabled but all channels are in use.
- Detect syncword on Ubertooth for known piconets
- Make logging configurable - possibly combine text based logging, file dumps and pcap writing in to one logging system.

19.3 Releases

- Binary packages for both libbtbb and ubertooth - rpm, deb, others?
- More frequent releases - requires better testing of code in git and separation of half-implemented features. This is more about process than an actionable todo item.
- DONE: Add uninstall to Makefile.
- Add WICD headers to support windows users (requires switch to libusbx).

19.4 GR-Bluetooth

- Finish migration to libbtbb.
- Add BTLE support.
- General performance improvements (there must be some!).

19.5 Bigger / more vague projects

- Transmit basic rate packets - inquiry scan to start with.
- Implement full BTLE device in firmware.
- ANT/ANT+ sniffing.
  - http://www.thisisant.com/developer/resources/downloads/#documents_tab
  - http://www.thisisant.com/developer/resources/tech-faq/
- Communicate between Ubertooth devices using SPI on the expansion port.
- Communicate with other devices using SPI on the expansion port.
- Write to SD card using SPI on the expansion port.

CHAPTER

TWENTY

RELEASE PROCEDURE

An Ubertooth release is not as simple as a standard GitHub release, there are a few extra tasks, as listed below.

- Tag both libbtbb and ubertooth with the release version. Use an annotated tag. e.g.
- git tag -a yyyy-mm-Rx
- Create a release directory using git archive
- Update the following files to set RELEASE to yyyy-mm-Rx:
  - ubertooth/libubertooth/src/CMakeLists.txt
  - libbtbb/lib/src/CMakeLists.txt
- Build binary firmware images for all board revs and all common firmware images.
- At time of writing, these are only bluetooth_rxtx and bootloader for Ubertooth One.
- Adjust the GIT_REVISION variable in firmware/common.mk to match the release number
- Build the bluetooth_rxtx firmware and rename bluetooth_rxtx.dfu to ubertooth-one-bin-firmware.dfu
- Copy built firmare hex/bin/dfu files to /ubertooth-one-firmware-bin
- Export gerbers from KiCad (or copy from previous release if unchanged)
- Write release notes, save to top level of archive, add to the wiki and send to the mailing list
- Write/update the build instructions on the wiki as needed
- Perhaps we should branch and do some of this on the branch before tagging, this seems like good practice.

CHAPTER

TWENTYONE

UBERTOOTH TWO WISHLIST

At some point the supply of CC2400 ICs will dry up and we will need to design and build a replacement for Ubertooth. The design goals are the same as Ubertooth One, but it would be nice to add some additional features if we can. This page is for a wishlist of features.

MCU candidates:

- STM32F2
- LPCxxxx (Which? 18xx? 43xx? 17xx?)

RF candidates:

- ADF7242
- CC2541
- TRC104 - Fixed FSK deviation - unsuitable for BLE
- Amiccom A7137 - Not easy to source

21.1 Wishlist

- Improved MCU
  - USB peripheral with 512 byte bulk endpoints
  - AES peripheral - ITAR may be an issue

Note: LPC185x, LPC183x, and LPC182x MCUs have a high speed USB peripheral capable of 512 byte bulk endpoints. Reference: LPC18xx User Manual (UM10430 (PDF)) Chapter 22 Section 22.4.5, PDF page 520.

Room for an SD expansion - preferably using an MCU with SDIO.

Lots of randomly blinking LEDs.

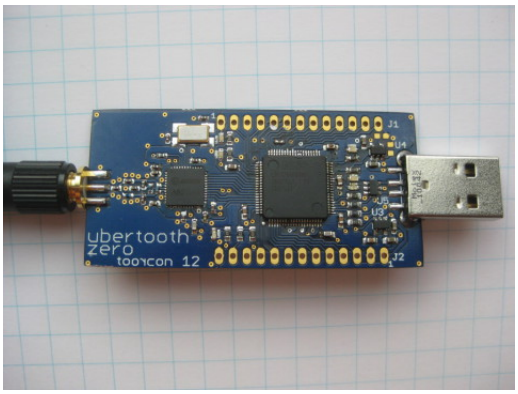
- a more sophisticated transceiver capable of handling EDR payloads

CHAPTER

TWENTYTWO

UBERTOOTH ZERO

Ubertooth Zero was the first working prototype hardware platform of Project Ubertooth. It has been superseded by Ubertooth One.



22.1 Architecture

- RP-SMA RF connector: connects to test equipment, antenna, or dummy load.
- CC2400 wireless transceiver IC.
- LPC175x ARM Cortex-M3 microcontroller with Full-Speed USB 2.0.
- USB A plug: connects to host computer running Kismet or other host code.

22.2 Features

- 2.4 GHz transmit and receive.
- Transmit power and receive sensitivity comparable to a Class 3 Bluetooth device.
- Non-standard JTAG connector.
- In-System Programming (ISP) serial connector.
- Expansion connector: intended for inter-Ubertooth communication or other future uses.
- Six indicator LEDs.

22.3 Design

Ubertooth Zero was designed in CadSoft EAGLE with surface mount components suitable for reflow.

22.4 Demonstration

Michael Ossmann presented Ubertooth Zero, a preview (video: part 1, part 2) at ToorCon 12 in October, 2010.