Programmable Testbed for Bluetooth Experimentation

by

Galahad M. B. Wernsing

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Master of Science
in
Electrical and Computer Engineering
by

____

December 2019

APPROVED:

____
Dr. Alexander Wyglinski, Research Advisor

____
Dr. Shamsnaz Bhada, Committee Member

____
Dr. David Medich, Committee Member

Abstract

Wireless device development requires extensive testing of the hardware and software that is being developed. Existing technologies used to test Bluetooth systems are limited by both design constraints and high hardware costs, restricting their usefulness. This project developed and demonstrated a Bluetooth testbed addressing limitations with existing systems by taking a unique approach to data collection. The three parts of this project are: a throughput test with data logging, a firmware loading utility for the hardware used in this project, and an interface with a commercially available Software-Defined Radio.

Acknowledgements

I would like to express my deepest gratitude to Professor Alexander Wyglinski for guiding me through both my undergraduate and master's degree programs, along with including me in the Wireless Innovation Laboratory at Worcester Polytechnic Institute.

I would like to thank Dr. Shamsnaz Bhada and Dr. David Medich for their comments on and suggestions for my thesis while serving on my committee.

I want to thank the people at octoScope, Inc. who helped me through technical problems and offered me the opportunity to work with them on my project.

A special thank you goes to my father, who helped me with the illustrations for this thesis.

I also give my thanks to the other WiLab team members, my friends, and my family. Without their support and encouragement this project could never have happened.

Contents

List of Figures **vi**

List of Tables **ix**

List of Acronyms **xii**

1 Introduction **1**

2 Overview of Bluetooth and Related Technologies **5**
2.1 Bluetooth Communication 6
2.1.1 Bluetooth Low Energy (BLE) Protocol Stack 6
2.1.2 Physical Properties 6
2.1.3 Pairing Process 11
2.2 Sniffing Systems 13
2.3 Cypress Development Tools 16
2.4 Data Formats 17
2.4.1 I32HEX Firmware Images 17
2.4.2 Cypress HCI Packets 18
2.4.3 BTSnoop 19
2.4.4 PCAP & PCAPNG 20
2.5 Summary 23

3 File Transfer Capabilities **24**
3.1 Reverse Engineering the Evaluation Board System 25
3.2 Throughput Test Using ClientControl 26
3.3 Throughput Test Using a Python Script 29
3.4 Recreating the Transferred File 30
3.5 Resolving Serial Buffer Overflow 31
3.6 Summary 34

4 Firmware Loader **35**
4.1 Firmware Loader Development 35
4.2 Firmware Loading Process 38
4.3 Summary 40

5 Promiscuous Sniffing Using a Litepoint SDR **41**
5.1 Packet Capture 42
5.2 Litepoint Interface 44
5.3 Summary 45

6 Conclusion **46**
6.1 Research Outcomes 46
6.2 Future Work 46

A Throughput Software **48**
A.1 PyControl 48
A.2 Communicator 52
A.3 Interpreter 56
A.4 Logger 59
A.5 Plotter 62

B Firmware Flasher Utility **63**

C Litepoint Interface **70**

Bibliography **79**

List of Figures

1.1 Historic and future trends for global internet-connected devices, originally from [23]. The steep increase between 2015 and 2025 is due to emerging markets and IoT devices. 2
1.2 A small octoScope system with a Wi-Fi router being tested [15]. On top of the anechoic chamber is the server and emulation hardware allowing for simulation of multiple Wi-Fi access points and end-users. This project worked towards adding Bluetooth support to these systems. 3

2.1 BLE layer diagram. The HCI is located between the link layer and higher parts of the protocol stack. 6
2.2 BLE channel locations in the Industrial, Scientific, and Medical (ISM) band. The large red background channels are the most frequently used Wi-Fi bands in the US market. The gold channels are the BLE advertising channels, chosen to avoid interference from Wi-Fi to increase reliability of discovering other devices. Note that they are numbered out-of-order from the regular channels. 7
2.3 BLE packet whitening LFSR [24]. The LFSR is initialized for each transmission with a value known to both the master and the slave. 8
2.4 BLE packet structure. The payload can be any size from 2-257 bytes [1]. The Preamble and AA are not whitened because they are used for packet detection by a receiver. 8
2.5 BLE connection interval timing diagram. Packets other than the heartbeat packets at the beginning of the connection event are not required. 9
2.6 Depiction of a possible Bluetooth (BT) scatternet [8]. Some devices belonging to multiple piconets can act as both a master (M) and a slave (S). Nodes marked "P" are advertising-only nodes and are not relevant to this project. 10
2.7 BLE pairing process selection. LE Secure Connections can only be used when connecting devices running at least BLE version 4.2. This process prioritizes more secure pairing methods, falling back to less secure methods when devices have lower functionality. 12
2.8 Promiscuous and inline sniffers in use. The promiscuous sniffer must listen in on a connection it is not a part of, while the inline sniffer must be part of the connection to be spied on. 14

2.9 Picture of the CYW20719 evaluation board [21]. The micro-USB connection used to communicate with a computer is on the left, the buttons and user-controllable LEDs are in the bottom right, and the CYW20719 and built-in antenna are on the daughter board on the right. 16
2.10 Representation of I32HEX record format. The data field is variable length, but was almost always 240 bytes long in firmware files made by Cypress. . . 18
2.11 HCI packet inside of a Cypress API wrapper. Note the flags in the HCI packet type field do not match those required for HCI H4 format. 18
2.12 BTSnoop file format. Note the data fields are arbitrary length and not aligned to any specific byte boundary. 20
2.13 Minimum PCAP file. Packet data is variable length and not padded to any specific size multiple. 21
2.14 Minimum PCAPNG file as used in this project. The options fields are unused, and packet data must be padded to be a multiple of 32-bits long. 22

3.1 Overview of file transfer architecture. The CYW20719 largely acts as a interface between the test phone and the test laptop. 24
3.2 Screenshot of BT sharing options on an Android phone. The evaluation board appears as "OPP Server" in this image. The specific text displayed on other devices is configurable by changing a field in the firmware source code. 27
3.3 Example CC screen at the end of a file transfer. The name of the file and size are visible, while the elapsed time can be calculated from the last two messages in the log view. 28
3.4 Screenshot of the hard disk latency. Note the extremely high utilization since the test started roughly 40 seconds previously. Average response time sporadically exceeded 50 milliseconds, representing a huge delay in the python code. 32
3.5 Throughput graph output of PyControl transferring an image from the test smartphone. The occasional dips below 150kbps are caused by hard drive latency spikes. 32

4.1 HCI packet format for firmware load packets. HCI commands sent to the CYW20719 do not need to be wrapped in an API header. 36
4.2 Flowchart of the firmware loading process. If any of the steps fail the solution is to power-cycle the eval board and try again. 38

5.1 Image of Litepoint IQxel-M16W [13]. Each of the ports on the front can be configured as an input or an output and monitored separately. 41
5.2 Eye diagram for data with incorrect receiver settings. The data was severely clipped, leading to random noise where a packet should be. 43

5.3 Eye diagram for data taken with correct receiver settings. Frequency offset of the received signal from the channel center, used in BT to represent packet data, is the vertical axis, while time offset from symbol center is represented on the horizontal axis. Individual symbols in the transmission are layered on top of each other, showing the repeatability and accuracy of captured symbol transitions. This is a good representation of clean, real-world data. 43
5.4 IQxel web interface. Input configuration controls are on the left, while user-configurable graphs are on the right. The data shown is advertising packets taken during initial testing of the IQxel. 45

List of Tables

2.1 Comparison between Bluetooth Classic (BTC) and BLE physical parameters. Important differences are the hopping times and number of channels. 6
2.2 Comparison of security vulnerabilities in the different BLE pairing schemes. Pairing using LE Secure Connections is much safer than LE Legacy Pairing. 11
2.3 Comparison of commercially available promiscuous BT sniffing options [2]. Note the widely varying price points and that the LimeSDR is not BT specific. 14

3.1 Reliability testing data of the throughput test using PyControl. No errors that occurred were caused by PyControl, only the smartphone used during the tests. 33

5.1 Settings used on the IQxel to successfully decode BLE packets. 42

List of Acronyms

AA Access Address

AGC Automatic Gain Control

API Application Programming Interface

ASPMON Advanced Serial Port Monitor

BLE Bluetooth Low Energy

BT Bluetooth

BTC Bluetooth Classic

BTS BTSnoop

CC ClientControl

CI Connection Interval

CPU Central Processing Unit

CRC Cyclic Redundancy Check

CS Checksum

dBm decibel-milliwatts

ECDH Elliptic-Curve Diffie-Hellman

EoF End of File

EPB Enhanced Packet Block

GFSK Gaussian Frequency-Shift Keying

GHz Gigahertz

GPIO General-Purpose Input/Output

GUI Graphical User Interface

HCI Host Controller Interface

IC Integrated Circuit

IDB Interface Description Block

IDE Integrated Development Environment

IO Input-Output

IoT Internet-of-Things

ISM Industrial, Scientific, and Medical

JW Just Works

kB kilobyte

kbps kilobits-per-second

LED Light Emitting Diode

LFSR Linear Feedback Shift Register

LTK Long Term Key

MB Megabyte

Mbps Megabits-per-second

MHz Megahertz

MITM Man-In-The-Middle

NC Numeric Comparison

NFC Near-Field Communication

OOB Out Of Bound

OPP Object Push Profile

PCAP Packet Capture

PCAPNG PCAP Next Generation

PER Packet Error Rate

RAM Random Access Memory

ROM Read Only Memory

rpm Rotations-per-Minute

SCPI Standard Commands for Programmable Instruments

SDR Software-Defined Radio

SHB Section Header Block

SNR Signal-to-Noise Ratio

STK Short Term Key

TDMA Time-Division Multiple Access

TK Temporary Key

UART Universal Asynchronous Receiver/Transmitter

USB Universal Serial Bus

Chapter 1

Introduction

Wireless communications technology is rapidly growing in speed, reliability, and importance in the modern world. The ubiquity of smartphones, which combine multiple wireless technologies into one device, causes high demand for improvements in wireless communication systems. Experts predict the number of internet connected devices to more than triple to 70 billion between 2015 and 2025; this increase is largely driven by expanded use of wireless technologies in third-world countries and Internet-of-Things (IoT) devices primarily using Wi-Fi and Bluetooth (BT) [23]. Existing data and predictions for number of internet-connected devices and devices per person are shown in Figure 1.1. This increase in the number and kinds of wireless devices drives demand for supporting equipment in all stages of wireless device development. This project focuses on one aspect of the development process: hardware validation of BT devices in a controlled wireless environment.

Development and testing of wireless devices requires both a controlled wireless environment and equipment capable of capturing, analyzing, and recording transmissions from the test device. Environmental control is usually done using a Faraday cage or an anechoic chamber, often large enough to walk into and set up a bench for equipment. Processing the wireless signals is usually performed by dedicated hardware that is limited to a single protocol. One leader in this market, octoScope, offers a unique combination of environmental isolation and protocol testing. Anechoic chambers are available in sizes ranging from a bread box to a small refrigerator, with an example shown in Figure 1.2. Chambers can be

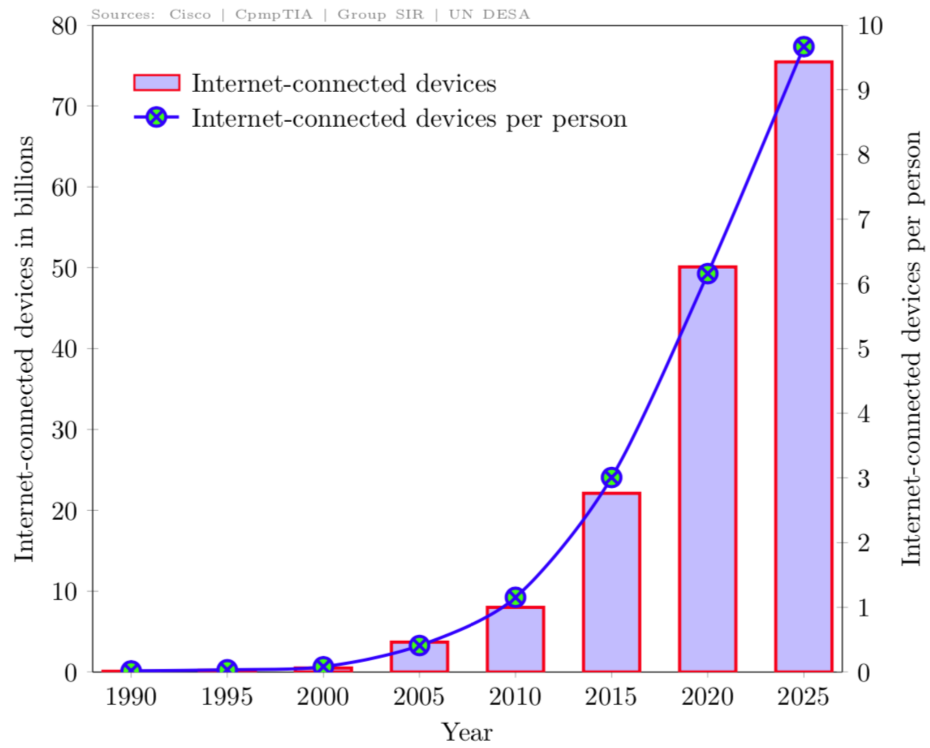

Figure 1.1: Historic and future trends for global internet-connected devices, originally from [23]. The steep increase between 2015 and 2025 is due to emerging markets and IoT devices.

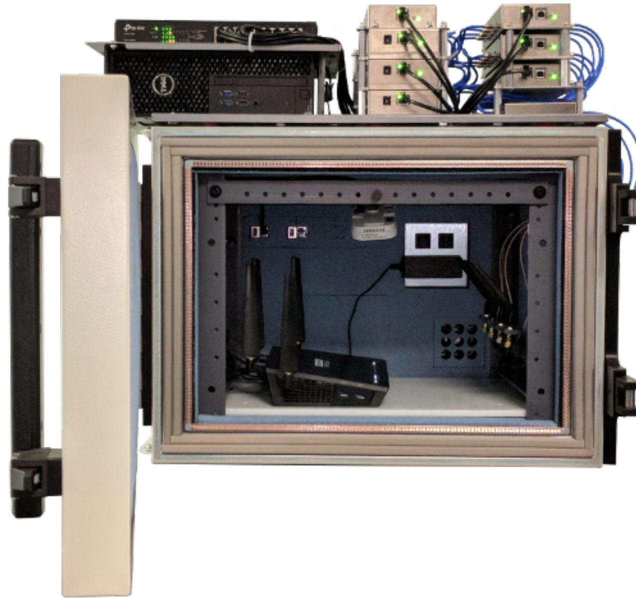

Figure 1.2: A small octoScope system with a Wi-Fi router being tested [15]. On top of the anechoic chamber is the server and emulation hardware allowing for simulation of multiple Wi-Fi access points and end-users. This project worked towards adding Bluetooth support to these systems.

connected to each other to extend the enclosed environment. In addition to such chambers, wireless hardware is also available to emulate many simultaneous Wi-Fi networks with connected devices to allow the device in development to be tested in a real-world environment. Such hardware also is available to emulate standardized wireless settings, such as inside a house, and make directional antenna performance measurements. This tight integration of environmental controls with protocol testing allows for faster, more accurate data collection and represents the future for this type of equipment.

Current BT development systems have a number of limitations even if a system like those produced by octoScope is used. Most of these limitations exist because the test system is not part of the BT connection it is monitoring. Data availability about the connection being tested is generally high, but hardware that can accommodate the channel hopping system used in BT is expensive and requires large computational resources. Additionally, while data about environmental factors is readily available, data contained within the packets themselves may be unavailable due to encryption used in the connection. Partially as a

result of the encryption issue, statistics about the connection itself are largely linked to physical parameters and do not include analysis of data carried over the link itself.

The goal of this project was to develop a BT system that will enable testbeds to be a partner in the link instead of an outside observer. Acting as a link partner lessens hardware performance requirements because a commercially available BT radio can be used that automatically manages the channel changes and encryption instead of a Software-Defined Radio (SDR) with custom software and enough processing power to perform all the calculations in real time. In addition to lessening hardware requirements, a commercial BT radio can give context-aware analysis of data crossing the link that the SDR cannot. To support the BT radio used in this project, a customizable firmware loader utility was built to allow rapid switching of test scenarios ranging from human interface devices to data transfer. Lastly, an external receiver was added to fill in gaps in the availability of environmental data, addressing one of the major limitations of a BT radio approach.

This thesis is organized as follows: Chapter 2 presents necessary background information on the design of BT communication, various existing BT testing systems, and different data formats encountered in this project. Chapter 3 presents the proof-of-concept data collection system, demonstrating capabilities for collection and analysis of a wireless file transfer. Chapter 4 covers the development of a customizable firmware loader purpose-built for the hardware used in this project. Chapter 5 presents the framework for a secondary receiver to augment environmental data collection capabilities. Lastly, the final chapter discusses the overall outcomes of the project and highlights areas for improvement in the future, should the project be continued.

Chapter 2

Overview of Bluetooth and Related Technologies

Bluetooth (BT) is a family of wireless communication standards using the 2.4GHz Industrial, Scientific, and Medical (ISM) band, a section of the electromagnetic spectrum with fewer regulations and no licensing requirements [24][25]. Originally conceived as a wireless replacement for RS-232 cables, it has grown to be used for a multitude of applications, from sensors and game controllers to audio and data transmissions [5]. A separate protocol named Bluetooth Low Energy (BLE) was introduced in 2010 alongside the renamed Bluetooth Classic (BTC) in BT version 4.0, focusing on reducing power requirements to allow better connections for battery powered Internet-of-Things (IoT) devices. BLE is similar to but not compatible with the original BTC, although many devices support both protocols. Key differences in physical parameters between BLE and BTC are listed in Table 2.1. This project was based on a BLE system so information will be for that protocol unless otherwise noted.

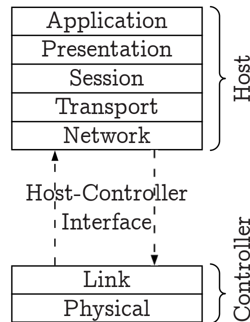

Figure 2.1: BLE layer diagram. The HCI is located between the link layer and higher parts of the protocol stack.

2.1 Bluetooth Communication

2.1.1 BLE Protocol Stack

Below the application layer, the BT stack is split into two sections, as illustrated in Figure 2.1: the “host” and the “controller”. The controller handles the physical and link layers including much of the pairing process, while the host handles everything between the link layer and the application layer. Between the host and controller is the Host Controller Interface (HCI). The host and controller can be either split between the BT chip and a higher level CPU, such as in a cell phone, or both can exist on the same BT Integrated Circuit (IC).

2.1.2 Physical Properties

BT has multiple device classes based on transmit power. Most devices transmit at +4dBm, which gives roughly 33ft of range, depending on the surrounding environment [24]. The most powerful transmitters broadcast at +20dBm, enabling over 300ft of range depending on receiver sensitivities.

Table 2.1: Comparison between BTC and BLE physical parameters. Important differences are the hopping times and number of channels.

Version	Base Data Rate	Coding Scheme	Number of Channels	Hopping Time
BTC	1Mbps	GFSK	79	625μs
BLE	1Mbps	GFSK	40	75ms-4s

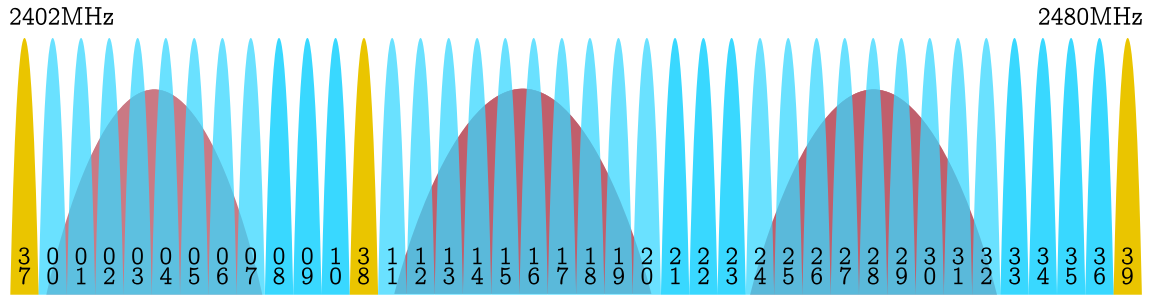

Figure 2.2: BLE channel locations in the ISM band. The large red background channels are the most frequently used Wi-Fi bands in the US market. The gold channels are the BLE advertising channels, chosen to avoid interference from Wi-Fi to increase reliability of discovering other devices. Note that they are numbered out-of-order from the regular channels.

BLE splits the ISM band into 40 different 2MHz channels. Three of these 40 channels are reserved as advertising channels, used to allow devices to see each other, and to initiate a connection [24]. These channels were selected to avoid the most commonly used Wi-Fi channels, as interference between Wi-Fi and BT can cause problems for both protocols [4]. Figure 2.2 highlights the advertising channels used to avoid Wi-Fi interference. The other 37 channels are used for transmissions between connected devices.

Detection of broadcast packets is based on a one byte preamble, used to synchronize receivers before transmission of packet bits, and a four byte Access Address (AA) [18][11]. The AA for connections is a semi-random number and created during connection setup, and the AA for advertising channels is a constant set by the BLE specification. Using both a preamble and an AA increases receiver efficiency since the receiver can stop processing packets not addressed to it.

Broadcast packets contain a payload and a Cyclic Redundancy Check (CRC) after the AA. Both of these fields are whitened before broadcast using the output of the Linear Feedback Shift Register (LFSR) shown in Figure 2.3. The whitening process randomizes bits before broadcast to remove long sequences of 1s or 0s that could confuse a receiver. This does not improve security, as the LFSR connections are known and the state can be easily brute-forced [6][18]. The structure of a broadcast BLE packet is shown in Figure 2.4.

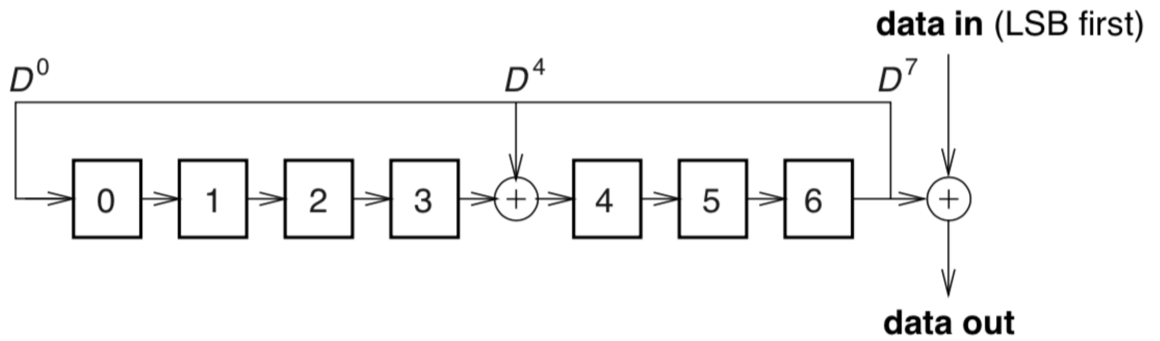

Figure 2.3: BLE packet whitening LFSR [24]. The LFSR is initialized for each transmission with a value known to both the master and the slave.

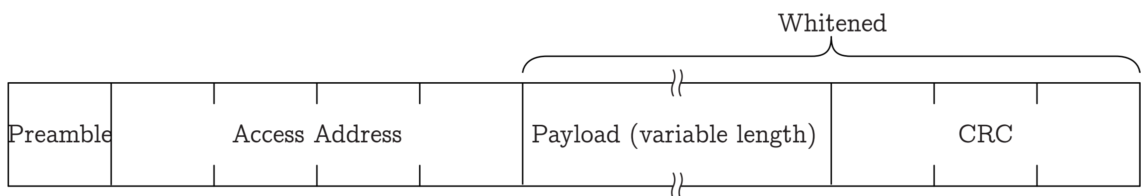

Figure 2.4: BLE packet structure. The payload can be any size from 2-257 bytes [1]. The Preamble and AA are not whitened because they are used for packet detection by a receiver.

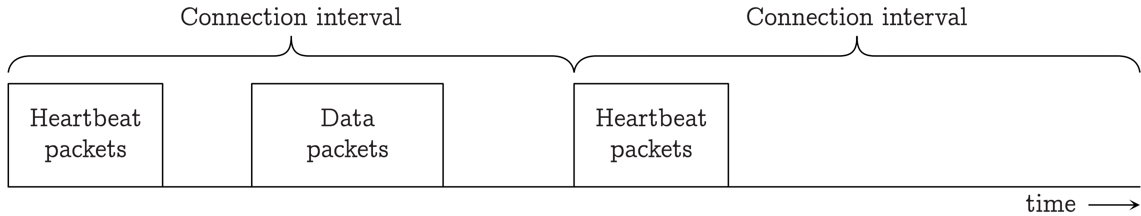

Figure 2.5: BLE connection interval timing diagram. Packets other than the heartbeat packets at the beginning of the connection event are not required.

Broadcast BLE packets are usually intended for a single device, either the master or the slave of the connection. Unlike many other master slave systems, a BT slave can have multiple masters and itself be a master for other devices. Each master and its slaves are referred to as a "piconet", and a group of interconnected piconets comprise a "scatternet" as shown in Figure 2.6. The master slave system primarily manages the Time-Division Multiple Access (TDMA) scheme, where the master tells the slaves which time segments they can use to communicate. BT connections can dynamically change the master/slave of a connection between two devices, which is necessary for some data transfer protocols [24].

BT uses frequency hopping across many channels to avoid environmental interference. Channel hopping is deterministic so the master does not need to inform the slave of the transition. Additionally, the hopping sequence is unique to each master/slave connection, not the entire piconet. The channel is changed every 625μs in BTC. For BLE, the channel changes every Connection Interval (CI), which can range from 75ms to 4s and is unique to each connection. A CI starts with a packet to and from the slave and may contain additional packets [24], as shown in Figure 2.5. These heartbeat packets are used to detect connection loss and manage other administrative functions of the connection. In adaptive modes, the master may mark channels as bad in the connection channel map due to low Signal-to-Noise Ratio (SNR) or high Packet Error Rate (PER), for example caused by interference from Wi-Fi or microwave ovens [9]. If a bad channel is chosen, a different good channel is picked instead. Depending on the BT version and channel hopping algorithm, an eavesdropper can learn the required parameters to follow a target connection with their own device and use targeted interference to cause the target connection to modify their channel map [3].

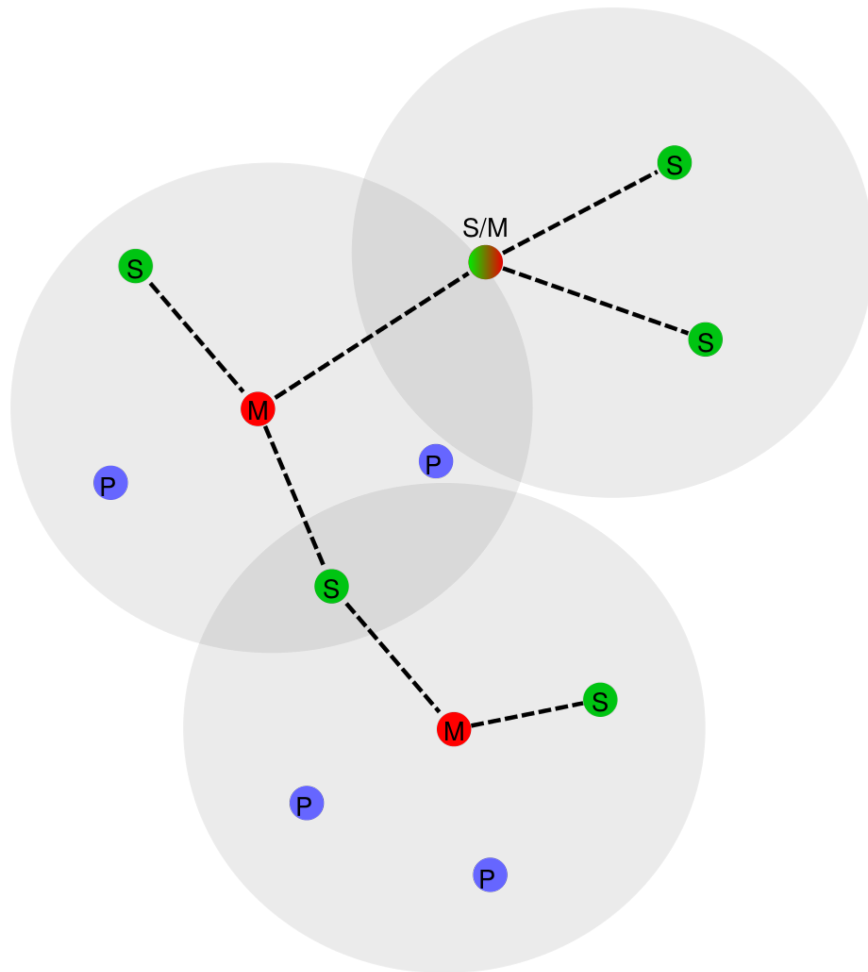

Figure 2.6: Depiction of a possible BT scatternet [8]. Some devices belonging to multiple piconets can act as both a master (M) and a slave (S). Nodes marked “P” are advertising-only nodes and are not relevant to this project.

2.1.3 Pairing Process

Before the master/slave direction can be established, the two devices must connect to each other in a process called "pairing". Pairing fulfills multiple goals, including establishing encryption keys, clock synchronization, and starting the channel hopping sequence [24]. BLE versions 4.0 and 4.1 use a pairing process very similar to older versions of BTC, while versions 4.2 and later use a new system aimed at minimizing security risks [24][16].

BLE 4.0 and 4.1 use a process called LE Legacy Pairing. At the beginning of the process the two devices exchange information about themselves including IO capabilities, which determine what key exchange system they use to transfer the Temporary Key (TK) and Short Term Key (STK) used for the rest of pairing. The simplest method is called Just Works (JW) [24], which requires no input capabilities on either device. This makes it the simplest pairing method, but it is also the most vulnerable. In this mode, the TK is set to 0, making the other keys relatively easy for an eavesdropper to brute force and spy on future connections. The second method is the Passkey method [24], in which the TK is a six digit number generated on one device and entered on the other device. Assuming the attacker has no knowledge of the passkey this method does prevent against Man-In-The-Middle (MITM) attacks unlike JW [16]. However, the additional key length of the TK is not enough to prevent an eavesdropper from brute forcing the keys used in the connection and decrypting future communications.

The final method is Out Of Bound (OOB) pairing [24], which uses a different communication system such as Near-Field Communication (NFC) to transfer the keys. An advantage here is the TK can be up to 128 bits, preventing brute force attacks. Assuming the OOB channel itself is secure, this process is secure against both MITM and eavesdropping attacks. Of the three legacy pairing options, OOB has the capability of being most secure. Potential security vulnerabilities for the different pairing systems are provided in Table 2.2.

Table 2.2: Comparison of security vulnerabilities in the different BLE pairing schemes. Pairing using LE Secure Connections is much safer than LE Legacy Pairing.

Version	Just Works	Passkey	OOB	NC
Legacy Pairing	eavesdropper/MITM	eavesdropper	secure	N/A
Secure Connections	MITM	secure	secure	secure

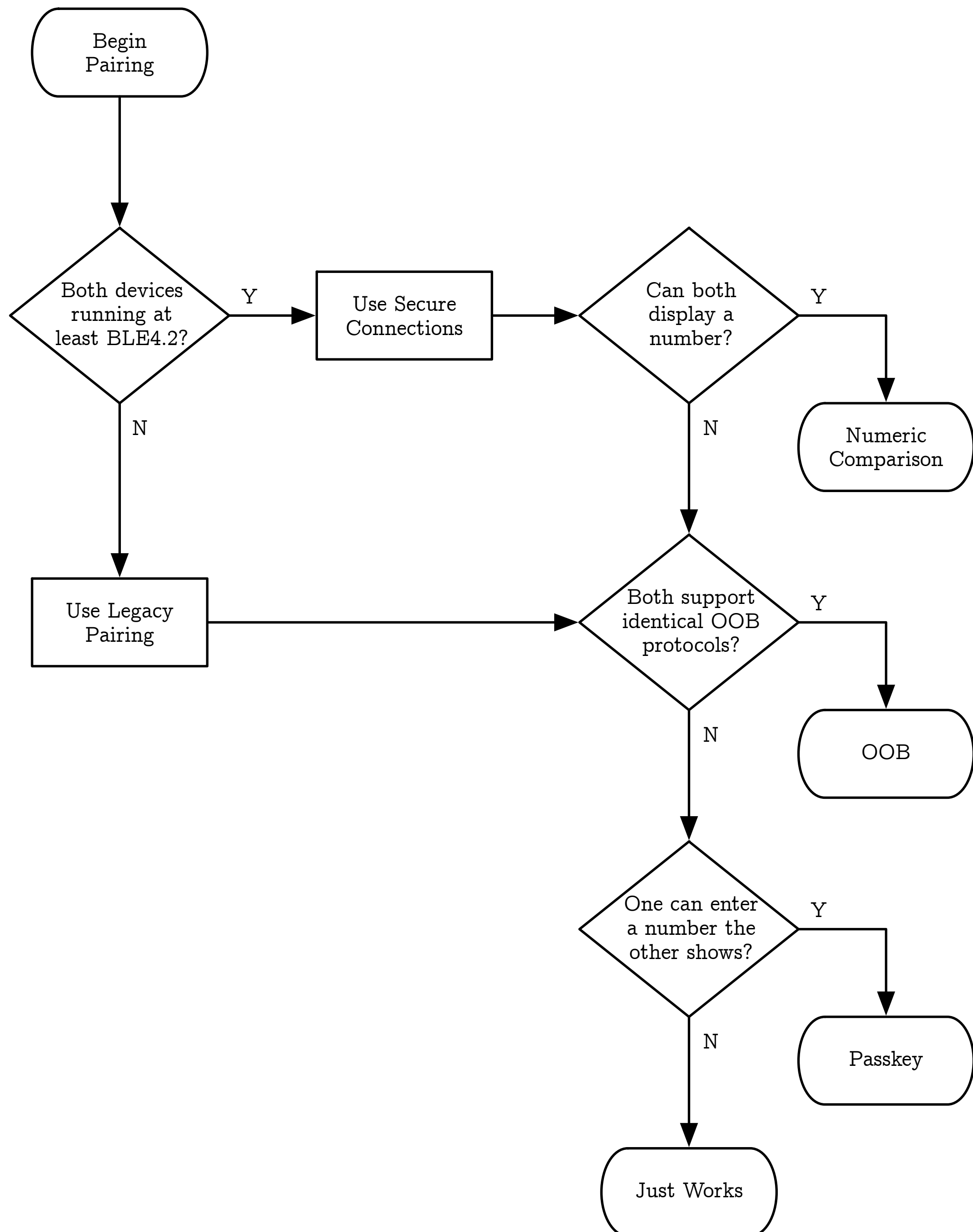

Figure 2.7: BLE pairing process selection. LE Secure Connections can only be used when connecting devices running at least BLE version 4.2. This process prioritizes more secure pairing methods, falling back to less secure methods when devices have lower functionality.

BLE 4.2 used a different scheme called LE Secure Connections, based on a single Long Term Key (LTK) derived from an Elliptic-Curve Diffie-Hellman (ECDH) authentication process [24]. The process still changes based on input capabilities and operates similarly to those used in LE Legacy Pairing. The pairing process selects a system to use by prioritizing security for a given set of device capabilities. The flowchart for process selection is presented in Figure 2.7.

The updated JW system does not add anything to the ECDH scheme, so it is still vulnerable to MITM attacks. However it is resilient against eavesdropping attacks due to the increased security of ECDH over the older system. All the other schemes add additional protections on top of JW for increased security. The updated Passkey method relies on inputting the same six digit number into both devices. The actual input digits are only used for preventing MITM attacks and do not have an effect on the final LTK. The updated OOB pairing system has no functional differences from the old system other than switching to ECDH. MITM resistance is again based on the other system used for the OOB data transfer. Finally, the updated system introduces a fourth option called Numeric Comparison (NC) [24]. Similar to Passkey, the system starts with JW and adds a confirmation to prevent MITM attacks. In NC mode, each device calculates a six digit number based on the exchanged LTK. The numbers are displayed on each end device and confirmed to be identical. Since a MITM attack would result in different LTKs for each end, the numbers would not match, thus preventing MITM attacks.

2.2 Sniffing Systems

When developing a wireless communications device it is important to have access to the data sent across the communications channel. Usually this is done by “sniffing” the link, meaning some device listens in on the channel to be studied and records what data is transmitted. Sniffing systems can be split into two categories: inline or promiscuous sniffers, as illustrated in Figure 2.8. A promiscuous sniffer is a third device that listens to communication between the device being tested and another device. The other choice is an inline sniffer, where the data across and about the link comes from a participating device

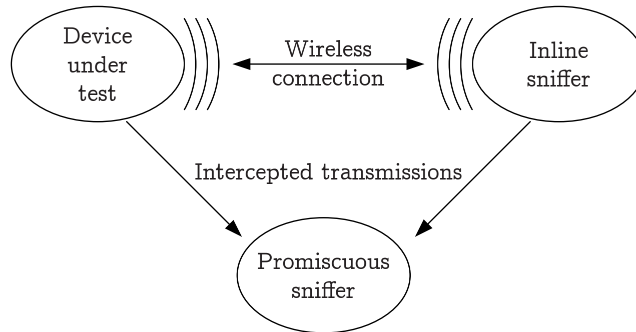

Figure 2.8: Promiscuous and inline sniffers in use. The promiscuous sniffer must listen in on a connection it is not a part of, while the inline sniffer must be part of the connection to be spied on.

you control. Promiscuous sniffing has potential advantages over in-line sniffing, namely the system can be designed to monitor multiple links at once and can offer more information about the wireless spectrum environment.

Few sniffers exist that are specifically designed for BT development, a selection of which are listed in Table 2.3. A couple commercial options exist based on BT chipsets that attempt to track an ongoing connection, but this type of sniffer has multiple issues in real world use. The first is finding the connection to be spied on. Due to the channel hopping used in the BT protocol, the sniffer needs to wait on a channel until it hears enough of the target connection to estimate the correct channel map and hopping distance. This introduces huge packet losses until the tracking starts, and packet drops when the predicted and actual channel jumps mismatch [3]. Depending on the product, it may miss the beginning of transmission windows due to channel switching latency [10][18]. Lastly,

Table 2.3: Comparison of commercially available promiscuous BT sniffing options [2]. Note the widely varying price points and that the LimeSDR is not BT specific.

Device	Type	Cost
Bluefruit LE	IC	$30
Ubertooth One	IC	$120
LimeSDR	SDR	$300
Frontline BPA	IC	$3,500+
Ellisys Bluetooth Tracker	SDR	$10,000+
Sodera BT Analyzer	SDR	$20,000+
Ellisys Bluetooth Explorer	SDR	$30,000+

these sniffers also have problems decoding the intercepted packets. The dewhitening is straightforward, but decrypting packets is largely impossible without the encryption key which may be unavailable [7].

The other commercial option for BT sniffing is Software-Defined Radio (SDR) based equipment. This approach has numerous advantages over using a BT IC. They are capable of monitoring the entire range of BT channels at once, meaning they do not drop packets by being on the wrong frequency, do not need the channel map, and can monitor an arbitrary number of connections at once. Additionally, SDR based equipment can be updated to support newer versions of BT as they come to market. However, SDR sniffers are not without disadvantages. They require much more processing power to handle the high data rates generated by the wide bandwidth. Additionally, they still suffer from the same encryption restrictions that the BT IC based sniffers have. Lastly, SDR based units specific to BT generally cost thousands of dollars [2], while the chip-based options cost up to a couple hundred dollars.

For BT development, inline sniffing is significantly easier to implement because it removes the need to actually follow the connection or deal with dewhitening and encryption as these are handled automatically by the BT controller. The major downside to inline sniffing is lower information availability. By the time packets are given to the host, the controller has already performed symbol identification and link layer processing, meaning raw packet data or other environmental factors are unavailable. A secondary issue specific to BT is the need for the in-line sniffer to be able to emulate multiple types of devices. Since applications of BT widely vary, the BT specification groups capabilities into "profiles" that together support an overall use case, such as an audio receiver or filer transfer service. Some devices such as laptops or cellphones support many profiles, but no single firmware image can support all possible profiles potentially required in a testing environment. This was not a problem for this project but needs to be considered for inline sniffing approaches to BT.

Figure 2.9: Picture of the CYW20719 evaluation board [21]. The micro-USB connection used to communicate with a computer is on the left, the buttons and user-controllable LEDs are in the bottom right, and the CYW20719 and built-in antenna are on the daughter board on the right.

2.3 Cypress Development Tools

A single Cypress chip model met project requirements, the CYW20719 [19]. The IC can act as a generic HCI device or use a custom serial protocol, referred to as "API" mode, that moves much of the host functionality onto the chip. Almost all of the relevant documentation for the chip was in a document named "WICED HCI UART Control Protocol" [22]. The majority of the document discusses the Application Programming Interface (API) mode communication protocol. The document also includes detailed information about the firmware loading process into both RAM and ROM. Most of the work conducted in this project involved writing programs based on information in this document.

Much of the work for this project was done using an evaluation board for the CYW20719 built by Cypress, which is shown in Figure 2.9. The board features a single CYW20719 IC, with both a built-in antenna and a connection for an external antenna. The rest of the board includes a USB interface so the chip can communicate with the development computer, power and reset buttons, and some LEDs controllable by the CYW20719 through

General-Purpose Input/Output (GPIO) ports. This project used the chip in API mode and turned on a debugging mode that pipes all the HCI messages out the main serial port as well. The disadvantage is all the physical and link layer packaging is missing, but that data is unavailable in all possible modes.

The Integrated Development Environment (IDE) used to program the CYW20719, named WICED, is a version of the Eclipse IDE modified by Cypress to include their APIs and example firmware images for multiple chip variants. The compilation and image loading process in WICED is well streamlined. If the evaluation board is plugged into the development computer and in the right mode, WICED automatically loads firmware images onto the CYW20719 after compilation.

Cypress provides a companion utility to its example images called ClientControl (CC) [20]. CC performs all the necessary communications over a serial link to the evaluation board to properly run the example images, in addition to offering firmware flashing capabilities. CC can also communicate with another provided utility, BTSpy. BTSpy can understand both API mode and HCI packets, turn them into a human readable format, and log the packets for later analysis, all of which is useful for CYW20719 development.

2.4 Data Formats

2.4.1 I32HEX Firmware Images

The firmware images produced by the compiler follow the I32HEX format as shown in Figure 2.10. Here, data is stored as a series of records in an I32HEX file. Records are a series of ASCII characters representing the data in hexadecimal format, and must be translated before use by an image loader. Each record has a header containing a 16-bit destination address and associated data. Multiple record types exist depending on what the data is, of which 3 are relevant to this project: memory offset, raw data, and End of File (EoF). Since the CYW20719 is a 32-bit device, the 16-bit memory address in each record is not sufficient to specify the destination. The memory offset records are used the specify the high 16 bits added to the address given in each following record. The data packets contain bytes to be written into memory, and the end of file record indicates the image is complete. Writing a

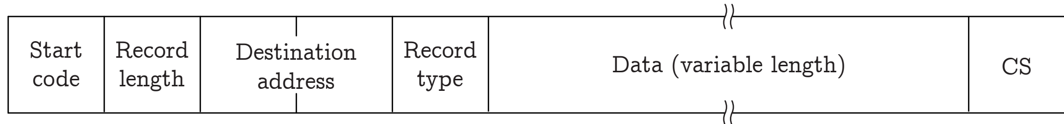

Figure 2.10: Representation of I32HEX record format. The data field is variable length, but was almost always 240 bytes long in firmware files made by Cypress.

firmware loader was an important part of this project because reflashing the CYW20719 is required for the final product to support multiple BT device types.

2.4.2 Cypress HCI Packets

The original intent of the inline sniffing system was to capture and record each bit that was sent or detected by the test antenna. Unfortunately, the CYW20719 does not support transferring raw antenna bits outside of the chip [19]. Cypress was contacted to determine if any additional capabilities existed that might be used, but there were not any options. The data available from the chip is either HCI packets or Cypress API packets [22]. HCI packets are two of layers removed from the antenna in the BT stack which obscures activity in the physical or link layers but is otherwise a detailed log of events in the BT link. The non-HCI Cypress packets use a format unique to Cypress, illustrated in Figure 2.11, and decoding them is not supported by Wireshark [28]. This project only used the HCI packets from the CYW20719, since the Cypress format packets were discovered to be redundant as the same information appears in the concurrent HCI packets.

The BT specification provides multiple options for how data transfers over the HCI are completed, chosen based on the transport layer protocol used. Most of the differences between protocols are in prepended flag bytes. Two protocols are relevant to this work: HCI H1 and HCI H4. H1 has nothing added to the HCI packets, and H4 has one flag byte

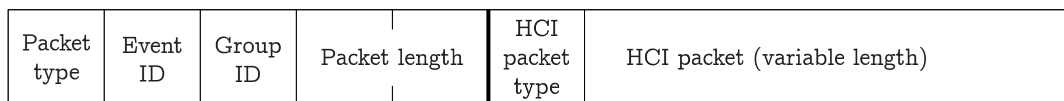

Figure 2.11: HCI packet inside of a Cypress API wrapper. Note the flags in the HCI packet type field do not match those required for HCI H4 format.

indicating if the packet is a command from the host, an event from the controller, or one of two other specific data types. Since Cypress wanted the same packet format to work for both their API mode and transferring HCI packets, the HCI packets are packaged into an API mode wrapper similarly to but are incompatible with HCI H4. The HCI packet format itself is specified in the BT Core Specification but did not have much of an impact on this project since Wireshark has built-in decoding capabilities.

2.4.3 BTSnoop

Once the HCI packets are extracted from the Cypress wrapper, they need to be packaged into a packet capture format for analysis in another program. A format called BTSnoop (BTS) is specifically used for BT HCI packet captures. It is the standard for BT logs created by the Android operating system and is supported by Wireshark [14]. Figure 2.12 shows a visual representation of the BTS file format. A BTS file consists of a file header followed by packet records, each of which contain one HCI packet. The file header contains an identification pattern common to all BTS files, making file type identification easy and independent of filename extension. The header also contains a BTS version number and a datalink type, identifying which HCI transport layer flags are included in the packet data. The packet records are structured as original packet length, included packet length, packet flags, cumulative packet drops, timestamp, and packet data [12]. Original and included length both measure the packet, not the record, with original representing as-transmitted and included as-recorded. The optional difference in length allows shortening of large packets if log file size is a concern. The packet flags indicate if the packet is a command or data and packet direction. Cumulative drops counts the total number of missing previous packets in case the logging system is slow or runs into errors. Timestamps have microsecond resolution which should be enough to avoid two packets having the same timestamp. BTS does not specify that packets must be sorted by timestamp, but out of order packets may cause errors in interpreters. After the timestamp is the packet itself, including any prefixes required by the packet packaging format. Integer data fields are declared to be big-endian, which avoids ambiguity when moving data files between system with different endianness.

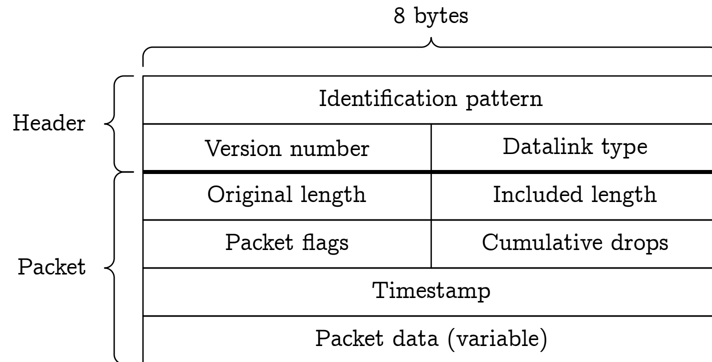

Figure 2.12: BTSnoop file format. Note the data fields are arbitrary length and not aligned to any specific byte boundary.

The BTS format supports both HCI H1 and HCI H4 packets. When using BTS in H1 mode, the file still needs a flag byte indicating packet direction and if the packet is a command/event or some kind of data. This can be translated from the H4 flags, but calculating the H4 flags from the BTS H1 flags is not possible. When the CYW20719 sends out HCI debug packets, the API wrapper only includes the H1 flags. Interestingly, the API format gives the logical inverse of the flags needed by the BTS format, possibly because it is simpler to implement or matches other flag systems already used in the API.

2.4.4 PCAP & PCAPNG

The most widely used format for storing captured wired and wireless packet data is called Packet Capture (PCAP) [27], which is shown in Figure 2.13. Generally, these files are opened with the networking utility Wireshark, which supports a wide range of useful network analysis features. PCAP is not without flaws - it is an older standard that does not support improvements as easily as newer standards and does not support multiple protocols in the same file [27]. That feature is built into PCAP Next Generation (PCAPNG) as one of the improvements over the older standard [17] (see Figure 2.14).

The PCAP file format is largely similar to BTS, with a global header followed by individual packet headers and packets. PCAP file headers are more complicated because the PCAP format is designed to support many different packet types. The global header starts with a Magic Number, primarily used to identify the file type as PCAP. Since the

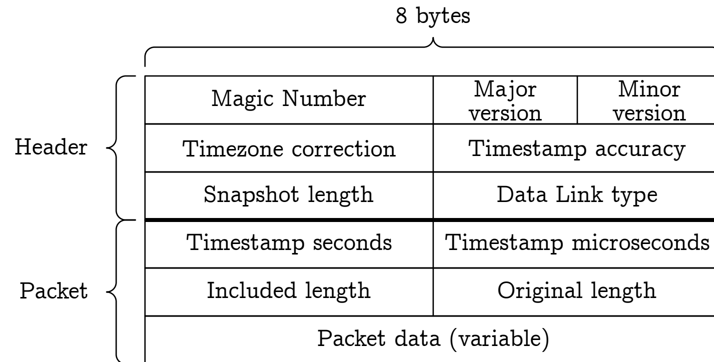

Figure 2.13: Minimum PCAP file. Packet data is variable length and not padded to any specific size multiple.

endianness of fields is not declared by the format, the Magic Number is also used to identify file endianness. Lastly, it also differentiates the more common microsecond resolution files from the nanosecond resolution ones. Following the Magic Number is the PCAP version information along with timezone correction and timestamp accuracy, neither of which are used in practice. The end of the global header contains a packet overall length limit and the link type information, which in this research is equal to BT HCI H4. The packet headers themselves contain timestamps, which are split into seconds and microseconds or nanoseconds, as well as the included and original packet lengths as in BTS.

The PCAPNG format was designed to improve upon the PCAP format, especially through design with future proofing and to remove weaknesses in the older standard. The official goals are “extensibility”, “portability”, and “file merger”, meaning new capabilities should be added in ways that do not break old interpreters, a file should contain all necessary information to interpret the file, and data can be appended to a file without issue [17].

A PCAPNG file starts with a Section Header Block (SHB), which describes blocks between it and the following SHB should one exist. A SHB starts with a block type identification, followed by the block total length and “Byte-Order Magic” used for endianness identification. Data endianness can change mid-file because it depends on the computer creating the section and sections can be appended to existing files. After these identifiers are the major and minor version numbers as well as the total section length. Version numbers are used by the interpreter to decide if it can parse the given section - different sections can

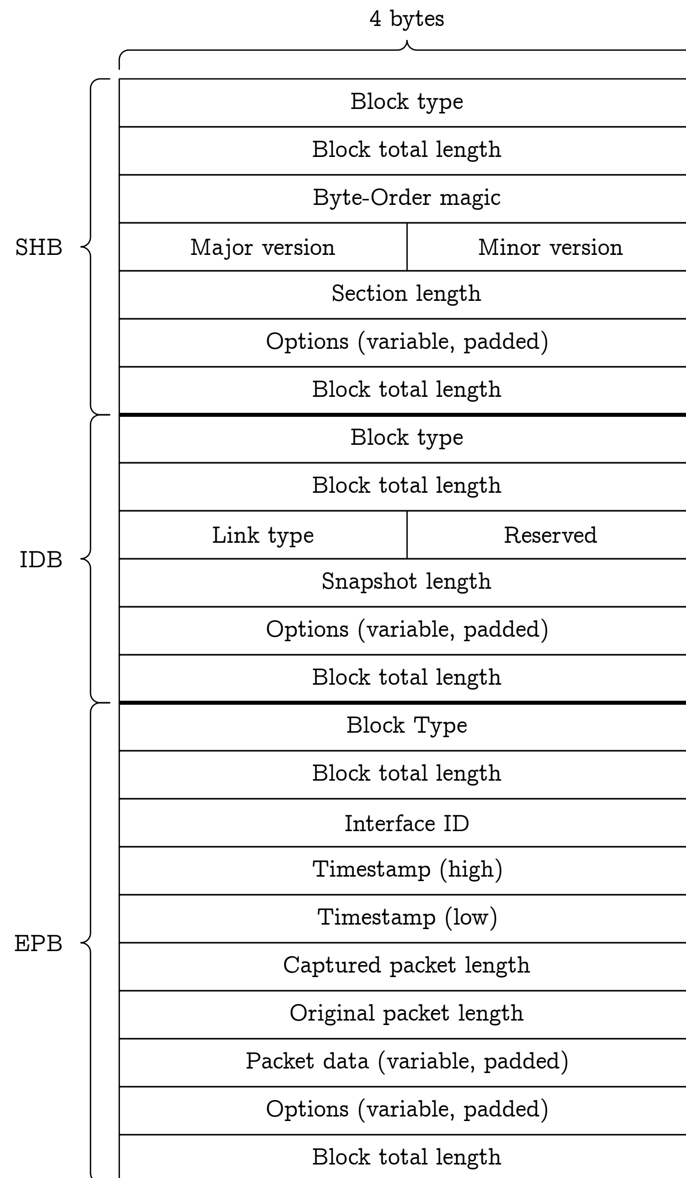

Figure 2.14: Minimum PCAPNG file as used in this project. The options fields are unused, and packet data must be padded to be a multiple of 32-bits long.

have different versions within the same file. Section length can be used to skip the entire section if it is not parsable, although it may be set to indicate the size to be calculated from the contained blocks. Several optional fields may be present, followed by a repetition of the block total length. All blocks end with the block total length to allow for easy reverse seeking through the file. Additionally, every field in the file is aligned to 32-bit boundaries. This does require padding of variable length fields such as packet data with zeros, although this greatly simplifies block identification and searching the file in both directions.

Following the SHB is an Interface Description Block (IDB), which is analogous to the file header in PCAP. The IDB contains a link type identification, HCI H4 in this project, and total packet size limit for associated blocks. Each section can have multiple interface descriptions, allowing for mixed technology files combining both Wi-Fi and BT data [17].

Enhanced Packet Blocks (EPBs) follow an IDB. They contain a reference to the associated IDB, timestamp information, and original and included packet lengths. The packet lengths do not include the length of padding required to align with a 32-bit boundary. The amount of padding is not required to be the minimum possible, so some interpreters might have issues finding the beginning of the optional data fields after the packet if any are included. Other block types are defined but not relevant to this work.

PCAP and PCAPNG do not support HCI H1 formatted packets, only H4 and other more complex systems. Switching from BTS to PCAPNG caused problems with the required packet flags since the API format flags do not exactly correspond to H4 flags. A collaborator created a method to accurately determine the H4 flags, which will be presented in his thesis.

2.5 Summary

This chapter presented relevant background info on BLE terminology along with data formats and development tools used throughout the rest of the project. Even though little of this project directly developed BT software, the overall structure of BT communications both informed and guided the direction of the project. The next chapter will focus on the first, and largest, section of the project: creating and testing a BLE in-line sniffing system demonstrating a file transfer throughput test.

Chapter 3

File Transfer Capabilities

This section presents a hands-on demonstration of simultaneous BT communication and sniffing capabilities using the system described in Figure 3.1. The demonstration uses a device under test, in this case a smartphone, a commercial BT IC running customized firmware, and a computer acting as a controller and ultimate data destination. During the demonstration, the smartphone negotiates with the BT IC to transfer a file, which is then sent to the computer alongside other data for further analysis. This part of the project successfully shows creation of a custom software interface between a computer and the BT IC, collection and logging of packets in real time, and data analysis after packet logging.

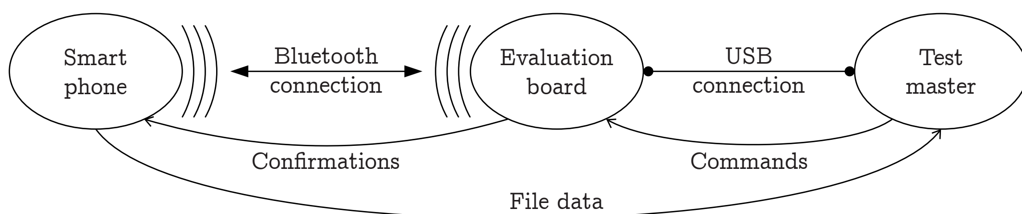

Figure 3.1: Overview of file transfer architecture. The CYW20719 largely acts as a interface between the test phone and the test laptop.

3.1 Reverse Engineering the Evaluation Board System

The first step was to generate a packet log based on data received from the BT chip itself. Project requirements did not constrain what kind of firmware needed to be running on the CYW20719. The startup guide for the evaluation board included instructions for how to generate human readable packet logs using CC and BTSpy. The guide was followed and the BTSpy output contained the same information the log needed to contain. It was unknown at the time how CC and BTSpy communicated with each other or which application was translating packets that came over the serial port into a human readable format. To try and figure things out, two different terminal emulators (PuTTY and TeraTerm) were used instead of CC to open the serial port with the evaluation board. TeraTerm did not support the baud rate normally used by the CYW20719 (3M), so the slower alternate speed (115200) was employed. Once the change was made, both terminal emulators created logs of the bits transmitted over the serial port instead of the human readable format BTSpy returned. Not surprisingly, BTSpy did not work with the terminal emulators. It was determined that CC sent BTSpy data over an internal port and translation was split between them, thus making a terminal emulator work with BTSpy infeasible.

Becasue the terminal emulators did not communicate with BTSpy, obtaining the same log in both a human readable format and serial characters for comparison was very difficult. Given the firmware image being used did not require any user input, it was possible to get logs from BTSpy, TeraTerm, and PuTTY separately that were similar enough to begin reverse engineering the protocol used between the BT chip and the computer, along with a translation into a human readable format. A python script was created that opened a serial port with the evaluation board and sorted packets based on parts of the packet header. Eventually, a coworker found documentation from Cypress detailing the same control protocol being reverse engineered. This documentation facilitated a return to test development using the BT chip because packets to and from the device could be easily interpreted.

3.2 Throughput Test Using ClientControl

After discovering the protocol documentation, work commenced on a throughput test. It was decided to start with a firmware image that emulated a keyboard while a collaborator worked withe other firmware images. Loading the keyboard example image onto the evaluation board did not work as intended. The associated documentation revealed that the firmware image expects to be run on hardware that appears to never have been manufactured. The documentation also gave instructions for how to run the image on the evaluation board. Keystrokes were successfully sent from the evaluation board to a second device after working out some errors in the instructions. However, The process was tied to using CC, which would not work as a long term solution.

Designing the throughput test system without CC started with a collaborator's discovery: an example firmware image written for a different chip variant demonstrating Object Push Profile (OPP), used to transfer file between devices, could easily be modified to work with the CYW20719 by copying some API libraries from the development environment of the other chip to the CYW20719 environment. The required changes were performed and the firmware was loaded onto the evaluation board. Fortunately, CC did not recognize that the firmware was running on unintended hardware. As a first step, correct operation of CC interfacing with the OPP firmware was successfully demonstrated. To do this, the firmware image needs to be loaded onto a CYW20719 and started. CC will automatically identify which example firmware is running. In OPP mode, a button is clicked in CC to enable the test and wait for a file transfer. The evaluation board acts as a data sink for OPP file transfers - often used to transmit images or contact VCards. Using a phone as a data source, an image or a VCard was selected to transfer to the development computer and the evaluation board was selected from the BT sharing options shown in Figure 3.2, initiating the transfer. The evaluation board sends all the data over the serial link to CC, which saves the transferred file to the computer and displays the total amount of data transferred and elapsed time. The results of a successful transfer are displayed in Figure 3.3.

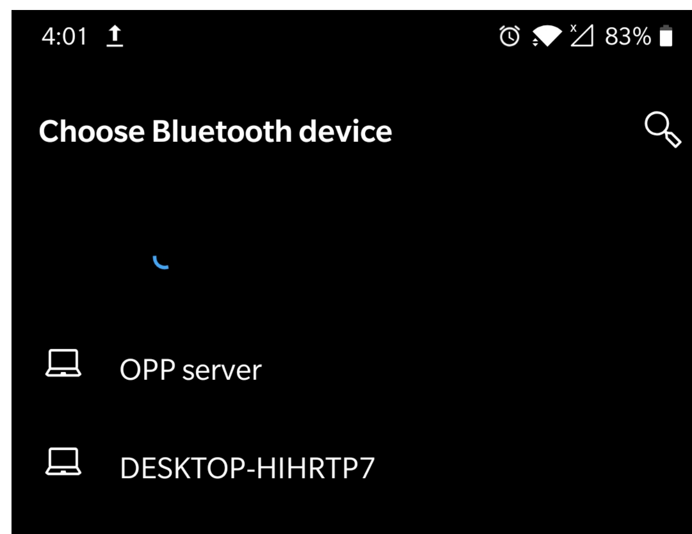

Figure 3.2: Screenshot of BT sharing options on an Android phone. The evaluation board appears as “OPP Server” in this image. The specific text displayed on other devices is configurable by changing a field in the firmware source code.

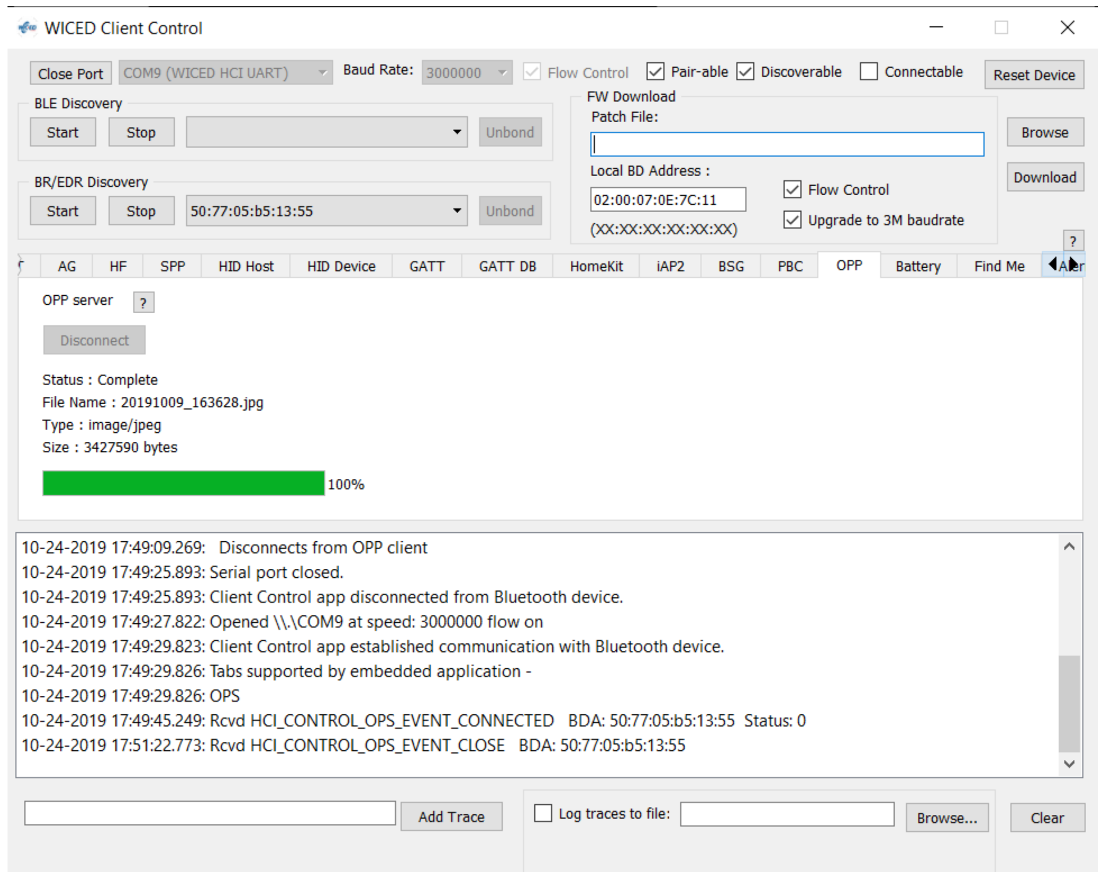

Figure 3.3: Example CC screen at the end of a file transfer. The name of the file and size are visible, while the elapsed time can be calculated from the last two messages in the log view.

3.3 Throughput Test Using a Python Script

After demonstrating file transfer capabilities using the provided software, the next step was to duplicate the necessary parts of CC in a custom python script. Since the serial protocol definition was available, it was decided to start by spying on the serial link between the CYW20719 and CC. The communication was known to be bidirectional, since the evaluation board was invisible to the test phone without using the utility.

To spy on the serial link a tool called Advanced Serial Port Monitor (ASPMON) was used. Other utilities such as "Free Serial Analyzer" and "Serial Port Sniffer" were available but were incapable of spying on the link since Cypress uses an uncommon speed of 3M baud [26][22]. Using the slower alternate speed was tried but found to introduce significant errors during file transfer even when using CC. Hardware link duplicators that accomplish the same task are available for serial links over RS-232 cables but do not support USB, which the evaluation board uses. Since everything over the link is the Cypress protocol instead of something more common, some trial and error was needed to set up ASPMON properly and force it to ignore what it thought were control characters.

Once a log file was successfully created, translation of Cypress protocol into a human readable format commenced. The beginning of the log was CC identifying what firmware image was running on the connected BT chip. Once that was determined, CC sent commands to enable visibility to other BT devices and accept incoming connections. This explained why the evaluation board did not appear on the test phone without using CC - the program starts in a non-advertising state and needs to be told to switch over. At this point the first indication that the Cypress protocol documentation [22] was questionable started to appear - some packet definitions did not match what was actually being sent, and entire groups of packets that appeared in the log were not described anywhere in the protocol definition document.

Implementation of the interface and translation program, named PyControl, was performed in multiple stages, with the first stage focused on configuring the CYW20719 to advertise and accept connections. Several required commands were added to the packet translator written during the reverse engineering process. Once added, the CYW20719

would accept connections and appear to accept file transfer requests. However, the process would stall with no data transferred but never actually fail on either end.

The VCard transfer logs were inspected to find the missing part of the puzzle, one area of the log looked promising but the command was missing from the protocol definition document. Cypress support was contacted informing them of the missing information and other issues discovered with the document, asking if a newer version was available with the errors resolved. Cypress support indicated that they were working on a new version but it would not be available for this project; instead they highlighted a lookup table in some source code used by firmware images to translate packets. While not ideal, this provided enough information to determine that the untranslated section was the CYW20719 telling CC of the incoming file transfer and asking for permission to continue, which CC confirmed. The confirmation messages were added to PyControl and the file transfers began to move data.

3.4 Recreating the Transferred File

The filename is sent over as part of the transfer request packet and used to name the file created on the test computer. Once the file transfer is confirmed, the file data itself comes over the link contained in HCI packets. To reconstruct the file itself, the data portions of the HCI packets are written into a file in the order they are sent. VCard transfers were used to determine the data portion of the HCI packets. An entire VCard fits within a single HCI packet, allowing total header and footer lengths to be determined so those sections can be separated from the file data. Much of the early testing was conducted using VCards and very small JPEG images since the effective data rate is so slow - under 400kbps in good environments and even slower with increased interference from other BT devices or Wi-Fi. This makes testing code iterations very time consuming when testing with files larger than 1MB. Once the smaller transfers were working consistently, testing started on larger files on the order of 4MB. These larger file sizes introduced a new error: the packet parser would randomly become misaligned to packet edges on the serial port and fail until the serial buffer was flushed. Errors generally occurred only on files 2MB or larger and were

not consistent test to test, as well as being frequently bunched together. It was suspected that these errors were caused by the serial buffer between the evaluation board and the control computer overflowing and losing a packet section.

3.5 Resolving Serial Buffer Overflow

Completely filling the serial buffer causes loss of data until buffer space is available, at which time the buffer starts accepting data again. If the buffer starts accepting data in the middle of an incoming packet, the parser does not know how to interpret the data since it is not a packet header, causing further loss of data. It was unclear why the buffer was overflowing. Originally it was thought to be caused by long processing times in python since it is an interpreted language. Significant inefficiencies that were part of the original packet translator were removed from the code such as unnecessary type conversions performed on all incoming bytes. Determining the effectiveness of any change was difficult due to the unpredictable nature of the errors, but these changes did not appear to improve the situation. Removing code that relied on user input and increasing the size of the serial buffer was tried, although it did not make any apparent changes. Increasing the size of the buffer may have been a slight improvement but the API call used for this change is limited to Windows systems and could not be the final solution.

Eventually it was determined the issue was related to hard disk latency on the project laptop, which had a 5400rpm hard disk. The code wrote every packet to the output file and log individually. Additionally, the project was saved in the a Dropbox folder at the time, and Dropbox attempts to stay current to every file update. These factors combined to cause 99% hard disk utilization for the entire time my script was transferring a file, as seen in Figure 3.4. If the operating system added in any disk access calls to the queue due to background tasks, PyControl needed to wait the additional time for the file write call to clear since python cannot continue until the write is complete. This additional time caused the serial buffer to fill with packets because the OPP firmware cannot delay data transfer. The modified PyControl was switched to cache everything in system memory and write it all out as a block at the end of the transfer. Transfers are effectively limited to under

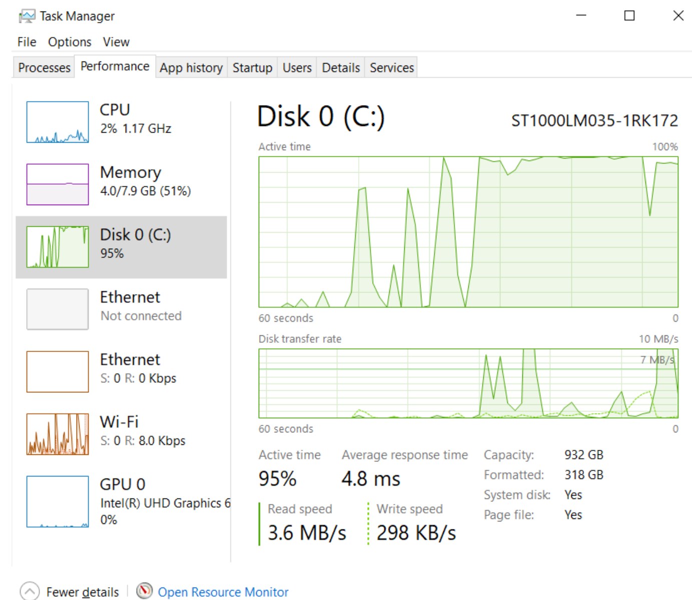

Figure 3.4: Screenshot of the hard disk latency. Note the extremely high utilization since the test started roughly 40 seconds previously. Average response time sporadically exceeded 50 milliseconds, representing a huge delay in the python code.

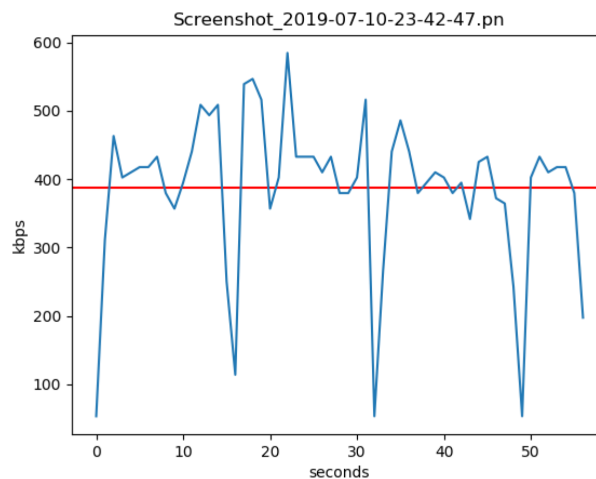

Figure 3.5: Throughput graph output of PyControl transferring an image from the test smartphone. The occasional dips below 150kbps are caused by hard drive latency spikes.

Table 3.1: Reliability testing data of the throughput test using PyControl. No errors that occurred were caused by PyControl, only the smartphone used during the tests.

File Size	10kB	9.7MB
Trials	20	20
Successes	12	15
Failures (PyControl)	0	0
Failures (Smartphone)	8	5
Average Time (s)	0.426	282.1
Throughput (kbps)	188	275

10MB due to slow speeds, which will easily fit inside system memory without issue. After implementing this change, buffer overflows were no longer experienced.

Reliability testing was conducted to ensure no remaining issues existed in PyControl that would negatively impact the throughput test. Testing was conducted using both small and large file sizes to ensure no more size-dependent errors existed. Testing data is shown in Table 3.1. Unfortunately the alternate smartphone used to create this data exhibited multiple failures, but no recorded failures were caused by the PyControl system. All trials fell within ±10% of the average time for the file size and data rates are comparable to transfers completed with ClientControl for large files, between 250-300kbps. The low measured throughput for small files may be caused by a lack of timing data points or a small measurement offset amplified by the shorter timeframes instead of a true throughput difference.

Once PyControl was confirmed to operate without errors and create log files properly, testing started using a custom octoScope board with two CYW20719 chips on it. Switching hardware did not present any issues, and a live demonstration was created showing successful file transfer, logging, and throughput calculation and graphing capabilities (shown in Figure 3.5). The demonstration itself went smoothly, except for an issue with the filename of the transfer. It turns out that the alternate smartphone puts space characters in some filenames, which the python script does not handle properly. Once a different file was selected, the demonstration succeeded without any further errors.

3.6 Summary

This part of the project successfully demonstrated the critical parts of an inline BLE sniffing system. In spite of many roadblocks encountered during the prototyping phase, the final system implementation creates a BLE link, and arranges a data transfer while simultaneously logging packets for further analysis and calculating throughput measurements. Future work should expand on what is described here by adding support for other BLE profiles and relevant metrics, such as an audio profile and measuring received audio quality. The next chapter focuses on the second part of this project, a custom firmware loader for the CYW20719.

Chapter 4

Firmware Loader

To support testing multiple types of BT devices, the CYW20719 needs to run different firmware images. Since the chip can only store one image at a time, a new way to load a the image onto the chip is needed. Cypress provides multiple utilities to do this, but they were either limited to Windows or had a GUI, which did not meet the project requirements. As a result of this, a custom firmware loader based on instructions from Cypress was developed.

4.1 Firmware Loader Development

The next task in this project was the creation of a custom firmware loader. Cypress provides this functionality built into the WICED IDE, CC, and a command line utility, but these are not portable solutions. Consequently, work was started on the firmware loader based on a description of the firmware loading process in the Cypress protocol document [22]. Implementation was fairly straightforward since the documentation was easy to understand and had good coverage in that section. However, initial attempts using the preliminary script gave errors which depended on the image being loaded.

Firmware loading would consistently fail at a seemingly arbitrary record depending on the specific image. After some minor code changes, the loader always failed at record number 52 no matter what image was being loaded. At this point, Cypress support was contacted for assistance. After a few weeks working on other tasks with no progress, it was decided to compare the firmware loader of this project to the firmware loading process performed by

HCI header	Packet length	Address (actual)/[sent]				Data (variable length)
		(0x12	34	56	78)	
		[0x34	12	78	56]	

Figure 4.1: HCI packet format for firmware load packets. HCI commands sent to the CYW20719 do not need to be wrapped in an API header.

the WICED IDE. ASPMON was used to capture a failed attempt using the custom loader and a successful image transfer using the IDE. While comparing the two, it was realized that the memory addresses in the successful transfer were in little endian format, which was not considered as an option since every other interaction with the chip had been big endian format up to this point, including the memory addresses in the firmware image created by the compiler. Little endian memory format causes the write RAM commands making up the majority of the firmware loading process to have a very strange arrangement: big endian header, little endian destination address, big endian data. Due to how the firmware is stored in the image file, the 32-bit destination address is split into the high 16 bits and low 16 bits. According to the file transfer specification, the destination address is sent over as high bytes followed by low bytes, which itself is not little endian. This ultimately causes a destination address of 0x12345678 to be transferred over the serial link as 0x34127856. The memory address structure, inside of a larger packet, is shown in Figure 4.1. It is unclear why it works this way.

It was determined that the loader consistently failed at the same record was because it was attempting to write to an invalid memory address, which the chip ignores and does not respond to. Because all records are the same size and all image files start at the beginning of flash memory, the script always jumped out of memory and failed at the same record.

The memory addressing issue was fixed by adding a method to swap the two bytes of each 16-bit address field before the packet is sent to the CYW20719. After the fix, the firmware loader claimed everything was working and the image was running on the board when it completed. However, scanning for Bluetooth devices using the test phone revealed that the chip was not running the new firmware image but continued running the previous one. While comparing the new code transfer to the successful one, two remaining differences

were discovered: the IDE was using an optional minidriver, and there was an undocumented command being sent to the chip by the IDE before loading the final image.

Cypress documentation claims that optional minidrivers are used when the base chip is not capable of writing the image, usually caused by storing the firmware in off-chip flash. Since this was not being done for this project, it was originally decided to ignore the minidrivers entirely. However, it was then decided to use it to minimize the differences between my implementation and the IDEs. Adding that in was very easy - the minidriver was stored in a folder in the IDE and it was in the same format as the regular firmware images such that it could use all the same loading code.

The only remaining difference between the two implementations was the undocumented command. However, the custom code was claiming it got the correct execution started response from the board. It was initially thought that a command towards the end of the loading process was missing since clearly the firmware was not actually executing. The issue here was that a trial version of ASPMON was being used which had a total character count limit; the final steps of the firmware loading process could not be observed because the size of the minidriver combined with the firmware image was too large. The IDE files were reviewed for a much smaller image to test with such that the entire process could be observed. While looking for a suitable image, a log of the IDE loading a firmware image was discovered. After loading the minidriver was the line "Transmitting 'Chip erase' command 01 CE FF 04 EF EE BE FC". The hexadecimal command was the same as the undocumented command in the serial log. Once the chip erase command was added to the loader it successfully loaded a new firmware image and began execution. The image survived power cycling the eval board, proving the firmware image was written to flash memory. In spite of running in python, the code loads firmware images faster than the IDE, largely because the code does not perform any integrity checks after writing the image. This is not an issue in this system because the chip images will be frequently updated so it can be reloaded easily if there are any errors in the process.

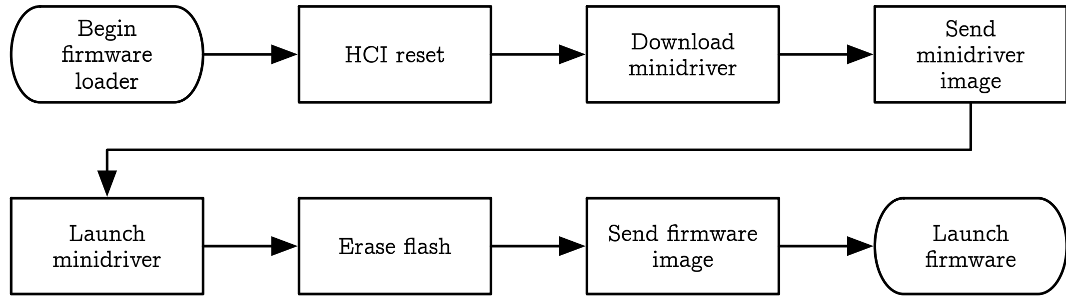

Figure 4.2: Flowchart of the firmware loading process. If any of the steps fail the solution is to power-cycle the eval board and try again.

4.2 Firmware Loading Process

The firmware loader script is simple as there is not much variation in what can happen during the loading process. An overview of the process is shown in Figure 4.2. The only complicated part is finding the correct serial port to communicate with the chip. Each chip has two serial ports: one for HCI and Cypress API messages and one called the "Peripheral UART" generally used for developer debug messages. The HCI port must be used to load firmware, so the script first looks for a port name starting with "WICED HCI UART", as used on the eval board. Two CYW20719s are installed on the octoScope board used in this project and the USB serial interface on the board does not preserve port names. Since the four ports are always consecutive, the script looks for four ports in a row and connects to the lowest port number. This always hits the HCI port for chip one because the order does not change. Limiting ourselves to only using one chip has not been a problem so far, but future versions of this code should have a more robust port selection scheme. Should both previous port selection methods fail, the script attempts to connect to the lowest available port number - this was added to support working with other hardware versions.

Once the serial port is opened, the loader script sends the "HCI reset" command as defined in the Bluetooth standard and waits for a response [24]. The script assumes the chip is expecting a connection at 3M baud. Generally this is true, since all the firmware images used in this project set that as the baud rate for all ports. If the chip does not have a firmware image at boot it defaults to autobaud mode, where it attempts to identify incoming HCI reset commands and select the same baud rate. The chip responds to very

few messages in autobaud mode, but fortunately behaves as expected to the HCI reset command [22].

After the chip sends a reset confirmation response, the loader script sends the "download minidriver" command. Interestingly, the documentation claims downloading a minidriver is optional after sending this command and the sender can skip directly to sending the final firmware image, for which there is no separate command. It is suspected that Cypress named the command for ease of understanding and not necessarily to reflect function, as it acts as a "expect write RAM commands" command.

At this point the behavior of the loader script differs from the firmware programming interfaces published by Cypress. First, the reset command does not succeed if the script is run within the first 10-15 seconds after plugging the eval board in. The IDE and CC may act similarly but it is difficult to test these as they both have long wait times before processing a firmware load. It is suspected this has to do with the boot process but this is a much longer time window than expected as normal boot times are less than 5 seconds. A second issue was also encountered in testing. After plugging the eval board in, the script can be run repeatedly until it passes HCI reset. The first time it passes reset, it will always pass download minidriver. After it has passed download minidriver once it will always fail there until power cycling the chip, whether or not a firmware image was loaded and no matter now many times it passes HCI reset. The official programs do not appear to exhibit this behavior, it is suspected this is caused by interactions with autobaud mode that the custom loader was not designed to handle but the official utilities can manage.

After passing download minidriver, the script starts processing and sending over the minidriver image. Data records in the minidriver firmware image usually contain 16 bytes of data each. Each data packet sent to the chip requires a confirmation response within 200ms. Originally, the serial port was set up with no timeout and the script was forced to wait the entire response period before checking the serial port. This worked but was much slower than the official utilities. The port timeout was set to slightly longer than the largest specified delay period and the script sleep time was removed. Most write responses come within 10ms, representing a significant improvement in processing times. The official utilities send over 240 bytes of data per packet, requiring fewer confirmation packets than the custom

script that sends each record individually. This project does not do this as the larger packet sizes require merging record segments together and recalculating the destination memory addresses. This change would speed up both the minidriver and final image loading, but without this the loading process takes less than five seconds to complete, so adding in the improvement is not worth the increase in code complexity.

After downloading the minidriver and sending the command to boot into the minidriver, the official applications switch to 115200 baud and send a command to switch back to 3M baud. Through testing it was discovered the baud switching is unnecessary. It is unknown why the official utility does this, other than potentially to deal with a minidriver that expects 115200 baud. After the loader script sends the "boot minidriver" command, it sends the "erase flash" command followed by the actual firmware image itself. Loading the firmware uses the same process as the minidriver, the only difference being the firmware is loaded into flash memory not RAM like the minidriver. When the image is complete, the script sends a command to reboot into flash memory and the image begins running.

4.3 Summary

This chapter presented the successful development of a custom firmware loader script for CYW20719 devices. In spite of both errors and missing information in the documentation, the firmware loader works as expected under normal conditions. Future work should investigate the situations where this utility operates differently from those provided by Cypress, as that is the major remaining unknown for this part of the project. The next chapter will examine the final part of the project, integration of a SDR-based packet decoder based on a commercially available SDR from Litepoint.

Chapter 5

Promiscuous Sniffing Using a Litepoint SDR

The Litepoint IQxel-M16W (IQxel) offers the ability to view the radio environment in the ISM band in the chamber while a test is running. The front panel of the IQxel is shown in Figure 5.1. It also has the ability to store a time slice of the waveform and can decode packet data as-transmitted from the waveform itself when configured correctly. OctoScope wants to use the IQxel to fill in gaps the HCI logs may have.

Figure 5.1: Image of Litepoint IQxel-M16W [13]. Each of the ports on the front can be configured as an input or an output and monitored separately.

5.1 Packet Capture

Work on the IQxel began with a web interface hosted on the device itself. With assistance from a coworker, the IQxel was configured to automatically recognize recognized where packets were in time. However, most packets could not be decoded and the IQxel thought the ones it could decode were BTC packets, when the input was known to be BLE advertising packets. Additionally, the eye diagram of the decoded packet (see Figure 5.2), a representation of the shape of the signal, revealed something was set incorrectly. That turned out to be the Automatic Gain Control (AGC) clipping the signal severely, since it was setting the reference level to just above the noise floor of -40dBm instead of the signal level of +15dBm. The signal level of +15dBm was much higher than expected, in part because at this point in testing the evaluation board was plugged into the IQxel input directly using the optional second antenna connection instead of attaching an antenna to the IQxel. Once the expected signal level was fixed, the eye diagram looked reasonable, seen in Figure 5.3, and which packets the IQxel could decode was predictable. The packets appeared in a capture in groups of three, of which the middle was always decoded and the other two were not. It was suspected that the CYW20719 was transmitting one packet on each BLE advertising channel, only one of which was close enough to the center frequency to be decoded. Additionally, the packets still appeared as BTC. At this point, Litepoint was contacted for assistance and given a raw capture file including a set of three packets. They responded with a number of settings that needed to change using the remote access python library. A preexisting script to inspect BTC packets using the IQxel, originally based on a script used for Wi-Fi, was modified to change the required settings and inspect BLE packets. Once the changes were implemented, the originally misidentified packets were

Table 5.1: Settings used on the IQxel to successfully decode BLE packets.

Parameter	Value
Sampling Rate	160MHz
Center Frequency	2441MHz
Reference Level	20dBm
Capture Length	50ms
Packet Type	BLE
Channel Number	Auto-Detect

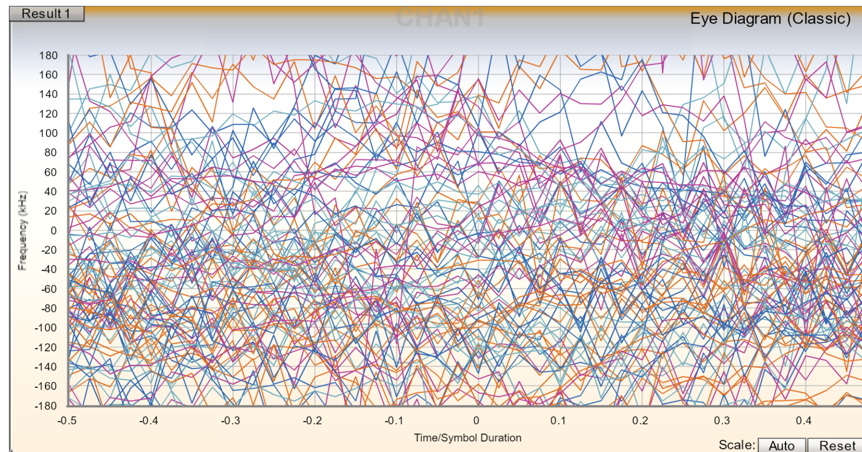

Figure 5.2: Eye diagram for data with incorrect receiver settings. The data was severely clipped, leading to random noise where a packet should be.

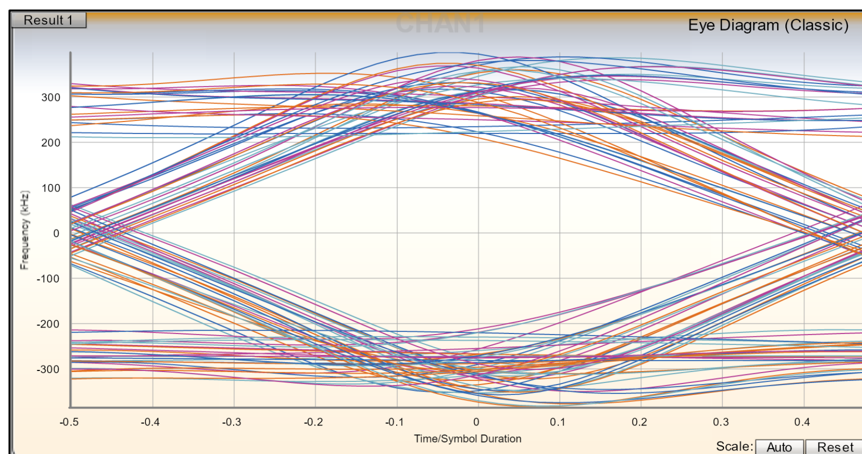

Figure 5.3: Eye diagram for data taken with correct receiver settings. Frequency offset of the received signal from the channel center, used in BT to represent packet data, is the vertical axis, while time offset from symbol center is represented on the horizontal axis. Individual symbols in the transmission are layered on top of each other, showing the repeatability and accuracy of captured symbol transitions. This is a good representation of clean, real-world data.

decoded by the IQxel into BLE data successfully. The final settings used for packet capture and decoding are given in Table 5.1. Once the packets were decoded on the IQxel, the script pulled the packet bits from the IQxel for processing into a PCAPNG file. For an unknown reason, the SCPI API returns packet data from the IQxel as a series of bytes with each byte representing one bit. This requires some processing to fix the formatting before creating the PCAPNG file. Additionally, at the time of writing the correct endianness and order of the bits is unknown. Currently, the script arranges the bits to present the same data reported by a BLE advertising packet sniffer application on the test smartphone, but this may need further refinement if the correct ordering is different. Once the packet data is corrected, the packets are written into a JSON file that is converted into a PCAPNG file using a different script. Future work will likely include changing the script to create PCAPNG files directly.

5.2 Litepoint Interface

Access to the IQxel is provided through an Ethernet port, connecting the test computer to the webserver running on the IQxel. The web interface provides real-time data and control over the IQxel, including port configurations alongside data acquisition and interpretation controls. A screenshot of the web interface is given in Figure 5.4. The IQxel can also be controlled with SCPI commands through either the web interface or the Ethernet port using a python API provided by the company. The SCPI web interface supports most but not all of the commands needed for this project and is difficult to extract data from for later analysis, so the python API was used.

The IQxel cannot decrypt or dewhiten packets it receives from an ongoing BLE connection. Future work will include code that pulls encrypted packets from the IQxel during a run of the image transfer test. Then it will find the encryption key in the log file created during the test and decrypt the IQxel data after dewhitening the packets. At the time of writing, the IQxel has only been used to decode advertising packets.

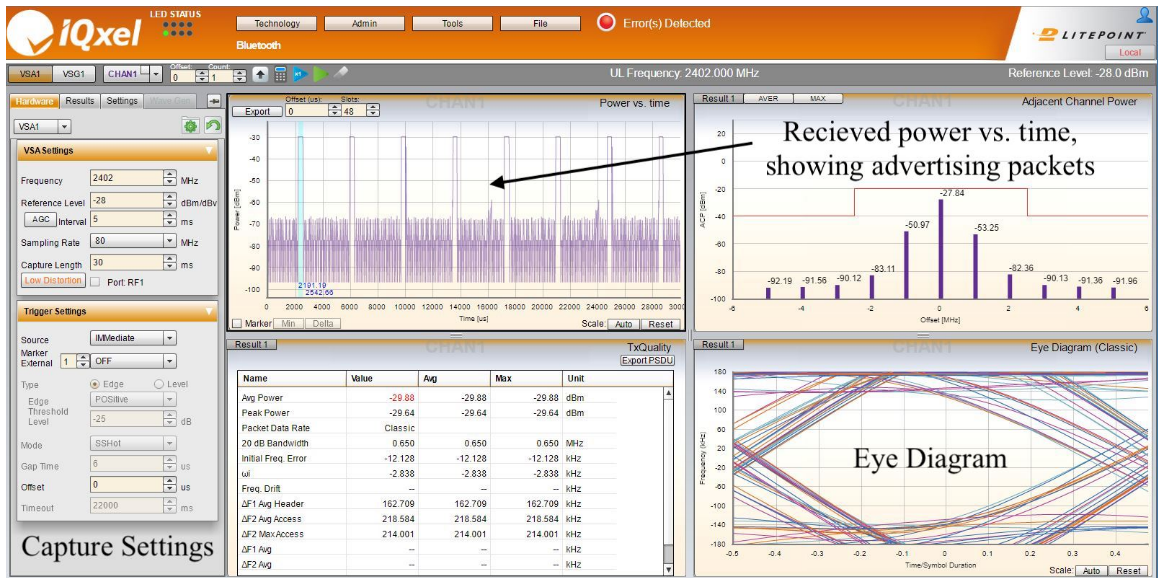

Figure 5.4: IQxel web interface. Input configuration controls are on the left, while user-configurable graphs are on the right. The data shown is advertising packets taken during initial testing of the IQxel.

5.3 Summary

This chapter presented the basis for enhancing the existing sniffing framework using an IQxel. This addition can be used to fill in gaps in data from the HCI packets and give a more complete picture of the wireless environment of the BLE connection. At the time of writing the IQxel has only been used to detect advertising packets; a collaborator is continuing the work presented here to develop a more complete system implementation.

Chapter 6

Conclusion

6.1 Research Outcomes

This project was successful in achieving the goals set at its inception. The inline sniffing system is reliable, does not negatively impact link performance, and provides a platform for analysis of both the link itself and application-layer data. Testing showed that the sniffing system does not introduce additional errors into the system or slow down data transfer speeds. Additionally, the system is based on a commercially available BLE IC and has full compatibility with evaluation hardware, significantly lowering required hardware costs. The firmware loading utility developed to support this hardware successfully updates the firmware on the IC using images created by the official compiler, broadening the number and types of supported tests. Finally, the interface with the IQxel system enhances capture performance by filling in gaps in data collected by the inline sniffer. Overall, this project successfully built and demonstrated a framework for future developments in BT research and development.

6.2 Future Work

While this project was successful, each phase of the project has areas that would need addressing in future work. The PyControl software has some small issues requiring input sanitization, but the major area for improvement is adding support for other test types. For

example, supporting the audio streaming or keyboard emulation example firmwares could be very useful for real-world testing. The firmware loader is predictable but fails frequently and does not properly manage autobaud mode like the official utilities. While these issues were unimportant to the goals of this project they would likely need to be resolved for future applications. Additionally, both the firmware loader and PyControl need a more robust system for identifying the correct serial port to connect to for communicating with the CYW20719. The final section of the project, interacting with the IQxel, is in earlier stages of development that the other two. The next step in this area would be using the IQxel to capture and export packets during a live file transfer. After that, some post-processing scripts should be written to translate the packets into a format Wireshark can interpret. While this project is far from complete, future work has clear objectives and concrete places to move forward from.

Appendix A

Throughput Software

A.1 PyControl

```
import time
#python -m cProfile -s tottime pycontrol.py
import logger_class
import interpreter_class
import communicator_class
import plotter

comms = communicator_class.Communicator()
interp = interpreter_class.Interpreter()
logs = logger_class.Logger()


hci_packets = []

#reset board
#send queued packets
#go into main loop

###The reset is still not working the way I expect
#returns the body of the first HCI packet
```

```
21 #packet_one = comms.reset_board()
22 #need to handle the first one manually, since interp doesn't see it
23 #hci_packets.append([ord(packet_one[0]),packet_one[1:]])
24
25 comms.send_initial_packets()
26
27
28 def transfer_a_file():
29     global time #I dunno why this needs to be here but not the others
30     #SLEEPING WHILE PROCESSING CAN CAUSE THE INPUT BUFFER TO OVERFLOW
31     #Also cache everything, since the hard drive can get angry
32     data_packets_timing = []
33     data_packets = []
34     file_name = ""
35     while True:
36         interp.new_packet()
37
38         header = comms.get_header_bytes()
39
40
41         len_needed = interp.parse_header(header)
42         #check len needed for errors
43         #flush buffer to log file in the future
44         #change this so that interp handles it silently
45         if len_needed < 0:
46             comms.clear_input()
47             continue
48
49         body = comms.get_packet_bytes(len_needed)
50         interp.parse_params(body)
51         #print interp
52
53         if interp.is_HCI_packet():
```

```
54                  pass
55                  hci_packets.append([interp.get_HCI_direction(),
                     ↪  interp.get_HCI_packet_bytes()])
56
57              elif interp.file_transfer_requested():
58                  print "file transfer started"
59                  file_name = body[13:-1]
60                  print "file name: %s" % file_name
61                  comms.send_packet(interp.get_transfer_req_response())
62                  file_transfer_started = True
63                  data_packets_timing.append([time.time(), 0])
64
65              elif interp.is_data_packet():
66                  data_packets.append(body)
67                  data_packets_timing.append([time.time(), len_needed])
68
69              elif interp.file_transfer_complete():
70                  print "file transfer complete"
71                  break
72
73
74          logs.open_file_transfer(file_name)
75          for packet in data_packets:
76              logs.write_to_file_transfer(packet)
77          logs.close_file_transfer()
78
79          times = [t for t,size in data_packets_timing]
80          time_zero = times[0]
81          time_offsets = [x - time_zero for x in times]
82          sizes = [size for t,size in data_packets_timing]
83
84          #fix getting the timestamps, this isn't a good way to do it
85          plotter.show_throughput_plot(time_offsets, sizes, file_name, logs.timestamp)
```

```
86
87
88  #to fix
89  #why doesn't it advertise after a reset?
90
91  num_transfers = 1
92  #while True:
93  print "%s transfers expected" % num_transfers
94  for i in range(num_transfers):
95      print "ready for transfer"
96      transfer_a_file()
97      #waiting at the prompt is blocking, don't leave it waiting here
98      #yes_no = raw_input("transfer another file? ")
99      #if yes_no[0] not in "yY": break
100
101 #HCI packets recieved after completing the final image transfer ARE NOT LOGGED
    ↪  CURRENTLY
102 logs.open_hci_log()
103 for dir, packet in hci_packets:
104     logs.log_HCI_packet(dir, packet)
105 logs.close_hci_log()
```

A.2 Communicator

```
import serial
import serial.tools.list_ports as lp
import time

#send reset command first?

#everything CC sends on connection
trace_enable = '19 02 00 02 00 01 01'
set_discoverable = '19 08 00 02 00 01 01'
set_pairable = '19 09 00 01 00 01 '
get_version_num = '19 02 FF 00 00'
set_psm_number = '19 05 23 02 00 13 00'
set_mcu_size = '19 04 23 02 00 00 02'

commands = [trace_enable, set_discoverable, set_pairable, get_version_num,
↪ set_psm_number, set_mcu_size]

class Communicator():

    def __init__(self):

        self.out_packet_queue = [trace_enable, set_discoverable, get_version_num,
        ↪ set_pairable]

        correct_port = ""
        #the above depends on your computer etc
        #you need the HCI uart, not the PUART

        #find the correct port based on name
        ports = lp.comports()
        for port in ports:
            if port[1].startswith("WICED HCI UART"):
```

```
31                  #Cypress development board
32                  correct_port = port[0]
33                  print "found WICED HCI:", correct_port
34                  break
35          else:
36              #look for 4 consecutive ports, pick the first one
37              #This needs to be updated to be more robust
38              names = [port[0] for port in ports]
39              if len(names) < 4:
40                  print "no reasonable port found"
41                  #this will need to change eventually
42                  quit()
43              #Two chip octoScope board, 2 ports per chip
44              correct_port = sorted(names)[0]
45              print "picked lowest port number:", correct_port
46
47
48
49
50          self.HCI_UART = serial.Serial(correct_port, 3000000, timeout=None)
51          #timeout none causes all reads to be blocking
52          #means I don't have to poll anything
53          #WINDOWS ONLY, and it's a request not a command
54          self.HCI_UART.set_buffer_size(rx_size = 12800, tx_size = 12800)
55          time.sleep(.100)
56          #pyserial doens't hold until the port is actually ready
57          #meaning it doesn't flush properly if you don't wait
58
59          self.clear_input()
60          print "input cleared on startup"
61
62
63      #returns the header, blocks until there is one
```

```
64      def get_header_bytes(self):
65          return [ord(x) for x in self.HCI_UART.read(5)]
66
67      def reset_board(self):
68          #this wasn't working right - going back to power resetting the board
69          self.send_packet("19 01 00 00 00")
70          print "sent reset command"
71          print "device starting",
72          while True:
73              #I don't know what it's sending over, so I can't do anything smarter
74              byte = self.HCI_UART.read(1)
75              if byte is chr(0x19):
76                  boot_response = [ord(x) for x in self.HCI_UART.read(4)]
77                  #chr(8) is a backspace, it gets rid of the space between prints
78                  print chr(8) + ".",
79                  if boot_response == [0x03, 0x00, 0x04, 0x00]: break
80          print "\ndevice started"
81          #the first hci packet is always the first of the group of 4
82          #so I know the length
83          return self.HCI_UART.read(4)
84
85      def get_packet_bytes(self, reqd_length):
86          return ''.join(self.HCI_UART.read(reqd_length))
87
88      def send_packet(self, packet_bytes_string):
89          self.HCI_UART.write(bytearray.fromhex(packet_bytes_string))
90
91      def send_queued_packet(self):
92          message = self.out_packet_queue.pop(0)
93          self.send_packet(message)
94          return message
95
96      #this needs to actually wait for confirmation that the instrucitons completed
```

```
97      #not sure how to keep hci packets during this. They would only be
98      #le rand changes, which aren't all that important.
99      #Maybe just toss them???
100     #Do we add the magic packets from before?
101     #Why does this still act differently than clientcontrol???
102     def send_initial_packets(self):
103         for packet in self.out_packet_queue:
104             #There isn't any sleep time when CC sends it
105             #time.sleep(.100)
106             self.send_packet(packet)
107         print "board initialized"
108
109     def add_to_queue(self, packet_bytes_string):
110         self.out_packet_queue.append(packet_bytes_string)
111
112     def num_out_queued(self):
113         return len(self.out_packet_queue)
114
115     def clear_input(self):
116         if self.HCI_UART.in_waiting > 0:
117             self.HCI_UART.reset_input_buffer()
118             #print "cleared input"
119             #if this starts in the middle of a RX, things may go sideways
120
121     def read_all_input_buffer(self):
122         return [ord(x) for x in self.HCI_UART.read(self.HCI_UART.in_waiting)]
123
124     def input_buffer_length(self):
125         return self.HCI_UART.in_waiting
126
127     def close_port(self):
128         self.HCI_UART.close()
```

A.3 Interpreter

```
1 file_transfer_request_response = "19 01 20 02 00 01 00"
2
3 class Interpreter:
4
5     def __init__(self):
6         self.parsed_packets = 0
7         self.new_packet()
8
9     #reset the stored packet info
10    def new_packet(self):
11        self.group = -1
12        self.event_num = -1
13        self.length = -1
14        self.params = []
15
16    #probably freaks out if there's an issue
17    #assumes it is a WICED header
18    #returns remaining length of packet
19    def parse_header(self, header_bytes):
20        if header_bytes[0] is not 0x19:
21            print "Header problem - likely overflowed serial buffer"
22            return -1
23
24        self.event_num = header_bytes[1]
25        self.group = header_bytes[2]
26        self.length = header_bytes[3] + ( header_bytes[4] << 8 )
27        return self.length
28
29
30    def parse_params(self, param_bytes):
31        if not self.is_HCI_packet():
32            self.params = param_bytes
```

```
33          else:
34              self.params = [ord(param_bytes[0]), param_bytes[1:]]
35          self.parsed_packets += 1
36
37      def is_HCI_packet(self):
38          return self.is_packet_type(0, 3)
39
40      def get_HCI_direction(self):
41          return self.params[0]
42
43      def get_HCI_packet_bytes(self):
44          return self.params[1]
45
46      def total_packets(self):
47          return self.parsed_packets
48
49      def hexify(self, p):
50          q = lambda x: [hex(y)[2:].zfill(2) for y in x]
51          return [q(x) for x in p]
52
53      def response_required(self):
54          return (self.group, self.event_num) in self.response_table.keys()
55
56      def get_required_response(self):
57          return self.response_table[(self.group, self.event_num)]
58
59      def is_data_packet(self):
60          return self.is_packet_type(32, 5)
61
62      def file_transfer_requested(self):
63          return self.is_packet_type(32, 4)
64
65      def file_transfer_complete(self):
```

```
66            return self.is_packet_type(32, 2)
67
68        def get_transfer_req_response(self):
69            return file_transfer_request_response
70
71        def is_packet_type(self, group_code, event_code):
72            return self.group == group_code and self.event_num == event_code
73
74        def __str__(self):
75            if self.is_HCI_packet():
76                return "packet %s (HCI)" % self.total_packets()
77
78            if self.is_data_packet():
79                return "data: %s" % " ".join([str(x) for x in self.params])
80
81            out = "packet %s (Cypress)\n" % self.total_packets()
82
83            out += "group: %s\n" % self.group
84            out += "event: %s\n" % self.event_num
85            out += "length: %s\n" % self.length
86            out += "parameters: %s" % self.params#self.hexify(self.params)
87
88            return out
```

A.4 Logger

```
1  import time
2
3  def get_timestamp():
4      t = time.time()*1000000 #seconds to useconds
5      t += 66463223296000000
6      #Magic number derived from testing. No idea where it comes from
7      #fixes timestamp numbering in wireshark
8      #it's over 2100 years
9      return int(t)
10
11 def get_pcap_timestamp():
12     t = time.time()
13     t_us = (t - int(t)) * 1000000
14     return int (t), int(t_us)
15
16 class Logger:
17
18     def __init__(self):
19         self.num_HCI_packets = 0
20         self.timestamp = get_pcap_timestamp()[0]
21         #print timestamp
22
23     def good_hex(self, x):
24         #you could replace lstrip with [2:] but that's less readable
25         return hex(x).rstrip("L").lstrip("0x").zfill(2)
26
27     #don't be a moron here
28     def log_int_to_file(self, num, pad_length_bits):
29         byte_character_length = pad_length_bits / 4
30         #number of characters the string should have, 4 bits per character
31         #turn int into hex string
32         num_string = self.good_hex(num).zfill(byte_character_length)
```

```
33          #python appends "L" to longs when you print them
34          #remarkably, L is not a number in hex
35          #pad to correct length
36          chrs = [chr(int(''.join(x),16)) for x in zip(*[iter(num_string)]*2)]
37          #see zip documentation
38          self.hci_log.write(''.join(chrs))
39
40      #Wireshark doesn't recognize all packet types for some reason
41      #packet bytes is an array of ints in 0-255
42      def log_HCI_packet(self, direction, packet_bytes):
43          #PCAP timestamp
44          ts_sec, ts_u_sec = get_pcap_timestamp()
45          #ts_sec - Seconds only epoch time
46          self.log_int_to_file(ts_sec,32)
47          #ts_u_usec - Microseconds offset to ts_sec
48          self.log_int_to_file(ts_u_sec, 32)
49          #PCAP length
50          direction_of_packet = direction % 2
51          #incl_len
52          self.log_int_to_file(len(packet_bytes) + 1 + 4, 32)
53          #orig_len
54          self.log_int_to_file(len(packet_bytes) + 1 + 4, 32)
55          self.log_int_to_file(direction_of_packet, 32)
56          #trying to fix h4 formatting with the below
57          total_packet = chr(direction) + packet_bytes
58          #if direction_of_packet == 0: total_packet = total_packet[1:]
59          self.hci_log.write(total_packet)
60
61          self.num_HCI_packets += 1
62
63      def close_hci_log(self):
64          self.hci_log.close()
65
```

```
66      def open_file_transfer(self, file_name):
67          self.file = open("./files/%s_%s" % (self.timestamp, file_name), "wb")
68
69      def open_hci_log(self):
70          self.hci_log = open("./files/%s_hci_log.pcap" % self.timestamp, "wb")
71          #some systems get angry if files don't have extensions
72          #timestamp included so files aren't overwritten
73          #need to have it opened as binary to fix accidentally writing CRLF
74          self.hci_log.write(bytearray.fromhex("a1b2c3d4 0002 0004 00000000 00000000
        ↪  0000ffff 000000c9"))
75          '''
76          Magic Number
77          Major Version
78          Minor Version
79          Timezone
80          Sig Figs
81          Max Packet Size
82          Data Link Type
83          '''
84
85      def write_to_file_transfer(self, data_bytes):
86          self.file.write(data_bytes)
87
88      def close_file_transfer(self):
89          self.file.close()
```

A.5 Plotter

```python
import matplotlib.pyplot as plt

def show_throughput_plot(time_offsets, packet_sizes, file_name, log_timestamp):
    #format:
    #time offsets started at beginning of file, 0 bit packet size
    number_of_seconds = int(time_offsets[-1]) + 1
    binned_sizes = [0.0]*number_of_seconds
    times = range(number_of_seconds)
    for time, size in zip(time_offsets, packet_sizes):
        binned_sizes[int(time)] += round(size / 125.0, 4) #8/1000, bytes to kb

    average = sum(binned_sizes) / number_of_seconds
    #plot line first so the curve is on top
    plt.axhline(average, color='r')
    plt.plot(times, binned_sizes)
    plt.ylabel("kbps")
    plt.xlabel("seconds")
    plt.title(file_name)
    #need to save before show, not sure why
    plt.savefig("./files/%s_%s_throughput.png" % (log_timestamp, file_name[:-4]))
    #I believe show is blocking, don't want to leave it up indefinitely
    #plt.show()
    plt.close()
```

Appendix B

Firmware Flasher Utility

```
1  ################################################################################
2
3  import serial
4  import serial.tools.list_ports as lp
5  import time
6  import sys
7  import os
8
9  #firmware image must be in the same folder as this script
10 #the below makes it so the terminal can call this script from anywhere
11 #print os.path.dirname(__file__)
12 #os.chdir(os.path.dirname(__file__))
13 ### port initialization ########################################################
14 correct_port = ""
15 #you need the HCI uart, not the PUART
16 #find the correct port based on name
17 ports = lp.comports()
18 for port in ports:
19     if port[1].startswith("WICED HCI UART"):
20         #Cypress development board
21         correct_port = port[0]
22         print "found WICED HCI:", correct_port
```

```
23          break
24  else:
25      if len(ports) > 0:
26          correct_port = ports[0][0]
27          print "attempting lowest port: %s" % correct_port
28      else:
29          print "no available com ports"
30          quit()
31
32
33  HCI_UART = serial.Serial(correct_port, 3000000, timeout=.300)
34  #Using a timeout longer than any wait period
35  #if the board responds faster than spec the script will take less time
36  #this also makes some other code easier
37  time.sleep(.100)
38  #pyserial doesn't hold until the port is actually ready
39  #meaning it doesn't flush properly if you don't wait
40  HCI_UART.reset_input_buffer()
41  print "input cleared on startup"
42
43  ### methods ####################################################################
44  def send_packet(packet_bytes_string):
45      HCI_UART.write(bytearray.fromhex(packet_bytes_string))
46
47  commands = {"hci_reset":"01 03 0C 00",
48              "download_minidriver":"01 2E FC 00",
49              "baud_3M":" 01 18 FC 06 00 00 C0 C6 2D 00",
50              "erase_chip":"01 CE FF 04 EF EE BE FC",
51              "launch_minidriver":"01 4E FC 04 00 00 22 00",
52              "launch_ram":"01 4E FC 04 FF FF FF FF"}
53
54  #Expected response only changes based on sent command type
55  #so the method calculates the correct "command succeeded" response
```

```
56 #compares the success response to the actual response
57 #returns true if they match, fatal errors if they don't
58 def recieve_response(command_name, p=True):
59     sent_command_type = commands.get(command_name, "01 4C FC")[3:8] #default is
     ↪ the header for write ram
60     expected_response = [int(x,16) for x in ("04 0E 04 01 " + sent_command_type +
     ↪ " 00").split()]
61     response = [ord(x) for x in HCI_UART.read(len(expected_response))]
62     if response == expected_response:
63         if p: print "response received"
64         return True
65     else:
66         print "\ncommand error"
67         print "expected", expected_response
68         response = response + [ord(x) for x in HCI_UART.read(HCI_UART.in_waiting)]
69         print "HCI Buffer:", response
70         quit()
71
72 #wrapper for sending control commands to the board
73 def send_command(command_name):
74     print "sending command %s:" % command_name,
75     send_packet(commands[command_name])
76     return recieve_response(command_name)
77     #recieve_response deals with the failures
78
79 #takes in an integer, returns the hex string representation without a leading 0x
80 #or trailing "L" for python longs, with a minimum character length of 2
81 def good_hex(x):
82     #you could replace lstrip with [2:] but that's less readable
83     return hex(x).rstrip("L").lstrip("0x").zfill(2)
84
85 def reset_board():
86     #Looping this doesn't actually do anything useful
```

```
87      if send_command("hci_reset"): return
88      print "could not reset device. Please cycle power"
89      quit()
90
91  def flip_halfs(two_byte_hex): #stupid little endian memory address nonsense
92      return two_byte_hex[2:] + two_byte_hex[:2]
93
94  def update_baud():
95      pass
96      #I haven't needed to do this, even though the documentation indicates it
     ↪   always happens
97
98  #break up an I32HEX line into it's parts
99  def parse_image_line(line):
100     #the following returns the image record string split into pairs of characters
101     #representing a single hex byte
102     pairs = ["".join(x) for x in zip(*[iter(line[1:])]*2)]
103     byte_count = int(pairs[0],16) #how many pairs we are supposed to have
104     address = flip_halfs(''.join(pairs[1:3])) #memory address destination
105     type = pairs[3] #record type indicator - see HEX image documentation
106     data = ''.join(pairs[4:-1]) #the data contained in the record
107     checksum = pairs[-1] #not actually used at the moment
108
109     if byte_count != len(data)/2:#checks for a basic length mismatch
110         #This only came up when I was doing something stupid
111         #but it's a good check to have
112         print "record length mismatch", line, byte_count, len(data)
113         quit()
114
115     return byte_count, address, type, data, checksum
116
117 ### process and write out image lines method ###################################
118 def write_image(file_name):
```

```
119     #make sure when you run this, the CWD in the terminal is a folder
120     #with this script and the firmware image in it
121     rom_image = ''
122     #I'm reading in the whole thing so that I can do the progress bar easily
123     #also helps avoid HDD random io slowness
124     with open(file_name, "r") as file:
125         rom_image = file.readlines()
126     print "writing image", file_name
127 
128     memory_offset = "FFFF" #always gets changed by the first line of the rom image
129     num_records = len(rom_image)
130     for index, line in enumerate(rom_image):
131         byte_count, address, type, data, checksum = parse_image_line(line)
132 
133         if type == "00": #normal data record
134             header = "014CFC"
135             length = good_hex(byte_count + 4)
136             message = header + " " + length + " " + address + memory_offset + " "
                ↪  + data
137             #There's no command in the table for this, since it changes based on
                ↪  the data
138             send_packet(message)
139             recieve_response("write_ram", p=False)
140             #if the recieve fails, the program quits
141             #No data validation is ever done, but here is an option for where to
                ↪  do it
142 
143         if type == "04": #extended memory addressing
144             #data will always be 2 bytes, big endian
145             #this covers the offset for the beginning of flash memory
146             #also deals with crossing page boundaries and discontinuous data
147             memory_offset = flip_halfs(''.join(data))
148 
```

```
149         if type == "05":
150             continue
151             #deal with this later
152             #has to do with launch ram addresses,
153             #per "WICED HCI UART Control Protocol" chapter 2
154             #the minidriver always launches from 0x00220000
155             #and the flash image always launches from the
156             #"launch flash image" address at 0xFFFFFFFF
157             #So I think this can just be ignored
158 
159         if type in ["02","03"]:
160             #x86 specific stuff.
161             #per documentation, this is I32HEX, meaning types 2 and 3 don't occur
162             print "\nimage error", line
163             quit() #no sense continuing from here
164 
165         #stdout allows me to do a carrage return and overwrite - print has auto
            ↪  CRLF which is harder to deal with
166         sys.stdout.write("\r%s of %s records processed" % (index+1,num_records))
167         if type == "01": #end of file indication
168             print
169             return #no more file writing to do
170 
171 
172 
173 ### setup file transfer ########################################################
174 reset_board()
175 #if reset fails, try again
176 send_command("download_minidriver")
177 #if the minidriver fails, you need to cycle board power - it only accepts once
178 #it might be in autobaud mode - look into this later
179 write_image("minidriver-20739A0-uart.hex")
180 send_command("launch_minidriver")
```

```
181 send_command("erase_chip")
182 ### write image file ############################################################
183 write_image("octoAdv20719_A.hex")
184 send_command("launch_ram")
185 print "\nimage running"
186 
187 HCI_UART.close()
```

Appendix C

Litepoint Interface

```
# coding: utf-8
#cd "C:\Users\Galahad Wensing\Desktop\pycontrol\Litepoint_interface"
#py PackettoJSONLitepointBT.py 169.254.22.24 test.json
'''
Instructs a Litepoint IQxel to re-analyze all
packets within an IQ capture in Channel 1 and write results
to a JSON file.
'''


import sys
import lime #provided by Litepoint
import json
import time
import datetime
import argparse

#takes in a "bytes" class where each byte is one bit
#I dunno why lime does it that way
def bytes_to_hex_str(input_bytes,base=2):
    bit_str = "".join([str(x) for x in input_bytes])[::-1] #[::-1] reverses the
     ↪ bit sequence
```

```
22      val = int(bit_str,base)
23      hex_str = hex(val)[2:] #gets rid of leading 0x
24
25      #swap hex string but preserve pairs
26      #may need to flip hex str pairs or val around to get the data correctly
        ↪  endianed
27      #groups = zip(*[iter(hex_str)]*2)
28      #pairs = ["".join(x) for x in groups][::-1]
29      #hex_str = "".join(pairs)
30
31      #new_order = []
32
33      return hex_str
34
35  def help():
36      print("Usage: ")
37      print("    python %s [<jsonFile>]" % __file__)
38      print("    [<jsonFile>] - filename for JSON packet storage")
39      exit()
40
41  packet_dict_entries = {
42  "Preamble Bytes": "PRE",
43  "Header Bytes": "PDUH",
44  #"Header Bytes": "HEAD",
45  "Payload Bytes": "PAYL",
46  "CRC Bytes": "CRC",
47  }
48
49  def packet_to_json(jsonFilename, LP_IP_address='169.254.22.24'):
50
51      single_packet_dict_default = {
52          "Capture Timestamp": "01:01:1980:13:00:00.000000",
53          "Preamble Bytes": "",
```

```
54          "Header Bytes": "",
55          "Payload Bytes": "",
56          "CRC Bytes": ""
57      }
58
59      dictofAllPackets = {}
60
61      lime.initLime()
62      con = lime.connect(address=LP_IP_address, port=24000)
63      con.setTimeout(60000)
64
65      lime.Print("Getting packet count:")
66      #turn this into a list, send after \n join?
67      scpi_commands = '''
68          BT;CONF:DRAT LEN;
69          BT;CONF:DEWH ON;
70          BT;CONF:CHAN:AUTO ON;
71          BT;CONF:SLOC CIND;
72          BT;CONF:LEN:SWOR:AUTO ON;
73          BT;CONF:LEN:PHE LINK;
74          BT
75          CHAN1
76          BT
77          CALC:POW 0,1
78          BT;FETC:SYNC?
79          '''
80          #set to channel 1
81          #set commands bluetooth
82          #calculate power of capture segment, no offset, one packet. Nothing is
        ↪   returned. I don't think this is used
83          #why set bluetooth again?
84          #fetch segment sync. Does it default to segm1? returns status, total
        ↪   packets, complete packets
```

```
85        r = con.query1d(scpi_commands)
86        #if r[0]=='0': #if fetch segment succeeded
87        if r[0] is not '0':
88            lime.error("No valid packets found\n")
89            #I don't know if this will stop execution or not
90
91        packet_count = int(r[2]) #number of complete packets
92        print("packet_count : %d" % packet_count)
93
94        ts = time.time()
95        capturetime =
          ↪ datetime.datetime.fromtimestamp(ts).strftime('%m:%d:%Y:%H:%M:%S.%f')
96        #this gives you the time the script is run why???
97
98        for packetNumber in range(packet_count):
99            print()
100           print("*****Packet : %d" % (packetNumber))
101
102           single_packet_dict = single_packet_dict_default.copy() #copy default
              ↪ packet dictionary
103           single_packet_dict['Capture Timestamp'] = capturetime
104
105           #get packet type here!!
106           #determine analysis type
107           #how do we get packet type????
108
109           scpi_commands_template = '''
110               CHAN1
111               BT
112               CLE:ALL
113               CALC:POW {0},1
114               CALC:TXQ {0},1
115               CALC:SPEC {0},1
```

```
116             '''
117         #set channel one, bluetooth, clear all
118         #dunno why we keep setting chan1 bt
119         #calculate power, offset packet num packets, one packet (calculate power
            ↪  for packet # packetNumber). no return.
120         #calculate transmit quality for current packet. what is transmit
            ↪  quality??? no return.
121         #calculate spectrum for current packet. no return.
122 
123         scpi_commands = scpi_commands_template.format(packetNumber, 1)
124         #print(scpi_commands)
125         con.scpi_exec(scpi_commands) #run calculation commands
126 
127         r = con.query1d("*wai;err:all?")
128         #wait for commands to finish. Request all error messages.
129         if int(r[0]) != 0: #if there are errors
130             print("error: %s" % r)
131             dictofAllPackets[packetNumber] = single_packet_dict
132             continue
133 
134 
135         for info_type in packet_dict_entries.keys():#this should preserve as typed
            ↪  order
136             query = packet_dict_entries[info_type]
137             scpi_commands = '''BT;FETC:SEGM1:TXQ:LEN:%s?''' % query
138             #print(scpi_commands)
139             r = con.query1d(scpi_commands)
140             if r[0] is "0":
141                 bits = r[1]
142                 print("Number of %s bits: %s" % (info_type, len(bits)))
143                 packet_hex = bytes_to_hex_str(bits)
144                 single_packet_dict[info_type] = packet_hex
145             else:
```

```
146                     print("%s fetch error %s" % (info_type, r[0]))
147 
148             r = con.query1d('''FETC:SEGM1:SYNC:PST?''')
149             if r[0] is "0":
150             #    print(r[1])
151                 single_packet_dict['Capture Timestamp'] = "".join([str(x) for x in
                     ↪  r[1]])
152 
153             key = str(int(packetNumber))
154             dictofAllPackets.update({key:single_packet_dict})
155 
156         #print(dictofAllPackets)
157         json_str = json.dumps(dictofAllPackets, indent=4)
158 
159         #print(json_str)
160         f = open(jsonFilename,"w")
161         f.write(json_str)
162         f.close()
163 
164 
165 
166 
167     if __name__ == "__main__":
168 
169         parser = argparse.ArgumentParser(description='This script commands a LitePoint
         ↪  analyzer to re-analyze and then save results to JSON.')
170         parser.add_argument('LP_IP_address', type=str, help='IP address of the
         ↪  LitePoint analyzer')
171         parser.add_argument('JSON_file', type=str, help='Name of output JSON file')
172         args = parser.parse_args()
173         LP_IP_address = args.LP_IP_address
174         jsonFile = args.JSON_file
175 
```

```
176     # Read the parameters
177     '''
178     if len(sys.argv) >= 3:
179         help()
180 
181     if len(sys.argv) <= 1:
182         help()
183     jsonFile           = sys.argv[1]
184 '''
185 
186     print("jsonFile         =%s " % jsonFile)
187 
188     packet_to_json(jsonFile, LP_IP_address=LP_IP_address)
```

Bibliography

[1] Mohammad Afaneh. *Bluetooth 5 speed: How to achieve maximum throughput for your BLE application*. NovelBits, 2017. URL: https://www.novelbits.io/bluetooth-5-speed-maximum-throughput/.

[2] Mohammad Afaneh. *How to use a Bluetooth (BLE) sniffer without pulling your hair out!* 2016. URL: https://www.novelbits.io/bluetooth-low-energy-sniffer-tutorial/.

[3] Wahhab Albazrqaoe, Jun Huang, and Guoliang Xing. *A Practical Bluetooth Traffic Sniffing System: Design, Implementation, and Countermeasure*. IEEE, 2018. DOI: 10.1109/TNET.2018.2880970.

[4] Argenox. *BLE Advertising primer*. URL: https://www.argenox.com/library/bluetooth-low-energy/ble-advertising-primer/.

[5] Jim Blom. *Bluetooth Basics*. Sparkfun. URL: https://learn.sparkfun.com/tutorials/bluetooth-basics/all.

[6] David Burnett. *All BLE guides are wrong including this one*. UC Berkeley, 2018. URL: http://www-inst.eecs.berkeley.edu/~ee290c/sp18/lec/Lecture7A.pdf.

[7] Ellisys. *Methods for Accessing a Link Key*. 2014.

[8] Hartmut Goebel. *BluetoothScatternet-de.svg*. 2006. URL: https://commons.wikimedia.org/wiki/File:BluetoothScatternet-de.svg.

[9] Goldtouch. *How to Stop Bluetooth Interference From Messing With Your Other Devices ... and other common Bluetooth annoyances*. 2014. URL: https://www.goldtouch.com/stop-bluetooth-interference-messing-devices/.

[10] Dmitry Grinberg. *Faking Bluetooth LE*. 2012. URL: https://dmitry.gr/?r=05.Projects&proj=11.%20Bluetooth%20LE%20fakery.

[11] Akshay Kumar, Darshan Bhat, and Freeze Francis. *Bluetooth Low Energy Security: A Case Study*. 2016. URL: https://www.slideshare.net/FREEZ7/bluetooth-low-energy-a-case-study.

[12] Teledyne LeCroy. *BTSnoop File Format*. Teledyne LeCroy. URL: http://www.fte.com/webhelp/bpa600/Content/Technical_Information/BT_Snoop_File_Format.htm.

[13] Litepoint. *IQxel-MW: High-performance test for 160MHz & 802.11ax Wi-Fi 6 devices*. 2019. URL: https://www.litepoint.com/products/iqxel-mw/.

[14] Hari Nagalla. *Capturing Bluetooth Host Controller Interface (HCI) Logs*. Texas Instruments, 2019.

[15] octoScope. *Stack-Benchtop testbed*. 2019. URL: https://www.octoscope.com/English/Products/Ordering/Testbeds_PreConfigured/STACK-BENCHTOP.html.

[16] John Padgette et al. *Guide to Bluetooth Security*. Vol. 800-121r2. Special Publications. National Institute of Standards and Technology, 2017. DOI: 10.6028/NIST.SP.800-121r2.

[17] F ulvio Risso et al. *PCAP Next Generation (pcapng) Capture File Format*. 2019. URL: http://xml2rfc.tools.ietf.org/cgi-bin/xml2rfc.cgi?url=https://raw.githubusercontent.com/pcapng/pcapng/master/draft-tuexen-opsawg-pcapng.xml&modeAsFormat=html/ascii&type=ascii.

[18] Mike Ryan. *Bluetooth: With Low Energy comes Low Security*.

[19] Cypress Semiconductor. *CYW20719 Enhanced Low Power, BR/EDR/BLE Bluetooth 5.0 SOC*. 2018.

[20] Cypress Semiconductor. *CYW920719Q40EVB-01 Evaluation Board User Guide*. 2018.

[21] Cypress Semiconductor. *CYW920719Q40EVB-01 Evaluation Kit*. 2018. URL: https://www.cypress.com/documentation/development-kitsboards/cyw920719q40evb-01-evaluation-kit.

[22] Cypress Semiconductor. *WICED HCI UART Control Protocol*. 2017.

[23] Muhammad Shafiq et al. *Ranked Sense Multiple Access Control Protocol for Multi-channel Cognitive Radio-Based IoT Networks*. MDPI, 2019. DOI: 10.3390/s19071703.

[24] Bluetooth SIG. *Bluetooth Core Specification v5.1*. 2019.

[25] Technopedia. *Industrial, Scientific and Medical Radio Band (ISM Band)*. URL: https://www.techopedia.com/definition/27785/industrial-scientific-and-medical-radio-band-ism-band.

[26] Olga Weis. *List of the best RS232 Sniffers*. Eltima Publishing, 2019. URL: https://www.virtual-serial-port.org/articles/best-serial-port-sniffer-solutions/.

[27] Wireshark. *Libpcap File Format*. 2015. URL: https://wiki.wireshark.org/Development/LibpcapFileFormat.

[28] Wireshark. *man page*. URL: https://www.wireshark.org/docs/man-pages/wireshark.html.

Figure 1.1 was obtained from [23] under the Creative Commons 4.0 license, available here: https://creativecommons.org/licenses/by/4.0/