NTNU

Norwegian University of
Science and Technology

Concurrent operation of Bluetooth low energy and ANT wireless protocols with an embedded controller

Per Magnus Østhus

Master of Science in Electronics
Submission date: June 2011
Supervisor: Per Gunnar Kjeldsberg, IET
Co-supervisor: Rune Brandsegg, Nordic Semiconductor ASA

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

Concurrent operation of Bluetooth low energy and ANT wireless protocols with an embedded controller

In this assignment the student shall analyze how the wireless protocols Bluetooth Low Energy (BLE) and ANT can co-exist. A scheduler and a control protocol that can handle the two radio protocols at the same time shall be designed and implemented. Both ANT and BLE are low duty-cycle protocols that should be able to co-exist, but it will be a challenge to fulfill the timing requirements of the two protocols and co-ordinate the activity on the two protocols.

The scheduler and the control protocol shall be prototyped in a microcontroller. The prototype system will consist of a three chip solution with one BLE radio device, one ANT radio device and a micro-controller.

The BLE and ANT communication devices are pre-programmed at the link layer level. The micro-controller chip will use SPI to control the two communication chips. The microcontroller should support profiles for both protocols. The task is divided in four steps:

Step 1: Study the Bluetooth Low Energy protocol and the ANT protocol and analyze how they can co-exist. Define the principles of a scheduler and a control protocol.

Step 2: Implement a simple dual-protocol solution where the microcontroller is first "listening" for both ANT and BLE activity. When activity on one of the protocols is detected, create a connection on that protocol and start communication. This is not true concurrency, just an ability to switch between BLE mode and ANT mode.

Step 3: Have the microcontroller communicate simultaneously with both ANT and BLE protocols where the communication is divided in time to prevent interference on the air.

Step 4: If time, create an example application with profile support for both BLE and ANT.

Nordic Semiconductor will mentor the assignment, and provide necessary development kits for Bluetooth Low Energy, ANT and the controller. The PCB for the three-chip prototype solution is already available.

Assignment given:	17. January 2011
Supervisors:	Per Gunnar Kjeldsberg, NTNU
	Rune Brandsegg, Nordic Semiconductor

Preface

This thesis concludes my Master Degree in Electronics Engineering at the Norwegian University of Science and Technology (NTNU). Although not directly related to my specialization, which is digital design, design of embedded software for a microcontroller is a challenging and fun task. It is also quite smart to know how software engineers think, when one is designing hardware for them. Working with wireless devices has also been a new, interesting and fun experience. I have never been bored while working with this thesis.

I would like to thank my tutor at NTNU, Per Gunnar Kjeldsberg, for excellent feedback and support for the academic part of the thesis. My tutor at Nordic Semiconductor, Rune Brandsegg, has helped me with everything from debugging, soldering of jumper wires, switching on my office computer when I wanted to work from home, and pointing me to the correct people to talk to when I had too hard questions. I would also like to thank Torstein Heggebø for reading through the thesis and providing feedback. I would guess I have talked to the entire software team at Nordic Semiconductor, so a thanks goes also to them.

Trondheim, June 20, 2011
Per Magnus Østhus

Abstract

With the introduction of low-power wireless technologies, new applications in the healthcare, fitness and home entertainment markets emerge through the use of ultra low-power sensors. These devices are designed to run for years on a single coin-cell battery.

ANT and Bluetooth Low Energy are two low-power protocols that emerge as competitors in this market. The ability to combine these in a single system not only takes away the element of choice from the manufacturers, but also provides compatibility between the two protocols. An ANT-enabled device can be coupled to a Bluetooth network, with the benefit of connecting to non-ANT central devices, such as smartphones, tablets and laptops.

In this thesis, the co-existence of these two protocols is discussed. An implementation with two distinct radios for each protocol, controlled by a single embedded microcontroller, is presented. The implementation is tested with regards to packet loss with a simple test application. Test results show that the obtained packet loss cannot be correlated to the co-existence of the two protocols.

Contents

Acronyms

ACI Application Controller Interface. 20–22, 35, 37

AFH Adaptive Frequency Hopping. 25, 26

API Application Programming Interface. 2, 34, 37, 49, 56

ATT Attribute Profile. 20, 21

BLE Bluetooth Low Energy. 1, 4, 5, 17, 18, 21–23, 25, 30, 31, 33–35, 39, 41, 48, 49, 52, 55–59

Bluetooth BR/EDR Bluetooth Basic Rate/Enhanced Data Rate. 5, 17, 18, 25

CDMA Code Division Multiple Access. 3

FHSS Frequency-Hopping Spread Spectrum. 18, 25–27, 30, 31

GAP Generic Access Profile. 20

GATT Generic Attribute Profile. 20–22

GFSK Gaussian Frequency Shift Keying. 5, 8, 18

GPIO General-Purpose Input/Output. 35, 40, 57

HAL Hardware Abstraction Layer. 35

IDE Integrated Development Environment. 35

IEEE Institute of Electrical and Electronics Engineers. 4, 17, 25

ISM Industrial, Scientific and Medical. 2, 7, 8, 27

ISO International Standardization Organization. 3

ISR Interrupt Service Routine. 37, 56

L2CAP Logical Link Control and Adaptation Protocol. 21

MAC Medium Access Layer. 25, 55

MCU Microcontroller Unit. 7, 8, 22, 23, 35, 40, 56, 57

OSI Open Systems Interconnection. 3, 4

PHY Physical OSI layer. 55

RF Radio Frequency. 7, 34

RX Receiver. 8, 11

SDK Software Development Kit. 35–37, 56

SIG Bluetooth Special Interest Group. 17

SPI Serial Peripheral Interface bus. 7, 14, 22, 35

TDMA Time Division Multiple Access. 5, 8, 9

TX Transmitter. 11

UART Universal Asynchronous Receiver/Transmitter. 7, 14, 35, 40

USART Universal Synchronous/Asynchronous Receiver/Transmitter. 7, 14

UUID Universally Unique ID. 20

WLAN Wireless Local Area Network. 3, 4, 25

WPAN Wireless Personal Area Network. 25–27

WSN Wireless Sensor Network. 4, 7, 8

List of Figures

List of Tables

1 Introduction

Ultra low-power wireless technologies introduce new possibilities, only limited by imagination. The industry focus especially on applications in the healthcare, fitness and home entertaining markets. For example, information about a patient's blood sugar can be sent online to his/her doctor, taking away the need for diabetes patients to go so often to the hospital, as well as reducing cost for the hospitals. A jogger's pulse can be sent to his/her mobile phone and be combined with a GPS track to be shared on social media. A home's temperature can automatically be adjusted according to the presence of its inhabitants. Other applications include complex remote controls for home entertainment devices as well as industrial applications requiring sensors. All these applications can last for years and years on a single coin cell battery.

Today, proprietary systems are most used for these applications. Proprietary protocols have the benefit of simplicity: the manufacturer can ignore anything that he does not consider essential. On the other side are large standardization organizations, which use years to develop a new standard, often resulting in lower performance than what is possible with the used technology. However, products conforming to standards can communicate regardless of the product's manufacturer, which means standard based products often have higher volumes.

ANT, developed by Dynastream Innovations Inc., is a proprietary protocol, used in many applications in the sports and fitness markets today. The ANT+ Alliance, consisting of over 100 leading manufacturers of sports and fitness equipment, provide profile support for often used applications. This leads to compatible devices produced by different manufacturers.

With the introduction of Bluetooth Low Energy (BLE), ANT gets severe competition due to the fact that BLE can easily be incorporated into regular Bluetooth chips. Bluetooth is the de-facto standard for Wireless Personal Area Networks (WPANs) in use today [16], and enjoys a respected and well-known brand name. This integration will provide an easy way of connecting BLE to larger central devices.

This thesis will explore the feasibility of co-existence between these two protocols, with the ultimate goal being a dual-protocol chip. This will provide manufacturers with support both for widely used ANT devices in the fitness and healthcare market, as well as the emerging Bluetooth Low Energy devices, which will most probably be the winning technology for smartphones, laptops and such.

The structure of the thesis is as follows: the rest of this introduction presents some background information necessary to understand the thesis. Then the details of the ANT and BLE protocols are discussed in chapter 2 and 3. Next follows a co-existence discussion in chapter 4. Chapter 5 will describe a dual-protocol implementation, while chapter 6 describes a simple example application used for

testing purposes. Some thoughts about a single-chip implementation is presented in chapter 7. In the end, the test results and implementation are discussed in chapter 8, and a conclusion is presented in chapter 9.

The main contributions of this work are:

- A fully working dual-protocol Application Programming Interface (API) for ANT and BLE.
- Thorough testing of issues relating to co-existence.
- A suggestion for setup parameters for ANT and BLE to ensure best possible performance.

1.1 The ISM bands

The Industrial, Scientific and Medical (ISM) bands are those parts of the radio frequency spectrum commonly referred to as the unlicensed bands. These frequencies were originally intended for industrial, scientific and medical applications which often requires powerful emissions. In 1985, they were made part of the United States Federal Communications Commission (FCC) Part 15 rules, which governs license-free devices [11]. Even though a license-free device does not need a permission to operate as a radio station - with all the approvals and fees that requires - its usage is not unregulated.

In general, FCC Part 15 users must conform to the following cardinal rules [11]:

- The user has no vested right to continue using any frequency.
- The device must accept any interference generated by all other users, including other unlicensed users.
- The device may not cause harmful interference.
- The user must cease operation if notified by FCC that the device is causing harmful interference
- The equipment must be authorized to show compliance with FCC standards before marketing/importation of device

The driving factor for these rules was the development of spread spectrum technology before, during, and after World War II. Spread spectrum technologies were very interesting for the US military, due to its resistance to interference and jamming. However, as noted by Michael J. Marcus in [21], military spread spectrum was fundamentally different than civil systems. They needed to be a lot more complex, robust to jamming, and difficult to detect, leading to very expensive equipment that civil users could not afford. Additionally, the strong resistance to jamming needed design details that had not been published in the open literature,

and were unlikely to be. Civil applications are not so concerned with security, and the industry saw some interesting civil applications for spread spectrum radios, such as Code Division Multiple Access (CDMA) for cellular telephones.

In 1985, the FCC decided to open the 902-928 MHz, 2400-2483.5 MHz and 5725-5850 MHz ISM bands for unlicensed spread spectrum usage, subject to conditions restricting the maximum peak output power to 1 Watt. What was most interesting with the Part 15 rules was what they did not contain: they did not limit the use of this unlicensed spectrum to any specific class of use or users. Part 15 devices, hence, 'may emit radio frequency energy without first obtaining a station or user authorization, but they are granted no protective rights' [11].

The consequence of these rules has been the development of WiFi, Bluetooth, cordless phones, and a range of other devices. Since nobody has exclusive rights to the spectrum, development of new devices is competition-driven, that is, developers are exploiting the technology available, instead of big companies squeezing whatever profit they can get out of an old technology. No end-user license also leads to cheap equipment. The frequencies selected were exactly right for the technology, leading to high data rates.

1.2 Design of Communication Protocols

In the early days of computer networking, network protocols were largely vendor-specific and proprietary, leading to non-interoperability between equipment. The universal need for interconnecting systems from different manufacturers quickly became apparent, leading the International Standardization Organization (ISO) to form a subcommittee called Open Systems Interconnection (OSI) [25].

OSI's work lead to a reference model for development of computer networks. It promoted the idea of a consistent model of protocol *layers*, dividing the total problem into smaller pieces. Seven layers were proposed, where the highest layer is closest to the end user, and interfaces software that implements a communicating component. The lowest layer is the hardware that actually performs the communication, for example an Ethernet network card. Independence is ensured between layers by defining services provided by a layer to the next higher layer, independent of how these services are performed. For example, a laptop may be connected to the Internet both via a cable and a Wireless Local Area Network (WLAN) interface. An Internet browser does not care which interface provides the connection: its functionality remains the same. This leads to a high grade of reusability of software components, and interoperability between equipment produced by different vendors. The layers are summarized in table 1.

The OSI reference model has become a major subject for computer students, and has shaped the development of communication protocols ever since. However, the use of seven layers was considered too complicated by some computer scientists, and

Table 1: The OSI layers. Description from [2]

#	Name	Function
1	Physical layer	Interface between a device and the transmission medium
2	Data link layer	Functional and procedural means to transfer data between network entities
3	Network layer	Performs network routing functions
4	Transport layer	Controls the reliability of a link through flow control, segmentation/desegmentation, and error control
5	Session layer	Establishes, manages, and terminates connections between the local and remote application
6	Presentation layer	Formats and encrypts data sent across a network
7	Application layer	Interacts with software that implement a communicating component

in many cases unimplementable. The TCP/IP protocol, for example, was designed with four layers: application, transport, internet, and link layer. Nonetheless, the OSI model remains a good abstraction of the design of communication protocols.

1.3 Terminology

Throughout the thesis, some terms will be used that require an explanation. These include:

- **Host microcontroller**. Both the ANT and BLE radios used in this project require a host microcontroller to control it. This will be referred to simply as the *host*.
- **Channel period/connection interval**. These terms will be used interchangeably, as they have the same meaning. In the ANT documentation, the term channel period is used, while connection interval is used for BLE. In the results section (chapter 6.2), the term *connection frequency* will be used. This is simply the inverse of the connection interval.
- **IEEE 802.11/WiFi**. This is a standard from the Institute of Electrical and Electronics Engineers (IEEE), describing a WLAN. 802.11 comes in several variations, such as a, b, g, and n. The term *WiFi* will also be used to describe an 802.11 network.
- **IEEE 802.15.4**. This is another standard from the IEEE, describing Wireless Sensor Networks (WSNs).

- **Gaussian Frequency Shift Keying (GFSK)**. This is a digital modulation scheme, where a positive deviation from the carrier frequency represents a one, and a negative deviation represents a zero. A Gaussian filter is applied to smooth these frequency deviations. Both ANT and BLE use GFSK modulation.
- **Time Division Multiple Access (TDMA)**. TDMA is a communication scheme were data transfer is divided in time to allow several networks to exist simultaneously.
- **Bluetooth Basic Rate/Enhanced Data Rate (Bluetooth BR/EDR)**. Bluetooth BR/EDR will be used to describe the standard Bluetooth protocol.

2 The ANT wireless protocol

The following chapter describes only the parts of the ANT wireless protocol that is considered essential for understanding the thesis. More details about the protocol is freely available for registered users at [1].

ANT is a WSN designed to run using low-cost, low-power Microcontroller Units (MCUs) and transceivers operating in the 2.4 GHz ISM band [4]. ANT is designed for simplicity, for low-power and low datarate applications, with a compact protocol stack. 2.4 GHz is good because a single product design can be shipped to a global customer base without modification (unlicensed band all over the world).

Before ANT, existing Radio Frequency (RF) protocols based on industry standards required a significant investment of time and technical resources to be adapted to ultra-low power applications. Bluetooth and ZigBee are compromised by additions made to the protocol in order to satisfy the wide application needs of all interested parties. The net result is a large protocol overhead, lower efficiency, increased power consumption and increased cost [4].

ANT provides the physical, network and transport OSI layers, fully integrated on a single silicon device. This device is a low-power radio, provided by leading manufacturers such as Nordic Semiconductor and Texas Instruments. ANT ships this device with the lower-level parts of the protocol stack programmed, which means the user has to implement the Application layer on a separate microcontroller, called the *host*. The interface between the two devices can be synchronous or asynchronous serial interfaces, such as Universal Asynchronous Receiver/Transmitter (UART), Universal Synchronous/Asynchronous Receiver/Transmitter (USART) and Serial Peripheral Interface bus (SPI). ANT puts minimal requirements on the host MCU: it can be as low as a 1kB flash device. Figure 1 depicts the complete ANT node, with a host MCU and the ANT device.

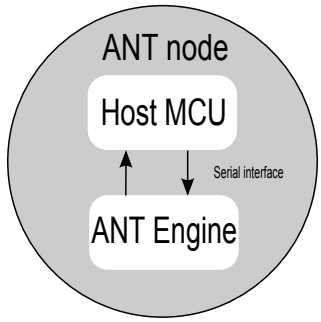

Figure 1: An ANT node consists of a host MCU and an ANT engine, communicating via a serial interface.

ANT also provides a set of profiles for regularly used applications in the ANT+

Alliance. ANT+ is a set of interoperability functions that can be added to the base ANT protocol. By using ANT+, manufacturers can be sure that their applications are compatible with other leading suppliers.

For sensors in a WSN, a host MCU may be seen as an unnecessary component adding to both area and power consumption. For these applications, ANT provides SensRCore®. An ANT module equipped with SensRCore® does not need a host MCU. Profiles for different sensor types exist, leading to short development time, low power consumption and low costs for a sensor.

2.1 Radio

The ANT radio uses the 2.4 GHz ISM band with GFSK modulation. It splits the band into 125 unique channels, each 1 MHz wide, providing a 1 Mbps RF link. This is good for low duty-cycle applications, since each radio only has to transmit for a very short period (less than $150\mu s$ per message according to [4]). The low duty cycle further allows each channel to be divided into timeslots in a TDMA fashion, allowing several networks to co-exist on the same channel.

Figure 2 describes the radio timing. The actual data transfer occurs in the light-grey area, while the Receiver (RX) window is used for synchronization with other masters, and data transfer in the reverse direction. By keeping the RX window a bit larger than the transmit window, the slave can adjust its timing relative to the master. For example, if a clock drift occurs between the two nodes, the slave can adjust its RX window to stay synchronized. The master transmits at a regular interval, called the channel period.

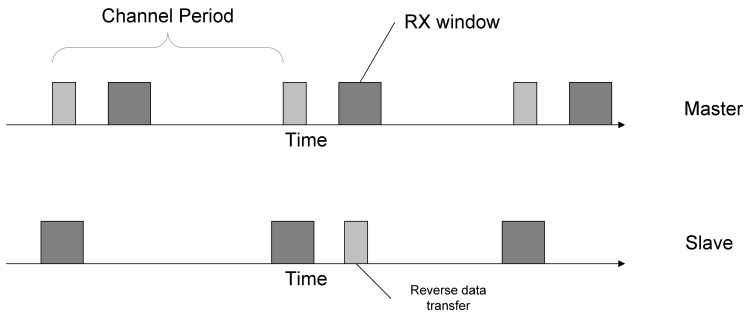

Figure 2: ANT radio timing. The RX window is used for synchronization with other masters, as well as data transfer from slave to master.

Before the two radios are properly synchronized, the slave searches for the master by turning on its radio with a 10% duty cycle [9]. This ensures a low-power, but still effective, channel search. The slave and master must share a common channel configuration, that consists of the used radio frequency as well as a unique channel

ID, which is transmitted along with user data. The channel configuration will be discussed later.

The transmit time is noted in [4] to be less than $150ms$ per message. The details of what is actually transmitted on air is not given in the ANT documentation. However, the data payload can be up to 8 bytes per packet [6].

2.2 Topology

An ANT network may have many networking topologies, such as peer-to-peer, star, tree and other types of mesh networks. It is optimized for peer-to-peer, star and tree [4], since a mesh network may be too complicated. An ANT network has a capability of up to 65 536 slave nodes listening to one master over a single channel, thanks to the before mentioned TDMA scheme.

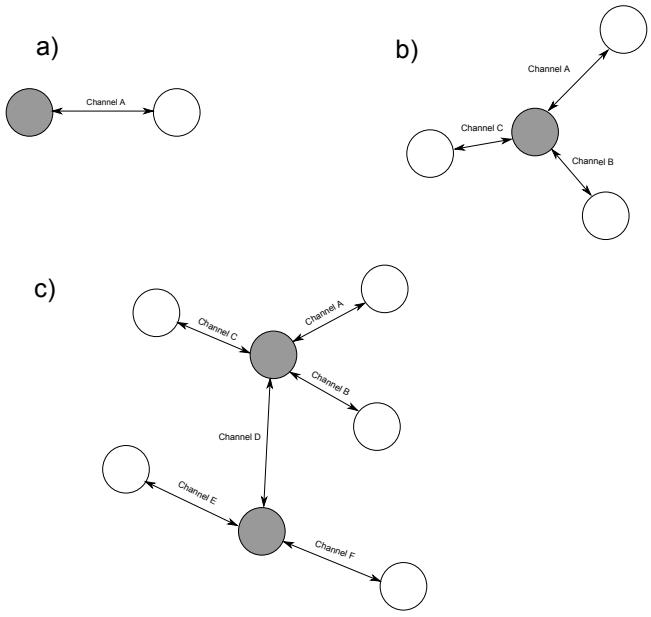

Figure 3: Different ANT network topologies. a) Peer-to-peer, b) Star, c) Tree Gray nodes are master nodes.

A single ANT node can have up to 8 different channels, and act both as a slave and a master on different channels. That means an ANT node may connect different networks together, i.e., act as a network hub.

2.3 Protocol stack

The ANT protocol stack handles the physical, network, and transport OSI layers [7]. These are the layers provided on the ANT chip. The higher OSI layers has to be implemented on a separate microcontroller called the host. The interface between the host and the ANT module is a simple, bidirectional serial message protocol, described in detail in [6].

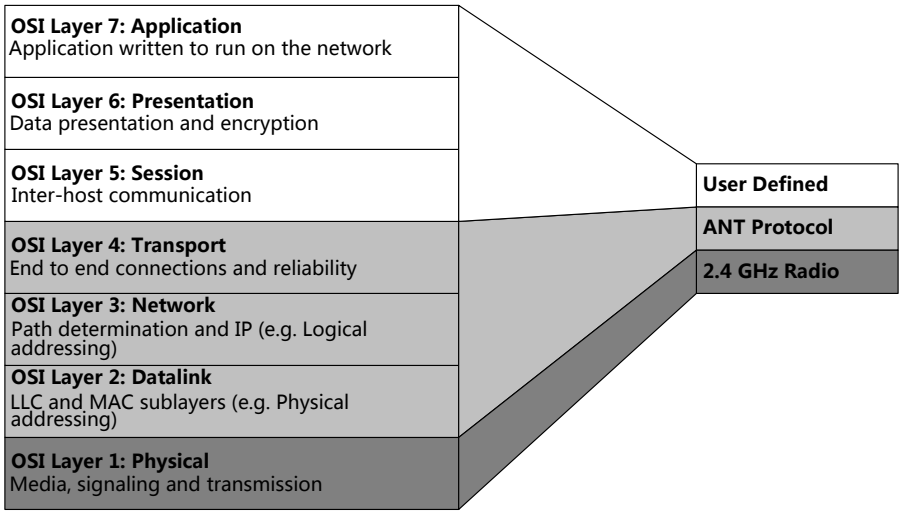

Figure 4: The OSI layers on the left, with the corresponding ANT layers on the right. The user of an ANT module only has to implement the top three layers, the rest is done on the ANT chip.

The ANT protocol (layers 2-4) is made as autonomous as possible. For example, in a master device, data is transmitted at every channel period without interaction from the higher layers. However, an *event* is sent to the host across the serial message protocol to identify that data has been sent. The host may use this event to set the next data that is to be transmitted. If the host does not send new data, the same data is retransmitted at the next channel period.

The before mentioned ANT+ interoperability functions operate at the user defined layers. ANT+ is merely a set of channel configurations, which can be acquired at Dynastream.

2.4 ANT Channels

ANT usage and configuration is channel based. Each channel connects two nodes together. However, a single master node can broadcast messages to several slave nodes through one channel, thereby connecting more than two nodes together.

A channel may be either unidirectional or bidirectional. The primary data flow is from the master to the slave, with three different message types: broadcast, acknowledged and burst messages. Broadcast messages can use a unidirectional channel, while acknowledged and burst messages require a bidirectional channel.

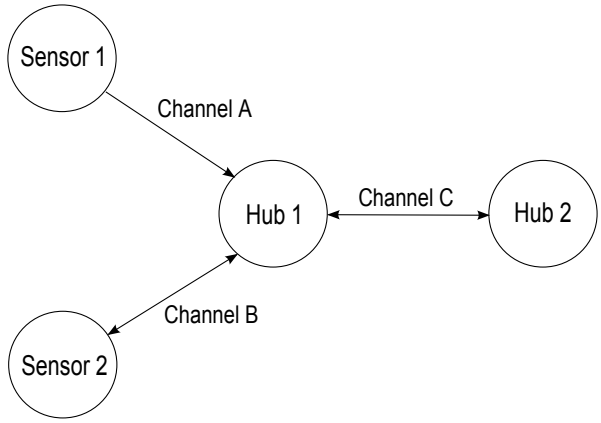

Figure 5: ANT channel types. Channel A is unidirectional, while channels B and C are bidirectional. Hub 1 acts as a slave for channels A and B, and is the master of channel C. Image courtesy of ANT [6]

For two ANT nodes to communicate, they require a common channel configuration. This configuration includes information related to the operating parameters of a channel, such as:

- Channel type (Transmitter (TX) only, RX only, unidirectional)
- RF frequency
- Channel ID
  - Transmission type
  - Device type (class or type of the master device)
  - Device number (unique number representing the master device)
- Channel period
- Network number

Most of these parameters may be changed while the channel is open. However, changes must be applied to both the master and the slave: there is no notification between the devices if a channel configuration changes. If, for example, the master decides to change its RF frequency, the slave will not change its own frequency automatically, and will have to be instructed to do so by its host.

2.4.1 RF frequency

The RF frequency is represented in the channel configuration as an 8-bit field, with acceptable values between 0 and 124. It represents an offset in 1 MHz increments from 2400 MHz, with the maximum frequency being 2524 MHz. The following equation can be used to determine the value for the RF frequency field:

$$RF\_freq\_val = \frac{Desired\_freq\_in\_MHz - 2400}{1MHz} \tag{1}$$

The operating RF frequency must be selected such that it complies with international standard frequencies. In Norway, for example, an unlicensed 2.4 GHz radio may not operate outside 2.4 - 2.4835 GHz [19]. Therefore, an ANT channel cannot use the frequencies outside this band. The ANT network may also be operated in an environment with other 2.4 GHz devices. If this is known beforehand, the frequency may be selected to avoid the other devices to ensure good performance.

2.4.2 Channel period

The channel period[1] represents the message rate of the master device. By default, a data packet will be sent and received, on every timeslot at this rate. The channel rate can range from 0.5 Hz to above 200 Hz, with the upper limit dependant on the specific implementation. The channel period is a 16-bit number determined by the following equation:

$$Channel\_period\_value = \frac{32768}{Message\_rate\_in\_Hz} \tag{2}$$

If, for example, a message rate of 4 Hz is wanted, the channel period value must be set to $32768/4 = 8192$.

A slave node may subscribe to a subset of the transmitted messages by setting the channel period to an integer fraction of the master period. For example, a slave can receive every fourth message by setting its period to 1 Hz when the master transmits at 4 Hz.

2.4.3 Channel ID

The most basic descriptor of a channel is the channel ID. It consists of 4 bytes that contains 3 fields: a transmission type, a device type and a device number. The

[1] The terms 'channel period' and 'connection interval' will be used interchangeably throughout the thesis

transmission type is used to define certain transmission characteristics of a device. The device type is used to differentiate between different classes of devices, i.e., heart rate monitors, bike speed sensors etc. In this way, the network participants are aware of the various classes of connected nodes and can decode the received data accordingly. The device number is a unique 16-bit field meant to be unique for each device in a device type. It can for example be correlated to a serial number, or it can be a random number. The channel ID must be specified on the master side, with values other than 0. The slave can search for a unique master, or it can set wildcard values (0) on the channel ID to search for any master on that frequency.

2.5 Frequency agility

An ANT channel normally use a single RF frequency throughout its existence. However, some ANT devices incorporate a technique called frequency agility, which is able to change the operating frequency if performance is degraded on the current frequency, for example due to interference from other 2.4 GHz devices. Not all devices incorporate this technique, but it can also be implemented in software on the host microcontroller.

The technique is described in an application note from ANT [5]. When configuring the channel, three frequencies are specified as possible channel frequencies, instead of just one. These frequencies must be specified both on the master and slave side. When link performance is degraded, the ANT node starts to use the next frequency in this list.

Central to the technique is the ability to track link performance. The slave node can easily do this as it regularly receives RF transmissions from the master. If a specific number of consecutive messages are missed, the slave will drop back into the search state at a different frequency. This number is based on the channel period, and the algorithm for deciding it for the slave node is presented below.

Algorithm 1 Algorithm for deciding the number of consecutive missed messages before switching frequency, at the slave node. T represents the channel period value, so $T > 29789$ means that the channel period is lower than $1.1Hz$. C is the number of consecutive failures before the radio returns to search mode at the next frequency in the frequency agility list.

if $T > 29789$ **then**
  $C = 4$
else
  $C = (65536/T) + 1$
end if

A master device requires its messages to be acknowledged in order to detect whether the slave successfully received the message. This, however, introduces a new

source for failure, as the slave may successfully have received the message but the acknowledgment was disturbed. Thus, the master uses a different technique to track link performance. It keeps count of consecutive missed messages by using a counter that is incremented on each successfully received acknowledgment, and decremented on each failed message. Once the counter is equal to 0 and a fixed number of consecutive missed messages are detected, the link is judged as poor and the device will switch to a different frequency.

The default frequency agility settings are 2.403, 2.439 and 2.475 GHz. These are selected as each one is sufficiently far away from two of the three most common 802.11 channels (1, 6 and 11). Figure 6 describes this setup. The 802.11 (WiFi) channels are 22 MHz wide, with center frequencies noted on the X-axis of the figure. The three ANT frequency agility channels are in blue, with their respective frequencies noted.

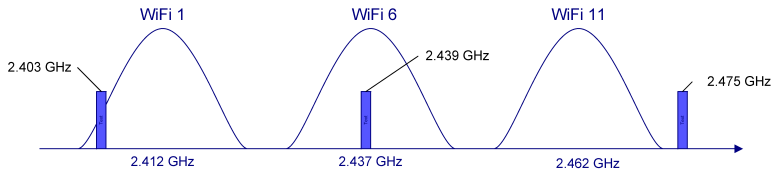

Figure 6: Default ANT frequency agility channels with the most common 802.11 channels.

2.6 Interfacing an ANT device

An ANT module can be interfaced using either a synchronous or asynchronous serial interface such as UART, USART and SPI. The interface is dependant on the manufacturer of the module; ANT does not produce its own modules, but has developed devices in co-operation with Nordic Semiconductor and other semiconductor manufacturers. The host can be any device with such serial interfaces, such as an embedded microcontroller or a personal computer.

The message protocol used is the same for all interfaces [6]. It is a simple bidirectional protocol with short messages, consisting of a synchronization byte used to determine the start of a message, a message length field, a message ID, the data bytes, and a checksum byte. The checksum is the bitwise exclusive-or (XOR) of all previous bytes (see figure 7).

The messages can be divided into five main classes:

- Configuration messages
- Control messages
- Data messages

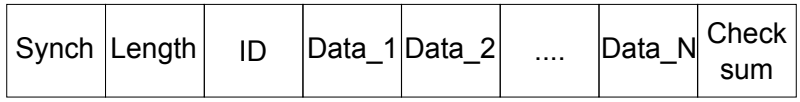

Figure 7: The structure of the ANT message protocol. N is the maximum data size. The checksum is the bitwise XOR of all previous bytes.

- Channel response/event messages
- Requested response messages

Configuration messages are used to setup the device. Control messages are used to open and close a channel and perform a soft reset. Data messages are used to send data over a channel. The channel response/event messages are sent from the ANT module to the host either in response to a configuration- or control message, or as generated by an RF event on the ANT device. The requested response messages are responses such as channel status, channel ID, version, capabilities and serial number of the device.

3 Bluetooth Low Energy

As with the ANT chapter, this chapter will only describe what is considered essential of the Bluetooth Low Energy protocol. The protocol is fully described in the Bluetooth Core Specification [10], but Bluetooth BR/EDR and Low Energy are mixed together in this document, and the specification is therefore difficult to read. Nick Hunn has described the recently introduced protocol in a clearer way in his book *Essentials of Short Range Wireless* [16].

The development of BLE started in Nokia, with a project called Wibree [16, p. 176]. Researchers at Nokia had determined that there were various scenarios contemporary wireless technologies did not address, such as the inclusion of low-power sensors in fitness devices, sports equipment, watches, and ID tags in mobile phone applications. They therefore began to develop a new technology adapted from the Bluetooth standard which would provide lower power usage and prize, while minimizing the difference from Bluetooth BR/EDR. Nokia proposed the new technology as one of the alternatives in the early stages of the IEEE 802.15.4 standard. Even though it was not selected at the time, development progressed when the technology was transferred to the Bluetooth Special Interest Group (SIG) in 2007. In 2009, the SIG announced BLE as the new exciting feature of the Bluetooth Core Specification Version 4.0. The technology is quite new, with the first BLE chips arriving in late 2010 and early 2011 from Nordic Semiconductor, Cambridge Silicon Radio (CSR), and Texas Instruments.

By basing the new technology on Bluetooth, BLE can easily be incorporated into a combined Bluetooth BR/EDR and BLE chip, saving space on already space-constrained devices such as smart phones. However, on the peripheral devices such as sensors, strict power considerations makes BLE the only standard desired. Therefore, new Bluetooth chips will either be single-mode (only BLE) or dual-mode (support for both protocols), introducing an asymmetry in the design. BLE makes the assumption that the receiving device has considerably more resources than the transmitter, and there is no need for devices to operate both as a master and a slave, although this is possible.

The close relationship with Bluetooth has another major advantage: billions of mobile phones and PCs have already embraced Bluetooth, and will probably continue to have it in the future. By combining the new standard with Bluetooth in a dual-mode chip, an infrastructure of billions of devices will quickly become available. This gives BLE a 'free-ride that will lead to economies of scale for chip vendors, and a vibrant ecosystem of devices for products to connect to' [16, p. 176].

Even though they are closely related, standard Bluetooth and BLE are not compatible. BLE features a new protocol stack, meaning it can never communicate directly with a standard Bluetooth chip. The main difference between the two is that BLE devices are only concerned with transferring the device *state*, rather than

streaming data or files. The maximum packet size in BLE is 47 bytes, and when protocol overhead is removed, only 20 bytes are left for the data payload.

3.1 Radio

The BLE radio is necessarily constrained by the requirement that it can be implemented using the same RF chain already present in a standard Bluetooth chip. It uses the 2.4 GHz ISM band, with GFSK modulation, as is present in Bluetooth BR/EDR. It divides the spectrum into 40 channels, each 2 MHz wide[2], and incorporates Frequency-Hopping Spread Spectrum (FHSS) to avoid interference. Three of these channels are used for advertising, initiating connections and broadcasting, while the 37 others are used for data transfer while in a connection. The over-the-air datarate is 1 Mbps.

The advertising channels (37, 38 and 39) are placed at 2.402, 2.426 and 2.480 GHz, respectively. These frequencies are chosen to avoid the areas of the spectrum mostly used by 802.11, in a similar way as the default ANT frequency agility settings are chosen. The data transfer channels (0 - 36) are placed in between the advertising channels (2.404 - 2.424 GHz, 2.428 - 2.478 GHz).

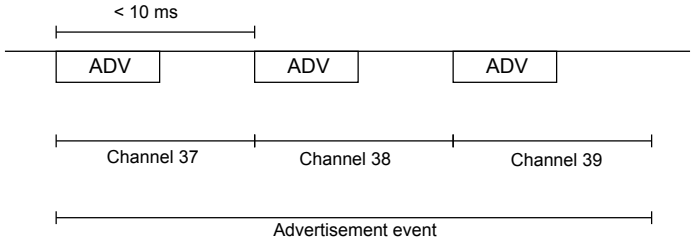

Figure 8: BLE advertiser timing. An advertisement event consists of three advertisements at channels 37, 38, and 39, with a random interval of maximum 10 ms between each transmit. In this interval, the device listens for a connection request or a request for more data. Figure from [16]

Figure 8 depicts a BLE advertisement event. An individual event consists of three transmits at channels 37, 38, and 39. The interval between each transmit is a random time up to 10 ms, to aid in interference avoidance [10]. In this interval, the device listens for a connection request or a request for more data. The time between each full advertisement event can be configured to a value above 20 ms[3]

When another device wants to initiate a connection with the advertiser, it sends a connection request (CONN_REQ) packet, 150μs after the advertisement packet. The advertiser immediately stops its advertising event, and uses the information

[2]compared to standard Bluetooth's 79 1 MHz wide channels

[3]This value is actually dependent on the *type* of the advertisement [16].

in the CONN_REQ packet to jump to the requested data channel to continue the connection sequence. The two devices will then exchange information, and the master device can configure the behavior of the slave [16].

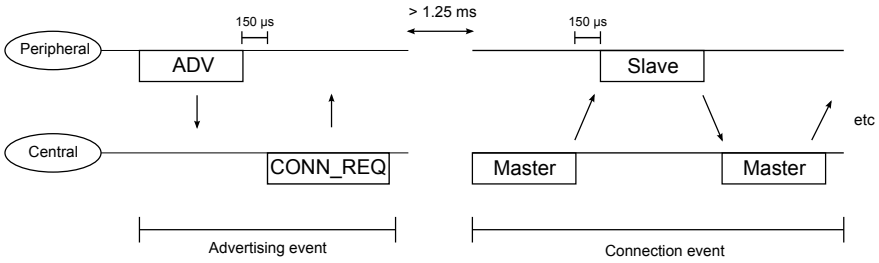

Figure 9: BLE connection procedure. After a CONN_REQ packet, the master and slave waits for a minimum of 1.25 ms before continuing at a data channel. From [16].

Figure 9 depicts the connection procedure. The CONN_REQ packet tells the slave how often the master will initiate connection events (the *connection interval*), and how many of these a slave is allowed to ignore before it must wake up for one (the *slave latency*). The connection interval must be a value between 7.5 and 4000 ms.

An individual connection event consists of several transfers between master and slave, depending on the application. The size of the packets may also vary, with a maximum possible packet size of 47 bytes.

3.2 Topology

The first release of the Bluetooth Low Energy protocol only supports piconets [16, p. 180]. A piconet is a simple network containing several slaves connected to a single master. BLE slaves only support having a single master, while standard Bluetooth allows slaves to be connected to many masters. Future releases of the specification are likely to extend the topology with switch and relay functionality to enable the construction of star networks.

In most of the envisaged use cases, the topology is based on devices that push information by broadcasting, advertising, or by notifications. Where connections are established, devices normally sleep for most of their lives, waking up at pre-agreed times to exchange information. The simplest devices, which only broadcast or receive information, may consist of just a transmitter or receiver, without the need for both [16].

This leads to a concept of *profile roles*. This is not to be confused with *application profiles*, which is a higher-level concept used to standardize different applications, i.e., a heart rate monitor. Profile roles are rather a description of the basic

functionality a device has when communicating with other BLE devices [16, p. 181]. The different profile roles are:

- Broadcaster
- Observer
- Central device
- Peripheral device

The broadcaster and observer roles can be grouped together as unidirectional devices, since they only require only a transmitter or receiver, not both. The broadcaster devices send advertising packets, containing data, which can be heard by any receiving device. The data are identified using a Universally Unique ID (UUID), which defines its format and type. The corresponding observer devices listen for these advertisements, and can filter the data according to a pre-installed application.

The central- and peripheral devices are bidirectional devices, containing both a receiver and a transmitter (transceivers). After a connection is made, peripheral devices will take the role of a slave, whilst a central device will become the master. Unlike standard Bluetooth, BLE does not support the concept of role reversal; after a device has become a slave, it will remain a slave for the duration of its connection. This allows a major asymmetry in the complexity of peripheral and central devices, as a peripheral device can be designed so that it will always take the role of a slave, leading to much simpler and cheaper slave devices. It is, however, possible for a device to support being both a master and a slave at the same time.

3.3 Protocol stack

Bluetooth Low Energy is designed to transmit the *state* of devices, rather than streaming data or files. This leads to a much simpler protocol stack than for standard Bluetooth. Also, standard Bluetooth allows multiple alternative transport protocols, such as TCP (telephony control protocol), RFCOMM (serial port emulation) and AVDTP (audio/video distribution transport protocol). BLE allows only the transport of *attributes* through the Attribute Profile (ATT).

Figure 10 illustrates Nordic Semiconductor's implementation of the BLE stack. Application developers access BLE functionality through the Application Controller Interface (ACI), which is Nordic's implementation of an interface between the BLE device and the host. The Generic Access Profile (GAP) layer handles discovery and connections, while the Generic Attribute Profile (GATT) is used to configure devices and transfer data. Individual advertisements and RF activity is handled autonomously in the link- and physical layers.

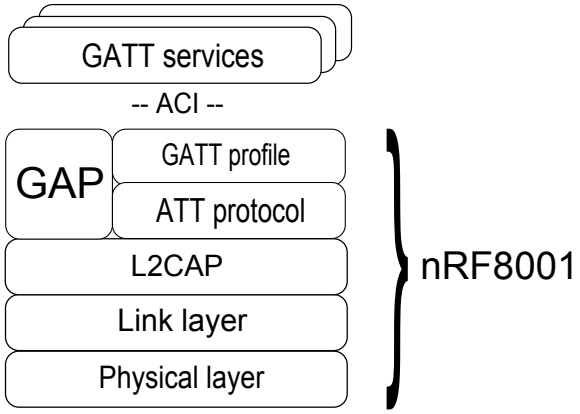

Figure 10: Nordic Semiconductor's implementation of the BLE stack. The BLE device (nRF8001) implements the lower layers, while a host MCU must implement the GATT services. The interface between the host and BLE device is called the Application Controller Interface (ACI).

GATT and ATT are the cornerstones of the BLE architecture. These define the concepts of attributes, characteristics, and services. An *attribute* is essentially a piece of data, which may be a sensor measurement, a setting of an actuator, or other data. It is uniquely defined by a 16 or 128-bit UUID, which defines what the attribute represents. A *characteristic* adds behaviour to the attribute, defining how the information will be used, i.e whether it is read-only, needs reliable write etc. A *service* is a container of attributes that represents a typical application. Examples of services are battery status, thermometer and proximity alert.

The Logical Link Control and Adaptation Protocol (L2CAP) layer passes packets to and from the link layer. It is a common interface layer for protocol stacks that support both standard Bluetooth and BLE, becoming the 'meeting point' of the two protocols. In standard Bluetooth, the L2CAP handles multiplexing of data between transport protocols, segmentation and reassembly of packets, and quality of service management. In BLE, the L2CAP only handles segmentation and reassembly of packets.

Before a connection can be established, the BLE devices will need to configure the GATT and ATT as either a *client* or *server*. A peripheral is typically configured as the attribute server, as it usually contains the state information (for example, a temperature sensor). After a connection has been made, the client searches the server for available services, and decides which services it wishes to interact with.

3.4 Frequency hopping

As discussed in the radio chapter, BLE uses frequency hopping to combat interference: each advertisement event is spread over three frequencies, and each connection event happens at a different frequency than the previous.

The hopping pattern of connection events is determined by a *channel map*, which is a 5-byte field describing which channels are used. The hopping increment is by default 5 channels.

The channel map and hop increment is set by the master device, and can be changed during a connection.

3.5 Interfacing a BLE device

The interface between a BLE device and its host MCU is not standardized, and therefore only Nordic Semiconductor's implementation of this is discussed.

As seen in figure 10, the nRF8001 device implements all layers of the protocol stack, except from the GATT services, which has to be implemented on the host MCU. The ACI is the interface between the BLE device and the host. The nRF8001 uses a slightly modified Serial Peripheral Interface bus (SPI), with two special signals for handshaking: *ready* and *request*. These hand-shake signals allow nRF8001 to notify the host processor when it has received new data over the air, and also to hold new data exchanges initiated by the host until it is ready to accept and process them [22].

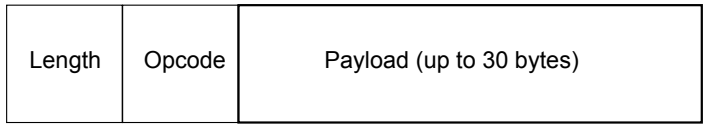

Figure 11: ACI packet structure. The payload length is dependent of the opcode of the message.

The packet structure of the ACI is described in figure 11. The packet header consists of two bytes: the packet length and an opcode. The packet payload is dependent of the opcode, and can be up to 30 bytes. Four types of messages are used:

- System command
- System event
- Data command
- Data event

System commands are used to control the nRF8001 configuration, operation mode and behavior, Data commands are used when application data exchange between the nRF8001 and a peer device is required. Events are messages sent from the nRF8001 to the host, either in response to a command, or as an indication of an event occurring on the BLE stack. For example, if data is received from a peer device, a data event is sent to the host.

Another signal, *radio_active*, is provided by the nRF8001 to indicate that the radio is active. This can be configured to occur up to $20ms$ before the radio is switched on, and can be used to save power by limiting the activity in the host processor while the radio is on. If the connection interval is less than $30ms$, the active signal will automatically be disabled.

Other manufacturers of BLE devices may have other interfaces between the host and the device, both physically and logically. It is even possible to implement the entire stack on a system-on-chip, which makes a host MCU unnecessary.

4 Co-existence analysis

This chapter will focus on analyzing how ANT and BLE can co-exist. First, some techniques used in co-existence is presented, along with a summary of similar projects. Next follows a discussion of the *probability of collision*, that is, the probability that two radios will transmit simultaneously on the same frequency. Then, this probability is derived for the case of ANT and BLE, before a hypothesis about the co-existence performance of ANT and BLE is presented.

4.1 Co-existence techniques

The IEEE 802.15 Working Group for Wireless Personal Area Networks (WPANs) have been addressing co-existence issues since the formation of its task group 2. The work of this group resulted in some recommended practices, a co-existence model, and a set of mechanisms to facilitate co-existence of WPANs and WLANs [3]. The mechanisms described can be divided into two classes: *collaborative* and *non-collaborative*. Collaborative techniques require some sort of communication between the devices, so that they can co-operate on sharing the spectrum. Non-collaborative techniques require no such communication.

Table 2: Examples of co-existence mechanisms proposed by IEEE 802.15 TG2.

Collaborative	**Non-collaborative**
Alternating media access	Adaptive interference suppression
Packet traffic arbitrator	Packet scheduling
Deterministic interference suppression	Adaptive frequency hopping

C. Karlsson summarized the methods mentioned in table 2 in his master thesis [18]. In general, the collaborative techniques co-operate at the Medium Access Layer (MAC) layer. That is, if one device is using its radio (medium), the other is prohibited from using its own. On the other hand, the non-collaborative techniques has to adapt a device to the environment, without co-operation with the other device.

Adaptive Frequency Hopping (AFH) is a non-collaborative technique that can be adopted in a Frequency-Hopping Spread Spectrum (FHSS) system to avoid using bad channels. This technique is of special interest as it has been adopted in Bluetooth BR/EDR version 1.2 [10]. The technique can be described as follows: if a transmission fails on a given frequency, this frequency is labeled as 'bad' in a table. At the next sequence, this frequency is not used. Over time, this will result in a better environment, both for the interferer and the system employing AFH. However, AFH will use some time to adapt to the environment, so in a quickly changing environment AFH can actually result in degraded performance. Thus,

AFH is best suited for an environment where a FHSS system exists next to a static system such as 802.11b.

Other researches have proposed strategies for reducing interference between users of the ISM bands. The most occurring techniques are non-collaborative variants of AFH ([13], [12], [23]) that seeks to dynamically change the channel hop sequence of FHSS systems, such that bad channels are avoided. A major drawback of AFH is that it takes some time to adopt to the environment, making it not so useful for quickly changing environments. Also, AFH increases the memory requirements and power consumption of a device [12], as it needs to keep track of 'good' and 'bad' channels, and compute a new hopping sequence based on these. However, in more or less static environments, AFH performs better than using only FHSS.

4.2 Previous work

In a study by Howitt et al., an analytical model for evaluating the impact of WPAN nodes on 802.11b performance [14, 15] is presented. Central to their approach is to group the nodes in an evenly distributed network, and place a 802.11b station in the midst of this network. An 802.11b access point is placed at a specific distance from the network. A probability of collision is then derived, where a collision is defined as 'the event where one or more WPAN signals corrupt an 802.11b packet, such that retransmission of the packet is required'. This probability is derived from the number of interferers in the network, and the probability of time- and frequency coincidence.

Their model is applicable to a wide range of network configurations and performance criteria, and the conclusions drawn may be very different depending on the parameters investigated. They have used their model both on Bluetooth [14] and 802.15.4 piconets [15]. For Bluetooth, they conclude that co-existence performance is dependent on traffic levels and piconet density. For 802.15.4, frequency management can be employed to virtually eliminate co-existence issues.

A practical study by A. Sikora and V. F. Groza [24] looked at the consequence of having more than one type of WPAN devices in close proximity to each other, i.e., they did not only check the impact of WPANs on 802.11 alone, but also other users of the spectrum. Sikora and Groza based their tests around the IEEE 802.15.4 specification, and checked the impact of both Bluetooth, 802.11b and a microwave oven on 802.15.4 packet loss. The result of their tests was that IEEE 802.11 stations with high duty cycles were critical to 802.15.4 performance, especially if the same carrier frequencies are selected. The impact of Bluetooth and the microwave oven was not as significant. They noted that a dynamic adaption of a frequency channel would be of major importance, but unfortunately, this is not part of the 802.15.4 standard.

In their article, Jim Landsford et al. also look at co-existence issues between

Bluetooth and 802.11b [17]. A key result from their investigation is that while performance of both systems can degrade when they are co-located, a number of techniques can be employed to virtually eliminate the problems. For example, as Bluetooth is most often used for short-range communications, reduced transmit power can be used to lessen the interference impact on WiFi. A rule change has been proposed by the FCC that would allow FHSS devices to hop through only a part of the ISM band, meaning Bluetooth and WiFi could completely avoid each others frequencies. Indeed, this rule has become effective. For example, BLE can set its channel map to avoid certain frequencies in its hopping pattern.

4.3 Probability of collision

As noted by Howitt et al. in [15], the ability to differentiate between operational conditions which will and will not result in the communication devices to meet the requirements of the application, is central to the co-existence issues between wireless devices. In their work with WPAN's and WiFi, they derived a probability of collision, $P_r(C)$, dependent on the number of interferers in a network, and time- and frequency coincidence. In this project, the ANT and BLE devices will be separated by about a centimeter. If they are both time- and frequency coincident, a collision is certain to occur. Therefore, only an analysis of time- and frequency coincidence is useful for this thesis.

4.3.1 Time coincidence

Nordic Semiconductor has been involved in the development of Bluetooth Low Energy since the Wibree days. The question of time coincidence in low duty-cycle protocols is a research area which Nordic has great expertise on. Therefore, the following discussion has been made through conversations with employees at Nordic, and especially through a de-classified internal document by a former employee [20].

During early development of Wibree, Nordic looked into the case when a single Wibree master handles more than one connection [20]. In Wibree, the connection interval has a value $n * 1250\mu s$, where n is a positive integer between 3 and 3200. Consider an example, where $n = 3$, the connection events occur at the following times: $\{3750, 7500, 11250, \ldots\}$. If a second slave should appear with the same connection interval, its connection events must be shifted relative to the first. That is, its connection events must occur at times: $\{3750 + \delta, 7500 + \delta, 11250 + \delta, \ldots\}$, where δ is chosen such that collisions are avoided.

One can consider connection events as *sets*. Let m, n_1 and n_2 be elements of Z^+ (positive integers). Define the two sets $M_1 = n_1 * m * Z^+$ and $M_2 = n_2 * m * Z^+$. Then M_1 and M_2 share the following elements: $M_1 \cup M_2 = [lcm(n_1 * m, n_2 * m)] *$

Z^+, where $lcm(\bullet)$ denotes the least common multiple. If $M_1 \cup M_2 \neq \emptyset$, the two connections will have colliding transmissions [20].

4.3.2 How to choose δ

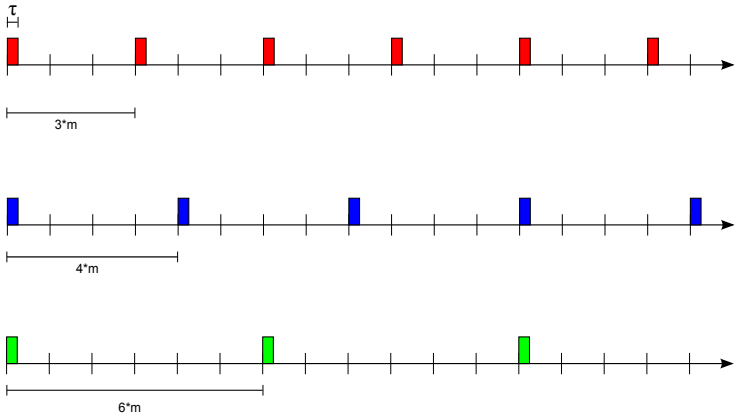

Figure 12: Three connection sets with $\delta = 0$. τ denotes the duration of a connection event. The periods are multiples of m

Consider the two top axes in figure 12. Here, $n_1 = 3$ and $n_2 = 4$. The two sets M_1 and M_2 therefore become:

$$M_1 = m * \{3, 6, 9, 12, \ldots, 24, \ldots, 36, \ldots\} \tag{3}$$
$$M_2 = m * \{4, 8, 12, \ldots, 24, \ldots, 36, \ldots\} \tag{4}$$

The shared elements (colliding transmissions) of M_1 and M_2 are:

$$\begin{aligned} M_1 \cup M_2 &= [lcm(3m, 4m)] * Z^+ \\ &= m * \{12, 24, 36, \ldots\} \\ &= 12m * Z^+ \end{aligned} \tag{5}$$

Note that an element of M_2 either overlaps with an element of M_1, or has a distance m from its 'nearest neighbor' in M_1. That is, elements of M_1 and M_2 either overlap, or has a distance m. Notice also that $gcd(3m, 4m) = 1m$, where $gcd(\bullet)$ denotes the greatest common divisor. Since the connection events each has duration $\tau > 0$, shifting either M_1 or M_2 by $\tau < \delta < m - \tau$ will prevent overlaps. Or, more generally:

$$km + \tau < \delta < (k+1)m - \tau \tag{6}$$

where k is an integer. Shifting either M_1 or M_2 by an *exact* multiple of m, results in just as many shared elements as before.

Now consider the two bottom axes of figure 12. Here, $n_2 = 4$ and $n_3 = 6$. By noting that $gcd(4m, 6m) = 2m$, one can see that shifting either M_2 or M_3 by $k * 2m + \tau < \delta < (k+1) * 2m - \tau$, collisions are avoided.

From the observations above, one can conclude that the choice of δ to avoid collisions should be in the interval:

$$k * gcd(T_1, T_2) + \tau_1 < \delta < (k+1) * gcd(T_1, T_2) - \tau_2 \tag{7}$$

In this equation, $n * m$ is replaced by T which is the connection interval, and τ_1 and τ_2 are the length of two different radios' connection events.

If δ is not in the interval given by equation 7, collisions will occur every $T_x / lcm(T_1, T_2)$ transmission of radio x.

4.3.3 Probability distribution of δ

In the above discussion, we considered a single master that handles two connections. In this project, however, two distinct radios should co-exist. If we assume two radios with no co-operation in radio timing, the offset δ between the radios can be modeled as a uniform distribution between 0 and $min(T_1, T_2)$, where T_1 and T_2 denotes the connection intervals. That is:

$$P_r(\delta = t) = \frac{1}{min(T_1, T_2)}, 0 < t < min(T_1, T_2) \tag{8}$$

This equation can then be integrated with respect to t, with the integral limits given by equation 7, to give an expression for the probability of a 'successful shift':

$$\begin{aligned} P_r(OK) &= \int_{k*gcd(T_1,T_2)+\tau_1}^{(k+1)gcd(T_1,T_2)-\tau_2} P_r(\delta = t)\,\mathrm{d}t \\ &= \frac{(k+1)gcd(T_1, T_2) - \tau_2 - (k * gcd(T_1, T_2) + \tau_2)}{min(T_1, T_2)} \\ &= \frac{gcd(T_1, T_2)}{min(T_1, T_2)} - \frac{\tau_1 + \tau_2}{min(T_1, T_2)} \end{aligned} \tag{9}$$

The probability of time coincidence, $P_r(TC)$, thus becomes:

$$\begin{aligned} P_r(TC) &= 1 - P_r(OK) \\ &= 1 - \frac{gcd(T_1, T_2)}{min(T_1, T_2)} + \frac{\tau_1 + \tau_2}{min(T_1, T_2)} \end{aligned} \tag{10}$$

4.3.4 Frequency coincidence

A radio using FHSS hops between frequencies at each connection event. The hopping sequence can be both deterministic and random. However, it is preferable that all channels are used equally often. This results in a probability of a FHSS radio to be at a specific frequency of:

$$P_r(f = F) = \frac{1}{N} \tag{11}$$

where N is the number of hopping frequencies. For two radios, where one employs FHSS and the other is stationary in frequency, the probability of frequency coincidence will thus be as in equation 11.

The bandwidth of each channel plays a role, as inter-channel interference will occur between two adjacent channels. A simple model of this can be based on the ratio of the bandwidths, that is, how many of the smaller channels the larger occupies. For a 2 MHz FHSS radio and a 1 MHz non-FHSS radio, the probability of frequency coincidence would be come $2/N$.

A more sophisticated model would include the out-of-band interference due to side lobes as well. Such a model is presented in [14], but for simplicity, this is not discussed in this thesis.

4.4 Concerning ANT and BLE

The above discussion about probability of collision may be suited to any radio protocols where connection events are regular in time and frequency. This section will fit the model to suit ANT and BLE.

4.4.1 Time coincidence

In equation 10, two values τ_1 and τ_2 were used to represent the duration of a radio transmit. For ANT, the message transmit time is noted to be less than $150\mu s$. It is not noted whether this includes the time the receiver is turned on, nor the RX window. However, the time is noted to be the duration of a *complete* message. Thus, $150\mu s$ will be used for τ_1.

For BLE, the duration of a connection event is dependent on the application. The master always initiates the event, with the slave responding within $150\mu s$ after the initiating packet, as noted in figure 9. The master may then request more data, depending on the application. In this discussion, we will assume that only two transfers happen: the initiating packet, and a single slave response. We will assume the maximum packet size of 47 bytes for each packet. This gives a duration of:

$$\begin{aligned} \tau_2 &= \frac{2 * 47 * 8bits}{1Mbps} + 150\mu s \\ &= 902\mu s \end{aligned} \tag{12}$$

per connection event.

If the two radios share the same connection interval, they will either collide in time at every connection event, or they will not. Inserting this condition into equation 10, with values for τ_1 and τ_2, the equation becomes:

$$\begin{aligned} P_r(TC) &= 1 - \frac{gcd(T_1, T_2)}{min(T_1, T_2)} + \frac{\tau_1 + \tau_2}{min(T_1, T_2} \\ &= 1 - 1 + \frac{\tau_1 + \tau_2}{T} \\ &= \frac{0.150 + 0.902}{T} \end{aligned} \tag{13}$$

where T is given in milliseconds.

The valid values for T in BLE is between $7.5ms$ and $4000ms$, while for ANT it can be as low as $5ms$, depending on the implementation, and go up to $2000ms$. In figure 13, this probability is plotted as a function of T. We see clearly from this figure that the probability of time coincidence decreases quickly with increased connection interval.

The case with different connection intervals is a bit more difficult to analyze. Consider the range in equation 7: here it is clear that for a valid δ to exist, the sum of τ_1 and τ_2 must be less than $gcd(T_1, T_2)$. If, for example, the two connection intervals are prime numbers, the $gcd(\bullet)$-function evaluates to 1, and a valid shift to totally avoid collisions is not possible. If this is the case, it is quite clear that the radios will collide in time every $T_x / lcm(T_1, T_2)$ transmission of radio x.

4.4.2 Frequency coincidence

BLE employs FHSS, that is, it hops between different frequencies at every connection event. The hopping pattern is chosen such that each of its 37 data

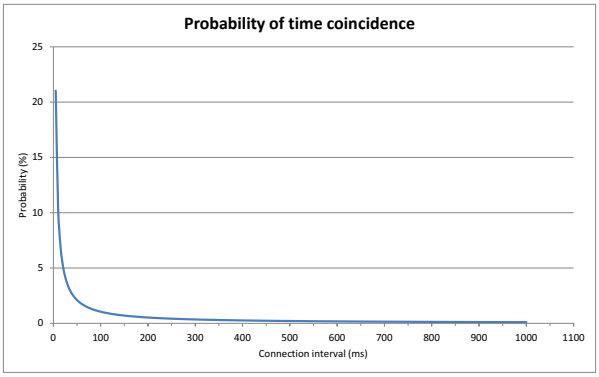

Figure 13: Probability of time coincidence, when ANT and BLE have the same connection interval.

channels are used equally often. This results in a probability of BLE to be at a specific frequency of $1/37$ (equation 11). Its channels are 2 MHz wide, with 2 MHz channel spacing, resulting in the center frequencies being at even frequencies. The ANT channels, however, are 1 MHz wide, with 1 MHz channel spacing. Figure 14 illustrates this.

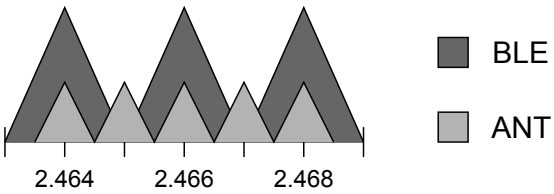

Figure 14: ANT and BLE spectrum usage. BLE uses 2 MHz wide channels with even center frequencies, while ANT has 1 MHz wide channels.

If we disregard interference due to side lobes, the probability of frequency coincidence will be:

$$\begin{aligned} P_r(FC)_{even} &= \frac{1}{37} \\ P_r(FC)_{odd} &= \frac{2}{37} \end{aligned} \tag{14}$$

for even and odd ANT channels, respectively.

4.4.3 Probability of collision

Combining the probability of time- and frequency coincidence, we get a probability of collision:

$$\begin{aligned} P_r(C) &= P_r(TC) * P_r(FC) \\ &= \frac{0.150 + 0.902}{T} * \frac{2}{37} \end{aligned} \tag{15}$$

This is for the case when the radios has the same connection interval. We can plot this for different values of T:

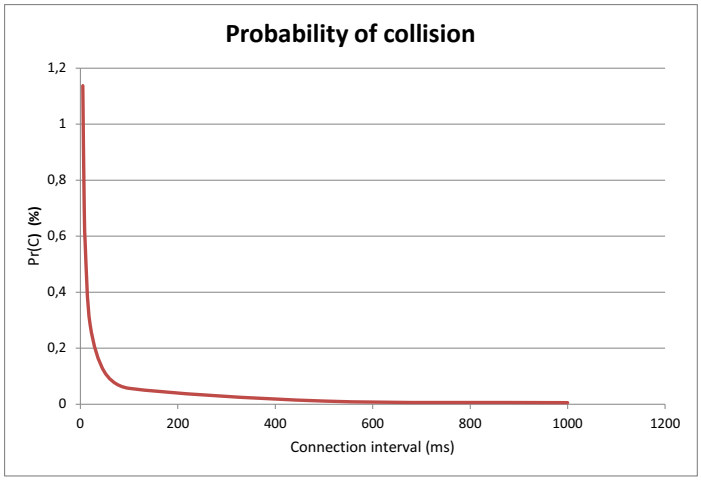

Figure 15: Probability of collision when both protocols share the same connection interval.

We can see that the low duty cycles lead to a very low probability of collision, even for the fastest connection interval (1.14%). This suggests that there may not be any need to coordinate radio activity between ANT and BLE. This will, however, be further investigated through experiments in chapter 6.

4.5 Dual-protocol sketch

Both ANT and Bluetooth Low Energy seem well suited for co-existence with other devices in the 2.4 GHz band. Both are low duty-cycle protocols, that is, their radios are active for a very short amount of time compared to the connection interval. While ANT employs a fixed frequency for the duration of communication, BLE uses frequency hopping both during discovery of devices, and during connections.

However, ANT has the possibility of adapting a frequency agility scheme, that will switch frequency if performance on the active channel is degraded. These features result in a very low probability of collision, as has been discussed in the previous section.

Adapting a collaborative dual-protocol scheme would require changes to the link layers to control the physical radio. Nordic Semiconductor writes its own protocol stack for BLE, but the ANT stack is proprietary. Even though Nordic produces ANT chips, they do not have access to the link layer, as this is programmed into the chip by Dynastream.

Thus, a collaborative protocol is difficult to implement. The *radio_active* signal from nRF8001 could, in theory, be used to disable the ANT radio. However, in practice, since the ANT link layer cannot be changed, this requires the ANT channel to be closed via a command sent over the UART interface. It cannot be guaranteed that this will turn off the radio in due time. Also, the channel will have to be re-opened shortly after, requiring a new synchronization with the other ANT device.

A non-collaborative solution would again require changes to the link layer, and also to the entire protocol, as this is such a fundamental change to how the radio should operate.

Both protocols provide a means of selecting the RF frequency: BLE by the channel map, and ANT in the channel configuration. Using these settings, a dual-protocol can be made that ensures the best possible performance by ensuring that the radios do not overlap in frequency. However, the BLE channel map can only be changed by the master device. The slave may not even suggest to remove a frequency from the hopping pattern. Also, the ANT channel may be implementing an ANT+ profile, where the radio frequency is specified, so that the application will not work unless using that frequency.

Based on the above discussion, it can be concluded that the protocol stacks should be left as-is. It should be up to the application designer to ensure that ANT and BLE do not overlap in frequency. Indeed, the discussion in chapter 4.4 suggests that such an overlap will not be critical, as a very low probability of collision is expected. Therefore, the focus in the implementation design was to provide an Application Programming Interface (API) for controlling an ANT and BLE device simultaneously, without any means of control.

5 Implementation

The hardware provided by Nordic Semiconductor for this project features one BLE module and one ANT module, to be controlled with a microcontroller from NXP. The BLE module is the nRF8001, which is Nordic's solution for peripheral devices. The ANT module is the nRF24AP2 device, part of an ANT development kit. This kit also features an USB stick and another AP2 module, that can be plugged into a PC. Nordic provides a Master Emulator USB board, that features the nRF80 radio. This is similar to nRF8001, but with flash memory instead of one-time programmable memory. Its firmware has been designed so it can emulate a central device, although the nRF80 is also meant to be used as a peripheral. Pictures of all this hardware can be found in Appendix A.

A special breakout board for the nRFgo Motherboard was designed before the start of this project. This board features a microcontroller from NXP, the LPC1114, with an ARM Cortex M0 core and peripherals for UART, SPI, and General-Purpose Input/Output (GPIO). The LPC MCU comes with an Integrated Development Environment (IDE) with all necessary compilers, as well as example design projects. This IDE is called LPCXpresso, and is based on the Eclipse platform.

5.1 Nordic μBlue™ Development Kit

The nRF8001 breakout board and Master Emulator are Nordic Semiconductor's development kit for their BLE solution, called *μBlue*™. It also consists of a Software Development Kit (SDK) that makes it easy to develop applications. However, this SDK is meant to run on a MCU with a 8051 core, not Cortex M0 which is the core of the LPC1114. The lower Hardware Abstraction Layer (HAL) thus needed to be ported to run on a Cortex M0.

The functionality of the SDK can be simplified as in figure 16. The HAL handles communication via SPI with the nRF8001. The nRF8001 is the SPI slave, but the host must be ready to receive data at any time. Thus, two special hand-shake signals, *ready* and *request*, is used to signal when the host must receive data, and to request to send data the opposite way, respectively.

Whenever data is received, a hook function in the *lib_aci* layer is called. This layer implements the ACI between the host MCU and the nRF8001. The hook function interprets the received event, and if the event is interesting for the application, a message is posted in the *dispatcher* via the function *post_msg*. All this happens in the interrupt context of the *ready* signal.

The dispatcher runs in main context. Whenever a new message is received from the ACI, an *event handler* is called for this specific event. This handler can respond to the event by calling *commands* in the ACI layer, which again calls a *send* function in

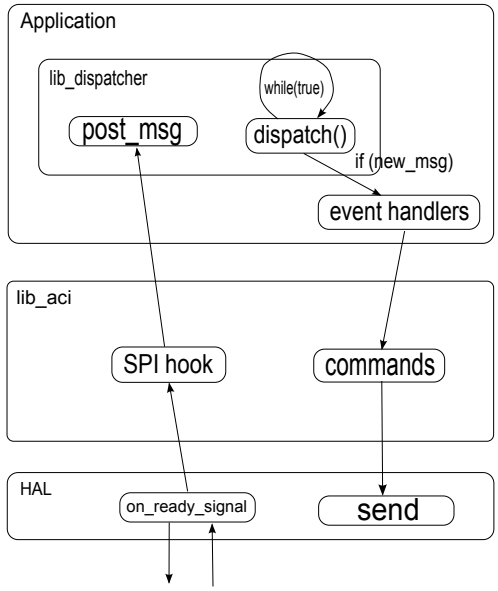

Figure 16: Nordic μBlue™ SDK dataflow. The flow on the left side runs in interrupt context, while the right side runs in the main context.

the HAL. The SPI communication is duplex, such that the send function only sets the *request*-signal high, and the actual transfer of the command to the nRF8001 is issued at the next *ready*-signal interrupt.

This design means that application developers only have to implement the *event handlers* to specify what the application functionality is. Nordic provides a software tool called nRFgo Studio that is used to automatically generate the source files needed to configure the dispatcher, as well as files for configuring the nRF8001 GATT server or client.

5.2 The ANT Development Kit

The ANT Development Kit contains several ANT modules, along with some simple I/O devices that can be used for example applications (pressing a button on one device to light a LED on another). There are two AP2-devices, where one supports 8 simultaneous channels and the other only one. There are also two simpler AP1 devices, which does not support frequency agility [4]. The ANT radios can be connected to a USB board and plugged into a PC. Images of this hardware is found in Appendix A.

[4]It can, however, be implemented in software on the host MCU

ANT also provides a software development kit. The flow in this software is similar to the μBlue™ SDK. However, instead of using a dispatcher, *callback functions* are implemented by the application developer, that trigger whenever an event that is interesting for the application occurs. These callbacks all run in interrupt context, and can possibly lead to a quite long Interrupt Service Routine (ISR). Also, the SDK is written in C++ and is intended for use with a PC.

5.3 Dual-protocol design

Both ANT and BLE SDKs are *event driven*, that is, the application reacts to events sent to the host MCU by the radio module. These events are quite similar in nature, and include status messages, responses to commands, and events happening on the radio interface. It can therefore be argued that it should not be hard to combine these SDKs.

The dispatcher used in the μBlue™ SDK works as a very simple operating system: the ACI posts messages that are placed in a queue, with different handlers for different message types. This suggests that an API can be designed, where the application is implemented as event handlers, and the underlying layers take care of sending and receiving messages to and from the radios. This design is shown in figure 17.

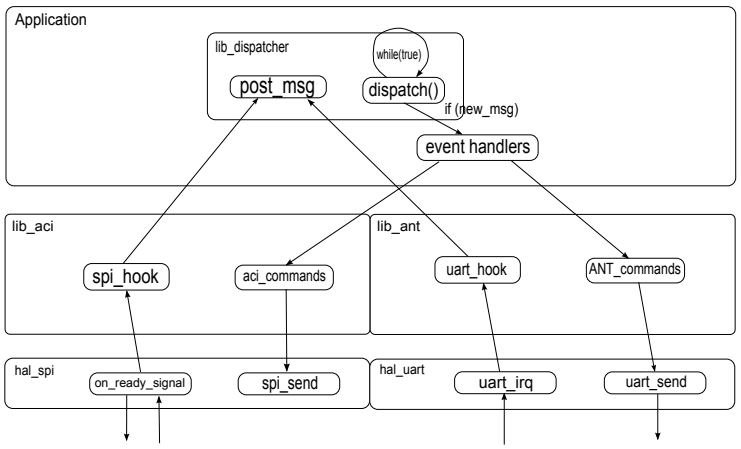

Figure 17: The ANT and BLE common API.

The ANT SDK was rewritten to work with lib_dispatcher by defining event handlers, and replacing the callback functions with a call to *post_msg*. A single buffer stores the command responses, while 8 buffers are created for channel events[5]. 8 more buffers are used for RX data.

[5]The maximum allowed channels on the AP2 device is 8

In the original ANT SDK, a single callback function handles channel events. The nature of this event is then interpreted inside this function. This is different than the μBlue™ SDK, where different events each has its own handler. The SDK was thus rewritten such that the event is interpreted before the call to *post_msg*, with a different handler for each event.

6 Test of implementation

In order to test how well ANT and BLE performs when co-located, a simple test application was made. The application consists of a counter that is incremented each connection interval, and the receiver side checks whether the counter has incremented by one since the last reception. If the difference is more than one, a lost packet is registered.

ANT and BLE allows for different connection parameters that can be changed according to the application. These parameters include:

- ANT:
  - Channel period
  - Radio frequency
  - Transmit power
  - Channel type (master or slave)
  - Frequency agility
  - Search timeout
- BLE:
  - Connection interval
  - Transmit power
  - Advertising interval
  - Channel map
  - Search timeout

The search timeout for both protocols was set to a fixed value of 15 seconds, since varying this would give too many test cases. Likewise, the transmit power was kept constant at $0dBm$ for the same reason. However, the transmit power was varied in a special stress test, which will be discussed later. The BLE advertising interval was also kept constant at $100ms$.

Three different tests were run with 8 different connection intervals: a reference test, a default parameters test, and an optimized test. Each individual test was run for 1 minute per connection interval, and with the ANT node as both master and slave. In the reference tests, only ANT or BLE ran the application, in order to differentiate between packet loss due to interference between ANT and BLE, and loss which would be present anyway. The default parameters tests ran ANT and BLE simultaneously, with the default BLE channel map. The optimized tests ran ANT and BLE, with the BLE channel map optimized to avoid the used ANT

frequency. All these test parameters, and test results, can be studied in detail in appendix B.

These three tests were repeated with 1 meter and 10 meter distance between the receiver and transmitter nodes. For the 10 meter tests, a laptop had to be used to control the motherboard application, while a desktop computer controlled the Master Emulator and ANT USB stick. Figure 18 illustrates this setup. For the 1 meter test, the RS232 cable was plugged directly into the desktop computer, and the laptop was not used.

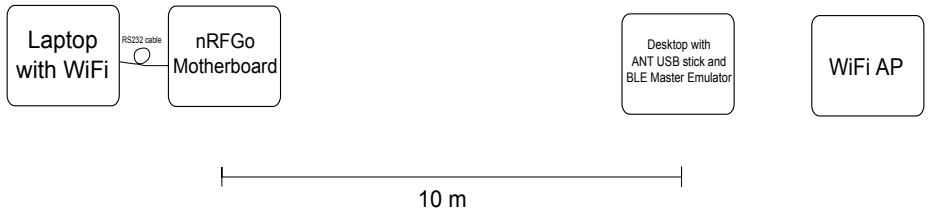

Figure 18: Test setup. For the 1 meter tests, the RS232 cable was plugged directly into the desktop computer.

A co-existence test with WiFi was also conducted. In this test, the WiFi access point was configured to use 802.11g channel 6, which spans between 2.426 and 2.448 GHz. In the optimized WiFi test, the ANT and BLE connections avoided the WiFi channel. The access point was placed two meters from the desktop computer, while the laptop was used as the WiFi station.

6.1 Test application design

The application consists of four components: the motherboard application which runs the dual-protocol implementation, two PC applications running the Master Emulator and ANT USB stick, and Python scripts for control and logging.

6.1.1 Motherboard application

The motherboard application was designed using the dual-protocol implementation presented in chapter 5. The required event handlers were implemented, and a simple user interface was added in order to control and setup the tests.

The user interface is implemented using a 'software UART' on the microcontroller, since the hardware UART is used to control the ANT device. The software UART is a library that came with the LPCXpresso IDE. The library uses two GPIO ports for input and output, and a hardware timer interrupts the MCU whenever these ports need to be read/written to. The baudrate is fixed at 9600 bps.

In order for the nRF8001 to send a new counter value at each connection interval, the *radio_active* signal was used to update the counter characteristic. The ANT node sends an *EVENT_TX* over the UART interface at each connection interval, and this was used to update the counter for ANT.

6.1.2 PC applications

The PC applications for controlling the Master Emulator and ANT USB stick were created by modifying reference designs from Nordic Semiconductor and Dynastream. The Master Emulator software is in the development stage, and lacked functionality for controlling the channel map. This was added with help from the Nordic software team. The resulting applications are started in the command-line, where device configuration parameters are given as startup arguments.

6.1.3 Python scripts

In order to control the tests, two Python scripts were made. A motherboard controller script creates a network socket, listens for data on this socket, and passes the data via the computer's serial port to the nRFgo motherboard. It also logs the output from the motherboard as a plain-text file. The other script forks out processes for the ANT- and BLE PC applications, and communicates with the motherboard controller via a network socket. The Python scripts are attached in appendix D.4.

6.2 Results

Below are graphs showing the packet loss in percent, as a function of the *connection frequency*. The term connection frequency must not be confused with the *radio* frequency, which is quite another matter. The term connection frequency was used because an ANT channel period is given as a frequency, and the user interface to the test application was thus designed to give the connection interval as a frequency.

The packet loss is effected by the time it takes for device discovery: for ANT, a low connection frequency results in a longer discovery time, while for BLE, the discovery time includes service discovery and channel map update. The reference for the packet loss percentage was the connection frequency times the test duration, which is 60 seconds. The results are discussed later in chapter 8, the data for the plots are attached in appendix B.

6.2.1 1 meter tests, ANT.

In these tests, the reference test is conducted with only ANT. The unoptimized test is with BLE traffic, without disabling the ANT frequency in the BLE channel map. The optimized test features BLE traffic, with the ANT frequency disabled in the channel map.

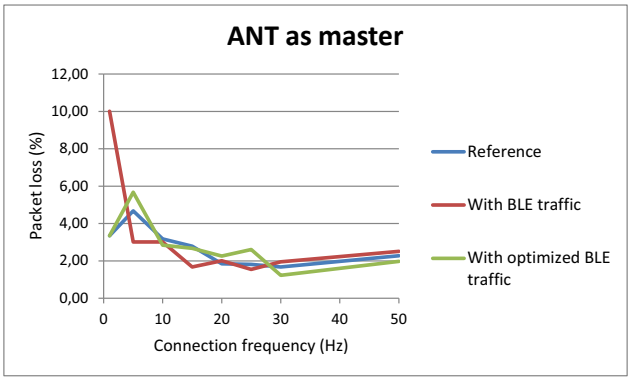

Figure 19: 1 meter tests, ANT. The motherboard node as master.

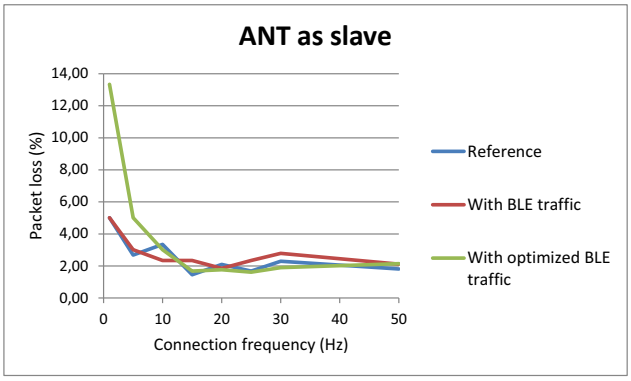

Figure 20: 1 meter tests, ANT. The motherboard node as slave.

6.2.2 1 meter tests, BLE.

In these tests, the reference test is conducted with only BLE. The unoptimized test is with ANT traffic, without disabling the ANT frequency in the BLE channel map. The optimized test also features ANT traffic, with the ANT frequency disabled in the channel map.

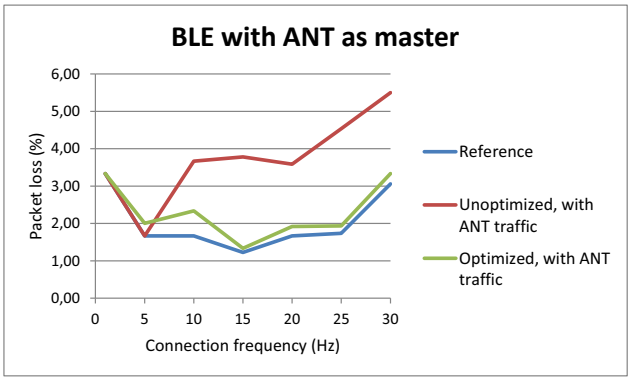

Figure 21: 1 meter tests, BLE. The ANT motherboard node as master.

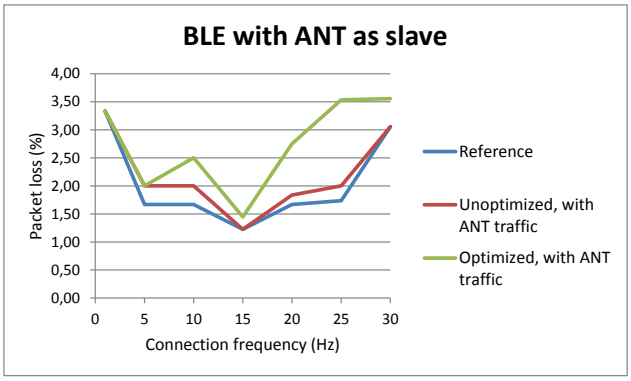

Figure 22: 1 meter tests, ANT. The ANT motherboard node as slave.

6.2.3 10 meter tests, ANT.

These tests are the same as the 1 meter ANT tests, only with a 10 meter distance between transmitter and receiver.

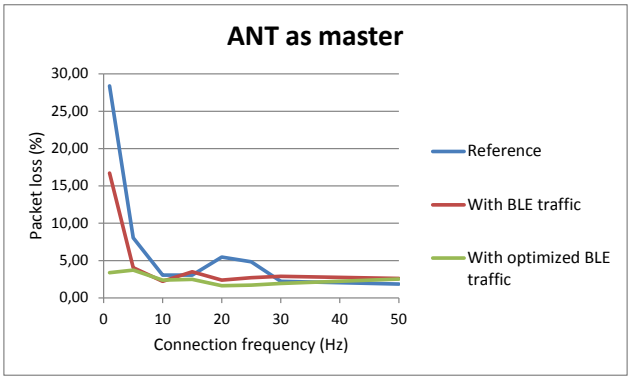

Figure 23: 10 meter tests, ANT. The motherboard node as master.

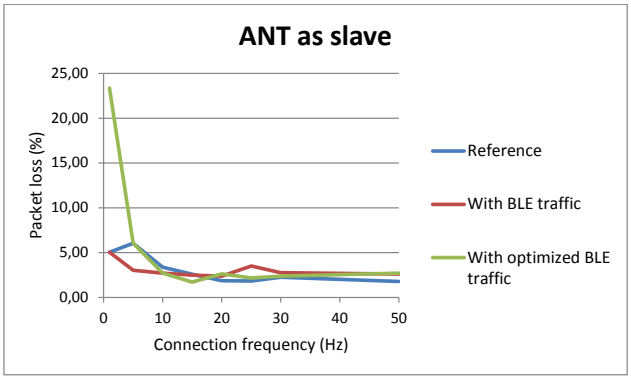

Figure 24: 10 meter tests, ANT. The motherboard node as slave.

6.2.4 10 meter tests, BLE.

These tests are the same as the 1 meter BLE tests, only with a 10 meter distance between transmitter and receiver.

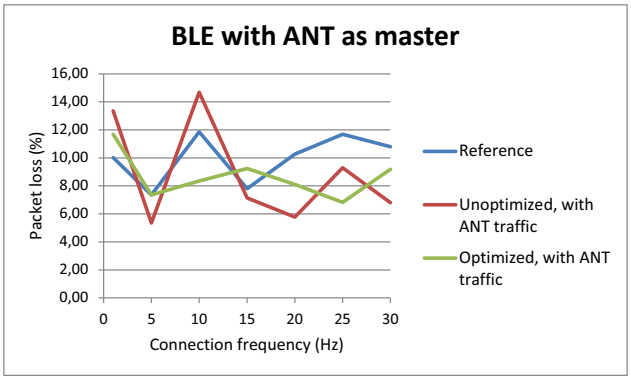

Figure 25: 10 meter tests, BLE. The ANT motherboard node as master.

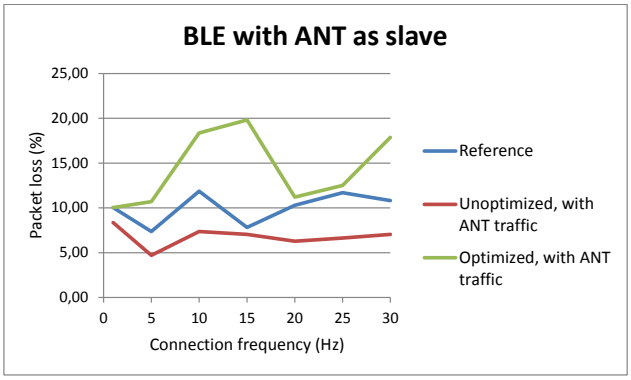

Figure 26: 10 meter tests, BLE. The ANT motherboard node as slave.

6.2.5 WiFi tests, ANT. (10 meter range)

During these tests, the laptop downloaded a large file over the WiFi network. In the unoptimized tests, the ANT frequency was chosen to be at the edge of the WiFi channel (2.446 GHz), while at the optimized test the ANT channel was outside the WiFi band (2.466 GHz).

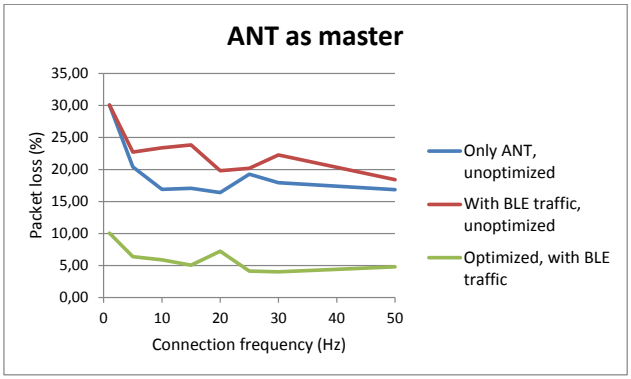

Figure 27: WiFi tests, ANT. The motherboard node as master.

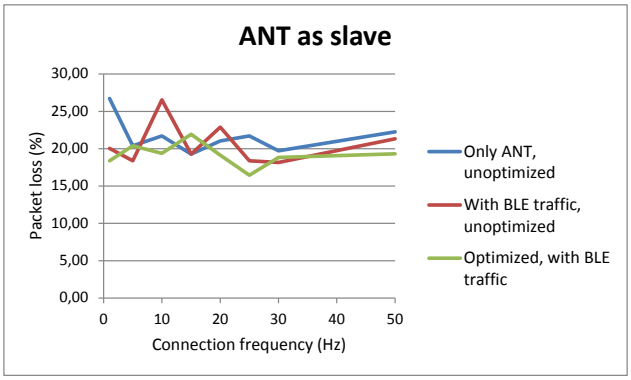

Figure 28: WiFi tests, ANT. The motherboard node as slave.

6.2.6 WiFi tests, BLE. (10 meter range)

In the unoptimized tests, the channel map was selected to use all frequencies, while the optimized channel map avoided both the WiFi band and the ANT channel. The laptop was set to download a large file during these tests as well.

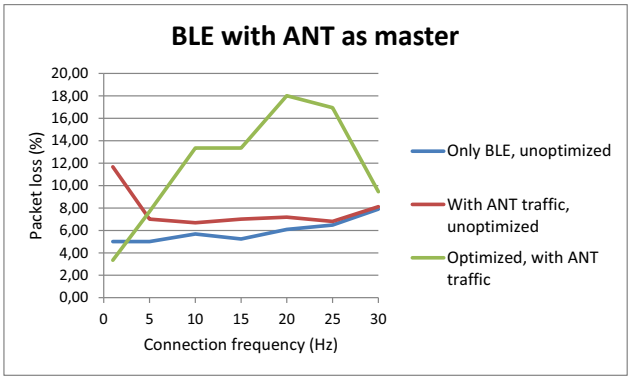

Figure 29: WiFi tests, BLE. The ANT motherboard node as master.

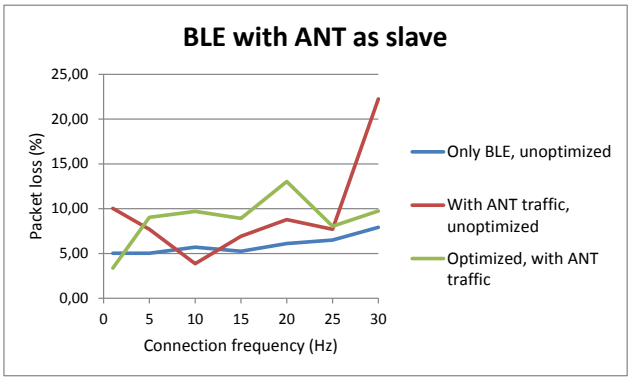

Figure 30: WiFi tests, BLE. The ANT motherboard node as slave.

6.2.7 Spectrum analysis

In addition to the other tests, a spectrum analyzer was used to demonstrate how ANT and BLE use the spectrum. In this experiment, the BLE radio was set only to advertise - not initiate a connection - since advertisements only occupy three channels. Also, since the radio protocols have a low duty cycle, the analyzer was configured to measure average power. The ANT radio was set to broadcast some data at channel 5 (2.405 GHz).

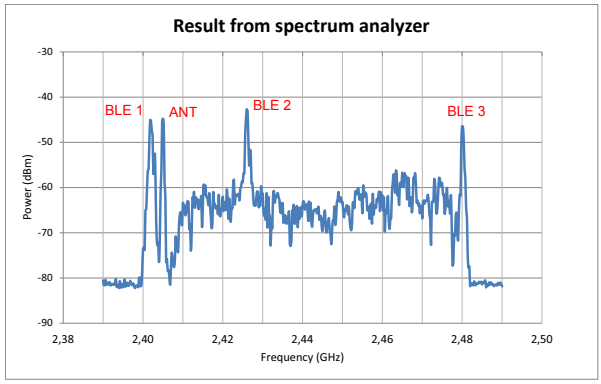

Figure 31: ANT and BLE advertising activity recorded by a spectrum analyzer. Other 2.4 GHz activity (probably WiFi) can also be seen.

Figure 31 is the result of this experiment. We can clearly see the three BLE advertisement channels, as well the ANT channel. There is also some other traffic in the band, as can be seen from the fact that the background noise is much lower at the edges of the band. This traffic is probably from WiFi access points that exists in the building. The figure clearly illustrates that the spectrum can be shared by several users of the band.

6.3 Example application

A more practical example application was also implemented. In a previous project by some summer interns at Nordic Semiconductor, a digital weighing scale was equipped with an ANT device. Using this scale, people can monitor their weight progress on their ANT-equipped smartphone.

The motherboard was programmed so that the ANT device received broadcasts from the scale. The weight was updated at 4 Hz. In the event handler handling received ANT broadcasts, this data was forwarded to the BLE device. The Master Emulator was controlled through a test program called Master Control Panel to receive this data, and display it on screen.

Although the usability of this application can be questioned, it is still a more realistic use case than the counter test: the application became a bridge between an ANT and a BLE network. The application was implemented in only three hours, using the dual-protocol API presented in chapter 5.

7 Suggestions for a single-chip solution

In the dual-protocol implementation, the two radios do not co-operate on radio timing. This is because the two separate chips run their own link layer, and it is therefore impossible to control the exact timing from the host MCU.

If the firmware of both chips were available for altering, a collaborative solution could be made. Figure 32 is a simplified view of such a protocol. Both ANT and BLE nodes are capable of running several connections[6] simultaneously. That is, a BLE module may be in a connection as a slave, while advertising data to other devices. An ANT module may run several channels simultaneously, both as master and slave. These connections would need internal scheduling in the protocols, as well as a scheduling mechanism between the protocols. When a scheduling that guarantees that the two radios will not be on simultaneously is available, the different connections reserve time slots in the protocol timer.

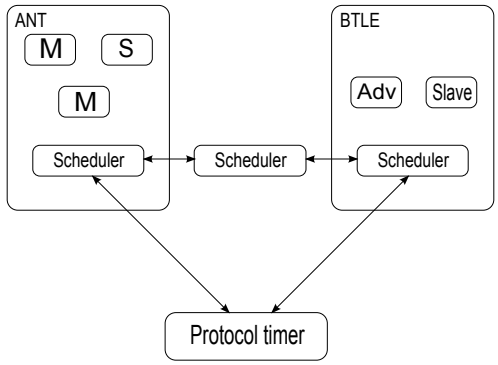

Figure 32: A shared protocol scheme. The two protocols can implement several connections simultaneously, and need to schedule these internally. They also need to schedule time slots between themselves, and only then can they reserve a time slot in the protocol timer.

An algorithm that can be used for this scheme is presented below (from [20]):

- Assume t non-overlapping connections, represented as sets $(M_1, M_2, \ldots, M_t)$. M_1 is the set of reference, that is, $\delta_1 = 0$ and $\delta_i \neq 0$ for $i = 1, 2, \ldots, t$.
- We want to add an additional set M_{t+1}, and find an appropriate shift δ_{t+1} that prevents overlaps.
- Identify for all $i = 2, 3, \ldots, t$ the *illegal* shifts δ_i versus M_{t+1}.
- Keeping in mind the illegal shifts, M_{t+1} must be shifted relative to M_1 by a value in the interval

[6] Connections may not be the correct term here: a BLE advertiser is not in a connection.

$$k * gdc(T_1, T_{t+1}) < \delta_{t+1} < (k+1) * gcd(T_1, T_{t+1})$$

However, this algorithm grows exponentially with the number of connections. The negotiation between the connections are done beforehand, to ensure that the connections will get their share of radio time. So, while the algorithm may work well for a simple module as the nRF8001 that can only handle a single connection while advertising, introducing an 8-channel ANT device will severely complicate the scheduling algorithm.

A simpler solution can be made if scheduling *rejections* are tolerated. That is, if a connection can tolerate a rejection to use the radio at a specific time. Indeed, in the envisaged use cases for ANT and BLE, it may not be critical that a connection event is missed. If that is the case, then no negotiations are needed beforehand: the different connections can request a time slot without knowledge of other connections. The scheduler then either accepts or rejects the request, and may keep track of rejected connections in order to ensure fairness. This simplified dual-protocol is shown in figure 33.

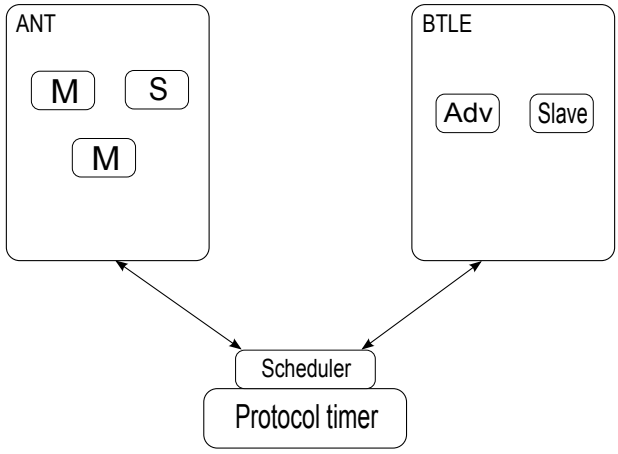

Figure 33: A simpler shared protocol scheme. A common scheduler handles requests for timeslots. Requests can be rejected.

In BLE, the master device controls all timing. The slave may only give hints such as maximum- and minimum connection intervals. The slave may however terminate the connection due to bad timing, and give this reason to the master. If it were possible for the slave to request a different *anchor point*, that is, request a shift of its connection interval, the connection could continue and allow for other (not necessarily BLE) radio traffic to and from the slave.

ANT employs an automatic scheme that enables masters on the same radio frequency to shift its timing according to other masters' activity. The shift is done

slowly, so a tracking slave may keep up. This autonomous shifting also handles clock drift between master and slave. However, it is still the slave that is the tracker: it cannot *request* such a shift.

Both these facts makes it difficult to implement a dual-protocol chip, and more extensive studies would have to be made in order to do it.

8 Discussion

In the previous chapters, a dual-protocol solution has been implemented and tested. This chapter will discuss the dual-protocol design and test results.

Howitt et al. states in [15] that *central to the co-existence issue between wireless devices is the ability to differentiate between operational conditions which will and will not result in the communication devices failing to meet the requirements of an application.* The envisaged use-cases for ANT and BLE are low duty-cycle applications, that are more concerned about updating a device *state* rather than streaming data at high throughput. This state is, for the envisaged applications, not a quickly changing attribute. For example, a heart rate sensor used in sports equipment does not need to send the heart rate very often, perhaps only twice a second, in order for the user to get a good experience. Even a computer mouse does not need extremely quick updates: an update rate of 10 ms will give a good enough experience, according to Nordic Semiconductor professionals. Also, the size of the attribute is not very large: a heart rate can for normal humans be represented by a single byte.

8.1 Probability of collision

The above discussion suggests that ANT and BLE applications will not be very sensitive to packet loss. However, should the devices interfere enough to destroy the functionality of one another, the application will of course fail to meet its requirements. This was discussed in chapter 4.3, where a probability of collision between ANT and BLE was derived.

The probability of collision was analyzed for the case where both radios have the same connection interval. In this case, the radios will either collide at every connection event, or they will not. If the radios have different connection intervals, the eventual collisions will happen less often.

Due to the low probability of collision, the proposed implementation did not try to strictly control radio timing, and relied on the interference avoidance techniques already present in the protocols.

8.2 Collaborative solution

It would be possible, in theory, to create a collaborative dual-protocol solution. However, this would require changes deep in the radios' protocol stacks, as most collaborative co-existence techniques alter the MAC and Physical OSI layer (PHY) layers. Nordic Semiconductor produces its own BLE stack, but the ANT stack is proprietary. Even though Nordic produces the ANT chip, they do not have insight

in the ANT protocol stack. Also, this project had a practical element: a solution should be implemented. It took considerable amounts of time to get to know the two protocols, and digging even deeper in the protocol stacks would taken even more time.

8.3 Power considerations

Both ANT and BLE are designed to run on small coin-cell batteries with limited capacity. Combining these two in a single system will lead to higher power usage.

What is especially critical for a coin-cell battery is the peak current drainage. For a typical coin-cell battery like the CR2032, the total peak current is advised to be below $20mA$ at any time [8]. The peak current of the nRF8001 is specified to be below $14mA$, while for the nRF24AP2 ANT device the peak current is $17mA$ in RX mode. In addition, the host MCU needed to control the devices will draw current. Thus, a combined solution with no co-operation in radio timing (control of peak current drainage) will severely reduce battery life.

8.4 Issues with the dual-protocol API

When designing the dual-protocol API, it was sought to re-use as much software as possible in order to reduce design-time. The μBlue™ SDK from Nordic was used as the basis for the design, as this will provide a common design flow for application development. This SDK includes a *dispatcher*, which provides simple operating system-like functionality.

The ANT SDK was rewritten to use the dispatcher. However, a problem occurred during testing of the ANT library: when using fast connection intervals (above 50 Hz) the ANT module reported that it was busy transmitting a packet, when the host tried to send it a new one. At every radio transmit, the ANT module sends an EVENT_TX message to the host, which is used to initiate new transmits. These messages were put in the dispatcher queue, and at fast connection intervals, the dispatcher did not have time to process all the messages in the queue before a new EVENT_TX happened. Therefore, the host MCU tried to transmit two or more packets in a relatively short period of time.

This problem suggests that the ANT library should have been left as-is. However, in the original ANT SDK, all message processing happens in the context of a UART interrupt. This leads to a long Interrupt Service Routine (ISR), which is thought of as a bad thing in software development. Even though, doing this would have solved the above problem.

8.5 Issues with the test application

In order to guarantee that a new counter value was sent at every connection interval, the BLE test application used the *radio_active* signal provided by the nRF8001 to update the counter characteristic. This is quite a hack, as the intended purpose of the signal is to reduce activity in the host MCU while the nRF8001 is transmitting. The usage introduced a new interrupt context, complicating the program flow. Late in the application development, it was discovered that a library in the μBlue™ SDK called *lib_timed_services* could have been used for this purpose.

The *radio_active* signal does not trigger when the connection interval is below $30ms$ (above 33 Hz). This, and the problem with any consecutive EVENT_TX messages in the dispatcher queue, meant that tests could not be performed for connection intervals above 30 Hz for BLE, and 50 Hz for ANT. Faster connection intervals means a higher probability of collision, and the results may have been different if these could be tested. However, most use cases for ANT and BLE are low duty cycle, as a fast connection intervals means higher power usage.

The software UART was crucial for debugging. It provided a means of displaying the sequence of events, as well as displaying test results and control of the test application. However, the library used a timer interrupt to read and write data to a GPIO port, and this interrupt had to run at highest priority in order to ensure correctness. This may cause data transfer between the MCU and radio modules to be interrupted, although this was not observed. The low baudrate available with this library (9600 bps) caused some problems, especially when printing long strings. It would have been better to have a second hardware UART available on the MCU for the purpose of debugging.

8.6 Test results

The test results are shown in figures 19 through 30. For the 1 meter tests, ANT packet loss stabilizes at around 2%. The reason for higher packet loss at the lower connection intervals is the time it takes to synchronize the master and slave: if the master transmits only once per second it takes longer time to synchronize with it. The BLE application was designed so that the fastest connection interval was used during service discovery, before the connection interval used for the test was set.

The 1 meter BLE tests also show around 2% packet loss. The results show an increasing trend as the connection interval increases. Figure 21 shows increased packet loss when ANT traffic is present and unoptimized. However, in the next figure, the optimized graph is higher than the unoptimized one. This may suggest that BLE performs better when the ANT node is a slave.

The 10 meter ANT results also show around 2% packet loss. Note that there is no special correlation between the reference, unoptimized, and optimized tests. In

the 10 meter BLE tests, the packet loss is somewhat higher, in the area of 10%. However, it is quite impossible to see any correlation with ANT traffic here as well. Indeed, in figure 26, the reference test has a higher packet loss than with unoptimized ANT traffic.

When a WiFi access point is introduced to the test environment, ANT packet loss increases quite dramatically. Figure 27 shows an expected result: the unoptimized reference- and combined test shows a considerably higher packet loss than the optimized test. However, when the ANT node is a slave, the optimized results jumps up to the same range as the unoptimized ones.

BLE seems to tackle the WiFi traffic better than ANT. However, the optimized tests generally shows a higher packet loss than the unoptimized tests. This may suggest that disabling many frequencies in the channel map has a negative impact on BLE interference avoidance. In the optimized tests, the channel map was set to avoid both the ANT channel and the 22 MHz wide WiFi channel.

The results are, in general, quite inconsistent. This may be due to lack of control of other 2.4 GHz traffic in the test environment. The test location was directly above the software department at the Nordic Semiconductor office, where other tests with especially BLE are run continuously. There are also several WiFi networks present in the office building, occupying a large amount of spectrum (2.417 - 2.472 GHz). These WiFi access points were, however, quite far away from the test area, with signal strengths at around $-70dB$. Even though, it would have been better to perform the tests in a more controlled environment.

The tests were also run only once per connection frequency. This results in inconsistent data points, as the results may be impacted by interference from other traffic which would not have been present at another point in time. Running many tests, and calculating average values, would improve test consistency.

9 Conclusions and future work

During this project, the ANT and Bluetooth Low Energy (BLE) protocols were studied and analyzed for co-existence. A dual-protocol solution with two separate radio chips - controlled by a single microcontroller - was implemented, and issues related to co-existence was tested. Although inconsistent, it is hard to derive from the test results any performance degradation due to the co-location of the two radios.

The 1 meter test results are expected to be most consistent, as the received signal strength is considerably larger than for the 10 meter tests. However, it is still hard to determine whether the packet loss is impacted by the co-existence of the two protocols. It would be preferable to repeat the tests in a more controlled environment, such as an echo-free room.

Running many instantiations of the tests and calculating average values would also improve the test consistency. This would however require a better test setup: the Python scripts logged the results as a plain-text file, and the results had to be manually inserted into a spreadsheet. Making the test applications output the results in a more parseable format, such as comma-separated values, would improve the automaticity of the tests, and average values could easily be computed.

A more realistic example application than the counter test was also implemented. This application had no issues relating to co-existence: the weight from the scale was correctly forwarded to the BLE device, and received at the Master Emulator which was connected to a PC.

As discussed in chapter 8.3, the ANT and BLE radios are meant to be driven by a coin-cell battery, which has limited capacity especially when it comes to peak current drainage. In order to control the peak current, it would have been preferable to have strict control of radio timing, that is, control of when each radio should be turned on. However, this will require changes to the link layer of both protocols as these are autonomous: the host microcontroller cannot control the radio timing exactly. In actual products featuring both an ANT and a BLE node, this issue has to be addressed, and will require further research.

A Used hardware and software

A.1 Software used

- LPCXpresso IDE 3.6.1 with ARM compiler
- Microsoft Visual Studio 2010
- Python 2.6
- nRFgo Studio

A.2 Hardware used

- nRFgo Motherboard
- nRF2743 Extension board for ANT and uBlue
- nRF2740 breakout board with an nRF8001 module
- ANT AP2 1-channel device
- nRF2739 uBlue Master Emulator
- ANT AP2 8-channel device on an USB stick
- NXP LPC-Link programmer

A.3 Images of hardware

Figure 34: ANT development kit. The modules labeled AP2-1CH and AP2-8CH were used in this project. The 8-channel device was connected to one of the USB modules and plugged in a PC.

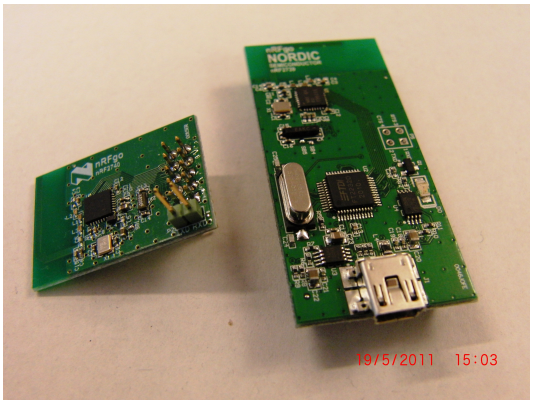

Figure 35: Nordic μBlue™ development kit. The left module contains an nRF8001 device, with an on-board antenna. The right module is the Master Emulator, connected to a PC via an USB cable.

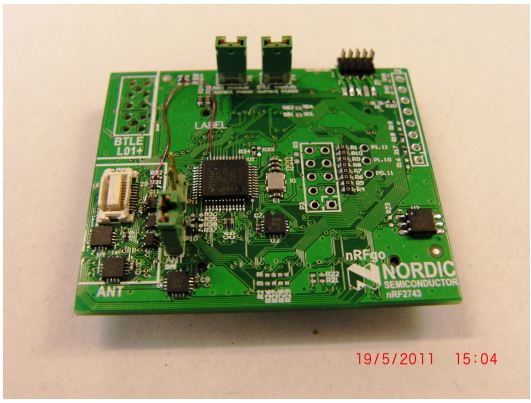

Figure 36: Nordic nRF2743 extension board. Features a NXP LPC1114 microcontroller, and interfaces for ANT and BLE breakout boards. This board was specially designed for this project.

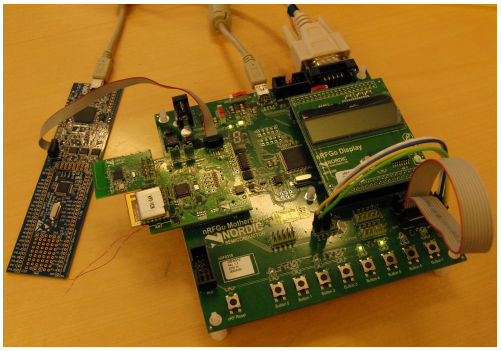

Figure 37: Nordic nRFgo Motherboard with the extension board, ANT and BLE devices, and NXP programmer

B Test result tables

1 meter: ANT reference test

Test #	Connection interval (Hz)	Frequency	Peripheral type	PC type	Results		
					Correctly transmitted	Loss in %	Discovery time
1	1	66	M	S	58	3,33	2150
2	5	66	M	S	286	4,67	1753
3	10	66	M	S	581	3,17	349
4	15	66	M	S	875	2,78	154
5	20	66	M	S	1178	1,83	148
6	25	66	M	S	1473	1,80	75
7	30	66	M	S	1770	1,67	115
8	50	66	M	S	2932	2,27	193
9	1	66	S	M	57	5,00	2107
10	5	66	S	M	292	2,67	1519
11	10	66	S	M	580	3,33	2109
12	15	66	S	M	887	1,44	1163
13	20	66	S	M	1175	2,08	1549
14	25	66	S	M	1475	1,67	1183
15	30	66	S	M	1759	2,28	1404
16	50	66	S	M	2946	1,80	1125

1 meter: BLE reference test

Test #	Connection interval (Hz)	Channel map	Results		
			Correctly transmitted	Loss in %	Discovery time
1	1	0x1FFFFFFFFF	58	3,33	1660
2	5	0x1FFFFFFFFF	295	1,67	1431
3	10	0x1FFFFFFFFF	590	1,67	1520
4	15	0x1FFFFFFFFF	889	1,22	1602
5	20	0x1FFFFFFFFF	1180	1,67	1410
6	25	0x1FFFFFFFFF	1474	1,73	1436
7	30	0x1FFFFFFFFF	1745	3,06	1422

1 meter: ANT + BTLE with default parameters

Test #	ANT setup				BTLE setup		Results				
	Channel period	Frequency	Peripheral type	PC type	Connection interval	Channel map	ANT			BTLE	
							Correctly transmitted	Loss in %	Discovery time	Correctly transmitted	Loss in %	Discovery time
1	1	66	M	S	1	0x1FFFFFFFFF	54	10,00	5201	58	3,33	4083
2	5	66	M	S	5	0x1FFFFFFFFF	291	3,00	416	295	1,67	4071
3	10	66	M	S	10	0x1FFFFFFFFF	582	3,00	234	578	3,67	4145
4	15	66	M	S	15	0x1FFFFFFFFF	885	1,67	147	866	3,78	4106
5	20	66	M	S	20	0x1FFFFFFFFF	1176	2,00	264	1157	3,58	4115
6	25	66	M	S	25	0x1FFFFFFFFF	1477	1,53	96	1432	4,53	4193
7	30	66	M	S	30	0x1FFFFFFFFF	1765	1,94	149	1701	5,50	4198
8	50	66	M	S	30	0x1FFFFFFFFF	2925	2,50	262	1745	3,06	4048
9	1	66	S	M	1	0x1FFFFFFFFF	57	5,00	2112	58	3,33	4132
10	5	66	S	M	5	0x1FFFFFFFFF	291	3,00	2296	294	2,00	4128
11	10	66	S	M	10	0x1FFFFFFFFF	586	2,33	1892	588	2,00	4223
12	15	66	S	M	15	0x1FFFFFFFFF	879	2,33	1632	889	1,22	4097
13	20	66	S	M	20	0x1FFFFFFFFF	1178	1,83	1150	1178	1,83	4126
14	25	66	S	M	25	0x1FFFFFFFFF	1465	2,33	1371	1470	2,00	4108
15	30	66	S	M	30	0x1FFFFFFFFF	1750	2,78	1313	1745	3,06	4058
16	50	66	S	M	30	0x1FFFFFFFFF	2937	2,10	1294	1742	3,22	4120

1 meter: ANT + BTLE optimized

Test #	ANT setup				BTLE setup		Results					
							ANT			BTLE		
	Channel period	Frequency	Peripheral type	PC type	Connection interval	Channel map	Correctly transmitted	Loss in %	Discovery time	Correctly transmitted	Loss in %	Discovery time
1	1	66	M	S	1	0x1F1FFFFFFF	58	3,33	1213	58	3,33	4162
2	5	66	M	S	5	0x1F1FFFFFFF	283	5,67	2825	294	2,00	4146
3	10	66	M	S	10	0x1F1FFFFFFF	583	2,83	1113	586	2,33	4427
4	15	66	M	S	15	0x1F1FFFFFFF	876	2,67	610	888	1,33	4247
5	20	66	M	S	20	0x1F1FFFFFFF	1173	2,25	119	1177	1,92	4215
6	25	66	M	S	25	0x1F1FFFFFFF	1461	2,60	296	1471	1,93	4155
7	30	66	M	S	30	0x1F1FFFFFFF	1778	1,22	157	1740	3,33	4173
8	50	66	M	S	30	0x1F1FFFFFFF	2941	1,97	301	1741	3,28	4204
9	1	66	S	M	1	0x1F1FFFFFFF	52	13,33	8109	58	3,33	4185
10	5	66	S	M	5	0x1F1FFFFFFF	285	5,00	3095	294	2,00	4108
11	10	66	S	M	10	0x1F1FFFFFFF	582	3,00	1707	585	2,50	4223
12	15	66	S	M	15	0x1F1FFFFFFF	885	1,67	1370	887	1,44	4055
13	20	66	S	M	20	0x1F1FFFFFFF	1179	1,75	1351	1167	2,75	4206
14	25	66	S	M	25	0x1F1FFFFFFF	1476	1,60	1216	1447	3,53	4680
15	30	66	S	M	30	0x1F1FFFFFFF	1766	1,89	1235	1736	3,56	4263
16	50	66	S	M	30	0x1F1FFFFFFF	2936	2,13	1333	1731	3,83	4088

10 meter: ANT packet counter reference test

Test #	Connection interval (Hz)	Frequency	Peripheral type	PC type	Results		
					Correctly transmitted	Loss in %	Discovery time
1	1	66	M	S	43	28,33	7102
2	5	66	M	S	276	8,00	2752
3	10	66	M	S	582	3,00	436
4	15	66	M	S	873	3,00	686
5	20	66	M	S	1135	5,42	146
6	25	66	M	S	1428	4,80	272
7	30	66	M	S	1761	2,17	78
8	50	66	M	S	2946	1,80	194
9	1	66	S	M	57	5,00	2097
10	5	66	S	M	282	6,00	3293
11	10	66	S	M	580	3,33	2099
12	15	66	S	M	877	2,56	1559
13	20	66	S	M	1178	1,83	1395
14	25	66	S	M	1473	1,80	1376
15	30	66	S	M	1760	2,22	1163
16	50	66	S	M	2948	1,73	1212

10 meter: BTLE packet counter reference test

Test #	Connection interval (Hz)	Channel map	Results		
			Correctly transmitted	Loss in %	Discovery time
1	1	0x1FFFFFFFFF	54	10,00	1426
2	5	0x1FFFFFFFFF	278	7,33	1482
3	10	0x1FFFFFFFFF	529	11,83	1700
4	15	0x1FFFFFFFFF	830	7,78	1577
5	20	0x1FFFFFFFFF	1077	10,25	1475
6	25	0x1FFFFFFFFF	1325	11,67	1466
7	30	0x1FFFFFFFFF	1606	10,78	1511

10 meter: ANT + BTLE with default parameters

Test #	ANT setup				BTLE setup		Results				
	Channel period	Frequency	Peripheral type	PC type	Connection interval	Channel map	Correctly transmitted	ANT Loss in %	Discovery time	Correctly transmitted	BTLE Loss in %	Discovery time
1	1	66	M	S	1	0x1FFFFFFFFF	50	16,67	9211	52	13,33	4144
2	5	66	M	S	5	0x1FFFFFFFFF	288	4,00	1420	284	5,33	4091
3	10	66	M	S	10	0x1FFFFFFFFF	587	2,17	715	512	14,67	4127
4	15	66	M	S	15	0x1FFFFFFFFF	869	3,44	821	836	7,11	4211
5	20	66	M	S	20	0x1FFFFFFFFF	1172	2,33	467	1131	5,75	4111
6	25	66	M	S	25	0x1FFFFFFFFF	1460	2,67	300	1361	9,27	4107
7	30	66	M	S	30	0x1FFFFFFFFF	1749	2,83	281	1678	6,78	4299
8	50	66	M	S	30	0x1FFFFFFFFF	2923	2,57	262	1655	8,06	4122
9	1	66	S	M	1	0x1FFFFFFFFF	57	5,00	4098	55	8,33	4229
10	5	66	S	M	5	0x1FFFFFFFFF	291	3,00	1889	286	4,67	4073
11	10	66	S	M	10	0x1FFFFFFFFF	584	2,67	1196	556	7,33	4296
12	15	66	S	M	15	0x1FFFFFFFFF	878	2,44	1225	837	7,00	4164
13	20	66	S	M	20	0x1FFFFFFFFF	1172	2,33	1196	1125	6,25	4202
14	25	66	S	M	25	0x1FFFFFFFFF	1448	3,47	1891	1401	6,60	4480
15	30	66	S	M	30	0x1FFFFFFFFF	1751	2,72	1264	1674	7,00	4419
16	50	66	S	M	30	0x1FFFFFFFFF	2923	2,57	1273	1634	9,22	4183

10 meter: ANT + BTLE with default parameters

Test #	ANT setup				BTLE setup		Results					
							ANT			BTLE		
	Channel period	Frequency	Peripheral type	PC type	Connection interval	Channel map	Correctly transmitted	Loss in %	Discovery time	Correctly transmitted	Loss in %	Discovery time
1	1	66	M	S	1	0x1FFFFFFFFF	50	16,67	9211	52	13,33	4144
2	5	66	M	S	5	0x1FFFFFFFFF	288	4,00	1420	284	5,33	4091
3	10	66	M	S	10	0x1FFFFFFFFF	587	2,17	715	512	14,67	4127
4	15	66	M	S	15	0x1FFFFFFFFF	869	3,44	821	836	7,11	4211
5	20	66	M	S	20	0x1FFFFFFFFF	1172	2,33	467	1131	5,75	4111
6	25	66	M	S	25	0x1FFFFFFFFF	1460	2,67	300	1361	9,27	4107
7	30	66	M	S	30	0x1FFFFFFFFF	1749	2,83	281	1678	6,78	4299
8	50	66	M	S	30	0x1FFFFFFFFF	2923	2,57	262	1655	8,06	4122
9	1	66	S	M	1	0x1FFFFFFFFF	57	5,00	4098	55	8,33	4229
10	5	66	S	M	5	0x1FFFFFFFFF	291	3,00	1889	286	4,67	4073
11	10	66	S	M	10	0x1FFFFFFFFF	584	2,67	1196	556	7,33	4296
12	15	66	S	M	15	0x1FFFFFFFFF	878	2,44	1225	837	7,00	4164
13	20	66	S	M	20	0x1FFFFFFFFF	1172	2,33	1196	1125	6,25	4202
14	25	66	S	M	25	0x1FFFFFFFFF	1448	3,47	1891	1401	6,60	4480
15	30	66	S	M	30	0x1FFFFFFFFF	1751	2,72	1264	1674	7,00	4419
16	50	66	S	M	30	0x1FFFFFFFFF	2923	2,57	1273	1634	9,22	4183

ANT WiFi test

Test #	Connection interval (Hz)	Frequency	Peripheral type	PC type	Results		
					Correctly transmitted	Loss in %	Discovery time
1	1	46	M	S	42	30,00	7149
2	5	46	M	S	239	20,33	2954
3	10	46	M	S	499	16,83	252
4	15	46	M	S	747	17,00	88
5	20	46	M	S	1004	16,33	454
6	25	46	M	S	1212	19,20	155
7	30	46	M	S	1478	17,89	283
8	50	46	M	S	2496	16,80	293
9	1	46	S	M	44	26,67	6090
10	5	46	S	M	239	20,33	1886
11	10	46	S	M	470	21,67	1491
12	15	46	S	M	727	19,22	2359
13	20	46	S	M	948	21,00	1298
14	25	46	S	M	1175	21,67	2166
15	30	46	S	M	1446	19,67	1327
16	50	46	S	M	2334	22,20	1347

BTLE WiFi Test

Test #	Connection interval (Hz)	Channel map	Results		
			Correctly transmitted	Loss in %	Discovery time
1	1	0x1FFFFFFFFF	57	5,00	1518
2	5	0x1FFFFFFFFF	285	5,00	1450
3	10	0x1FFFFFFFFF	566	5,67	1530
4	15	0x1FFFFFFFFF	853	5,22	1415
5	20	0x1FFFFFFFFF	1127	6,08	1583
6	25	0x1FFFFFFFFF	1403	6,47	1551
7	30	0x1FFFFFFFFF	1658	7,89	1447

ANT+BTLE + WiFi stress test

Test #	ANT setup				BTLE setup		Results				
	Channel period	Frequency	Peripheral type	PC type	Connection interval	Channel map	Correctly transmitted	ANT Loss in %	Discovery time	Correctly transmitted	BTLE Loss in %	Discovery time
1	1	46	M	S	1	0x1FFFFFFFFF	42	30,00	1197	53	11,67	4119
2	5	46	M	S	5	0x1FFFFFFFFF	232	22,67	628	279	7,00	4107
3	10	46	M	S	10	0x1FFFFFFFFF	460	23,33	123	560	6,67	4195
4	15	46	M	S	15	0x1FFFFFFFFF	686	23,78	766	837	7,00	3992
5	20	46	M	S	20	0x1FFFFFFFFF	963	19,75	286	1114	7,17	4454
6	25	46	M	S	25	0x1FFFFFFFFF	1198	20,13	272	1398	6,80	3983
7	30	46	M	S	30	0x1FFFFFFFFF	1400	22,22	145	1654	8,11	4065
8	50	46	M	S	30	0x1FFFFFFFFF	2449	18,37	248	1653	8,17	4368
9	1	46	S	M	1	0x1FFFFFFFFF	48	20,00	3228	54	10,00	4042
10	5	46	S	M	5	0x1FFFFFFFFF	245	18,33	1889	277	7,67	4252
11	10	46	S	M	10	0x1FFFFFFFFF	441	26,50	2074	577	3,83	3972
12	15	46	S	M	15	0x1FFFFFFFFF	727	19,22	1177	838	6,89	4296
13	20	46	S	M	20	0x1FFFFFFFFF	926	22,83	1563	1095	8,75	4217
14	25	46	S	M	25	0x1FFFFFFFFF	1225	18,33	1341	1385	7,67	4206
15	30	46	S	M	30	0x1FFFFFFFFF	1474	18,11	1225	1400	22,22	4369
16	50	46	S	M	30	0x1FFFFFFFFF	2362	21,27	1360	1590	11,67	4389

ANT+BTLE + WiFi optimized

Test #	ANT setup				BTLE setup		Results				
	Channel period	Frequency	Peripheral type	PC type	Connection interval	Channel map	Correctly transmitted	ANT Loss in %	Discovery time	Correctly transmitted	BTLE Loss in %	Discovery time
1	1	66	M	S	1	0x1F1FC007FF	54	10,00	1216	58	3,33	4343
2	5	66	M	S	5	0x1F1FC007FF	281	6,33	813	277	7,67	4554
3	10	66	M	S	10	0x1F1FC007FF	565	5,83	621	520	13,33	4604
4	15	66	M	S	15	0x1F1FC007FF	855	5,00	678	780	13,33	4210
5	20	66	M	S	20	0x1F1FC007FF	1114	7,17	474	984	18,00	4299
6	25	66	M	S	25	0x1F1FC007FF	1439	4,07	98	1246	16,93	4400
7	30	66	M	S	30	0x1F1FC007FF	1729	3,94	89	1630	9,44	4428
8	50	66	M	S	30	0x1F1FC007FF	2858	4,73	281	1630	9,44	4428
9	1	66	S	M	1	0x1F1FC007FF	49	18,33	2092	58	3,33	4148
10	5	66	S	M	5	0x1F1FC007FF	239	20,33	1899	273	9,00	4373
11	10	66	S	M	10	0x1F1FC007FF	484	19,33	1891	542	9,67	4215
12	15	66	S	M	15	0x1F1FC007FF	703	21,89	1167	820	8,89	4689
13	20	66	S	M	20	0x1F1FC007FF	971	19,08	1437	1044	13,00	5082
14	25	66	S	M	25	0x1F1FC007FF	1254	16,40	1138	1380	8,00	4372
15	30	66	S	M	30	0x1F1FC007FF	1462	18,78	1302	1625	9,72	4163
16	50	66	S	M	30	0x1F1FC007FF	2422	19,27	1264	1652	8,22	4162

C lib_ant source code

```
/*
 * lib_ant.c
 *
 * This is the ANT application interface.
 */


#include <string.h>  // memcpy
#include <stdio.h>
#include "uart.h"
#include "gpio.h"
#include "lib_ant.h"
#include "LPC11xx.h"
#include "lib_dispatcher.h"
#include "dispatcher_config.h"
#include "system.h"
#include "hal_io.h"


#define MESG_CHANNEL_OFFSET       3
#define MESG_EVENT_ID_OFFSET      4
#define MESG_EVENT_CODE_OFFSET    5



/*
 * Handler defines. If not defined in dispatcher_config.h, set to a value
 * higher than NB_MAX_MSG to disregard the message.
 */

#ifndef HANDLE_ANT_CMD_RESPONSE_IS_DEFINED
#define HANDLE_ANT_CMD_RESPONSE                     NB_MAX_MSG + 20
#endif
#ifndef HANDLE_ANT_CHANNEL_EVENT_IS_DEFINED
#define HANDLE_ANT_CHANNEL_EVENT                    NB_MAX_MSG + 50
#endif
#ifndef HANDLE_ANT_EVENT_RX_BROADCAST_IS_DEFINED
#define HANDLE_ANT_EVENT_RX_BROADCAST               NB_MAX_MSG + 21
#endif
#ifndef HANDLE_ANT_EVENT_RX_ACKNOWLEDGED_IS_DEFINED
#define HANDLE_ANT_EVENT_RX_ACKNOWLEDGED            NB_MAX_MSG + 22
#endif
#ifndef HANDLE_ANT_EVENT_RX_BURST_PACKET_IS_DEFINED
#define HANDLE_ANT_EVENT_RX_BURST_PACKET            NB_MAX_MSG + 23
#endif
#ifndef HANDLE_ANT_EVENT_RX_EXT_BROADCAST_IS_DEFINED
#define HANDLE_ANT_EVENT_RX_EXT_BROADCAST           NB_MAX_MSG + 24
#endif
#ifndef HANDLE_ANT_EVENT_RX_EXT_ACKNOWLEDGED_IS_DEFINED
#define HANDLE_ANT_EVENT_RX_EXT_ACKNOWLEDGED        NB_MAX_MSG + 25
#endif
#ifndef HANDLE_ANT_EVENT_RX_EXT_BURST_PACKET_IS_DEFINED
#define HANDLE_ANT_EVENT_RX_EXT_BURST_PACKET        NB_MAX_MSG + 26
#endif
#ifndef HANDLE_ANT_EVENT_RX_RSSI_BROADCAST_IS_DEFINED
#define HANDLE_ANT_EVENT_RX_RSSI_BROADCAST          NB_MAX_MSG + 27
#endif
#ifndef HANDLE_ANT_EVENT_RX_RSSI_ACKNOWLEDGED_IS_DEFINED
#define HANDLE_ANT_EVENT_RX_RSSI_ACKNOWLEDGED       NB_MAX_MSG + 28
#endif
#ifndef HANDLE_ANT_EVENT_RX_RSSI_BURST_PACKET_IS_DEFINED
#define HANDLE_ANT_EVENT_RX_RSSI_BURST_PACKET       NB_MAX_MSG + 29
#endif
#ifndef HANDLE_ANT_EVENT_RX_BTH_BROADCAST_IS_DEFINED
#define HANDLE_ANT_EVENT_RX_BTH_BROADCAST           NB_MAX_MSG + 30
#endif
#ifndef HANDLE_ANT_EVENT_RX_BTH_ACKNOWLEDGED_IS_DEFINED
#define HANDLE_ANT_EVENT_RX_BTH_ACKNOWLEDGED        NB_MAX_MSG + 31
#endif
#ifndef HANDLE_ANT_EVENT_RX_BTH_BURST_PACKET_IS_DEFINED
#define HANDLE_ANT_EVENT_RX_BTH_BURST_PACKET        NB_MAX_MSG + 32
#endif
#ifndef HANDLE_ANT_EVENT_RX_BTH_EXT_BROADCAST_IS_DEFINED
#define HANDLE_ANT_EVENT_RX_BTH_EXT_BROADCAST       NB_MAX_MSG + 33
#endif
#ifndef HANDLE_ANT_EVENT_RX_BTH_EXT_ACKNOWLEDGED_IS_DEFINED
#define HANDLE_ANT_EVENT_RX_BTH_EXT_ACKNOWLEDGED    NB_MAX_MSG + 34
#endif
#ifndef HANDLE_ANT_EVENT_RX_BTH_EXT_BURST_PACKET_IS_DEFINED
#define HANDLE_ANT_EVENT_RX_BTH_EXT_BURST_PACKET    NB_MAX_MSG + 35
#endif
#ifndef HANDLE_ANT_MSG_NVM_DATA_ID_IS_DEFINED
#define HANDLE_ANT_MSG_NVM_DATA_ID                  NB_MAX_MSG + 36
```

```
#endif
#ifndef HANDLE_ANT_MESG_NVM_CMD_ID_IS_DEFINED
#define HANDLE_ANT_MESG_NVM_CMD_ID                  NB_MAX_MSG + 37
#endif
#ifndef HANDLE_ANT_MESG_INVALID_ID_IS_DEFINED
#define HANDLE_ANT_MESG_INVALID_ID                  NB_MAX_MSG + 38
#endif
#ifndef HANDLE_ANT_SYSTEM_STARTUP_IS_DEFINED
#define HANDLE_ANT_SYSTEM_STARTUP                   NB_MAX_MSG + 39
#endif
#ifndef HANDLE_ANT_EVENT_TX_IS_DEFINED
#define HANDLE_ANT_EVENT_TX                         NB_MAX_MSG + 40
#endif
#ifndef HANDLE_ANT_EVENT_RX_FAIL_IS_DEFINED
#define HANDLE_ANT_EVENT_RX_FAIL                    NB_MAX_MSG + 41
#endif
#ifndef HANDLE_ANT_EVENT_RX_SEARCH_TIMEOUT_IS_DEFINED
#define HANDLE_ANT_EVENT_RX_SEARCH_TIMEOUT          NB_MAX_MSG + 42
#endif
#ifndef HANDLE_ANT_EVENT_CHANNEL_CLOSED_IS_DEFINED
#define HANDLE_ANT_EVENT_CHANNEL_CLOSED             NB_MAX_MSG + 43
#endif
#ifndef HANDLE_ANT_UNEXPECTED_EVENT_IS_DEFINED
#define HANDLE_ANT_UNEXPECTED_EVENT                 NB_MAX_MSG + 44
#endif
#ifndef HANDLE_ANT_EVENT_RX_ACKNOWLEDGED_IS_DEFINED
#define HANDLE_ANT_EVENT_RX_ACKNOWLEDGED            NB_MAX_MSG + 45
#endif
#ifndef HANDLE_ANT_EVENT_TRANSFER_TX_COMPLETED_IS_DEFINED
#define HANDLE_ANT_EVENT_TRANSFER_TX_COMPLETED      NB_MAX_MSG + 46
#endif
#ifndef HANDLE_ANT_EVENT_TRANSFER_TX_FAILED_IS_DEFINED
#define HANDLE_ANT_EVENT_TRANSFER_TX_FAILED         NB_MAX_MSG + 47
#endif


#define MAX_CHANNELS          ((uint8_t)8)

#define MAX_BURST_MSG_SIZE    256
static uint8_t burst_rx_buffer[MAX_CHANNELS][MAX_BURST_MSG_SIZE + 1];
static uint8_t burst_rx_buffer_index[MAX_CHANNELS];

static uint8_t channel_rx_buffers[MAX_CHANNELS][MESG_DATA_SIZE];
static uint8_t channel_event_buffers[MAX_CHANNELS][MESG_DATA_SIZE];
static uint8_t cmd_response_buffer[MESG_DATA_SIZE];

static volatile bool is_transaction_finished = true;    // Set this to false if a reply is expected from the ANT
       device


/*
 * Initialize the ANT library
 */
bool ANT_Init(ushort usBaudRate)
{
   uint8_t br[3];
   uint8_t i;

   for (i = 0; i < MAX_CHANNELS; i++)
   {
      burst_rx_buffer_index[i] = 0;
   }


   switch (usBaudRate)
   {
   case 1200:
      br[0] = 0;
      br[1] = 0;
      br[2] = 1;
      break;

   case 2400:
      br[0] = 0;
      br[1] = 1;
      br[2] = 1;
      break;

   case 4800:
      br[0] = 0;
      br[1] = 0;
      br[2] = 0;
      break;

   case 9600:
      br[0] = 1;
```

```
      br[1] = 0;
      br[2] = 1;
      break;

   case 19200:
      br[0] = 0;
      br[1] = 1;
      br[2] = 0;
      break;

   case 38400:
      br[0] = 1;
      br[1] = 0;
      br[2] = 0;
      break;

   case 50000:
      br[0] = 1;
      br[1] = 1;
      br[2] = 0;
      break;

   case 57600:
      br[0] = 1;
      br[1] = 1;
      br[2] = 1;
      break;

   default:
      return(false);
   }

   GPIOInit();

   GPIOSetDir(PORT2, 4, 1);         //BR1/SFLOW
   GPIOSetValue(2, 4, br[0]);

   GPIOSetDir(PORT2, 1, 1);         //BR2/SSCK BAUD 19200
   GPIOSetValue(2, 1, br[1]);

   GPIOSetDir(PORT2, 0, 1);         //BR3
   GPIOSetValue(2, 0, br[2]);


   GPIOSetDir(PORT2, 5, 1);         //SLEEP/MRDY
   GPIOSetValue(2, 5, 0);

   GPIOSetDir(PORT2, 6, 1);         //!SUSPEND/SRDY
   GPIOSetValue(2, 6, 1);

   GPIOSetDir(PORT2, 7, 1);         // !RESET

   UARTInit((uint32_t)usBaudRate);

   return(true);
}


static bool SendMessage(uint8_t ucID, uint8_t ucSize, uint8_t ucByte0, uint8_t *pucData)
{
   uint8_t pucOutBuf[ucSize + 4];

   pucOutBuf[0] = MESG_TX_SYNC;
   pucOutBuf[1] = ucSize;
   pucOutBuf[2] = ucID;
   pucOutBuf[3] = ucByte0;
   if ((pucData != NULL) && (ucSize > 1))
   {
      memcpy(&pucOutBuf[4], pucData, ucSize - 1);
   }

   pucOutBuf[ucSize + 3] = return_XOR(pucOutBuf, ucSize + 3);
   while (!is_transaction_finished)
   {
      // wait for other transactions
   }
   UARTSend(pucOutBuf, ucSize + 4);
   return(true);
}


bool ANT_SetChannelSearchTimeout(uint8_t ucANTChannel, uint8_t ucSearchTimeout)
{
   bool out = SendMessage(MESG_CHANNEL_SEARCH_TIMEOUT_ID, MESG_CHANNEL_SEARCH_TIMEOUT_SIZE, ucANTChannel, &
         ucSearchTimeout);
   is_transaction_finished = false;
```

```
   return(out);
}


bool ANT_SetChannelLowPrioritySearchTimeout(uint8_t ucANTChannel, uint8_t ucSearchTimeout)
{
   bool out = SendMessage(MESG_SET_LP_SEARCH_TIMEOUT_ID, MESG_SET_LP_SEARCH_TIMEOUT_SIZE, ucANTChannel, &
         ucSearchTimeout);

   is_transaction_finished = false;
   return(out);
}


bool ANT_AssignChannel(uint8_t ucANTChannel, uint8_t ucParam, uint8_t ucNetNumber)
{
   uint8_t aucData[2];

   aucData[0] = ucParam;
   aucData[1] = ucNetNumber;

   bool out = SendMessage(MESG_ASSIGN_CHANNEL_ID, MESG_ASSIGN_CHANNEL_SIZE, ucANTChannel, &aucData[0]);
   is_transaction_finished = false;
   return(out);
}


bool ANT_AssignChannelExt(uint8_t ucAntChannel, uint8_t chan_type, uint8_t network, uint8_t ext)
{
   uint8_t aucData[3];

   aucData[0] = chan_type;
   aucData[1] = network;
   aucData[2] = ext;

   bool out = SendMessage(MESG_ASSIGN_CHANNEL_ID, MESG_ASSIGN_CHANNEL_SIZE + 1, ucAntChannel, &aucData[0]);
   is_transaction_finished = false;
   return(out);
}


bool ANT_UnAssignChannel(uint8_t ucANTChannel)
{
   bool out = SendMessage(MESG_UNASSIGN_CHANNEL_ID, MESG_UNASSIGN_CHANNEL_SIZE, ucANTChannel, NULL);

   is_transaction_finished = false;
   return(out);
}


bool ANT_ResetSystem(void)
{
   // no reply expected
   return(SendMessage(MESG_SYSTEM_RESET_ID, MESG_SYSTEM_RESET_SIZE, 0, NULL));
}


bool ANT_RequestMessage(uint8_t ucANTChannel, uint8_t ucMessageID)
{
   bool out = SendMessage(MESG_REQUEST_ID, MESG_REQUEST_SIZE, ucANTChannel, &ucMessageID);

   is_transaction_finished = false;
   return(out);
}


bool ANT_SetChannelId(uint8_t ucANTChannel, ushort usDeviceNumber, uint8_t ucDeviceType, uint8_t ucManufactureID
      )
{
   uint8_t aucData[4];

   aucData[0] = (uint8_t)(usDeviceNumber & 0xFF);
   aucData[1] = (uint8_t)((usDeviceNumber >> 8) & 0xFF);
   aucData[2] = ucDeviceType;
   aucData[3] = ucManufactureID;

   bool out = SendMessage(MESG_CHANNEL_ID_ID, MESG_CHANNEL_ID_SIZE, ucANTChannel, &aucData[0]);
   is_transaction_finished = false;
   return(out);
}


bool ANT_SetChannelPeriod(uint8_t ucANTChannel, ushort usMesgPeriod)
{
   uint8_t aucData[2];
```

```
   aucData[0] = (uint8_t)(usMesgPeriod & 0xFF);
   aucData[1] = (uint8_t)((usMesgPeriod >> 8) & 0xFF);

   bool out = SendMessage(MESG_CHANNEL_MESG_PERIOD_ID, MESG_CHANNEL_MESG_PERIOD_SIZE, ucANTChannel, &aucData[0])
         ;
   is_transaction_finished = false;
   return(out);
}


bool ANT_SetChannelRFFreq(uint8_t ucANTChannel, uint8_t ucRFFreq)
{
   bool out = SendMessage(MESG_CHANNEL_RADIO_FREQ_ID, MESG_CHANNEL_RADIO_FREQ_SIZE, ucANTChannel, &ucRFFreq);

   is_transaction_finished = false;
   return(out);
}


bool ANT_SetFrequencyAgilitySettings(uint8_t ant_channel, uint8_t *freq_settings)
{
   bool out = SendMessage(MESG_CONFIG_FREQ_AGILITY_ID, MESG_CONFIG_FREQ_AGILITY_SIZE, ant_channel, freq_settings
         );

   is_transaction_finished = false;
   return(out);
}


bool ANT_SetNetworkKey(uint8_t ucNetNumber, uint8_t *pucKey)
{
   bool out = SendMessage(MESG_NETWORK_KEY_ID, MESG_NETWORK_KEY_SIZE, ucNetNumber, pucKey);

   is_transaction_finished = false;
   return(out);
}


bool ANT_SetTransmitPower(uint8_t ucTransmitPower)
{
   bool out = SendMessage(MESG_RADIO_TX_POWER_ID, MESG_RADIO_TX_POWER_SIZE, 0, &ucTransmitPower);

   is_transaction_finished = false;
   return(out);
}


bool ANT_OpenChannel(uint8_t ucANTChannel)
{
   bool out = SendMessage(MESG_OPEN_CHANNEL_ID, MESG_OPEN_CHANNEL_SIZE, ucANTChannel, NULL);

   is_transaction_finished = false;
   return(out);
}


bool ANT_CloseChannel(uint8_t ucANTChannel)
{
   bool out = SendMessage(MESG_CLOSE_CHANNEL_ID, MESG_CLOSE_CHANNEL_SIZE, ucANTChannel, NULL);

   is_transaction_finished = false;
   return(out);
}


bool ANT_InitCWTestMode(void)
{
   bool out = SendMessage(MESG_RADIO_CW_INIT_ID, MESG_RADIO_CW_INIT_SIZE, 0, NULL);

   is_transaction_finished = false;
   return(out);
}


bool ANT_SetCWTestMode(uint8_t ucTransmitPower, uint8_t ucRFChannel)
{
   uint8_t aucData[2];

   aucData[0] = ucTransmitPower;
   aucData[1] = ucRFChannel;

   bool out = SendMessage(MESG_RADIO_CW_MODE_ID, MESG_RADIO_CW_MODE_SIZE, 0, &aucData[0]);
   is_transaction_finished = false;
   return(out);
}
```

```
bool ANT_SendBroadcastData(uint8_t ucANTChannel, uint8_t *pucData)
{
   // no reply expected
   return(SendMessage(MESG_BROADCAST_DATA_ID, MESG_DATA_SIZE, ucANTChannel, pucData));
}


bool ANT_SendAcknowledgedData(uint8_t ucANTChannel, uint8_t *pucData)
{
   // no reply expected
   return(SendMessage(MESG_ACKNOWLEDGED_DATA_ID, MESG_DATA_SIZE, ucANTChannel, pucData));
}


bool ANT_SendBurstTransferPacket(uint8_t channel, uint8_t *data, uint8_t size)
{
   // no reply expected
   uint8_t sequence = 0x00;
   uint8_t seq_chan;
   uint8_t i;

   for (i = 0; i < size; i += MESG_DATA_SIZE)
   {
      seq_chan = sequence | channel;
      if (!SendMessage(MESG_BURST_DATA_ID, (i + MESG_DATA_SIZE > size) ? size - i : MESG_DATA_SIZE, seq_chan, &
            data[i]))
      {
         return(false);
      }

      if (sequence == SEQUENCE_NUMBER_ROLLOVER)
      {
         sequence = SEQUENCE_NUMBER_INC;
      }
      else
      {
         sequence += SEQUENCE_NUMBER_INC;
      }
      if ((i + 2 * MESG_DATA_SIZE) >= size)
      {
         // next run is last
         sequence |= SEQUENCE_LAST_MESSAGE;
      }
   }
   return(true);
}


/*
 * This function is called by the UART ISR when a new message has arrived.
 * Interpret, and post a message in the dispatcher if the handler is defined.
 */
void ANT_SerialHaveMessage(uint8_t *received_data)
{
   uint8_t size;
   uint8_t id;
   uint8_t channel_nr;
   uint8_t sequence_nr;

   if (!checkMsg(received_data))
   {
      ENABLE_UART_IRQ();
      return;
   }


   size        = received_data[MESG_SIZE_OFFSET];
   id          = received_data[MESG_ID_OFFSET];
   channel_nr  = received_data[MESG_CHANNEL_OFFSET] & CHANNEL_NUMBER_MASK;
   sequence_nr = received_data[MESG_CHANNEL_OFFSET] & SEQUENCE_NUMBER_MASK;

   uint8_t msg_handle     = NB_MAX_MSG;
   uint8_t *handle_buffer = NULL;

   switch (id)
   {
   case MESG_RESPONSE_EVENT_ID:
      if (received_data[MESG_EVENT_ID_OFFSET] != MESG_EVENT_ID)
      {
         // this is a command response
         handle_buffer = &cmd_response_buffer[0];
         msg_handle    = HANDLE_ANT_CMD_RESPONSE;
      }

      else
```

```
   {
      // this is a channel event
      handle_buffer = &channel_event_buffers[channel_nr][0];
      switch (received_data[MESG_EVENT_CODE_OFFSET])
      {
      case EVENT_TX:
         msg_handle = HANDLE_ANT_EVENT_TX;
         break;

      case EVENT_RX_FAIL:
         msg_handle = HANDLE_ANT_EVENT_RX_FAIL;
         break;

      case EVENT_RX_SEARCH_TIMEOUT:
         msg_handle = HANDLE_ANT_EVENT_RX_SEARCH_TIMEOUT;
         break;

      case EVENT_CHANNEL_CLOSED:
         msg_handle = HANDLE_ANT_EVENT_CHANNEL_CLOSED;
         break;

      case EVENT_TRANSFER_TX_COMPLETED:
         msg_handle = HANDLE_ANT_EVENT_TRANSFER_TX_COMPLETED;
         break;

      case EVENT_TRANSFER_TX_FAILED:
         msg_handle = HANDLE_ANT_EVENT_TRANSFER_TX_FAILED;
         break;

      default:
         msg_handle = HANDLE_ANT_UNEXPECTED_EVENT;
         break;
      }
   }
   break;

// other events
case MESG_BROADCAST_DATA_ID:
   handle_buffer = &channel_rx_buffers[channel_nr][0];
   msg_handle    = HANDLE_ANT_EVENT_RX_BROADCAST;
   break;

case MESG_ACKNOWLEDGED_DATA_ID:
   handle_buffer = &channel_rx_buffers[channel_nr][0];
   msg_handle    = HANDLE_ANT_EVENT_RX_ACKNOWLEDGED;
   break;

case MESG_BURST_DATA_ID:
   if (burst_rx_buffer_index[channel_nr] + size < MAX_BURST_MSG_SIZE)
   {
      memcpy(&burst_rx_buffer[channel_nr][burst_rx_buffer_index[channel_nr]], &received_data[MESG_DATA_OFFSET
            + 1], size - 1);
      burst_rx_buffer_index[channel_nr] += size - 1;
   }
   if ((sequence_nr & SEQUENCE_LAST_MESSAGE) == 0x80)
   {
      // This is the last burst packet
      handle_buffer = &burst_rx_buffer[channel_nr][0];
      msg_handle    = HANDLE_ANT_EVENT_RX_BURST_PACKET;
   }
   else
   {
      // Do not wake application until entire message is received
      msg_handle = NB_MAX_MSG;
   }
   break;

case MESG_SYSTEM_STARTUP:
   handle_buffer = &cmd_response_buffer[0];
   msg_handle    = HANDLE_ANT_SYSTEM_STARTUP;
   break;

default:
   handle_buffer = &cmd_response_buffer[0];
   msg_handle    = HANDLE_ANT_MESG_INVALID_ID;
   break;
}
if (msg_handle < NB_MAX_MSG)
{
   // If the handle exists, post a message in the dispatcher
   memcpy(handle_buffer, &received_data[MESG_DATA_OFFSET], size);
   lib_dispatcher_post_msg(msg_handle, handle_buffer, size, HIGH_PRIORITY);
}

// re-enable transactions
if (!is_transaction_finished)
```

```
   {
      is_transaction_finished = true;
   }
   ENABLE_UART_IRQ();
}

/*
 * Helper function. Return the message checksum.
 * Used in ANT_SendMessage().
 */
uint8_t return_XOR(uint8_t *array, uint8_t length)
{
   uint8_t i;
   uint8_t xor_return = 0;

   for (i = 0; i < length; i++)
   {
      xor_return ^= *(array + i);
   }

   return(xor_return);
}

/*
 * Helper function. Checks received message integrity. Used in
 * ANT_SerialHaveMessage().
 */
bool checkMsg(uint8_t *msg)
{
   if ((msg[0] != MESG_TX_SYNC) && (msg[0] != MESG_RX_SYNC))
   {
      return(false);
   }
   if (msg[msg[MESG_SIZE_OFFSET] + MESG_HEADER_SIZE] != return_XOR(msg, msg[MESG_SIZE_OFFSET] + MESG_HEADER_SIZE
       ))
   {
      return(false);
   }
   return(true);
}
```

D Test application source code

D.1 Motherboard application

```
/*
 * counter_app_ant.c
 *
 *  This is the ANT application code.
 *
 */
#include <stdio.h>  // printf/puts
#include <string.h> // strncmp
#include "lib_ant.h"
#include "lib_dispatcher.h"
#include "counter_app_ant.h"
#include "LPC11xx.h"
#include "system.h"
#include "timer32.h"
#include "hal_io.h"

typedef enum
{
   ANT_IDLE = 0,
   ANT_SETUP_SET_CHANNEL_ID,
   ANT_SETUP_SET_CHANNEL_PERIOD,
   ANT_SETUP_SET_CHANNEL_SEARCH_TIMEOUT,
   ANT_SETUP_SET_TX_POWER,
   ANT_SETUP_SET_RADIO_FREQUENCY,
   ANT_SETUP_OPEN_CHANNEL,
   ANT_RUN,
   ANT_COMPLETE,
   ANT_FAILURE
} ant_state_t;



ant_state_t state                        = ANT_IDLE;
ant_state_t next_state;


static uint8_t           send_buffer[MESG_DATA_SIZE - 1];
static volatile uint8_t  counter         = 0;
static volatile uint32_t sent_packets    = 0, retransmitted_tx_packets = 0;
static volatile uint32_t received_packets = 0, retransmitted_rx_packets = 0;
static volatile uint32_t lost_packets    = 0, invalid_counter_count = 0;
static volatile uint32_t discovery_time  = 0;
static volatile bool     test_timed_out  = false, received_first_packet = false;


void on_process_ant_counter_app(uint8_t size, uint8_t *buffer)
{
   static bool lp_search_timeout_set = false;

   if (strncmp((const char *)&buffer[0], "reset", 5) == 0)
   {
      reset_ant_radio();
      ANT_ResetSystem();
      state = ANT_IDLE;
   }
   else
   {
      switch (state)
      {
      case ANT_IDLE:
         if (strncmp((const char *)buffer, "start", 5) == 0)
         {
            puts("Starting␣ANT␣with:");
            printf("\tChannel␣type:␣%s\n", (ant_setup.channel_type == PARAMETER_TX_NOT_RX) ? "master" : "slave")
                  ;
            printf("\tPeriod:␣%d␣Hz\n", ant_setup.channel_period);
            printf("\tTx␣Power:␣%d\n", ant_setup.tx_power);
            printf("\tRadio␣freqs:␣%d␣%d␣%d\n", ant_setup.frequencies[0], ant_setup.frequencies[1], ant_setup.
                  frequencies[2]);
            printf("\tFreq␣agility:␣%s\n", (ant_setup.frequency_agility_enabled == true) ? "enabled" : "disabled
                  ");

            if (ant_setup.frequency_agility_enabled)
            {
               ANT_AssignChannelExt(ANT_CHANNEL, ant_setup.channel_type, ANT_NET, 0x04);
            }
            else
            {
```

```
            ANT_AssignChannel(ANT_CHANNEL, ant_setup.channel_type, ANT_NET);
         }

         next_state = ANT_SETUP_SET_CHANNEL_ID;
      }
      break;

   case ANT_SETUP_SET_CHANNEL_ID:
      ANT_SetChannelId(ANT_CHANNEL, ANT_DEVICE_NO, ANT_DEVICE_TYPE, ANT_MANUFACTURE_ID);
      next_state = ANT_SETUP_SET_CHANNEL_PERIOD;
      break;

   case ANT_SETUP_SET_CHANNEL_PERIOD:
      ANT_SetChannelPeriod(ANT_CHANNEL, CLOCK_FREQUENCY / ant_setup.channel_period);
      next_state = ANT_SETUP_SET_CHANNEL_SEARCH_TIMEOUT;
      break;

   case ANT_SETUP_SET_CHANNEL_SEARCH_TIMEOUT:
      if (!lp_search_timeout_set)
      {
         ANT_SetChannelLowPrioritySearchTimeout(ANT_CHANNEL, 0);
         lp_search_timeout_set = true;
         next_state            = ANT_SETUP_SET_CHANNEL_SEARCH_TIMEOUT;
      }
      else
      {
         ANT_SetChannelSearchTimeout(ANT_CHANNEL, ant_setup.search_timeout);
         lp_search_timeout_set = false;
         next_state            = ANT_SETUP_SET_TX_POWER;
      }
      break;

   case ANT_SETUP_SET_TX_POWER:
      ANT_SetTransmitPower(ant_setup.tx_power);
      next_state = ANT_SETUP_SET_RADIO_FREQUENCY;
      break;

   case ANT_SETUP_SET_RADIO_FREQUENCY:
      if (ant_setup.frequency_agility_enabled)
      {
         ANT_SetFrequencyAgilitySettings(ANT_CHANNEL, &ant_setup.frequencies[0]);
      }
      else
      {
         ANT_SetChannelRFFreq(ANT_CHANNEL, ant_setup.frequencies[0]);
      }
      next_state = ANT_SETUP_OPEN_CHANNEL;
      break;

   case ANT_SETUP_OPEN_CHANNEL:
      ANT_OpenChannel(ANT_CHANNEL);
      next_state = ANT_RUN;
      break;

   case ANT_RUN:
      next_state = ANT_RUN;
      break;

   case ANT_COMPLETE:
      puts("\n================================================\nANT test results:");
      printf("\tSent packets: %d\n", sent_packets);
      printf("\tReceived packets: %d\n", received_packets);
      printf("\tLost packets: %d\n", lost_packets);
      printf("\tRetransmitted rx packets: %d\n", retransmitted_rx_packets);
      printf("\tRetransmitted tx packets: %d\n", retransmitted_tx_packets);
      printf("\tPackets with invalid counter: %d\n", invalid_counter_count);
      printf("\tDiscovery time: %d\n", discovery_time);
      puts("================================================\n");

      counter                  = 0;
      received_packets         = 0;
      lost_packets             = 0;
      sent_packets             = 0;
      retransmitted_tx_packets = 0;
      retransmitted_rx_packets = 0;
      invalid_counter_count    = 0;
      received_first_packet    = false;
      discovery_time           = 0;
      state = ANT_IDLE;
      break;

   case ANT_FAILURE:
      puts("ANT failed\n");
      break;

   default:
```

```
         printf("ant␣invalid␣state:␣%d\n", state);
         break;
      }
   }
}


void on_ant_cmd_response_event(uint8_t size, uint8_t *buffer)
{
   if (buffer[2] != RESPONSE_NO_ERROR)
   {
      printf("ANT␣cmd␣0x%X␣failed␣with␣code␣0x%X.␣Timestamp:␣%d␣ms\n", buffer[1], buffer[2], timer32_get_time())
          ;
      //next_state = ANT_FAILURE;
   }

   switch (buffer[1])
   {
   case MESG_UNASSIGN_CHANNEL_ID:
      next_state = ANT_COMPLETE;
      break;

   default:
      break;
   }
   state = next_state;
   lib_dispatcher_post_msg(HANDLE_PROCESS_ANT_COUNTER_APP, NULL, 0, NORMAL_PRIORITY);

}


void on_ant_event_tx(uint8_t size, uint8_t *buffer)
{

   static uint8_t toggle = 0;

   HAL_IO_SET_STATE(HAL_IO_LED2, toggle);
   toggle = !toggle;

   if (ant_setup.frequency_agility_enabled == false)
   {
      sent_packets++;
      send_buffer[0] = counter++;
      ANT_SendBroadcastData(ANT_CHANNEL, &send_buffer[0]);
   }
}


void on_ant_event_transfer_tx_completed(uint8_t size, uint8_t *buffer)
{
   static uint8_t toggle = 0;

   HAL_IO_SET_STATE(HAL_IO_LED3, toggle);
   toggle          = !toggle;
   send_buffer[0] = counter++;
   ANT_SendAcknowledgedData(ANT_CHANNEL, &send_buffer[0]);
   sent_packets++;
}


void on_ant_event_transfer_tx_failed(uint8_t size, uint8_t *buffer)
{
   retransmitted_tx_packets++;
}


void on_ant_event_rx_broadcast(uint8_t size, uint8_t *buffer)
{
   // When not using freq agility, this will process received data
   uint8_t counter_diff = buffer[1] - counter;

   if (!received_first_packet)
   {
      discovery_time = timer32_get_time();
   }

   if ((counter_diff == 0) && received_first_packet)
   {
      retransmitted_rx_packets++;
   }
   else if (counter_diff == 1)
   {
      received_packets++;
   }
   else
   {
```

```
      invalid_counter_count++;
   }
   counter = buffer[1];
   received_first_packet = true;
}


void on_ant_event_rx_acknowledged(uint8_t size, uint8_t *buffer)
{
   // When using freq agility, this method will process data
   uint8_t counter_diff = buffer[1] - counter;

   if (!received_first_packet)
   {
      discovery_time = timer32_get_time();
   }

   if ((counter_diff == 0) && received_first_packet)
   {
      retransmitted_rx_packets++;
   }
   else if (counter_diff == 1)
   {
      received_packets++;
   }
   else
   {
      invalid_counter_count++;
   }
   counter = buffer[1];
   received_first_packet = true;
}


void on_ant_event_rx_fail(uint8_t size, uint8_t *buffer)
{
   printf("ANT␣rx␣fail␣at␣%d␣ms\n", timer32_get_time());
   lost_packets++;
}


void on_ant_event_search_timeout(uint8_t size, uint8_t *buffer)
{
   puts("ANT␣rx␣search␣timeout\n");
   // Channel is closed automatically
}


void on_ant_event_channel_closed(uint8_t size, uint8_t *buffer)
{
   if (test_timed_out || !ant_setup.frequency_agility_enabled)
   {
      puts("ANT␣channel␣closed.␣Unassigning␣channel..\n");
      test_timed_out = false;
      ANT_UnAssignChannel(ANT_CHANNEL);
   }
   else
   {
      // When using freq agility and the node is the slave node, channel closed events
      // will occur when the frequency is changed.
      puts("ANT␣changed␣frequency\n");
   }
}


void on_ant_unexpected_event(uint8_t size, uint8_t *buffer)
{
   printf("ANT␣unknown␣event:␣0x%X␣0x%X\n", buffer[1], buffer[2]);
}


void on_ant_mesg_invalid_id(uint8_t size, uint8_t *buffer)
{
   uint8_t i;

   puts("ANT␣Invalid␣id:␣");
   for (i = 0; i < size; i++)
   {
      printf("0x%X␣", buffer[i]);
   }
   puts("\n");
}


void on_ant_system_startup(uint8_t size, uint8_t *buffer)
{
```

```
   puts("ANT␣device␣started.\n");
   counter                  = 0;
   received_packets         = 0;
   lost_packets             = 0;
   sent_packets             = 0;
   retransmitted_tx_packets = 0;
   retransmitted_rx_packets = 0;
   received_first_packet    = false;
   discovery_time           = 0;
   state      = ANT_IDLE;
   next_state = ANT_IDLE;
   //lib_dispatcher_post_msg(HANDLE_PROCESS_ANT_COUNTER_APP, NULL, 0, NORMAL_PRIORITY);
}


void on_ant_timer_interrupt(uint8_t size, uint8_t *buffer)
{
   if (state == ANT_RUN)
   {
      test_timed_out = true;
      ANT_CloseChannel(ANT_CHANNEL);
   }
}


/*
 * counter_app_btle.c
 *
 *  This is the BLE application code.
 *
 */


#include <stdio.h>
#include <string.h>

#include "hal_io.h"
#include "lib_aci.h"
#include "lib_dispatcher.h"
#include "system.h"
#include "services.h"
#include "counter_app_btle.h"
#include "bool.h"
#include "timer32.h"

static volatile btle_state_t app_state, next_app_state;
static uint8_t btle_counter = 0;
static volatile uint32_t sent_packets = 0;
static uint32_t discovery_time = 0;
static volatile bool connected = false;

void on_transaction_finished(uint8_t size, uint8_t* buffer)
{
   app_state = next_app_state;
   lib_dispatcher_post_msg(HANDLE_PROCESS_COUNTER_APP, NULL, 0, NORMAL_PRIORITY);
}


void on_process_counter_app(uint8_t size, uint8_t* buffer)
{
   static bool connect_cmd_sent = false;

   if (strncmp((const char*)&buffer[0], "reset", 5) == 0)
   {
      puts("BTLE␣reset\n");
      lib_aci_radio_reset();
   }
   else
   {
      switch(app_state)
      {
      case COUNTER_APP_SEND_CONFIG_UPLOAD :
         next_app_state = COUNTER_APP_SEND_CONFIG_UPLOAD;
         if (lib_aci_send_setup_msg())
         {
            next_app_state = COUNTER_APP_WAIT_STDBY;
         }
         break;
      case COUNTER_APP_WAIT_STDBY :
         next_app_state = COUNTER_APP_WAIT_STDBY;
         if (!lib_aci_is_host_setup_complete())
         {
            lib_aci_send_setup_msg();
         }
         break;
      case COUNTER_APP_IDLE :
         if (strncmp((const char*)&buffer[0], "start", 5) == 0)
```

```
         {
            printf("Starting␣BTLE␣with:␣\n\tTx␣Power:␣%d\n", btle_setup.tx_power);
            printf("\tAdv␣timeout:␣%d\n", btle_setup.adv_timeout);

            app_state = COUNTER_APP_SET_TX_POWER;
            lib_dispatcher_post_msg(HANDLE_PROCESS_COUNTER_APP, NULL, 0, NORMAL_PRIORITY);
         }

         break;
      case COUNTER_APP_SET_TX_POWER :
         lib_aci_set_radio_tx_power(btle_setup.tx_power);
         connect_cmd_sent = false;
         next_app_state = COUNTER_APP_CONNECT;
         break;
      case COUNTER_APP_CONNECT :
         if (!connect_cmd_sent)
         {
            puts("BTLE␣connecting..\n");
            connect_cmd_sent = lib_aci_connect(btle_setup.adv_timeout, 160); //  160*0.625 ms = 100 ms adv
                  interval
            //connect_cmd_sent = lib_aci_connect(0, 160);
         }

         next_app_state = COUNTER_APP_CONNECT;
         break;
      case COUNTER_APP_RUN :
         //services_update_pipes();
         next_app_state = COUNTER_APP_RUN;
         break;
      default :
         printf("wrong␣app␣state:␣%d\n", app_state);
         break;
      }
   }
   lib_aci_enable_transmission();
}

void on_radio_started_stdby(uint8_t size, uint8_t* buffer)
{
   puts("BTLE␣started␣in␣standby␣mode\n");
   app_state = COUNTER_APP_IDLE;
   //lib_dispatcher_post_msg(HANDLE_PROCESS_COUNTER_APP, "start", 5, NORMAL_PRIORITY);
   lib_aci_enable_transmission();
}



void on_radio_started_setup(uint8_t size, uint8_t* buffer)
{

   puts("BTLE␣started␣in␣setup␣mode\n");
   app_state = COUNTER_APP_SEND_CONFIG_UPLOAD;
   lib_dispatcher_post_msg(HANDLE_PROCESS_COUNTER_APP, NULL, 0, NORMAL_PRIORITY);
   lib_aci_enable_transmission();
}


void on_radio_connected (uint8_t size, uint8_t* buffer)
{
   aci_evt_params_connected_t addr;
   lib_aci_interpret_evt_connected(&addr);
   printf("BTLE␣connected␣to␣0x%X%X%X%X%X%X\nConnection␣interval:␣%d␣ms\n", addr.dev_addr[5], addr.dev_addr[4],
         addr.dev_addr[3], addr.dev_addr[2], addr.dev_addr[1], addr.dev_addr[0], addr.conn_rf_interval*5/4);
   discovery_time = timer32_get_time();

   HAL_IO_SET_STATE(HAL_IO_LED0, 0);
   HAL_IO_SET_STATE(HAL_IO_LED1, 1);
   connected = true;
   app_state = COUNTER_APP_RUN;
   lib_dispatcher_post_msg(HANDLE_PROCESS_COUNTER_APP, NULL, 0, NORMAL_PRIORITY);
   lib_aci_enable_transmission();
}

void on_disconnect_event(uint8_t size, uint8_t* buffer)
{
   puts("\n================================================\nBTLE␣disconnected.␣Test␣results:");
   printf("\tSent␣packets:␣%d\n", sent_packets);
   printf("\tDiscovery␣time:␣%d␣ms\n", discovery_time);
   puts("================================================\n");
   sent_packets = 0;
   discovery_time = 0;
   HAL_IO_SET_STATE(HAL_IO_LED0, 0);
   HAL_IO_SET_STATE(HAL_IO_LED1, 0);
   connected = false;
   app_state = COUNTER_APP_IDLE;
```

```
   next_app_state = COUNTER_APP_IDLE;
   lib_aci_enable_transmission();
}

void on_error (uint8_t size, uint8_t* buffer)
{

   lib_aci_error_descriptor_t error;
   lib_aci_interpret_error(&error);
   printf("BTLE error: 0x%X 0x%X 0x%X\n", error.error_code, error.error_sub_code1, error.error_sub_code2);
   lib_aci_enable_transmission();
   //while (1)
      //;
}

void on_data_credit_event(uint8_t size, uint8_t* buffer)
{
   /*
   if (lib_aci_send_data(PIPE_PACKET_COUNTER_COUNTER_TX, &btle_counter, 1))
   {
      sent_packets++;
      btle_counter++;
   }
   */
   lib_aci_enable_transmission();
}

void on_hw_error (uint8_t size, uint8_t* buffer)
{

   puts("BTLE hw error!\n");

   HAL_IO_SET_STATE(HAL_IO_LED0, 1);
   HAL_IO_SET_STATE(HAL_IO_LED1, 1);
   lib_aci_enable_transmission();
   while (1)
      ;
}

void on_btle_pipe_error(uint8_t size, uint8_t* buffer)
{
   aci_evt_params_pipe_error_t error;
   lib_aci_interpret_evt_pipe_error(&error);
   printf("BTLE pipe no %d error. Code 0x%x\n", error.pipe_number, error.error_code);
   lib_aci_enable_transmission();
}

void on_btle_pipe_status_event(uint8_t size, uint8_t* buffer)
{

   aci_evt_params_pipe_status_t status;
   lib_aci_interpret_evt_pipes_status(&status);
   puts("BTLE pipe status event: ");
   if ((status.pipes_open_bitmap[0] & 0x01) == 0)
      puts("pipe discovery not complete. ");
   if ((status.pipes_open_bitmap[0] & (PIPE_PACKET_COUNTER_COUNTER_TX+1)) != 0)
   {
      puts("pipe is open\n");
      /*
      if (lib_aci_send_data(PIPE_PACKET_COUNTER_COUNTER_TX, &btle_counter, 1))
      {
         sent_packets++;
         btle_counter++;
      }
      */
   }

   else
      puts("pipe is closed\n");

   lib_aci_enable_transmission();

}

void on_advertise_timeout(uint8_t size, uint8_t* buffer)
{
   puts("BTLE advertising timedout.\n");
   app_state = COUNTER_APP_IDLE;
   next_app_state = COUNTER_APP_IDLE;
   lib_aci_enable_transmission();
}


void on_btle_timer_interrupt(uint8_t size, uint8_t* buffer)
{
```

```
   if (connected)
   {
      GPIOIntDisable(PORT1, 9);
      while(!lib_aci_disconnect(ACI_REASON_TERMINATE))
         ;
      GPIOIntEnable(PORT1, 9);
   }

}


void PIOINT1_IRQHandler(void)
{
   // Active signal interrupt handler
   uint32_t regVal;
   regVal = GPIOIntStatus(PORT1, 9);
   static uint8_t toggle = 0;
   if (regVal) {
      GPIOIntClear(PORT1, 9);

      if (connected && lib_aci_is_pipe_available(PIPE_PACKET_COUNTER_COUNTER_TX))
      {
         if (lib_aci_send_data(PIPE_PACKET_COUNTER_COUNTER_TX, &btle_counter, 1))
         {
            sent_packets++;
            btle_counter++;
         }
      }

      HAL_IO_SET_STATE(HAL_IO_LED0, toggle);
      toggle = !toggle;
   }
}


/*
 * main.c
 *
 * This is the startup code.
 */

#include <stdio.h>  // puts
#include <string.h>

#include "LPC11xx.h"
#include "system.h"
#include "gpio.h"
#include "lib_dispatcher.h"
#include "lib_aci.h"
#include "lib_ant.h"
#include "counter_app_ant.h"
#include "counter_app_btle.h"
#include "lpc_swu.h"
#include "timer32.h"

/*
 * This function is called by lib_aci when a valid message is
 * received. We use the dispatcher, so post a message there.
 */
void lib_aci_post_msg_hook(uint8_t ident)
{
   lib_dispatcher_post_msg(ident, NULL, 0, HIGH_PRIORITY);
}


int main(void)
{

   init_timer();        // Initialize timer
   swu_init(LPC_TMR32B1);  // Initialize software UART. Used for the user interface
   system_init();       // Initialize GPIO, interrupts, etc.

   lib_aci_init();         // Initialize BLE library
   ANT_Init(19200);        // Initialize ANT library

   lib_dispatcher_init();  // Initialize dispatcher

   reset_btle_radio();     // Hardware reset of radios
   reset_ant_radio();
   ANT_ResetSystem();      // ANT also needs a soft reset..

   // Set default setup values
   ant_setup.channel_type   = PARAMETER_TX_NOT_RX;
   ant_setup.channel_period = 4;
   ant_setup.frequencies[0] = 3;
   ant_setup.frequencies[1] = 39;
   ant_setup.frequencies[2] = 75;
```

```
    ant_setup.tx_power       = DEFAULT_RADIO_TX_POWER; // default = 0 dBm
    ant_setup.search_timeout = 6;      // 6*2.5 = 15 sec

    btle_setup.tx_power    = ACI_DEVICE_OUTPUT_POWER_0DBM;
    btle_setup.adv_timeout = 15;


    while (1)
    {
       lib_dispatcher_dispatch();
    }

    return(0);
}
```

D.2 ANT PC application

```
/*
 * This is the ANT PC application. Based on an example application from Dynastream.
 *
 * Dynastream Innovations Inc.
 * Cochrane, AB, CANADA
 *
 * Copyright © 1998-2009 Dynastream Innovations Inc.
 * All rights reserved. This software may not be reproduced by
 * any means without express written approval of Dynastream
 * Innovations Inc.
 */

#include "demo.h"

#include "types.h"
#include "dsi_framer_ant.hpp"
#include "dsi_thread.h"
#include "dsi_serial_generic.hpp"

#include <stdio.h>
#include <assert.h>
#include <string.h>

//#define ENABLE_EXTENDED_MESSAGES

#define USER_BAUDRATE         (57600)  // For AT3/AP2, use 57600
#define USER_RADIOFREQ        (35)

#define USER_ANTCHANNEL       (0)
#define USER_DEVICENUM        (1)
#define USER_DEVICETYPE       (2)
#define USER_TRANSTYPE        (3)

#define USER_NETWORK_KEY      {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,}
#define USER_NETWORK_NUM      (0)      // The network key is assigned to this network number

#define MESSAGE_TIMEOUT       (1000)

// Indexes into message recieved from ANT
#define MESSAGE_BUFFER_DATA1_INDEX ((UCHAR) 0)
#define MESSAGE_BUFFER_DATA2_INDEX ((UCHAR) 1)
#define MESSAGE_BUFFER_DATA3_INDEX ((UCHAR) 2)
#define MESSAGE_BUFFER_DATA4_INDEX ((UCHAR) 3)
#define MESSAGE_BUFFER_DATA5_INDEX ((UCHAR) 4)
#define MESSAGE_BUFFER_DATA6_INDEX ((UCHAR) 5)
#define MESSAGE_BUFFER_DATA7_INDEX ((UCHAR) 6)
#define MESSAGE_BUFFER_DATA8_INDEX ((UCHAR) 7)
#define MESSAGE_BUFFER_DATA9_INDEX ((UCHAR) 8)
#define MESSAGE_BUFFER_DATA10_INDEX ((UCHAR) 9)
#define MESSAGE_BUFFER_DATA11_INDEX ((UCHAR) 10)
#define MESSAGE_BUFFER_DATA12_INDEX ((UCHAR) 11)
#define MESSAGE_BUFFER_DATA13_INDEX ((UCHAR) 12)
#define MESSAGE_BUFFER_DATA14_INDEX ((UCHAR) 13)


#define CLOCK_FREQUENCY (32768)


////////////////////////////////////////////////////////////////////////////////
// main
//
////////////////////////////////////////////////////////////////////////////////
int main(int argc, char **argv)
{

   if (argc < 8)
   {
      printf("Usage:␣%s␣type␣period␣freq1␣freq2␣freq3␣enable_freq_agility␣tx_power\n", argv[0]);
      return -1;
   }

   BYTE master_slave = (BYTE) atoi(argv[1]);
   USHORT period = (USHORT) atoi(argv[2]);
   BYTE freq1 = (UCHAR) atoi(argv[3]);
   BYTE freq2 = (UCHAR) atoi(argv[4]);
   BYTE freq3 = (UCHAR) atoi(argv[5]);
   BOOL en_freq_agility = (BOOL) atoi(argv[6]);
   BYTE tx_power = (BYTE) atoi(argv[7]);

   AntSetup* setup = new AntSetup(master_slave, period, freq1, freq2, freq3, en_freq_agility, tx_power, 4);
    Demo* pclDemo = new Demo(setup);

    UCHAR ucDeviceNumber = 0x00;
```

```
   if(pclDemo->Init())
      pclDemo->Start();
   else
      delete pclDemo;
   return 0;
}


AntSetup::AntSetup(BYTE type, USHORT period, BYTE freq1, BYTE freq2, BYTE freq3, bool freq_agility_enabled, BYTE
      tx_power, BYTE search_timeout)
{
   this->channel_type = type;
   this->channel_period = period;
   this->frequencies[0] = freq1;
   this->frequencies[1] = freq2;
   this->frequencies[2] = freq3;
   this->frequency_agility_enabled = freq_agility_enabled;
   this->tx_power = tx_power;
   this->search_timeout = search_timeout;
}

AntSetup::~AntSetup()
{
}


///////////////////////////////////////////////////////////////////////////////
// Demo
//
// Constructor, intializes Demo class
//
///////////////////////////////////////////////////////////////////////////////
Demo::Demo(AntSetup* setup)
{
   pclSerialObject = (DSISerialGeneric*)NULL;
   pclMessageObject = (DSIFramerANT*)NULL;
   uiDSIThread = (DSI_THREAD_ID)NULL;
   bMyDone = FALSE;
   bDone = FALSE;
   bDisplay = TRUE;
   bStarted = FALSE;

   counter = 0;
   received_packets = 0;
   sent_packets = 0;
   lost_packets = 0;
   failed_sent_packets = 0;
   retransmitted_rx_packets = 0;
   discovery_time = 0;
   invalid_counter = 0;

   ant_setup = setup;
   stop_cmd_sent = false;
   received_first_packet = false;
   connected = false;
   memset(aucTransmitBuffer,0,ANT_STANDARD_DATA_PAYLOAD_SIZE);
}

///////////////////////////////////////////////////////////////////////////////
// ~Demo
//
// Destructor, clean up and loose memory
//
///////////////////////////////////////////////////////////////////////////////
Demo::~Demo()
{
   if(pclMessageObject)
      delete pclMessageObject;

   if(pclSerialObject)
      delete pclSerialObject;

   if (ant_setup)
      delete ant_setup;
}

///////////////////////////////////////////////////////////////////////////////
// Init
//
// Initize the Demo and ANT Library.
//
// ucDeviceNumber_: USB Device Number (0 for first USB stick plugged and so on)
//                  If not specified on command line, 0xFF is passed in as invalid.
// ucChannelType_:  ANT Channel Type. 0 = Master, 1 = Slave
//                  If not specified, 2 is passed in as invalid.
```

```
//
//////////////////////////////////////////////////////////////////////////////
BOOL Demo::Init()
{

   BOOL bStatus;

   // Initialize condition var and mutex
   UCHAR ucCondInit = DSIThread_CondInit(&condTestDone);
   assert(ucCondInit == DSI_THREAD_ENONE);

   UCHAR ucMutexInit = DSIThread_MutexInit(&mutexTestDone);
   assert(ucMutexInit == DSI_THREAD_ENONE);

#if defined(DEBUG_FILE)
   // Enable logging
   DSIDebug::Init();
   DSIDebug::SetDebug(TRUE);
#endif

   // Create Serial object.
   pclSerialObject = new DSISerialGeneric();
   assert(pclSerialObject);

   // NOTE: Will fail if the module is not available.
   // If no device number was specified on the command line,
   // prompt the user for input.


   // Initialize Serial object.
   // The device number depends on how many USB sticks have been
   // plugged into the PC. The first USB stick plugged will be 0
   // the next 1 and so on.
   //
   // The Baud Rate depends on the ANT solution being used. AP1
   // is 50000, all others are 57600
   bStatus = pclSerialObject->Init(USER_BAUDRATE, 0x00);
   assert(bStatus);

   // Create Framer object.
   pclMessageObject = new DSIFramerANT(pclSerialObject);
   assert(pclMessageObject);

   // Initialize Framer object.
   bStatus = pclMessageObject->Init();
   assert(bStatus);

   // Let Serial know about Framer.
   pclSerialObject->SetCallback(pclMessageObject);

   // Open Serial.
   bStatus = pclSerialObject->Open();

   // If the Open function failed, most likely the device
   // we are trying to access does not exist, or it is connected
   // to another program
   if(!bStatus)
   {
      printf("Failed␣to␣connect␣to␣device␣at␣USB␣port␣0\n");
      return FALSE;
   }

   // Create message thread.
   uiDSIThread = DSIThread_CreateThread(&Demo::RunMessageThread, this);
   assert(uiDSIThread);

   printf("Initialization␣was␣successful!\n"); fflush(stdout);

   return TRUE;
}


//////////////////////////////////////////////////////////////////////////////
// Close
//
// Close connection to USB stick.
//
//////////////////////////////////////////////////////////////////////////////
void Demo::Close()
{

   DSIThread_MutexLock(&mutexTestDone);
   bDone = TRUE;

   UCHAR ucWaitResult = DSIThread_CondTimedWait(&condTestDone, &mutexTestDone, DSI_THREAD_INFINITE);
   assert(ucWaitResult == DSI_THREAD_ENONE);
```

```
   DSIThread_MutexUnlock(&mutexTestDone);

   //Destroy mutex and condition var
   DSIThread_MutexDestroy(&mutexTestDone);
   DSIThread_CondDestroy(&condTestDone);

   //Close all stuff
   if(pclSerialObject)
      pclSerialObject->Close();

#if defined(DEBUG_FILE)
   DSIDebug::Close();
#endif
   printf("Exiting...\n");

}

////////////////////////////////////////////////////////////////////////////////
// Start
//
// Starts the Demo
//
////////////////////////////////////////////////////////////////////////////////
void Demo::Start()
{
   BOOL bStatus;

   // Print out the menu to start
   //PrintMenu();



   // Start ANT channel setup
   if (!InitANT())
   {
      printf("failed to initialize ANT\n");
      return;
   }

   UCHAR ucCmd[5];

   while (!bMyDone)
   {
      scanf("%s", &ucCmd[0]);
      if (strncmp((const char*)&ucCmd[0], "start", 5) == 0)
      {
         bStarted = TRUE;
         test_start_time = clock();
         // Open channel by setting network key. Channel setup is done as each setup command is successful
         printf("Opening ANT channel with:\n");
         printf("\tChannel type: %s\n", (ant_setup->channel_type == CHANNEL_TYPE_MASTER) ? "master" : "slave");
         printf("\tChannel period: %d\n", ant_setup->channel_period);
         printf("\tRadio frequencies: %d, %d, %d\n", ant_setup->frequencies[0], ant_setup->frequencies[1],
               ant_setup->frequencies[2]);
         printf("\tFreq agility: %s\n", ant_setup->frequency_agility_enabled ? "enabled" : "disabled");
         printf("\tTx power: %d\n", ant_setup->tx_power);


         UCHAR ucNetKey[8] = USER_NETWORK_KEY;
         pclMessageObject->SetNetworkKey(USER_NETWORK_NUM, ucNetKey, MESSAGE_TIMEOUT);
      }
      else if (strncmp((const char*)&ucCmd[0], "stop", 4) == 0)
      {
         stop_cmd_sent = true;
         pclMessageObject->CloseChannel(USER_ANTCHANNEL, MESSAGE_TIMEOUT);
      }
      else
      {
         break;
      }
      DSIThread_Sleep(0);
   }


   printf("Disconnecting module....\n");
   this->Close();
   printf("Test completed successfully\n");

   return;
}

////////////////////////////////////////////////////////////////////////////////
// InitANT
//
```

```
// Resets the system and starts the test
//
////////////////////////////////////////////////////////////////////////////////
BOOL Demo::InitANT(void)
{
   BOOL bStatus;

   // Reset system
   printf("Resetting␣module...\n");
   bStatus = pclMessageObject->ResetSystem();
   DSIThread_Sleep(1000);


   return bStatus;
}

////////////////////////////////////////////////////////////////////////////////
// RunMessageThread
//
// Callback function that is used to create the thread. This is a static
// function.
//
////////////////////////////////////////////////////////////////////////////////
DSI_THREAD_RETURN Demo::RunMessageThread(void *pvParameter_)
{
   ((Demo*) pvParameter_)->MessageThread();
   return NULL;
}

////////////////////////////////////////////////////////////////////////////////
// MessageThread
//
// Run message thread
////////////////////////////////////////////////////////////////////////////////
void Demo::MessageThread()
{
   ANT_MESSAGE stMessage;
   USHORT usSize;
   bDone = FALSE;

   while(!bDone)
   {
      if(pclMessageObject->WaitForMessage(1000 /*DSI_THREAD_INFINITE*/))
      {
         usSize = pclMessageObject->GetMessage(&stMessage);

         if(bDone)
            break;

         if(usSize == DSI_FRAMER_ERROR)
         {
            // Get the message to clear the error
            usSize = pclMessageObject->GetMessage(&stMessage, MESG_MAX_SIZE_VALUE);
            continue;
         }

         if(usSize != DSI_FRAMER_ERROR && usSize != DSI_FRAMER_TIMEDOUT && usSize != 0)
         {
            ProcessMessage(stMessage, usSize);
         }
      }
   }

   DSIThread_MutexLock(&mutexTestDone);
   UCHAR ucCondResult = DSIThread_CondSignal(&condTestDone);
   assert(ucCondResult == DSI_THREAD_ENONE);
   DSIThread_MutexUnlock(&mutexTestDone);
   //this->Close();
}


////////////////////////////////////////////////////////////////////////////////
// ProcessMessage
//
// Process ALL messages that come from ANT, including event messages.
//
// stMessage: Message struct containing message recieved from ANT
// usSize_:
////////////////////////////////////////////////////////////////////////////////
void Demo::ProcessMessage(ANT_MESSAGE stMessage, USHORT usSize_)
{
   BOOL bStatus;
   BOOL bPrintBuffer = FALSE;
   UCHAR ucDataOffset = MESSAGE_BUFFER_DATA2_INDEX;   // For most data messages

   static bool is_low_priority_search_timeout_set = false;
```

```
if (!bStarted && stMessage.ucMessageID != MESG_STARTUP_MESG_ID) return;

switch(stMessage.ucMessageID)
{
   //RESPONSE MESG
   case MESG_RESPONSE_EVENT_ID:
   {
      //RESPONSE TYPE
     if ((stMessage.aucData[1] != MESG_EVENT_ID) && (stMessage.aucData[2] != RESPONSE_NO_ERROR))
     {
        printf("cmd␣0x%X␣failed␣with␣status␣0x%X\n", stMessage.aucData[1], stMessage.aucData[2]);
        if (stop_cmd_sent)
        {
           bMyDone = TRUE;
        }
     }
     else
     {
       switch(stMessage.aucData[1])
       {

         case MESG_NETWORK_KEY_ID:
         {
            //printf("Network Key set.\n\n");
            //printf("Assigning channel...\n");
            UCHAR channelType;
            if (ant_setup->channel_type == CHANNEL_TYPE_MASTER) channelType = PARAMETER_TX_NOT_RX;
            else if (ant_setup->channel_type == CHANNEL_TYPE_SLAVE) channelType = PARAMETER_RX_NOT_TX;
            else
            {
               printf("Error:␣invalid␣channel␣type\n");
               return;
            }

            if (ant_setup->frequency_agility_enabled)
            {
               UCHAR msg[2];
               msg[0] = channelType;
               msg[1] = EXT_PARAM_FREQUENCY_AGILITY;
               bStatus = pclMessageObject->AssignChannelExt(USER_ANTCHANNEL, &msg[0], 2, 0, MESSAGE_TIMEOUT);
            }
            else
            {
               bStatus = pclMessageObject->AssignChannel(USER_ANTCHANNEL, channelType, 0, MESSAGE_TIMEOUT);
            }
            break;
         }

         case MESG_ASSIGN_CHANNEL_ID:
         {
            //printf("Channel Assigned\n\n");
            //printf("Setting Channel ID...\n");
            bStatus = pclMessageObject->SetChannelID(USER_ANTCHANNEL, USER_DEVICENUM, USER_DEVICETYPE,
                  USER_TRANSTYPE, MESSAGE_TIMEOUT);
            break;
         }

         case MESG_CHANNEL_ID_ID:
         {
            //printf("Channel ID set\n\n");
            //printf("Setting channel period...\n");
            bStatus = pclMessageObject->SetChannelPeriod(USER_ANTCHANNEL, CLOCK_FREQUENCY/ant_setup->
                  channel_period, MESSAGE_TIMEOUT);
            break;
         }

         case MESG_CHANNEL_MESG_PERIOD_ID :
         {
            //printf("Channel period set\n\n");
            //printf("Setting LP search timout...\n");
            bStatus = pclMessageObject->SetLowPriorityChannelSearchTimeout(USER_ANTCHANNEL, 0,
                  MESSAGE_TIMEOUT);
            break;
         }


         case MESG_SET_LP_SEARCH_TIMEOUT_ID :
         {
            //printf("LP search timeout set\n\n");
            //printf("Setting HP search timeout...\n");
            bStatus = pclMessageObject->SetChannelSearchTimeout(USER_ANTCHANNEL, 6, MESSAGE_TIMEOUT);
            break;
         }
```

```
case MESG_CHANNEL_SEARCH_TIMEOUT_ID :
{
   //printf("HP search timeout set\n\n");
   //printf("Setting channel tx power...\n");
   bStatus = pclMessageObject->SetChannelTransmitPower(USER_ANTCHANNEL, ant_setup->tx_power,
         MESSAGE_TIMEOUT);
   break;
}



case MESG_CHANNEL_RADIO_TX_POWER_ID :
{
   //printf("Tx power set\n\n");
   //printf("Setting radio frequency.....\n");
   if (ant_setup->frequency_agility_enabled)
   {
      bStatus = pclMessageObject->ConfigFrequencyAgility(USER_ANTCHANNEL, ant_setup->frequencies[0],
            ant_setup->frequencies[1], ant_setup->frequencies[2], MESSAGE_TIMEOUT);
   }
   else
   {
      bStatus = pclMessageObject->SetChannelRFFrequency(USER_ANTCHANNEL, ant_setup->frequencies[0],
           MESSAGE_TIMEOUT);
   }
   break;
}

case MESG_CHANNEL_RADIO_FREQ_ID:
{
   //printf("Radio frequency set\n\n");
   //printf("Opening channel...\n");
   bStatus = pclMessageObject->OpenChannel(USER_ANTCHANNEL, MESSAGE_TIMEOUT);
   break;
}

case MESG_AUTO_FREQ_CONFIG_ID :
{
   //printf("Frequency agility configured\n\n");
   //printf("Opening channel...\n");
   bStatus = pclMessageObject->OpenChannel(USER_ANTCHANNEL, MESSAGE_TIMEOUT);
   break;
}


case MESG_OPEN_CHANNEL_ID:
{
   printf("Channel␣opened.\n\n");
   break;
}

case MESG_UNASSIGN_CHANNEL_ID:
{
   printf("Channel␣unassigned\n\n");
   printf("Test␣complete.␣Results:␣\n");
   printf("\tSent␣packets:␣%d\n", sent_packets);
   printf("\tReceived␣packets:␣%d\n", received_packets);
   printf("\tLost␣packets:␣%d\n", lost_packets);
   printf("\tFailed␣sent␣packets:␣%d\n", failed_sent_packets);
   printf("\tRetransmitted␣rx␣packets:␣%d\n", retransmitted_rx_packets);
   printf("\tRetransmitted␣tx␣packets:␣%d\n", failed_sent_packets);
   printf("\tPackets␣with␣invalid␣counter:␣%d\n", invalid_counter);
   printf("\tDiscovery␣time:␣%d\n", discovery_time);

   bMyDone = TRUE;
   break;
}

case MESG_CLOSE_CHANNEL_ID:
{
   break;
}


case MESG_EVENT_ID:
{
   switch(stMessage.aucData[2])
   {
      case EVENT_CHANNEL_CLOSED:
      {
         if (stop_cmd_sent)
         {
            printf("Channel␣closed\n\n");
            printf("Unassigning␣channel...\n");
            bStatus = pclMessageObject->UnAssignChannel(USER_ANTCHANNEL, MESSAGE_TIMEOUT);
```

```
            }
            else
            {
               printf("ANT␣changed␣freqency/channel␣closed␣due␣to␣timeout\n");
            }
            break;
         }
        case EVENT_TX:
        {
          aucTransmitBuffer[0] = counter++;
          //pclMessageObject->SendAcknowledgedData(USER_ANTCHANNEL, aucTransmitBuffer, MESSAGE_TIMEOUT)
               ;
          pclMessageObject->SendBroadcastData(USER_ANTCHANNEL, aucTransmitBuffer);
          if (ant_setup->frequency_agility_enabled == false)
          {
             sent_packets++;
          }
          break;

        }
        case EVENT_RX_SEARCH_TIMEOUT:
        {
          printf("Search␣Timeout␣on␣channel\n");
          break;
        }
        case EVENT_RX_FAIL:
        {
           if (connected)
           {
             printf("Rx␣Fail␣at␣%d␣ms\n", (clock() - test_start_time) * CLOCKS_PER_SEC/1000);
             lost_packets++;
           }
          break;
        }
        case EVENT_TRANSFER_RX_FAILED:
        {
          printf("Burst␣recieve␣has␣failed\n");
          break;
        }
        case EVENT_TRANSFER_TX_COMPLETED:
        {
          aucTransmitBuffer[0] = counter++;
          pclMessageObject->SendBroadcastData(USER_ANTCHANNEL, aucTransmitBuffer);
          sent_packets++;
          break;
        }
        case EVENT_TRANSFER_TX_FAILED:
        {
          //aucTransmitBuffer[0] = counter++;
          //pclMessageObject->SendAcknowledgedData(USER_ANTCHANNEL, aucTransmitBuffer, MESSAGE_TIMEOUT)
               ;
          failed_sent_packets++;
          break;
        }
        case EVENT_RX_FAIL_GO_TO_SEARCH:
        {
          printf("Goto␣Search.\n");
          break;
        }
        case EVENT_CHANNEL_COLLISION:
        {
          printf("Channel␣Collision\n");
          break;
        }
        case EVENT_TRANSFER_TX_START:
        {
          printf("Burst␣Started\n");
          break;
        }
        default:
        {
              printf("Unknown␣Channel(%d)␣Event:␣0x%X\n", stMessage.aucData[0], stMessage.aucData[2]);
          break;
        }

      }

      break;
    }

    default:
    {
          printf("Unknown␣Response␣0x%X,␣Code␣0%d\n", stMessage.aucData[1], stMessage.aucData[2]);
      break;
    }
  }
```

```
}
break;
}

case MESG_STARTUP_MESG_ID:
{
   printf("RESET␣Complete,␣reason:␣");

   UCHAR ucReason = stMessage.aucData[MESSAGE_BUFFER_DATA1_INDEX];

   if(ucReason == RESET_POR)
      printf("RESET_POR");

   if(ucReason & RESET_SUSPEND)
      printf("RESET_SUSPEND␣");
   if(ucReason & RESET_SYNC)
      printf("RESET_SYNC␣");
   if(ucReason & RESET_CMD)
      printf("RESET_CMD␣");
   if(ucReason & RESET_WDT)
      printf("RESET_WDT␣");
   if(ucReason & RESET_RST)
      printf("RESET_RST␣");

   printf("\n");

   break;
}

case MESG_ACKNOWLEDGED_DATA_ID:
{
   byte counter_diff = stMessage.aucData[1] - counter;

   if (stMessage.aucData[2] != 0)
   {
      printf("Disconnect␣time:␣%d\n", (clock()- test_start_time) * CLOCKS_PER_SEC/1000);
      connected = false;
   }
   else
   {
      connected = true;
   }

   if (!received_first_packet)
   {
      discovery_time = (clock() - test_start_time) * CLOCKS_PER_SEC/1000;
   }

   if (counter_diff == 0 && received_first_packet)
   {
       retransmitted_rx_packets++;
   }
   else if (counter_diff == 1)
   {
       received_packets++;
   }
   else
   {
      invalid_counter++;
   }
   counter = stMessage.aucData[1];
   received_first_packet = true;
   break;
}
case MESG_BURST_DATA_ID:
{
    printf("Burst(0x%02x)␣Rx:(%d):␣", ((stMessage.aucData[MESSAGE_BUFFER_DATA1_INDEX] & 0xE0) >> 5),
          stMessage.aucData[MESSAGE_BUFFER_DATA1_INDEX] & 0x1F );
    break;
}
case MESG_BROADCAST_DATA_ID:
{
   byte counter_diff = stMessage.aucData[1] - counter;

   if (stMessage.aucData[2] != 0)
   {
      printf("Disconnect␣time:␣%d\n", (clock()- test_start_time) * CLOCKS_PER_SEC/1000);
      connected = false;
   }
   else
   {
      connected = true;
   }

   if (!received_first_packet)
   {
```

```
            discovery_time = (clock() - test_start_time) * CLOCKS_PER_SEC/1000;
         }

         if (counter_diff == 0 && received_first_packet)
         {
            retransmitted_rx_packets++;
         }
         else if (counter_diff == 1)
         {
            received_packets++;
         }
         else
         {
            invalid_counter++;
         }
         counter = stMessage.aucData[1];
         received_first_packet = true;
         break;
      }

      default:
      {
         break;
      }
   }

   fflush(stdout);
   return;
}
```

D.3 BLE PC application

```
ï»¿using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
using Nordicsemi;

namespace masterEmulatorTester
{
    class Program
    {
        private static byte packet_counter = 0, lost_packets = 0;
        private static int received_packets = 0;
        private static bool first_run = true;

        private static string channel_map = "1FFFFFFFFF";
        private static double connectionIntervalMs = 40;
        private static double connectionSearchTimeout = 15;
        private static byte tx_power = 0;

        static void Main(string[] args)
        {
            #region Parse arguments
            if (args.Length != 4)
            {
                Console.WriteLine("Usage:␣masterEmulatorTester␣channel_map␣connection_interval␣search_timeout␣
                      tx_power");
                return;
            }

            channel_map = args[0];
            connectionIntervalMs = 1000.0/Double.Parse(args[1]);
            connectionSearchTimeout = Double.Parse(args[2]);
            tx_power = byte.Parse(args[3]);

            #endregion

            MasterEmulator master = new MasterEmulator();
            master.LogMessage += new EventHandler<ValueEventArgs<string>>(master_LogMessage);
            master.DataReceived += new EventHandler<PipeDataEventArgs>(master_DataReceived);
            master.Connected += new EventHandler<EventArgs>(master_Connected);
            master.Disconnected += new EventHandler<ValueEventArgs<DisconnectReason>>(master_Disconnected);



            #region pipesetup
            // Counter pipe
            BtUuid serviceUuid1 = new BtUuid(0x1234, BtUuid.BtBaseUuid); //Counter service
            PipeStore pipeStore = PipeStore.Remote;
            master.SetupAddService(serviceUuid1, pipeStore);

            BtUuid charDefUuid1 = new BtUuid(0x2345); //Counter characteristic
            int maxDataLength = 1;
            byte[] data = new byte[] { };
            master.SetupAddCharacteristicDefinition(charDefUuid1, maxDataLength, data);

            PipeType pipeType1 = PipeType.Receive;
            int pipeNumber1 = master.SetupAssignPipe(pipeType1);

            #endregion

            /*
            Stack<string> usbDevs = new Stack<string>(master.EnumerateUsb());
            if (usbDevs.Count == 0)
            {
                Console.WriteLine("No usb devices found");
                Console.ReadKey();
                Environment.Exit(0);
            }
            string usbDev0 = string.Empty;
            while (usbDevs.Count > 0)
            {
                string dev = usbDevs.Pop();
                if (dev.Contains("FLE"))
                    continue;
                usbDev0 = dev;
            }

            if (usbDev0 == string.Empty)
            {
                Console.WriteLine("Found no suitable USB device. Press a key to exit");
                Console.ReadKey();
                Environment.Exit(0);
            }
```

```
*/

master.Open("006RYE3E");
Console.WriteLine(string.Format("IsOpen:␣{0}", master.IsOpen.ToString()));
master.Run();
Console.WriteLine(string.Format("IsRunning:␣{0}", master.IsRunning.ToString()));

//int scanDuration = 10;
BtScanParameters scanParameters = new BtScanParameters();
scanParameters.ScanIntervalMs = 10;
scanParameters.ScanWindowMs = 10;


// Ready to run device discovery
Console.WriteLine("ready");
while (Console.ReadLine().CompareTo("start") != 0)
    ;



/*
List<BtDevice> devices = new List<BtDevice>(master.DiscoverDevices(scanDuration, scanParameters));
if (devices.Count == 0)
{
    Console.WriteLine("No devices discovered");
}

*/
BtDeviceAddress addressToConnectTo = new BtDeviceAddress("36E2313993F5");


/*
foreach (BtDevice dev in devices)
{
    Console.WriteLine(dev.DeviceAddress.Value);
    foreach (KeyValuePair<DeviceInfoType, string> devInfo in dev.DeviceInfo)
    {
        if (devInfo.Key == DeviceInfoType.CompleteLocalName)
        {
            if (devInfo.Value == "Per Magnus")
            {
                addressToConnectTo = dev.DeviceAddress;
                TimeSpan discoveryDuration = DateTime.Now - startDiscoveryTime;
                Console.WriteLine("Discovery took " + discoveryDuration.TotalMilliseconds + "
                      milliseconds");
            }
        }
        Console.WriteLine(string.Format("{0}:{1}", devInfo.Key, devInfo.Value));
    }
}
*/


if (addressToConnectTo != null)
{
    BtConnectionParameters connectionParameters = new BtConnectionParameters();
    connectionParameters.ConnectionIntervalMs = 7.5;
    connectionParameters.SlaveLatency = 0;
    connectionParameters.SupervisionTimeoutMs = connectionSearchTimeout * 1000.0;

    DateTime startDiscoveryTime = DateTime.Now;
    bool connectSuccess = master.Connect(addressToConnectTo, ConnectionMode.ConnectWithoutBonding,
          connectionParameters);


    //bool disconnectsuccess = master.Disconnect();

    //bool bondSuccess = master.Connect(addressToConnectTo, ConnectionMode.ConnectWithBonding,
          connectionParameters);
    Console.WriteLine(string.Format("IsConnected:␣{0}", master.IsConnected.ToString()));

    if (master.IsConnected)
    {
        try
        {
            master.SetChannelMap(channel_map);
            master.SetRadioTxPower(tx_power);
            master.DiscoverPipes();
            connectionParameters.ConnectionIntervalMs = connectionIntervalMs;
            master.UpdateConnectionParameters(connectionParameters);
            TimeSpan discoveryDuration = DateTime.Now - startDiscoveryTime;
            Console.WriteLine("Discovery␣took␣" + discoveryDuration.TotalMilliseconds + "␣ms");

            while (Console.ReadLine().CompareTo("stop") != 0) // Lets hope the thread blocks at
                  Console.ReadLine()...
```

```
                    ;
                }
                catch (Exception e)
                {
                    Console.WriteLine(e.ToString());
                    while (Console.ReadLine().CompareTo("stop") != 0)
                        ;

                }

                //master.Disconnect();
                //Console.ReadKey();
            }
            else
            {
                Console.WriteLine("Could not connect to device.");
                while (Console.ReadLine().CompareTo("stop") != 0)
                    ;
            }
        }


        master.Reset();
        master.Close();
    }

    #region Hook functions
    static void master_Connected(object sender, EventArgs e)
    {
        string msg = "CONNECTED EVENT";
        Console.WriteLine(msg);
        Debug.WriteLine(msg);
    }

    static void master_Disconnected(object sender, ValueEventArgs<DisconnectReason> e)
    {
        /*
        string msg = "DISCONNECTED EVENT, REASON: " + e.Value.ToString();
        Console.WriteLine(msg);
        Debug.WriteLine(msg);
        */
        Console.WriteLine("Disconnected.");
        Console.WriteLine(String.Format("\tReceived packets: {0:d}", received_packets));
        Console.WriteLine(String.Format("\tLost packets: {0:d}", lost_packets));
    }

    static void master_DataReceived(object sender, PipeDataEventArgs e)
    {

        byte current_counter = e.PipeData[0];
        byte tmp_lost_packets = (byte)(current_counter - packet_counter - 1);
        if (!first_run && (tmp_lost_packets > 0))
        {
            Console.WriteLine(String.Format("Lost {0:d} packets", tmp_lost_packets));
            lost_packets += (byte)(tmp_lost_packets);
        }

        //Console.WriteLine(String.Format("Current counter: {0:d}", current_counter));

        packet_counter = current_counter;
        received_packets++;
        first_run = false;
        /*
        string msg = "DATA_RECEIVED_EVENT PIPENUMBER " + e.PipeNumber + " DATA " + BitConverter.ToString(e.
            PipeData);
        Console.WriteLine(msg);
        Debug.WriteLine(msg);
        */

    }

    static void master_LogMessage(object sender, ValueEventArgs<string> e)
    {
        if (e.Value.Contains("Received AttHandleValueNotification"))
            return;
        string msg = "[Log] " + e.Value;
        Console.WriteLine(msg);
        Debug.WriteLine(msg);
    }


    #endregion
  }
}
```

D.4 Python scripts

```
import serial
import threading
import socket
import time
import logging
import signal
import sys


def SerialReceiveThread(serial, stop_event):
    log_file = open("motherboard_log.txt", "w")
    while stop_event.isSet() == False:
        line = serial.readline()
        log_file.write(line)

    log_file.close()
    serial.close()


class MotherboardTestController:
    serial_interface = 0
    serial_thread = 0
    serial_thread_event = 0

    def __init__(self):
        self._setupSerial()
        self._setupLogging()


    def _setupSerial(self):
        self.serial_interface = serial.Serial(port=0, baudrate=9600, bytesize=serial.EIGHTBITS,
            parity=serial.PARITY_NONE, stopbits=serial.STOPBITS_ONE, timeout=10)
        self.serial_interface.close()  # why do I need this exactly??
        self.serial_interface.open()

    def _setupLogging(self):
        if self.serial_interface != 0:
            self.serial_thread_event = threading.Event()
            self.serial_thread = threading.Thread(target=SerialReceiveThread, args=(self.serial_interface, self.
                serial_thread_event))
            self.serial_thread.start()


    def write_cmd(self, cmd):
        self.serial_interface.write(cmd)
        self.serial_interface.flush()

    def __del__(self):
        self.serial_thread_event.set()
        self.serial_thread.join(2)
        self.serial_thread_event.clear()
        print "Goodbye"


def sigint_handler(signal, frame):
    print "Ctrl-C. Aborting"
    sys.exit(1)


try:
    logging.basicConfig(level=logging.DEBUG)
    signal.signal(signal.SIGINT, sigint_handler)

    testController = MotherboardTestController()

    listen_socket_fd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    listen_socket_fd.bind(('', 12345))
    print "Listening for connections..."
    listen_socket_fd.listen(1)
    (socket_fd, addr) = listen_socket_fd.accept()
    print 'Connected to', addr
    cmd = ''
    while 1:
        try:
            cmd = socket_fd.recv(32)
            if not cmd:
                print 'Lost connection. Listening...'
                (socket_fd, addr) = listen_socket_fd.accept()
                print 'Connected to', addr
            else:
                testController.write_cmd(cmd)
                print cmd
        except socket.error, (errno, msg):
```

```
            print 'Lost␣connection␣due␣to␣exception.␣Listening...'
            (socket_fd, addr) = listen_socket_fd.accept()
            print 'Connected␣to', addr

        except KeyboardInterrupt, k:
            print 'Ctrl-C!␣Exiting..'
            sys.exit(0)

except Exception, ex:
    logging.exception(ex)

socket_fd.close()
print "Socket␣closed"
sys.exit(0)


import serial
import sniffer
import time
import socket
from subprocess import Popen, PIPE
import ctypes

def kill(pid):
    """kill function for Win32"""
    kernel32 = ctypes.windll.kernel32
    handle = kernel32.OpenProcess(1, 0, pid)
    return (0 != kernel32.TerminateProcess(handle, 0))

ANT_PC_APP_PATH = "ANT␣PC␣app\DEMO_LIB.exe"
BTLE_PC_APP_PATH = "BTLE␣PC␣app\masterEmulatorTester.exe"
ANT_PC_LOG_PATH = "ANT␣PC␣app␣log␣files"
BTLE_PC_LOG_PATH = "BTLE␣PC␣app␣log␣files"
HOST = "192.9.200.188"
PORT = 12345




class BTLETestParams:
    connection_interval = 4
    channel_map = 0x1FFFFFFFFF
    master_tx_power = 3          # 0 = -18dB, 1 = -12dB, 2 = -6dB, 3 = 0dB
    slave_tx_power = 3
    search_timeout = 15

    def __init__(self):
        connection_interval = 4
        channel_map = 0x1FFFFFFFFF
        master_tx_power = 3
        slave_tx_power = 3
        search_timeout = 15


class ANTTestParams:
    pc_channel_type = 1          # 0 is master, 1 is slave
    peripheral_channel_type = 0
    freqs = (3, 39, 75)
    enable_freq_agility = 1
    channel_period = 4
    peripheral_tx_power = 3      # 0 = -20dB, 1 = -10dB, 2 = -5dB, 3 = 0dB
    pc_tx_power = 3

    def __init__(self):
        pc_channel_type = 1
        peripheral_channel_type = 0
        freqs = (3, 39, 75)
        enable_freq_agility = 1
        channel_period = 4
        peripheral_tx_power = 3
        pc_tx_power = 3



class Test:

    test_duration = 60                  # test duration in seconds
    test_name = None

    ant_pc_app = None
    btle_pc_app = None

    sniffer_module = None

    ant_pc_log = None
```

```
btle_pc_log = None
motherboard_log = None

sock_fd = None

def __init__(self):
    self.sock_fd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.sock_fd.connect((HOST, PORT))

def __del__(self):
    if self.sock_fd:
        self.sock_fd.close()
    if self.ant_pc_log:
        self.ant_pc_log.close()
    if self.btle_pc_log:
        self.btle_pc_log.close()
    if self.sniffer_module:
        self.sniffer_module.TerminateConnectionToAutomationServer()


def socket_write_cmd(self, cmd):
    self.sock_fd.send(cmd)
    time.sleep(0.5)


def motherboard_ant_setup(self, ant_test_params):
    self.socket_write_cmd("ant_type␣%d\n" % ant_test_params.peripheral_channel_type)
    self.socket_write_cmd("ant_period␣%d\n" % ant_test_params.channel_period)
    self.socket_write_cmd("ant_freq␣%d␣%d␣%d\n" % (ant_test_params.freqs[0], ant_test_params.freqs[1],
          ant_test_params.freqs[2]))
    self.socket_write_cmd("ant_tx_power␣%d\n" % ant_test_params.peripheral_tx_power)
    self.socket_write_cmd("ant_enable_freq_agility␣%d\n" % ant_test_params.enable_freq_agility)
    print "Motherboard␣ANT␣setup␣complete!"

def motherboard_btle_setup(self, btle_test_params):
    self.socket_write_cmd("btle_tx_power␣%d\n" % btle_test_params.slave_tx_power)
    self.socket_write_cmd("btle_adv_timeout␣%d\n" % btle_test_params.search_timeout)
    print "Motherboard␣BTLE␣setup␣complete!"


def sniffer_setup(self):
    print "Setting␣up␣sniffer.."
    self.sniffer_module = sniffer.Sniffer()
    self.sniffer_module.ConnectToAutomationServer()
    self.sniffer_module.StartFTS()              # FTS should be started manually before running this script
    self.sniffer_module.SendConfig()
    print "Sniffer␣setup␣complete!"

def ant_pc_app_setup(self, ant_test_params):
    self.ant_pc_log = open(("%s\%s.txt" % (ANT_PC_LOG_PATH, self.test_name)), "w")
    ant_cmd = ("%s␣%d␣%d␣%d␣%d␣%d␣%d␣%d" % (ANT_PC_APP_PATH, ant_test_params.pc_channel_type,
          ant_test_params.channel_period, ant_test_params.freqs[0], ant_test_params.freqs[1],
          ant_test_params.freqs[2], ant_test_params.enable_freq_agility, ant_test_params.pc_tx_power))
    self.ant_pc_app = Popen(ant_cmd, bufsize=-1, stdin=PIPE, stdout=self.ant_pc_log, stderr=self.ant_pc_log,
           cwd="ANT␣PC␣app")
    print "ANT␣PC␣setup␣complete!"

def btle_pc_app_setup(self, btle_test_params):
    self.btle_pc_log = open(("%s\%s.txt" % (BTLE_PC_LOG_PATH, self.test_name)), "w")
    btle_cmd = ("%s␣%x␣%d␣%d␣%d" % (BTLE_PC_APP_PATH, btle_test_params.channel_map, btle_test_params.
          connection_interval, btle_test_params.search_timeout, btle_test_params.master_tx_power))
    self.btle_pc_app = Popen(btle_cmd, bufsize=-1, stdin=PIPE, stdout=self.btle_pc_log, stderr=self.
          btle_pc_log, cwd="BTLE␣PC␣app")
    time.sleep(10)
    print "BTLE␣PC␣setup␣complete!"

def run_test(self, ant_test_params, btle_test_params):
    print "Starting␣test␣%s.." % self.test_name
    self.socket_write_cmd("reset\n")
    time.sleep(1)
    self.socket_write_cmd("=========\n%s\n=========\n" % self.test_name)

    if (btle_test_params != None):
        self.motherboard_btle_setup(btle_test_params)
        self.btle_pc_app_setup(btle_test_params)
        # Start sniffer capture
        if (self.sniffer_module != None):
            self.sniffer_module.StartCapture()
        self.btle_pc_app.stdin.write("start\r\n")
        self.btle_pc_app.stdin.flush()


    if (ant_test_params != None):
        self.motherboard_ant_setup(ant_test_params)
        self.ant_pc_app_setup(ant_test_params)
        self.ant_pc_app.stdin.write("start\r\n")
```

```
            self.ant_pc_app.stdin.flush()


        # Start motherboard application
        if ((ant_test_params != None) and (btle_test_params != None)):
            self.socket_write_cmd("start\n")
        elif (ant_test_params != None):
            self.socket_write_cmd("ant_start\n")
        elif (btle_test_params != None):
            self.socket_write_cmd("btle_start\n")

        # Wait until test is complete
        print "Test␣%s␣successfully␣started!" % self.test_name
        time.sleep(self.test_duration)
        print "Test␣complete!"

        # Transmissions are stopped by stopping the motherboard application
        self.socket_write_cmd("stop\n")


        # Stop PC applications. Dont exit if an exception is caught..
        try:
            if (ant_test_params != None):
                self.ant_pc_app.stdin.write("stop\r\n")
                self.ant_pc_app.stdin.flush()
                time.sleep(3)
                self.ant_pc_app.stdin.write("exit\r\n")
                self.ant_pc_app.stdin.flush()

            if (btle_test_params != None):
                self.btle_pc_app.stdin.write("stop\r\n")
                self.btle_pc_app.stdin.flush()
                time.sleep(3)
            if (self.sniffer_module != None):
                self.sniffer_module.StopCapture()
                self.sniffer_module.SaveCapture("%s" % self.test_name)
                self.sniffer_module.ClearBuffer()
        except Exception, ex:
            print ex
            if (ant_test_params != None and self.ant_pc_app.poll() == None):
                kill(self.ant_pc_app.pid)
            if (btle_test_params != None and self.btle_pc_app.poll() == None):
                kill(self.btle_pc_app.pid)

from pc_test import ANTTestParams, BTLETestParams, Test
import sys
import logging
import signal

def sigint_handler(signal, frame):
    print "Ctrl-C.␣Aborting"
    sys.exit(1)


def RunANTBTLEDefaultTests():
    channel_periods = (1, 5, 10, 15, 20, 25, 30, 50)
    test = Test()
    ant_test_params = ANTTestParams()
    btle_test_params = BTLETestParams()

    test.test_duration = 60
    ant_test_params.freqs = (66, 0, 0)
    ant_test_params.enable_freq_agility = 0
    ant_test_params.pc_channel_type = 1
    ant_test_params.peripheral_channel_type = 0

    #test.sniffer_setup()
    for i in range(0, len(channel_periods)):
        test.test_name = "%s%d" % ("ant_btle_default_test_", i+1)
        if (channel_periods[i] < 30):
            btle_test_params.connection_interval = channel_periods[i]
        else:
            btle_test_params.connection_interval = 30
        ant_test_params.channel_period = channel_periods[i]
        test.run_test(ant_test_params, btle_test_params)

    ant_test_params.pc_channel_type = 0
    ant_test_params.peripheral_channel_type = 1

    for i in range(len(channel_periods)+1, 2*len(channel_periods)+1):
        test.test_name = "%s%d" % ("ant_btle_default_test_", i)
        if (channel_periods[i-len(channel_periods)-1] < 30):
            btle_test_params.connection_interval = channel_periods[i-len(channel_periods)-1]
        else:
            btle_test_params.connection_interval = 30
        ant_test_params.channel_period = channel_periods[i-len(channel_periods)-1]
```

```
        test.run_test(ant_test_params, btle_test_params)



def RunANTReferenceTests():
    channel_periods = (1, 5, 10, 15, 20, 25, 30, 50)
    test = Test()
    ant_test_params = ANTTestParams()

    ant_test_params.freqs = (66, 0, 0)
    ant_test_params.enable_freq_agility = 0
    ant_test_params.pc_channel_type = 1
    ant_test_params.peripheral_channel_type = 0

    for i in range(0, len(channel_periods)):
        test.test_name = "%s%d" % ("ant_reference_test_", i+1)
        ant_test_params.channel_period = channel_periods[i]
        test.run_test(ant_test_params, None)

    ant_test_params.pc_channel_type = 0
    ant_test_params.peripheral_channel_type = 1

    for i in range(len(channel_periods)+1, 2*len(channel_periods)+1):
        test.test_name = "%s%d" % ("ant_reference_test_", i)
        ant_test_params.channel_period = channel_periods[i-len(channel_periods)-1]
        test.run_test(ant_test_params, None)



def RunBTLEReferenceTests():
    channel_periods = (1, 5, 10, 15, 20, 25, 30)
    test = Test()
    btle_test_params = BTLETestParams()
    test.test_duration = 60

    for i in range(0, len(channel_periods)):
        test.test_name = "%s%d" % ("btle_reference_test_", i+1)
        btle_test_params.connection_interval = channel_periods[i]
        test.run_test(None, btle_test_params)



def RunANTBTLEStressTests():
    test = Test()
    test.test_duration = 60

    ant_test_params = ANTTestParams()
    btle_test_params = BTLETestParams()

    ant_test_params.channel_period = 50
    ant_test_params.freqs = (66, 0, 0)
    ant_test_params.enable_freq_agility = 0
    btle_test_params.connection_interval = 30
    btle_test_params.channel_map = 0x0040000000

    ant_channel_types = (0, 1, 1, 0, 0, 1, 1, 0)
    ant_tx_powers = (3, 3, 0, 0)
    btle_tx_powers = (0, 0, 3, 3)

    for i in range(0, 4):
        test.test_name = "%s%d" % ("ant_btle_stress_test_", i+1)
        ant_test_params.peripheral_channel_type = ant_channel_types[i*2]
        ant_test_params.pc_channel_type = ant_channel_types[i*2+1]
        ant_test_params.peripheral_tx_power = ant_tx_powers[i]
        ant_test_params.pc_tx_power = ant_tx_powers[i]
        btle_test_params.master_tx_power = btle_tx_powers[i]
        btle_test_params.slave_tx_power = btle_tx_powers[i]
        test.run_test(ant_test_params, btle_test_params)



def RunANTBTLEOptimizedTests():
    test = Test()
    test.test_duration = 60

    ant_test_params = ANTTestParams()
    btle_test_params = BTLETestParams()

    channel_periods = (1, 5, 10, 15, 20, 25, 30, 50)

    ant_test_params.freqs = (66, 0, 0)
    ant_test_params.enable_freq_agility = 0 # does not work correctly...
    #ant_test_params.peripheral_channel_type = 0
```

```
    #ant_test_params.pc_channel_type = 1
    btle_test_params.channel_map = 0x1F1FFFFFFF

    #test.sniffer_setup()
    #for i in range(0, len(channel_periods)):
        #test.test_name = "%s%d" % ("ant_btle_optimized_test_", i+1)
        #if (channel_periods[i] < 30):
            #btle_test_params.connection_interval = channel_periods[i]
        #else:
            #btle_test_params.connection_interval = 30
        #ant_test_params.channel_period = channel_periods[i]
        #test.run_test(ant_test_params, btle_test_params)

    ant_test_params.pc_channel_type = 0
    ant_test_params.peripheral_channel_type = 1

    for i in range(len(channel_periods)+1, 2*len(channel_periods)+1):
        test.test_name = "%s%d" % ("ant_btle_optimized_test_", i)
        if (channel_periods[i-len(channel_periods)-1] < 30):
            btle_test_params.connection_interval = channel_periods[i-len(channel_periods)-1]
        else:
            btle_test_params.connection_interval = 30
        ant_test_params.channel_period = channel_periods[i-len(channel_periods)-1]
        test.run_test(ant_test_params, btle_test_params)


def RunANTWiFiTests():
    channel_periods = (1, 5, 10, 15, 20, 25, 30, 50)
    test = Test()
    ant_test_params = ANTTestParams()

    ant_test_params.freqs = (46, 0, 0)
    ant_test_params.enable_freq_agility = 0
    ant_test_params.pc_channel_type = 1
    ant_test_params.peripheral_channel_type = 0

    for i in range(0, len(channel_periods)):
        test.test_name = "%s%d" % ("ant_wifi_test_", i+1)
        ant_test_params.channel_period = channel_periods[i]
        test.run_test(ant_test_params, None)

    ant_test_params.pc_channel_type = 0
    ant_test_params.peripheral_channel_type = 1

    for i in range(len(channel_periods)+1, 2*len(channel_periods)+1):
        test.test_name = "%s%d" % ("ant_wifi_test_", i)
        ant_test_params.channel_period = channel_periods[i-len(channel_periods)-1]
        test.run_test(ant_test_params, None)


def RunBTLEWiFiTests():
    channel_periods = (1, 5, 10, 15, 20, 25, 30)
    test = Test()
    btle_test_params = BTLETestParams()
    btle_test_params.channel_map = 0x1fffffffff
    test.test_duration = 60
    #test.sniffer_setup()
    for i in range(0, len(channel_periods)):
        test.test_name = "%s%d" % ("btle_wifi_test_", i+1)
        btle_test_params.connection_interval = channel_periods[i]
        test.run_test(None, btle_test_params)



def RunANTBTLEWiFiOptimizedTests():
    channel_periods = (1, 5, 10, 15, 20, 25, 30, 50)
    test = Test()
    btle_test_params = BTLETestParams()
    btle_test_params.channel_map = 0x1F1FC007FF

    ant_test_params = ANTTestParams()
    ant_test_params.freqs = (66, 0, 0)
    ant_test_params.enable_freq_agility = 0
    ant_test_params.pc_channel_type = 1
    ant_test_params.peripheral_channel_type = 0

    #test.sniffer_setup()
    for i in range(0, len(channel_periods)):
        test.test_name = "%s%d" % ("ant_btle_optimized_wifi_test_", i+1)
        ant_test_params.channel_period = channel_periods[i]
        if (channel_periods[i] < 30):
            btle_test_params.connection_interval = channel_periods[i]
        else:
            btle_test_params.connection_interval = 30
        test.run_test(ant_test_params, btle_test_params)
```

```
    ant_test_params.pc_channel_type = 0
    ant_test_params.peripheral_channel_type = 1

    for i in range(len(channel_periods)+1, 2*len(channel_periods)+1):
        test.test_name = "%s%d" % ("ant_btle_optimized_wifi_test_", i)
        ant_test_params.channel_period = channel_periods[i-len(channel_periods)-1]
        if (channel_periods[i-len(channel_periods)-1] < 30):
            btle_test_params.connection_interval = channel_periods[i-len(channel_periods)-1]
        else:
            btle_test_params.connection_interval = 30
        test.run_test(ant_test_params, btle_test_params)


def RunANTBTLEWiFiStressTests():
    channel_periods = (1, 5, 10, 15, 20, 25, 30, 50)
    test = Test()
    btle_test_params = BTLETestParams()
    btle_test_params.channel_map = 0x1fffffffff

    ant_test_params = ANTTestParams()
    ant_test_params.freqs = (46, 0, 0)
    ant_test_params.enable_freq_agility = 0
    ant_test_params.pc_channel_type = 1
    ant_test_params.peripheral_channel_type = 0

    #test.sniffer_setup()
    #for i in range(0, len(channel_periods)):
        #test.test_name = "%s%d" % ("ant_btle_stress_wifi_test_", i+1)
        #ant_test_params.channel_period = channel_periods[i]
        #if (channel_periods[i] < 30):
            #btle_test_params.connection_interval = channel_periods[i]
        #else:
            #btle_test_params.connection_interval = 30
        #test.run_test(ant_test_params, btle_test_params)

    ant_test_params.pc_channel_type = 0
    ant_test_params.peripheral_channel_type = 1

    for i in range(16, 2*len(channel_periods)+1):
        test.test_name = "%s%d" % ("ant_btle_stress_wifi_test_", i)
        ant_test_params.channel_period = channel_periods[i-len(channel_periods)-1]
        if (channel_periods[i-len(channel_periods)-1] < 30):
            btle_test_params.connection_interval = channel_periods[i-len(channel_periods)-1]
        else:
            btle_test_params.connection_interval = 30
        test.run_test(ant_test_params, btle_test_params)


def RunANTBTLEWiFiMoreOptimizedTests():
    channel_periods = (1, 5, 10, 15, 20, 25, 30, 50)
    test = Test()
    btle_test_params = BTLETestParams()
    btle_test_params.channel_map = 0x1C0000003F

    ant_test_params = ANTTestParams()
    ant_test_params.freqs = (16, 0, 0)
    ant_test_params.enable_freq_agility = 0
    ant_test_params.pc_channel_type = 1
    ant_test_params.peripheral_channel_type = 0

    #test.sniffer_setup()
    for i in range(0, len(channel_periods)):
        test.test_name = "%s%d" % ("ant_btle_more_optimized_wifi_test_", i+1)
        ant_test_params.channel_period = channel_periods[i]
        if (channel_periods[i] < 30):
            btle_test_params.connection_interval = channel_periods[i]
        else:
            btle_test_params.connection_interval = 30
        test.run_test(ant_test_params, btle_test_params)

    ant_test_params.pc_channel_type = 0
    ant_test_params.peripheral_channel_type = 1

    for i in range(len(channel_periods)+1, 2*len(channel_periods)+1):
        test.test_name = "%s%d" % ("ant_btle_more_optimized_wifi_test_", i)
        ant_test_params.channel_period = channel_periods[i-len(channel_periods)-1]
        if (channel_periods[i-len(channel_periods)-1] < 30):
            btle_test_params.connection_interval = channel_periods[i-len(channel_periods)-1]
        else:
            btle_test_params.connection_interval = 30
        test.run_test(ant_test_params, btle_test_params)



logging.basicConfig(level=logging.DEBUG)
```

```
signal.signal(signal.SIGINT, sigint_handler)

try:
    #RunANTReferenceTests()
    #RunBTLEReferenceTests()
    #RunANTBTLEDefaultTests()
    #RunANTBTLEOptimizedTests()
    #RunANTBTLEStressTests()

    #RunANTWiFiTests()
    #RunBTLEWiFiTests()
    #RunANTBTLEWiFiStressTests()
    RunANTBTLEWiFiMoreOptimizedTests()

except Exception, ex:
    logging.exception(ex)
    sys.exit(1)
```

References

[1] The ant alliance. `http://www.thisisant.com`.

[2] Osi model. `http://en.wikipedia.org/wiki/OSI_model`. Retrieved 17. June 2011.

[3] Ieee 802.15 wpan task group 2. `http://www.ieee802.org/15/pub/TG2.html`, June 2011.

[4] The ANT Alliance. Ant frequently asked questions. `http://www.thisisant.com/images/Resources/PDF/ant_qandas.pdf`, February 2011.

[5] The ANT Alliance. *ANT Frequency Agility*, 2.0 edition. ANT Application Note no 10.

[6] The ANT Alliance. *ANT Message Protocol and Usage*, 4.1 edition.

[7] The ANT Alliance. *Interfacing with ANT General Purpose Chipsets and Modules*, 2.1 edition.

[8] Nordic Semiconductor AS. nrf8001 product brief. `www.nordicsemi.com/eng/nordic/download_resource/8137/1/9265588`.

[9] Sebastian Barnowski. Ant protocol basics. `http://www.thisisant.com`, September 2010. Slides from ANT+ Alliance symposiurm 2010.

[10] Bluetooth Special Interest Group. *Bluetooth Specification*, 4.0 edition. Volumes 1-6.

[11] Kenneth R. Carter. Unlicensed to kill: a brief history of the part 15 rules. *Info: The Journal of Policy, Regulation and Safety for Telecommunications, Information and Media*, 2009.

[12] M. Cho-Hoi Chek and Yu-Kwong Kwok. On adaptive frequency hopping to combat coexistence interference between bluetooth and ieee 802.11b with practical resource constraints. In *Parallel Architectures, Algorithms and Networks, 2004. Proceedings. 7th International Symposium on*, pages 391 – 396, may 2004.

[13] N. Golmie, O. Rebala, and N. Chevrollier. Bluetooth adaptive frequency hopping and scheduling. In *Military Communications Conference, 2003. MILCOM 2003. IEEE*, volume 2, pages 1138 – 1142 Vol.2, oct. 2003.

[14] I. Howitt. Wlan and wpan coexistence in ul band. *Vehicular Technology, IEEE Transactions on*, 50(4):1114 –1124, jul 2001.

[15] Ivan Howitt and Jose A. Gutierrez. 802.15.4 low rate: Wireless personal area network coexistence issues. *IEEE Wireless Communications and Networking*, 2003.

[16] Nick Hunn. *Essentials of Short-Range Wireless*. Number ISBN 978-0-521-76069. Cambridge University Press, 2010.

[17] Adrian Stephens Jim Lansford and Ron Nevo. Wi-fi (802.11b) and bluetooth: Enabling coexistence. *IEEE Network*, 2001.

[18] Carl Karlsson. Impulsive noise modeling and coexistence study of ieee 802.11 and bluetooth. Master's thesis, University of Gavle, 2008.

[19] Norges Lover. Forskrift om generelle tillatelser til bruk av frekvenser. `http://www.lovdata.no`. Norwegian law for regulating the use of radio frequencies.

[20] John Torjus Flåm. Anchorpoint analysis. Internal Nordic Semiconductor document, April 2007.

[21] Michael J. Marcus. Wi-fi and bluetooth: the path from carter and reagan-era faith in deregulation to widespread products impacting our world. *Info: The Journal of Policy, Regulation and Safety for Telecommunications, Information and Media*, 2009.

[22] Nordic Semiconductor AS. *nRF8001 Single-chip Bluetooth low energy solution*, preliminary product specification 0.9 edition.

[23] M. Pandey, D. Delorey, Qiuyi Duan, Lei Wang, C. Knutson, D. Zappala, and R. Woodings. Ria: An rf interference avoidance algorithm for heterogeneous wireless networks. In *Wireless Communications and Networking Conference, 2007.WCNC 2007. IEEE*, pages 4051 –4056, march 2007.

[24] A. Sikora and V.F. Groza. Coexistence of ieee802.15.4 with other systems in the 2.4 ghz-ism-band. In *Instrumentation and Measurement Technology Conference, 2005. IMTC 2005. Proceedings of the IEEE*, volume 3, pages 1786 –1791, may 2005.

[25] Hubert Zimmermann. Osi reference model - the iso model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 1980.