ENHANCING CRITICAL INFRASTRUCTURE SECURITY USING BLUETOOTH LOW ENERGY TRAFFIC SNIFFERS

THESIS

José A. Gutiérrez del Arroyo, Capt, USAF
AFIT-ENG-MS-17-M-034

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY
Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.
The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.
ENHANCING CRITICAL INFRASTRUCTURE SECURITY USING BLUETOOTH LOW ENERGY TRAFFIC SNiffERS

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Engineering

José A. Gutiérrez del Arroyo, B.S.E.E.
Capt, USAF

March 23, 2017

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.
ENHANCING CRITICAL INFRASTRUCTURE SECURITY USING
BLUETOOTH LOW ENERGY TRAFFIC SNIFFERS

THESIS

José A. Gutiérrez del Arroyo, B.S.E.E.
Capt, USAF

Committee Membership:

Maj Jason M. Bindewald, Ph.D.
Chair

Scott R. Graham, Ph.D.
Member

LTC Mason J. Rice, Ph.D.
Member
Abstract

Bluetooth Low Energy (BLE) is a wireless communications protocol used in Critical Infrastructure (CI) applications. Based on recent research trends, it is likely that the next generation of wireless sensor networks, a CI application that the Department of Defense (DoD) regularly employs in surveillance and reconnaissance missions, will include BLE as an inter-sensor communications protocol. Thus, future U.S. military missions may be directly impacted by the security of BLE. One natural way to help protect BLE sensors is to use BLE traffic sniffer to detect attacks. The primary limitation with current sniffer is that they can only capture one connection at a time, making them impractical for applications employing multiple BLE devices. This work aims to overcome that limitation to help secure the types of BLE sensor networks employed by the DoD. First, this work identifies vulnerabilities and enumerates attack vectors against a BLE wireless industrial sensor, presenting a list of security “best practices” that vendors and end-users can follow and demonstrating how users can employ BLE sniffers to detect attacks. The work then introduces BLE-Multi, an enhancement to an open-source BLE sniffer that can simultaneously and reliably capture multiple connections. Finally, the work presents and executes a methodology to evaluate BLE sniffers. Under the evaluation conditions applied, BLE-Multi achieves simultaneous capture of multiple active connections, paving the way for automated defensive tools that can be used by the DoD and security community. The contributions within are published in one journal article and one conference paper and were presented at three conferences focused on wireless security and CI protection.
Acknowledgements

I am grateful to AFIT for letting me participate in this unique and rare opportunity. To my wife, my parents, and my sister: thank you for your love, patience and support. Without you cheering me along the way, this would have not been possible. I’m also indebted to a number of colleagues and students who embarked on this journey with me and whose conversations clarified my thinking on this and other matters. Their friendship and professional collaboration was invaluable. Finally, I must acknowledge my faculty mentors, Maj Bindewald, Dr. Graham, LTC Rice, and Maj Ramsey for their support, encouragement, and guidance throughout this process.

José A. Gutiérrez del Arroyo
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xi</td>
</tr>
<tr>
<td>I. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Problem Statement</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Hypothesis</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Investigative Questions</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Document Structure</td>
<td>5</td>
</tr>
<tr>
<td>II. Background</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Bluetooth Low Energy Security</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Other Wireless Protocols</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Technical Overview</td>
<td>9</td>
</tr>
<tr>
<td>2.3.1 Attribute Protocol and Generic Attribute Profile</td>
<td>9</td>
</tr>
<tr>
<td>2.3.2 Security Manager</td>
<td>10</td>
</tr>
<tr>
<td>2.3.3 Physical and Link Layer</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Tools</td>
<td>14</td>
</tr>
<tr>
<td>2.5 Summary</td>
<td>14</td>
</tr>
<tr>
<td>III. Securing Bluetooth Low Energy Enabled Industrial Monitors</td>
<td>16</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>16</td>
</tr>
<tr>
<td>3.2 Attack Methodology</td>
<td>16</td>
</tr>
<tr>
<td>3.2.1 Selecting the Target</td>
<td>17</td>
</tr>
<tr>
<td>3.2.2 Performing Enumeration and Reconnaissance</td>
<td>17</td>
</tr>
<tr>
<td>3.2.3 Reverse Engineering the Application</td>
<td>18</td>
</tr>
<tr>
<td>3.2.4 Attack</td>
<td>19</td>
</tr>
<tr>
<td>3.3 Attack Detection</td>
<td>22</td>
</tr>
<tr>
<td>3.4 Recovery Methodology</td>
<td>23</td>
</tr>
<tr>
<td>3.5 Securing BLE Devices</td>
<td>25</td>
</tr>
<tr>
<td>3.5.1 Recommendations for Vendors</td>
<td>25</td>
</tr>
<tr>
<td>3.5.2 Recommendations for End Users</td>
<td>26</td>
</tr>
<tr>
<td>3.6 Conclusion</td>
<td>27</td>
</tr>
</tbody>
</table>
IV. Enabling Bluetooth Low Energy Auditing through Synchronized Tracking of Multiple Connections

4.1 Introduction .................................................. 29
4.2 Problems with Current BLE Sniffers ..................... 30
4.3 Design Considerations ..................................... 31
  4.3.1 General Considerations ................................. 32
  4.3.2 Ubertooth One Platform Considerations ............. 33
4.4 Following a Single Connection ............................. 34
  4.4.1 Synchronization ......................................... 34
  4.4.2 Link Layer Control ..................................... 38
4.5 Following Multiple Connections .......................... 39
  4.5.1 Procedure ............................................... 39
  4.5.2 Limitations .............................................. 41
4.6 Evaluation .................................................. 42
  4.6.1 Experiment Design ..................................... 42
  4.6.2 Experiment 1: One Master to One Slave ............. 45
  4.6.3 Experiment 2: One Master to Multiple Slaves ....... 47
  4.6.4 Experiment 3: Multiple Masters to Multiple Slaves 48
4.7 Analysis .................................................. 49
4.8 Conclusion ................................................ 50

V. Future Work and Conclusion ................................. 52

  5.1 Use Case for BLE-Multi ................................... 53
  5.2 Future Work ............................................... 54
    5.2.1 Vulnerability Assessments ............................ 54
    5.2.2 Sniffer Tuning ........................................ 55
    5.2.3 Capability Expansion ................................ 55
    5.2.4 Mobile Masters and Slaves ......................... 56
    5.2.5 Intrusion Detection System (IDS) .................. 56
    5.2.6 Intrusion Protection System (IPS) .................. 57
  5.3 Conclusion ............................................... 57

Appendix A. BLE-Multi Source Code ............................ 58

  1.1 firmware/bluetooth_rxtx/bluetooth_le.c ................. 59
  1.2 firmware/bluetooth_rxtx/bluetooth_le.h ............... 63
  1.3 firmware/bluetooth_rxtx/bluetooth-rxtx.c .......... 69
  1.4 firmware/bluetooth_rxtx/ubertooth_clock.h .......... 134
  1.5 firmware/common/ubertooth.h ......................... 136
  1.6 host/libubertooth/src/ubertooth_control.c .......... 143
  1.7 host/libubertooth/src/ubertooth_interface.h ....... 163
  1.8 host/libubertooth/src/ubertooth.c .................... 167
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.9  host/libubertooth/src/ubertooth.h</td>
<td>179</td>
</tr>
<tr>
<td>1.10 host/ubertooth-tools/src/ubertooth-btle.c</td>
<td>181</td>
</tr>
<tr>
<td>Bibliography</td>
<td>188</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The Bluetooth Low Energy network stack.</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>The BLE connection process, beginning with an advertisement.</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>A single BLE connection showing the effect of connection parameters.</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>Bluetooth Low Energy channel locations; darker lobes represent the three advertisement channels.</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>Onset MX1101 Temperature and Humidity data logger in operation.</td>
<td>17</td>
</tr>
<tr>
<td>6</td>
<td>Configured password “I’MS3cur3” captured by a Bluefruit LE sniffer during authentication.</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>Firmware upload process determined by reverse engineering the application.</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>Device information (a) before and (b) after the attack.</td>
<td>21</td>
</tr>
<tr>
<td>9</td>
<td>Flash memory map for nRF51822 SoC.</td>
<td>23</td>
</tr>
<tr>
<td>10</td>
<td>Connections from the MX1101 access pads to the J-LINK pins on the nRF51.</td>
<td>24</td>
</tr>
<tr>
<td>11</td>
<td>J-Link Commander writing 0xFFFFFFF to the application start address (0x014000).</td>
<td>24</td>
</tr>
<tr>
<td>12</td>
<td>Commercially-available Bluetooth Low Energy sniffers.</td>
<td>31</td>
</tr>
<tr>
<td>13</td>
<td>Relationship between anchors, connection parameters and BLE-Multi timers.</td>
<td>36</td>
</tr>
<tr>
<td>14</td>
<td>Connection Event Scheduler in BLE-Multi.</td>
<td>40</td>
</tr>
<tr>
<td>15</td>
<td>Generic experiment setup (numbered bubbles indicate active links for Experiments 1, 2 and 3).</td>
<td>43</td>
</tr>
<tr>
<td>16</td>
<td>Frame capture success of a single connection over time (95% CI for proportion).</td>
<td>46</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>17</td>
<td>Frame capture success of two simultaneous connections with one master (95% CI for proportion)</td>
<td>47</td>
</tr>
<tr>
<td>18</td>
<td>Frame capture success of two simultaneous connections with two masters (95% CI for proportion)</td>
<td>49</td>
</tr>
</tbody>
</table>
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Summary of BLE tools used throughout this work.</td>
</tr>
<tr>
<td>2</td>
<td>Default services exposed by the MX1101 BLE server.</td>
</tr>
<tr>
<td>3</td>
<td>Vulnerabilities found on the MX1101 (firmware ver. R57-P72.)*</td>
</tr>
<tr>
<td>4</td>
<td>Comparison of commercially-available sniffer and BLE-Multi.</td>
</tr>
<tr>
<td>5</td>
<td>Critical advertisement channel messages and their effects on BLE connections.</td>
</tr>
<tr>
<td>6</td>
<td>Critical data channel control messages and their effects on BLE connections.</td>
</tr>
<tr>
<td>7</td>
<td>Summary of control, independent and dependent variables for experiments.</td>
</tr>
<tr>
<td>8</td>
<td>Raw results for Experiment 1 (one master to one slave).</td>
</tr>
<tr>
<td>9</td>
<td>Raw results for Experiment 2 (one master to two slaves).</td>
</tr>
<tr>
<td>10</td>
<td>Raw results for Experiment 3 (two masters to two slaves, one each).</td>
</tr>
</tbody>
</table>
I. Introduction

Bluetooth Low Energy (BLE) [6] is a wireless communications protocol widely used in applications that benefit from its low implementation overhead and minimal energy consumption. BLE is already built into many modern laptops, tablets and smartphones, making it an ideal candidate for short-range wireless communications for Internet of Things (IoT) devices and other Critical Infrastructure (CI) applications. Due to the simplicity, reliability and widespread availability [33] of BLE, manufacturers will likely continue to employ it in future applications.

In 2014, the Secretary of the Air Force, Deborah L. James, prioritized “Balancing Today’s Readiness with Tomorrow’s Modernization” as one of her top goals for her tenure as Secretary [31]. Modernization implies the updating and upgrading of AF mission systems, as well as the CI that supports them, to stay ahead of the “threats of the future.” Unfortunately, while new technologies make CI systems more capable, more efficient and easier to manage, they can inadvertently introduce new cyber vulnerabilities. Those vulnerabilities could have a major impact on Air Force and DoD missions. In fact, Maj. Gen. James K. McLaughlin, 24th Air Force Commander, predicted in 2014 that the security of CI would be critical to the security of Air Force networks and mission systems [10].

One military application likely to be impacted by the security of BLE is Wireless Sensor Networks (WSN). The U.S. Department of Defense (DoD) regularly employs WSNs for surveillance and reconnaissance missions [50]. WSNs are built from de-
centralized sensors that collaborate to detect a target phenomenon and communicate findings to a central node [12]. Inter-sensor communications are normally achieved through mesh routing algorithms, where each sensor acts as a network hop. With the extended mesh routing capabilities available in Bluetooth v4.2 [7] and the introduction of viable multi-hop algorithms [16, 19, 30], recent research turns to BLE as an inter-sensor communications protocol for WSN applications ranging from in-hospital medical monitoring [32] to environmental monitoring [16].

Outside of WSNs, vendors already use BLE in CI applications that combine computational capabilities with physical processes, mainly in building security and automation. For example, BLE door locks provide facility managers access control mechanisms that are easy to deploy and manage [9]. BLE-enabled temperature/humidity monitors [39], airflow measurement tools [1] and compressed air pressure sensors [54] provide wireless access to key metrics that affect heating, ventilation and air conditioning (HVAC) systems. Unfortunately, several BLE locks can be opened by unauthorized users [45], and as this work shows, some HVAC-related devices are susceptible to malware and application-level attacks. Attacks to these types of systems can impact the operational success of an organization, the security of its assets and the safety of its people [27].

As the DoD continues to upgrade its CI to counter the threat of the future, the security of BLE will become essential in ensuring the success of U.S. military missions.

1.1 Problem Statement

Unfortunately, the security of BLE sensors depends on the level of protection implemented by the vendors. While the BLE specification [7] defines security mechanisms for link layer encryption, authentication and data integrity, developers often ignore these mechanisms, leaving link-layer communications completely unprotected.
In 2016 alone, published exploitation frameworks took advantage of the lack of link-layer security to perform man-in-the-middle attacks [29][48], track and locate private BLE devices [45] and deploy rogue BLE devices to steal user information [44].

Defending WSNs and other CI applications against attack is difficult for two main reasons. First, research into BLE vulnerabilities is relatively recent. Hence, the range and types of attacks that can be used against BLE have not yet been fully enumerated. Second, there is a general lack of BLE defensive tools, especially those that can detect active or imminent attacks. The natural candidate tools for attack detection are BLE traffic sniffers, but current sniffers have limitations that make them impractical for WSNs. Specifically, the Bluefruit LE [3], TI CC2540 [52] and Ubertooth One [42] can follow only one connection at a time, and the Ubertooth One fails to maintain synchronization with active connections [21], which significantly degrades its performance over time. The goal of this work is to determine if BLE sniffers can be used to enhance the security of CI applications, to include WSNs. This thesis answers the following research question:

**Can BLE traffic sniffers be used to enhance CI security?**

### 1.2 Hypothesis

The security of CI and WSNs can be enhanced with a BLE sniffer that detects attacks on multiple simultaneous connections. This is shown by (i) identifying vulnerabilities and attack vectors on CI devices, (ii) determining if those attacks can be detected by BLE traffic sniffers, (iii) expanding the capabilities of current BLE sniffers to remove their detection limitations in WSNs and (iv) developing and executing a methodology for evaluating BLE sniffers.
1.3 Investigative Questions

Respectively, the work is structured to answer investigative questions (IQ) that address each research area.

IQ1: What vulnerabilities exist on BLE devices used in CI? The first step is to expose the types of vulnerabilities and attack vectors that exist on CI BLE devices. To that end, this work executes an attack on an HVAC sensor used to monitor the temperature and humidity of an environment. The attack highlights the vulnerabilities on the device and exploits them to download malicious firmware onto the monitor. Ultimately, the types of weaknesses discovered here inform the security tool built by the rest of this work.

IQ2: Can BLE sniffers detect attacks against BLE devices? This research presents the challenges of attack detection from the perspective of a peripheral, an end-user and a BLE sniffer. It argues that the sniffer is positioned well for detection because it can capture the attack traffic in its entirety. It also provides a method for attack recovery and lists actionable security recommendations for vendors and end-users, including using sniffers to closely monitor unsecured BLE devices.

The fact that a BLE sniffer can detect attacks supports the hypothesis that it can enhance CI security. However, to impact WSN security, current sniffers must be altered to capture more than one simultaneous connection.

IQ3: Can current BLE sniffers be altered to capture multiple connections to enhance their impact to WSNs? Unfortunately, current traffic sniffers can only follow one connection at a time, making them impractical for applications that employ more than one simultaneous connection. Additionally, some open-source sniffers lack the functionality to synchronize to active connections, causing a drastic drop in the likelihood of capture over the length of a connection.

This work extends BLE sniffer detection capabilities by introducing BLE-Multi, a
firmware solution built on the Ubertooth One [42] platform. BLE-Multi can reliably
sniff multiple connections simultaneously by using a timesharing capture technique
and employing a novel connection synchronization mechanism. By doing so, the work
amplifies the impact that sniffers can have on WSN security.

**IQ4: How can the performance of BLE sniffers be measured?** As sniffers
are used in future security appliances, their effectiveness will be key to the success of
the systems that employ them. This work presents a technique to evaluate sniffer per-
formance and executes a baseline evaluation of several commercial and open-source
sniffers. It compares how well different BLE sniffers can capture traffic and confirms
the multi-connection tracking capabilities of BLE-Multi. In the end, BLE-Multi out-
performs other open-source sniffers, while capturing and synchronizing with multiple
simultaneous BLE connections.

### 1.4 Document Structure

The bulk of this work has been presented, published or submitted for publication
by the author in [20], [21] and [24]. The background sections of those publications
are combined and expanded in Chapter II, which provides a summary of the state of
BLE security and a technical overview of the BLE protocol. Chapter III, covering the
contributions of [20] and [21], addresses IQ1 and IQ2 by presenting an attack on an
industrial monitor and analysis of attack detection from the perspective of the moni-
tor, the end-user and a BLE sniffer. Chapter III also presents a recovery mechanism
and outlines recommendations for vendors and end-users on how to enhance the se-
curity of BLE. Chapter IV, covering the contents of [24], highlights the limitations of
current BLE sniffers and addresses IQ3 by introducing BLE-Multi. Chapter IV also
addresses IQ4 with a comparison of currently available BLE sniffers and BLE-Multi.
Finally, Chapter V presents the key areas for future work in this field and concludes.
II. Background

This chapter captures the current state of BLE security and related work, motivating the need for further security research. It follows with a technical summary of the BLE protocol, as necessary to the discussion in this work.

2.1 Bluetooth Low Energy Security

The Bluetooth Special Interest Group (SIG) first introduced BLE, also known as Bluetooth Smart, within Bluetooth Core Specification v4.0 [6] as a low-energy, low-data rate wireless communications protocol. Although BLE is lumped under the Bluetooth name, it is a different protocol from Bluetooth Basic Rate/Enhanced Data Rate (BR/EDR), otherwise known as Bluetooth Classic [26]. While Bluetooth Classic can be used for high data-rate applications, like audio streaming and data sharing, BLE is designed to transmit stateful information, such as whether a lock is open/closed or the temperature of a room. Those transmissions, which require short bursts at a much lower data rate, are particularly useful for CI and IoT applications, where transmitted data often consists of stateful information.

The BLE specification [7] provides low-level security mechanisms to enhance BLE security. It defines procedures to enable device-specific authentication, data encryption and redundancy checks for data integrity at the link layer. The main procedure, called binding or pairing, outputs a set of 128-bit symmetric keys used for encryption and authentication. In 2013, Mike Ryan [46] published a security evaluation highlighting flaws with this process that left it vulnerable to an observant attacker. An attacker that can capture a pairing exchange in its entirety can crack the 128-bit symmetric encryption key and fully decrypt the BLE conversation. Since that publication, the Bluetooth SIG improved the security of the key exchange by adopting an
Elliptic Curve Diffie Hellman (ECDH) algorithm in Bluetooth v4.2 [7].

In practice, few devices implement link-layer security, instead relying upon custom higher-level security or no security at all [21]. Link-layer security mechanisms are optional because each contributes to energy consumption; increased security comes with an increased energy cost [26]. For this reason, the Bluetooth SIG places the burden on developers and vendors to make design decisions about security mechanisms. Whether developers are motivated by energy consumption, by pushing products to market quickly, or by other design decisions, the end result is that most devices lack appropriate levels of security at the link layer.

In 2016 alone, the lack of link layer security contributed to the development of multiple BLE attacks, exploitation frameworks and reconnaissance tools. Two separate teams developed man-in-the-middle frameworks that can be used against vulnerable targets [29, 48]. Both frameworks prey on devices that lack link-layer authentication and encryption. Other researchers targeted commercially-available BLE locks and were able to open 12 out of 16 from up to a quarter mile away [45], and another two teams created BLE reconnaissance tools that exfiltrate private information and track in real time the approximate location of nearby BLE devices [43, 44].

In the same year, research in BLE security and privacy violations drove the development of an active BLE privacy protection tool [14]. With BLE-Guardian, Fawaz et al. [14] offer a front end for authentication and a novel process to restrict connections to authenticated devices. They use jamming to hide advertisements by protected BLE devices from unauthorized users, preventing unauthorized access. While BLE-Guardian focuses on protecting the privacy of unconnected devices, it does not offer protection after a connection is initiated to those devices. Chapter IV shows vulnerabilities that can be exploited after connection initiation and which are not detected or mitigated by BLE-Guardian. With BLE-Multi, this work presents a more capable
BLE traffic sniffer to drive development of advanced security appliances, particularly those that can act on a connection after it is already initiated.

2.2 Other Wireless Protocols

Other protocols used in CI and WSN applications include those based on the IEEE 802.15.4 specification (e.g., WirelessHART and ZigBee) and on the ITU-T G.9959 recommendation (e.g., Z-Wave) [5]. Many of the security challenges found in BLE are also present in these other wireless protocols. In fact, researchers find that the attack tools and mechanisms in one wireless domain often evolve into those in used in other domains [13]. Generally, these attack frameworks are the key drivers in the development of defensive mechanisms and security appliances, especially for new and emerging protocols [17].

One of the most recent examples is Z-Wave security. Z-Wave posed a challenge to researchers because access to the proprietary protocol required signing a non-disclosure agreement [15]. Researchers began developing open-source traffic capture and packet injection tools, which enabled sophisticated attacks on Z-Wave networks. Badenhop et al. [5] outlined Z-Wave attacks ranging from reconnaissance to packet injection and denial-of-service. Other researchers then followed with defensive mechanisms. Fuller et al. [15] built a Misuse-Based Intrusion Detection System (MBIDS) that detects malicious activity captured by an open-source Z-Wave traffic sniffer, while Hall et al. [25] demonstrated a mechanism to perform Z-Wave device fingerprinting to protect against rogue devices.

This work seeks to advance security research in BLE by extending the capabilities of current BLE sniffers. More capable sniffers will enable future security appliances to protect devices that have been left unsecured.
2.3 Technical Overview

The key design objective for BLE is minimal energy consumption. This goal is achieved by minimizing implementation overhead and simplifying communications protocols [26]. This section highlights aspects of the BLE network stack (shown in Figure 1) necessary to the security discussion in this work.

![Figure 1. The Bluetooth Low Energy network stack.](image)

**GATT** - Generic Attribute Profile  
**ATT** - Attribute Protocol  
**L2CAP** - Logical Link Control and Adaptation Protocol  
**SM** - Security Manager

2.3.1 Attribute Protocol and Generic Attribute Profile

The Generic Attribute Profile (GATT), critical to the attack outlined in Chapter IV, defines the interactions between BLE peripherals and BLE centrals. BLE peripherals are devices that serve information (e.g., BLE locks, lightbulbs, sensors), while BLE centrals are devices that consume information (e.g., smartphones, tablets) [26]. Each peripheral contains a set of handles that can be altered by the central using the Attribute Protocol (ATT). Handles are variables that are employed to communicate state or to serve as a device control point. For example, a BLE lightbulb has a handle that holds the current state of the light (i.e., on/off). A smartphone running a
home automation application sends an ATT Read Request to read this handle and report the state back to a user. When the user decides to turn the light on or off, the application sends an ATT Write Request to the appropriate control handle, and the lightbulb changes states. The attack presented by this thesis is informed with the ATT commands that a BLE central sends to a BLE peripheral.

2.3.2 Security Manager

The Security Manager (SM) handles low-level security for BLE connections by using a device pairing and key exchange mechanism called binding [26]. While there are several pairing methods specified to fit the range of BLE applications, the output is always the same: a shared 128-bit Long Term Key (LTK) used to encrypt communications with the AES-128 block cipher encryption algorithm. They also exchange the Connection Signature Resolving Key (CSRK) and the Identity Resolving Key (IRK), which can be used for authentication and privacy, respectively. Different implementations of the SM can yield different levels of security. The devices analyzed in this work are vulnerable because they implement an SM that does not enforce encryption or authentication.

2.3.3 Physical and Link Layer

The Physical and Link Layers are responsible for establishing and maintaining BLE connections. The structure and characteristics of BLE connections are imperative to the mechanisms presented in this work to enhance BLE sniffers. BLE connections occur between two parties: (i) one master and (ii) one slave. Note that these are link-layer roles and are abstracted from the GATT/ATT roles previously discussed. In other words, while the BLE central is typically the master and the BLE peripheral is typically the slave, the opposite is also permitted by the BLE speci-
Each BLE connection is comprised of a series of Connection Events (CE) with varying lengths. The master dictates when, on what frequency, and how often CEs take place. The slave generally has no control over these parameters but can choose to leave the connection at any time. Although the BLE specification allows one master to have connections to multiple slaves and one slave to have connections to multiple masters, BLE treats each link as a separate connection with its own parameters. Figure 2 provides an overview of the process, and each step is described below.

1- **Slave Advertises Presence.** A slave broadcasts frames on one of three advertisement channels. Advertisements are used to announce device presence, execute device scans, and initiate connections. They also contain information about connectability and services provided by the slave. A master observes these advertisements to determine which targets are available for connection; it can only connect to a slave that advertises its presence.

2- **Master Sends Connection Request.** To begin a connection, the master sends a Connection Request (CR) containing critical connection parameters within 150 $\mu$s after the end of an advertisement. The most important parameters to this discussion are the Connection Interval, Window Size, Window Offset, Hop Interval, Channel Map and master Sleep Clock Accuracy (SCA). Window Size and Window Offset determine the timing of the first CE, and the Connection Interval determines
the timing of every CE after the first. The Hop Interval and Channel Map describe
the frequencies on which the master and slave communicate. Finally, the master
SCA is key to the synchronization mechanism. Figure 3 shows the effect of these
connection parameters.

![Figure 3. A single BLE connection showing the effect of connection parameters.]

Note that the start time and frequency for each CE is essentially pre-determined
by the connection parameters exchanged at the start of the connection. This is
generally a valid assumption with two exceptions. First, the master and slave may
have different expectations about where any given CE should start, depending on
how far the clocks (master and slave) have drifted from each other. BLE handles
this phenomenon through a synchronization mechanism that relies on the timing of
the first frame in each CE. This poses challenges to the sniffer presented in this
work, as third-party listeners cannot determine the directionality of captured frames.
Second, the master may choose to change connection parameters at any point in
the conversation. When the master makes a change, it selects a time in the future
at which the change will take place, and both parties adjust the schedule of CEs
accordingly. To continue following the connection, the sniffer presented in this work
is required to implement those changes at the correct time.

**3- Hop.** BLE devices operate in the 2.4 GHz Industrial, Scientific, and Medical
(ISM) frequency band. The BLE specification divides the band into 40x2-MHz chan-
nels, as illustrated in Figure 4. It reserves three channels for advertisements and uses
the rest for data transmission in active connections. During a hop, both the master
and slave tune their radios to a new frequency, as defined by the Channel Map, Hop Increment and hopping algorithm. Note that any two arbitrary BLE connections will likely be talking on two different channels at any point in time. This is the primary physical limitation when expanding BLE sniffer capabilities.

![Bluetooth Low Energy channel locations; darker lobes represent the three advertisement channels.](image)

4- **Master Sends Packet.** At the beginning of the CE, the master sends a data frame (empty or otherwise) on the appropriate frequency. This first frame is called an anchor and is critical for connection synchronization.

5- **Did the Slave Respond?** The slave is not required to respond to the anchor frame. It can remain inactive for a predetermined number of CEs. If the slave responds, the master and slave begin exchanging frames.

6- **Alternate Packets until Connection Event End Condition Met.** Once the slave responds to the anchor, the CE continues, and one or both sides can send data as required, taking turns on who transmits. Each packet contains a sequence number and a next expected sequence number, which serves as a receipt acknowledgement for the previous packet. The length of a CE is determined by how much data are available for transmission, which depends on higher layers in the BLE stack. The CE ends if both parties have no more data or acknowledgments to transmit, if neither party sends a frame within 150 \( \mu s \), or if the connection ends.

Many CEs consist of only two frames: (i) an empty anchor frame from the master and (ii) an empty acknowledgement from the slave. BLE requires frame transmission
(even of empty frames) to periodically synchronize connections and to ensure that both parties are still engaged in communications.

7- **Connection Event Ends.** The end of a CE does not necessarily imply the end of a connection. If the connection has not ended, the master and slave remain inactive until the master sends the anchor of the next CE.

8- **Did the Connection End?** If either party sends an LL TERMINATE IND frame, the connection ends, and the slave starts advertising its presence again.

### 2.4 Tools

This work employs several BLE-specific tools for traffic capture, attack and firmware recovery. As an aid to the reader, a summary of tools is provided in Table 1.

### 2.5 Summary

This chapter presented the state of BLE security and provided a technical overview of the BLE protocol as an aid to the reader. While there has been a recent rise in BLE security research, much of it focuses on attacks and attack frameworks and not on defense of vulnerable devices. Attacks are enabled by the fact that vendors do not implement appropriate link-level security in their applications. This same observation motivates the defensive mechanisms presented by this thesis.

Because link-level security is not robust, a BLE sniffer is a good candidate to detect attacks to these devices. However, sniffers are currently limited to one connection at a time, making them impractical for applications that require monitoring of multiple links. With BLE-Multi, this work creates a practical mechanism to monitor multiple devices. This tool can be the cornerstone of future security appliances that focus on the security of unprotected devices.
Table 1. Summary of BLE tools used throughout this work.

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ubertooth One [42]</td>
<td>Utilities: 2015-10-R1 Firmware: git-579f25c*</td>
<td>Bluetooth sniffer with open-source firmware and hardware; BLE support added in 2013 by Mike Ryan</td>
</tr>
<tr>
<td>Bluefruit LE Sniffer [3]</td>
<td>1.0</td>
<td>Commercial nRF51822-based BLE sniffer</td>
</tr>
<tr>
<td>TI CC2540 Sniffer [52]</td>
<td>2.18.1</td>
<td>Evaluation kit of the CC2540 SoC with pre-loaded sniffer firmware; used with the SmartRF Packet Sniffer software</td>
</tr>
<tr>
<td>BlueZ [8]</td>
<td>5.41</td>
<td>Linux Bluetooth stack, includes utilities to interact with BLE peripherals</td>
</tr>
<tr>
<td>Scapy [55]</td>
<td>2.3.2</td>
<td>Python library used to craft BLE messages</td>
</tr>
<tr>
<td>nRF51822 [38]</td>
<td>-</td>
<td>Multiprotocol SoC used as an all-in-one BLE deployment solution</td>
</tr>
<tr>
<td>nRF51 DK [37]</td>
<td>SDK: 12.0.0 Hardware: 1.2.0</td>
<td>Development platform for the nRF51822 SoC; includes a J-Link SWD programmer and on-board I/O</td>
</tr>
<tr>
<td>J-Link Commander [47]</td>
<td>5.12F</td>
<td>Software to interact with the J-Link SWD programmer on the nRF51 DK</td>
</tr>
<tr>
<td>Bleno [34]</td>
<td>1.0</td>
<td>Node.js module for BLE peripheral implementation</td>
</tr>
</tbody>
</table>

*Two bugs were discovered in Ubertooth firmware that prevented the sniffer from running indefinitely. Both were fixed and reported to the Ubertooth team [22, 23].
III. Securing Bluetooth Low Energy Enabled Industrial Monitors

3.1 Introduction

This chapter addresses investigative questions IQ1 and IQ2 of this work, which focus on identifying the types of vulnerabilities found on BLE devices and the attacks that can be used against them and determining whether BLE sniffers can detect those attacks. To tackle those subtopics, this work first executes an example attack against an industrial monitor, highlighting the vulnerabilities that enable the attack. It follows with a discussion on the challenges of attack detection from the perspective of a peripheral, an end-user and a BLE sniffer, arguing that the sniffer is best positioned for attack detection. Finally, this chapter presents a methodology for attack recovery and provides actionable recommendations for vendors and end-users to enhance BLE security.

The contents of this chapter were originally presented by the author in the Wireless Village at DEF CON 24 [20] and in the proceedings of the Twelfth International Conference on Cyber Warfare and Security [21]. This chapter is structured as follows: Section 3.2 presents an example attack methodology, and Section 3.3 outlines the key challenges with attack detection. Sections 3.4 and 3.5 show a recovery technique and actionable mitigation steps to enhance security, and Section 3.6 concludes.

3.2 Attack Methodology

To determine the types of vulnerabilities that exist on CI devices, this section executes an attack on an HVAC sensor used to monitor the temperature and humidity of an environment. The attacker enumerates the GATT services on the target, noting which are the most important by observing ATT traffic to and from the target.
Focusing on those important services, the attacker determines when and how the
BLE central and BLE peripheral communicate, determines which handles to target
and performs an attack to achieve a desired effect.

### 3.2.1 Selecting the Target

The Onset MX1101 Temperature and Humidity Data Logger [39], shown in Figure
5, is selected as the target. The MX1101 uses the nRF51822 SoC, a common BLE chip
used in many types of applications including several WSNs [28, 16]. Consequently,
some of the vulnerabilities shown here apply to multiple classes of BLE devices. Under
its intended use case, the MX1101, henceforth known as the “Target,” is placed in
an environment to monitor for an extended period of time. A technician interacts
with it using an Android or iOS device through the HOBOMobile Application [40],
henceforth known as the “App.” The App configures the Target, reads and stores
logged data, and updates SoC firmware.

![Onset MX1101 Temperature and Humidity data logger in operation.](image)

**Figure 5.** Onset MX1101 Temperature and Humidity data logger in operation.

### 3.2.2 Performing Enumeration and Reconnaissance

Using BlueZ utilities, the attacker determines that the address of the Target is
D1:55:D5:00:68:C7. She uses a BLE sniffer (e.g., Bluefruit LE) to follow connections
to the Target and capture frames in the conversation. Because the frames contain
visible ATT Write Requests and Notifications, seen in the background of Figure 6,
the SM has not enabled link encryption. In general, many devices do not use link layer encryption. This may be for “ease of use” purposes, as the devices would otherwise require pairing before communicating. Because traffic is not encrypted, the attacker extracts the Target services from the service discovery messages exchanged after connection initiation. The Target BLE server, outlined in Table 2, implements the Device Information Service defined by the Bluetooth SIG and an application-specific service defined by Onset.

Table 2. Default services exposed by the MX1101 BLE server.

<table>
<thead>
<tr>
<th>Service</th>
<th>Handle Range</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic Access</td>
<td>0x0001-0x0007</td>
<td>Required in BLE</td>
</tr>
<tr>
<td>Generic Attribute</td>
<td>0x0008-0x000b</td>
<td>Required in BLE</td>
</tr>
<tr>
<td>Device Information</td>
<td>0x000c-0x0016</td>
<td>Contains identification information about the device; defined by SIG</td>
</tr>
<tr>
<td>Onset Application</td>
<td>0x0017-0xffff</td>
<td>Service used by the HOBOMobile Application; defined by Onset</td>
</tr>
</tbody>
</table>

3.2.3 Reverse Engineering the Application

Through reconnaissance, the attacker determines that handle 0x0019 in the Onset service is a command and control (C2) handle employed by the application. To
authenticate, the App sends the configured password in cleartext to the C2 handle. Because the Target does not use encryption, a BLE sniffer captures the password when it is sent in an ATT Write Request, as shown in Figure 6.

The attacker uses a combination of open source documentation and information extracted from the App to gather details on firmware update capabilities. The Target and App user manuals claim that the Target can perform firmware updates over-the-air. Because the Target uses an nRF51822 SoC, its mechanism to update firmware is through the Device Firmware Update (DFU) Service designed by Nordic Semiconductor, the manufacturer of the SoC [35].

The Target user manual claims that the “Update Firmware” button on the App does not appear unless an update is required [39]. In other words, the DFU Service is not enabled unless the App requests it. To determine what messages are sent by the App to enable the DFU service, the attacker creates a clone of the Target using the Bleno Node.js module. The clone implements the same services from Table 2 and includes enough functionality to deceive the App. It reports an outdated firmware number, which triggers an update prompt from the App. When the attacker presses the “Update Firmware” button, the App sends the command 0x010108010855 to the Target to enable the DFU service.

3.2.4 Attack

After reconnaissance and reverse engineering, the attacker enumerates the vulnerabilities on the Target and select which ones to exploit to achieve an effect. Table 3 lists the vulnerabilities discovered for the Target.

The attack described below is a malicious firmware update on the Target. Figure 7 shows the general process by which the attack is executed. The following sub-sections expand on the functionality of each of the blocks.
Table 3. Vulnerabilities found on the MX1101 (firmware ver. R57-P72.)*

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Default Password</td>
<td>The default logger configuration does not require a password for login. If left misconfigured, an attacker could connect to the device and gain full control of its operation.</td>
</tr>
<tr>
<td>Cleartext Password</td>
<td>The message sent to the device to perform authentication includes the logger password and is sent in plain text. This password could be easily sniffed and used to perform a legitimate login on the device to gain full control of its operation.</td>
</tr>
<tr>
<td>Unhandled Input</td>
<td>Once authenticated, a specific write command can trigger a system reset. The reset stops all data logging and requires reconfiguration.</td>
</tr>
<tr>
<td>Unverified Firmware</td>
<td>The logger does not verify the binary provided during over-the-air firmware updates. An attacker can upload arbitrary firmware to the device to affect device operations.</td>
</tr>
</tbody>
</table>

*Vulnerabilities were reported to Onset Computer Corporation in August 2016.

Figure 7. Firmware upload process determined by reverse engineering the application.

**Connect.** The attacker sends a connection request to the address captured during reconnaissance. The Target cannot authenticate the link-level connection because it does not use pairing/bonding. Furthermore, the Target does not enforce a connection timeout and allows the attacker to remain connected indefinitely. At this point, the Target is under a Denial of Service (DoS) attack because the attacker prevents legitimate users from connecting to the Target.

**Enumerate Services and DFU Enabled?** As in the reconnaissance stage, the attacker enumerates the services on the Target to determine whether the DFU service is enabled. If the DFU service is not enabled, the attacker authenticates with the
Target and sends the firmware update command.

**Authenticate.** The attacker sends an authentication request to the Target using the password captured during the reconnaissance stage. If no password is set (i.e., default operation), she sends an empty authentication request.

**Send Firmware Update Request and Wait for Reset.** The attacker sends the command 0x010108010855 to handle 0x0019, triggering a reboot process on the Target. After the Target resets, it enables the DFU service, which becomes visible to the attacker.

**Upload Firmware.** During the reconaissance stage, the attacker determined that the Target implemented the Nordic Semiconductor DFU Service to update firmware. While the service uses a 16-bit redundancy check to guarantee the integrity of the firmware, it requires no authentication and does not verify the validity of the file. The attacker alters the original firmware file to display custom information in the Device Information Service and downloads it to the Target using a series of ATT Write Commands transmitted using a custom Scapy script. The result of the attack is shown in Figure 8.

![Figure 8. Device information (a) before and (b) after the attack.](image)

While the choice of firmware here is benign, the attacker is not limited to the type of malware she can download. For example, she can create malware that presents itself as the original firmware but feeds fake data back to the App. The malware could cause the user to take action on false alarms or hide critical alarms from view,
potentially damaging the system the sensor is set to monitor. If crafted well, such malware would be difficult to detect by the App and end user. Because DFU is enabled by the App through a command to the Target, the malware can prevent any future firmware updates, significantly increasing the difficulty of device recovery.

3.3 Attack Detection

Three parties are positioned to detect an attack like the one outlined in this work: the Target, the App and a listening third party. The Target can detect an attack if it can determine that a malicious user has connected to it. Options to accomplish this include whitelisting known BLE central addresses and requiring strong link-layer authentication using shared keys. Unfortunately, BLE central addresses can be spoofed, as shown in a recent BLE security evaluation [46], and the Target cannot perform strong authentication because it does not require pairing. An end user may detect the attack if the device behaves erratically with its new firmware. However, a skilled attacker can develop malware that behaves enough like the original firmware to deceive the App and end user.

The most likely method for detection is using BLE snifffers. Since the attack is executed through a series of ATT Write Commands, a sniffer would witness and record the firmware update and could alert an attentive administrator of the change. Presence of snifffers would be difficult to detect by an attacker, as snifters do not transmit any information. However, their use comes with its own challenges. Chapter IV highlights some the challenges of capturing traffic with sniffers and proposes novel mechanisms to overcome them.
3.4 Recovery Methodology

This section provides an overview of the method used to recover the Target after its firmware is replaced. While it is possible to completely recover the Target, the process requires opening the case, soldering onto access pads, creating a device clone using Bleno, and overwriting portions of memory using a special programmer. Not only would it be difficult for a typical user to complete this process, it would only be useful to someone who can detect that the device has been compromised.

The approach consists of re-enabling the DFU service and downloading the original firmware onto the Target. The key insight for recovery is that by exploiting the legacy DFU service, the attacker can only overwrite the application binary, leaving other sections of memory intact. If the application binary is replaced with the original firmware binary, the device returns to its original state. Unfortunately, the command to enable the DFU service, used to perform the attack in the first place, can no longer be used reliably because the new firmware may not know how to interpret it, or worse, may choose not to interpret it. Recovery requires direct interaction with the memory on the SoC.

![Figure 9. Flash memory map for nRF51822 SoC.](image)

Figure 9 shows the memory layout of an nRF51822 SoC with DFU capabilities. If the content at the Application Start address is 0xFFFFFFFF (i.e., empty), the
bootsloader assumes no firmware is present and automatically enables the DFU service. Accordingly, the presented approach overwrites the content at 0x014000 with value 0xFFFFF000. The memory location is accessed directly through the Serial Wire Debug (SWD) [4] programming interface of the nRF51822 chip. On the MX1101, the programming lines (SWDIO and SWDCLK) are connected directly to a set of larger access pads, presumably for programming the device on the manufacturing line. The programmer on the nRF51 DK is connected to the Target as shown in Figure 10. Once physically connected, J-Link Commander is used to create an SWD connection with the Target, write 0xFFFFF000 to the memory address, and perform a soft reboot, enabling the DFU service.

Figure 10. Connections from the MX1101 access pads to the J-LINK pins on the nRF51.

Figure 11. J-Link Commander writing 0xFFFFF000 to the application start address (0x014000).
Once the Target enables the DFU Service, it is ready to receive new firmware. Because a normal end-user is not likely to have a copy of the original firmware, the App is employed to recover the Target. The Bleno clone is again used to trigger a firmware update from the App. After the App sends the command 0x010108010855 to the Bleno clone, it expects the device to reboot with the DFU Service enabled. Since the Target (i.e., the real device) is already actively advertising the DFU Service, the App automatically connects to the Target and uploads the latest vendor firmware, fully recovering the device.

3.5 Securing BLE Devices

This section outlines key actionable steps for both vendors and end users to take to reduce the risk of using BLE. These steps would have prevented the firmware attack presented in this work.

3.5.1 Recommendations for Vendors

Use BLE Encryption. Encryption is enabled by the key exchange performed during pairing and bonding. Vendors should force pairing using the Numerical Comparison method with Elliptical Curve Diffie-Hellman (ECDH) key exchange. Compatibility issues (e.g. devices support don’t support Bluetooth v4.2) can be resolved by using one of the other available pairing methods. In that case, Out of Band (OOB) is preferred, as it is more resilient to brute force, albeit more difficult to deploy. Even the vulnerable Just Works and 6-digit PIN methods are preferable over no encryption at all. With encryption enabled, the eavesdropping attacker would not have learned the device password and would not have been able to reverse engineer the application protocol.
**Provide Stronger Authentication.** Pairing provides a mechanism for device authentication, since LTKs are unique to each pair of devices. Even if data are not encrypted, bonded devices can use the CSRK to sign the data. Vendors should also provide stronger authentication at the application layer, especially if not using BLE encryption. Higher layers could also employ ECDH to exchange encryption or signature keys. Using strong authentication, the Target would have been able to detect that the attacker was not an authorized user and could have ignored the attack commands.

**Implement Security in the DFU Service.** The default Nordic DFU service does not require authentication to accept new firmware. Vendors should add a layer of protection by requiring a signature with the firmware to determine its authenticity. The device bootloader would need to store the vendor public key from a public-private key pair, which it could use to validate the signature. Vendors should also require physical interaction (e.g., a button press/hold, or a switch flip) with the device to perform a firmware update. While this may make it harder to update firmware on devices located in remote environments, it provides an added layer of security that is difficult to defeat by an attacker without physical access.

### 3.5.2 Recommendations for End Users

**Enable Security Wherever Possible.** Users should enable security mechanisms whenever the option is available. This includes, but is not limited to, using passwords, enabling optional pairing, and adding physical security. As an example, the password on the MX1101 caused no hindrance to the user and added a hurdle against an attacker.

**Use Risk Management Strategies.** Use of BLE devices should be approached with risk management strategies. Users should perform routine security evaluations,
especially on those devices that are part of critical systems. With that information, users can determine the amount of risk they incur by employing these sensors and balance it with how much risk they are willing to accept. For example, while a compromised temperature sensor in a break room poses relatively low risk, a compromised pressure sensor on a manufacturing line can have a much larger impact.

**Monitor Devices for Abnormal Behavior.** Some devices provide activity logs that can show abnormal behavior to an attentive administrator. Those should be collected and audited with some regularity. Unfortunately, the attack presented in this paper is an example in which logs cannot be trusted. For critical assets, users should deploy BLE snifffers to monitor communications and generate trusted third-party auditable logs. Unfortunately, current snifffers have limitations that make them impractical for some types of applications. Those limitations, along with enhancements to overcome them are presented in Chapter IV.

### 3.6 Conclusion

This chapter presented the challenges of attack detection on BLE devices. Using an example attack methodology, it illustrated how an industrial monitor is vulnerable to a firmware attack. The monitor cannot detect the attack because it does not require strong authentication, and an end user is unlikely to detect the attack because the malware can behave similarly to the original firmware. The best way to detect an attack is by using a BLE sniffer. While remaining undetectable to the attacker, the sniffer would capture and log the series of ATT commands executed in the attack, and an attentive administrator (or security appliance) could process those logs to detect that the attack took place.

Added layers of security, like encryption at the link layer, stronger authentication and DFU service authentication would have thwarted the attack, but it is the re-
sponsibility of vendors to implement those mechanisms in future applications. In the meantime, end users should use BLE sniffer to audit communications, and approach use of BLE industrial monitors with risk management strategies.
IV. Enabling Bluetooth Low Energy Auditing through Synchronized Tracking of Multiple Connections

4.1 Introduction

The previous chapter described how BLE sniffers can be used to detect attacks. Unfortunately, current sniffers have two main limitations that make them impractical for WSNs and other CI applications. First, current sniffers cannot listen to more than one connection at a time. WSNs can employ many simultaneous connections to share information and would therefore require as many sniffers as there are links to cover all communications. A second limitation is that some sniffers fail to maintain synchronization with active connections, which significantly degrades the capture capabilities of the sniffer over time.

The goal of this chapter is to extend BLE sniffer detection capabilities to monitor WSNs by introducing and evaluating BLE-Multi, a sniffer that can reliably capture multiple connections simultaneously. In doing so, the work tackles IQ3 and IQ4, which focus on enhancing BLE sniffers to better protect WSNs and performing evaluations of available sniffers. More specifically, this chapter achieves the following contributions: (i) improved connection synchronization over other open-source sniffer platforms, (ii) novel multi-connection tracking scheme and implementation, (iii) baseline evaluation of frame capture rates when sniffing multiple connections and (iv) comparison of commercially-available and open source sniffers.

The contents of this chapter have been submitted for publication in the proceedings of the Eleventh Annual IFIP WG 11.10 Conference on Critical Infrastructure Protection [24]. This chapter is structured as follows: Section 4.2 provides an overview of commercially available sniffers. Section 4.3 addresses general design goals and considerations for BLE-Multi. Next, Section 4.4 outlines the process to follow and
synchronize to a single connection, and Section 4.5 presents the process for tracking more than one connection. Section 4.6 evaluates BLE-Multi against commercially-available sniffers and forms a baseline for frame capture rates when sniffing more than one connection, while Section 4.7 provides a top-level analysis of evaluation results as they relate to auditing.

4.2 Problems with Current BLE Sniffers

Figure 12 shows commonly used BLE sniffers. The TI CC2540 and the Bluefruit LE are pre-loaded with proprietary sniffer firmware and are designed to aid with application debugging. Note that Nordic Semiconductor provides sniffer firmware that can be installed on any BLE radio chip in the nRF51 family [36], to include the nRF51 Development Kit (DK) [37] and the nRF51822 chip [38] used in the Adafruit Bluefruit LE platform. This work focuses only on the Adafruit Bluefruit LE as a representation of Nordic Semiconductor sniffer firmware.

As an open source BLE sniffer, the Ubertooth One can be easily flashed with the latest firmware from the Ubertooth online code repository [18] or with custom firmware for specialized applications. BLE-Multi extends the latest Ubertooth One firmware to achieve synchronized sniffing of multiple connections.

Table 4 highlights the key differences and limitations of the three currently available sniffers, as compared to BLE-Multi. While capabilities vary across sniffers, the single limitation shared among all currently available sniffers is the restriction to monitor a single active BLE connection. When using a sniffer as traffic analysis tool for debugging or troubleshooting, this restriction makes sense because it maximizes the probability of capturing every frame in the conversation. However, when using sniffers under a risk management framework or for a security appliance, it may be acceptable to sacrifice the probability of frame capture in exchange for other more
Figure 12. Commercially-available Bluetooth Low Energy sniffers.

Table 4. Comparison of commercially-available sniffers and BLE-Multi.

<table>
<thead>
<tr>
<th>Sniffer</th>
<th>Platform</th>
<th>Firmware</th>
<th>Connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>TI CC2540 [52]</td>
<td>Windows</td>
<td>Proprietary</td>
<td>One, synchronized</td>
</tr>
<tr>
<td>Ubertooth One [42]</td>
<td>Linux</td>
<td>Open source</td>
<td>One, unsynchronized</td>
</tr>
<tr>
<td>BLE-Multi</td>
<td>Linux</td>
<td>Open source</td>
<td>Multiple, synchronized</td>
</tr>
</tbody>
</table>

important capabilities.

A second limitation affecting only the Ubertooth One is that its current firmware does not employ mechanisms to maintain synchronization with an active connection. Connection synchronization is necessary because clocks on different devices can drift from each other. Because BLE communications require interactions on specific frequencies at specific times, the BLE specification provides a procedure to account for clock drift. Unfortunately, the procedure relies on frame directionality, which is not inherent through frame inspection, and is therefore difficult to extract by a sniffer.

This work removes these limitations by proposing firmware for a multi-connection BLE traffic capture platform and a novel adaptation of the BLE synchronization procedure to enhance the Ubertooth One.

4.3 Design Considerations

This section discusses the assumptions and considerations that drive the design of BLE-Multi as an extension to the Ubertooth One firmware.
4.3.1 General Considerations

The goal of a sniffer is to capture every frame sent between the master and slave. Recall that the master controls the link layer parameters that control the connection. Thus, in order to follow a connection, a sniffer can simply act like the slave, capturing and interpreting link layer control messages from the master. As a relayer of data, a sniffer is not required to process data intended for higher layers in the BLE stack; instead, it can indiscriminately forward those data to the host or end user.

One unique situation arises with the encryption procedure. When a master or slave enables encryption, both sides use a symmetric 128-bit key and AES-128 block cipher to perform encryption at the link layer. Traffic captured by a sniffer, including link control messages, would be incomprehensible to the sniffer and end user. This work focuses on securing devices that do not employ encryption and therefore assumes that neither master or slave enables encryption.

Another key assumption is that a sniffer is a passive third-party participant in the conversation. This assumption implies that the sniffer cannot transmit packets to alter the conversation in any way. For example, it cannot request packet re-transmissions, negotiate more convenient connection parameters or otherwise actively interfere with the connection.

An additional implication of its third-party status is that a sniffer cannot easily determine the directionality of an observed frame because frames do not encode sender or receiver information. Such information is obvious in a two-party conversation and is therefore omitted (i.e. if the slave receives a frame, it must have come from the master and vice versa). A passive third-party listener cannot make the same inference. This creates problems with the BLE synchronization mechanism because it relies on the timing of anchor frames sent by the master. Section 4.4.1 addresses this issue by proposing a synchronization mechanism that relies on consecutive empty frames.
Finally, this work does not seek to overcome physical limitations imposed by radio frequency (RF) communications. It does not address radio factors that affect how well it can decode bit sequences from RF energy, nor does it address the effects of interference and noise on the crowded 2.4 GHz ISM band. It abstracts factors like signal-to-noise ratio and bit errors as an overall degradation in the likelihood of frame capture.

### 4.3.2 Ubertooth One Platform Considerations

This work builds BLE-Multi as an extension to the current Ubertooth One firmware for two reasons. First, Ubertooth One is a flexible open-source hardware platform that lends itself for rapid proof-of-concept development. Second, the Ubertooth One is widely used in the security community because it is one of the only affordable means to capture Bluetooth Classic traffic. Thus, development on this platform can generate meaningful impact to the security community.

The current firmware, named `bluetooth-rxtx`, is capable of sniffering Bluetooth Classic and BLE connections [18]. Its main mode of operation relies on clock interrupts to drive channel hops and radio retuning. Although the available clock resolution for timestamps is 100 ns, clock interrupts run 3,125 times slower, at a frequency of 3,200 Hz. While it is possible to increase its frequency by writing to registers on the on-board CC2400 radio [51], this work seeks to have minimal impact on all other functionality of `bluetooth-rxtx`. Hence, BLE-Multi is restricted to a clock interrupt precision of 312,500 ns. To account for the relatively low precision, BLE-Multi makes conservative approximations when calculating its deadlines and timers. For example, when calculating the timer related to a connection interval, BLE-Multi rounds down to the nearest factor of 312,500 ns, ensuring that the radio performs a hop before the connection interval actually begins. In contrast, when calculating the timer related
to the length of a connection event, BLE-Multi rounds up, conservatively extending the amount of time the radio spends on the channel.

One final implementation consideration of the Ubertooth One is that frame timestamps are taken after a frame is completely received, but many calculations in BLE require knowledge about the timing of the first bit in the frame. Adding to the problem, it takes time for the radio front end to capture the frame and forward it to the processor. Since there is currently no built-in method to know when a packet is first received, BLE-Multi subtracts a constant from every frame timestamp. This constant encapsulates the frame transmission delay and any other processing delays, and it was experimentally tuned throughout the firmware development process.

4.4 Following a Single Connection

In theory, a sniffer need only act like a slave to follow an active connection. In practice, a sniffer does not have the benefit of active participation in a connection. Although this limitation removes its responsibilities in link layer control, it consequently limits its capability to synchronize to a connection. This section highlights how BLE-Multi addresses connection synchronization.

4.4.1 Synchronization

Clock synchronization is essential to BLE communications because BLE employs frequency hopping triggered on time intervals. Poor synchronization can have adverse effects on sniffing, as described by the following example.

Suppose the master has clock drift of 250 parts per million (ppm), normal for a standard Broadcom/Cypress BRCM20702 BLE transceiver [11] used in many commercial BLE dongles. The 16 MHz oscillator in the Ubertooth One is advertised to have a clock drift of 20 ppm [2]. In the worst case, after just 45 seconds, the clocks
on the Ubertooth One and the master could have drifted a full 12.15 ms from each other. A normal frame is transmitted in less than 150 µs, meaning that the sniffer would have potentially missed some (or all) frames across multiple connection events.

The following subsections describe the built-in BLE synchronization mechanism and the adaptations made in BLE-Multi to combat this problem.

4.4.1.1 Synchronization in BLE

The BLE specification addresses the issue of clock drift with a synchronization mechanism based on the timing of messages from the master. Using the connection request, the master informs the slave of its Sleep Clock Accuracy (SCA), or clock drift. Once a slave receives an anchor (i.e., first packet sent by master), it predicts when the next anchor will arrive using the connection interval:

\[ \text{nextExpectedAnchor} = \text{lastAnchor} + \text{connectionInterval} \]

Because the master and slave clocks can drift apart, the slave calculates a window of time during which it should listen for the next anchor. Since the slave knows its own SCA \((SCA_s)\) and the SCA of the master \((SCA_m)\), it can calculate the window of time for the next anchor based on the maximum amount of time the clocks could have drifted. The BLE specification defines this as window widening [7]:

\[ \text{windowWidening} = \frac{SCA_m + SCA_s}{1,000,000} \times (\text{nextExpectedAnchor} - \text{lastAnchor}) \]

As illustrated in Figure 13, the slave listens for \(\text{windowWidening}\) before and \(\text{windowWidening}\) after \(\text{nextExpectedAnchor}\). Each time a new anchor arrives, the slave resets \(\text{lastAnchor}\), narrowing \(\text{windowWidening}\). Note that it is the responsibility of the slave, not the master, to maintain synchronization. By extension, it is the
responsibility of the sniffer to maintain synchronization with the master. However, this presents unique challenges for third-party listeners.

Figure 13. Relationship between anchors, connection parameters and BLE-Multi timers.

4.4.1.2 Synchronization in Sniffers

Since the slave synchronizes to the master, the sniffer does not need to know $SCA_S$ of the actual slave; it simply needs to use its own sleep clock accuracy ($SCA_{Ubertooth}$). However, as previously discussed, anchors are not marked in any way. The sniffer does not know and cannot infer whether a frame came from the master or the slave based strictly on frame content. Clearly, this becomes a problem when calculating $windowWidening$.

One approach is to assume the first frame received by the sniffer is an anchor. However, if that frame is not actually an anchor, the sniffer incorrectly approximates $nextExpectedAnchor$, causing problems on successive CEs. Another approach is to follow frame sequence numbers and make guesses about missed or dropped packets, but such an approach is not practical for this work. BLE-Multi could deliberately choose not to listen to a CE in lieu of another connection, causing the sniffer to lose track of sequence numbers.

While it may not be possible to determine an anchor for every CE, one specific sequence of frames guarantees the presence of an anchor. If two frames are received consecutively (per [7], within 150 $\mu$s of each other), and they both contain no data.
and have the More Data (MD) bit set to 0, then the first frame is an anchor. Claim 1 presents a proof of this assertion.

**Claim 1.** If two consecutive frames, \( f_1 \) and \( f_2 \), have no data and \( MD = 0 \), then \( f_1 \) is an anchor.

*Proof.* Suppose that \( f_1 \) is not an anchor. By definition, this means that at least one frame was transmitted before \( f_1 \). Let the frame that was transmitted immediately before \( f_1 \) be \( f_0 \). \( MD_{f_0} \) is either 0 or 1:

- **Case 1:** \( MD_{f_0} = 0 \). In this case, the sender of \( f_0 \) will not send any more data frames. Since \( f_1 \) has no data and \( MD_{f_1} = 0 \), both parties will interpret \( f_1 \) as the end of the connection event, and neither side will send any more frames. However, this contradicts the assumption that \( f_2 \) was sent. It follows that Case 1 cannot occur based on the BLE specification [7].

- **Case 2:** \( MD_{f_0} = 1 \). In this case, the sender of \( f_0 \) will transmit more data the next time it sends a frame. Since the master and slave alternate sending frames, the data will be transmitted in the slot for \( f_2 \). But, this contradicts the assumption that \( f_2 \) has no data. It follows that Case 2 cannot occur based on [7].

Since neither Case 1 nor Case 2 can occur, \( f_0 \) could not have been transmitted. Therefore, \( f_1 \) must be the first frame in the communication, and by definition, an anchor. \( \square \)

Under this assumption, BLE-Multi cannot extract an anchor from every connection event like a slave would be able to do. However, the sequence of frames described above occurs frequently while the link layer awaits data from higher layers. It is therefore a feasible mechanism to maintain connection synchronization. When the sniffer observes the sequence of empty frames, it updates *lastAnchor*, which automatically
narrow$\text{s }\windowWidening$.

In practice, BLE-Multi applies synchronization by using two timers that decrement with every clock interrupt: (i) $\intervalTimer$ and (ii) $\connEventTimer$. When $\intervalTimer$ expires, BLE-Multi triggers the BLE radio to hop onto and begin listening on the next channel in the hopping scheme. At that point, the sniffer recalculates $\windowWidening$ and updates $\intervalTimer$ as:

$$\intervalTimer = \nextExpectedAnchor - \windowWidening - \currentTime$$

and it enables $\connEventTimer$, which is set to:

$$\connEventTimer = 2 \times \windowWidening$$

The end of $\connEventTimer$ signifies the latest time at which the next anchor could arrive. If at any point BLE-Multi receives a frame before $\connEventTimer$ expires, it assigns 150 $\mu s$ to $\connEventTimer$, which is the maximum amount of time without transmission before a connection event must end.

### 4.4.2 Link Layer Control

Acting as a slave, a sniffer must appropriately react to link layer control frames, specifically those that dictate or affect connection parameters. Tables 5 and 6 list critical commands sent on advertisement and data channels, respectively, along with actions required by the sniffer. Note that a sniffer only needs to take action on 3 of 7 types of frames sent on advertisement channels and 3 of 22 types sent on data channels. These six types of frames are critical because they signal changes to the underlying connection. To clarify, this does not mean that the sniffer does not capture other types of frames – it does. It simply does not take any action.
Table 5. Critical advertisement channel messages and their effects on BLE connections.

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Effect on Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADV_IND</td>
<td>Advertiser is connectable</td>
</tr>
<tr>
<td></td>
<td><strong>Action:</strong> Capture CONNECT_REQ.</td>
</tr>
<tr>
<td>ADV_DIRECT_IND</td>
<td>Advertiser is connectable</td>
</tr>
<tr>
<td></td>
<td><strong>Action:</strong> Capture CONNECT_REQ.</td>
</tr>
<tr>
<td>CONNECT_REQ</td>
<td>Sender requests connection initiation with preceding advertiser</td>
</tr>
<tr>
<td></td>
<td><strong>Action:</strong> Extract connection parameters and actively follow connection</td>
</tr>
</tbody>
</table>

Table 6. Critical data channel control messages and their effects on BLE connections.

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Effect on Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>LL_CONNECTION_UPDATE_REQ</td>
<td>Master dictates a change in connection parameters</td>
</tr>
<tr>
<td></td>
<td><strong>Action:</strong> Apply new connection parameters</td>
</tr>
<tr>
<td>LL_CHANNEL_MAP_REQ</td>
<td>Master dictates a change in channel map</td>
</tr>
<tr>
<td></td>
<td><strong>Action:</strong> Apply new channel map</td>
</tr>
<tr>
<td>LL_TERMINATE_IND</td>
<td>Sender terminates the connection</td>
</tr>
<tr>
<td></td>
<td><strong>Action:</strong> Stop following the connection</td>
</tr>
</tbody>
</table>

4.5 Following Multiple Connections

A sniffer can take advantage of the characteristics of BLE connections to opportunistically follow multiple conversations. This section shows how BLE-Multi is structured to accomplish this task.

4.5.1 Procedure

Figure 3 showed the empty space after a CE that can be used to listen to other connections. By simultaneously tracking the state of multiple active connections, BLE-Multi can choose a different connection after a CE ends. Figure 14 illustrates how the empty space between CEs can be utilized more efficiently by a sniffer cap-
turing three simultaneous connections. CEs on the left side of the Scheduler have already been captured, and CEs on the right side are expected to occur in the future.

![Figure 14. Connection Event Scheduler in BLE-Multi.](image)

**Algorithm 1:** Schedule. Determine the next link to capture based on the time remaining before the next Connection Event.

```
1: procedure Schedule(activeLinks)
   2:   \textbf{Input:} the current set of active links being tracked, activeLinks
   3:   \textbf{Output:} the next link to capture
   4:   \hspace{1em} minTimer \leftarrow \infty
   5:   \hspace{1em} minLink \leftarrow \text{NULL}
   6:   \hspace{1em} for link \in activeLinks do \hspace{1em} \triangledown \text{Find the link with minimum timer.}
   7:   \hspace{2em} if link.intervalTimer < minTimer then
   8:   \hspace{3em} minLink \leftarrow link
   9:   \hspace{3em} minTime \leftarrow link.intervalTimer
   10:  \hspace{1em} if minTimer \geq \text{MIN\_ADV\_TIME} then \hspace{1em} \triangledown \text{If there’s enough time to listen for advertisements, do so.}
   11:     \hspace{2em} return advertisementLink
   12:   \hspace{1em} else
   13:     return minLink
   14: end
```

At the end of each CE, the Scheduler selects the next CE to capture using the procedure in Algorithm 1. BLE-Multi also employs an advertisementLink, which it treats as a default link. If there are no active links, or if there is still enough time before the next CE, the Scheduler selects the advertisementLink, which is always configured to track an advertisement channel. This is how the sniffer adds new connections to the set of active links.
Currently, the Scheduler selects the link with the most imminent CE (i.e., the link with minimum \textit{intervalTimer}). However, the sniffer is not limited in the selection algorithm it uses. For example, another implementation may select the link that has been active the longest (oldest first) or the link that has been captured least recently (longest since last capture). Each resulting algorithm would have unique implications on the data capture capabilities for each link.

4.5.2 Limitations

To be clear, the physical limitation of the BLE radio still exists: the sniffer is only capable of listening to one channel at a time. However, because CEs do not normally utilize the entire Connection Interval, BLE-Multi is able to timeshare across multiple connections. This is in contrast to [41], which performs wideband sniffing of Bluetooth Classic communications by capturing multi-channel chunks of the 2.4-GHz spectrum and extracting the Bluetooth conversation via post-processing. This physical limitation of sniffing one channel at a time manifests itself when sniffing multiple connections with multiple masters.

If active connections are initiated by separate masters, there is a probability that CEs in separate connections will overlap. BLE-Multi is forced to choose one connection over the other, and the overall frame capture rate may drop. By contrast, connections that share a master are optimal for BLE-Multi because the master automatically deconflicts CEs. In that case, events do not overlap, and BLE-Multi can listen to a greater number of events. However, even if BLE-Multi encounters multiple connections with multiple masters, CEs often consist of empty frames (no data); this was the insight that allowed BLE-Multi to synchronize with active connections. Hence, missed CEs do not necessarily imply missed data.

Finally, the master may change connection parameters in the middle of the con-
versation. When listening to multiple active connections, BLE-Multi has greater probability of missing a parameter update, which could cause the sniffer to fall out of synchronization with the corresponding link.

4.6 Evaluation

Sniffer evaluation is inherently difficult because there is no mechanism to extract the number of link layer frames transmitted or received by transceivers in BLE dongles. Ideally, evaluation would consist of comparing the number of frames captured by the sniffer with the number of frames actually transmitted. Unfortunately, dongles hide low-level communications from the host, including empty frames and link control frames, disqualifying the most obvious metric of percentage of frames captured. Instead, the experiments in this work target the reception of usable and actionable data embodied by a single command sent from the master to the slave at a random time point after connection start.

4.6.1 Experiment Design

The three experiments in this work use the same experiment setup, procedure and data analysis techniques. The following paragraphs highlight the generic design of these experiments.

**Setup.** Figure 15 provides an overview of the experiment setup. A laptop with a 2.4-GHz Intel Core i7 processor, 8 GB RAM, and Kali Rolling 2016.1 OS, controls the timing and data collection for each experiment. The sniffers under test (SUT) are connected to a powered USB 2.0 Hub that is connected directly to the laptop. Two BLE dongles (D1 and D2) with BCM20702 transceivers [11] are connected to an additional powered USB 2.0 Hub, which is also connected to the control laptop. D1 and D2 are placed 30 cm away from one Onset MX1101 Temperature/Humid-
ity sensor and one August Smartlock. These two targets represent different classes of critical infrastructure applications: environmental sensing and building security. Additionally, these targets do not implement short timeouts on active connections, meaning that each trial can run for over 45 seconds without receiving an automatic disconnect from the target.

Figure 15. Generic experiment setup (numbered bubbles indicate active links for Experiments 1, 2 and 3).

Figure 15 also illustrates the active links for each of the three experiments. Experiment 1 only establishes the link between a single dongle and a single target (D1→MX1101), Experiment 2 connects a single dongle to two targets (D1→MX1101 and D1→August), and Experiment 3 uses two separate dongles to connect to two targets (one each) (D1→MX1101 and D2→August).

Trials. This work aggregates trials to extract the likelihood of a frame being captured at increasing times in the BLE connection. Data points at increasing times enable validation of the synchronization techniques introduced by this work. A Python script controls the experiments, where each trial consists of the following:

1. Establish required connections (shown in Figure 15) in pseudorandom order using random.randint() in Python.
2. Wait for a pseudorandomly-selected delay of 0, 15, 30, or 45 seconds.

3. Send a single target frame with byte sequence 0x010108010855 to each slave in the order connections were established.

4. Disconnect from each slave in the order connections were established.

Table 7 summarizes the control, independent, and dependent variables in each experiment. Experiment 1 runs 500x4 trials, while Experiments 2 and 3 run 150x4 trials due to time limitations. A trial is deemed successful if the sniffer captures the target frame with the specified byte sequence. Because trials are not conducted in a Faraday cage, electromagnetic interference could impact the capture success of the SUTs. To reduce the effects of unpredictable interference for any particular time delay, the message delay for each trial is determined pseudo-randomly.

Table 7. Summary of control, independent and dependent variables for experiments.

<table>
<thead>
<tr>
<th>Control Variables</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance, BLE dongles and targets</td>
<td>30 cm</td>
</tr>
<tr>
<td>Distance, BLE sniffers and active links</td>
<td>1 m</td>
</tr>
<tr>
<td>Sniffers under test</td>
<td>TI CC2540, Bluefruit LE, Ubertooth One, BLE-Multi</td>
</tr>
<tr>
<td>Number of trials</td>
<td>4x500 (Exp 1), 4x150 (Exp 2/3) (due to time)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Independent Variables</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of active links</td>
<td>1 (Exp 1), 2 (Exp 2/3)</td>
</tr>
<tr>
<td>Configuration of active links</td>
<td>D1→MX1101 (Exp 1), D1→MX1101 and D1→August (Exp 2), D1→MX1101 and D2→August (Exp 3)</td>
</tr>
<tr>
<td>Time of ATT message in each trial</td>
<td>0, 15, 30, or 45 sec</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dependent Variables</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Capture success/failure of ATT message by each sniffer, per trial</td>
<td></td>
</tr>
</tbody>
</table>

**Calculating Success Proportion.** Trials are aggregated to calculate a success proportion for each time delay. The number of trials in an experiment, \( n \), represents
a single population sample of length $n$. The sample proportion, $\hat{p}_i$ is calculated using the number of successful captures $s_i$ at time delay $i \in \{0, 15, 30, 45\}$ and $n$:

$$\hat{p}_i = \frac{s_i}{n_i}$$

The resulting $\hat{p}_i$ is used as an unbiased estimator for the proportion to calculate a 95% confidence interval:

$$\hat{p}_i \pm 1.96 \left( \sqrt{\frac{\hat{p}_i(1 - \hat{p}_i)}{n}} \right)$$

The calculated confidence interval represents how well a sniffer can capture frames at time delay $i$. Performing this calculation across four points in the connection (0, 15, 30 and 45 seconds) shows how the effectiveness of each sniffer changes with time.

### 4.6.2 Experiment 1: One Master to One Slave

This experiment compares success proportions for four different SUTs over the length of a single active connection. The SUTs are the BLE-Multi, the Ubertooth One, the TI 2540 sniffer and the Adafruit Bluefruit LE sniffer. Figure 16 and Table 8 show the results of Experiment 1.

Of note, both the TI 2540 and Bluefruit LE perform the best in capture success and synchronization. Both are closed-source commercial sniffers that use hardware specifically designed for BLE communications. This is in contrast to the open-source Ubertooth One (and by extension, BLE-Multi), which tunes an older generic 2.4-GHz radio not specifically designed for BLE. The difference in hardware could account for the gap in performance between the open-source and commercial sniffers. However, this work uses the Ubertooth One platform because of its flexibility for rapid proof-of-concept development and its impact to the security community.

The decreasing trend line on the success proportion of the Ubertooth One is
Table 8. Raw results for Experiment 1 (one master to one slave).

<table>
<thead>
<tr>
<th></th>
<th>TI CC2540</th>
<th>Bluefruit LE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i</td>
<td>Successes</td>
</tr>
<tr>
<td>0</td>
<td>499/500</td>
<td>0.998</td>
</tr>
<tr>
<td>15</td>
<td>498/500</td>
<td>0.996</td>
</tr>
<tr>
<td>30</td>
<td>500/500</td>
<td>1.000</td>
</tr>
<tr>
<td>45</td>
<td>497/500</td>
<td>0.994</td>
</tr>
</tbody>
</table>

Table 8. Raw results for Experiment 1 (one master to one slave).

<table>
<thead>
<tr>
<th></th>
<th>Ubertooth One</th>
<th>BLE-Multi</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i</td>
<td>Successes</td>
</tr>
<tr>
<td>0</td>
<td>469/500</td>
<td>0.938</td>
</tr>
<tr>
<td>15</td>
<td>409/500</td>
<td>0.818</td>
</tr>
<tr>
<td>30</td>
<td>362/500</td>
<td>0.724</td>
</tr>
<tr>
<td>45</td>
<td>314/500</td>
<td>0.628</td>
</tr>
</tbody>
</table>

Figure 16. Frame capture success of a single connection over time (95% CI for proportion).

likely due to the lack of connection synchronization mechanism. Conversely, the non-decreasing trend line on the BLE-Multi data suggests that the adapted synchronization mechanism that this work introduces achieves synchronization over the length of the connection. An apparent side effect of BLE-Multi is that while it achieves overall synchronization and stability, its capture success is 5-15% lower than the Ubertooth
One for connections lasting less than 15 seconds. This could be due to the active use of advertisementLink, where the sniffer moves away from an active connection to listen to an advertisement channel, or it could stem from an expanded processor payload as a result of the added logic. There may be ways to optimize BLE-Multi to surpass the Ubertooth One, but this work focuses instead on developing the proof-of-concept mechanisms to perform synchronization and capture multiple connections.

4.6.3 Experiment 2: One Master to Multiple Slaves

This experiment tests the effectiveness of BLE-Multi on two simultaneous connections between a single master and multiple slaves. Using the generic setup, the

<table>
<thead>
<tr>
<th>i</th>
<th>MX1101 Sensor Successes</th>
<th>MX1101 Sensor 95% CI</th>
<th>August Smartlock Successes</th>
<th>August Smartlock 95% CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>129/150</td>
<td>0.860 ± 0.056</td>
<td>122/150</td>
<td>0.813 ± 0.062</td>
</tr>
<tr>
<td>15</td>
<td>133/150</td>
<td>0.887 ± 0.051</td>
<td>128/150</td>
<td>0.853 ± 0.057</td>
</tr>
<tr>
<td>30</td>
<td>130/150</td>
<td>0.867 ± 0.054</td>
<td>122/150</td>
<td>0.813 ± 0.062</td>
</tr>
<tr>
<td>45</td>
<td>127/150</td>
<td>0.847 ± 0.058</td>
<td>120/150</td>
<td>0.800 ± 0.064</td>
</tr>
</tbody>
</table>

Figure 17. Frame capture success of two simultaneous connections with one master (95% CI for proportion).
experiment uses a single SUT with BLE-Multi, a single BLE dongle and both targets.

A single master connected to two slaves automatically deconflicts both connections by ensuring no overlap between connection events. In other words, these two connections are optimal for simultaneous capture because BLE-Multi never has to miss a connection event in favor of another. Table 9 shows the raw results from the experiment, while Figure 17 illustrates the same results and compares them to the proportion of successful trials of the Ubertooth One in Experiment 1.

The stable BLE-Multi data trends for the MX1101 and August SmartLock show that BLE-Multi is successfully able to capture and maintain synchronization with both connections simultaneously. Notably, on long connections, BLE-Multi on multiple connections performs better than Ubertooth One on a single connection.

### 4.6.4 Experiment 3: Multiple Masters to Multiple Slaves

This experiment tests the effectiveness of BLE-Multi on two simultaneous connections between two masters and two slaves (one slave per master). Employing the generic setup, the experiment uses a single SUT with BLE-Multi, two BLE dongles and both targets. Figure 18 and Table 10 show the results of Experiment 3. The results for the Ubertooth One from Experiment 1 are again added for reference.

Use of two master-slave pairs adds the possibility of overlapping connection events. In that case, BLE-Multi could be forced to deliberately ignore a connection event and potentially miss data. However, the results here show that, as in Experiment 2, BLE-Multi is able to capture data from and maintain synchronization with both connections simultaneously. Under the conditions in this experiment, overlapping events did not have a statistically significant effect on BLE-Multi capabilities.

One contributing factor is that the connections observed in this experiment are relatively quiet; that is, most CEs consist of only two empty packets. This allows
Table 10. Raw results for Experiment 3 (two masters to two slaves, one each).

<table>
<thead>
<tr>
<th>i</th>
<th>MX1101 Sensor</th>
<th>August Smartlock</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Successes</td>
<td>successes</td>
</tr>
<tr>
<td>0</td>
<td>130/150</td>
<td>126/150</td>
</tr>
<tr>
<td>15</td>
<td>123/150</td>
<td>128/150</td>
</tr>
<tr>
<td>30</td>
<td>135/150</td>
<td>127/150</td>
</tr>
<tr>
<td>45</td>
<td>128/150</td>
<td>134/150</td>
</tr>
</tbody>
</table>

Figure 18. Frame capture success of two simultaneous connections with two masters (95% CI for proportion).

the sniffer more time between CEs and consequently, more time to listen to other connections. Following more than two connections, or more active connections, would likely yield a lower proportion of successful trials per connection.

4.7 Analysis

Results from Experiments 2 and 3 support the claim that BLE-Multi is capable of following and capturing more than one simultaneous BLE connection. Furthermore, the non-decreasing trend lines in all BLE-Multi data support the claim that the synchronization mechanism introduced in this work, which relies on frequent transmissions of empty frames, is a practical solution for long-lived connections. Note that
both proprietary snifffers are also able to synchronize to long-lived connections, but BLE-Multi can do so for multiple connections at the same time.

Because of its synchronization mechanism, BLE-Multi (on multiple connections) can capture frames better than the Ubertooth One (on a single connection) for connections lasting longer than 15 seconds. However, on connections lasting less than 15 seconds, BLE-Multi is 5-15% less likely than the Ubertooth One to capture frames. Since BLE-Multi is an extension to Ubertooth One firmware, the drop in trial success is likely due to the added workload required to capture multiple connections. Even then, BLE-Multi captured target frames with a capture success of more than 70% in all evaluated cases.

Admittedly, while these proportions of successful frame captures may be acceptable for security applications that aim to generally bolster security, they may not be acceptable for applications that require greater scrutiny of traffic. Ideally, each of the simultaneous BLE connections would be captured by one commercial sniffer, maximizing the likelihood of frame capture. Unfortunately, such a system of sniffers may be difficult to scope, design and deploy, as the number of active connections in an environment would not be known in advance. Cost-benefit analyses should be performed for specific applications to determine whether BLE-Multi is an acceptable traffic capture solution. In the end, BLE-Multi provides security professionals the flexibility necessary to audit evolving BLE environments by capturing more than one active conversation at a time while simultaneously listening for new connections.

4.8 Conclusion

This chapter presented BLE-Multi, a firmware update to the Ubertooth One platform that can capture multiple BLE connections simultaneously. Code for BLE-Multi firmware is included in Appendix A and should be used in conjunction with
the other files in the Ubertooth One repository. BLE-Multi was evaluated against other commercially available sniffers on a single active connection, where it successfully demonstrated synchronization to long-lived connections and outperformed other open-source sniffers. BLE-Multi was also evaluated on multiple connections with varying master-slave configurations. Even under the most challenging configuration (multiple masters to multiple slaves), BLE-Multi successfully captured a majority of the traffic in both connections, while outperforming an Ubertooth One tuned to a single connection.

BLE-Multi is a viable auditing solution for CI applications, including WSNs, that use many communication links simultaneously. A single BLE-Multi sniffer could track and listen to multiple active connections and create logs that an administrator or security appliance can audit for malicious or unusual activity. While those logs by themselves would not provide active defense of deployed sensors, they are a step closer to automated WSN/BLE defensive tools.
V. Future Work and Conclusion

This chapter highlights the key contributions of this work and lays out areas for future research, placing emphasis on further enhancing and employing BLE-Multi to provide more robust defensive measures. The work focused on the research question, “Can BLE traffic sniffers be used to enhance CI security?,” by individually addressing four investigative questions. Contributions presented in this thesis support the hypothesis that the security of CI and WSNs can be enhanced with a BLE sniffer that detects attacks on multiple simultaneous connections.

IQ1: What vulnerabilities exist on BLE devices used in CI? Chapter III demonstrated an attack on a temperature/humidity HVAC monitor that exposed its vulnerabilities and exploited them to download malware onto the monitor. Unfortunately, the level of security in BLE devices depends on the implementation by the manufacturer. Vendors should enforce pairing/binding to enable device authentication and link encryption, but they often choose to leave the link layer unprotected. Users should approach BLE usage with risk management strategies, minimizing risk by enabling as much optional security as possible, and balancing the amount of risk they take with the amount of risk they are willing to accept.

IQ2: Can BLE sniffers detect attacks against BLE devices? Chapter III also presented the challenges of attack detection from the perspective of a peripheral, an end-user and a BLE sniffer. The peripheral is not well-positioned to detect an attack because it cannot differentiate between an attacker and a legitimate user, and the end user is unlikely to detect an attack if it does not visibly affect the device functionality. By contrast, the BLE sniffer provides enough visibility into traffic (i.e., shows ATT commands) to expose anomalous behavior to an administrator or security appliance.
IQ3: Can current BLE sniffers be altered to capture multiple connections to enhance their impact to WSNs? Current traffic sniffers can only follow one connection at a time, making them impractical for applications that employ more than one simultaneous connection. Additionally, some open-source sniffers lack the functionality to synchronize to active connections, causing a drastic drop in capture success over the length of a connection. Chapter IV introduced BLE-Multi, which successfully extended sniffer capabilities to capture and synchronize with more than one simultaneous connection. This upgrade directly enhances the impact of BLE sniffers on WSN security.

IQ4: How can the performance of BLE sniffers be measured? The most natural way to evaluate sniffers is to compare the total number of sniffed frames with the total number transmitted between devices. Unfortunately, BLE transceivers do not provide insight into the total number of frames they exchange with other devices, making it inherently difficult to evaluate sniffers. This work presented an evaluation technique that targeted the reception of usable data, embodied by single ATT Write Command sent from master to slave. The evaluation showed that BLE-Multi could successfully capture multiple connections, maintaining synchronization throughout the connection and outperforming other open-source sniffers.

5.1 Use Case for BLE-Multi

A facility manager deploys an Ubertooth One sniffer with BLE-Multi firmware in a small building to monitor its HVAC wireless sensors. BLE-Multi captures communications to and from all of the sensors in the building, even if the sensors communicate simultaneously. At the end of the day, the administrator downloads and audits the security logs from BLE-Multi, paying particular attention to ATT Read Requests and ATT Write Requests. She finds that sometime in the afternoon, a device connected
to one of the temperature/humidity monitors and interacted directly with the over-
the-air firmware update service implemented by the monitor. The logs show that
the device downloaded a large file onto the monitor, after which the monitor started
to advertise unusual data. She immediately removes the sensor from operation and
alerts security personnel of the attack.

In the future, an IPS could automatically perform this process. The IPS would
continuously monitor traffic to and from any devices of interest, searching for any
known attack patterns. For example, if the IPS detects interactions with the firmware
update service, it could begin jamming the connection, disconnecting the would-be
attacker. At the same time, the IPS would send email alerts to security personnel,
highlighting the attack traffic that triggered the response.

5.2 Future Work

Future research can focus on tuning BLE-Multi to improve frame capture, as well
as automating auditing and detection mechanisms to provide more robust defensive
measures. The following subsections capture some of the future work to be done in
BLE/WSN security.

5.2.1 Vulnerability Assessments

While this work provides an example of the types of vulnerabilities found in BLE
devices, it is not an absolute enumeration of all types of vulnerabilities. Research
should continue to thoroughly assess devices to better understand the risks of using
BLE as well as the potential threat vectors that attackers can use. This information
will continue to inform the development of defensive security tools.
5.2.2 Sniffer Tuning

BLE-Multi uses several constant parameters that affect timing calculations. These parameters can be experimentally tuned to improve sniffer reliability. Additionally, the period between clock interrupts can be shortened to increase the precision of \textit{intervalTimer} and \textit{connEventTimer}. A more precise clock will enable tighter windows around connection events and will allow more time for capturing other links. Once tuned, future work can form more practical comparisons between BLE-Multi and commercial sniffers, specifically in the areas of advertisement and connection request capture (i.e. how good is BLE-Multi at hopping onto a new connection?).

5.2.3 Capability Expansion

The BLE advertisement mechanism allows a slave to use between one and three advertisement channels. The slave sends an advertisement in each advertisement channel, in order of increasing frequency, and a master is allowed up to 150 \(\mu\text{s}\) to respond. A tuned, more precise, BLE-Multi should be able to follow the advertising device as it moves through the advertisement channels, increasing the probability of capturing a connection request, and consequently, the probability of following a connection.

Another area of capability expansion is in the selection algorithms employed by the Scheduler. Other algorithms may prioritize links that have been active the longest (oldest first), while other may select links that have been “waiting” the longest to capture (longest since last capture). Different algorithms would have different implications on how well BLE-Multi captures data.
5.2.4 Mobile Masters and Slaves

Mobile masters and slaves pose a challenge to traffic capture tools because the relative position and orientation between the target transceivers and sniffer antenna are constantly changing. With the wide availability of BLE in mobile devices (e.g., smartphones and tablets), future research should analyze the impact to BLE-Multi from mobile attackers and mobile targets.

5.2.5 Intrusion Detection System (IDS)

WSN IDSs can be challenging to deploy because they normally require a significant power draw from the sensors themselves [49]. BLE-Multi provides the traffic visibility necessary to perform intrusion detection without drawing additional power from the sensor network. Specifically, the following attacks may be detected by sniffers:

- **Man in the Middle.** If the IDS detects that BLE-Multi follows two distinct active connections to the same “target” address, an attacker may be placing herself between the real target device and the victim.

- **Password Attack.** The IDS may correlate multiple connection events corresponding to the same connection to determine that an attacker is attempting a dictionary or brute force password attack.

- **Firmware Attacks.** An IDS filter on data higher in the BLE protocol stack, including ATT Read and Write commands. Commands to the firmware update services described in Chapter III would indicate a firmware change.

- **Rogue Devices.** The BLE protocol requires that all connectable targets advertise their presence. Thus, an IDS could register when an unknown device begins advertising in its proximity.
5.2.6 Intrusion Protection System (IPS)

Using a jam-based denial mechanism, similar to that shown in [14], an IDS could actively block connections it determines to be malicious, turning an IDS to an IPS.

5.3 Conclusion

BLE sniffer can be used to enhance the security of CI applications by exposing the traffic generated by an attack. For WSNs and other applications that require capture of multiple simultaneous links, BLE-Multi is a viable monitoring solution. A single BLE-Multi sniffer can track and listen to active conversations and create logs that an administrator can audit for malicious or unusual activity. In the future, those logs could be forwarded to automated defensive solutions that require granular visibility into BLE traffic. Those tools could mirror other defensive systems, like IDSs and IPSs, used in other types of communications networks. In the meantime, end users should approach use of BLE in CI with risk management strategies, understanding that its use comes with a risk that they must be willing to accept.
Appendix A. BLE-Multi Source Code

The following sections include source code for BLE-Multi, a firmware enhancement to the Ubertooth One capture tool for Bluetooth Low Energy traffic. Enhancements from these firmware files include: (i) multi-connection capture capabilities; (ii) enhanced clock synchronization; and (iii) updated host tools to provide command-line interfaces with BLE-Multi. The following files from the Ubertooth One code repository (github.com/greatscottgadgets/ubertooth) are altered and included below:

- firmware/bluetooth_rxtx/bluetooth_le.c
- firmware/bluetooth_rxtx/bluetooth_le.h
- firmware/bluetooth_rxtx/bluetooth-rxtx.c
- firmware/bluetooth_rxtx/ubertooth_clock.h
- firmware/common/ubertooth.h
- host/libubertooth/src/ubertooth_control.c
- host/libubertooth/src/ubertooth_interface.h
- host/libubertooth/src/ubertooth.c
- host/libubertooth/src/ubertooth.h
- host/ubertooth-tools/src/ubertooth-btle.c
1.1 firmware/bluetooth_rxtx/bluetooth_le.c

#include "bluetooth_le.h"

extern u8 le_channel_idx;
extern u8 le_hop_amount;

u16 btle_next_hop(le_state_t *le) {
    u16 phys = btle_channel_index_to_phys(le->channel_idx);
    le->channel_idx = (le->channel_idx + le->channel_increment) % 37;
    return phys;
}

u8 btle_channel_index(u8 channel) {
    u8 idx;
    channel /= 2;
    if (channel == 0)
        idx = 37;
    else if (channel < 12)
        idx = channel - 1;
    else if (channel == 12)
        idx = 38;
    else if (channel < 39)
        idx = channel - 2;
    else
        idx = 39;
    return idx;
}

u8 btle_afh_channel_index(le_state_t* l) {
    u8 idx = 0;

    /* Is the channel an advertisement channel? */
    if (l->channel_idx == 37 || l->channel_idx == 38 ||
        l->channel_idx == 39) { idx = l->channel_idx; }

    /* Is the unmapped channel in use? */
else if (l->channel_map[l->channel_idx] != 0x00) {
    idx = l->channel_idx;
} /* Unmapped channel is not used, let's map it to a used channel */
else {
    int used_channel_idx = l->channel_idx % l->num_used_channels;
    int search_idx = -1, i = -1;
    while (search_idx != used_channel_idx && i < 36) {
        i++;
        if (l->channel_map[i] != 0x00) search_idx++;
    }
    idx = i;
}
return idx;

u16 btle_channel_index_to_phys(u8 idx) {
    u16 phys;
    if (idx < 11)
        phys = 2404 + 2 * idx;
    else if (idx < 37)
        phys = 2428 + 2 * (idx - 11);
    else if (idx == 37)
        phys = 2402;
    else if (idx == 38)
        phys = 2426;
    else
        phys = 2480;
    return phys;
}

/* calculate CRC */
/* note 1: crc_init's bits should be in reverse order */
/* note 2: output bytes are in reverse order compared to wire */
/* */
/* example output: */
/* 0x6ff46e */
/* */
/* bytes in packet will be: */
/* { 0x6e, 0xf4, 0x6f } */
/* */
u32 btle_calc_crc(u32 crc_init, u8 *data, int len) {
    u32 state = crc_init & 0xffffffff;
    u32 lfsr_mask = 0x5a600000; /* 0101101010001100000000000000000 */
    int i, j;
    for (i = 0; i < len; ++i) {
        u8 cur = data[i];
        for (j = 0; j < 8; ++j) {
            int next_bit = (state ^ cur) & 1;
            cur >>= 1;
            state >>= 1;
            if (next_bit) {
                state |= 1 << 23;
                state = lfsr_mask;
            }
        }
    }
}
return state;

/* runs the CRC in reverse to generate a CRCInit */
/* */
/* crc should be big endian */
/* the return will be big endian */
/* */

u32 btle_reverse_crc(u32 crc, u8 *data, int len) {
  u32 state = crc;
  u32 lfsr_mask = 0xb4c000; /* 101101001100000000000000 */
  u32 ret;
  int i, j;

  for (i = len - 1; i >= 0; --i) {
    u8 cur = data[i];
    for (j = 0; j < 8; ++j) {
      int top_bit = state >> 23;
      state = (state << 1) & 0xffffff;
      state |= top_bit ^ ((cur >> (7 - j)) & 1);
      if (top_bit)
        state ^= lfsr_mask;
    }
  }

  ret = 0;
  for (i = 0; i < 24; ++i)
    ret |= ((state >> i) & 1) << (23 - i);

  return ret;
}

u32 btle_crc_lut[256] = {
  0x000000, 0x01b4c0, 0x036980, 0x02dd40, 0x06d300, 0x0767c0, 0x05ba80, 0x040e40, 0x0d
  a600, 0x0c12c0, 0x0ecf80, 0x0f7b40, 0x0b7500, 0x0ac1c0, 0x081c80, 0x09a840, 0x1b4c00, 0x1af
  8c0, 0x182580, 0x199140, 0x1d9f00, 0x1c2bc0, 0x1ef680, 0x1f4240, 0x16ea00, 0x175ec0, 0x158
  380, 0x143740, 0x103900, 0x118dc0, 0x135080, 0x12e440, 0x369800, 0x372cc0, 0x35f180,
  0x344540, 0x304b00, 0x31ffc0, 0x332280, 0x329640, 0x3b3e00, 0x3a8ac0, 0x385780, 0x39e
  340, 0x3ded00, 0x3c59c0, 0x3e8480, 0x3f3040, 0x2dd400, 0x2c60c0, 0x2ebd80, 0x2f0940, 0x2b
  0700, 0x2ab3c0, 0x286e80, 0x29da40, 0x207200, 0x21c6c0, 0x231b80, 0x22af40, 0x26a100, 0x2
  715c0, 0x25c880, 0x247c40, 0x6d
  3000, 0x6c84c0, 0x6e5980, 0x6fed40, 0x6be300, 0x6a57c0, 0x688a80, 0x693e40, 0x609600,
  0x6122c0, 0x63ff80, 0x624b40, 0x664500, 0x67f1c0, 0x652c80, 0x649840, 0x6767c0, 0x67c8c0,
  0x611bc0, 0x73c680, 0x727240, 0x7bda00, 0x7a6ec0, 0x78b380, 0x790740, 0x7d0900, 0x76d000,
  0x76f440, 0x597540, 0x5d7b00, 0x5ccfc0, 0x5e1280, 0x5fa640, 0x55d0e0, 0x57bac0, 0x556780,
  0x543d40, 0x50dd00, 0x5169c0, 0x53ff80, 0x52af40, 0x56a100, 0x55c880, 0x547c40, 0x40e
  400, 0x4150c0, 0x438d80, 0x423940, 0x463700, 0x4783c0, 0x455e80, 0x44ea40, 0x4d4200,
  0x4fc6c0, 0x4e2b80, 0x4fb9100, 0x4a25c0, 0x48f880, 0x494c40, 0xda6000, 0xdbd4c0, 0xd90980,
  0x40e000, 0xda6f90, 0xdcb300, 0xdd07c0, 0xde6e40, 0xd7c680, 0xda67c0, 0xda4af0, 0xd51b50,
  0xda1150, 0xda0a10, 0xda27c0, 0xda3c80, 0xda12c0, 0xda09c0, 0xda2580, 0xda3f140, 0xda7f00,
  0xda64c0, 0xda4960, 0xda5220, 0xda8ca0, 0xdc3ec0, 0xcf380, 0xce5740, 0xca5900, 0xcbedc0,
Calculate a BTLE CRC one byte at a time. Thanks to Dominic Spill & Michael Ossmann for writing and optimizing this.

Arguments: CRCInit, pointer to start of packet, length of packet in bytes

```c
u32 btle_crcgen_lut(u32 crc_init, u8 *data, int len) {
    u32 state;
    int i;
    u8 key;

    state = crc_init & 0xffffffff;
    for (i = 0; i < len; ++i) {
        key = data[i] ^ (state & 0xff);
        state = (state >> 8) ^ btle_crc_lut[key];
    }

    return state;
}
```
1.2 firmware/bluetooth_rxtx/bluetooth_le.h

/*
 * Copyright 2016 Air Force Institute of Technology, U.S. Air Force
 * Copyright 2012 Dominic Spill
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 * You should have received a copy of the GNU General Public License
 * along with this program; see the file COPYING. If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110-1301, USA.
 */

#include "ubertooth.h"

/* General BLE parameters */
#define BTLE_CHANNELS 40
#define ADVERTISING_CHANNELS 3
#define DATA_CHANNELS 37

/* Byte locations of data within every LE packet. */
#define ADV_ADDRESS_IDX 0
#define HEADER_IDX 4
#define DATA_LEN_IDX 5
#define DATA_START_IDX 6

/* Byte locations of data within a CONNECT_REQ packet */
#define CRC_INIT (2+4+6+6+4)
#define WIN_SIZE (2+4+6+6+4+3)
#define WIN_OFFSET (2+4+6+6+4+3+1)
#define CONN_INTERVAL (2+4+6+6+4+3+1+2)
#define CHANNEL_MAP (2+4+6+6+4+3+1+2+2+2+2)
#define CHANNEL_INC (2+4+6+6+4+3+1+2+2+2+2+5)

typedef enum {
    LINK_INACTIVE,
    LINK_LISTENING,
    LINK_CONN_PENDING,
    LINK_CONNECTED,
} link_state_t;

typedef struct le_state_t {
    /* Access Address to filter by */
    u32 access_address;
    /* Access address in CC2400 syncword format */
    u16 synch;
    /* lower 16 bits thereof */
    u16 syncl;
}
/* CrcInit: used to calculate CRC */

u32 crc_init;

/* bits-reversed version of the previous */

u32 crc_init_reversed;

/* true to reject packets with bad CRC */

int crc_verify;

/* current link layer state */

link_state_t link_state;

/* current channel index */

u8 channel_idx;

/* amount to hop */

u8 channel_increment;

/* unused if 0, else used */

u8 channel_map[37];

/* total number of used channels */

u8 num_used_channels;

/* reference time for the start of the connection */

u32 conn_epoch;

/* number of intervals remaining before next hop */

u32 volatile interval_timer;

/* connection-specific hop interval */

u32 conn_interval;

/* number of intervals since the start of the connection */

int volatile conn_count;

/* Master's sleep clock accuracy in PPM
* (20, 30, 50, 75, 100, 150, 250, 500) */

u16 sca;

/* window size */

u32 win_size;

/* offset of first window from start of connection */

u32 win_offset;

/* whether a connection update is pending */

int update_pending;

/* the connection count when the update takes effect */

u16 update_instant;

/* the new hop_internal */

u32 interval_update;

/* the new window size */

u32 win_size_update;

/* the new window offset */

u32 win_offset_update;

/* whether a channel map update is pending */

int map_update_pending;

/* unused if 0, else used */

u8 channel_map_update[37];

/* total number of used channels in the updated map */

u8 num_used_channels_update;

/* target MAC for connection following (byte order reversed) */

u8 target[6];

/* whether a target has been set (default: false) */

int target_set;

/* when was the last packet received */

u32 last_packet;
/* time of the last event start in CLK100NS precision */
static const u8 whitening[] = {
    1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0,
    0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0,
    0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0,
    1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1,
    1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0,
    1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1
};

static const u8 whitening_index[] = {
    70, 62, 120, 111, 77, 46, 15, 101, 66, 39, 31, 26, 80,
    83, 125, 89, 10, 35, 8, 54, 122, 17, 33, 0, 58, 115, 6,
    94, 86, 49, 52, 20, 40, 27, 84, 90, 63, 112, 47, 102
};

static const u8 hop_interval_lut[] = {
    0, 1, 19, 25, 28, 15, 16, 14, 33, 26, 27, 34, 20, 8,
    5, 7, 24, 35, 2, 13, 30, 32, 29, 17, 3, 10, 11, 4, 23, 21,
    6, 22, 9, 12, 18, 36,
};

u16 btle_next_hop(le_state_t *le);
u8 btle_afh_channel_index(le_state_t* le);
u8 btle_channel_index(u8 channel);
u16 btle_relative_channel_index(u8 phys);
u32 btle_calc_crc(u32 crc_init, u8 *data, int len);
u32 btle_reverse_crc(u32 crc, u8 *data, int len);
u32 btle_crcgen_lut(u32 crc_init, u8 *data, int len);

static const u32 whitening_word[40][12] = {
    {0xc3bc9240, 0x5f9a371f, 0x9a9c9f685f, 0x44c5d6c1, 0xe1de5920, 0xb066a73da8, 0x113175b0, 0x9a9c9f685f, 0x44c5d6c1, 0xe1de5920, 0xb066a73da8, 0x113175b0},
    {0xabc92409, 0x5f9a371f3c, 0x9a9c9f685f, 0x44c5d6c19a, 0x9a9c9f685f, 0x44c5d6c19a, 0x9a9c9f685f, 0x44c5d6c19a, 0x9a9c9f685f, 0x44c5d6c19a, 0x9a9c9f685f, 0x44c5d6c19a},
    {0x3da157d2, 0x75b066a7, 0x9a9c9f685f, 0x44c5d6c19a, 0x9a9c9f685f, 0x44c5d6c19a, 0x9a9c9f685f, 0x44c5d6c19a, 0x9a9c9f685f, 0x44c5d6c19a, 0x9a9c9f685f, 0x44c5d6c19a},
    {0x42afa51b, 0x6dcd4e7b, 0x9a9c9f685f, 0x44c5d6c19a, 0x9a9c9f685f, 0x44c5d6c19a, 0x9a9c9f685f, 0x44c5d6c19a, 0x9a9c9f685f, 0x44c5d6c19a, 0x9a9c9f685f, 0x44c5d6c19a},
    {0x3f877964, 0x0a9c9f685f, 0x44c5d6c19a, 0x9a9c9f685f, 0x44c5d6c19a, 0x9a9c9f685f, 0x44c5d6c19a, 0x9a9c9f685f, 0x44c5d6c19a, 0x9a9c9f685f, 0x44c5d6c19a, 0x9a9c9f685f, 0x44c5d6c19a},
};
/∗ LE Multi Connection Stuff ∗/

typedef enum {
ANCHOR_SEARCH,
ANCHOR_CANDIDATE
} anchor_state_t;

204 {0xcb240898, 0xa371fc3b, 0xcf6855f4, 0x5d6c19a9, 0xe592044c, 0x51b8fe1d, 
205 0xe42afa51, 0xb60cd4e7, 0xc902262e, 0xda73da57, 0xb240898b, 0x371fc3bc
},
206 {0x5d6c19a9, 0xe592044c, 0x51b8fe1d, 0xe7b42afa, 0x2eb60cd4, 0xf2c90226, 
207 0xd28dc7f0, 0xda73da57, 0x371fc3bc, 0x5d6c19a9, 0xe592044c, 0x51b8fe1d
},
208 {0xe42afa51, 0xb60cd4e7, 0xc902262e, 0xda73da57, 0xb240898b, 0x371fc3bc
},
209 {0x371fc3bc, 0x6e3f8779, 0xed0abe94, 0xad833539, 0x2044c5d6, 0x8fe1de5
},
210 {0xe42afa51, 0xb60cd4e7, 0xc902262e, 0xda73da57, 0xb240898b, 0x371fc3bc
},
211 {0x371fc3bc, 0x6e3f8779, 0xed0abe94, 0xad833539, 0xb240898b, 0x371fc3bc
},
212 {0xe42afa51, 0xb60cd4e7, 0xc902262e, 0xda73da57, 0xb240898b, 0x371fc3bc
},
213 {0x371fc3bc, 0x6e3f8779, 0xed0abe94, 0xad833539, 0xb240898b, 0x371fc3bc
};
typedef enum {
    ADV_SEARCH,
    ADV_CANDIDATE
} adv_state_t;

#define F_ADVERTISEMENT 0x01
#define F_CONN_EVENT 0x02
#define F_DECISION 0x04
#define F_CONN_UPDATE 0x08
#define F_INTERVAL 0x10
1.3 firmware/bluetooth_rtxx/bluetooth-rtxx.c

/*
 * Copyright 2016 Air Force Institute of Technology, U.S. Air Force
 * Copyright 2010−2013 Michael Ossmann
 * Copyright 2011−2013 Dominic Spill
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110−1301, USA.
 */

#include <string.h>
#include "ubertooth.h"
#include "ubertooth_usb.h"
#include "ubertooth_interface.h"
#include "ubertooth_rssi.h"
#include "ubertooth_cs.h"
#include "ubertooth_dma.h"
#include "ubertooth_clock.h"
#include "bluetooth.h"
#include "bluetooth_le.h"
#include "cc2400_rangetest.h"
#include "ego.h"

/******************** ****************** DEFINES ******************/
#define MIN(x,y) (((x)<(y)?(x):(y))
#define MAX(x,y) (((x)>(y)?(x):(y))
#define DIVIDE_ROUND(N, D) ((N) + (D)/2) / (D)
#define DIVIDE_CEIL(N, D) (1 + ((N−1) / D))
#define AA_LIST_SIZE (int) (sizeof(le_promisc.active_aa) / sizeof(le_promisc_active_aa_t))
/* Max # of links to follow in multi-link LE mode */
#define MAX_LINKS 5
#define MAX_INACTIVE_LINK_TIME 5 /* in seconds */
#define JAM_COUNT_DEFAULT 40
#define MAXREAD_REG 0x2d
#define MAX_NUM_TARGETS 1
#define MIN_ADV_INTERVALS 3
/* # of clk intervals needed to capture one frame */
#define PACKET_OFFSET 2
/* # of clk intervals until we pass 150us */
#define T_IFS 5 /* tuned to 5 which is >>> 150us */
/**************************** GLOBALS *****************************/
volatile u32 debug_clk100ns_count = 0;
/* build info */
const char compile_info[] = "ubertooth " GIT_REVISION " ( " COMPILERS " @ " COMPILERS_HOST " ) " TIMESTAMP;

/* hopping stuff */
volatile uint8_t hop_mode = HOP_NONE;
volatile uint8_t do_hop = 0;
volatile uint16_t channel = 2441;
volatile uint16_t hop_direct_channel = 0;
volatile uint16_t hop_timeout = 158;
volatile uint16_t requested_channel = 0;
volatile uint16_t saved_request = 0;

/* bulk USB stuff */
volatile uint8_t idle_buf_clkns_high = 0;
volatile uint32_t idle_buf_clkns_low = 0;
volatile uint16_t idle_buf_channel = 0;
volatile uint8_t dma_discard = 0;
volatile uint8_t status = 0;

/* operation mode */
volatile uint8_t mode = MODE_IDLE;
volatile uint8_t requested_mode = MODE_IDLE;
volatile uint8_t jam_mode = JAM_NONE;
volatile uint8_t ego_mode = EGO_FOLLOW;
volatile uint8_t modulation = MOD_BT_BASIC_RATE;

/* specan stuff */
volatile uint8_t low_freq = 2400;
volatile uint16_t high_freq = 2483;
volatile int8_t rssi_threshold = -30;

/* Generic TX stuff */
generic_tx_packet tx_pkt;

/* le stuff */
uint8_t slave_mac_address[6] = { 0, };
le_state_t le = { .access_address = 0x8e89bed6, /* advertising channel access address */ .synch = 0x6b7d, /* bit-reversed adv channel AA */ .sync = 0x9171, .crc_init = 0x555555, /* advertising channel CRCInit */ .crc_init_reversed = 0xaaaaaa, .crc_verify = 0,
.link_state = LINK_INACTIVE, .conn_epoch = 0, .target_set = 0, .last_packet = 0,
};

typedef struct le_promisc_active_aa_t {
    u32 aa;
    int count;
} le_promisc_active_aa_t;
```
typedef struct _le_promisc_state_t {
    // LFU cache of recently seen AA's */
    le_promisc_active_aa_t active_aa[32];
    /* recovering hop interval */
    u32 smallest_hop_interval;
    int consec_intervals;
} le_promisc_state_t;
le_promisc_state_t le_promisc;
/* LE Multiple Connection Following Stuff */
/*
 * This structure has two timers/timeouts:
 *   decision_timer – predetermined timeout that forces the handler
 *       to "move on"
 *   conn_event_timer – calculated timeout, occurs when no packet is
 *       received for 150 us
 */
typedef struct _le_multi_link_hdlr_t {
    /* Container to hold all connection states */
    le_state_t links[MAX_LINKS];
    /* Generic advertisement listener */
    le_state_t adv_link;
    /* Pointer to the connection currently being followed */
    le_state_t *cur_link;
    /* Number of targets set by the user */
    u8 num_targets;
    /* target MACs for connection following (byte order reversed) */
    u8 targets[MAX_NUM_TARGETS][6];
    /* State of the anchor-capturing SM */
    anchor_state_t cur_anchor_state;
    /* CLK100NS timestamp of latest candidate packet for anchor */
    u32 candidate_anchor;
    /* Enable/disable decision timer */
    int decision_timer_en;
    /* # of CLOCK intervals until we have to make a decision */
    u16 volatile decision_timer;
    /* Enable/disable the conn_event_timer, useful when listening to ADV */
    int conn_event_timer_en;
    /* if 0, the current connection event has invalidated */
    u16 volatile conn_event_timer;
    /* Enable/disable advertisement timer */
    int adv_timer_en;
    /* If 0, it's time to move to a different adv channel */
    u16 volatile adv_timer;
    adv_state_t cur_adv_state;
    /* target MAC for advertisement following (byte order reversed) */
    u8 adv_target[6];
    /* if 1, we should run the scheduler */
    int scheduler_en;
} le_multi_link_hdlr_t;
le_multi_link_hdlr_t le_hdlr = {
    .links = {
        .access_address = 0x8e89bed6,
        .synch = 0x6b7d,
```
.syncl = 0x9171,
.crc_init = 0x555555,
.crc_init_reversed = 0xAAAAAA,
.crc_verify = 0,
.link_state = LINK_INACTIVE,
.conn_epoch = 0,
.target_set = 0,
.last_packet = 0,
.channel_map = {0x00,},
.num_used_channels = 0,
.last_confirmed_event = 0,
.next_expected_event = 0,
.linger_time = 0,
.window_widening = 0,
}

.adv_link = {
.access_address = 0x8e89bed6, /* advertising channel AA */
.synch = 0x6b7d, /* bit-reversed adv channel AA */
.syncl = 0x9171,
.crc_init = 0x555555, /* advertising channel CRCInit */
.crc_init_reversed = 0xAAAAAA,
.crc_verify = 0,
.link_state = LINK_LISTENING,
.channel_idx = 37,
.channel_increment = 0,
.conn_epoch = 0,
.target_set = 0,
.last_packet = 0,
}

.u8 hop_reason = 0;

/* LE jamming Stuff */
.int le_jam_count = 0;

/* General BT/LE Stuff */
typedef int (*data_cb_t)(char *);
data_cb_t data_cb = NULL; /* Pointer to data handler */
typedef void (*packet_cb_t)(u8 *);
packet_cb_t packet_cb = NULL; /* Pointer to frame handler */

/* Unpacked symbol buffers (two rxbufs) */
char unpacked[DMA_SIZE*8*2];

/**************************** FUNCTION PROTOTYPES ****************************/
static int enqueue(uint8_t type, uint8_t* buf);
int enqueue_with_ts(u8_t type, u8_t* buf, uint32_t ts);
int debug_log(u32 message);
static int vendor_request_handler(u8_t request, \
    uint16_t* request_params, u8_t* data, int* data_len);
void TIMER0_IRQHandler(void);
static void msleep(uint32_t millis);
void DMA_IRQHandler(void);
static void cc2400_idle(void);
static void cc2400_rx(void);
static void cc2400_rx_sync(u32 sync);
static void cc2400_tx_sync(uint32_t sync);
void le_transmit(u32 aa, u8 len, u8 *data);
void le_jam(void);
void hop(void);
void bt_stream_rx(void);
static uint8_t reverse8(uint8_t data);
static uint16_t reverse16(uint16_t data);
void br_transmit(void);
void le_set_access_address(u32 aa, le_state_t *conn);
void reset_le(le_state_t *conn);
void le_multi_scheduler(void);
void reset_le_promisc(void);
void bt_generic_le(u8 active_mode);
void bt_le_sync(u8 active_mode);
void bt_multi_le_sync(void);
int cb_follow_le(char *unpacked);
void connection_follow_cb(u8 *packet);
void connection_multi_follow_cb(u8 *packet);
void bt_follow_le(void);
void bt_multi_follow_le(void);
void le_promisc_state(u8 type, void *data, unsigned len);
void promisc_recover_hop_increment(u8 *packet);
void promisc_recover_hop_interval(u8 *packet);
void promisc_follow_cb(u8 *packet);
void see_aa(u32 aa);
int cb_le_promisc(char *unpacked);
void bt_promisc_le(void);
void bt_slave_le(void);
void rx_generic_sync(void);
void rx_generic(void);
void tx_generic(void);
void specan(void);
void led_specan(void);
void le_multi_reset_interval_timer(le_state_t *link);
void le_multi_cleanup(void);
void le_multi_update_adv_state(void);

/******************** MAIN **********************/
int main() {
    ubertooth_init();
    clkn_init();
    ubertooth_usb_init(vendor_request_handler);
    cc2400_idle();
    while (1) {
handle_usb(clkn);
if (requested_mode != mode) {
    switch (requested_mode) {
    case MODE_RESET:
        /* Allow time for the USB command to return correctly */
        wait(1);
        reset();
        break;
    case MODE_AFH:
        mode = MODE_AFH;
        bt_stream_rx();
        break;
    case MODE_RX_SYMBOLS:
        mode = MODE_RX_SYMBOLS;
        bt_stream_rx();
        break;
    case MODE_TX_SYMBOLS:
        mode = MODE_TX_SYMBOLS;
        br_transmit();
        break;
    case MODE_BT_FOLLOW:
        mode = MODE_BT_FOLLOW;
        bt_stream_rx();
        break;
    /* Ubertooth Sniffing request */
    case MODE_BT_FOLLOW_LE:
        bt_follow_le();
        break;
    case MODE_BT_MULTIFOLLOW_LE:
        bt_multi_follow_le();
        break;
    case MODE_BT_PROMISC_LE:
        bt_promisc_le();
        break;
    case MODE_BT_SLAVE_LE:
        bt_slave_le();
        break;
    case MODE_TX_TEST:
        mode = MODE_TX_TEST;
        cc2400_txtest(&modulation, &channel);
        break;
    case MODE_RANGE_TEST:
        mode = MODE_RANGE_TEST;
        cc2400_rangetest(&channel);
        requested_mode = MODE_IDLE;
        break;
    case MODE_REPEATER:
        mode = MODE_REPEATER;
        cc2400_repeater(&channel);
        break;
    case MODE_SPECAN:
        specan();
        break;
    case MODE_LED_SPECAN:
        led_specan();
        break;
    case MODE_EGO:
mode = MODE_EGO;
    ego_main(ego_mode);
    break;
  case MODE_RX_GENERIC:
    mode = MODE_RX_GENERIC;
    rx_generic();
    break;
  case MODE_TX_GENERIC:
    tx_generic();
    break;
  case MODE_IDLE:
    cc2400_idle();
    break;
  default:
    /* This is really an error state, but what can you do? */
    break;
  }
}

/**************************** Function Definitions ****************************/

/* Function: enqueue
 * Puts something in the DMA buffer to send to host.
 * type:
 * buf:
 */

static int enqueue(uint8_t type, uint8_t *buf) {
  usb_pkt_rx* f = usb_enqueue();

  /* fail if queue is full */
  if (f == NULL) {
    status |= FIFO_OVERFLOW;
    return 0;
  }

  f->pkt_type = type;
  if (type == SPECAN) {
    f->clkn_high = (clkn >> 20) & 0xff;
    f->clk100ns = CLK100NS;
  } else {
    /*f->clkn_high = idle_buf_clkn_high; */
    /*f->clk100ns = idle_buf_clk100ns; */
    f->clkn_high = (clkn >> 20) & 0xff;
    f->clk100ns = CLK100NS;
    f->channel = ((idle_buf_channel - 2402) & 0xff);
    f->rssi_min = rssi_min;
    f->rssi_max = rssi_max;
    f->rssi_avg = rssi_get_avg(idle_buf_channel);
    f->rssi_count = rssi_count;
  }

  memcpy(f->data, buf, DMA_SIZE);
  f->status = status;
status = 0;
    return 1;
  }

/* Function: enqueue_with_ts */
/* Puts something in the DMA buffer to send to host, include timestamp. */
* - type:
* - buf:
* - ts:
int enqueue_with_ts(uint8_t type, uint8_t* buf, uint32_t ts) {
  usb_pkt_rx* f = usb_enqueue();
  /* fail if queue is full */
  if (f == NULL) {
    status |= FIFO_OVERFLOW;
    return 0;
  }
  f->clkn_high = 0;
  f->clkn100ns = ts;
  f->channel = (uint8_t)((channel - 2402) & 0xff);
  f->rssi_avg = 0;
  f->rssi_count = 0;
  memcpy(f->data, buf, DMA_SIZE);
  f->status = status;
  status = 0;
  return 1;
}

/* Function: debug_log */
/* Send debug messages back to host via DMA queue. Message shows up as */
/* the access address of the packet. The DMA queue must already be */
/* initialized for this to work! */
* - message:
int debug_log(u32 message) {
  uint8_t type = LE_PACKET;
  usb_pkt_rx* f = usb_enqueue();
  /* fail if queue is full */
  if (f == NULL) {
    status |= FIFO_OVERFLOW;
    return 0;
  }
  f->pkt_type = type;
  f->clkn_high = idle_buf_clkn_high;
  f->clkn100ns = idle_buf_clkn100ns;
  f->channel = (uint8_t)((idle_buf_channel - 2402) & 0xff);
  f->rssi_min = rssi_min;
```c
f->rssi_max = rssi_max;
f->rssi_avg = rssi_get_avg(idle_buf_channel);
f->rssi_count = rssi_count;
uint32_t* p = (uint32_t*) f->data;
p[0] = message;
return 1;
}

/*
 * Function: vendor_request_handler
 * Respond to requests from host to set global parameters and start
 * sniffing.
 * - request:
 * - request_params:
 * - data:
 * - data_len:
 */
static int vendor_request_handler(uint8_t request, 
        uint16_t* request_params, uint8_t* data, int* data_len) {
    uint32_t command[5];
    uint32_t result[5];
    uint64_t t_copy;
    uint32_t tlock;
    size_t length; /* string length */
    usb_pkt_rx* p = NULL;
    uint16_t reg_val;
    uint8_t i;

    switch (request) {
    case UBERTOOTH_PING:
        *data_len = 0;
        break;
    case UBERTOOTH_RX_SYMBOLS:
        requested_mode = MODE_RX_SYMBOLS;
        *data_len = 0;
        break;
    case UBERTOOTH_TX_SYMBOLS:
        hop_mode = HOP_BLUETOOTH;
        requested_mode = MODE_TX_SYMBOLS;
        *data_len = 0;
        break;
    case UBERTOOTH_GET_USRLED:
        data[0] = (USRLED) ? 1 : 0;
        *data_len = 1;
        break;
    case UBERTOOTH_SET_USRLED:
        if (request_params[0])
            USRLED_SET;
        else
            USRLED_CLR;
        break;
    case UBERTOOTH_GET_RXLED:
        data[0] = (RXLED) ? 1 : 0;
        *data_len = 1;
        break;
    case UBERTOOTH_SET_RXLED:
```
if (request_params[0])
  RXLED_SET;
else
  RXLED_CLR;
break;
case UBERTOOTH_GET_TXLED:
  data[0] = (TXLED) ? 1 : 0;
  *data_len = 1;
  break;
case UBERTOOTH_SET_TXLED:
  if (request_params[0])
    TXLED_SET;
  else
    TXLED_CLR;
  break;
case UBERTOOTH_GET_1V8:
  data[0] = (CC1V8) ? 1 : 0;
  *data_len = 1;
  break;
case UBERTOOTH_SET_1V8:
  if (request_params[0])
    CC1V8_SET;
  else
    CC1V8_CLR;
  break;
case UBERTOOTH_GET_PARTNUM:
  get_part_num(data, data_len);
  break;
case UBERTOOTH_RESET:
  requested_mode = MODE_RESET;
  break;
case UBERTOOTH_GET_SERIAL:
  get_device_serial(data, data_len);
  break;
#endif
#ifdef UBERTOOTH_ONE
  case UBERTOOTH_GET_PAEN:
    data[0] = (PAEN) ? 1 : 0;
    *data_len = 1;
    break;
case UBERTOOTH_SET_PAEN:
    if (request_params[0])
      PAEN_SET;
    else
      PAEN_CLR;
    break;
case UBERTOOTH_GET_HGM:
    data[0] = (HGM) ? 1 : 0;
    *data_len = 1;
    break;
case UBERTOOTH_SET_HGM:
    if (request_params[0])
      HGM_SET;
    else
      HGM_CLR;
    break;
#endif
```c
#define TX_ENABLE

    case UBERTOOTH_TX_TEST:
        requested_mode = MODE_TX_TEST;
        break;
    case UBERTOOTH_GET_PALEVEL:
        data[0] = cc2400_get(FRIEND) & 0x7;
        *data_len = 1;
        break;
    case UBERTOOTH_SET_PALEVEL:
        if (request_params[0] < 8) {
            cc2400_set(FRIEND, 8 | request_params[0]);
        } else {
            return 0;
        }
        break;
    case UBERTOOTH_RANGE_TEST:
        requested_mode = MODE_RANGE_TEST;
        break;
    case UBERTOOTH_REPEATER:
        requested_mode = MODE_REPEATER;
        break;
#else
    case UBERTOOTH_RANGE_CHECK:
        data[0] = rr_valid;
        data[1] = rr_request_pa;
        data[2] = rr_request_num;
        data[3] = rr_reply_pa;
        data[4] = rr_reply_num;
        *data_len = 5;
        break;
    case UBERTOOTH_STOP:
        requested_mode = MODE_IDLE;
        break;
    case UBERTOOTH_GET_MOD:
        data[0] = modulation;
        *data_len = 1;
        break;
    case UBERTOOTH_SET_MOD:
        modulation = request_params[0];
        break;
    case UBERTOOTH_GET_CHANNEL:
        data[0] = channel & 0xFF;
        data[1] = (channel >> 8) & 0xFF;
        *data_len = 2;
        break;
    case UBERTOOTH_SET_CHANNEL:
        requested_channel = request_params[0];
        /* bluetooth band sweep mode, start at channel 2402 */
        if (requested_channel > MAX_FREQ) {
            hop_mode = HOP_SWEEP;
            requested_channel = 2402;
        }
        /* fixed channel mode, can be outside bluetooth band */
        else {
            hop_mode = HOP_NONE;
            requested_channel = MAX(requested_channel, MIN_FREQ);
        }
```
requested_channel = MIN(requested_channel, MAX_FREQ);
}

if (mode != MODE_BT_FOLLOW_LE && mode != MODE_BT_MULTIFOLLOW_LE) {
  channel = requested_channel;
  requested_channel = 0;
  /* CS threshold is mode-dependent. Update it after
     * possible mode change. TODO - kludgy. */
  cs_threshold_calc_and_set(channel);
}
break;

case UBERTOOTH_SETISP:
  set_isp();
  /* data_len = 0; /* should never return */
  break;

  bootloader_ctrl = DFU_MODE;
  requested_mode = MODE_RESET;
  break;

  case UBERTOOTH_SPECAN:
    if (request_params[0] < 2049 || request_params[0] > 3072 ||
        request_params[1] < request_params[0])
      return 0;

      low_freq = request_params[0];
      high_freq = request_params[1];
      requested_mode = MODE_SPECAN;
  break;

  case UBERTOOTH_RX_GENERIC:
    requested_mode = MODE_RX_GENERIC;
  break;

  case UBERTOOTH_LED_SPECAN:
    if (request_params[0] > 256)
      return 0;

      rssi_threshold = 54 - request_params[0];
      requested_mode = MODE_LED_SPECAN;
  break;

case UBERTOOTH_GET_REV_NUM:
  data[0] = 0x00;
  data[1] = 0x00;
  length = (u8) strlen(GIT_REVISION);
  data[2] = length;
  memcpy(&data[3], GIT_REVISION, length);
  *data_len = 2 + 1 + length;
  break;

  case UBERTOOTH_GET_COMPILE_INFO:
    length = (u8) strlen(compile_info);
    data[0] = length;
    memcpy(&data[1], compile_info, length);
  break;
case UBERTOOTH_GET_BOARD_ID:
    data[0] = BOARD_ID;
    *data_len = 1;
    break;
case UBERTOOTH_SET_SQUELCH:
    cs_threshold_req = (int8_t)request_params[0];
    cs_threshold_calc_and_set(channel);
    break;
case UBERTOOTH_GET_SQUELCH:
    data[0] = cs_threshold_req;
    *data_len = 1;
    break;
case UBERTOOTH_SET_BDAC:
    target.address = 0;
    target.syncword = 0;
    for(int i=0; i < 8; i++) {
        target.address |= (uint64_t)data[i] << 8*i;
    }
    for(int i=0; i < 8; i++) {
        target.syncword |= (uint64_t)data[i+8] << 8*i;
    }
    precalc();
    break;
case UBERTOOTH_START_HOPPING:
    clkn_offset = 0;
    for(int i=0; i < 4; i++) {
        clkn_offset <<= 8;
        clkn_offset |= data[i];
    }
    hop_mode = HOP_BLUETOOTH;
    dma_discard = 1;
    DIO_SEL_SET;
    clk100ns_offset = (data[4] << 8) | (data[5] << 0);
    requested_mode = MODE_BT_FOLLOW;
    break;
case UBERTOOTH_AFH:
    hop_mode = HOP_AFH;
    requested_mode = MODE_AFH;
    for(int i=0; i < 10; i++) {
        afh_map[i] = 0;
    }
    used_channels = 0;
    afh_enabled = 1;
    break;
case UBERTOOTH_HOP:
    do_hop = 1;
    break;
case UBERTOOTH_SET_CLOCK:
    clkn = clock;
    cs_threshold_calc_and_set(channel);
    break;
case UBERTOOTH_SET_AFHMAP:
    for(int i=0; i < 10; i++) {
        afh_map[i] = data[i];
    }
afh_enabled = 1;
*data_len = 10;
break;
case UBERTOOTH_CLEAR_AFHMAP:
  for (int i=0; i < 10; i++) {
    afh_map[i] = 0;
  }
  afh_enabled = 0;
  *data_len = 10;
  break;
case UBERTOOTH_GET_CLOCK:
  clock = clkn;
  for (int i=0; i < 4; i++) {
    data[i] = (clock >> (8*i)) & 0xff;
  }
  *data_len = 4;
  break;
case UBERTOOTH_TRIM_CLOCK:
  clk100ns_offset = (data[0] << 8) | (data[1] << 0);
  break;
case UBERTOOTH_FIX_CLOCK_DRIFT:
  clk_drift_ppm += (int16_t)(data[0] << 8) | (data[1] << 0);
  /* Too slow */
  if (clk_drift_ppm < 0) {
    clk_drift_correction = 320 / (uint16_t)(−clk_drift_ppm);
    clkn_next_drift_fix = clkn_last_drift_fix +
    clk_drift_correction;
  }
  /* Too fast */
  else if (clk_drift_ppm > 0) {
    clk_drift_correction = 320 / clk_drift_ppm;
    clkn_next_drift_fix = clkn_last_drift_fix +
    clk_drift_correction;
  }
  /* Don’t trim */
  else {
    clk_drift_correction = 0;
    clkn_next_drift_fix = 0;
  }
  break;
case UBERTOOTH_BTLE_SNIFFING:
  *data_len = 0;
  do_hop = 0;
  hop_mode = HOP_BTLE;
  requested_mode = MODE_BT_FOLLOW_LE;
  queue_init();
  cs_threshold_calc_and_set(channel);
  break;
case UBERTOOTH_BTLE_MULTISNIFING:
  *data_len = 0;
  do_hop = 0;
  hop_mode = HOP_BTLE_MULTI;
  requested_mode = MODE_BT_MULTIFOLLOW_LE;
queue_init();
cs_threshold_calc_and_set(channel);
break;
case UBERTOOTH_GET_ACCESS_ADDRESS:
    for(int i=0; i < 4; i++) {
        data[i] = (le.access_address >> (8*i)) & 0xff;
    }
data_len = 4;
break;
case UBERTOOTH_SET_ACCESS_ADDRESS:
    le_set_access_address(data[0] | data[1] << 8 | \
    le.target_set = 1;
break;
case UBERTOOTH_DO_SOMETHING:
    /* do something! just don't commit anything here */
    break;
case UBERTOOTH_DO_SOMETHING_REPLY:
    /* after you do something, tell me what you did! */
    /* don't commit here please */
    data[0] = 0x13;
    data[1] = 0x37;
    data_len = 2;
break;
case UBERTOOTH_GET_CRC_VERIFY:
    data[0] = le.crc_verify ? 1 : 0;
    data_len = 1;
break;
case UBERTOOTH_SET_CRC_VERIFY:
    le.crc_verify = request_params[0] ? 1 : 0;
    break;
case UBERTOOTH_POLL:
    p = dequeue();
    if (p != NULL) {
        memcpy(data, (void *)p, sizeof(usb_pkt_rx));
        data_len = sizeof(usb_pkt_rx);
    } else {
        data[0] = 0;
    }
data_len = 1;
break;
case UBERTOOTH_BTLE_PROMISC:
    data_len = 0;
    hop_mode = HOP_NONE;
    requested_mode = MODE_BT_PROMISC_LE;
    queue_init();
    cs_threshold_calc_and_set(channel);
break;
case UBERTOOTH_READ_REGISTER:
    reg_val = cc2400_get(request_params[0]);
    data[0] = (reg_val >> 8) & 0xff;
    data[1] = reg_val & 0xff;
    data_len = 2;
break;
case UBERTOOTH_WRITE_REGISTER:
    cc2400_set(request_params[0] & 0xff, request_params[1]);
    break;

case UBERTOOTH_WRITE_REGISTERS:
    for (i=0; i<request_params[0]; i++) {
        reg_val = (data[(i*3)+1] << 8) | data[(i*3)+2];
        cc2400_set(data[i*3], reg_val);
    }
    break;

case UBERTOOTH_READ_ALL_REGISTERS:
    for (i=0; i<MAX_READ_REG; i++) {
        reg_val = cc2400_get(i);
        data[i*3] = i;
        data[(i*3)+1] = (reg_val >> 8) & 0xff;
        data[(i*3)+2] = reg_val & 0xff;
    }
    *data_len = MAX_READ_REG*3;
    break;

case UBERTOOTH_TX_GENERIC_PACKET:
    i = 7 + data[6];
    memcpy(&tx_pkt, data, i);
    /*tx_pkt.channel = data[4] << 8 | data[5]; */
    requested_mode = MODE_TX_GENERIC;
    *data_len = 0;
    break;

case UBERTOOTH_BTLE_SLAVE:
    memcpy(slave_mac_address, data, 6);
    requested_mode = MODE_BT_SLAVE_LE;
    break;

case UBERTOOTH_BTLE_SET_TARGET:
    /* Addresses appear in packets in reverse–octet order. */
    /* Store the target address in reverse order so that */
    /* we can do a simple memcmp later */
    le_target[0] = data[5];
    le_target[1] = data[4];
    le_target[2] = data[3];
    le_target[3] = data[2];
    le_target[4] = data[1];
    le_target[5] = data[0];
    le_target_set = 1;
    break;

#ifdef TX_ENABLE
    case UBERTOOTH_JAM_MODE:
        jam_mode = request_params[0];
        break;
    #endif

    case UBERTOOTH_EGO:
        #ifndef TX_ENABLE
            if (ego_mode == EGO_JAM)
                return 0;
        #endif
        requested_mode = MODE_EGO;
        ego_mode = request_params[0];
        break;
    #endif

    case UBERTOOTH_GET_API_VERSION:
for (int i = 0; i < 4; ++i) {
    data[i] = (UBERTOOTH_API_VERSION >> (8*i)) & 0xff;
    *data_len = 4;
    break;
}

default:
    return 0;
}

return 1;
*/

/* Function: TIMER0_IRQHandler */

* Handle interrupts triggered by the on-board clock. This function
* serves as the "driver" for following connections by using timeouts.
* Updates CLKN.
*/

void TIMER0_IRQHandler() {

    if (T0IR & TIR_MRO_Interrupt) {
        debug_clk100ns_count++;
        clkn += clkn_offset + 1;
        clkn_offset = 0;

        uint32_t le_clk = (clkn - le.conn_epoch) & 0x03;

        /* Trigger hop based on mode */
        /* NONE or SWEEP -> 25 Hz */
        if (hop_mode == HOP_NONE || hop_mode == HOP_SWEEP) {
            if (((clkn & 0x7f) == 0)
                do_hop = 1;
        }

        /* BLUETOOTH -> 1600 Hz */
        else if (hop_mode == HOP_BLUETOOTH) {
            if (((clkn & 0x1) == 0)
                do_hop = 1;
        }

        /* BLUETOOTH Low Energy -> 7.5ms - 4.0s in multiples of 1.25 ms */
        else if (hop_mode == HOP_BTLE) {
            /* Only hop if connected */
            if (le.link_state == LINK_CONNECTED && le_clk == 0) {
                le.interval_timer = le.interval_count;

                /* We reached the # of intervals until a hop is necessary */
                if (le.interval_timer == 0) {
                    do_hop = 1;
                    ++le.conn_count;
                    le.interval_timer = le.conn_interval; /* Reset intervals */
                } else {
                    TXLED_CLR; /* hack! */
                }
            }
        }
}
/* Multi-connection tracking. Here's where we detect that a
  * connection event ends */
else if (hop_mode == HOP_BTLE_MULTI ) {
  /* Decrease the advertisement timer, trigger if enabled */
  if (le_hdlr.adv_timer_en == 1) {
    --le_hdlr.adv_timer;
    if (le_hdlr.adv_timer == 0) {
      le_multi_update_adv_state();
      do_hop = 1;
      hop_reason |= FADVERTISEMENT;
    }
  }
  /* Decrease the connection event timer, trigger if enabled */
  if (le_hdlr.conn_event_timer_en == 1) {
    --le_hdlr.conn_event_timer;
    if (le_hdlr.conn_event_timer == 0) {
      do_hop = 1;
      hop_reason |= FCONN_EVENT;
    }
  }
  /* Decrease the decision timer, trigger if enabled */
  if (le_hdlr.decision_timer_en == 1) {
    --le_hdlr.decision_timer;
    if (le_hdlr.decision_timer == 0) {
      do_hop = 1;
      hop_reason |= FDECISION;
    }
  }
  /* Progress all connection interval timers */
  /* If an interval timer expires, we simply increase the
  * connection count and reset the interval timer. If the timer
  * changed our current link, we'll have to make sure we hop to
  * a new channel. */
  le_state_t *l = NULL;
  for (int i = 0; i < MAX_LINKS; i++) {
    l = &(le_hdlr.links[i]);
    /* If the link is active, decrease its interval timer */
    if (l->link_state != LINK_INACTIVE) {
      --(l->interval_timer);
      if (l->interval_timer == 0) {
        /* Automatically increase connection count */
        ++(l->conn_count);
        /* Automatically adjust to proper unmapped channel */
        u8 old_idx = l->channel_idx; /* to revert, if needed.
        l->channel_idx =
          (l->channel_idx + l->channel_increment) % 37;
        /* Apply any connection parameter update if necessary */
        if (l->update_pending && l->conn_count == 

1->update Instant) \{ 
  /* Extend previous connection event */
  --(l->conn_count);
  l->channel_idx = old_idx;

  /* Disable pending update (we're doing it!) */
  l->update_pending = 0;
  /* Treat the connection as if it is pending again */
  l->link_state = LINK_CONN_PENDING;
  l->conn_interval = l->interval_update;
  l->win_size = l->win_size_update;
  l->win_offset = l->win_offset_update;

  /* Set linger time and next expected */
  l->linger_time = l->win_size;
  l->next_expected_event = CLK100NS + l->win_offset;

  /* We need to recalculate priorities. */
  do_hop = 1;
  hop_reason |= F_CONN_UPDATE;
}

/* No connection update pending */
else \{
  /* Apply any pending channel map updates */
  if (l->map_update_pending & & l->conn_count == \ 
    l->update Instant) \{ 
    l->map_update_pending = 0;
    for (int i = 0; i < 37; i++)\{
      l->channel_map[i] = l->channel_map_update[i];
    }
    l->num_used_channels = l->
  num_used_channels_update;
  }

  /* Set where the next expected event start will be */
  l->next_expected_event = l->conn_interval;
  /* If our current link needs a hop, physically hop. */
  if(l == le_hdlr.cur_link)\{
    do_hop = 1;
    hop_reason |= F_INTERVAL;
  }
}

/* Sanitize next_expected_event */
if (l->next_expected_event >= CLK100NS_MAX) \{
  l->next_expected_event = 
  l->next_expected_event - CLK100NS_MAX;
}

/* Reset the interval timer, based on last confirmed event */
  * and next expected event */

87
le_multi_reset_interval_timer(1);

}  
}  

/* Every 1.28 seconds, clear any OLD connections. */
/ * 312.5us * 4096 */
if (! (clkn & 0x00000fff)) {
  le_multi_cleanup();
}

else if (hop_mode == HOP_AFH) {
  if ( (last_hop + hop_timeout) == clkn ) {
    do_hop = 1;
  }
}

/* Fix linear clock drift deviation */
if (clkn_next_drift_fix != 0 && clk100ns_offset == 0) {
  if (clkn >= clkn_next_drift_fix) {
    /* Too fast */
    if (clk_drift_ppm >= 0) {
      clk100ns_offset = 1;
    }
    /* Too slow */
    else {
      clk100ns_offset = 6249;
    }
    clkn_last_drift_fix = clkn;
    clkn_next_drift_fix = clkn_last_drift_fix + clk_drift_correction;
  }
}

/* Negative clock correction */
if (clk100ns_offset > 3124)
  clkn += 2;
T0MR0 = 3124 + clk100ns_offset;
clock100ns_offset = 0;
/* Ack interrupt */
T0IR = TIR_MR0_Interrupt;
}

}  

/* Function: EINT3_IRQHandler */
/* Also defined in ubertooth.c for TC13BADGE. */

#ifndef TC13BADGE
#endif

void EINT3_IRQHandler() {
  /* TODO – check specific source of shared interrupt */
  IO2IntClr = PIN_GIO6; /* clear interrupt */
  DIO_SSEL_CLR; /* enable SPI */

88
cs_trigger = 1;  /* signal trigger */
if (hop_mode == HOP_BLUETOOTH)
  dma_discard = 0;
#endif /* TC13BADGE */

/* Function: msleep */
/* Sleep (busy wait) for 'millis' milliseconds. The 'wait' routines in ubertooth.c are matched to the clock setup at boot time and can not be used while the board is running at 100MHz.
* msleep: */
static void msleep(uint32_t millis) {
  /* millis -> clkn ticks */
  uint32_t stop_at = clkn + millis * 3125 / 1000;
  do {} while (clkn < stop_at); /* TODO: handle wrapping */
}

/* Function: DMA_IRQHandler */
/* Handler for DMA-related interrupts. */
void DMA_IRQHandler() {
  if (mode == MODE_RX_SYMBOLS
      || mode == MODE_SPECAN
      || mode == MODE_BT_FOLLOW_LE
      || mode == MODE_BT_MULTIFOLLOW_LE
      || mode == MODE_BT_PROMISC_LE
      || mode == MODE_BT_SLAVE_LE
      || mode == MODE_RX_GENERIC)

    /* interrupt on channel 0 */
    if (DMACIntStat & (1 << 0)) {
      if (DMACIntTCStat & (1 << 0)) {
        DMACIntTCClear = (1 << 0);
      }
      DMACIntErrClr = (1 << 0);
      ++rx_tc;
    } else {
      if (DMACIntErrStat & (1 << 0)) {
        DMACIntErrClr = (1 << 0);
        ++rx_err;
      }
    }
}

volatile uint8_t* tmp = active_rxbuf;
active_rxbuf = idle_rxbuf;
idle_rxbuf = tmp;
++rx_tc;
if (DMACIntErrStat & (1 << 0)) {
  DMACIntErrClr = (1 << 0);
  ++rx_err;
}
static void cc2400_idle() {
    cc2400_strobe(SRFF);
    while ((cc2400_status() & FS_LOCK)); /* need to wait for unlock? */

    #ifdef UBERTOOTH
    PAEN_CLR;
    HGM_CLR;
    #endif

    RXLED_CLR;
    TXLED_CLR;
    USRLED_CLR;

    clkn_stop();
    dio_ssp_stop();
    cs_reset();
    rssi_reset();

    /* hopping stuff */
    hop_mode = HOP_NONE;
    do_hop = 0;
    channel = 2441;
    hop_direct_channel = 0;
    hop_timeout = 158;
    requested_channel = 0;
    saved_request = 0;

    /* bulk USB stuff */
    idle_buf_clkn_high = 0;
    idle_buf_clk100ns = 0;
    idle_buf_channel = 0;
    dma_discard = 0;
    status = 0;

    /* operation mode */
    mode = MODE_IDLE;
    requested_mode = MODE_IDLE;
    jam_mode = JAM_NONE;
    ego_mode = EGO_FOLLOW;
    modulation = MOD_BT_BASIC_RATE;

    /* specan stuff */
    low_freq = 2400;
    high_freq = 2483;
    rssi_threshold = -30;
    target.address = 0;
    target.syncword = 0;
}
/* Function: cc2400_rx
 * Start un-buffered rx.
 */
static void cc2400_rx() {
  u16 mdmctrl = 0;
  if ((modulation == MOD_BT_BASIC_RATE) || (modulation == MOD_BT_LOW_ENERGY)) {
    if (modulation == MOD_BT_BASIC_RATE) {
      mdmctrl = 0x0029; /* 160 kHz frequency deviation */
    } else if (modulation == MOD_BT_LOW_ENERGY) {
      mdmctrl = 0x0040; /* 250 kHz frequency deviation */
    }
    cc2400_set(MANAND, 0x7fff);
    cc2400_set(LMTST, 0x2b22);
    cc2400_set(MDMIST0, 0x134b); /* without PRNG */
    cc2400_set(MDMCTRL, 0x134b); /* un-buffered mode, GFSK */
  }
  cc2400_set(MANAND, 0x7fff);
  cc2400_set(LMTST, 0x2b22);
  cc2400_set(MDMIST0, 0x134b); /* without PRNG */
  cc2400_set(MDMCTRL, mdmctrl);
  /* Set up CS register */
  cs_threshold_calc_and_set(channel);
  clkn_start();
  while (!(cc2400_status() & XOSC16M_STABLE));
  cc2400_strobe(SFSO);
  while (!(cc2400_status() & FS_LOCK));
  cc2400_strobe(SRX);
  #ifdef UBERTOOTHONE
  PAEN_SET;
  HGM_SET;
  #endif
}

/* Function: cc2400_rx_sync
 * Start un-buffered rx of packets with AA of 'sync'.
 * sync: target access address
 */
static void cc2400_rx_sync(u32 sync) {
  u16 grmdm, mdmctrl;
  if (modulation == MOD_BT_BASIC_RATE) {
    mdmctrl = 0x0029; /* 160 kHz frequency deviation */
    grmdm = 0x0461; /* un-buffered, packet w/ sync word detection */
  }
  /* 0 00 00 1 000 11 0 00 0 1 */
}
else if (modulation == MOD_BT_LOW_ENERGY) {
    mdmctrl = 0x0040; /* 250 kHz frequency deviation */
    grmdm = 0x0561; /* un-buffered, packet w/ sync word detection */
    /* 0 00 00 1 010 11 0 00 0 1 */
    /* -> CRC off */
    /* 32 MSB bits of SYNCWORD */
    /* 0 preamble bytes of 01010101 */
    /* packet mode */
    /* 2 preamble bytes of 01010101 */
    /* packet mode */
    /* un-buffered mode */
    /* sync error bits: 0 */

} else {
    /* oops */
    return;
}

cc2400_set(MANAND, 0x7fff);
cc2400_set(LMTST, 0x2b22);
cc2400_set(MDMTST0, 0x124b);
/* 124b */
/* 00 0 1 0010011 */
/* AFC_DELTA = ?? */
/* AFC setting = 4 pairs (8 bit preamble) */
/* no AFC adjust on packet */
/* do not invert data */
/* TX IF freq 1 0Hz */
/* PRNG off */
/* * */
/* ref: CC2400 datasheet page 67 */
/* AFC settling explained page 41/42 */
cc2400_set(GRMDM, grmdm);
cc2400_set(SYNCL, sync & 0xffff);
cc2400_set(SYNCH, (sync >> 16) & 0xffff);
cc2400_set(FSDIV, channel - 1); /* 1 MHz IF */
cc2400_set(MDMCTRL, mdmctrl);
/* Set up CS register */
cs_threshold_calc_and_set(channel);
clkn_start();

while (!(cc2400_status() & XOSC16M_STABLE));
c2400_strobe(SFSON);
while (!(cc2400_status() & FS_LOCK));
c2400_strobe(SRX);
#endif
PAEN_SET;
HGM_SET;

#ifend

/*
 * Function: cc2400_tx_sync
 * Start buffered tx with AA of 'sync'.
 */
static void cc2400_tx_sync(uint32_t sync) {
#ifdef TX_ENABLE
    /* Bluetooth-like modulation */
    cc2400_set(MANAND, 0x7fff);
    cc2400_set(LMTST, 0x2b22); /* LNA and receive mixers test register */
    cc2400_set(MDMTST0, 0x134b); /* no PRNG */
    cc2400_set(GRMDM, 0x0c01);
    /* 0 0 01 1 000 00 00 0 1 */
    /* sync word: 8 MSB bits of SYNCWORD */
    /* packet mode */
    /* buffered mode */
    cc2400_set(SYNCL, sync & 0xffff);
    cc2400_set(SYNCH, (sync >> 16) & 0xffff);
    cc2400_set(FSDIV, channel);
    /* amplifier level (-7 dBm, picked from hat) */
    cc2400_set(FREND, 0b1011);
    if (modulation == MOD_BT_BASIC_RATE) {
        cc2400_set(MDMCTRL, 0x0029); /* 160 kHz frequency deviation */
    } else if (modulation == MOD_BT_LOW ENERGY) {
        cc2400_set(MDMCTRL, 0x0040); /* 250 kHz frequency deviation */
    } else {
        /* oops */
        return;
    }

    clkn_start();

    while (!(cc2400_status() & XOSCI16M_STABLE));
    cc2400_strobe(SFSON);
    while (!(cc2400_status() & FS_LOCK));

    #ifdef UBERTOOTH_ONE
    PAEN_SET;
    #endif

    while ((cc2400_get(FSMSTATE) & 0x1f) != STATE_STROBE_FS_ON);
    cc2400_strobe(STX);
#endif
Function: le_transmit

Transmit a BTLE packet with the specified access address.

All modulation parameters are set within this function. The data
should not be pre-whitened, but the CRC should be calculated and
included in the data length.

aa:
len:
data:

```c
void le_transmit(u32 aa, u8 len, u8 *data) {
    unsigned i, j;
    int bit;
    u8 txbuf[64];
    u8 tx_len;
    u8 byte;
    u16 gio_save;

    /* first four bytes: AA */
    for (i = 0; i < 4; ++i) {
        byte = aa & 0xff;
        aa >>= 8;
        txbuf[i] = 0;
        for (j = 0; j < 8; ++j) {
            txbuf[i] |= (byte & 1) << (7 - j);
            byte >>= 1;
        }
    }

    /* whiten the data and copy it into the txbuf */
    int idx = whitening_index[btle_channel_index(channel - 2402)];
    for (i = 0; i < len; ++i) {
        byte = data[i];
        txbuf[i+4] = 0;
        for (j = 0; j < 8; ++j) {
            bit = (byte & 1) ^ whitening[idx];
            idx = (idx + 1) % sizeof(whitening);
            byte >>= 1;
            txbuf[i+4] |= bit << (7 - j);
        }
    }

    len += 4; /* include the AA in len */

    /* Bluetooth-like modulation */
    cc2400_set(MANAND, 0x7fff);
    /* LNA and receive mixers test register */
    cc2400_set(LMTRT, 0x2b22);
    cc2400_set(MDMTST0, 0x134b); /* no PRNG */
    cc2400_set(GLMDM, 0x0c01);
    /* 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 */
    /* | | | | +----------> CRC off */
    /* | | | | +----------> sync word: 8 MSB bits of SYNCWORD */
```
cc2400_set(FSDIV, channel);
/* amplifier level (-7 dBm, picked from hat) */
cc2400_set(FREND, 0b1011);
cc2400_set(MDMCTRL, 0x0040); /* 250 kHz frequency deviation */
cc2400_set(INT, 0x0014); /* FIFO_THRESHOLD: 20 bytes */
/* sync byte depends on the first transmitted bit of the AA */
if (aa & 1)
  cc2400_set(SYNCH, 0xaaaa);
else
  cc2400_set(SYNCH, 0x5555);
/* set GIO to FIFO_FULL */
gio_save = cc2400_get(IOCFG);
cc2400_set(IOCFG, (GIO_FIFO_FULL << 9) | (gio_save & 0x1ff));
while (!((cc2400_status() & XOSC16M_STABLE)));
cc2400_strobe(FSON);
while (!((cc2400_status() & FS_LOCK)));
ifdef UBERTOOTHONE
  PAEN_SET;
#endif
while (((cc2400_get(FSMSTATE) & 0x1f) != STATE_STROBE_FS_ON);
cc2400_strobe(STX);
/* put the packet into the FIFO */
for (i = 0; i < len; i += 16) {
  while (GIO6); /* wait for the FIFO to drain (FIFO_FULL false) */
  tx_len = len - i;
  if (tx_len > 16)
    tx_len = 16;
  cc2400_fifo_write(tx_len, txbuf + i);
}
while (((cc2400_get(FSMSTATE) & 0x1f) != STATE_STROBE_FS_ON);
TXLED_CLR;
cc2400_strobe(SRFOFF);
while (((cc2400_status() & FS_LOCK)));
ifdef UBERTOOTHONE
  PAEN_CLR;
#endif
/* reset GIO */
cc2400_set(IOCFG, gio_save);
}
```c
void le_jam(void) {
    #ifdef TX_ENABLE
    cc2400_set(MANAND, 0x7fff);
    /* LNA and receive mixers test register */
    cc2400_set(LMITST, 0x2b22);
    cc2400_set(MDMTST, 0x234b); /* PRNG, 1 MHz offset */
    cc2400_set(GRMDM, 0x0c01);
    /* 0 00 01 1 000 00 0 00 0 1 */
    /* CRC off */
    /* sync word: 8 MSB bits of SYNCWORD */
    /* 0 preamble bytes of 01010101 */
    /* packet mode */
    /* buffered mode */
    /* amplifier level (-7 dBm, picked from hat) */
    cc2400_set(FREND, 0b1011);
    cc2400_set(MDACCTRL, 0x0040); /* 250 kHz frequency deviation */
    while (!((cc2400_status() & XOSC16MSTABLE)))
        cc2400_strobe(SFSON);
    while (!((cc2400_status() & FS_LOCK));
    TXLED_SET;
    #ifdef UBERTOOTHONE
    PAEN_SET;
    #endif
    while (((cc2400_get(FSMSTATE) & 0x1f) != STATE_STROBE_FS_ON);
        cc2400_strobe(STX);
    #endif
    /*
    * Function: hop
    * Physically change frequency channel based on hop mode and
    * hopping parameters.
    */
    void hop(void) {
        /* TODO - return whether hop happened, or should caller have to keep
        * track of this? */
        hop_reason = 0;
        do_hop = 0;
        last_hop = clkn;
        /* No hopping, if channel is set correctly, do nothing */
        if (hop_mode == HOP_NONE) {
            if (cc2400_get(FSDIV) == (channel - 1))
                return;
        }
        /* Slow sweep (100 hops/sec)
        * only hop to currently used channels if AFH is enabled
        */
        else if (hop_mode == HOP_SWEEP) {
            do {
                channel += 32;
                if (channel > 2480)
                    channel -= 79;
            
```
while ( used_channels != 0 && afh_enabled && 
  !( afh_map[ (channel - 2402) / 8 ] & 0x1 < ((channel - 2402) % 8) ) );

/* AFH detection 
 * only hop to currently unused channels */
else if (hop_mode == HOP_AFH) {
  do {
    channel += 32;
    if (channel > 2480)
      channel -= 79;
  } while ( used_channels != 79 && 
  (afh_map[(channel - 2402) / 8] & 0x1 < ((channel - 2402) % 8) ) );
}
else if (hop_mode == HOP_BLUETOOTH) {
  channel = next_hop(clkn);
}
else if (hop_mode == HOP_BTLE) {
  channel = btle_next_hop(&le);
}
else if (hop_mode == HOP_BTLE_MULTI) {
  /* For re-schedules: scheduler_en == 1 */
  /* For hops only: scheduler_en == 0 */
  if (le_hdlr.scheduled_en) {
    le_multi_scheduler(); /* Changes cur_link */
  } else le_hdlr.scheduled_en = 1; /* Re-enable if disabled */
  /* Reset the access address in case our link changed */
  cc2400_set(SYNC1, le_hdlr.cur_link->sync1);
  cc2400_set(SYNCH, le_hdlr.cur_link->synch);
  /* Move on the link’s current channel. */
  channel = btle_channel_index_to_phys(
    btle_afh_channel_index(le_hdlr.cur_link));
}
else if (hop_mode == HOP_DIRECT) {
  channel = hop_direct_channel;
}
/* IDLE mode, but leave amp on, so don’t call cc2400_idle(). */
cc2400_strobe(SRFOFF);
while ( (cc2400_status() & FS_LOCK) ); /* need to wait for unlock? */
/* Retune */
if (mode == MODE_TX_SYMBOLS)
  cc2400_set(FDIV, channel);
else
  cc2400_set(FDIV, channel - 1);
/* Update CS register if hopping. */
if (hop_mode > 0) {
  cs_threshold_calc_and_set(channel);
}  
    
    /* Wait for lock */  
    cc2400_strobe(SFSON);  
    while (!(cc2400_status() & FS_LOCK));  
    dma_discard = 1;  
    if (mode == MODE_TX_SYMBOLS)  
        cc2400_strobe(STX);  
    else  
        cc2400_strobe(SRX);  
    }  
  
  /* Function: bt_stream_rx  
   * Bluetooth packet monitoring.  
   */  
  void bt_stream_rx() {  
    int8_t rssi;  
    int8_t rssi_at_trigger;  
    RXLED_CLR;  
    queue_init();  
    dio_ssp_init();  
    dma_init();  
    dio_ssp_start();  
    cc2400_rx();  
    cs_trigger_enable();  
    while (requested_mode == MODE_RX_SYMBOLS || 
            requested_mode == MODE_BT_FOLLOW)  
      {  
        RXLED_CLR;  
        /* Wait for DMA transfer. TODO – need more work on  
         * RSSI. Should send RSSI indications to host even  
         * when not transferring data. That would also keep  
         * the USB stream going. This loop runs 50–80 times  
         * while waiting for DMA, but RSSI sampling does not  
         * cover all the symbols in a DMA transfer. Can not do  
         * RSSI sampling in CS interrupt, but could log time  
         * at multiple trigger points there. The MAX() below  
         * helps with statistics in the case that cs_trigger  
         * happened before the loop started. */  
        rssi_reset();  
        rssi_at_trigger = INT8_MIN;  
        while (!rx_tc) {  
          rssi = (int8_t)(cc2400_get(RSSI) >> 8);  
          if (cs_trigger && (rssi_at_trigger == INT8_MIN))  
            rssi = MAX(rssi,(cs_threshold_cur+54));  
            rssi_at_trigger = rssi;  
          }  
        }  
    }  
  
}
rssi_add(rssi);
handle_usb(clkn);

/* If timer says time to hop, do it. */
if (do_hop) {
    hop();
} else {
    TXLED_CLR;
}

/* TODO – set per-channel carrier sense threshold. 
   * Set by firmware or host. */
RXLED_SET;

if (rx_err) {
    status |= DMAERROR;
}

/* Missed a DMA transfer? */
if (rx_tc > 1)
    status |= DMAOVERFLOW;

if (dma_discard) {
    status |= DISCARD;
    dma_discard = 0;
}

rssi_iir_update(channel);

/* Set squelch hold if there was either a CS trigger, squelch
   * is disabled, or if the current rssi_max is above the same
   * threshold. Currently, this is redundant, but allows for
   * per-channel or other rssi triggers in the future. */
if (cs_trigger || cs_no_squelch) {
    status |= CS_TRIGGER;
    cs_trigger = 0;
}

if (rssi_max >= (cs_threshold_cur + 54)) {
    status |= RSSI_TRIGGER;
}

enqueue(BR_PACKET, (uint8_t *)idle_rxbuf);
rx_continue:
    handle_usb(clkn);
    rx_tc = 0;
    rx_err = 0;

/* This call is a nop so far. Since bt_rx_stream() starts the
 * stream, it makes sense that it would stop it. TODO – how
 * should setup/teardown be handled? Should every new mode be
 * starting from scratch? */
dio_ssp_stop();
cs_trigger_disable();

/*
 * Function: reverse8
 * Reverse the bit order of an 8-bit unsigned number.
 * data: 8-bit value to be reversed.
 */
static uint8_t reverse8(uint8_t data) {
    uint8_t reversed = 0;
    for (size_t i = 0; i < 8; i++)
        reversed |= ((data >> i) & 0x01) << (7 - i);
    return reversed;
}

/*
 * Function: reverse16
 * Reverse the bit order of a 16-bit unsigned number.
 * data: 16-bit value to be reversed.
 */
static uint16_t reverse16(uint16_t data) {
    uint16_t reversed = 0;
    for (size_t i = 0; i < 16; i++)
        reversed |= ((data >> i) & 0x01) << (15 - i);
    return reversed;
}

/*
 * Function: br_transmit
 * Transmit a BTBR packet with the specified access code.
 * All modulation parameters are set within this function.
 */
void br_transmit() {
    uint16_t gio_save;
    uint32_t clkn_saved = 0;
    uint16_t preamble = (target.syncword & 1) == 1 ? 0x5555 : 0xaaaa;
    uint8_t trailer = ((target.syncword >> 63) & 1) == 1 ? 0xaa : 0x55;
    uint8_t data[16] = {
        reverse8((target.syncword >> 0) & 0xFF),
        reverse8((target.syncword >> 8) & 0xFF),
        reverse8((target.syncword >> 16) & 0xFF),
        reverse8((target.syncword >> 24) & 0xFF),
        reverse8((target.syncword >> 32) & 0xFF),
        reverse8((target.syncword >> 40) & 0xFF),
    }
reverse8((target.syncword >> 48) & 0xFF),
reverse8((target.syncword >> 56) & 0xFF),
reverse8(trailer),
reverse8(0x77),
reverse8(0x66),
reverse8(0x55),
reverse8(0x44),
reverse8(0x33),
reverse8(0x22),
reverse8(0x11);
}

cC2400_tx_sync(reverse16(preamble));
cC2400_set(INT, 0x0014); /* FIFO_THRESHOLD: 20 bytes */

// set GIO to FIFO_FULL */
gio_save = cc2400_get(IOCFG);
cC2400_set(IOCFG, (GIO_FIFO_FULL << 9) | (gio_save & 0x1ff));

while (requested_mode == MODE_TX_SYMBOLS )
{
    while ((clk >> 1) == (clk_saved >> 1) || T0TC < 2250) {
        /* If timer says time to hop, do it. */
        if (do_hop) {
            hop();
        }
    }

    clkn_saved = clkn;
    TXLED_SET;
    cc2400_fifo_write(16, data);

    while ((cc2400_get(FSMSTATE) & 0x1f) != STATE_STROBE_FS_ON);
    TXLED_CLR;

    cc2400_strobe(SROFF);
    while ((cc2400_status() & FS_LOCK));

    while (!cc2400_status() & XOSC16MSTABLE);
    cc2400_strobe(SFSON);

    while (!cc2400_status() & FS_LOCK);

    while ((cc2400_get(FSMSTATE) & 0x1f) != STATE_STROBE_FS_ON);
    cc2400_strobe(STX);

    handle_usb(clkn);
}

#ifdef UBERTOOTH_ONE
    PAEN_CLR;
#endif

/* reset GIO */

101
cc2400_set(IOCFG, gio_save);
}

/*
 * Function: le_set_access_address
 * Set the access address and SYNCH and SYNCL values for an
 * LE link/connection.
 * - aa: requested access address.
 * - link: target link for address change.
 */
static void le_set_access_address(u32 aa, le_state_t *link) {
  u32 aa_rev;

  link->access_address = aa;
  aa_rev = rbit(aa);
  link->syncl = aa_rev & 0xffff;
  link->synch = aa_rev >> 16;
}

/*
 * Function: reset_le
 * Reset the LE state variable of the target link.
 * - link: target link to be reset.
 */
void reset_le(le_state_t *link) {
  le_set_access_address(0x8e89bed6, link); /* Adv AA */

  link->crc_init = 0x555555;
  link->crc_init_reversed = 0xAAAAAA;
  link->crc_verify = 0;
  link->last_packet = 0;

  link->link_state = LINK_INACTIVE;

  link->channel_idx = 0;
  link->channel_increment = 0;

  for (int i = 0; i < 37; i++) {link->channel_map[i] = 0x00;}

  link->num_used_channels = 0;

  link->conn_epoch = 0;
  link->interval_timer = 0;
  link->conn_interval = 0;
  link->conn_count = 0;

  link->win_size = 0;
  link->win_offset = 0;

  link->update_pending = 0;
  link->update_instant = 0;
  link->interval_update = 0;
  link->win_size_update = 0;
  link->win_offset_update;

  link->last_confirmed_event = 0;
  link->next_expected_event = 0;
  link->linger_time = 0;
  link->window_widening = 0;

1938 }
1939
1940 /* */
1941 * Function: reset_le_promisc
1942 * Reset the global state for LE promiscuous mode.
1943 * TODO: target a specific link, as in reset_le
1944 */
1945
1946 void reset_le_promisc(void) {
1947     memset(&le_promisc, 0, sizeof(le_promisc));
1948     le_promisc.smallest_hop_interval = 0xffffffff;
1949 }
1950
1951 /* */
1952 * Function: le_multi_scheduler
1953 * Observes all active connections and their priorities and makes a
1954 * decision about which connection to focus on next. The function runs
1955 * at the end of every connection event, as well as when the connection
1956 * handler internal timer expires.
1957 */
1958
1959 void le_multi_scheduler(void) {
1960     /* Disable all timers – we will re-enable them as necessary */
1961     le_hdlr.conn_event_timer_en = 0;
1962     le_hdlr.decision_timer_en = 0;
1963
1964     /* Reset any other necessary stuff */
1965     le_hdlr.cur_anchor_state = ANCHOR_SEARCH;
1966
1967     /* Find the very NEXT connection event across all connections
1968     * (i.e., lowest interval) */
1969     u32 min_interval = 2147483647;
1970     le_state_t *min_link = NULL, *l = NULL;
1971     for (int i = 0; i < MAX_LINKS; i++) {
1972         l = &(le_hdlr.links[i]);
1973         /* Check that the link exists */
1974         if (l->link_state != LINK_INACTIVE) {
1975             if (l->interval_timer < min_interval) {
1976                 min_interval = l->interval_timer;
1977                 min_link = l;
1978             }
1979         }
1980     }
1981
1982     /* TODO: The logic here may have to change depending on how we
1983     * select the next channel. */
1984     if (min_link != NULL) {
1985         /* Plenty of time until the next event! */
1986         if (min_interval > MIN_ADV_INTERVALS) {
1987             le_hdlr.cur_link = &(le_hdlr.adv_link);
1988         }
1989         /* Set the timer for when we need to come back! */
1990         le_hdlr.decision_timer =
1991         min_link->interval_timer - MIN_ADV_INTERVALS;
1992         le_hdlr.decision_timer_en = 1;
1993     }
1994
1995 103
/ Not enough time – just go to the correct connection event when
* the interval timer expires. */
else {
  le_hdlr.adv_timer_en = 0;
  le_hdlr.scheduler_en = 0;
  le_hdlr.cur_link = min_link;
  l = le_hdlr.cur_link;
  /* Set the connection event timer – once we move to the
  * connection, how long are we willing to wait for the first
  * frame? It depends... */

  /* add the chunk of time before the hop */
  le_hdlr.conn_event_timer = l->interval_timer;
  /* Adjust for 312.5us interrupts... the longer the better here,
  * so let's take ceiling */
  le_hdlr.conn_event_timer +=
      DIVIDE_CEIL((2*l->window_widening + l->linger_time),3125);
  le_hdlr.conn_event_timer += PACKET_OFFSET;

  /* Enable the timer */
  le_hdlr.conn_event_timer_en = 1;
} else {
  /* We don't have connections to listen to, listen to advs */
  /* No timers necessary... not a care in the world! */
  le_hdlr.cur_link = &(le_hdlr.adv_link);
}

/*
 * Function: bt_generic_le
 * __________
 * Generic le mode.
 * active_mode:
 **/
void bt_generic_le(u8 active_mode) {
  u8 *tmp = NULL;
  u8 hold;
  int i, j;
  int8_t rssi, rssi_at_trigger;
  modulation = MOD_BTLOWENERGY;
  mode = active_mode;
  reset_le(&le);

  /* enable USB interrupts */
  ISER0 = ISER0_JSE_USB;
  RXLED_CLR;

  queue_init();
  dio_ssp_init();
  dma_init();
  dio_ssp_start();
  cc2400_rx();
cs_trigger_enable();
hold = 0;

while (requested_mode == active_mode) {
    if (requested_channel != 0) {
        cc2400_strobe(SRFOFF);
        /* need to wait for unlock? */
        while (cc2400_status() & FS_LOCK);
        /* Retune */
        cc2400_set(FSDIV, channel - 1);
        /* Wait for lock */
        cc2400_strobe(SFSON);
        while (!cc2400_status() & FS_LOCK);
        /* RX mode */
        cc2400_strobe(SRX);
        requested_channel = 0;
    }
    if (do_hop) {
        hop();
    } else {
        TXLED_CLR;
    }
}
RXLED_CLR;
/* Wait for DMA. Meanwhile keep track of RSSI. */
rssi_reset();
rssi_at_trigger = INT8_MIN;
while ((rx_tc == 0) && (rx_err == 0))
{
    rssi = (int8_t)(cc2400_get(RSSI) >> 8);
    if (cs_trigger && (rssi_at_trigger == INT8_MIN)) {
        rssi = MAX(rssi, (cs_threshold_cur+04));
        rssi_at_trigger = rssi;
    }
    rssi_add(rssi);
}
if (rx_err) {
    status |= DMA_ERROR;
}
/* No DMA transfer? */
if (!rx_tc)
    goto rx_continue;
/* Missed a DMA transfer? */
if (rx_tc > 1)
    status |= DMA_OVERFLOW;
rssi_iir_update(channel);
/ Set squelch hold if there was either a CS trigger, squelch
* is disabled, or if the current rssi_max is above the same
* threshold. Currently, this is redundant, but allows for
* per-channel or other rssi triggers in the future. */

int status = CS_TRIGGER;
int hold = CS_HOLD_TIME;
int cs_trigger = 0;

if (rssi_max >= (cs_threshold_cur + 54)) {  
  status |= RSSI_TRIGGER;
  hold = CS_HOLD_TIME;
}

/* Hold expired? Ignore data. */
if (hold == 0) {
  goto rx_continue;
}
hold --;

/* copy the prev unpacked symbols to the front of the buffer */
memcpy(unpacked, unpacked + DMA_SIZE*8, DMA_SIZE*8);

/* unpack the new packet to the end of the buffer */
for (i = 0; i < DMA_SIZE; ++i) {
  /* output one byte for each received symbol (0x00 or 0x01) */
  for (j = 0; j < 8; ++j) {
    unpacked[DMA_SIZE*8 + i * 8 + j] = (idle_rxbuf[i] & 0x80) >>
    7;
    idle_rxbuf[i] <<= 1;
  }
  int ret = data_cb(unpacked);
  if (!ret) break;
rx_continue:
  rx_tc = 0;
x_err = 0;
}

/* disable USB interrupts */
ICER0 = ICER0|ICE_USB;

/* reset the radio completely */
c2400_idle();
dio_ssp_stop();
cs_trigger_disable();

/* Function: bt_le_sync
* Time-synced version of bt_generic_le. This function waits for DMA,
* processes incoming packets, and enqueues the packets to send to the
* host. */
* - active_mode: mode of operation, used to detect if user wants to change it.

```c
void bt_le_sync(u8 active_mode) {
    int i;
    int8_t rssi;
    static int restart_jamming = 0;
    modulation = MOD_BT_LOW_ENERGY;
    mode = active_mode;
    le.link_state = LINK_LISTENING;

    /* enable USB interrupts */
    ISER0 = ISER0|ISE_USB;

    RXLED_CLR;

    queue_init();
    dio_ssp_init();
    dma_init_le();
    dio_ssp_start();

    /* bit-reversed access address */
    cc2400_rx_sync(rbit(le.access_address));

    /* Exit if the mode switches for any reason */
    while (requested_mode != active_mode) {
        /* First, tune to the requested channel, if any */
        if (requested_channel != 0) {
            cc2400_strobe(SR_OFF);
            /* need to wait for unlock? */
            while (cc2400_status() & FS_LOCK);

            /* Retune */
            cc2400_set(FS_DIV, channel - 1);

            /* Wait for lock */
            cc2400_strobe(SF_ON);
            while (!cc2400_status() & FS_LOCK);

            /* RX mode */
            cc2400_strobe(SR_X);

            saved_request = requested_channel;
            requested_channel = 0;
        }

        RXLED_CLR;

        /* Wait for DMA. Meanwhile keep track of RSSI. */
        /* Jose's Notes: DMA will tell us that it has a packet for us! */
        rssi_reset();
        while ((rx_tc == 0) && (rx_err == 0) &&
               (do_hop == 0) && requested_mode == active_mode)
        ;
```
rssi = (int8_t)(cc2400_get(RSSI) >> 8);
rssi_min = rssi_max = rssi;

if (requested_mode != active_mode) {
    goto cleanup;
}

if (rx_err) {
    status |= DMA_ERROR;
}

if (do_hop)
    goto rx_flush;

/* No DMA transfer? */
if (!rx_tc)
    continue;

/* We assume that rx_tc==1 is what brought us here, let's build a
 * frame with the info waiting for us in the buffer */
uint32_t packet[48/4+1] = { 0, };
u8 *p = (u8*)packet;
packet[0] = le.access_address;

const uint32_t *whit =
    whitening_word[pbtle_channel_index(channel-2402)];
for (i = 0; i < 4; i+= 4) {
    uint32_t v = rxbuf1[i+0] << 24
        | rxbuf1[i+1] << 16
        | rxbuf1[i+2] << 8
        | rxbuf1[i+3] << 0;
    packet[i/4+1] = rbit(v) ^ whit[i/4];
}

unsigned len = (p[5] & 0x3f) + 2;
if (len > 39)
    goto rx_flush; /* length is bad, throw it all away. */

/* transfer the minimum number of bytes from the CC2400, this
 * allows us enough time to resume RX for subsequent frames
 * on the same channel */
unsigned total_transfers = ((len + 3) + 4 - 1) / 4;
if (total_transfers < 11) {
    while (DMACC0DestAddr < 
            (uint32_t)rxbuf1 + 4 * total_transfers && rx_err == 0)
        ;
} else { /* max transfers? just wait till DMA's done */
    while (DMACC0Config & DMACCxConfig_E && rx_err == 0)
        ;
}

DIO_SSP_DMACR &= ~SSPDMACR_RXDMAE;

/* strobe SFSON to allow the resync to occur while we process
 * the frame */
cc2400_strobe(SFSON);

/* unwhiten the rest of the packet */
for (i = 4; i < 44; i += 4) {
    uint32_t v = rxbuf1[i+0] << 24
    | rxbuf1[i+1] << 16
    | rxbuf1[i+2] << 8
    | rxbuf1[i+3] << 0;
    packet[i/4+1] = rbit(v) ^ whit[i/4];
}

if (le.crc_verify) {
    u32 calc_crc = btle_crcgen_lut(le.crc_init_reversed, p + 4, len);
    u32 wire_crc = (p[4+len+2] << 16)
    | (p[4+len+1] << 8)
    | (p[4+len+0] << 0);
    if (calc_crc != wire_crc) /* skip packets with a bad CRC */
        goto rx_flush;
}

RXLED_SET;

/* Change the state of connection based on packet contents */
packet_cb((uint8_t *)packet);

/* Queue up the frame to send back to our user */
enqueue(LE_PACKET, (uint8_t *)packet);

le.last_packet = CLK100NS;

rx_flush:
/* this might happen twice, but it's safe to do so */
cc2400_strobe(SFSON);

/* flush any excess bytes from the SSP’s buffer */
DIO_SSP_DMACR &= ~SPDMACR_RXDMAE;
while (SSPSR & SSPSR_RNE) {
    u8 tmp = (u8)DIO_SSP_DR;
}

/* timeout - FIXME this is an ugly hack */
u32 now = CLK100NS;
if (now < le.last_packet)
    now += 3276800000; /* handle rollover */
if (/* timeout */
    ((le.link_state == LINK_CONNECTED || 
    le.link_state == LINK_CONN_PENDING)
    & (now - le.last_packet > 50000000))
    /* jam finished */
    || (le.jam_count == 1)) {
    reset_le(&le);
    le.jam_count = 0;
    TXLED_CLR;

    if (jam_mode == JAM_ONCE) {
        jam_mode = JAM_NONE;
        requested_mode = MODE_IDLE;
        goto cleanup;
    }
if (active_mode == MODE_BT_PROMISC_LE)
    goto cleanup;

le.link_state = LINK_LISTENING;

cc2400_strobe(SRFOFF);
while ((cc2400_status() & FS_LOCK));

/* Retune */
channel = (saved_request != 0) ? saved_request : 2402;
restart_jamming = 1;
}

cc2400_set(SYNC, le.syncl);
cc2400_set(SYNCH, le.synch);

if (do_hop)
    hop();
/* you can jam but you keep turning off the light */
if (le_jam_count > 0) {
    le_jam();
    --le_jam_count;
} else {
    /* RX mode */
dma_init_le();
dio_ssp_start();

    if (restart_jamming) {
        cc2400_rx_sync(rbit(le.access_address));
        restart_jamming = 0;
    } else {
        /* wait till we're in FSLOCK before strobing RX */
        while (!cc2400_status() & FS_LOCK);
        cc2400_strobe(SRX);
    }
}

rx_tc = 0;
rx_err = 0;
} /* end of while (requested_mode==active_mode) */
cleanup:
/* disable USB interrupts */
ICER0 = ICER0_USB;
/* reset the radio completely */
cc2400_idle();
dio_ssp_stop();
cs_trigger_disable();
}

/* Function: bt_multile_sync

110
void bt_multiele_sync() {
    int i;
    int8_t rssi;
    static int restart_jamming = 0;
    modulation = MOD_BT_LOWENERGY;
    mode = MODE_BT_MULTIFOLLOW_LE;

    /* enable USB interrupts */
    ISER0 = ISER0_ISE_USB;
    RXLED_CLR;
    queue_init();
    dio_ssp_init();
    dma_init_le();
    dio_ssp_start();

    /* Tune in to the access address of the current channel */
    cc2400_rx_sync(rbit(le_hdlr.cur_link->access_address));

    /* Exit if the mode switches for any reason */
    while (requested_mode == MODE_BT_MULTIFOLLOW_LE) {
        RXLED_CLR;
        /* Wait for DMA. Meanwhile keep track of RSSI. */
        /* DMA will tell us that it has a packet for us! */
        rssi_reset();
        while ((rx_tc == 0) && (rx_err == 0) && (do_hop == 0) &&
               requested_mode == MODE_BT_MULTIFOLLOW_LE)
            ;
        rssi = (int8_t)(cc2400_get(RSSI) >> 8);
        rssi_min = rssi_max = rssi;

        /* User wants to change the mode.. let's exit. */
        if (requested_mode != MODE_BT_MULTIFOLLOW_LE) {
            goto cleanup;
        }

        /* Oops! There was an error with DMA */
        if (rx_err) {
            status |= DMAERROR;
        }

        /* We’re done with this channel. Let’s move on with our lives. */
        if (do_hop)
            goto rx_flush;

        /* No DMA transfer? */
        if (!rx_tc)
```c
continue; /* Do nothing. */

/* We assume that rx_tc==1 is what brought us here, let's build a
* frame with the info waiting for us in the buffer */
uint32_t packet[48/4+1] = { 0, };

u8 *p = (u8 *) packet;
packet[0] = le_hdlr.cur_link->access_address;

const uint32_t *whit =
  whitening_word[btle_channel_index(channel-2402)];
for (i = 0; i < 4; i+= 4) {
  uint32_t v = rxBuf1[i+0] << 24
  | rxBuf1[i+1] << 16
  | rxBuf1[i+2] << 8
  | rxBuf1[i+3] << 0;

  packet[i/4+1] = rbit(v) ^ whit[i/4];
}

unsigned len = (p[5] & 0x3f) + 2;
if (len < 39) goto rx_flush; /* length is bad, throw it all away. */
/* transfer the minimum number of bytes from the CC2400 */
/* this allows us enough time to resume RX for subsequent frames
* on the same channel */
unsigned total_transfers = ((len + 3) + 4 - 1) / 4;
if (total_transfers < 11) {
  while (DMACC0DestAddr < 
    (uint32_t) rxBuf1 + 4 * total_transfers && rx_err == 0)
  ;
} else { /* max transfers? just wait till DMA's done */
  while (DMACC0Config & DMACCxConfigE && rx_err == 0)
  ;
}
DIO_SSP_DMACR &= ~SSPDMACRXDMAE;

/* strobe SFSON to allow the resync to occur while we process
* the frame */
cc2400_strobe(SFSON);

/* unwhiten the rest of the packet */
for (i = 4; i < 44; i += 4) {
  uint32_t v = rxBuf1[i+0] << 24
  | rxBuf1[i+1] << 16
  | rxBuf1[i+2] << 8
  | rxBuf1[i+3] << 0;

  packet[i/4+1] = rbit(v) ^ whit[i/4];
}

RXLED_SET;

/* Change the state of connection based on packet contents */
connection_multi_follow_cb((u8 *) packet);
/* Oh no! The packet was discarded for some reason */
/*if (packet == NULL) goto rx_flush; */
```
/* Good packet – queue it up to send to host */
enqueue(LE_PACKET, (uint8_t *)packet);

/* If we weren’t on an Adv channel, let’s reset and enable
the event timer. */
if (le_hdlr.cur_link != le_hdlr.adv_link) {
    le_hdlr.conn_event_timer = TIFS + PACKET_OFFSET;
    le_hdlr.conn_event_timer_en = 1;
}

rx_flush:
/* this might happen twice, but it’s safe to do so */
cc2400_strobe(SFSON);
    /* flush any excess bytes from the SSP’s buffer */
DIO_SSP_DMACR &= ~SPDMACR_RXDMAE;
while (SPISR & SSPSR_RNE) {
    u8 tmp = (u8)DIO_SSP_DR;
}
    if (do_hop) {
        hop();
    }
    dma_init_le();
    dio_ssp_start();
    /* wait till we’re in FSLOCK before strobing RX */
while (!cc2400_status() & FSLOCK);
cc2400_strobe(SRX);
rx_tc = 0;
rx_err = 0;
} /* end of while (requested_mode==active_mode) */
cleanup:
/* disable USB interrupts */
ICER0 = ICER0_ICE_USB;
/* reset the radio completely */
cc2400_idle();
dio_ssp_stop();
cs_trigger_disable();
}

/* Function: cb_follow_le
* Legacy LE packet-following function.
* unpacked:
*
int cb_follow_le(char* unpacked) {
    int i, j, k;
    int idx = whitening_index[btle_channel_index(channel−2402)];
    u32 access_address = 0;
    for (i = 0; i < 31; ++i) {
        access_address >>= 1;

113
access_address |= (unpacked[i] << 31);
}

for (i = 31; i < DMA_SIZE * 8 + 32; i++) {
    access_address >>= 1;
    access_address |= (unpacked[i] << 31);
    if (access_address == le.access_address) {
        for (j = 0; j < 46; ++j) {
            u8 byte = 0;
            for (k = 0; k < 8; k++) {
                int offset = k + (j * 8) + i - 31;
                if (offset >= DMA_SIZE*8+2) break;
                int bit = unpacked[offset];
                if (j >= 4) { /* unwhiten data bytes */
                    bit ^= whitening[idx];
                    idx = (idx + 1) % sizeof(whitening);
                }
                byte |= bit << k;
            }
            idle_rxbuf[j] = byte;
        }
    }
}

/* verify CRC */
if (le.crc_verify) {
    int len = (idle_rxbuf[5] & 0x3f) + 2;
    u32 calc_crc = btle_crcgen_lut(le.crc_init_reversed, 4, len);
    u32 wire_crc = (idle_rxbuf[4+len] << 16)
                   | (idle_rxbuf[4+len+1] << 8)
                   | idle_rxbuf[4+len+0];
    if (calc_crc != wire_crc) /
        /* skip packets with a bad CRC */
        break;
}

/* send to PC */
enqueue(LE_PACKET, (uint8_t*)idle_rxbuf);
RXLED_SET;
packet_cb((uint8_t*)idle_rxbuf);
break;
}
return 1;

/* Function: connection_follow_cb */
/* Observes an LE packet and changes link state accordingly */
/* (e.g., move from LINK_CONN_PENDING to LINK_CONNECTED). */
/* packet: pointer to the packet to be analyzed. */
/* void connection_follow_cb(u8 *packet) { */
int i;
uint8_t adv_addr = &packet[ADV_ADDRESS_IDX];

u8 *adv_addr = &packet[ADV_ADDRESS_IDX];
u8 header = packet[HEADER_IDX];

u8 *data_len = &packet[DATA_LEN_IDX];

u8 *data = &packet[DATA_START_IDX];

u8 *crc = &packet[DATA_START_IDX + *data_len];

if (le.link_state == LINK_CONN_PENDING) {
    /* We received a packet in the connection pending state, */
    /* so now the device *should* be connected */
    le.link_state = LINK_CONNECTED;
    le.conn_epoch = clkn;
    le.interval_timer = le.conn_interval - 1;
    le.conn_count = 0;
    le.update_pending = 0;

    /* hue hue hue */
    if (jam_mode != JAM_NONE)
        le_jam_count = JAM_COUNT_DEFAULT;
}
else if (le.link_state == LINK_CONNECTED) {
    u8 llid = header & 0x03;

    /* Apply any connection parameter update if necessary */
    if (le.update_pending && le.conn_count == le.update_instance) {
        /* This is the first packet received in the connection interval
            * for which the new parameters apply */
        le.conn_epoch = clkn;
        le.conn_interval = le.interval_update;
        le.interval_timer = le.interval_update - 1;
        le.win_size = le.win_size_update;
        le.win_offset = le.win_offset_update;
        le.update_pending = 0;
    }

    if (llid == 0x03 && data[0] == 0x00) {
        /* This is a CONNECTION_UPDATE_REQ. */
        /* The host is changing the connection parameters. */
        le.win_size_update = packet[7];
        le.win_offset_update = packet[8] + ((u16)packet[9] << 8);
        le.update_instance = packet[16] + ((u16)packet[17] << 8);
        if (le.update_instance - le.conn_count < 32767)
            le.update_pending = 1;
    }
}
else if (le.link_state == LINK_LISTENING) {
    u8 pkt_type = packet[4] & 0x0F;
    if (pkt_type == 0x05) {
        /* debug stuff!!! */
        TXLED_SET;
        debug_clk100ns_count = 0;

        /* This is a CONNECT_REQ */
        /* Master is requesting a connection to the slave */
        uint16_t conn_interval;

        /* ignore packets with incorrect length */
    }
if (*data_len != 34) return;

/* conn interval must be [7.5 ms, 4.0 s] in units of 1.25 ms */
conn_interval = (packet[29] << 8) | packet[28];
if (conn_interval < 6 || conn_interval > 3200) return;

/* if we have a target, see if InitA or AdvA matches */
if (le.target_set &&
/* Target address doesn’t match Initiator. */
memcmp(le.target, &packet[6], 6) &&
/* Target address doesn’t match Advertiser. */
memcmp(le.target, &packet[12], 6)) {
  return;
}

/* We are now PENDING, since the first hop has not occurred */
le.link_state = LINK_CONN_PENDING;
/* we will drop many packets if we attempt to filter by CRC */
le.crc_verify = 0;

for (i = 0; i < 4; ++i) aa |= packet[18+i] << (i*8);
/* We now have an access address */
le.set_access_address(aa, &le);
le.crc_init = (packet[CRC_INIT+2] << 16)
  | (packet[CRC_INIT+1] << 8)
  | packet[CRC_INIT+0];
le.crc_init_reversed = rbit(le.crc_init);
le.win_size = packet[WIN_SIZE];
le.win_offset = packet[WIN_OFFSET];
le.conn_interval = (packet[CONN_INTERVAL+1] << 8)
  | packet[CONN_INTERVAL+0];
le.channel_increment = packet[CHANNEL_INC] & 0x1f;
le.channel_idx = le.channel_increment;

/* Hop to the initial channel immediately... */
/* We’ll wait for the first packet there */
do_hop = 1;

/* Function: connection_multi_follow_cb */
* Observes an LE packet and changes link state accordingly
* (e.g., move from LINK_CONN_PENDING to LINK_CONNECTED). Used when
* following multiple connections, the "active" connection is assumed
* to be le_hdlr.cur_link.
* packet: pointer to the packet to be analyzed.
*/
void connection_multi_follow_cb(u8 *packet) {
  int i;
  u32 aa = 0;
u8 *adv_addr = &packet[ADV_ADDRESS_IDX];
u8 header = packet[HEADER_IDX];
int data_len = &packet[DATA_LEN_IDX];
uint16_t data = &packet[DATA_START_IDX];
uint16_t crc = &packet[DATA_START_IDX + *data_len];

/* Our working variable for the current link */
le_state_t *l = NULL;

/* Are we dealing with a connection, or just an adv packet? */
if (le_hdlr.cur_link == &le_hdlr.adv_link){
  /* If we SEE a connection request for a target of interest,
   * create a new connection */
  u8 pkt_type = packet[4] & 0x0F;
  if (pkt_type == 0x05) {
    /* This is a CONNECT_REQ */
    /* Master is requesting a connection to the slave */
    uint16_t conn_interval;

    /* ignore packets with incorrect length */
    if (*data_len != 34)
      return;

    /* conn_interval must be [7.5 ms, 4.0 s] in units of 1.25 ms */
    conn_interval = (packet[29] << 8) | packet[28];
    if (conn_interval < 6 || conn_interval > 3200)
      return;

    /* TODO: Does the connection req match one of the targets? */

    /* Find some room for our new connection */
    l = NULL;
    for (int i = 0; i < MAX_LINKS; i++){
      if (le_hdlr.links[i].link_state == LINK_INACTIVE) {
        l = &le_hdlr.links[i];
        break;
      }
    }

    /* We don’t have room for the connection! */
    if (l == NULL) return;

    /* We are now PENDING, since the first hop has not occurred */
    l->link_state = LINK_CONN_PENDING;

    /* we will drop many packets if we attempt to filter by CRC */
    l->crc_verify = 0;
    for (i = 0; i < 4; ++i)
      aa |= packet[18+i] << (i*8);
    le_set_access_address(aa, 1); /* We now have an AA */

    l->crc_init = (packet[CRC_INIT+2] << 16)
      | (packet[CRC_INIT+1] << 8)
      | packet[CRC_INIT+0];
    l->crc_init_reversed = rbit(l->crc_init);
    l->win_size = packet[WIN_SIZE] * 12500;
    l->win_offset =
      (packet[WIN_OFFSET + 1] << 8 | packet[WIN_OFFSET]) * 12500;
l->conn_interval = conn_interval * 12500;
l->channel_increment = packet[CHANNEL\_INC] & 0x1f;
l->channel_idx = 0;

l->num\_used\_channels = 0;
for (int i = 0; i < 37; i++)
  l->channel_map[i] =
  (packet[CHANNEL\_MAP + i/8] >> (i % 8)) & 0x01;
  if (l->channel_map[i] != 0x00) l->num\_used\_channels++;

u16 sca = 0;
switch ((packet[CHANNEL\_INC] >> 5) & 0x07) {
  case 0: sca = 500; break;
  case 1: sca = 250; break;
  case 2: sca = 150; break;
  case 3: sca = 100; break;
  case 4: sca = 75; break;
  case 5: sca = 50; break;
  case 6: sca = 30; break;
  case 7: sca = 20; break;
}

l->sca = sca;

/* At the expiration of the timer, we move to count = 0. */
l->conn\_count = -1;

/* Set the connection's first interval timer */
*(when is the window opening?) */
l->linger\_time = l->win\_size;
l->last\_confirmed\_event = CLK100NS;
l->next\_expected\_event =
l->last\_confirmed\_event + 12500 + l->win\_offset;
if (l->next\_expected\_event >= CLK100NS\_MAX) {
l->next\_expected\_event = l->next\_expected\_event -
CLK100NS\_MAX;
}
le\_multi\_reset\_interval\_timer(1);

TXLED\_SET;

/* Run the scheduler immediately */
do\_hop = 1;
hop\_reason |= F\_INTERVAL;

} /* So it's not a connection request... */
/* Is the advertisement packet an adv for one of our targets? */
else {
  /* if we have a target, see if the address matches it. */
  if (le\_target\_set && !memcmp(le\_target, &packet[6], 6)) {
    /* Target matches our set target */
    le\_hdlr\_cur\_adv\_state = ADV\_CANDIDATE;
    le\_hdlr\_adv\_timer = 1;
    le\_hdlr\_adv\_timer\_en = 1;
  }
}
else {
/* Grab the connection, make it easier to work with. */
ll = le_hdlr.cur_link;
/* TODO: First, let's verify that the packet is valid */
* (if requested!) */
if (l->crc_verify)
/* */
}
/* We received a conn req and are waiting to begin hopping */
if (l->link_state == LINK_CONN_PENDING)
/* We received a packet in the connection pending state, */
/* so now the device *should* be connected */
l->link_state = LINK_CONNECTED;
l->conn_epoch = CLK100NS;
l->update_pending = 0;
/* Set the interval timer */
if (le_hdlr.conn_event_timer < l->window_widening)
{l->linger_time = 0;
} else if (le_hdlr.conn_event_timer - l->window_widening < 
l->win_size)
{l->linger_time =
l->window_widening;
l->win_size = (le_hdlr.conn_event_timer - l->window_widening);
}
else {
{l->linger_time = l->win_size;
}
if (l->next_expected_event >= CLK100NS_MAX) {
1->next_expected_event = l->next_expected_event -
CLK100NS_MAX;
}
le_multi_reset_interval_timer(1);
/* We are already connected and hopping */
else if (l->link_state == LINK_CONNECTED)
u8 llid = header & 0x03;
/* */
if (llid == 0x03 && data[0] == 0x00)
{l->win_size_update = packet[7] * 12500;
1->win_offset_update =
1->interval_update =
1->update_instant = packet[16] + ((u16)packet[17] << 8);
if (l->update_instant - l->conn_count < 32767)
1->update_pending = 1;
}
/* */
else if (llid == 0x03 && data[0] == 0x01) {

l->num_used_channels_update = 0;
for (int i = 0; i < 37; i++) {
    l->channel_map_update[i] =
    (packet[7 + i/8] >> (i % 8) & 0x01);
    if (l->channel_map_update[i] != 0x00)
        l->num_used_channels_update++;
}

if (l->update.instant - l->conn_count < 32767)
    l->map_update_pending = 1;

/* LL_TERMINATE_IND */
else if (llid == 0x03 && data[0] == 0x02) {
    /* Connection ended .. release the link. */
    reset_le(l);
}

/* For all DATA CHANNEL frames, see if any is an anchor */
/* is the packet empty? */
int isEmpty = (*data_len == 0) ? 1 : 0;
/* is the MD bit set? */
int isMDClear = (header & 0x10) ? 0 : 1;

if (le_hdlr.cur_anchor_state == ANCHOR_SEARCH) {
    /* Looking for the first EMPTY packet with MD=0 */
    if (isEmpty && isMDClear) {
        /* Found it! */
        le_hdlr.cur_anchor_state = ANCHOR_CANDIDATE;
        le_hdlr.candidate_anchor = CLK100NS;
    }
} 
else if (le_hdlr.cur_anchor_state == ANCHOR_CANDIDATE) {
    /* Looking for the second EMPTY packet with MD=0 */
    if (isEmpty && isMDClear) {
        /* Found the second – let’s set the anchor point, 
         * and calibrate our timer */
        l->linger_time = 0;
        l->last_confirmed_event = le_hdlr.candidate_anchor;
        l->next_expected_event =
        l->last_confirmed_event + l->conn_interval;
        l->next_expected_event =
        l->next_expected_event - PACKET_OFFSET;
        le_multi.reset_interval_timer(1);
        le_hdlr.cur_anchor_state = ANCHOR_SEARCH;
    }
}

if (l != NULL) {
    l->last.packet = CLK100NS;
}

/* Function: bt_follow_le */
* Setup function to follow a single LE connection.

```c
void bt_follow_le() {
    /* The current requested_mode is MODE_BT_FOLLOW_LE */
    reset_le(&le);
    packet_cb = connection_follow_cb;
    bt_le_sync(MODE_BT_FOLLOW_LE);
    mode = MODE_IDLE;
}
```

* Function: bt_multi_follow_le
* Setup function to follow multiple LE connections simultaneously.

```c
void bt_multi_follow_le() {
    /* Reset all of our link states */
    for (int i = 0; i < MAX_LINKS; i++) {
        reset_le(&(le_hdlr.links[i]));
    }
    /* The first place we should start is on an advertisement channel */
    le_hdlr.cur_link = &(le_hdlr.adv_link);
    /* Start pulling packets out of the air */
    bt_multi_le_sync();
    /* All done - go IDLE */
    mode = MODE_IDLE;
}
```

*************** Functions for PROMISCUOUS MODE ***************

* Function: le_promisc_state
* Issue state change message.

```c
void le_promisc_state(u8 type, void *data, unsigned len) {
    u8 buf[50] = { 0, };
    if (len > 49)
        len = 49;
    buf[0] = type;
    memcpy(&buf[1], data, len);
    enqueue(LE_PROMISC, (uint8_t *)buf);
}
```

* Function: promisc_recover_hop_increment
* Determine the LE hop increment parameter while in promiscuous mode.
* packet: pointer to the packet to be analyzed.
void promisc_recover_hop_increment(u8 *packet) {
    static u32 first_ts = 0;
    if (channel == 2404) {
        first_ts = CLK100NS;
        hop_direct_channel = 2406;
        do_hop = 1;
    } else if (channel == 2406) {
        u32 second_ts = CLK100NS;
        if (second_ts < first_ts)
            second_ts += 3276800000; /* handle rollover */
        /* # of channels hopped between prev and current timestamp. */
        u32 channels_hopped = DIVIDE_ROUND(second_ts - first_ts,
                                              le.conn_interval * LE_BASECLK);
        if (channels_hopped < 37) {
            /* Get the hop increment based on the # of channels hopped. */
            le.channel_increment = hop_interval_lut[channels_hopped];
            le.interval_timer = le.conn_interval / 2;
            le.conn_count = 0;
            le.conn_epoch = 0;
            do_hop = 0;
            /* Move on to regular connection following. */
            le.channel_idx = (1 + le.channel_increment) % 37;
            le.link_state = LINK_CONNECTED;
            le.crc_verify = 0;
            hop_mode = HOP_BTLE;
            packet_cb = connection_follow_cb;
            le_promisc_state(3, &le.channel_increment, 1);

            if (jam_mode != JAM_NONE)
                le_jam_count = JAM_COUNT_DEFAULT;

            return;
        }
        hop_direct_channel = 2404;
        do_hop = 1;
    } else {
        hop_direct_channel = 2404;
        do_hop = 1;
    }
}

/* Function: promisc_recover_hop_interval *
* Determine the LE hop interval parameter while in promiscuous mode.
* packet: pointer to the packet to be analyzed. */
void promisc_recover_hop_interval(u8 *packet) {
    static u32 prev_clk = 0;
    u32 cur_clk = CLK100NS;
    if (cur_clk < prev_clk)
        cur_clk += 3267800000; /* handle rollover */
    u32 clk_diff = cur_clk - prev_clk;
    u16 obsv_hop_interval; /* observed hop interval */
/ * probably consecutive data packets on the same channel */
  if (clk_diff < 2 * LE_BASECLK)
    return;
  if (clk_diff < le_promisc.smallest_hop_interval)
    le_promisc.smallest_hop_interval = clk_diff;
  obsv_hop_interval =
    DIVIDE_ROUND(le_promisc.smallest_hop_interval, 37 * LE_BASECLK);
  if (le.conn_interval == obsv_hop_interval) {
    /* 5 consecutive hop intervals: consider it legit and move on */
    ++le_promisc.consec_intervals;
    if (le_promisc.consec_intervals == 5) {
      packet_cb = promisc_recover_hop_increment;
      hop_direct_channel = 2404;
      hop_mode = HOP_DIRECT;
      do_hop = 1;
      le_promisc_state(2, &le.conn_interval, 2);
    }
  } else {
    le.conn_interval = obsv_hop_interval;
    le_promisc.consec_intervals = 0;
  }
  prev_clk = cur_clk;
} /* Function: promisc_follow_cb */
void promisc_follow_cb(u8 *packet) {
  int i;
  /* get the CRCInit */
  if (!le_crc_verify && packet[4] == 0x01 && packet[5] == 0x00) {
    le_crc_init = btle_reverse_crc(crc, packet + 4, 2);
    le_crc_init_reversed = 0;
    for (i = 0; i < 24; ++i)
      le_crc_init_reversed |= ((le_crc_init >> i) & 1) << (23 - i);
    le_crc_verify = 1;
    packet_cb = promisc_recover_hop_interval;
    le_promisc_state(1, &le_crc_init, 3);
  }
} /* Function: see_aa */
/* In promiscuous mode, when an access address is seen on the wire,
* add it to the running list of AA candidates. */
void see_aa(u32 aa) {
    int i, max = -1, killme = -1;
    for (i = 0; i < AA_LIST_SIZE; ++i)
        if (le_promisc.active_aa[i].aa == aa) {
            ++le_promisc.active_aa[i].count;
            return;
        }
    /* evict someone */
    for (i = 0; i < AA_LIST_SIZE; ++i)
        if (le_promisc.active_aa[i].count < max || max < 0) {
            killme = i;
            max = le_promisc.active_aa[i].count;
        }
    le_promisc.active_aa[killme].aa = aa;
    le_promisc.active_aa[killme].count = 1;
}

/* Function: cb_le_promisc */
/* LE promiscuous mode. */
/* unpacked: */
int cb_le_promisc(char *unpacked) {
    int i, j, k;
    int idx;

    /* empty data PDU: 01 00 */
    char desired[4][16] = {
        { 1, 0, 0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, },
        { 1, 0, 0, 1, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, },
        { 1, 0, 1, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, },
        { 1, 0, 1, 1, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, },
    };

    for (i = 0; i < 4; ++i) {
        idx = whitening_index[btle_channel_index(channel - 2402)];

        /* whiten the desired data */
        for (j = 0; j < (int)sizeof(desired[i]); ++j) {
            desired[i][j] ^= whitening[idx];
            idx = (idx + 1) % sizeof(whitening);
        }
    }

    /* then look for that bitsream in our receive buffer */
    for (i = 32; i < (DMA_SIZE*8*2 - 32 - 16); i++) {
        int ok[4] = { 1, 1, 1, 1, 1};
        int matching = -1;

        ...
for (j = 0; j < 4; ++j) {
    for (k = 0; k <= (int)sizeof(desired[j]); ++k) {
        if (unpacked[i+k] != desired[j][k]) {
            ok[j] = 0;
            break;
        }
    }
}

/* see if any match */
for (j = 0; j < 4; ++j) {
    if (ok[j]) {
        matching = j;
        break;
    }
}

/* skip if no match */
if (matching < 0)
    continue;

/* found a match! unwhiten it and send it home */
idx = whitening_index[btle_channel_index(channel−2402)];
for (j = 0; j < 4+3+3; ++j) {
    u8 byte = 0;
    for (k = 0; k < 8; k++) {
        int offset = k + (j * 8) + i - 32;
        if (offset >= DMA_SIZE*8*2) break;
        int bit = unpacked[offset];
        if (j >= 4) { /* unwhiten data bytes */
            bit ^= whitening[idx];
            idx = (idx + 1) % sizeof(whitening);
        }
        byte |= bit << k;
    }
    idle_rxbuf[j] = byte;
}

u32 aa = (idle_rxbuf[3] << 24) |
         (idle_rxbuf[2] << 16) |
         (idle_rxbuf[1] << 8) |
         (idle_rxbuf[0]);

see_aa(aa);
enqueue(LE_PACKET, (uint8_t*)idle_rxbuf);

/* once we see an AA 5 times, start following it */
for (i = 0; i < AA_LIST_SIZE; ++i) {
    if ((le_promisc.active_aa[i].count > 3) {
        le_set_access_address(le_promisc.active_aa[i].aa, &le);
        data_cb = cb_follow_le;
        packet_cb = promisc_follow_cb;
        le_crc_verify = 0;
        le_promisc_state(0, &le.access_address, 4);
/* quit using the old stuff and switch to sync mode */
return 0;
}
return 1;

/* Function: bt_promisc_le
* Grab LE packets out of the air while in promiscuous mode.
*/
void bt_promisc_le() {
while (requested_mode == MODE_BT_PROMISC_LE) {
reset_le_promisc();
/* jump to a random data channel and turn up the squelch */
if (((channel & 1) == 1)
channel = 2440;
/* if the PC hasn’t given us AA, determine by listening */
if (!le.target_set) {
/* cs_threshold_req = -80; */
cs_threshold_calc_and_set(channel);
data_cb = cb_le_promisc;
broadcast(MODE_BT_PROMISC_LE);
}
/* could have got mode change in middle of above */
if (requested_mode != MODE_BT_PROMISC_LE)
break;
le_promisc_state(0, &le.access_address, 4);
packet_cb = promisc_follow_cb;
le_crc_verify = 0;
broadcast(MODE_BT_PROMISC_LE);
}

/****************** MISC ******************/
/* Function: bt_slave_le
*/
void bt_slave_le(void) {
u32 calc_crc;
int i;

u8 adv_ind[] = {
/* LL header */
0x00, 0x09,
/* advertising address */
0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
/* advertising data */
0x02, 0x01, 0x05,
/* CRC (calc) */
0xff, 0xff, 0xff,
};

u8 adv_ind_len = sizeof(adv_ind) - 3;

/* copy the user-specified mac address */
for (i = 0; i < 6; ++i)
    adv_ind[i+2] = slave_mac_address[5-i];

calc_crc = btle_calc_crc(le_crc_init_reversed, adv_ind, adv_ind_len);
adv_ind[adv_ind_len+0] = (calc_crc >> 0) & 0xff;
adv_ind[adv_ind_len+1] = (calc_crc >> 8) & 0xff;
adv_ind[adv_ind_len+2] = (calc_crc >> 16) & 0xff;

clkn_start();

/* spam advertising packets */
while (requested_mode == MODE_BT_SLAVE_LE)
{
    ICER0 = ICER0_ICE_USB;
    ICER0 = ICER0_ICE_DMA;
    le_transmit(0x8e89bed6, adv_ind_len+3, adv_ind);
    ISER0 = ISER0_ISE_USB;
    ISER0 = ISER0_ISE_DMA;
    msleep(100);
}

/* Function: rx_generic_sync */
void rx_generic_sync(void)
{
int i, j;
    u8 len = 32;
    u8 buf[len+4];
    u16 reg_val;

    /* Put syncword at start of buffer
     * DGS: fix this later, we don’t know number of syncword bytes, etc
     */
    reg_val = cc2400_get(SYNCH);
    buf[0] = (reg_val >> 8) & 0xFF;
    buf[1] = reg_val & 0xFF;
    reg_val = cc2400_get(SYNCL);
    buf[2] = (reg_val >> 8) & 0xFF;
    buf[3] = reg_val & 0xFF;

    queue_init();
    clkn_start();

    while (!(cc2400_status() & XOSCI16M_STABLE));
    cc2400_strobe(SFSON);
    while (!(cc2400_status() & FS_LOCK));
    RXLED_SET;
```c
#define UBERTOOTH_ONE
    PAEN_SET;
    HGM_SET;
#endif
while (1) {
    while ((cc2400_get(FSMSTATE) & 0x1f) != STATE_STROBE_FS_ON);
    cc2400_strobe(SRX);
    USRLED_CLR;
    while (!((cc2400_status() & SYNC_RECEIVED)));
    USRLED_SET;
    cc2400_fifo_read(len, buf+4);
    enqueue(BR_PACKET, buf);
    handle_usb(clkn);
}

/*
 * Function: rx_generic

*/

void rx_generic(void) {
    /* Check for packet mode */
    if (cc2400_get(GRDMM) & 0x0400) {
        rx_generic_sync();
    } else {
        modulation == MOD_NONE;
        bt_stream_rx();
    }
}

/*
 * Function: tx_generic

*/

void tx_generic(void) {
    u32 i;
    u16 synch, syncl;
    u8 tx_len;
    u8 prev_mode = mode;
    mode = MODE_TX_GENERIC;
    /* Save existing syncword */
    synch = cc2400_get(SYNCH);
    syncl = cc2400_get(SYNCL);
    cc2400_set(SYNCH, tx_pkt.synch);
    cc2400_set(SYNCL, tx_pkt.syncl);
    cc2400_set(MDMCTRL, 0x0057);
    cc2400_set(MDMTS0, 0x134b);
    cc2400_set(GRM, 0x0f61);
    cc2400_set(FSDIV, tx_pkt.channel);
    cc2400_set(FRIEND, tx_pkt.pa_level);
    while (!(cc2400_status() & XO汐16M_STABLE));
```

cc2400_strobe(SFSON);
while (!(cc2400_status() & FS_LOCK));
TXLED_SET;
#endif
while ((cc2400_get(FSMSTATE) & 0x1f) != STATE_STROBE_FS_ON);
cc2400_fifo_write(tx_pkt.length, tx_pkt.data);
cc2400_strobe(STX);
while ((cc2400_get(FSMSTATE) & 0x1f) != STATE_STROBE_FS_ON);
TXLED_CLR;
cc2400_strobe(SROFF);
while ((cc2400_status() & FS_LOCK));
#endif
/* Restore state */
cc2400_set(SYNCH, synch);
cc2400_set(SYNCL, syncl);
requested_mode = prev_mode;
}

/******************************* Spectrum analyzers *******************************
*/
*/
* Function: specan
* Run spectrum analyzer.
*/
void specan(void) {
  u16 f;
  u8 i = 0;
  u8 buf[DMA_SIZE];
  RXLED_SET;
  queue_init();
  clkn_start();
#endif
/* UBERTOOTH_ONE */
PAEN_SET;
/*HGM_SET; */
#endif
cc2400_set(LMTST, 0x2b22);
cc2400_set(MDMIST0, 0x134b); /* without PRNG */
cc2400_set(GRAHM, 0x0101); /* un-buffered mode, GFSK */
cc2400_set(MDAMCTRL, 0x0029); /* 160 kHz frequency deviation */
/*FIXME maybe set RSSI.RSSI_FILT */
while (!(cc2400_status() & XOSC16M_STABLE));
while ((cc2400_status() & FS_LOCK));
while (requested_mode == MODESPECAN) {
  for (f = low_freq; f < high_freq + 1; f++) {
    cc2400_set(FSDIV, f - 1);
    cc2400_strobe(SFSON);
while (!(cc2400_status() & FS_LOCK));
cc2400_strobe(SRX);

/* give the CC2400 time to acquire RSSI reading */
volatile u32 j = 500; while (--j); /*FIXME crude delay */
buf[3 * i] = (f >> 8) & 0xFF;
buf[(3 * i) + 1] = f & 0xFF;
buf[(3 * i) + 2] = cc2400_get(RSSI) >> 8;
i++;
if (i == 16) {
    enqueue(SPECAN, buf);
    i = 0;
    handle_usb(clkn);
}
cc2400_strobe(SRFF);
while ((cc2400_status() & FS_LOCK));

RXLED_CLR;

/* Function: led_specan */
* Run LED-based spectrum analyzer. */
void led_specan(void) {
    int8_t lvl;
    u8 i = 0;
    u16 channels[3] = {2412, 2437, 2462};
    /*void (*set[3]) = {TXLED_SET, RXLED_SET, USRLED_SET}; */
    /*void (*clr[3]) = {TXLED_CLR, RXLED_CLR, USRLED_CLR}; */
    #ifdef UBERTOOTH_ONE
    PAEN_SET;
    #endif
    cc2400_set(LMTST, 0x2b22);
    cc2400_set(MDMST0, 0x134b); /* without PRNG */
    cc2400_set(GFMMA, 0x0101); /* un-buffered mode, GFSK */
    cc2400_set(MDACTRL, 0x0029); /* 160 kHz frequency deviation */
    cc2400_set(RSSI, 0x00F1); /* RSSI Sample over 2 symbols */
    while (!((cc2400_status() & XOSC16MSTABLE)))
    while (((cc2400_status() & FS_LOCK)));

    while (requested_mode == MODE_LED_SPECAN) {
cc2400_set(FSDIV, channels[i] - 1);
cc2400_strobe(SFSON);
    while (!((cc2400_status() & FS_LOCK)));
cc2400_strobe(SRX);

    /* give the CC2400 time to acquire RSSI reading */
volatile u32 j = 500; while (--j); /*FIXME crude delay */
    lvl = (int8_t)((cc2400_get(RSSI) >> 8) & 0xff);
    if (lvl > rssi_threshold) {

switch (i) {
    case 0:
        TXLED_SET;
        break;
    case 1:
        RXLED_SET;
        break;
    case 2:
        USRLED_SET;
        break;
}
} else {
    switch (i) {
    case 0:
        TXLED_CLR;
        break;
    case 1:
        RXLED_CLR;
        break;
    case 2:
        USRLED_CLR;
        break;
    }
    i = (i+1) % 3;
    handle_usb(clkn);
    cc2400_strobe(SRFOFF);
    while (((cc2400_status() & FS_LOCK)));
}

/* Function : le_multi_reset_interval_timer */
/* Reset the connection’s interval timer based on whether an anchor */
/* and/or epoch are present. */
/* link : target link */
/* */
void le_multi_reset_interval_timer(le_state_t *l) {
    u32 now = CLK100NS;
    u32 next_minus_window = 0;
    /* IT = (nextEventStart - wideningWindow) - now; */
/ * Handle clock rollover cases, first for the window, then for the */
/* current time "now" */
if ( l->next_expected_event < l->window_widening ) {
    next_minus_window =
    CLK100NS_MAX - l->window_widening + l->next_expected_event;
} else
    next_minus_window =
    l->next_expected_event - l->window_widening;

if ( next_minus_window < now ) {
    l->interval_timer = (CLK100NS_MAX - now) + next_minus_window;
} else
    l->interval_timer = (next_minus_window - now);

/* Adjust for 312.5us interrupts, truncating down... we want to be */
/* on the short side */
l->interval_timer = l->interval_timer/3125;

/* Account for the fact that we observe the packet at its END. */
l->interval_timer = l->interval_timer - PACKET_OFFSET;

/* Function: le_multi_cleanup */
/* Gets rid of stale connections */
void le_multi_cleanup(void){
  /* Drop any stale connections */
  le_state_t *l = NULL;
  u32 now = CLK100NS, elapsed = 0;
  for (int i = 0; i < MAX_LINKS; i++){
    l = & (le_hdr.links[i]);
    if (l->link_state != LINK_INACTIVE) {
      /* Our window is just too wide... get rid of the connection! */
      if (l->window_widening >= l->conn_interval/4){
        reset_le(1);
        do_hop = 1; /* in case this connection was next/active */
      }
    /* Calculate time elapsed since the last packet */
    if ( now < l->last_packet )
      elapsed = (CLK100NS_MAX - l->last_packet) + now;
    else
      elapsed = now - l->last_packet;
    /* It’s been too long since the last packet, */
    /* get rid of the connection */
    if (elapsed >= MAX_INACTIVE_LINK_TIME*10000000){
      reset_le(1);
      do_hop = 1; /* in case this connection was next/active */
    }
  }
}
/* Function: le_multi_update_adv_state */
/* Progress the advertisement state when the timer expires. */
```c
void le_multi_update_adv_state(void) {
    le_state_t *a = &le_hdlr.adv_link;
    if (le_hdlr.cur_adv_state == ADV_CANDIDATE) {
        if (a->channel_idx == 37) {
            a->channel_idx = 38;
            le_hdlr.adv_timer = 32; /* 10 ms */
        } else if (a->channel_idx == 38) {
            a->channel_idx = 39;
            le_hdlr.adv_timer = 32; /* 10 ms */
        } else {
            a->channel_idx = 37;
            le_hdlr.cur_adv_state = ADV_SEARCH;
            le_hdlr.adv_timer_en = 0;
        }
    } else {
        a->channel_idx = 37;
        le_hdlr.adv_timer_en = 0;
    }
}
```
1.4 firmware/bluetooth_rxtx/ubertooth_clock.h

/*
 * Copyright 2016 Air Force Institute of Technology, U.S. Air Force
 * Copyright 2015 Hannes Ellinger
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; see the file COPYING. If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110-1234, USA.
 */

#ifndef _UBERTOOTH_CLOCK_H
#define _UBERTOOTH_CLOCK_H value

#include "inttypes.h"

/*CLK100NS is a free-running clock with a period of 100 ns. It resets
 * every 2^15 * 10^5 cycles (about 5.5 minutes) — computed from clkn
 * and timer0 (T0TC)
 * clkn is the native (local) clock as defined in the Bluetooth
 * specification. It advances 3200 times per second. Two clkn periods
 * make a Bluetooth time slot.
 */
#define UBERTOOTH_CLOCK_PPM 20
#define CLK100NS_MAX 3276800000

volatile uint32_t clkn;
volatile uint32_t last_hop;
volatile uint32_t clkn_offset;
volatile uint16_t clk100ns_offset;

/* linear clock drift */
volatile int16_t clk_drift_ppm;
volatile int16_t clk_drift_correction;
volatile int32_t clkn_last_drift_fix;
volatile int32_t clkn_next_drift_fix;

#define CLK100NS (3125*(clkn & 0xffffffff) + T0TC)
#define LE_BASECLK (12500) /* 1.25 ms in units of 100ns */

void clkn_stop();
void clkn_start();
void clkn_init();

#endif
1.5 firmware/common/ubertooth.h

/*
 Copyright 2016 Air Force Institute of Technology, U.S. Air Force
 Copyright 2010, 2011 Michael Ossmann

 This program is free software; you can redistribute it and/or modify
 it under the terms of the GNU General Public License as published by
 the Free Software Foundation; either version 2, or (at your option)
 any later version.

 This program is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 GNU General Public License for more details.

 You should have received a copy of the GNU General Public License
 along with this program; see the file COPYING. If not, write to
 the Free Software Foundation, Inc., 51 Franklin Street,
 Boston, MA 02110-1301, USA.
 */

#ifndef _UBERTOOTH_H
#define _UBERTOOTH_H

#include "lpc17.h"
#include "types.h"
#include "cc2400.h"
#include "ubertooth_interface.h"

typedef void (* IAP_ENTRY) (u32 [], u32 []);
extern const IAP_ENTRY iap_entry;

/* operating modes */
enum operating_modes {
    MODE_IDLE = 0,
    MODE_RX_SYMBOLS = 1,
    MODE_TX_SYMBOLS = 2,
    MODE_TX_TEST = 3,
    MODE_SPECAN = 4,
    MODE_RANGE_TEST = 5,
    MODE_REPEATER = 6,
    MODE_LED_SPECAN = 7,
    MODE_BT_FOLLOW = 8,
    MODE_BT_FOLLOW_LE = 9,
    MODE_BT_MULTIFOLLOW_LE = 10, /* */
    MODE_BT_PROMISC_LE = 11,
    MODE_RESET = 12,
    MODE_BT_SLAVE_LE = 13,
    MODE_EGO = 14,
    MODE_AFH = 15,
    MODE_RX_GENERIC = 16,
    MODE_TX_GENERIC = 17,
};

/* hardware identification number */
#define BOARD_ID_UBERTOOTH_ZERO 0
```c
#define BOARD_ID_UBERTOOTH_ONE 1
#define BOARD_ID_TC13BADGE 2

#ifndef UBERTOOTH_ZERO
#define BOARD_ID BOARD_ID_UBERTOOTH_ZERO
#endif

#ifndef UBERTOOTH_ONE
#define BOARD_ID BOARD_ID_UBERTOOTH_ONE
#endif

#ifndef TC13BADGE
#define BOARD_ID BOARD_ID_TC13BADGE
#endif

/* GPIO pins */
#ifndef UBERTOOTH_ZERO
#define PIN_USRLED (1 << 11) /* P0.11 */
#define PIN_RXLED (1 << 28) /* P4.28 */
#define PIN_TXLED (1 << 29) /* P4.29 */
#define PIN_CC1V8 (1 << 29) /* P1.29 */
#define PIN_CC3V3 (1 << 0) /* P1.0 */
#define PIN_VBUS (1 << 30) /* P1.30 */
#define PIN_RX (1 << 1) /* P1.1 */
#define PIN_TX (1 << 4) /* P1.4 */
#define PIN_CS (1 << 8) /* P1.8 */
#define PIN_SCLK (1 << 9) /* P1.9 */
#define PIN_MOSI (1 << 10) /* P1.10 */
#define PIN_MISO (1 << 14) /* P1.14 */
#define PIN_GIO6 (1 << 15) /* P1.15 */
#define PIN_BTGR (1 << 31) /* P1.31 */
#define PIN_SSEL0 (1 << 9) /* P2.9 */
#endif

#ifndef UBERTOOTH_ONE
#define PIN_USRLED (1 << 1) /* P1.1 */
#define PIN_RXLED (1 << 4) /* P1.4 */
#define PIN_TXLED (1 << 8) /* P1.8 */
#define PIN_CC1V8 (1 << 9) /* P1.9 */
#define PIN_CC3V3 (1 << 14) /* P1.14 */
#define PIN_VBUS (1 << 30) /* P1.30 */
#define PIN_RX (1 << 15) /* P1.15 */
#define PIN_TX (1 << 29) /* P4.29 */
#define PIN_CS (1 << 5) /* P2.5 */
#define PIN_SCLK (1 << 4) /* P2.4 */
#define PIN_MOSI (1 << 0) /* P2.0 */
#define PIN_MISO (1 << 1) /* P2.1 */
#define PIN_GIO6 (1 << 2) /* P2.2 */
#define PIN_BTGR (1 << 10) /* P1.10 */
#define PIN_SSEL1 (1 << 28) /* P4.28 */
#define PIN_PAEN (1 << 7) /* P2.7 */
#define PIN_HGM (1 << 8) /* P2.8 */
#endif

#ifndef TC13BADGE
#define PIN_CC1V8 (1 << 0) /* P1.0 */
#define PIN_CC3V3 (1 << 14) /* P1.14 */
#define PIN_VBUS (1 << 30) /* P1.30 */
#define PIN_DIGITAL2 (1 << 0) /* P2.0 */
#define PIN_DIGITAL3 (1 << 1) /* P2.1 */
```
#ifdef PIN_DIGITAL4 (1 << 2) /* P2.2 */
#define PIN_DIGITAL5 (1 << 3) /* P2.3 */
#define PIN_DIGITAL6 (1 << 4) /* P2.4 */
#define PIN_DIGITAL7 (1 << 5) /* P2.5 */
#define PIN_DIGITAL8 (1 << 6) /* P2.6 */
#define PIN_DIGITAL9 (1 << 7) /* P2.7 */
#define PIN_SW1 (1 << 8) /* P2.8 */
#define PIN_CS (1 << 1) /* P1.1 */
#define PIN_SCL (1 << 4) /* P1.4 */
#define PIN_MOSI (1 << 8) /* P1.8 */
#define PIN_MISO (1 << 9) /* P1.9 */
#define PIN_GIO0 (1 << 10) /* P1.10 */
#define PIN_SCL1 (1 << 28) /* P4.28 */
#define PIN_R8C_CTL (1 << 22) /* P1.22 connects to R8C's P4.5 */
#define PIN_R8C_ACK (1 << 19) /* P1.19 connects to R8C's P0.0 */
/* RX, TX, and BT/GR are fixed to ground on TC13BADGE */
#endif

/* indicator LED control */
#else if UBERTOOTH_ZERO
#define USRLED (FIO0PIN & PIN_USRLED)
#define USRLED_SET (FIO0SET = PIN_USRLED)
#define USRLED_CLR (FIO0CLR = PIN_USRLED)
#define RXLED (FIO4PIN & PIN_RXLED)
#define RXLED_SET (FIO4SET = PIN_RXLED)
#define RXLED_CLR (FIO4CLR = PIN_RXLED)
#define TXLED (FIO4PIN & PIN_TXLED)
#define TXLED_SET (FIO4SET = PIN_TXLED)
#define TXLED_CLR (FIO4CLR = PIN_TXLED)
#endif
#ifdef UBERTOOTH_ONE
#define USRLED (FIO1PIN & PIN_USRLED)
#define USRLED_SET (FIO1SET = PIN_USRLED)
#define USRLED_CLR (FIO1CLR = PIN_USRLED)
#define RXLED (FIO1PIN & PIN_RXLED)
#define RXLED_SET (FIO1SET = PIN_RXLED)
#define RXLED_CLR (FIO1CLR = PIN_RXLED)
#define TXLED (FIO1PIN & PIN_TXLED)
#define TXLED_SET (FIO1SET = PIN_TXLED)
#define TXLED_CLR (FIO1CLR = PIN_TXLED)
#endif
#ifdef TC13BADGE
/*FIXME The LEDs need to be controlled by talking to the R8C. */
#define USRLED 0
#define USRLED_SET
#define USRLED_CLR
#define RXLED 0
#define RXLED_SET
#define RXLED_CLR
#define TXLED 0
#define TXLED_SET
#define TXLED_CLR
#endif
#ifdef SW1
/* SW1 button press */
#else if TC13BADGE

138
#define SW1 (FIO2PIN & PIN_SWAP)
#endif

/* R8C control */
#define TC13BADGE
#define R8C_CTL_SET (FIO1SET = PIN_R8C_CTL)
#define R8C_CTL_CLR (FIO1CLR = PIN_R8C_CTL)
#define R8C_ACK (FIO1PIN & PIN_R8C_ACK)
#endif

/* SSEL (SPI slave select) control for CC2400 DIO */
/* (un-buffered) serial */
#if defined UBERTOOTHZERO
#define DIO_SSEL_SET (FIO2SET = PIN_SSEL0)
#define DIO_SSEL_CLR (FIO2CLR = PIN_SSEL0)
#endif
#if defined UBERTOOTHONE || defined TC13BADGE
#define DIO_SSEL_SET (FIO4SET = PIN_SSEL1)
#define DIO_SSEL_CLR (FIO4CLR = PIN_SSEL1)
#endif

/* 1V8 regulator control */
#define CC1V8 (FIO1PIN & PIN_CC1V8)
#define CC1V8_SET (FIO1SET = PIN_CC1V8)
#define CC1V8_CLR (FIO1CLR = PIN_CC1V8)

/* CC2400 control */
#if defined UBERTOOTHZERO
#define CC3V3_SET (FIO1SET = PIN_CC3V3)
#define CC3V3_CLR (FIO1CLR = PIN_CC3V3)
#define RX_SET (FIO1SET = PIN_RX)
#define RX_CLR (FIO1CLR = PIN_RX)
#define TX_SET (FIO1SET = PIN_TX)
#define TX_CLR (FIO1CLR = PIN_TX)
#define CSN_SET (FIO1SET = PIN_CSN)
#define CSN_CLR (FIO1CLR = PIN_CSN)
#define SCLK_SET (FIO1SET = PIN_SCLK)
#define SCLK_CLR (FIO1CLR = PIN_SCLK)
#define MOSI_SET (FIO1SET = PIN_MOSI)
#define MOSI_CLR (FIO1CLR = PIN_MOSI)
#define GIO6 (FIO2PIN & PIN_GIO6)
#define GIO6_SET (FIO1SET = PIN_GIO6)
#define GIO6_CLR (FIO1CLR = PIN_GIO6)
#define BTGR_SET (FIO1SET = PIN_BTGR)
#define BTGR_CLR (FIO1CLR = PIN_BTGR)
#define MISO (FIO1PIN & PIN_MISO)
#endif
#if defined UBERTOOTHONE
#define CC3V3_SET (FIO1SET = PIN_CC3V3)
#define CC3V3_CLR (FIO1CLR = PIN_CC3V3)
#define RX_SET (FIO1SET = PIN_RX)
#define RX_CLR (FIO1CLR = PIN_RX)
#define TX_SET (FIO1SET = PIN_TX)
#define TX_CLR (FIO1CLR = PIN_TX)
#define CSN_SET (FIO2SET = PIN_CSN)
#define CSN_CLR (FIO2CLR = PIN_CSN)
#endif
#define SCLK_SET (FIO2SET = PIN_SCLK)
#define SCLK_CLR (FIO2CLR = PIN_SCLK)
#define MOSL_SET (FIO2SET = PIN_MOSI)
#define MOSL_CLR (FIO2CLR = PIN_MOSI)
#define GIO6 (FIO2PIN & PIN_GIO6)
#define GIO6_SET (FIO2SET = PIN_GIO6)
#define GIO6_CLR (FIO2CLR = PIN_GIO6)
#define BTGR_SET (FIO1SET = PIN_BTGR)
#define BTGR_CLR (FIO1CLR = PIN_BTGR)
#define MISO (FIO2PIN & PIN_MISO)
#endif
#elif defined TC13BADGE
#define CC3V3_SET (FIO1SET = PIN_CC3V3)
#define CC3V3_CLR (FIO1CLR = PIN_CC3V3)
#define CSN_SET (FIO1SET = PIN_CS)
#define CSN_CLR (FIO1CLR = PIN_CS)
#define SCLK_SET (FIO1SET = PIN_SCLK)
#define SCLK_CLR (FIO1CLR = PIN_SCLK)
#define MOSL_SET (FIO1SET = PIN_MOSI)
#define MOSL_CLR (FIO1CLR = PIN_MOSI)
#define GIO6 (FIO1PIN & PIN_GIO6)
#define GIO6_SET (FIO1SET = PIN_GIO6)
#define GIO6_CLR (FIO1CLR = PIN_GIO6)
#define MISO (FIO1PIN & PIN_MISO)
#endif

/* DIO_SSP is the SSP assigned to the CC2400’s secondary */
/* (“un-buffered”) serial interface */
#elif defined UBERTOOTHZERO
#define DIO_SSP_CR0 SSP0CR0
#define DIO_SSP_CR1 SSP0CR1
#define DIO_SSP_DR SSP0DR
#define DIO_SSP_DMACR SSP0DMACR
#define DIO_SSP_SRC (1 << 1) /* for DMACCxConfig register */
#endif
#if defined UBERTOOTHONE || defined TC13BADGE
#define DIO_SSP_CR0 SSP1CR0
#define DIO_SSP_CR1 SSP1CR1
#define DIO_SSP_DR SSP1DR
#define DIO_SSP_DMACR SSP1DMACR
#define DIO_SSP_SRC (3 << 1) /* for DMACCxConfig register */
#endif

/* CC2591 control */
#elif defined UBERTOOTHONE
#define PAEN (FIO2PIN & PIN_PAEN)
#define PAEN_SET (FIO2SET = PIN_PAEN)
#define PAEN_CLR (FIO2CLR = PIN_PAEN)
#define HGM (FIO2PIN & PIN_HGM)
#define HGM_SET (FIO2SET = PIN_HGM)
#define HGM_CLR (FIO2CLR = PIN_HGM)
#endif

/* USB VBUS monitoring */
#define VBUS (FIO1PIN & PIN_VBUS)

/*
 * clock configuration
 * main oscillator: 16 MHz (from CC2400)
 * CPU clock (PLL0): 100 MHz
 * USB clock (PLL1): 48 MHz
 * The ToorCon 13 badge is configured with a 30 MHz CPU clock instead
 * of 100 MHz to reduce heat at the voltage regulator. This is a
 * sufficient clock speed for passive Bluetooth monitoring.
 */

#ifndef TC13BADGE
#define MSEL0 14
#define NSEL0 0
#define CCLKSEL 15
#else
#define MSEL0 24
#define NSEL0 1
#define CCLKSEL 3
#endif

#define MSEL1 34
#define PSEL1 0

/* flash accelerator configuration */
#define FLASHTIM 0x4 /* up to 100 MHz CPU clock */

/* bootloader ctrl is a fixed memory location used for passing
 * information from the application to the bootloader across a reset
 */
extern uint32_t bootloader_ctrl;
#define DFU_MODE 0x4305BB21

void wait(u8 seconds);
void wait_ms(u32 ms);
void wait_us(u32 us);
u32 rbit(u32 value);
void gpio_init(void);
void all_pins_off(void);
void ubertooth_init(void);
void dio_ssp_init(void);
void atest_init(void);
void cc2400_init(void);
u32 cc2400_spi(u8 len, u32 data);
u16 cc2400_get(u8 reg);
void cc2400_set(u8 reg, u16 val);
u8 cc2400_get8(u8 reg);
void cc2400_set8(u8 reg, u8 val);
void cc2400_fifo_write(u8 len, u8 *data);
void cc2400_fifo_read(u8 len, u8 *buf);
u8 cc2400_status(void);
u8 cc2400_strobe(u8 reg);
void cc2400_reset(void);
void clock_start(void);
void reset(void);
void r8c_takeover(void);
void cc2400_tune_rx(uint16_t channel);
void cc2400_tune_tx(uint16_t channel);
void cc2400_hop_rx(uint16_t channel);
void cc2400_hop_tx(uint16_t channel);
void get_part_num(uint8_t *buffer, int *len);
void get_device_serial(uint8_t *buffer, int *len);
void set_isp(void);

#endif /* _UBERTOOTH_H */
# show_libusb_error

```c
void show_libusb_error(int error_code) {
    char *error_hint = "";
    const char *error_name;

    /* Available only in libusb > 1.0.3 */
    /* error_name = libusb_error_name(error_code); */

    switch (error_code) {
        case LIBUSB_ERROR_TIMEOUT:
            error_name = "Timeout";
            break;
        case LIBUSB_ERROR_NODEVICE:
            error_name = "No Device";
            error_hint = "Check Ubertooth is connected to host";
            break;
        case LIBUSB_ERROR_ACCESS:
            error_name = "Insufficient Permissions";
            break;
        case LIBUSB_ERROR_OVERFLOW:
            error_name = "Overflow";
            error_hint = "Try resetting the Ubertooth";
            break;
        default:
            error_name = "Command Error";
            break;
    }

    fprintf(stderr, "libUSB Error: %s: %s (%d)\n", \
            error_name, error_hint, error_code);
}
```
static void callback(struct libusb_transfer* transfer)
{
    if (transfer->status != 0) {
        show_libusb_error(transfer->status);
    }
    libusb_free_transfer(transfer);
}

void cmd_trim_clock(struct libusb_device_handle* devh, uint16_t offset)
{
    uint8_t data[2] = {
        (offset >> 8) & 0xff,
        (offset >> 0) & 0xff
    };
    ubertooth_cmd_async(devh, CTRL_OUT, UBERTOOTH_TRIM_CLOCK, data, 2);
}

void cmd_fix_clock_drift(struct libusb_device_handle* devh, int16_t ppm)
{
    uint8_t data[2] = {
        (ppm >> 8) & 0xff,
        (ppm >> 0) & 0xff
    };
    ubertooth_cmd_async(devh, CTRL_OUT, UBERTOOTH_FIX_CLOCK_DRIFT, data, 2);
}

int cmd_ping(struct libusb_device_handle* devh)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_PING, 0, 0,
                                 NULL, 0, 1000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return 0;
}

int cmd_rx_syms(struct libusb_device_handle* devh)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_RX_SYMBOLS, 0, 0,
                                 NULL, 0, 1000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return 0;
}

int cmd_tx_syms(struct libusb_device_handle* devh)
{
return \n    ubertooth_cmd_sync(devh, CTRL_OUT, UBERTOOTH_TX_SYMBOLS, 0, 0);
}

int cmd_sync(struct libusb_device_handle* devh, \n              u16 low_freq, u16 high_freq)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SYMBOLS, \n                                low_freq, high_freq, NULL, 0, 1000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return 0;
}

int cmd_specan(struct libusb_device_handle* devh, \n                u16 low_freq, u16 high_freq)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SPECAN, \n                                low_freq, high_freq, NULL, 0, 1000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return 0;
}

int cmd_led_specan(struct libusb_device_handle* devh, \n                    u16 rssi_threshold)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_LED_SPECAN, \n                                rssi_threshold, 0, NULL, 0, 1000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return 0;
}

int cmd_set_usrled(struct libusb_device_handle* devh, u16 state)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SET_USRLED, \n                                state, 0, NULL, 0, 1000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return 0;
}

int cmd_get_usrled(struct libusb_device_handle* devh)
{
    u8 state;
    int r;
    r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_GET_USRLED, 0, \n                                0, &state, 1, 1000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return state;
}
int cmd_set_rxled(struct libusb_device_handle* devh, u16 state)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SET_RXLED, 
        state, 0, NULL, 0, 1000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return 0;
}

int cmd_get_rxled(struct libusb_device_handle* devh)
{
    u8 state;
    int r;
    r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_GET_RXLED, 0, 0, 
        &state, 1, 1000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return state;
}

int cmd_set_txled(struct libusb_device_handle* devh, u16 state)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SET_TXLED, 
        state, 0, NULL, 0, 1000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return 0;
}

int cmd_get_txled(struct libusb_device_handle* devh)
{
    u8 state;
    int r;
    r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_GET_TXLED, 0, 
        &state, 1, 1000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return state;
}

int cmd_get_modulation(struct libusb_device_handle* devh)
{
    u8 modulation;
    int r;
r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_GET_MOD, 0, 0, 
   &modulation, 1, 1000);
if (r < 0) {
    show_libusb_error(r);
    return r;
}
return modulation;
}

int cmd_get_channel(struct libusb_device_handle* devh)
{
    u8 result[2];
    int r;
    r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_GET_CHANNEL, 
       0, 0, result, 2, 1000);
    if (r == LIBUSB_ERROR_PIPE) {
        fprintf(stderr, "control message unsupported\n");
        return r;
    } else if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return result[0] | (result[1] << 8);
}

int cmd_set_channel(struct libusb_device_handle* devh, u16 channel)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SET_CHANNEL, 
       channel, 0, NULL, 0, 1000);
    if (r == LIBUSB_ERROR_PIPE) {
        fprintf(stderr, "control message unsupported\n");
        return r;
    } else if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return 0;
}

int cmd_get_partnum(struct libusb_device_handle* devh)
{
    u8 result[5];
    int r;
    r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_GET_PARTNUM, 
       0, 0, result, 5, 1000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    if (result[0] != 0) {
        fprintf(stderr, "result not zero: %d\n", result[0]);
    }
```c
    return 0;
  }

void print_serial(u8 *serial, FILE *fileptr)
{
    int i;
    if (fileptr == NULL)
        fileptr = stdout;
    fprintf(fileptr, "Serial No: ");
    for (i=1; i<17; i++)
        fprintf(fileptr, "%02x", serial[i]);
    fprintf(fileptr, "\n");
}

int cmd_get_serial(struct libusb_device_handle* devh, u8 *serial)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_GET_SERIAL, \   0, 0, serial, 17, 1000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    if (serial[0] != 0) {
        fprintf(stderr, "result not zero: \n", serial[0]);
        return 1;
    }
    return 0;
}

int cmd_set_modulation(struct libusb_device_handle* devh, u16 mod)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SET_MOD, \   mod, 0, NULL, 0, 1000);
    if (r == LIBUSB_ERRORPIPE) {
        fprintf(stderr, "control message unsupported\n");
        return r;
    } else if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return 0;
}

int cmd_set_isp(struct libusb_device_handle* devh)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SET_ISP, 0, 0, \   NULL, 0, 1000);
    /* LIBUSB_ERRORPIPE or LIBUSB_ERROROTHER is expected */
    if (r && (r != LIBUSB_ERRORPIPE) && (r != LIBUSB_ERROROTHER) &&
```
(r != LIBUSB_ERROR_NO_DEVICE)) {
    show_libusb_error(r);
    return r;
}
return 0;
}

int cmd_reset(struct libusb_device_handle* devh) {
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_RESET, 0, 0,
                     NULL, 0, 1000);
    /* LIBUSB_ERROR_PIPE or LIBUSB_ERROR_OTHER is expected */
    if (r && (r != LIBUSB_ERROR_PIPE) && (r != LIBUSB_ERROR-other) &&
        (r != LIBUSB_ERROR_NO_DEVICE)) {
        show_libusb_error(r);
        return r;
    }
    return 0;
}

int cmd_stop(struct libusb_device_handle* devh) {
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_STOP, 0, 0,
                     NULL, 0, 1000);
    if (r == LIBUSB_ERROR_PIPE) {
        fprintf(stderr, "control message unsupported\n");
        return r;
    } else if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return 0;
}

int cmd_set_paen(struct libusb_device_handle* devh, u16 state) {
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SET_PAEN, \
                     state, 0, NULL, 0, 1000);
    if (r == LIBUSB_ERROR_PIPE) {
        fprintf(stderr, "control message unsupported\n");
        return r;
    } else if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return 0;
}

int cmd_set_hgm(struct libusb_device_handle* devh, u16 state) {
    int r;
```c
r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SET_HGM, \  
    state, 0, NULL, 0, 1000);
if (r == LIBUSB_ERROR_PIPE) {
    fprintf(stderr, "control message unsupported\n");
    return r;
} else if (r < 0) {
    show_libusb_error(r);
    return r;
}
return 0;

int cmd_tx_test(struct libusb_device_handle* devh)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_TX_TEST, 0, 0, \  
        NULL, 0, 1000);
    if (r == LIBUSB_ERROR_PIPE) {
        fprintf(stderr, "control message unsupported\n");
        return r;
    } else if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return 0;
}

int cmd_flash(struct libusb_device_handle* devh)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_FLASH, 0, 0, \  
        NULL, 0, 1000);
    if (r != LIBUSB_SUCCESS) {
        show_libusb_error(r);
        return r;
    }
    return 0;
}

int cmd_get_palevel(struct libusb_device_handle* devh)
{
    u8 level;
    int r;
    r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_GET_PALEVEL, \  
        0, 0, &level, 1, 3000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return level;
}

int cmd_set_palevel(struct libusb_device_handle* devh, u16 level)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_SET_PALEVEL, \  
        0, 0, &level, 1, 3000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return level;
}
```

r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SET_PALEVEL, \
    level, 0, NULL, 0, 3000);
if (r != LIBUSB_SUCCESS) {
    if (r == LIBUSB_ERRORPIPE) {
        fprintf(stderr, "control message unsupported\n");
    } else {
        show_libusb_error(r);
    }
    return r;
}
return 0;

int cmd_get_rangeresult(struct libusb_device_handle* devh, 
    rangetest_result *rr)
{
u8 result[5];
int r;

r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_RANGE_CHECK, \
    0, 0, result, sizeof(result), 3000);
if (r < LIBUSB_SUCCESS) {
    if (r == LIBUSB_ERRORPIPE) {
        fprintf(stderr, "control message unsupported\n");
    } else {
        show_libusb_error(r);
    }
    return r;
}

rr->valid = result[0];
rr->request_pa = result[1];
rr->request_num = result[2];
rr->reply_pa = result[3];
rr->reply_num = result[4];
return 0;

int cmd_range_test(struct libusb_device_handle* devh)
{
    int r;

    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_RANGE_TEST, \
        0, 0, NULL, 0, 1000);
    if (r != LIBUSB_SUCCESS) {
        if (r == LIBUSB_ERRORPIPE) {
            fprintf(stderr, "control message unsupported\n");
        } else {
            show_libusb_error(r);
        }
        return r;
    }
    return 0;
519 int cmd_repeater(struct libusb_device_handle* devh)
520 {
521 int r;
522
523 r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_REPEATER, \
524 0, 0, NULL, 0, 1000);
525 if (r != LIBUSB_SUCCESS) {
526 if (r == LIBUSB_ERROR_PIPE) {
527 fprintf(stderr, "control message unsupported\n");
528 } else {
529 show_libusb_error(r);
530 }
531 return r;
532 }
533 return 0;
534 }
535
536 void cmd_get_rev_num(struct libusb_device_handle* devh, \
537 char *version, u8 len)
538 {
539 u8 result[2 + 1 + 255];
540 u16 result_ver;
541 int r;
542
543 r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_GET_REV_NUM, \
544 0, 0, result, sizeof(result), 1000);
545 if (r == LIBUSB_ERROR_PIPE) {
546 fprintf(stderr, "control message unsupported\n");
547 snprintf(version, len - 1, "error: %d", r);
548 version[len - 1] = '\0';
549 return;
550 }
551 else if (r < 0) {
552 show_libusb_error(r);
553 snprintf(version, len - 1, "error: %d", r);
554 version[len - 1] = '\0';
555 return;
556 }
557
558 result_ver = result[0] | (result[1] << 8);
559 if (r == 2) { /* old-style SVN rev */
560 snprintf(version, "%u", result_ver);
561 } else {
562 len = MIN(r - 3, MIN(len - 1, result[2]));
563 memcpy(version, &result[3], len);
564 version[len] = '\0';
565 }
566 }
567
568 void cmd_get_compile_info(struct libusb_device_handle* devh, \
569 char *compile_info, u8 len)
570 {
571 u8 result[1 + 255];
572 int r;
573
574 r = libusb_control_transfer(devh, CTRL_IN, \
575 UBERTOOTH_GET_COMPILEINFO, 0, 0, result, sizeof(result), 1000);
576 if (r == LIBUSB_ERROR_PIPE) {
577 fprintf(stderr, "control message unsupported\n");
578 snprintf(compile_info, len - 1, "error: %d", r);
compile_info[LEN - 1] = '\0';
return;
} else if (r < 0) {
    show_libusb_error(r);
    snprintf(compile_info, len - 1, "error: %d", r);
    compile_info[LEN - 1] = '\0';
    return;
}

len = MIN(r - 1, MIN(len - 1, result[0]));
memcpy(compile_info, &result[1], len);
compile_info[len] = '\0';

int cmd_get_board_id(struct libusb_device_handle* devh) {
    u8 board_id;
    int r;
    r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_GET_BOARD_ID,
        0, 0, &board_id, 1, 1000);
    if (r == LIBUSB_ERROR_PIPE) {
        fprintf(stderr, "control message unsupported\n");
        return r;
    } else if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return board_id;
}

int cmd_set_squelch(struct libusb_device_handle* devh, u16 level) {
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SET_SQUELCH,
        level, 0, NULL, 0, 3000);
    if (r != LIBUSB_SUCCESS) {
        if (r == LIBUSB_ERROR_PIPE) {
            fprintf(stderr, "control message unsupported\n");
        } else {
            show_libusb_error(r);
        }
        return r;
    }
    return 0;
}

int cmd_get_squelch(struct libusb_device_handle* devh) {
    u8 level;
    int r;
    r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_GET_SQUELCH,
        0, 0, &level, 1, 3000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
int cmd_set_bdaddr(struct libusb_device_handle* devh, u64 address)
{
    int r, data_len;
    u64 syncword;
    data_len = 16;
    unsigned char data[data_len];

    syncword = btbb_gen_syncword(address & 0xffffffff);
    /*printf("syncword=%#llx\n", syncword); */
    for(r=0; r < 8; r++)
        data[r] = (address >> (8*r)) & 0xff;
    for(r=0; r < 8; r++)
        data[r+8] = (syncword >> (8*r)) & 0xff;

    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SET BDADDR, \
                                0, 0, data, data_len, 1000);
    if (r < 0) {
        if (r == LIBUSB_ERROR_PIPE) {
            fprintf(stderr, "control message unsupported\n");
        } else {
            show_libusb_error(r);
        }
        return r;
    } else if (r < data_len) {
        fprintf(stderr, "Only %d of %d bytes transferred\n", r, data_len);
        return 1;
    }
    return 0;
}

int cmd_start_hopping(struct libusb_device_handle* devh, \
                        int clkn_offset, int clk100ns_offset)
{
    int r;
    uint8_t data[6];
    for(r=0; r < 4; r++)
        data[r] = (clkn_offset >> (8*(3-r))) & 0xff;
    data[4] = (clk100ns_offset >> 8) & 0xff;
    data[5] = (clk100ns_offset >> 0) & 0xff;

    r = ubertooth_cmd_async(devh, CTRL_OUT, \
                             UBERTOOTH_START_HOPPING, data, 6);
    if (r < 0) {
        if (r == LIBUSB_ERROR_PIPE) {
            fprintf(stderr, "control message unsupported\n");
        } else {
            show_libusb_error(r);
        }
        return r;
    }
    return 0;
}
```c
int cmd_set_clock(struct libusb_device_handle* devh, u32 clkn)
{
    int r;
    u8 data[4];
    for(r=0; r < 4; r++)
    {
        data[r] = (clkn >> (8*r)) & 0xff;
    }
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_SET_CLOCK, \ 
        0, 0, data, 4, 1000);
    if (r < 0) {
        if (r == LIBUSB_ERRORPIPE) {
            fprintf(stderr, "control message unsupported\n");
        } else {
            show_libusb_error(r);
        }
    return r;
}
return 0;
}

uint32_t cmd_get_clock(struct libusb_device_handle* devh)
{
    u32 clock = 0;
    unsigned char data[4];
    int r;
    r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_GET_CLOCK, 0, 0, \ 
        data, 4, 3000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    printf("Read clock = 0x%x\n", clock);
    return clock;
}

int cmd_btle_sniffing(struct libusb_device_handle* devh, u16 num)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_BTLE_SNIFFING, \ 
        num, 0, NULL, 0, 1000);
    if (r < 0) {
        if (r == LIBUSB_ERRORPIPE) {
            fprintf(stderr, "control message unsupported\n");
        } else {
            show_libusb_error(r);
        }
    return r;
    }
    return 0;
}

int cmd_btle_multi_sniffing(
    struct libusb_device_handle* devh, u16 num) {
```
printf("in the function to send multi\n");
int r;
r = libusb_control_transfer(devh, CTRL_OUT, 
    UBERTOOTH_BTLE_MULTI_SNIFFING, num, 0, NULL, 0, 1000);
printf("Returned from the control handler, r = %d\n", r);
if (r < 0) {
    if (r == LIBUSB_ERROR_PIPE) {
        fprintf(stderr, "control message unsupported\n");
    } else {
        show_libusb_error(r);
    }
    return r;
}
return 0;

int cmd_set_afh_map(struct libusb_device_handle* devh, 
    uint8_t* afh_map)
{
    uint8_t buffer[LIBUSB_CONTROL_SETUP_SIZE+10];
    struct libusb_transfer *xfer = libusb_alloc_transfer(0);
    libusb_fill_control_setup(buffer, CTRL_OUT, 
        UBERTOOTH_SET_AFHMAP, 
        LIBUSB_CONTROL_SETUP_SIZE, 0, 0, 10);
    memcpy(&buffer[LIBUSB_CONTROL_SETUP_SIZE], afh_map, 10);
    libusb_fill_control_transfer(xfer, devh, buffer, callback, 
        NULL, 1000);
    libusb_submit_transfer(xfer);
    return 0;
}

int cmd_clear_afh_map(struct libusb_device_handle* devh)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_CLEAR_AFHMAP, 
        0, 0, NULL, 0, 1000);
    if (r < 0) {
        if (r == LIBUSB_ERROR_PIPE) {
            fprintf(stderr, "control message unsupported\n");
        } else {
            show_libusb_error(r);
        }
        return r;
    }
    return 0;
}

u32 cmd_get_access_address(struct libusb_device_handle* devh)
{
    u32 access_address = 0;
    unsigned char data[4];
    int r;
    r = libusb_control_transfer(devh, CTRL_IN, 
        UBERTOOTH_GET_ACCESS_ADDRESS, 0, 0, data, 4, 3000);
if (r < 0) {
    show_libusb_error(r);
    return r;
}

access_address = data[0] | data[1] << 8 | \
return access_address;

int cmd_set_access_address(
    struct libusb_device_handle* devh, u32 access_address)
{
    int r;
    u8 data[4];
    for(r=0; r < 4; r++)
        data[r] = (access_address >> (8*r)) & 0xff;
    r = libusb_control_transfer(devh, CTRL_OUT, \
        UBERTOOTH_SET_ACCESS_ADDRESS, 0, 0, data, 4, 1000);
    if (r < 0) {
        if (r == LIBUSB_ERROR_PIPE) {
            fprintf(stderr, "control message unsupported\n");
        } else {
            show_libusb_error(r);
        }
        return r;
    }
    return 0;
}

int cmd_do_something(
    struct libusb_device_handle *devh, \
    unsigned char *data, int len)
{
    int r = libusb_control_transfer(devh, CTRL_OUT, \
        UBERTOOTH_DO_SOMETHING, 0, 0, data, len, 1000);
    if (r < 0) {
        if (r == LIBUSB_ERROR_PIPE) {
            fprintf(stderr, "control message unsupported\n");
        } else {
            show_libusb_error(r);
        }
        return r;
    }
    return 0;
}

int cmd_do_something_reply(
    struct libusb_device_handle* devh, unsigned char *data, int len)
{
    int r = libusb_control_transfer(devh, CTRL_IN, \
        UBERTOOTH_DO_SOMETHING_REPLY, 0, 0, data, len, 3000);
    if (r < 0) {
        if (r == LIBUSB_ERROR_PIPE) {
            fprintf(stderr, "control message unsupported\n");
        } else {
            show_libusb_error(r);
        }
    }
return r;
}
return r;
}

int cmd_get_crc_verify(struct libusb_device_handle* devh)
{
    u8 verify;
    int r;
    r = libusb_control_transfer(devh, CTRL_IN, \
        UBERTOOTH_GET_CRC_VERIFY, 0, 0, &verify, 1, 1000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return verify;
}

int cmd_set_crc_verify(struct libusb_device_handle* devh, int verify)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, \
        UBERTOOTH_SET_CRC_VERIFY, verify, 0, NULL, 0, 1000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return 0;
}

int cmd_poll(struct libusb_device_handle* devh, usb_pkt_rx *p)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_IN, UBERTOOTH_POLL, 0, 0, \
        (u8 *)p, sizeof(usb_pkt_rx), 1000);
    if (r < 0) {
        show_libusb_error(r);
        return r;
    }
    return r;
}

int cmd_btle_promisc(struct libusb_device_handle* devh)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, \
        UBERTOOTH_BTLE_PROMISC, 0, 0, NULL, 0, 1000);
    if (r < 0) {
        if (r == LIBUSB_ERROR_PIPE) {
            printf(stderr, "control message unsupported\n");
        } else {
            show_libusb_error(r);
        }
    }
    return r;
}
int cmd_read_register(struct libusb_device_handle* devh, u8 reg)
{
    int r;
    u8 data[2];
    r = libusb_control_transfer(devh, CTRL_IN, 
                               UBERTOOTH_READ_REGISTER, reg, 0, data, 2, 1000);
    if (r < 0) {
        if (r == LIBUSB_ERRORPIPE) {
            fprintf(stderr, "control message unsupported\n");
        } else {
            show_libusb_error(r);
        }
    return r;
    }
    return (data[0] << 8) | data[1];
}

int cmd_btle_slave(struct libusb_device_handle* devh, u8 *mac_address)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, 
                               UBERTOOTH_BTLE_SLAVE, 0, 0, mac_address, 6, 1000);
    if (r < 0) {
        if (r == LIBUSB_ERRORPIPE) {
            fprintf(stderr, "control message unsupported\n");
        } else {
            show_libusb_error(r);
        }
    return r;
    }
    return 0;
}

int cmd_btle_set_target(
    struct libusb_device_handle* devh, u8 *mac_address)
{
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, 
                               UBERTOOTH_BTLE_SET_TARGET, 0, 0, mac_address, 6, 1000);
    if (r < 0) {
        if (r == LIBUSB_ERRORPIPE) {
            fprintf(stderr, "control message unsupported\n");
        } else {
            show_libusb_error(r);
        }
    return r;
    }
    return 0;
}
int cmd_set_jam_mode(struct libusb_device_handle* devh, int mode) {
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, \
        UBERTOOTH_JAM_MODE, mode, 0, NULL, 0, 1000);
    if (r < 0) {
        if (r == LIBUSB_ERROR_PIPE) {
            fprintf(stderr, "control message unsupported\n");
        } else {
            show_libusb_error(r);
        }
    return r;
    }
    return 0;
}

int cmd_ego(struct libusb_device_handle* devh, int mode) {
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_EGO, mode, 0, \
        NULL, 0, 1000);
    if (r < 0) {
        if (r == LIBUSB_ERROR_PIPE) {
            fprintf(stderr, "control message unsupported\n");
        } else {
            show_libusb_error(r);
        }
    return r;
    }
    return 0;
}

int cmd_afh(struct libusb_device_handle* devh) {
    int r;
    r = libusb_control_transfer(devh, CTRL_OUT, UBERTOOTH_AFH, 0, 0, \
        NULL, 0, 1000);
    if (r < 0) {
        if (r == LIBUSB_ERROR_PIPE) {
            fprintf(stderr, "control message unsupported\n");
        } else {
            show_libusb_error(r);
        }
    return r;
    }
    return 0;
}

int cmd_hop(struct libusb_device_handle* devh) {
    uint8_t buffer [LIBUSB_CONTROL_SETUP_SIZE];
    struct libusb_transfer *xfer = libusb_alloc_transfer(0);
    libusb_fill_control_setup(buffer, CTRL_OUT, UBERTOOTH_HOP, 0, 0, 0);
    libusb_fill_control_transfer(xfer, devh, buffer, \
callback, NULL, 1000);
libusb_submit_transfer(xfer);
return 0;
}

int32_t cmd_api_version(struct libusb_device_handle* devh) {
  unsigned char data[4];
  int r;
  r = libusb_control_transfer(devh, CTRL_IN, \
    UBERTOOTH_GET_API_VERSION, 0, 0, data, 4, 3000);
  if (r < 0) {
    show_libusb_error(r);
    return r;
  }
}

int ubertooth_cmd_sync(struct libusb_device_handle* devh,
                        uint8_t type,
                        uint8_t command,
                        uint8_t* data,
                        uint16_t size)
{
  int r;
  r = libusb_control_transfer(devh, type, command, 0, 0,
    data, size, 1000);
  if (r < 0) {
    if (r == LIBUSB_ERROR_PIPE) {
      fprintf(stderr, "control message unsupported\n");
    } else {
      show_libusb_error(r);
    }
    return r;
  }
  return 0;
}

int ubertooth_cmd_async(struct libusb_device_handle* devh,
                         uint8_t type,
                         uint8_t command,
                         uint8_t* data,
                         uint16_t size)
{
  int r = 0;
  uint8_t buffer[LIBUSB_CONTROL_SETUP_SIZE + size];
  struct libusb_transfer* xfer = libusb_alloc_transfer(0);
  libusb_fill_control_setup(buffer, type, command, 0, 0, size);
  if (size > 0)
    memcpy(&buffer[LIBUSB_CONTROL_SETUP_SIZE], data, size);
  libusb_fill_control_transfer(
    xfer, devh, buffer, callback, NULL, 1000);
  xfer->status = LIBUSB_TRANSFER_FREE_BUFFER | LIBUSB_TRANSFER_FREE_TRANSFER;
r = libusb_submit_transfer(xfer);
if (r < 0)
    show_libusb_error(r);
return r;
1.7 host/libubertooth/src/ubertooth_interface.h

/*
 * Copyright 2016 Air Force Institute of Technology, U.S. Air Force
 * Copyright 2012 Dominic Spill
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 * You should have received a copy of the GNU General Public License
 * along with this program; see the file COPYING. If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110-1301, USA.
*/

#ifndef _UBERTOOTH_INTERFACE_H
#define _UBERTOOTH_INTERFACE_H

#include <stdint.h>

/* increment on every API change */
#define UBERTOOTH_API_VERSION 1
#define DMA_SIZE 50
#define NUM_BREDR_CHANNELS 79

/*
 * CLK_TUNE_TIME is the duration in units of 100 ns that we reserve
 * for tuning the radio while frequency hopping. We start the tuning
 * process CLK_TUNE_TIME * 100 ns prior to the start of an upcoming
 * time slot.
*/
#define CLK_TUNE_TIME 2250
#define CLK_TUNE_OFFSET 200

enum ubertooth_usb_commands {
    UBERTOOTH_PING = 0,
    UBERTOOTH_RX_SYMBOLS = 1,
    UBERTOOTH_TX_SYMBOLS = 2,
    UBERTOOTH_GET_USRLED = 3,
    UBERTOOTH_SET_USRLED = 4,
    UBERTOOTH_GET_RXLED = 5,
    UBERTOOTH_SET_RXLED = 6,
    UBERTOOTH_GET_TXLED = 7,
    UBERTOOTH_SET_TXLED = 8,
    UBERTOOTH_GET_1V8 = 9,
    UBERTOOTH_SET_1V8 = 10,
    UBERTOOTH_GET_CHANNEL = 11,
    UBERTOOTH_SET_CHANNEL = 12,
};

163
UBERTooth_RESET = 13,
UBERTooth_GET_SERIAL = 14,
UBERTooth_GET_PARTNUM = 15,
UBERTooth_GET_PAEN = 16,
UBERTooth_SET_PAEN = 17,
UBERTooth_GET_HGM = 18,
UBERTooth_SET_HGM = 19,
UBERTooth_TX_TEST = 20,
UBERTooth_STOP = 21,
UBERTooth_GET_MOD = 22,
UBERTooth_SET_MOD = 23,
UBERTooth_SET_ISP = 24,
UBERTooth_FLASH = 25,
BOOTLOADER_FLASH = 26,
UBERTooth_SPECAN = 27,
UBERTooth_GET_PALEVEL = 28,
UBERTooth_SET_PALEVEL = 29,
UBERTooth_REPEATER = 30,
UBERTooth_RANGE_TEST = 31,
UBERTooth_RANGE_CHECK = 32,
UBERTooth_GET_REV_NUM = 33,
UBERTooth_LED_SPECAN = 34,
UBERTooth_GET_BOARD_ID = 35,
UBERTooth_SET_SQUELCH = 36,
UBERTooth_GET_SQUELCH = 37,
UBERTooth_SET_B_ADDR = 38,
UBERTooth_START_HOPPING = 39,
UBERTooth_SET_CLOCK = 40,
UBERTooth_GET_CLOCK = 41,
UBERTooth_BTLE_SNIFFING = 42,
UBERTooth_GET_ACCESS_ADDRESS = 43,
UBERTooth_SET_ACCESS_ADDRESS = 44,
UBERTooth_DO_SOMETHING = 45,
UBERTooth_DO_SOMETHING_REPLY = 46,
UBERTooth_GET_CRC_VERIFY = 47,
UBERTooth_SET_CRC_VERIFY = 48,
UBERTooth_POLL = 49,
UBERTooth_BTLE_PROMISC = 50,
UBERTooth_SET_AFHMAP = 51,
UBERTooth_CLEAR_AFHMAP = 52,
UBERTooth_READ_REGISTER = 53,
UBERTooth_BTLE_SLAVE = 54,
UBERTooth_GET_COMPILE_INFO = 55,
UBERTooth_BTLE_SET_TARGET = 56,
UBERTooth_BTLE_PHY = 57,
UBERTooth_WRITE_REGISTER = 58,
UBERTooth_JAM_MODE = 59,
UBERTooth_EGO = 60,
UBERTooth_AFI = 61,
UBERTooth_HOP = 62,
UBERTooth_TRIM_CLOCK = 63,
UBERTooth_GET_API_VERSION = 64,
UBERTooth_WRITE_REGISTERS = 65,
UBERTooth_READ_ALL_REGISTERS = 66,
UBERTooth_RX_GENERIC = 67,
UBERTooth_TX_GENERIC_PACKET = 68,
UBERTooth_FIX_CLOCK_DRIFT = 69,
UBERTooth_BTLE_MULTI_SNIFFING = 70,
enum jam_modes {
  JAM_NONE = 0,
  JAM_ONCE = 1,
  JAM_CONTINUOUS = 2,
};

enum modulations {
  MOD_BT_BASIC_RATE = 0,
  MOD_BT_LOW_ENERGY = 1,
  MOD_80211_FHSS = 2,
  MOD_NONE = 3
};

enum usb_pkt_types {
  BR_PACKET = 0,
  LE_PACKET = 1,
  MESSAGE = 2,
  KEEP_ALIVE = 3,
  SPECAN = 4,
  LE_PROMISC = 5,
  EGO_PACKET = 6,
};

enum hop_mode {
  HOP_NONE = 0,
  HOP_SWEEP = 1,
  HOP_BLUETOOTH = 2,
  HOP_BTLE = 3,
  HOP_BTLE_MULTI = 4,
  HOP_DIRECT = 5,
  HOP_AFH = 6,
};

enum usb_pkt_status {
  DMA_OVERFLOW = 0x01,
  DMA_ERROR = 0x02,
  FIFO_OVERFLOW = 0x04,
  CS_TRIGGER = 0x08,
  RSSI_TRIGGER = 0x10,
  DISCARD = 0x20,
};

typedef struct {
  u8 pkt_type;
  u8 status;
  u8 channel;
  u8 clkn_high;
  u32 clk100ns;
  /* Max RSSI seen while collecting symbols in this packet */
  char rssi_max;
  /* Min ... */
  char rssi_min;
  /* Average ... */
}
char rssi_avg;

/* Number of ... (0 means RSSI stats are invalid) */

u8 rssi_count;

u8 reserved[2];

u8 data[DMA_SIZE];

} usb_pkt_rx;

typedef struct {
    u64 address;
    u64 syncword;
} bdaddr;

typedef struct {
    u8 valid;
    u8 request_pa;
    u8 request_num;
    u8 reply_pa;
    u8 reply_num;
} rangetest_result;

typedef struct {
    u16 synch;
    u16 syncl;
    u16 channel;
    u8 length;
    u8 pa_level;
    u8 data[DMA_SIZE];
} generic_tx_packet;

#endif /* _UBERTOOTH_INTERFACE_H */
1.8  host/libubertooth/src/ubertooth.c

/*
 * Copyright 2016 Jose Gutierrez del Arroyo
 * Copyright 2010, 2011 Michael Ossmann
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 * You should have received a copy of the GNU General Public License
 * along with this program; see the file COPYING. If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110-1301, USA.
 */

#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

#include "ubertooth.h"
#include "ubertooth_callback.h"
#include "ubertooth_control.h"
#include "ubertooth_interface.h"

#ifndef RELEASE
#define RELEASE "unknown"
#endif
#ifndef VERSION
#define VERSION "unknown"
#endif

uint32_t systime;
FILE* infile = NULL;
FILE* dumpfile = NULL;
int max_ac_errors = 2;

int do_exit = 1;
pthread_t poll_thread;

unsigned int packet_counter_max;

void print_version() {
    printf("libubertooth %s (%s), libbtbb %s (%s)\n", VERSION, RELEASE,
        btbb_get_version(), btbb_get_release());
}

167
ubertooth_t * cleanup_devh = NULL;

static void cleanup(int sig __attribute__((unused)))
{
    if (cleanup_devh)
        cleanup_devh->stop_ubertooth = 1;
}

static void cleanup_exit(int sig __attribute__((unused)))
{
    if (cleanup_devh)
        ubertooth_stop(cleanup_devh);
    exit(0);
}

void register_cleanup_handler(ubertooth_t* ut, int do_exit)
{
    cleanup_devh = ut;
    
    /* Clean up on ctrl-C. */
    if (do_exit)
    {
        signal(SIGINT, cleanup_exit);
        signal(SIGQUIT, cleanup_exit);
        signal(SIGTERM, cleanup_exit);
    } else {
        signal(SIGINT, cleanup);
        signal(SIGQUIT, cleanup);
        signal(SIGTERM, cleanup);
    }
}

ubertooth_t * timeout_dev = NULL;

void stop_transfers(int sig __attribute__((unused)))
{
    if (timeout_dev)
        timeout_dev->stop_ubertooth = 1;
}

void ubertooth_set_timeout(ubertooth_t* ut, int seconds)
{
    /* Upon SIGALRM, call stop_transfers() */
    if (signal(SIGALRM, stop_transfers) == SIG_ERR) {
        perror("Unable to catch SIGALRM");
        exit(1);
    }
    timeout_dev = ut;
    alarm(seconds);
}

static struct libusb_device_handle* \\nfind_ubertooth_device(int ubertooth_device)
{
    struct libusb_context *ctx = NULL;
    struct libusb_device **usb_list = NULL;
    struct libusb_device_handle *devh = NULL;
    struct libusb_device_descriptor desc;
    int usb_devs, i, r, ret, ubertooths = 0;
    int ubertooth_devs[] = {0,0,0,0,0,0,0,0};
    
    usb_devs = libusb_get_device_list(ctx, &usb_list);
    for (i = 0 ; i < usb_devs ; ++i) {

r = libusb_get_device_descriptor(usb_list[i], &desc);
if (r < 0)
  fprintf(stderr, "couldn't get usb descriptor for dev #%d\n", i);
if ((desc.idVendor == TC13_VENDORID &&
     desc.idProduct == TC13_PRODUCTID)
  || (desc.idVendor == U0_VENDORID &&
     desc.idProduct == U0_PRODUCTID)
  || (desc.idVendor == U1_VENDORID &&
     desc.idProduct == U1_PRODUCTID))
  {  
    ubertooth_devs[ubertooths] = i;
    ubertooths++;
  }
if (ubertooths == 1) {
  ret = libusb_open(usb_list[ubertooth_devs[0]], &devh);
  if (ret)
    show_libusb_error(ret);
} else if (ubertooths == 0)
  return NULL;
else {
  if (ubertooth_device < 0) {
    fprintf(stderr, "multiple Ubertooth devices found!\nUse '-U' to specify device number\n"");
    uint8_t serial[17], r;
    for (i = 0; i < ubertooths; ++i) {
      libusb_get_device_descriptor(
        usb_list[ubertooth_devs[i]], &desc);
      ret = libusb_open(usb_list[ubertooth_devs[i]], &devh);
      if (ret) {
        fprintf(stderr, "Device %d: ", i);
        show_libusb_error(ret);
      }
    }
    else {
      r = cmd_get_serial(devh, serial);
      if (r==0) {
        fprintf(stderr, "Device %d: ", i);
        print_serial(serial, stderr);
      }
      libusb_close(devh);
    }
  }
  devh = NULL;
} else {
  ret = libusb_open(usb_list[ubertooth_devs[ubertooth_device]], &devh);
  if (ret) {
    show_libusb_error(ret);
    devh = NULL;
  }
}
return devh;
}

/* based on http://*libusb.sourceforge.net/api-1.0/
group__asyncio.html#ga9fcb2a23d342060ebda1d0cf7478856 */
static void rx_xfer_status(int status)
{
char *error_name = "";
switch (status) {
    case LIBUSB_TRANSFER_ERROR:
        error_name = "Transfer error."
        break;
    case LIBUSB_TRANSFER_TIMED_OUT:
        error_name = "Transfer timed out."
        break;
    case LIBUSB_TRANSFER_CANCELLED:
        error_name = "Transfer cancelled."
        break;
    case LIBUSB_TRANSFERSTALL:
        error_name = "Halt condition detected, or control request not supported."
        break;
    case LIBUSB_TRANSFER_NO_DEVICE:
        error_name = "Device disconnected."
        break;
    case LIBUSB_TRANSFER_OVERFLOW:
        error_name = "Device sent more data than requested."
        break;
}
fprintf(stderr, "rx_xfer status: %s (%d)\n", error_name, status);
}

static void cb_xfer(struct libusb_transfer *xfer)
{
int r;
ubertooth_t * ut = (ubertooth_t *)xfer->user_data;
if (xfer->status != LIBUSB_TRANSFER_COMPLETED) {
    if (xfer->status == LIBUSB_TRANSFER_TIMED_OUT) {
        r = libusb_submit_transfer(ut->rx_xfer);
        if (r < 0)
            fprintf(stderr, "Failed to submit USB transfer (%d)\n", r);
        return;
    }
    if (xfer->status != LIBUSB_TRANSFER_CANCELLED)
        rx_xfer_status(xfer->status);
    libusb_free_transfer(xfer);
    ut->rx_xfer = NULL;
    return;
}
if (ut->stop_ubertooth)
    return;
fifo_inc_write_ptr(ut->fifo);
ut->rx_xfer->buffer = (uint8_t*)fifo_get_write_element(ut->fifo);

r = libusb_submit_transfer(ut->rx_xfer);
if (r < 0)
    fprintf(stderr, "Failed to submit USB transfer (%d)\n", r);

static void* poll_thread_main(void* arg __attribute__((unused)))
{
    int r = 0;
    while (!do_exit) {
        struct timeval tv = { 1, 0 };
        r = libusb_handle_events_timeout(NULL, &tv);
        if (r < 0) {
            do_exit = 1;
            break;
        }
        usleep(1);
    }
    return NULL;
}

int ubertooth_bulk_thread_start()
{
    do_exit = 0;
    return pthread_create(&poll_thread, NULL, poll_thread_main, NULL);
}

void ubertooth_bulk_thread_stop()
{
    do_exit = 1;
    pthread_join(poll_thread, NULL);
}

int ubertooth_bulk_init(ubertooth_t* ut)
{
    int r;
    ut->rx_xfer = libusb_alloc_transfer(0);
    libusb_fill_bulk_transfer(ut->rx_xfer, ut->devh, DATA_IN, 
        (uint8_t*)fifo_get_write_element(ut->fifo), PKTLEN, 
        cb_xfer, ut, TIMEOUT);
    r = libusb_submit_transfer(ut->rx_xfer);
    if (r < 0) {
        fprintf(stderr, "rx_xfer submission: %d\n", r);
        return -1;
    }
    return 0;
}

void ubertooth_bulk_wait(ubertooth_t* ut)
{
    while (fifo_empty(ut->fifo) && !ut->stop_ubertooth)
usleep(1);

int ubertooth_bulk_receive(ubertooth_t* ut, rx_callback cb, void* cb_args)
{
    if (!fifo_empty(ut->fifo)) {
        (*cb)(ut, cb_args);
        if (ut->stop_ubertooth) {
            if (ut->rx_xfer)
                libusb_cancel_transfer(ut->rx_xfer);
            return 1;
        }
    } else {
        usleep(1);
        return -1;
    }
}

static int stream_rx_usb(ubertooth_t* ut, rx_callback cb, void* cb_args)
{
    /* init USB transfer */
    int r = ubertooth_bulk_init(ut);
    if (r < 0)
        return r;
    r = ubertooth_bulk_thread_start();
    if (r < 0)
        return r;
    /* tell ubertooth to send packets */
    r = cmd_rx_syms(ut->devh);
    if (r < 0)
        return r;
    /* receive and process each packet */
    while (!ut->stop_ubertooth) {
        ubertooth_bulk_wait(ut);
        r = ubertooth_bulk_receive(ut, cb, cb_args);
    }
    ubertooth_bulk_thread_stop();
    return 1;
}

/* file should be in full USB packet format (ubertooth-dump -f) */
int stream_rx_file(ubertooth_t* ut, FILE* fp, rx_callback cb, void* cb_args)
{
    uint8_t buf[PKT_LEN];
    size_t nitems;
    while (1) {
        uint32_t systime_be;
        ...)
nitems = fread(&systime_be, sizeof(systime_be), 1, fp);
if (nitems != 1)
    return 0;
systime = (time_t)be32toh(systime_be);

nitems = fread(buf, sizeof(buf[0]), PKT_LEN, fp);
if (nitems != PKT_LEN)
    return 0;
fifo_push((ut->fifo, (usb_pkt_rx*)buf);
(*cb)(ut, cb_args);
}

/* Receive and process packets. For now, returning from
* stream_rx_usb() means that UAP and clocks have been found, and that
* hopping should be started. A more flexible framework would be
* nice. */
void rx_live(ubertooth_t* ut, btbb_piconet* pn, int timeout)
{
    int r = btbb_init(max_ac_errors);
    if (r < 0)
        return;

    if (timeout)
        ubertooth_set_timeout(ut, timeout);
    if (pn != NULL && btbb_piconet_get_flag(pn, BTBB_CLK27_VALID))
        cmd_set_clock(ut->devh, 0);
    else {
        stream_rx_usb(ut, cb_br_rx, pn);
        /* Allow pending transfers to finish */
        sleep(1);
    }
    /* Used when follow_pn is preset OR set by stream_rx_usb above
    * i.e. This cannot be rolled into the above if...else
    */
    if (pn != NULL && btbb_piconet_get_flag(pn, BTBB_CLK27_VALID)) {
        ut->stop_ubertooth = 0;
        /* cmd_stop(ut->devh); */
        cmd_set_bdaddr(ut->devh, btbb_piconet_get_bdaddr(pn));
        cmd_start_hopping(ut->devh, btbb_piconet_get_clk_offset(pn), 0);
        stream_rx_usb(ut, cb_br_rx, pn);
    }
}

void rx_afh(ubertooth_t* ut, btbb_piconet* pn, int timeout)
{
    int r = btbb_init(max_ac_errors);
    if (r < 0)
        return;

    cmd_set_channel(ut->devh, 9999);
    if (timeout) {
        ubertooth_set_timeout(ut, timeout);
        cmd_afh(ut->devh);
    }
stream_rx_usb(ut, cb_afh_initial, pn);

cmd_stop(ut->devh);

ut->stop_ubertooth = 0;

btbb_print_afh_map(pn);
}

/* Monitor changes in AFH channel map */
cmd_clear_afh_map(ut->devh);
cmd_afh(ut->devh);
stream_rx_usb(ut, cb_afh_monitor, pn);
}

void rx_afh_r(ubertooth_t * ut, btbb_piconet* pn, 
    int timeout __attribute__((unused)))
{
    static uint32_t lasttime;

    int r = btbb_init(max_ac_errors);
    int i, j;
    if (r < 0)
        return;

    cmd_set_channel(ut->devh, 9999);
    cmd_afh(ut->devh);

    /* init USB transfer */
r = ubertooth_bulk_init(ut);
    if (r < 0)
        return;

    r = ubertooth_bulk_thread_start();
    if (r < 0)
        return;

    /* tell ubertooth to send packets */
r = cmd_rx_sym(ut->devh);
    if (r < 0)
        return;

    /* receive and process each packet */
while (!ut->stop_ubertooth) {
    ubertooth_bulk_receive(ut, cb_afh_r, pn);
    if (lasttime < time(NULL)) {
        lasttime = time(NULL);
        printf("%u ", (uint32_t)time(NULL));
        /* btbb_print_afh_map(pn); */

        uint8_t* afh_map = btbb_piconet_get_afh_map(pn);
        for (i=0; i<10; i++)
            for (j=0; j<8; j++)
                if (afh_map[i] & (1<<j))
                    printf("1");
            else
printf("0");
printf("\n");
ubertooth_bulk_thread_stop();

/* sniff one target LAP until the UAP is determined */
void rx_file(FILE* fp, btbb_piconet* pn)
{
    int r = btbb_init(max_ac_errors);
    if (r < 0)
        return;
    ubertooth_t* ut = ubertooth_init();
    if (ut == NULL)
        return;
    stream_rx_file(ut, fp, cb_br_rx, pn);
}
void rx_btle_file(FILE* fp)
{
    ubertooth_t* ut = ubertooth_init();
    if (ut == NULL)
        return;
    stream_rx_file(ut, fp, cb_btle, NULL);
}
void ubertooth_unpack_symbols(const uint8_t* buf, char* unpacked)
{
    int i, j;
    for (i = 0; i < SYM_LEN; i++) {
        /* output one byte for each received symbol (0x00 or 0x01) */
        for (j = 0; j < 8; j++) {
            unpacked[i * 8 + j] = ((buf[i] << j) & 0x80) >> 7;
        }
    }
}
static void cb_dump_bitstream(const ubertooth_t* ut, void* args _attribute__((unused)))
{
    int i;
    char nl = '\n';
    usb_pkt_rx usb = fifo_pop(ut->fifo);
    usb_pkt_rx* rx = &usb;
    char bitstream[BANK_LEN];
    ubertooth_unpack_symbols((uint8_t*)rx->data, bitstream);
    /* convert to ascii */
    for (i = 0; i < BANK_LEN; ++i)
        bitstream[i] += 0x30;
fprintf(stderr, "rx block timestamp %u * 100 nanoseconds\n",
rx->clk100ns);
if (dumpfile == NULL) {
    fwrite(bitstream, sizeof(uint8_t), BANKLEN, stdout);
    fwrite(&nl, sizeof(uint8_t), 1, stdout);
} else {
    fwrite(bitstream, sizeof(uint8_t), BANKLEN, dumpfile);
    fwrite(&nl, sizeof(uint8_t), 1, dumpfile);
}
}

static void cb_dump_full(
    ubertooth_t* ut, void* args
attribute__((unused)))
{
    usb_pkt_rx usb = fifo_pop(ut->fifo);
    usb_pkt_rx* rx = &usb;
    fprintf(stderr, "rx block timestamp %u * 100 nanoseconds\n", rx->clk100ns);
    uint32_t time_be = htobe32((uint32_t)time(NULL));
    if (dumpfile == NULL) {
        fwrite(&time_be, 1, sizeof(time_be), stdout);
        fwrite((uint8_t*)rx, sizeof(uint8_t), PKT_LEN, stdout);
    } else {
        fwrite(&time_be, 1, sizeof(time_be), dumpfile);
        fwrite((uint8_t*)rx, sizeof(uint8_t), PKT_LEN, dumpfile);
        fflush(dumpfile);
    }
}

/* dump received symbols to stdout */
void rx_dump(ubertooth_t* ut, int bitstream)
{
    if (bitstream)
        stream_rx_usb(ut, cb_dump_bitstream, NULL);
    else
        stream_rx_usb(ut, cb_dump_full, NULL);
}

void ubertooth_stop(ubertooth_t* ut)
{
    /* make sure xfers are not active */
    if (ut->rx_xfer != NULL)
        libusb_cancel_transfer(ut->rx_xfer);
    if (ut->devh != NULL) {
        cmd_stop(ut->devh);
        libusb_release_interface(ut->devh, 0);
    }
    libusb_close(ut->devh);
    libusb_exit(NULL);
    if (ut->h_pcap_bredr) {
        btbb_pcap_close(ut->h_pcap_bredr);
        ut->h_pcap_bredr = NULL;
    }
    if (ut->h_pcap_le) {
        /* dump extracted symbols to stdout */
        cb_dump_full(ut, NULL);
    }
}
l ell_pcap_close (ut->h_pcap_le);
    ut->h_pcap_le = NULL;
  }

if (ut->h_pcapng_bredr) {
    btbb_pcapng_close (ut->h_pcapng_bredr);
    ut->h_pcapng_bredr = NULL;
}
if (ut->h_pcapng_le) {
    l ell_pcapng_close (ut->h_pcapng_le);
    ut->h_pcapng_le = NULL;
}
}

ubertooth_t * ubertooth_init ()
{
    ubertooth_t * ut = (ubertooth_t *) malloc (sizeof (ubertooth_t));
    if (ut == NULL) {
        fprintf (stderr, "Unable to allocate memory \n");
        return NULL;
    }
    ut->fifo = fifo_init ();
    if (ut->fifo == NULL)
        fprintf (stderr, "Unable to initialize ringbuffer \n");
    ut->devh = NULL;
    ut->rx_xfer = NULL;
    ut->stop_ubertooth = 0;
    ut->abs_start_ns = 0;
    ut->start_clk100ns = 0;
    ut->last_clk100ns = 0;
    ut->clk100ns_upper = 0;
    ut->h_pcap_bredr = NULL;
    ut->h_pcap_le = NULL;
    ut->h_pcapng_bredr = NULL;
    ut->h_pcapng_le = NULL;
    return ut;
}

int ubertooth_connect (ubertooth_t * ut, int ubertooth_device)
{
    int r = libusb_init (NULL);
    if (r < 0) {
        fprintf (stderr, "libusb_init failed (got 1.0?) \n");
        return -1;
    }
    ut->devh = find_ubertooth_device (ubertooth_device);
    if (ut->devh == NULL) {
        fprintf (stderr, "could not open Ubertooth device \n");
        ubertooth_stop (ut);
        return -1;
    }
    r = libusb_claim_interface (ut->devh, 0);
}
if (r < 0) {
    fprintf(stderr, "usb_claim_interface error %d\n", r);
    ubertooth_stop(ut);
    return -1;
}

return 1;

ubertooth_t* ubertooth_start(int ubertooth_device) {
    ubertooth_t* ut = ubertooth_init();
    int r = ubertooth_connect(ut, ubertooth_device);
    if (r < 0)
        return NULL;
    return ut;
}

int ubertooth_check_api(ubertooth_t *ut) {
    int r;
    r = cmd_api_version(ut->devh);
    if (r < 0) {
        fprintf(stderr, "Ubertooth running very old firmware found.\n");
        fprintf(stderr, "Please upgrade to latest released firmware.\n");
        ubertooth_stop(ut);
        return -1;
    } else if (r < UBERTOOTH_APIVERSION) {
        fprintf(stderr, "Ubertooth API version %d found, libubertooth requires %d.\n", r, UBERTOOTH_APIVERSION);
        fprintf(stderr, "Please upgrade to latest released firmware.\n");
        ubertooth_stop(ut);
        return -1;
    } else if (r > UBERTOOTH_APIVERSION) {
        fprintf(stderr, "Ubertooth API version %d found, newer than that \n supported by libubertooth (%d).\n", r, UBERTOOTH_APIVERSION);
        fprintf(stderr, "Things will still work, but you might want to \ update your host tools.\n");
    }
    return 0;
}

// Return all connected Ubertooth devices */
int ubertoothEnumerate() {
    return 0;
}
1.9   host/libubertooth/src/ubertooth.h

/* Copyright 2016 Air Force Institute of Technology, U.S. Air Force
 * Copyright 2010 – 2013 Michael Ossmann, Dominic Spill,
 * Will Code, Mike Ryan
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 * You should have received a copy of the GNU General Public License
 * along with this program; see the file COPYING. If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110-1301, USA.
 */

#ifndef _UBERTOOTH_H_
#define _UBERTOOTH_H_

#include "ubertooth_control.h"
#include "ubertooth_fifo.h"
/*#include <btbb.h> */
#include <btbb.h>

/* specan output types 
 * see https://github.com/dkogan/feedgnuplot for plotter */
enum specan_modes {
    SPECAN_STDOUT         = 0,
    SPECAN_GNUPLOT_NORMAL = 1,
    SPECAN_GNUPLOT_3D     = 2,
    SPECAN_FILE           = 3
};

enum board_ids {
    BOARD_ID_UBERTOOTH_ZERO = 0,
    BOARD_ID_UBERTOOTH_ONE  = 1,
    BOARD_ID_TC13BADGE      = 2
};

typedef struct {
    /* Ringbuffers for USB and Bluetooth symbols */
    fifo_t* fifo;

    struct libusb_device_handle* devh;
    struct libusb_transfer* rx_xfer;
    uint8_t stop_ubertooth;
    uint64_t abs_start_ns;
    uint32_t start_clk100ns;
    uint64_t last_clk100ns;

    /*
     * ... (more definitions and implementations)
     */

    /*
     * ... (more comments and explanations)
     */

    /*
     * ... (more code and logic)
     */

    /*
     * ... (more comments and explanations)
     */

    /*
     * ... (more code and logic)
     */
}

179
typedef void (*rx_callback)(ubertooth_t* ut, void* args);

typedef struct {
    unsigned allowed_access_address_errors;
    } btle_options;

extern uint32_t systime;
extern FILE* infile;
extern FILE* dumpfile;
extern int max_ac_errors;

void print_version();
void register_cleanup_handler(ubertooth_t* ut, int do_exit);
ubertooth_t* ubertooth_init();
int ubertooth Enumerate();
int ubertooth_connect(ubertooth_t* ut, int ubertooth_device);
ubertooth_t* ubertooth_start(int ubertooth_device);
void ubertooth_stop(ubertooth_t* ut);
int ubertooth_check_api(ubertooth_t* ut);
void ubertooth_set_timeout(ubertooth_t* ut, int seconds);
int ubertooth_bulk_init(ubertooth_t* ut);
void ubertooth bulk wait(ubertooth_t* ut);
int ubertooth bulk receive(ubertooth_t*, rx_callback, void*);
int ubertooth bulk thread_start();
void ubertooth bulk thread_stop();
int stream rx file(ubertooth_t* ut, FILE*, rx_callback, void*);
void rx alive(ubertooth_t* ut, btbp piconet* pn, int timeout);
void rx file(FILE* fp, btbp piconet* pn);
void rx dump(ubertooth_t* ut, int full);
void rx btle(ubertooth_t* ut);
void rx btle file(FILE* fp);
void rx afh (ubertooth_t* ut, btbp piconet* pn, int timeout);
void rx afh r (ubertooth_t* ut, btbp piconet* pn, int timeout);
void ubertooth unpack_symbols(const uint8_t* buf, char* unpacked);

#elif defined(UBERTOOTH_H) */
#include "ubertooth.h"
#include "ubertooth-callback.h"
#include <ctype.h>
#include <err.h>
#include <getopt.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

int convert_mac_address(char *s, uint8_t *o) {
    int i;

    /* validate length */
    if (strlen(s) != 6 * 2 + 5) {
        printf("Error: MAC address is wrong length\n");
        return 0;
    }

    /* validate hex chars and : separators */
    for (i = 0; i < 6*3; i += 3) {
        if (!isxdigit(s[i]) || !isxdigit(s[i+1])) {
            printf("Error: MAC address contains invalid character(s)\n");
            return 0;
        }

        if (i < 5*3 && s[i+2] != ':') {
            printf("Error: MAC address contains invalid character(s)\n");
            return 0;
        }
    }

    /* sanity: checked; convert */
    for (i = 0; i < 6; ++i) {
        unsigned byte;
        sscanf(&s[i*3], "%02x", &byte);
```c
static void usage(void)
{
    printf("ubertooth-btle – passive Bluetooth Low Energy monitoring\n");
    printf("Usage:\n");
    printf("\t-h this help\n");
    printf("\n");
    printf("\t-M Major modes:\n");
    printf("\t-f follow single LE connection\n");
    printf("\t-m follow multiple LE connections\n");
    printf("\t-p promiscuous: sniff active connections\n");
    printf("\t-a [address] get/set access address \n"  
        (example: -a8e89bed6)\n");
    printf("\t-s [address] faux slave mode, using MAC addr \n"  
        (example: -s22:44:66:88:aa:cc)\n");
    printf("\t-t [address] set connection following target \n"  
        (example: -t22:44:66:88:aa:cc)\n");
    printf("\n");
    printf("\t-I Interference (use with -f or -p)\n");
    printf("\t-i interfere with one connection and return to idle\n");
    printf("\t-I interfere continuously\n");
    printf("\n");
    printf("\t-D Data source:\n");
    printf("\t-U<0-7> set ubertooth device to use\n");
    printf("\n");
    printf("\t-M Misc:\n");
    printf("\t-r [filename] capture packets to PCAPNG file\n");
    printf("\t-q [filename] capture packets to PCAP file \n"  
        (DLT_BLUETOOTH_LE_LL_WITH_HDR)\n");
    printf("\t-c [filename] capture packets to PCAP file (DLT_PPI)\n");
    printf("\t-A <index> advertising channel index (default 37)\n");
    printf("\t-v [01] verify CRC mode, get status or enable/disable\n");
    printf("\t-x <index> allow n access address offenses (default 32)\n");
    printf("\nIf an input file is not specified, an Ubertooth device \n"  
        is used for live capture.\n");
    printf("In get/set mode no capture occurs.\n");
}

int main(int argc, char *argv[])
{
    int opt;
    int do_follow, do_follow_multi, do_promisc;
    int do_set_aa, do_set_iaa;
    int do_crc;
    int do_adv_index;
    int do_slave_mode;
    int do_target;
    enum jam_modes jam_mode = JAM_NONE;
    char ubertooth_device = -1;
    ubertooth_t *ut = ubertooth_init();
```

btle_options cb_opts = { .allowed_access_address_errors = 32 };

int r;
uint8_t mac_address[6] = { 0, };
dofollow = do_follow_multi = do_promisc = 0;
do_get_aa = do_set_aa = 0;
do_crc = -1; /* 0 and 1 mean set, 2 means get */
do_adv_index = 37;
do_slave_mode = do_target = 0;
while ((opt = getopt(argc, argv, "a::r:hfmpU:v::A:s:t:x:c:q:jJiI")) != \
EOF) {
    switch(opt) {
    /* User wants a specific access address */
    case 'a':
        if (optarg == NULL) {
            do_get_aa = 1;
        } else {
            do_set_aa = 1;
            sscanf(optarg, "%08x", &access_address);
        }
        break;
    /* User wants to follow connection */
    case 'f':
        do_follow = 1;
        break;
    /* User wants to follow connection */
    case 'm':
        do_follow_multi = 1;
        break;
    /* User wants to operate in promiscuous mode */
    case 'p':
        do_promisc = 1;
        break;
    /* User wants to provide a specific Ubertooth interface */
    case 'U':
        ubertooth_device = atoi(optarg);
        break;
    /* User wants to capture files to PCAPNG */
    case 'r':
        if (!ut->h_pcapng_le) {
            if (lelenium_pcapng_create_file("\n                  optarg, "Ubertooth", &ut->h_pcapng_le)) {
                err(1, "lelenium_pcapng_create_file: ");
            }
        } else {
            printf("Ignoring extra capture file: %s\n", optarg);
        }
    }
break;

/* User wants to capture files to PCAP */
case 'q':
    if (!ut->h_pcap_le) {
        if (lpell_pcap_create_file(optarg, &ut->h_pcap_le)) {
            err(1, "lpell_pcap_create_file: ");
        }
    }
    else {
        printf("Ignoring extra capture file: %s\n", optarg);
    }
    break;

/* User wants to capture files to pcap ppi */
case 'c':
    if (!ut->h_pcap_le) {
        if (lpell_pcap_ppi_create_file(optarg, 0, &ut->h_pcap_le)) {
            err(1, "lpell_pcap_ppi_create_file: ");
        }
    }
    else {
        printf("Ignoring extra capture file: %s\n", optarg);
    }
    break;

/* User wants to ensure CRC is verified */
case 'v':
    if (optarg)
        do_crc = atoi(optarg) ? 1 : 0;
    else
        do_crc = 2; /* get */
    break;

/* User wants to listen on a specific advertisement channel */
case 'A':
    do_adv_index = atoi(optarg);
    if (do_adv_index < 37 || do_adv_index > 39) {
        printf("Error: advertising index must be 37, 38, or 39\n");
        usage();
        return 1;
    }
    break;

/* User wants to operate as a slave */
case 's':
    do_slave_mode = 1;
    r = convert_mac_address(optarg, mac_address);
    if (!r) {
        usage();
        return 1;
    }
    break;

/* User wants to follow a specific target */
case 't':
    do_target = 1;
    r = convert_mac_address(optarg, mac_address);
if (!r) {
    usage();
    return 1;
}  
break;

/* How many access address errors should we tolerate */
case 'x':
    cb_opts.allowed_access_address_errors = (unsigned) atoi(optarg);
    if (cb_opts.allowed_access_address_errors > 32) {
        printf("Error: can tolerate 0–32 access address bit errors\n") ;
        usage();
        return 1;
    }
    break;

/* Jamming stuff */
case 'i':
case 'j':
    jam_mode = JAM_ONCE;
    break;
case 'I':
case 'J':
    jam_mode = JAM_CONTINUOUS;
    break;
case 'h':
default:
    usage();
    return 1;
}  

/* Connect to the Ubertooth One */
r = ubertooth_connect(ut, ubertooth_device);
if (r < 0) {
    usage();
    return 1;
}

/* Check the firmware and tell the user if they need to update */
r = ubertooth_check_api(ut);
if (r < 0)
    return 1;

/* Clean up on exit. */
register_cleanup_handler(ut, 1);

if (do_target) {
r = cmd_btle_set_target(ut->devh, mac_address);
    if (r == 0) {
        int i;
        printf("target set to: ");
        for (i = 0; i < 5; ++i)
            printf("%02x: ", mac_address[i]);
        printf("%02x\n", mac_address[5]);
    }
/* Check if the user asked for more than one option */
if (!(do_follow ^ do_follow_multi ^ do_promisc)) {
    printf("Error: must choose only one of -f, -m, -p\n");
    return 1;
}
if (do_follow || do_follow_multi || do_promisc) {
    usb_pkt_rx rx;
    r = cmd_set_jam_mode(ut->devh, jam_mode);
    if ((jam_mode != JAMNONE && r != 0) {
        printf("Jamming not supported\n");
        return 1;
    }
    cmd_set_modulation(ut->devh, MOD_BT_LOW_ENERGY);
    if (do_follow || do_follow_multi) {
        u16 channel;
        if (do_adv_index == 37)
            channel = 2402;
        else if (do_adv_index == 38)
            channel = 2426;
        else
            channel = 2480;
        cmd_set_channel(ut->devh, channel);
        printf("About to send the 'follow' command via USB\n");
        /* Follow */
        if (do_follow) cmd_btle_sniffing(ut->devh, 2);
        else if (do_follow_multi) cmd_btle_multi_sniffing(ut->devh, 2);
    } else {
        /* Promiscuous */
        cmd_btle_promisc(ut->devh);
    }
    while (1) {
        int r = cmd_poll(ut->devh, &rx);
        if (r < 0) {
            printf("USB error\n");
            break;
        }
        if (r == sizeof(usb_pkt_rx)) {
            fifo_push(ut->fifo, &rx);
            cb_btle(ut, &cb_opts);
        } else
            usleep(500);
    }
    ubertooth_stop(ut);
}
if (do_get_aa) {
    access_address = cmd_get_access_address(ut->devh);
    printf("Access address: %08x\n", access_address);
}
return 0;

if (do_set_aa) {
    cmd_set_access_address(ut->devh, access_address);
    printf("access address set to: %08x\n", access_address);
}

if (do_crc >= 0) {
    int r;
    if (do_crc == 2) {
        r = cmd_get_crc_verify(ut->devh);
    } else {
        cmd_set_crc_verify(ut->devh, do_crc);
        r = do_crc;
    }
    printf("CRC: %s verify\n", r ? "" : "DO NOT ");
}

if (do_slave_mode) {
    u16 channel;
    if (do_adv_index == 37)
        channel = 2402;
    else if (do_adv_index == 38)
        channel = 2426;
    else
        channel = 2480;
    cmd_set_channel(ut->devh, channel);
    cmd_btle_slave(ut->devh, mac_address);
}

if (!(do_follow || do_promisc || do_get_aa || do_set_aa ||
    do_crc >= 0 || do_slave_mode || do_target || do_follow_multi ))
    usage();

return 0;


190


36. Nordic Semiconductor, nRF Sniffer,
(www.nordicsemi.com/eng/Products/Bluetooth-low-energy/nRF-Sniffer), 2014.

37. Nordic Semiconductor, nRF51 Development Kit,
(www.nordicsemi.com/eng/Products/nRF51-DK), 2014.

38. Nordic Semiconductor, nRF51822 Product Specification v3.1,


40. Onset Computer Corporation, HOBOMobile User’s Guide for iOS,


43. Pwnie Express, Blue Hydra, (github.com/pwnieexpress/blue_hydra), 2016.


53. Texas Instruments, SmartRF Protocol Packet Sniffer,

54. Transducers Direct, Data Sheet for CirrusSense TDWLB Series Wireless
    Bluetooth Pressure Transducer,
    2013.

55. G. Valadon and P. Lalet, Scapy: the python-based interactive packet
Bluetooth Low Energy (BLE) is a wireless communications protocol used in Critical Infrastructure (CI) applications. Based on recent research trends, it is likely that the next generation of wireless sensor networks, a CI application that the Department of Defense (DoD) regularly employs in surveillance and reconnaissance missions, will include BLE as an inter-sensor communications protocol. Thus, future U.S. military missions may be directly impacted by the security of BLE. One natural way to help protect BLE sensors is to use BLE traffic sniffers to detect attacks. The primary limitation with current sniffers is that they can only capture one connection at a time, making them impractical for applications employing multiple BLE devices. This work aims to overcome that limitation to help secure the types of BLE sensor networks employed by the DoD. First, this work identifies vulnerabilities and enumerates attack vectors against a BLE wireless industrial sensor, presenting a list of security “best practices” that vendors and end-users can follow and demonstrating how users can employ BLE sniffers to detect attacks. The work then introduces BLE-Multi, an enhancement to an open-source BLE sniffer that can simultaneously and reliably capture multiple connections. Finally, the work presents and executes a methodology to evaluate BLE sniffers. Under the evaluation conditions applied, BLE-Multi achieves simultaneous capture of multiple active connections, paving the way for automated defensive tools that can be used by the DoD and security community. The contributions within are published in one journal article and one conference paper and were presented at three conferences focused on wireless security and CI protection.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:
a. REPORT
b. ABSTRACT
c. THIS PAGE
U
U
U

17. LIMITATION OF ABSTRACT
U

18. NUMBER OF PAGES
207

19. a. NAME OF RESPONSIBLE PERSON
Maj Jason M. Bindewald, AFIT/ENG

19b. TELEPHONE NUMBER *(include area code)*
(937) 255-3636, x4614; jason.bindewald@afit.edu