UNIVERSITÄT DUISBURG-ESSEN
FAKULTÄT FÜR WIRTSCHAFTSWISSENSCHAFTEN
INSTITUT FÜR INFORMATIK UND WIRTSCHAFTSINFORMATIK
LEHRSTUHL FÜR PERVASIVE COMPUTING

Bachelorarbeit

Evaluation of a lightweight debugging platform for mobile sensor networks

Michael Krane
Matrikelnummer: 2233018

Networked Embedded Systems Group
Institut für Informatik und Wirtschaftsinformatik
Fakultät für Wirtschaftswissenschaften
Universität Duisburg-Essen

September 27, 2015

Erstprüfer: Prof. Dr. Pedro José Marrón
Zweitprüfer: Prof. Dr. Gregor Schiele
Zeitraum: 7.Juli 2015 - 29.September 2015

Contents

1. Introduction **3**
1.1. Related work 4

2. Technical Background **5**
2.1. SHAMPU 5
2.2. ANT 6
2.2.1. ANT Topology 6
2.2.2. ANT Channels 7
2.2.3. ANT Communication 9
2.2.4. ANT messages 10
2.3. ANT library 10
2.4. ANTAP1MxIB RF 11

3. Evaluation of SHAMPU **13**
3.1. Common experiment parameters 15
3.2. Experiment 1: Broadcast Data Transfer between two nodes 17
3.3. Experiment 2: Broadcast Data Transfer with two channels 22
3.4. Experiment 3: Acknowledge Data Transfer between two nodes 27
3.5. Experiment 4: Acknowledge Data Transfer delay 31
3.6. Experiment 5: Burst Data Transfer between two nodes 35
3.7. Experiment 6: Maximum communication Range 38
3.8. Maximum Data Throughput 41

4. Discussion **43**
4.1. Summary 43
4.2. SHAMPU network set up 44
4.3. Future Work 45

A. Sourcecode **47**

Bibliography **67**

List of Figures

2.1. Overview of the SHAMPU Framework [1] 5
2.2. OSI-Layer vs. ANT Protocol[2] 6
2.3. Example ANT Topologies[3] 7
2.4. ANT channel establishment[3] 8
2.5. ANT channel communication [3] 9
2.6. ANT message structure[3] 10
2.7. SHAMPU base station 11

3.1. Experiment setup 15
3.2. Topology experiment 1 17
3.3. Broadcast data throughput (0.5Hz - 198.6Hz) 19
3.4. Broadcast data throughput (129Hz - 198.6Hz) 20
3.5. Broadcast data throughput over time (198.6Hz) 21
3.6. Topology experiment 2 22
3.7. Broadcast data throughput - 2 channels (0.5Hz - 129Hz) 25
3.8. Broadcast data throughput - 2 channels (64Hz - 113Hz) 26
3.9. Topology experiment 3 27
3.10. Acknowledge data throughput (0.5Hz - 129Hz) 29
3.11. Acknowledge data throughput (82Hz - 113Hz) 29
3.12. Topology experiment 4 31
3.13. Acknowledge delay (0.5Hz - 8Hz) 33
3.14. Acknowledge delay (16Hz - 129Hz) 33
3.15. Topology experiment 5 35
3.16. Burst data throughput 37
3.17. Topology experiment 6 38
3.18. Maximum communication range 40

Abstract

In this thesis, we evaluate a component of SHAMPU (Single chip Host for Autonomous Mote Programming over USB), developed at the University of Duisburg-Essen. SHAMPU is a framework which allows the monitoring, debugging, and reprogramming of wireless sensor networks (WSN). Our focus for this evaluation is on the wireless communication between SHAMPU nodes, for which the ANT protocol is used. We designed and ran several different experiments in order to determine the capabilities of the ANT-chip in use.

Chapter 1.

Introduction

In the present years the number of Cyber Physical Systems (CPSs) has drastically increased. This increase is mostly due to the emergence of the Internet of Things (IoT), as well as to the existence of several user friendly kits, such as the RaspberryPi or the Arduino. With this recent trend new challenges are created in the monitoring and debugging of the above mentioned devices. This is especially difficult for applications, where sensors are deployed in a hard to reach area. Moreover, sensors can often fulfill multi-purpose roles and might have to be reconfigured for different tasks.

In order to address both of these problems, SHAMPU (Single chip Host for Autonomous Mote Programming over USB)[1] was developed at the University of Duisburg-Essen. SHAMPU is a design framework for monitoring and reprogramming WSNs. The main goal of SHAMPU is to be as small, lightweight and energy efficient as possible. Furthermore, a SHAMPU node is OS independent and can easily be attached to an existing node over USB. The SHAMPU nodes have multiple ways to transmit data between them, one of which is a wireless interface. In the current configuration, an ANT[3] radio chip is used to transfer data between the SHAMPU nodes. Additionally, the nodes are connected to a base station which not only acts as a central data sink, but is also capable of pushing commands and data to the SHAMPU nodes. These wireless capabilities make it possible to remotely monitor, debug, and even reconfigure already deployed sensor nodes.

In this thesis we evaluate the wireless capabilities of the SHAMPU framework. For that purpose we investigate use-cases for the different tasks the SHAMPU framework can perform: Program a device, collect data during operation, and interact with the system itself. For each of the use-cases we designed different experiments to assess how well ANT performs.

1.1. Related work

There are several different WSN test beds available. Each of these test beds is capable of monitoring and debugging an attached sensor node. The main difference between them is the size and specific role they were designed for.

Some test beds like FlockLab [4] allow to attach a wide array of different sensors and use a JTAG interface to be able to precisely capture debugging and timing information.

However, these types of test beds rely on a wired connection to interface with the test bed. Additionally, the platform itself has a large form factor. This makes it exceedingly difficult to test and debug already deployed WSNs since it might not be possible to easily get to the sensor in the planted location.

Other available solutions address this infrastructure dependency by attaching the test bed directly to the node and use a wireless connection to communicate with it. An example is Sensei-UU [5], which uses a wireless 802.11 network. This setup allows the WSN to be tested and debugged while it is deployed. One drawback is the power draw of 802.11 devices, which is huge compared to other technologies. To run the node itself on an external power source would not be a problem, but the use of a battery to power the node will hardly be feasible.

To address the power issue, it is possible to use a different technology which offers a lower power mode. Two examples are BTNodes [6] or Smart-Its [7], which use a Bluetooth connection. The structure of the network itself is similar to Sensei-UU: each node in the network has its own test bed attached and the test beds communicate with a central base station.

The main problem with Bluetooth is the limited size of the network, which makes it difficult to set up and use more complex network topologies. Also, Bluetooth communication is always 1:1 and does not allow broadcasts. The newest version of Bluetooth addresses the network size with the introduction of ScatterNets, but the setup and maintenance of the network remains challenging.

None of the above mentioned test beds exactly fit the design goals of SHAMPU. The test bed has to be directly attached to a fixed infrastructure, either because it has no wireless capabilities or because the power consumption is too high. BTNodes could be a possible alternative for SHAMPU. However, Bluetooth connections are not well suited for the use in WSNs since they mostly use unicast.

Chapter 2.

Technical Background

2.1. SHAMPU

SHAMPU[1] is a design framework which allows remote debugging and reprogramming of sensor nodes. Its main advantages over other testbeds (see section 1.1) are portability, low cost, small size and low energy consumption. SHAMPU is used as an extension to an already existing sensor node. The only requirement is that the node provides an USB-Interface which is connected to SHAMPU. This single connection to the attached node makes SHAMPU OS independent.

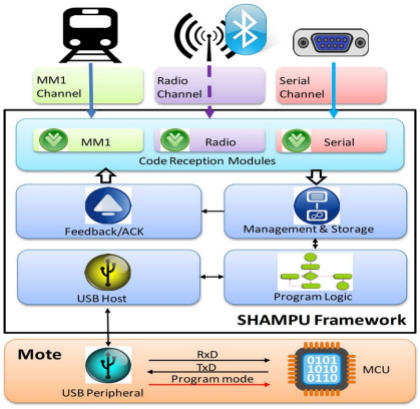

Figure 2.1.: Overview of the SHAMPU Framework [1]

The SHAMPU framework (see Figure 2.1) itself is split into multiple modules. The most important part for this thesis is the Code Reception Module, which allows the use of different protocols to connect to and communicate with the SHAMPU device. One option for the wireless communication with a SHAMPU device is the ANT protocol, on which we will focus in this evaluation.

2.2. ANT

ANT [3] is a wireless protocol which operates in the 2.4 GHz ISM Band. It was originally developed in 2003 by Dynastream Innovations Inc. for the use in wireless sensors. The ANT protocol is designed for the use in low power WSNs, focusing on small size and ease of use.

One of the advantages ANT has over other protocols, such as Bluetooth or ZigBee, is the high level of abstraction the ANT Protocol provides.

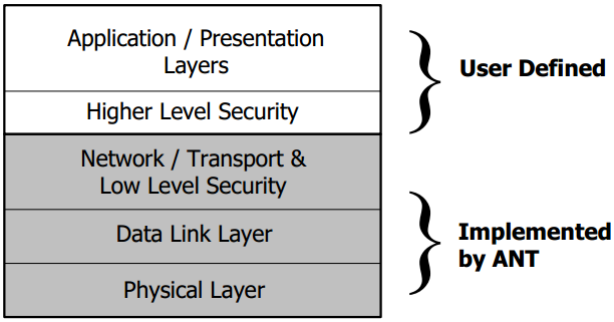

Figure 2.2.: OSI-Layer vs. ANT Protocol[2]

This level of abstraction is achieved by incorporating the first 4 OSI-layers (see Figure 2.2) into the ANT protocol, thus allowing even low-cost microcontrollers to set up and maintain complex wireless networks, since all the details of the communication are handled by the ANT-chip.

2.2.1. ANT Topology

In order for the ANT protocol to work each mote needs to be part of a network. As shown in figure 2.3 the ANT protocol can be used to create simple or considerably more complex networks. Each mote inside a network is called an ANT node. In order for two nodes to communicate with each other they need to be connected via a channel.

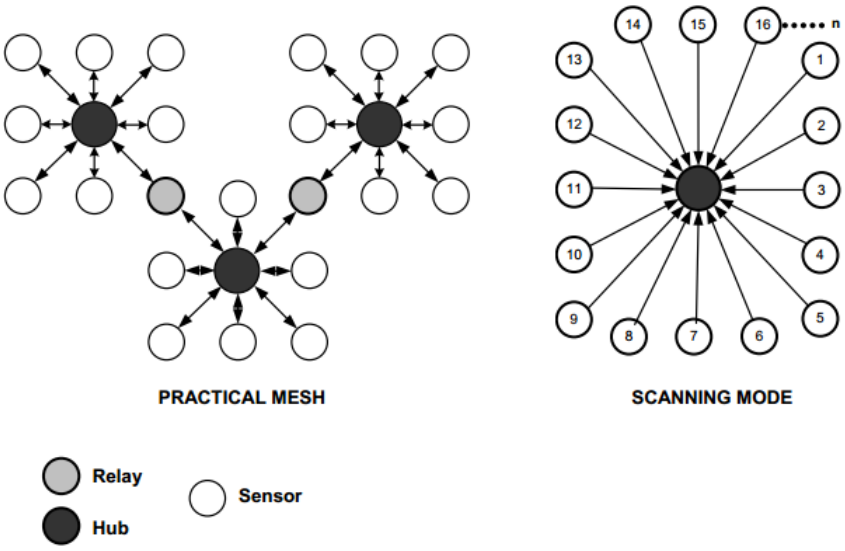

Figure 2.3.: Example ANT Topologies[3]

2.2.2. ANT Channels

The ANT protocol uses virtual channels to allow different nodes to transmit data, without interfering with each other. Each channel can be assigned to a 1 MHz wide RF frequency band between 2400 MHz and 2524 MHz.[3] In theory this allows 125 different channels to exist without interference between them. It is possible for multiple channels to use the same frequency depending on the rate of transmission, since the data rate of each 1MHz band is limited to 1MBps. To avoid interference between nodes which use the same frequency isochronous self adjusting Time Division Multiple Access (TDMA) is used. TDMA allows the ANT protocol to adjust the transmit timings of the channels individually.

Each node has to specify how the assigned channel is being used. The node can either be a master or a slave node. While a master node mostly sends data and a slave node mostly receives data, the slave retains the ability to respond to the incoming data.

The ANT protocol further differentiates between two different channel types:

Independent Channels
: Independent Channels are used if there is only one node which transmits data. There is no limit to the amount of slave devices which receive messages. Furthermore, the message being sent out is broadcast to all nodes. Tt is not possible to address only a specific node.

Shared Channels

Shared Channels are used if there is more than one node which sends data. This type of channel is facilitated by the use of a Shared Channel Address, which in turn reduces the amount of data that can be transmitted at a time. All ANT nodes still receive every message, but only pass on messages which have a matching address. The Channel master can decide to either use one or two bytes as the address, which allows for either 255 or 65535 slave devices in the same channel.

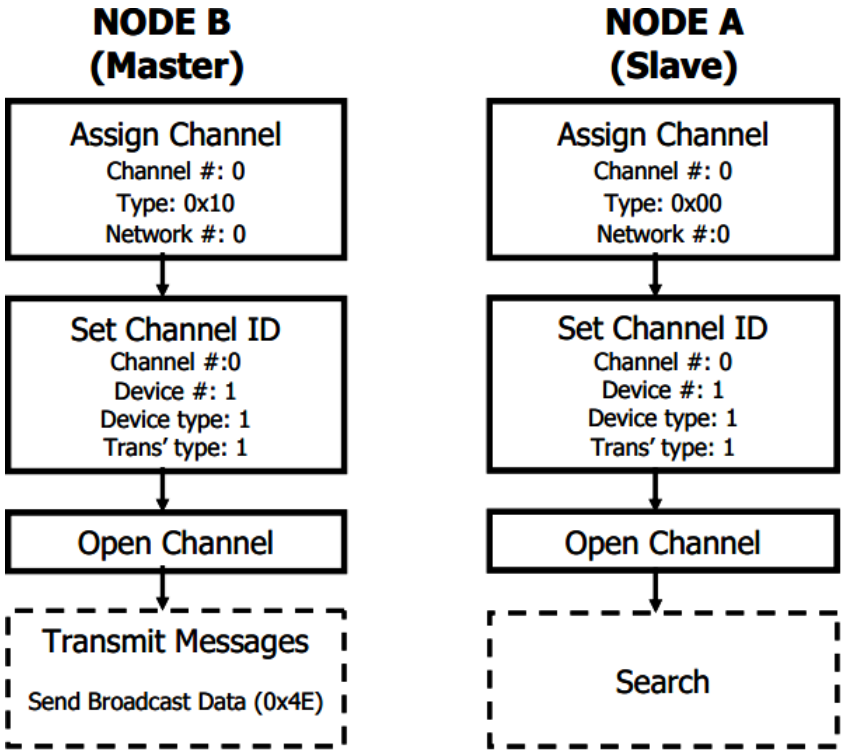

Figure 2.4.: ANT channel establishment[3]

Figure 2.4 shows the procedure of opening up a channel between two ANT nodes. In the first step the channel configuration is set, most important in this context being the channel type: 0x00 signifies that the node is opening the channel as a slave while 0x10 means that the node acts as a master. The next step is to set the Channel ID. Since it is possible for two nodes to transmit on the same frequency, the device number, device type and transmission type of the slave node need to match the values of the master it wants to connect to. It is possible, however, to set wild card values, which allows the slave to connect to any device sending on the same frequency. Before the final step, it is optionally possible to set the frequency of the channel, the power setting and the message period. These steps are not mandatory though, since ANT defaults to fixed

values: 2466 Hz transmission frequency, 0dBm power and a message period of 8192. The last step constitutes of the opening of the channel itself. The master should open the channel before the slave to ensure a successful connection.

2.2.3. ANT Communication

The ANT protocol supports three different data types: broadcast, acknowledge and burst. The data type is not part of the channel configuration, thus channels are able to use any combination of data types. The only exceptions are legacy unidirectional channels, which can only send broadcast data. The various data types differ in the way data is handled and transmitted.

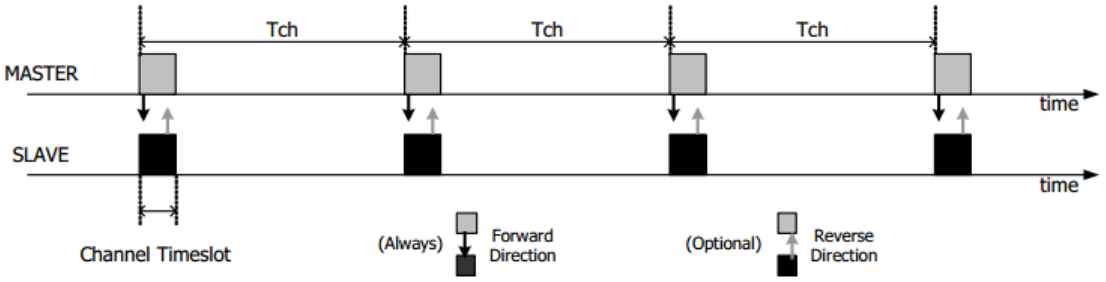

Figure 2.5.: ANT channel communication [3]

- Broadcast data
  Broadcast data represents the most basic data type and, at the same time, the default. To start a broadcast transmission, the command needs to be issued just once, since the last sent packet is continuously resent as a broadcast. It does not matter whether the last packet was part of a burst or acknowledge transmission. If no new data is available this packet is resent. Figure 2.5 shows, how each broadcast transmission is aligned to the channel with a message period T_{ch}. Since there is no answer from the receiving node, it is not possible to determine if the packet was transmitted correctly.

- Acknowledge data
  Acknowledge data can be used to ensure a node has received a transmitted packet. After receiving an acknowledge packet the node will send a message back to the sender. Acknowledge data should only be used as a unicast, since if multiple nodes send an acknowledge message back, the messages can interfere with each other. Figure 2.5 shows that, just like broadcast data, acknowledge data is always aligned to a time slot, yet the receiver does not wait for the next time slot and instead sends the answer immediately back to the sender.

- Burst data
  Burst data provides a method to quickly transmit larger amounts of data. This is achieved by ignoring the normal channel time slots and sending the packets immediately one after the other. This allows for a transmission rate of up to 20 kbps [3], which is much higher than the other data transmission types. Similar to acknowledge data at the end of the transmission the sender is informed whether the transfer failed or succeeded. Just like acknowledge data, burst transmission should be unicast, since if one node fails to receive a packet the transmission is stopped. The drawback of this method is, that burst data is prioritized over all other transmissions and will interrupt other transmissions over the same RF frequency.

2.2.4. ANT messages

In the ANT protocol each message has the basic format as specified in Figure 2.6.

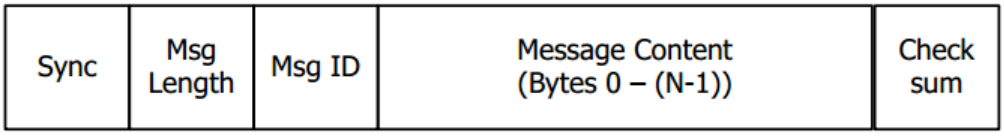

Figure 2.6.: ANT message structure[3]

Each message starts with a special Sync-Byte and ends with a checksum, which is calculated by xoring all previous bytes. The Msg Length byte shows the number of Message Content bytes. The Msg ID byte specifies which kind of data is contained in the message. The ANT protocol also provides an extended message format, which allows to attach further information to each message. The length of the message varies between message types, but the size of the payload for each of the three different data-transmission types is always at 8 bytes.

2.3. ANT library

There is an official ANT library [8] available, however this library is written in C++ for Windows. Since SHAMPU uses a PIC24FJ64GB002 microcontroller[1] it is not possible to use the official library. Instead we use an already existing ANT library for this thesis [9]. This library implements almost the complete ANT API, except for two important features. The library uses busy-waiting to receive packets. Also sending burst transmission does not work correctly. In order to be able to use ANT efficiently with SHAMPU we implemented a non-blocking packet receiver. This is an important

addition to the code, because otherwise the microcontroller is not able to function, while it is waiting for a packet. Unfortunately we were not able to get burst mode working correctly. Burst packets need to have a precise timing, but the black box nature of the ANT protocol makes it a challenge to debug the code.

2.4. ANTAP1MxIB RF

Each SHAMPU device is equipped with an ANTAP1MxIB RF Transceiver Module[2]. The module was chosen because of its small form factor (20mm x 20mm) and its very low power draw. The ANTAP1M can handle up to 4 different ANT channels with a combined message rate of 200 Hz.

Figure 2.7.: SHAMPU base station

In order to test the capabilities of the ANT chip, we used two SHAMPU base stations for all the experiments. This base station (see Figure 2.7) contains the ANTAP1MxIB and a MAX3224CPP microcontroller acting as an UART to serial interface, which allows the ANT to asynchronously communicate with a PC over RS-232. A transmission speed of 19200 baud is used which allows a data rate of 1920 Bps, since the RS-232 protocol adds a start and stop bit to each transmitted byte[10].

Chapter 3.

Evaluation of SHAMPU

In order to assess the capabilities of SHAMPU, we planed and ran experiments designed to evaluate the SHAMPU framework according to the following use-cases:

- **Scheduled data-transmission**
  In order for SHAMPU to work as a debugging and logging platform, the base station needs to periodically receive data from all the nodes in the network. Furthermore the base station has to be able to send commands to nodes in the network.
- **Unscheduled data-transmission**
  There are several cases, where it is not feasible to use a scheduled data-transmission. Because either the data only needs to be transmitted once, or it is not known at what point in time the transmission happens:
  - Reprogramming of a node: SHAMPU is able to reprogram the attached node. For this process, SHAMPU needs to receive a new firmware, which can amount to several hundred kB.
  - SHAMPU RAM-Dumps: SHAMPU has 128kB of RAM, which can be used to save collected data during an experiment. At the end of the experiment the complete memory needs to be transmitted back to the base station.

To evaluate ANT according to these two use-cases, we identified three metrics which indicate how well a use-case can be handled:

- **Data throughput**
  The speed with which ANT can transmit data, directly affects the network performance and how many nodes can be part of the network at the same time. ANT provides three different data types which can be used to transport data: Broadcast data and acknowledge data is typically used for scheduled data-transmission, whereas burst mode is used mostly for unscheduled data-transmission.
- **Message delay**
  A SHAMPU base station not only acts as a data sink for incoming logging information, but is also able to send commands to other SHAMPU devices in the

network. It is thus important to know how long it takes ANT to verify whether a message was correctly received or not.

- **Communication range**
  Since SHAMPU is architecture independent it can be used in different situations. Therefore it is important to know, how far away from the base station the nodes can be placed. As SHAMPU focuses on energy efficiency, it might also be viable to reduce the range for smaller setups to save even more energy.

To cover the mentioned metrics we designed different experiments, which test one or more of the described categories. The following section describes the experiments according to the following template:

Description
A description of the experiment and the category being evaluated.

Use-Case
The use-case which the experiment tries to test.

Network topology and pseudo code
A diagram of the network topology in which the experiment is run and pseudo code which describes the program being run on the master and the slave. Missing values are default values and described in section 3.1.

Testing methodology
A description how the experiment is performed.

Result
The results of the experiment and any additional data collected during the experiment.

3.1. Common experiment parameters

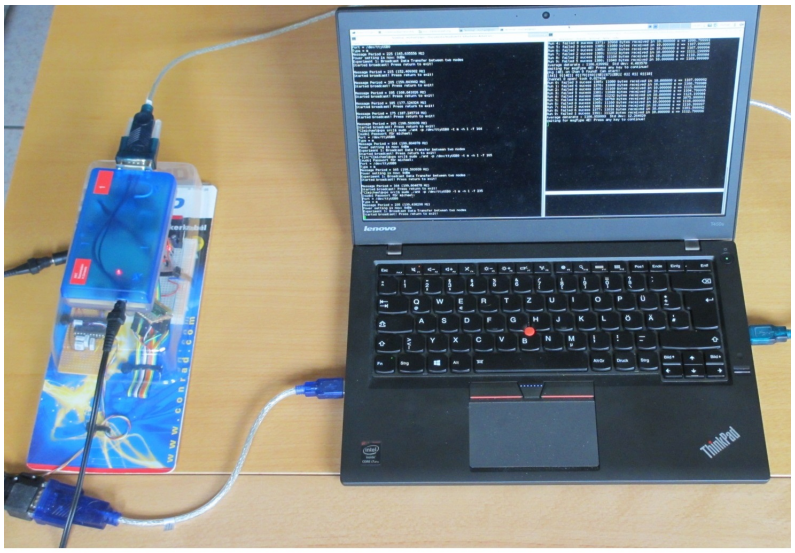

Figure 3.1.: Experiment setup

If not otherwise noted in the description each experiment was run in the Mobility Lab of the Networked Embedded Systems group at the University of Duisburg-Essen (SA 327), with the two base stations in the configuration which can be seen in figure 3.1. The following table describes the parameters which were used in the experiments.

Device number	33	Channel ID (see section 2.2.2)
Device type	1	
Transmission type	1	
ID_CHAN1	0	Configuration for the first channel
FREQ_CHAN1	66 Hz	
ID_CHAN2	1	Configuration for the second channel
FREQ_CHAN2	77 Hz	
STD_FREQ	8192	4 Hz (default frequency)
min_Channel_Period	165	198.6 Hz (closest to 200 Hz)
max_Channel_Period	65535	0.5 Hz (smallest frequency)
STD_POWER	0 dBm	1 mW (default transmit power)

Table 3.1.: ANT default configuration

The channel period p can be used to calculate the frequency of the messages $f_t = \frac{32678s^{-1}}{p}$. For example a message period of 8192 results in a message frequency of 4 Hz, which means that 4 messages are sent every second.

ANT supports different transmit power levels x: 0 dBm, -5 dBm, -10 dBm and -20 dBm. dBm is a way to express broadcast power as a power ratio in decibels. The power of the broadcast p can be calculated as $p = 1mW * 10^{\frac{x}{10}}$.

Furthermore in experiments which measure data throughput, we show the maximum theoretical value for each frequency. This is the maximum value which can be achieved, if there are no transmission errors and no interference with other channels. The value depends on the message frequency f and is calculated as $rate_{max}(f) = 8 * f$. Each message contains a payload of 8 bytes and f messages are sent each second.

The results for each experiment is always the average over all measured values. The displayed error margins are the standard deviation of all measured values. Additionally in some experiments we show the measured error rate, which is also averaged over all measured values.

3.2. Experiment 1: Broadcast Data Transfer between two nodes

Description

Broadcasting is one way of periodically transmitting data between two or more ANT nodes. Since all broadcast packets are synchronized to a fixed time slot, the data throughput can be increased by decreasing the channel period. The experiment itself is split into two parts. In the first part we try to determine the highest possible data throughput. Also we try to determine whether the channel period has an effect on the time it takes for a slave node to find and join an existing channel. The second part is a test of the highest detected data throughput. Here we try to evaluate if there are any variations of the data throughput over a much longer interval.

Use-Case

Scheduled data-transmission

Network topology and pseudo code

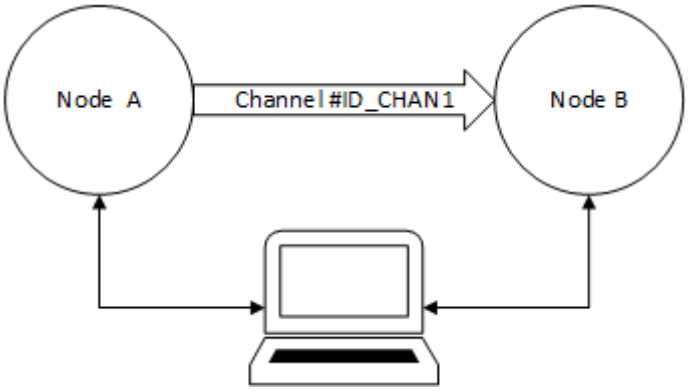

Figure 3.2.: Topology experiment 1

```
channelPeriod = max_Channel_Period
while (channelPeriod >= min_Channel_Period)
  ANT_SetChannelPeriod(ID_CHAN1, channelPeriod)
  ANT_OpenChannel(ID_CHAN1, ANT_Bidirectional_Master)
  ANT_SendBroadcastData(ID_CHAN1, [0x70, 0x69, 0x6E, 0x67])
  wait_for_user_input()
  ANT_CloseChannel(ID_CHAN1)
  if (channelPeriod >= 0x00FF)
    channelPeriod = channelPeriod >> 1
  else
    channelPeriod = channelPeriod - 10
```

Pseudo code 1: Broadcast data single channel (Master)

```
channelPeriod = max_Channel_Period
while (channelPeriod >= min_Channel_Period)
  for (i in 0..10)
    ANT_SetChannelPeriod(ID_CHAN1, channelPeriod)
    ANT_OpenChannel(ID_CHAN1, ANT_Bidirectional_Slave)
    count = 0, fail = 0
    for (100 seconds)
      if (receivedPacket() == ANT_BROADCAST_DATA)
        count++;
      else if (receivedPacket() == ANT_MESSAGE_EVENT_RX_FAIL)
        fail++;
    print (count * 8 / 100) + " Bytes per second"
    print fail + " failed transmissions"
    wait_for_user_input()
    ANT_CloseChannel(ID_CHAN1)
  if (channelPeriod >= 0x00FF)
    channelPeriod = channelPeriod >> 1
  else
    channelPeriod = channelPeriod - 10
```

Pseudo code 2: Broadcast data single channel (Slave)

Testing methodology

Experiment 1 is split into two parts. In the first part node A acts as the master and node B as the slave. For both nodes the channel period is set to the highest value and the channel is opened. Node B records how long it takes to join the channel and how many bytes it receives over a 100s interval. The measurement is repeated 10 times and the average values are saved. Then the channel period is

decreased and the process is repeated.
In the second part of the experiment, the channel period is set to the value which achieved the highest speed. The experiment is then left running for a period of 10 hours and the data throughput is recorded in one continuous run.

Result

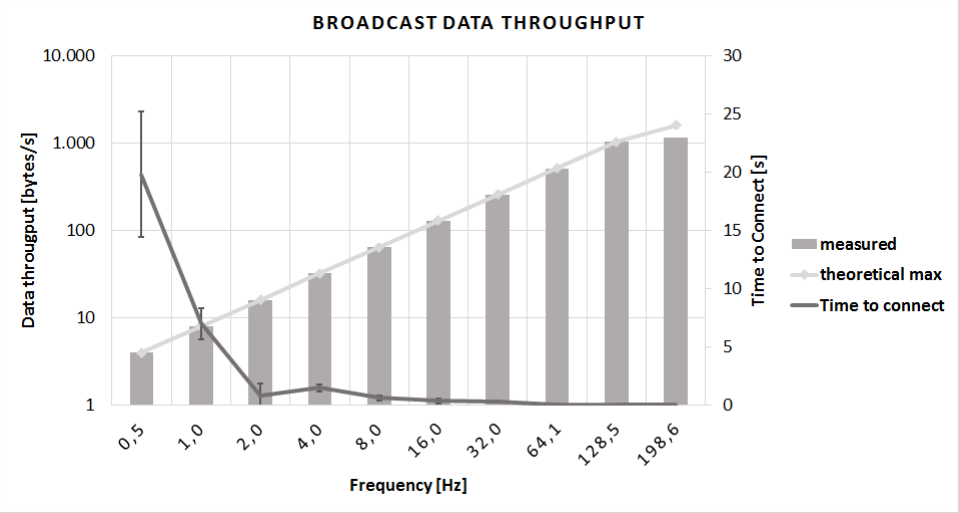

Figure 3.3.: Broadcast data throughput (0.5Hz - 198.6Hz)

Figure 3.3 shows the transmission speeds achieved for the different frequencies. Up to a frequency of 128 Hz, the measured data throughput matches the expected theoretical maximum rate. For the highest measured frequency (198.6 Hz), the data throughput is lower than the maximum rate. The measured errors are insignificant. Therefore the frequencies between 128 Hz and 198.6 Hz were analyzed in more detail (see page 20).

The time it takes for a node to join a channel is very short once the frequency is above 1 Hz. For frequencies above 64 Hz, the time it takes to join a channel becomes negligible, with times around 75 ms (see Figure 3.3). All the measured values are below the specified worst case channel acquisition times of the ANT protocol [11].

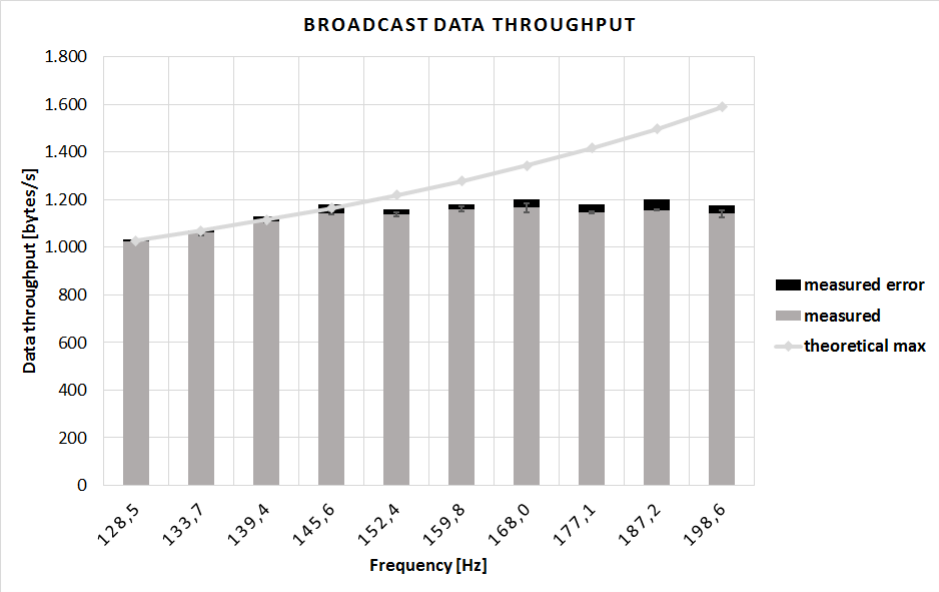

Figure 3.4.: Broadcast data throughput (129Hz - 198.6Hz)

As seen in figure 3.4 the measured data throughputs for frequencies above 140 Hz all fall short of the maximum values. The average data throughput of these frequencies remains consistently at around 1100 Bps. The experiment was repeated multiple times, running it at different times and places, thus an environmental factor can be excluded. That means, the reason for the upper limit has to be found with the test setup itself. See section 3.8 for a discussion about possible reasons for this upper limit.

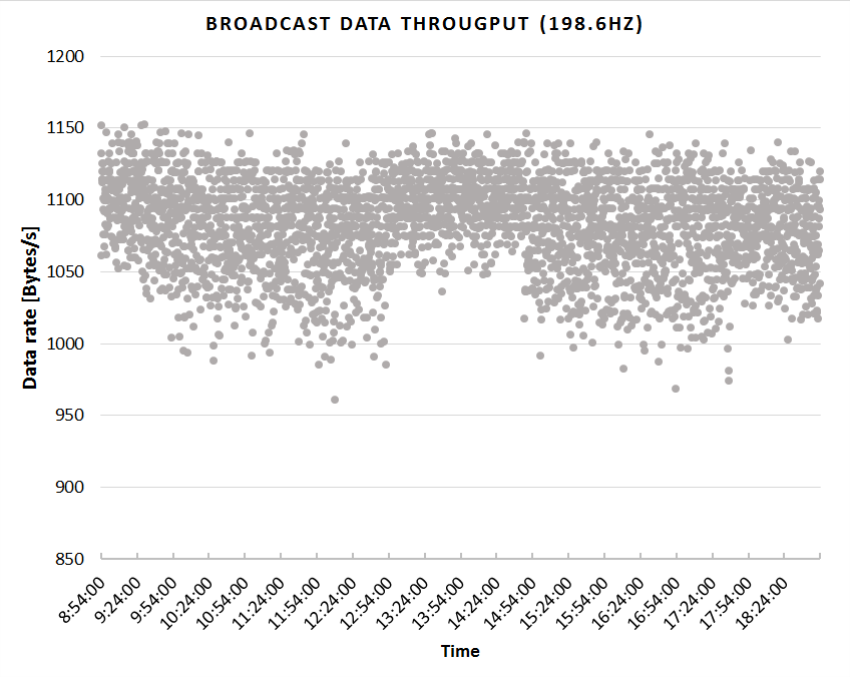

Figure 3.5.: Broadcast data throughput over time (198.6Hz)

Figure 3.5 shows the transmission speed for the highest supported frequency 198.6 Hz over time. The results show that the data throughput stays fairly consistent at around 1100 Bps with a standard deviation of only 30 Bps, or 2.7%. Interesting to note is the time period from 1:00PM to 2:50 PM, where the deviation from the average is much smaller. The exact reason for this phenomenon in unknown, since during this time period the environment of the test setup did not change in any known way.

3.3. Experiment 2: Broadcast Data Transfer with two channels

Description

In experiment 1 we determined the channel period, which allows for the maximum throughput. In this experiment we try to determine, how the maximum throughput is affected by the number of channels in the network. SHAMPU needs two channels to work correctly: One channel which sends data from the base station to the nodes in order to control them, and another channel by which the nodes can send debugging and other information.

Use-Case

Scheduled data-transmission

Network topology and pseudo code

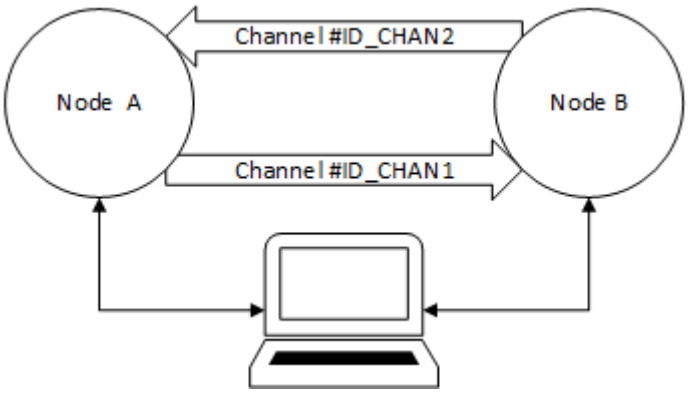

Figure 3.6.: Topology experiment 2

```
channelPeriod = max_Channel_Period
while (channelPeriod >= min_Channel_Period)
  ANT_SetChannelPeriod(ID_CHAN1, channelPeriod)
  openChannel(ID_CHAN1, ANT_Bidirectional_Master)
  ANT_SendBroadcastData(ID_CHAN1, [0x70, 0x69, 0x6E, 0x67])
  ANT_SetChannelPeriod(ID_CHAN2, channelPeriod)
  openChannel(ID_CHAN2, ANT_Bidirectional_Slave)
  count = 0
  for (100 seconds)
    if (receivedPacket() == ANT_BROADCAST_DATA)
      count++
  print (count * 8 / 100) + " Bytes per second"
  wait_for_user_input()
  ANT_CloseChannel(ID_CHAN1)
  ANT_CloseChannel(ID_CHAN2)
  if (channelPeriod >= 0x01FF)
    channelPeriod = channelPeriod >> 1
  else
    channelPeriod = channelPeriod - 10
```

Pseudo code 3: Broadcast data transfer two channels (Master)

```
channelPeriod = max_Channel_Period
while (channelPeriod >= min_Channel_Period)
  ANT_SetChannelPeriod(ID_CHAN1, channelPeriod)
  openChannel(ID_CHAN1, ANT_Bidirectional_Slave)
  ANT_SetChannelPeriod(ID_CHAN2, channelPeriod)
  openChannel(ID_CHAN2, ANT_Bidirectional_Master)
  ANT_SendBroadcastData(ID_CHAN2, [0x70, 0x69, 0x6E, 0x67])
  count = 0
  for (100 seconds)
    if (receivedPacket() == ANT_BROADCAST_DATA)
      count++
  print (count * 8 / 100) + " Bytes per second"
  wait_for_user_input()
  ANT_CloseChannel(ID_CHAN1)
  ANT_CloseChannel(ID_CHAN2)
  if (channelPeriod >= 0x01FF)
    channelPeriod = channelPeriod >> 1
  else
    channelPeriod = channelPeriod - 10
```

Pseudo code 4: Broadcast data transfer two channels (Slave)

Network topology and pseudo code

The two nodes are placed right next to each other.

Testing methodology

In this experiment each node acts as a master for a different channel. Node A is the master for Channel 0 and Node B is the master for Channel 1. The measurements themselves are identical to the ones in experiment 1, except that the data is recorded on both nodes. The channel period is then decreased until the data throughput no longer increases, or the connections breaks completely.

Result

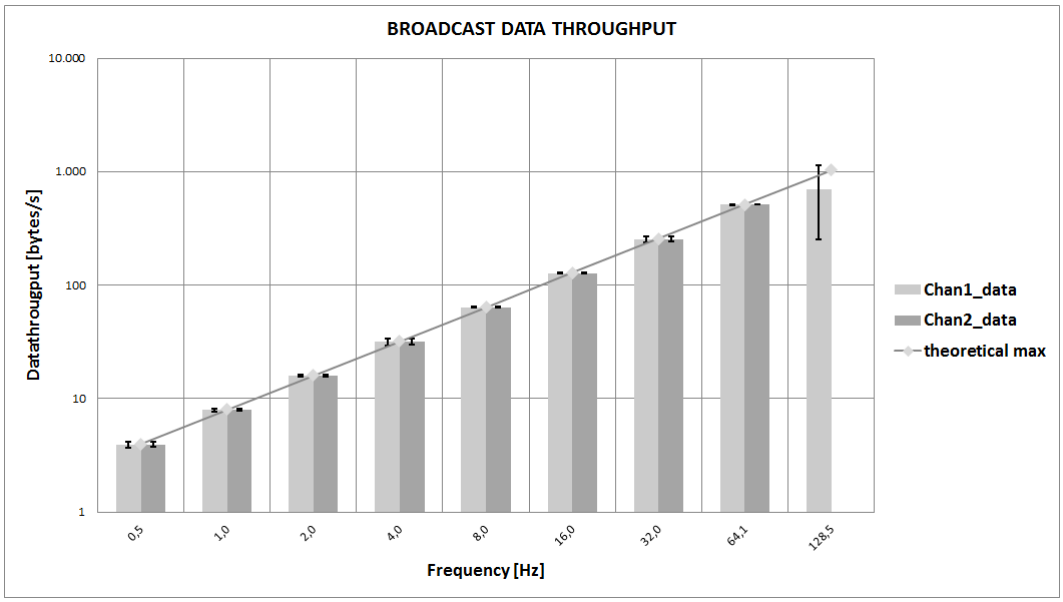

Figure 3.7.: Broadcast data throughput - 2 channels (0.5Hz - 129Hz)

Figures 3.7 shows the results of the measurements, the left bar displaying the throughput for channel 1, the right bar displaying the throughput for channel 2. Just as in experiment 1 the line shows the theoretical maximum of the data throughput for the given frequency. Up to a frequency of 64 Hz the data throughput increases with the frequency for both channels and matches the maximum value very closely. However the data throughput of channel 2 drops to zero at 128 Hz, while the data throughput of channel 1 almost matches the expected value, although the standard deviation indicates performance problems in this channel as well.

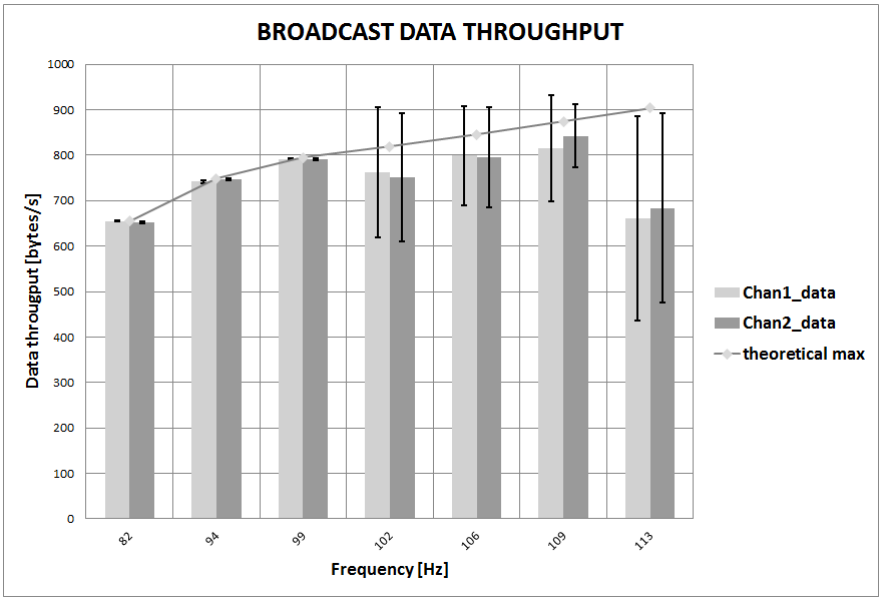

Figure 3.8.: Broadcast data throughput - 2 channels (64Hz - 113Hz)

To further investigate this drop off, figure 3.8 shows the data throughput for chosen frequencies between 82 Hz and 113 Hz. Up to a frequency of 99 Hz the data throughput matches the expected values. For frequencies over 100 Hz however the data throughput drops off, while the measured errors increase. From this result we conclude two things: The 200 Hz capacity of the ANT chip is split between all channels the chip opened. In the test set up each chip can broadcast and receive with 100 Hz each, resulting in a data throughput of around 800 Bps for each channel. Above 100 Hz the increased standard deviation can be interpreted as an indication of interference between the sending and receiving unit of the ANT chip. This interference eventually leads to a breakdown of the data throughput at even higher frequencies. See section 3.8 for a discussion about this upper limit.

3.4. Experiment 3: Acknowledge Data Transfer between two nodes

Description

This experiment is almost identical with experiment 1. The main difference is that we use acknowledge data instead of broadcast. It is also important to note that the master records how many successful packets are transmitted. The main goal of the experiment is to see whether the data throughput is decreased compared to broadcast data, especially for smaller channel periods.

Use-Case

Scheduled data-transmission

Network topology and pseudo code

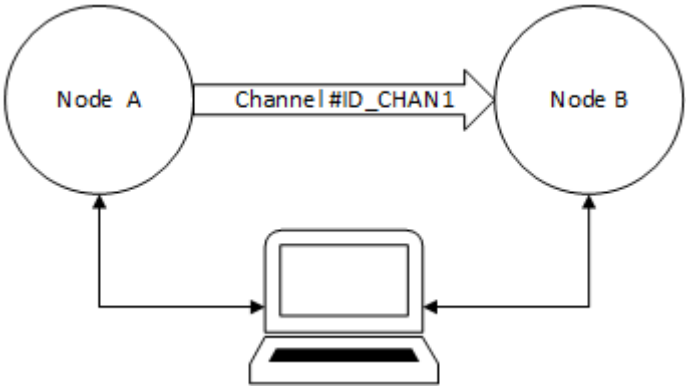

Figure 3.9.: Topology experiment 3

```
channelPeriod = max_Channel_Period
while (channelPeriod >= min_Channel_Period)
  ANT_SetChannelPeriod(ID_CHAN1, channelPeriod)
  ANT_OpenChannel(ID_CHAN1, ANT_Bidirectional_Master)
  count = 0, fail = 0
  for (10 seconds)
    ANT_SendAcknowledgedData(ID_CHAN1, [0x70, 0x69, 0x6E, 0x67])
    if (receivedPacket() == ANT_BROADCAST_DATA)
      count++;
    else if (receivedPacket() == ANT_MESSAGE_EVENT_RX_FAIL)
      fail++;
  print (count * 8 / 10) + " Bytes per second"
  print fail + " failed transmissions
  ANT_CloseChannel(ID_CHAN1)
  if (channelPeriod >= 0x00FF)
    channelPeriod = channelPeriod >> 1
  else
    channelPeriod = channelPeriod - 10
```

Pseudo code 5: Acknowledge data transfer (Master)

```
channelPeriod = max_Channel_Period
while (channelPeriod >= min_Channel_Period)
  ANT_SetChannelPeriod(ID_CHAN1, channelPeriod)
  ANT_OpenChannel(ID_CHAN1, ANT_Bidirectional_Slave)
  wait_for_user_input()
  ANT_CloseChannel(ID_CHAN1)
  if (channelPeriod >= 0x00FF)
    channelPeriod = channelPeriod >> 1
  else
    channelPeriod = channelPeriod - 10
```

Pseudo code 6: Acknowledge data transfer (Slave)

Testing methodology

The testing methodology is the same as in experiment 1, except that the master sends acknowledge messages and waits for the slave to confirm the successful transmission before sending the next packet.

Result

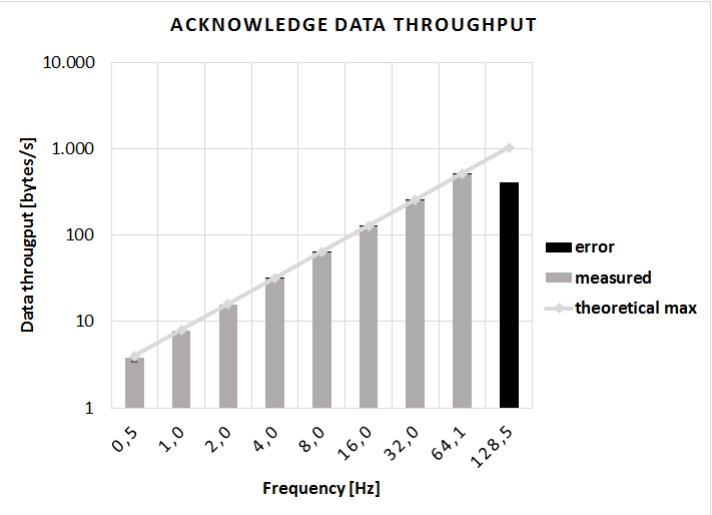

Figure 3.10.: Acknowledge data throughput (0.5Hz - 129Hz)

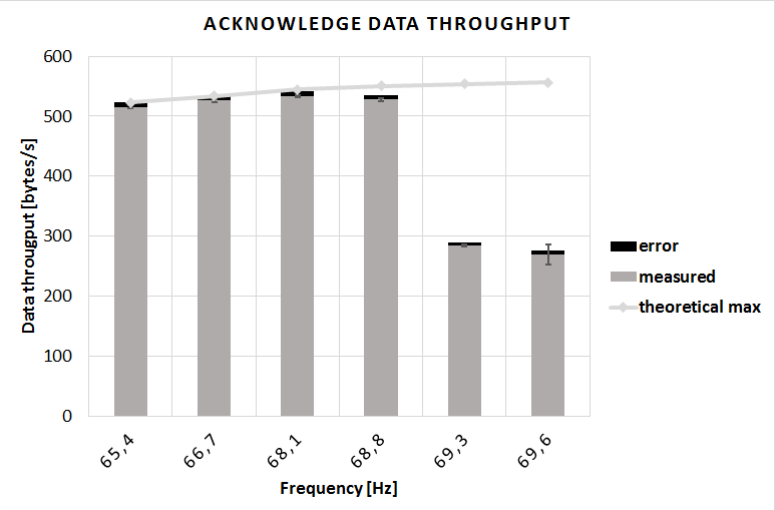

Figure 3.11.: Acknowledge data throughput (82Hz - 113Hz)

Figure 3.10 shows the measured transmission speeds for the different tested frequencies. For the lower frequencies the values align very well with the maximum theoretical data throughput. However, the tranmisson breaks down at 129 Hz.

Figure 3.11 shows the data throughput for some values between 65 Hz and 70 Hz, demonstrating that a drop off can be observed at around 69 Hz.

Since there is no increase of the data throughput above 68 Hz, but a huge drop off to about 50% of the maximum theoretical throughput at higher frequencies and eventually a complete break down, the maximum data throughput for acknowledge data-transmissions is approximately 550 Bps.

3.5. Experiment 4: Acknowledge Data Transfer delay

Description

In this experiment we try to determine the time it takes for a node to receive and acknowledge a transmitted packet. The value is important in order to be able to determine the reaction time of SHAMPU to commands sent by the base station, for example to set up a separate channel for a burst transmission.

Use-Case

Scheduled data-transmission

Network topology and pseudo code

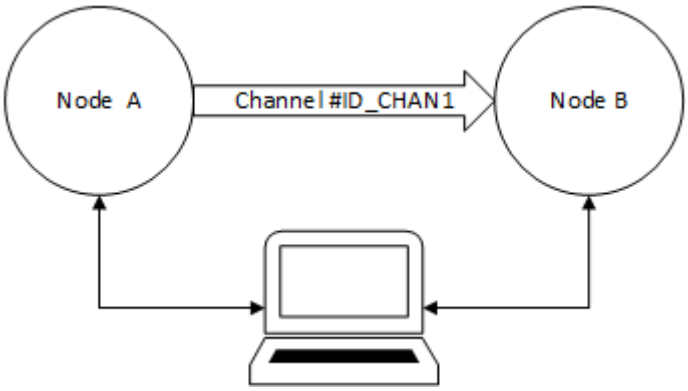

Figure 3.12.: Topology experiment 4

```
channelPeriod = max_Channel_Period
while (channelPeriod >= min_Channel_Period) {
  ANT_SetChannelPeriod(ID_CHAN1, channelPeriod);
  ANT_OpenChannel(ID_CHAN1, ANT_Bidirectional_Master);
  duration = 0.0
  for (i in 0..10) {
    ANT_SendAcknowledgedData(ID_CHAN1, [0x70, 0x69, 0x6E, 0x67])
    start = getTime()
    wait_for_ack()
    print (getTime() - start) + " s"
  ANT_CloseChannel(ID_CHAN1)
  if (channelPeriod >= 0x00FF)
    channelPeriod = channelPeriod >> 1
  else
    channelPeriod = channelPeriod - 10
```

Pseudo code 7: Acknowledge data delay (Master)

```
channelPeriod = max_Channel_Period
while (channelPeriod >= min_Channel_Period)
  ANT_SetChannelPeriod(ID_CHAN1, channelPeriod);
  ANT_OpenChannel(ID_CHAN1, ANT_Bidirectional_Slave);
  wait_for_user_input();
  ANT_CloseChannel(ID_CHAN1);
  if (channelPeriod >= 0x00FF)
    channelPeriod = channelPeriod >> 1
  else
    channelPeriod = channelPeriod - 10
```

Pseudo code 8: Acknowledge data delay (Slave)

Testing methodology

Node A acts as the master and node B as the slave. For both nodes the channel period is set to the highest value and the channel is opened. Node A then sends a total of 10 acknowledge messages and measures how long it takes until it receives the acknowledge signal. The channel period is then decreased and the experiment repeated.

Result

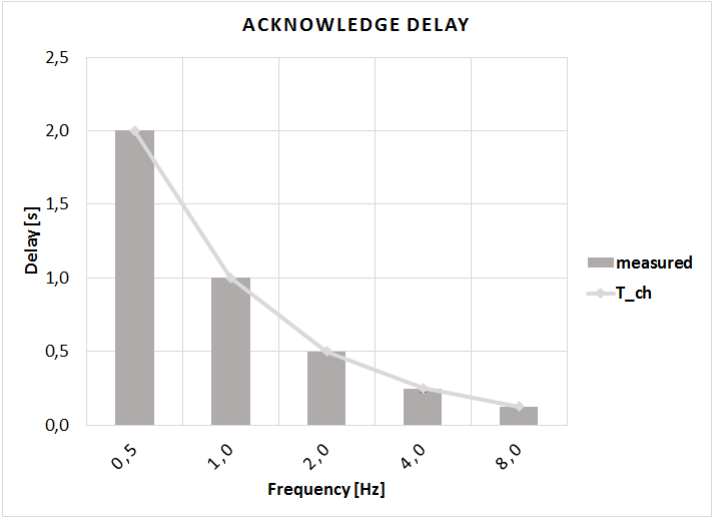

Figure 3.13.: Acknowledge delay (0.5Hz - 8Hz)

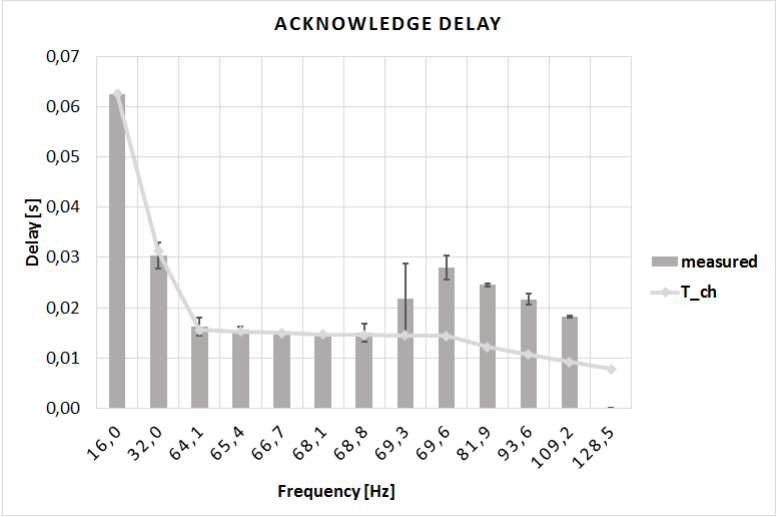

Figure 3.14.: Acknowledge delay (16Hz - 129Hz)

Figure 3.13 and 3.14 show the delays for the tested frequencies. The lines show the time T_{ch} between two timeslots for various frequencies f, where $T_{ch} = \frac{1}{f}$. These values are in line with the measured delays until a frequency of 68.8Hz. 69.3 Hz

represents a transition area, where the individual data points show delays with a duration of either one or two times T_{ch}. This explains the increased standard deviation. Above that frequency the delay corresponds with $2T_{ch}$ up to a frequency of at least 109 Hz. At 129 Hz no delay could be measured, as no acknowledge messages were correctly received.

We conclude, that in the given setup ANT requires at least 15 ms (equivalent to a frequency of 68.1 Hz) to confirm that a message has been successfully transmitted. If the time between the timeslots is shorter, it takes an additional timeslot to confirm the message. Due to lack of information about ANT, we cannot provide an explanation for this behaviour. Similarly we cannot offer a reason, why the acknowledge data-transmission breaks down completely at 128 Hz. The fact of the maximum delay of 15 ms explains why the maximum data throughput of acknowledge data is only 550 Bps.

3.6. Experiment 5: Burst Data Transfer between two nodes

Description

Burst data transmissions make it possible to drastically increase the throughput rate. This allows a SHAMPU base station to quickly transmit a new firmware to a node or a node to dump its RAM back to the base station. According to the specification rates of up to 20 kbps can be achieved. To fully utilize this speed a baud rate of 50000 is needed[12]. In the current setup the base station uses 19200 baud, since it is the only value that both the ANT chip and the RS-232 interface support. Because of this we expect the maximum speed to be less than 20 kbps. Furthermore we try to determine whether the size of the burst transfer has an impact on the speed, since longer bursts will disrupt communications on other channels.

Use-Case

Unscheduled data-transmission

Network topology and pseudo code

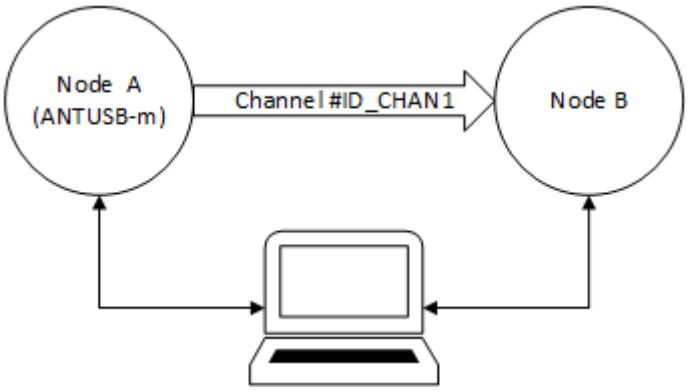

Figure 3.15.: Topology experiment 5

```
size = START_SIZE
ANT_OpenChannel(ID_CHAN1, ANT_Bidirectional_Slave);
while (size >= END_SIZE)
  for (i in 0..10)
    // The first packet of a burst has the 3 MSB of the channelID field set to 0
    wait_until(received_Packet == ANT_BURST_DATA &&
               (received_message_channelID[0] & 0xE0) == 0x00)
    start = getTime()
    // The last packet of a burst has the MSB of the channelID field set to 1
    wait_until(received_Packet == ANT_BURST_DATA &&
               (received_message_channelID[0] & 0x80) > 0)
  print (size - 1) / (getTime() - start) " Bytes per second"
  size = 2 * size
```

Pseudo code 9: Burst data transfer (Slave)

Testing methodology

Since the burst transmission mode of our ANT-library is not working correctly (see section 4.3) we use an ANTUSB-m Stick, which acts as a master. Node B is a normal base station and receives the burst transfers. On the master side, the program ANTWareII [13] is used to create the channel and then send bursts with different sizes. Each size is sent 10 times and the values are recorded and averaged. The size is then doubled and the experiment repeated. The first burst packet cannot be counted directly since the start of the burst can only be determined after the ANT chip has already received the first packet.

Result

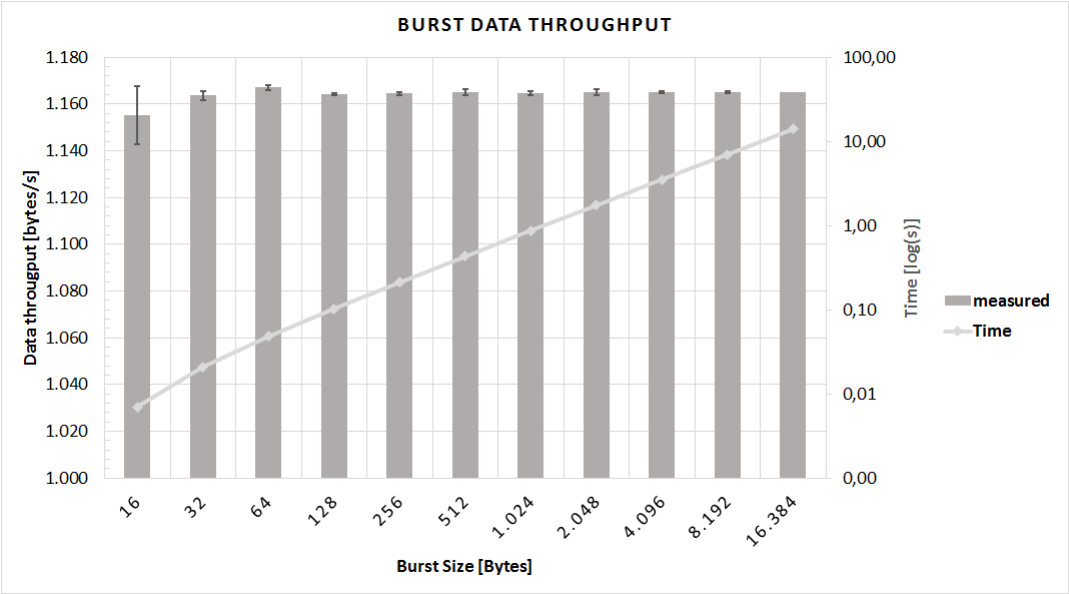

Figure 3.16.: Burst data throughput

Figure 3.16 shows the measured data throughput for the different packet sizes and also the length of the transfer. The values all hover around the same value of 1165 Bps. This is approximately the same value as the one which can be achieved with the other two types of data. Again see section 3.8 for a discussion about possible reasons for this upper limit.

3.7. Experiment 6: Maximum communication Range

Description

In this experiment we try to determine the correlation between the maximum range and the power setting of the ANT radio. One of SHAMPU's advantages is the low power consumption. It might be possible to further reduce the power consumption by decreasing the power level of the ANT radio, especially in smaller environments where there is no need for a long range. According to the datasheet the maximum range for communication is 30m, however the ANT documentation does not provide different ranges for different power settings.

Use-Case

Unscheduled data-transmission and scheduled data-transmission

Network topology and pseudo code

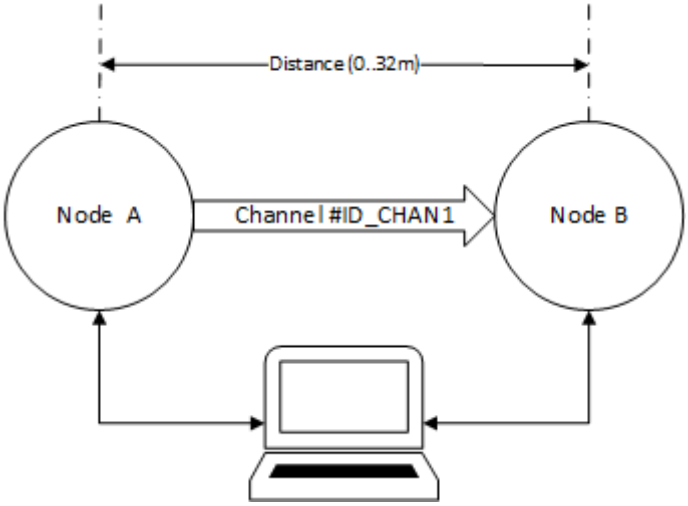

Figure 3.17.: Topology experiment 6

```
for (pSetting in Available_PowerSettings)
  ANT_SetTransmitPower(pSetting)
  openChannel(ID_CHAN1, ANT_Bidirectional_Master)
  ANT_SendBroadcastData(ID_CHAN1, [0x70, 0x69, 0x6E, 0x67])
  wait_for_user_input();
  closeChannel(ID_CHAN1);
```

Pseudo code 10: Maximum communication range (Master)

```
distance = 0.0
stopInc = false
loop
  openChannel(ID_CHAN1, ANT_Bidirectional_Slave)
  wait_until(received_Packet == ANT_BROADCAST_DATA ||
             received_Packet == ANT_MESSAGE_EVENT_RX_SEARCH_TIMEOUT)
  if (stopInc)
    if (wasTimeout)
      distance -= .4
    else
      print "Connection found : " + distance
  else
    if (wasTimeout)
      stopInc = true
      distance -= .4
      print "Connections lost : " + distance
    else distance += .4
  closeChannel(ID_CHAN1);
```

Pseudo code 11: Maximum communication range (Slave)

Testing methodology

At the beginning of the experiment Node A and B are placed right next to each other. Node A acts as a master and keeps broadcasting the same message. Node B is the slave and tries to connect to the channel. If the connection is successful, the distance between the two nodes is increased by 0.4 meters. This process is repeated until Node B can no longer connect to the channel and the connection times out. This happens after 30s of searching. At this point the distance is no longer increased, but rather decreased until Node B is able to successfully connect to the channel. Both of these values are recorded and averaged. The whole experiment is then repeated for each available power setting.

Result

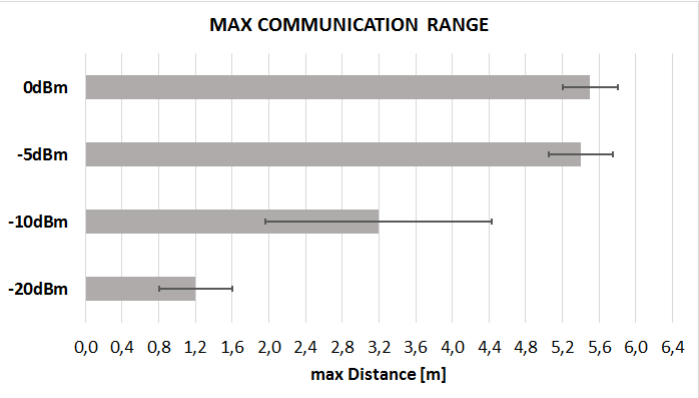

Figure 3.18.: Maximum communication range

Figure 3.18 shows the transmission range for each power setting. As expected the maximum distance goes up with the higher power settings. But the result at the highest power setting is disappointing, since we did not get close to the claimed maximum range of 30m. The ANT documentation states that the maximum range of 30m can only be achieved in "optimal conditions" [3]. It is possible for several different unfavorable factors to interfere with the ANT signal, such as multiple 802.11 networks in the vicinity, or even the plastic case of the base station.

3.8. Maximum Data Throughput

In our experiments we were able to confirm two important limits for the maximum data throughput which can be achieved with our current setup. The ANT AP1MxIB supports a maximum message frequency of 200 Hz, which means that we would expect to see a maximum transmission rate of 1600 Bps. However, this limit can only be achieved by splitting the available bandwidth between receiving and sending. If we try to utilize the full 200 Hz only for either receiving or sending, we are limited to 1100 Bps, around 500 Bps or 30% lower than the theoretical maximum. A similar limit of 1165 Bps holds for burst transmissions, where we loose around 1335 Bps or 53% of the maximum theoretical data throughput.

To balance environmental influences, the experiments were repeated in different locations and times of day and night. Since the throughput did not change noticeably, it can be assumed that there are no easily eliminated environmental factors affecting the maximum rate. We conclude that the root cause for the lower than expected throughput must be attributed to the hardware or software.

If we exclude environmental factors, the limitations could be based on certain parts of the test set up:

- **RS-232 Connection**
  The base station is connected to a PC over a serial connection. For the speed we chose 19200 baud, since it is the highest value supported by the ANT chip and the serial-Interface. With 19200 baud the maximum data throughput of the RS-232 Connection is 1920 Bps, since the interface adds a start and stop bit to each byte. If the serial connection would be the bottle neck for burst transmission, we would expect to get the full 1920 Bps of data throughput. Since this is not the case we can conclude that the RS-232 connection is not the limiting factor. This is supported by the similarity of the limits for broadcast and burst transmissions.
- **ANT API**
  The software we use is not officially supported by ANT. It is thus possible that there are some bugs which have an impact on the performance. Except for the missing burst mode (see 4.3), there were no unusual measured values during the experiments. If there is a bug, it might be hard to find without rewriting all the experiments and running them with the official ANT library [8]
- **ANT AP1MxIB**
  Due to the black box nature of ANT, it is very hard to exactly determine what might cause a performance deficit. The inner workings of the chip are not documented in any way, and the error messages of the protocol are not very specific.

We suspect that the ANT chip receiver and sender are somehow limiting the maximum data throughput. The chip itself is from 2007 and the manufacturer no longer recommends the use of the chip[14]. There are alternatives available, like the newer ANTAP281M5IB chip [15].

Chapter 4.

Discussion

4.1. Summary

In our experiments ANT showed useful properties, although it fell short of the theoretical performance in some experiments.

- **Data throughput**
  The ANT chip is able to utilize the advertised 200 Hz message rate, which translates into a data throughput of 1600 Bps. However this limit can only be reached by combining sending and receiving. The maximum data throughput drops down to around 1100 Bps or 128 Hz, if the chip is only sending or receiving. Surprisingly even the burst mode is limited by a similar 1165 Bps limit.
- **Connections/Acknowledge delay**
  We determined that for frequencies above 8 Hz the delay to establish a connection is well below 1 second. This means that for any channel with around 100 Bps or more the delay is insignificant and can be ignored.
  Additionally the delay for an ANT node to confirm whether a message was correctly received by a second node was analyzed. The minimum delay was determined to be 15ms, as is expected for a frequency of 68 Hz. At higher frequencies the ANT-chip seems to need two timeslots to acknowledge a transfer. This explains that the highest throughput of acknowledge data is 550 Bps, almost exactly half the speed of one-directional broadcast transfer.
- **Communication Range**
  The maximum communication range of 6m is probably the most disturbing result of this thesis. The short range implies, that the physical location of the base station is critical for a successful deployment of SHAMPU. It also means that a larger network requires more than one such base station to be able to reach all nodes.

We could show that SHAMPU can be used together with ANT. There are limitations, but most of them seem to be of little relevance in real world situations. Even the range limitation can be overcome by appropriate design decisions.

4.2. SHAMPU network setup

In order to correctly function each SHAMPU node needs to be connected to two channels. Each node can then receive and simultaneously send 800 Bps or can either send or receive with 1100 Bps. This limitations affects the SHAMPU base stations the most, because they have to act as a data sink for multiple SHAMPU devices.

With these limitations of data throughput, the network setup has to be chosen very carefully. One option is to use additional SHAMPU nodes as relays, which serve a dual purpose. They allow to extend the range of the network, while increasing the data throughput of the whole network.
Another possibility to counteract the limitation is to leverage the numerous advantages of the SHAMPU framework, such as its low weight, form factor and power draw. A mobile base station, e.g. TrainSense [16], which can easily be moved around to the location where the SHAMPU nodes are deployed and act as data mule, makes it possible to address all above mentioned problems. The range no longer represents a problem, since the base station is simply moved towards the node until a connection can be achieved. At the same time, the limited range provides a solution for the limited data throughput. Since the number of nodes, which are able to connect to the base station can be controlled, it is possible to ensure that only a small number of nodes transmit data to the base station at the same time.

4.3. Future Work

Due to lack of time and the limited hardware availability we were not able to fully explore and evaluate the possibilities of ANT in the SHAMPU network. Especially the following four areas are worth to be investigated in the future:

- **Power consumption**
  Each experiment tries to find the maximum of either data throughput or the communication range. We did not measure how the power consumption changes, if for example the message period is reduced. Since SHAMPU tries to be very low-powered, this is an important measurement for the decision whether SHAMPU can be used for a specific application. The data sheet of the ANT AP1MxIB module provides interesting data [2]: The maximum current draw seems to be around 5 mA for a continuous burst transmission and around 40 μA for a normal broadcast operation.
- **Burst mode**
  We used a custom library, which provides an API to interface with the ANT-Chip. For this thesis an attempt was made to add the missing burst transfer mode. However, due to time constraints we were unable to get the mode working correctly.
- **Shared channels**
  Due to missing hardware, we were only able to test the communication between two nodes. Shared channels could not be tested, yet we expect the available data throughput to be in line with the results from our experiments. ANT uses up to 2 bytes of the 8 byte payload to specify the address of the receiver. Therefore we expect to lose approximately 12.5% to 25% of throughput if shared channels are used. It would be important to confirm this to fully assess the usefulness of the ANT chip.
- **New ANT-chip**
  As mentioned before the chip currently in use is old and no longer recommended for use. The successor of the current chip, the ANTAP281M4IB, has roughly the same specifications as the current chip. Therefore the experiments should be rerun with the newer model in order to determine whether this allows to exploit the full data throughput potential of the ANT-chip in all use cases.

Appendix A.

Sourcecode

```
/* main.c */
#include <unistd.h>
#include <stdio.h>
#include <getopt.h>
#include "experiment.h"

char* ProgName; /* error.c */
extern uint16_t period;

int main(int argc, char** argv) {
  int option = -1;
  char deviceType = 0;
  int experimentNum = -1;
  char* port = "";

  ProgName = fileName(argv[0]); /* error.c */
  if (argc == 1) {
    error("Usage: %s -p /dev/ttyUSB[Number] \
          -t [m(aster) or s(lave)] \
          -n experiment [-f start_period]\n", ProgName);
  }

  while ((option = getopt(argc, argv, "p:t:n:f:")) != -1) {
    switch (option) {
      case 'p' :
        port = optarg;
        break;
      case 't' :
        deviceType = optarg[0];
        break;
      case 'n' :
```

```
        experimentNum = atoi(optarg);
        break;
      case 'f' :
          period = atoi(optarg);
          break;
      default:
        exit(EXIT_FAILURE);
    }
  }

  if (deviceType != 's' && deviceType != 'm') {
    error("Wrong deviceType! Must be 'm' or 's' is: %c\n",
          deviceType);
  }

  if (strncmp(port, "/dev/ttyUSB", 11)) {
    error("Port information wrong!");
  }

  initANT(port);
  setTransmitPower(ANT_TRANSMIT_POWER_0DBM);

  switch (experimentNum) {
    case 1:
      printf("Experiment 1: \
              Broadcast Data Transfer between two nodes\n");
      doExperiment1(deviceType);
    break;

    case 2:
      printf("Experiment 2: \
              Broadcast Data Transfer with two channels\n");
      doExperiment2(deviceType);
    break;

    case 3:
      printf("Experiment 3: \
              Ackowledge Data Transfer between two nodes\n");
      doExperiment3(deviceType);
    break;

    case 4:
```

```
      printf("Experiment 4: \
              Ackowledge Data Delay\n");
      doExperiment4(deviceType);
    break;

    case 5:
      printf("Experiment 5: \
              Burst Data Transfer between two nodes\n");
      doExperiment5(deviceType);
    break;

    case 6:
      printf("Experiment 6: \
              Communication Distance\n");
      doExperiment6(deviceType);
    break;

    default:
      error("%d not a valid! \n", experimentNum);

    break;
  }
  flushBuffer();
  return EXIT_SUCCESS;
}
```

```
/* experiment.h */
#ifndef EXPERIMENT_H_
#define EXPERIMENT_H_

#include <stdbool.h>
#include "ant.h"

#define   DEVICE_NUMBER       33
#define   DEVICE_TYPE          1
#define   TRANSMISSION_TYPE    1

#define   ID_CHAN1             0
#define   FREQ_CHAN1          66
#define   ID_CHAN2             1
#define   FREQ_CHAN2          77

#define   START_FREQ      0xFFFF /*   0.5 Hz */
#define   STD_FREQ        0x2000 /*   4.0 Hz */
#define   END_FREQ        0x00A5 /* 198.6 Hz */
#define   STOP_SINGLE     0x00FF /* 255.0 Hz */
#define   STOP_DOUBLE     0x01FF /*  64.2 Hz */

#define   RUN_COUNT           10
#define   SLICE_COUNT         10
#define   TRANSFER_TIME_S     10 /* = 10 * 10 = 100 s */

typedef enum {
  DS_INCRESING = 0,
  DS_DECREASING,
  DS_DONE
} distanceState_t;

typedef enum {
  DM_SENDING_PACKET = 0,
  DM_WAITING_FOR_ACK
} DelayMeasureState_t;

typedef struct {
  uint16_t sucess;
  uint16_t fail;
  double duration;
} expriment_t;
```

```
typedef expriment_t (*exprHandler)(uint8_t);

void initANT(char* ttyUSBDevice);
void resetANT();
void openChannel(uint8_t channel, uint8_t freq, uint16_t period,
                 bool isMaster);
void closeANT(uint8_t channel);

void setTransmitPower(uint8_t pSetting);
uint16_t halfPeriod(uint16_t period);
void printBuffer(uint8_t *buffer);
int kbhit(void);

void doExperiment(uint8_t msgType, uint8_t channel, exprHandler fn);
expriment_t receiveMessageAndCount(uint8_t msgType);
expriment_t receiveMessageAck(uint8_t msgType);
void doExperiment1(char deviceType);
void doExperiment2(char deviceType);
void doExperiment3(char deviceType);
void doExperiment4(char deviceType);
void doExperiment5(char deviceType);
void doExperiment6(char deviceType);

#endif /* EXPERIMENT_H_ */
```

```
/* experiment.c */

#include <stdio.h>
#include <stdlib.h>
#include <sys/select.h>
#include <termios.h>
#include <time.h>
#include <math.h>
#include "experiment.h"

uint8_t buffer[BC_ANT_BUFFSIZE];
double results[SLICE_COUNT];
uint16_t period = START_FREQ;
uint32_t s = 0; /* holds the result for received messages */

uint8_t data[4] = {0x70, 0x69, 0x6E, 0x67};
struct timespec tstart = {0,0}, tend = {0,0};
double total_t = 0.0;

void initANT(char* ttyUSBDevice) {
  if(antUARTinit(ttyUSBDevice))
    error("Can't initialize ANT port");
}

void resetANT() {
  ANT_ResetSystem();
  flushBuffer();
  ANT_delayMs(500);
}

void openChannel(uint8_t channel,
                 uint8_t freq,
                 uint16_t period,
                 bool isMaster) {
  ANT_delayMs(100);
  ANT_AssignChannel(channel,
                    isMaster ? ANT_Bidirectional_Master :
                               ANT_Bidirectional_Slave,
                    0);
  ANT_delayMs(100);
  ANT_SetChannelId(channel,
           DEVICE_NUMBER,
```

```
            isMaster? DEVICE_TYPE : 0,
            isMaster? TRANSMISSION_TYPE : 0);
  ANT_delayMs(100);
  ANT_SetChannelRFFreq(channel, freq);
  ANT_delayMs(100);
  if (period != STD_FREQ)
    ANT_SetChannelPeriod(channel, period);
  ANT_delayMs(100);
  ANT_OpenChannel(channel);
}

void closeANT(uint8_t channel) {
  ANT_CloseChannel(channel);
}

void setTransmitPower(uint8_t pSetting) {
  switch (pSetting) {
    case ANT_TRANSMIT_POWER_MINUS_20DBM:
      printf("Power setting is now: -20dBm\n");
      break;
    case ANT_TRANSMIT_POWER_MINUS_10DBM:
      printf("Power setting is now: -10dBm\n");
      break;
    case ANT_TRANSMIT_POWER_MINUS_5DBM:
      printf("Power setting is now: -5dBm\n");
      break;
    case ANT_TRANSMIT_POWER_0DBM:
      printf("Power setting is now: 0dBm\n");
      break;
    default:
    error("Unkown power setting : %02X!\n", pSetting);
  }
  ANT_SetTransmitPower(pSetting);
}

uint16_t decreasePeriod(uint16_t period, uint16_t stopShiftingAt) {
  uint16_t result = END_FREQ;
  if (period >= stopShiftingAt) {
    result =  period >> 1;
  } else if ((period - 10) > END_FREQ) {
    result = period - 10;
  } else if ((period -1) > END_FREQ) {
```

```
    result--;
  }
  printf("Message Period = %d (%f Hz)\n", result, 32768.0 / result);
  return result;
}

void printBuffer(uint8_t *buffer) {
  uint8_t i;
  for (i = 0;i < buffer[1] + 4; ++i) {
    printf("[%2X]", buffer[i]);
  }
  printf("\n");
}

void printAndWait(uint8_t msgType, uint8_t channel) {
  printf("Waiting for msgType %2X! Press any key to continue!\n",
         msgType);
  while (!kbhit()) {
    s = ANT_RecvPacket_Blockfree(buffer, BC_ANT_BUFFSIZE);
    if (s == RS_PACKET_COMPLETE) {
      switch(msgType) {
        case ANT_BROADCAST_DATA:
          if (buffer[2] == msgType && buffer[3] == channel) {
            printf("pkg (%2X) on Chan %d found! Can start!\n",
                   buffer[2],
                   buffer[3]);
            printBuffer(buffer);
            clock_gettime(CLOCK_MONOTONIC, &tend);
            total_t = (tend.tv_sec - tstart.tv_sec);
            total_t += (tend.tv_nsec - tstart.tv_nsec) / 1000000000.0;
            printf("Channel %d open! took %f\n", buffer[3], total_t);
            return;
          }
        break;
        case ANT_ACKNOWLEDGED_DATA:
          if (buffer[5] == ANT_MESSAGE_EVENT_TRANSFER_TX_COMPLETED &&
              buffer[3] == channel) {
            printf("pkg (%2X) on Chan %d found! Can start!\n",
                   buffer[2],
                   buffer[3]);
            printBuffer(buffer);
            return;
```

```
          }
        break;
        case 0xFF:
        break;
        default:
          printBuffer(buffer);
        break;
      }
    }
  }
}

int kbhit(void) {
  struct termios oldt, newt;
  int ch;
  int oldf;

  tcgetattr(STDIN_FILENO, &oldt);
  newt = oldt;
  newt.c_lflag &= ~(ICANON | ECHO);
  tcsetattr(STDIN_FILENO, TCSANOW, &newt);
  oldf = fcntl(STDIN_FILENO, F_GETFL, 0);
  fcntl(STDIN_FILENO, F_SETFL, oldf | O_NONBLOCK);

  ch = getchar();

  tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
  fcntl(STDIN_FILENO, F_SETFL, oldf);

  if(ch != EOF) {
    ungetc(ch, stdin);
    int c;
    while ((c = getchar()) != '\n' && c != EOF);
    return 1;
  }
  return 0;
}

void doExperiment(uint8_t msgType,
                  uint8_t channel,
                  exprHandler fn) {
  double speed_avr = 0.0;
```

```
  uint16_t i;
  flushBuffer();
  for (i = 0; i < SLICE_COUNT; ++i) {
    expriment_t result = fn(msgType);
    results[i] = result.sucess * 8 / result.duration;
    speed_avr += results[i];
    printf("Run %d: failed %d sucess %d: %d bytes in %f s => %f\n",
           i, result.fail, result.sucess,
           result.sucess * 8, result.duration, results[i]);
  }
  speed_avr /= SLICE_COUNT;
  double stdDev = 0.0;
  for (i = 0; i < SLICE_COUNT; ++i) {
    double cur = results[i] - speed_avr;
    stdDev += cur * cur;
  }
  stdDev /= SLICE_COUNT;
  printf("Average datarate : %f \tStd dev: %f\n",
         speed_avr,
         sqrt(stdDev));
}
expriment_t receiveMessageAndCount(uint8_t msgType) {
  uint16_t count = 0, fail = 0;
  clock_gettime(CLOCK_MONOTONIC, &tstart);
  total_t = 0.0;
  while (total_t < TRANSFER_TIME_S) {
    s = ANT_RecvPacket_Blockfree(buffer, BC_ANT_BUFFSIZE);
    if (s == RS_PACKET_COMPLETE) {
      if (buffer[2] == msgType) {
        count++;
      } else if (buffer[5] == ANT_MESSAGE_EVENT_RX_FAIL) {
        fail++;
      }
    }
    clock_gettime(CLOCK_MONOTONIC, &tend);
    total_t = (tend.tv_sec - tstart.tv_sec);
    total_t += (tend.tv_nsec - tstart.tv_nsec) / 1000000000.0;
  }
  expriment_t result =  {count, fail, total_t};
  return result;
}
expriment_t receiveMessageAck(uint8_t msgType) {
```

```
  uint16_t count = 0, fail = 0;
  DelayMeasureState_t state = DM_SENDING_PACKET;
  ANT_SendAcknowledgedData(ID_CHAN1, data);
  clock_gettime(CLOCK_MONOTONIC, &tstart);
  total_t = 0.0;
  while (total_t < TRANSFER_TIME_S) {
    if (state == DM_SENDING_PACKET) {
      ANT_SendAcknowledgedData(ID_CHAN1, data);
      state = DM_WAITING_FOR_ACK;
    }
    s = ANT_RecvPacket_Blockfree(buffer, BC_ANT_BUFFSIZE);
    if (s == RS_PACKET_COMPLETE) {
      if (buffer[5] == ANT_MESSAGE_EVENT_TRANSFER_TX_COMPLETED) {
        count++;
        state = DM_SENDING_PACKET;
      } else if (buffer[5] == ANT_MESSAGE_EVENT_TRANSFER_TX_FAILED) {
        fail++;
        state = DM_SENDING_PACKET;
      }
    }
    clock_gettime(CLOCK_MONOTONIC, &tend);
    total_t = (tend.tv_sec - tstart.tv_sec);
    total_t += (tend.tv_nsec - tstart.tv_nsec) / 1000000000.0;
  }
  expriment_t result =  {count, fail, total_t};
  return result;
}
expriment_t measureMessageAck() {
  uint16_t count = 1, fail = 0;
  DelayMeasureState_t state = DM_SENDING_PACKET;
  double total_t = 0.0, res = 0.0;
    while(true) {
    if (state == DM_SENDING_PACKET) {
      ANT_SendAcknowledgedData(ID_CHAN1, data);
      clock_gettime(CLOCK_MONOTONIC, &tstart);
      state = DM_WAITING_FOR_ACK;
    }
    s = ANT_RecvPacket_Blockfree(buffer, BC_ANT_BUFFSIZE);
    if (s == RS_PACKET_COMPLETE) {
      if (buffer[5] ==ANT_MESSAGE_EVENT_TRANSFER_TX_COMPLETED) {
        clock_gettime(CLOCK_MONOTONIC, &tend);
        total_t = (tend.tv_sec - tstart.tv_sec);
```

```
        total_t += (tend.tv_nsec - tstart.tv_nsec) / 1000000000.0;
        expriment_t result = {count, fail, total_t};
        return result;
      } else if (buffer[5] == ANT_MESSAGE_EVENT_TRANSFER_TX_FAILED) {
        fail++;
        state = DM_SENDING_PACKET;
      }
    }
  }
  expriment_t result = {count, fail, res};
  return result;
}
/* Experiment 1: Broadcast Data Transfer between two nodes */
void doExperiment1(char deviceType) {
  if (deviceType == 'm') {
    while (period >= END_FREQ) {
      resetANT();
      openChannel(ID_CHAN1, FREQ_CHAN1, period, true);
      ANT_SendBroadcastData(ID_CHAN1, data);
      printf("Started broadcast! Press return to exit!\n");
      getchar();
      closeANT(ID_CHAN1);
      ANT_delayMs(100);
      period = decreasePeriod(period, STOP_SINGLE);
    }
  } else {
    while (period >= END_FREQ) {
      uint8_t count;
      for (count = 0; count < RUN_COUNT; count++) {
        resetANT();
        openChannel(ID_CHAN1, FREQ_CHAN1, period, false);
        clock_gettime(CLOCK_MONOTONIC, &tstart);
        printAndWait(ANT_BROADCAST_DATA, ID_CHAN1);
        doExperiment(ANT_BROADCAST_DATA, ID_CHAN1,
                     &receiveMessageAndCount);
        closeANT(ID_CHAN1);
        ANT_delayMs(2000);
      }
      printf("done! Press Any key!\n");
      getchar();
      period = decreasePeriod(period, STOP_SINGLE);
    }
```

```
  }
}
/* Experiment 2: Broadcast Data Transfer between multiple nodes */
void doExperiment2(char deviceType) {
  if (deviceType == 'm') {
    while (period >= END_FREQ) {
      uint8_t count;
      for (count = 0; count < RUN_COUNT; count++) {
        resetANT();
        printf("press key!\n");
        getchar();
        openChannel(ID_CHAN1, FREQ_CHAN1, period, true);
        ANT_SendBroadcastData(ID_CHAN1, data);
        printf("press key once slave received first packet!\n");
        getchar();
        openChannel(ID_CHAN2, FREQ_CHAN2, period, false);
        clock_gettime(CLOCK_MONOTONIC, &tstart);
        printAndWait(ANT_BROADCAST_DATA, ID_CHAN2);
        doExperiment(ANT_BROADCAST_DATA, ID_CHAN2,
                     &receiveMessageAndCount);
        printf("done! press key for next round\n");
        getchar();
        closeANT(ID_CHAN1);
        closeANT(ID_CHAN2);
        ANT_delayMs(2000);
      }
      period = decreasePeriod(period, STOP_DOUBLE);
    }
  } else {
    while (period >= END_FREQ) {
      uint8_t count;
      for (count = 0; count < RUN_COUNT; count++) {
        resetANT();
        printf("press key!\n");
        getchar();
        openChannel(ID_CHAN1, FREQ_CHAN1, period, false);
        clock_gettime(CLOCK_MONOTONIC, &tstart);
        printAndWait(ANT_BROADCAST_DATA, ID_CHAN1);
        openChannel(ID_CHAN2, FREQ_CHAN2, period, true);
        ANT_SendBroadcastData(ID_CHAN2, data);
        doExperiment(ANT_BROADCAST_DATA, ID_CHAN1,
                     &receiveMessageAndCount);
```

```
        printf("done! press key for next round\n");
        getchar();
        closeANT(ID_CHAN1);
        closeANT(ID_CHAN2);
        ANT_delayMs(2000);
      }
      period = decreasePeriod(period, STOP_DOUBLE);
    }
  }
}
/* Experiment 3: Ackowledge Data Transfer between two nodes */
void doExperiment3(char deviceType) {
  if (deviceType == 'm') {
    uint8_t data[4] = {0x70, 0x69, 0x6E, 0x67};
    while (period >= END_FREQ) {
      resetANT();
      openChannel(ID_CHAN1, FREQ_CHAN1, period, true);
      ANT_SendBroadcastData(ID_CHAN1, data);
      printAndWait(ANT_ACKNOWLEDGED_DATA, ID_CHAN1);
      doExperiment(ANT_ACKNOWLEDGED_DATA, ID_CHAN1,
                   &receiveMessageAck);
      closeANT(ID_CHAN1);
      getchar();
      period = decreasePeriod(period, STOP_SINGLE);
    }
  } else {
    while (period >= END_FREQ) {
      openChannel(ID_CHAN1, FREQ_CHAN1, period, false);
      printf("Slave node is listning!\n");
      printAndWait(0x00, ID_CHAN1);
      closeANT(ID_CHAN1);
      getchar();
      ANT_delayMs(2000);
      period = decreasePeriod(period, STOP_SINGLE);
    }
  }
}
/* Experiment 4: Ackowledge Data Delay */
void doExperiment4(char deviceType) {
  if (deviceType == 'm') {
    while (period >= END_FREQ) {
      openChannel(ID_CHAN1, FREQ_CHAN1, period, true);
```

```
      ANT_SendBroadcastData(ID_CHAN1, data);
      getchar();
      double speed_avr = 0;
      uint16_t i;
      for (i = 0; i < SLICE_COUNT; ++i) {
        flushBuffer();
        expriment_t result = measureMessageAck();
        results[i] = result.duration;
        speed_avr += results[i];
        printf("Run %d: Delay %f\n", i, result.duration);
      }
      speed_avr /= SLICE_COUNT;
      double stdDev = 0.0;
      for (i = 0; i < SLICE_COUNT; ++i) {
        double cur = results[i] - speed_avr;
        stdDev += cur * cur;
      }
      stdDev /= SLICE_COUNT;
      printf("Average delay : %f \tStd dev: %f\n",
             speed_avr,
             sqrt(stdDev));
      closeANT(ID_CHAN1);
      ANT_delayMs(2000);
      period = decreasePeriod(period, STOP_SINGLE);
      }
  } else {
    while (period >= END_FREQ) {
      openChannel(ID_CHAN1, FREQ_CHAN1, period, false);
      printf("Slave node is listning!\n");
      printAndWait(0x00, ID_CHAN1);
      closeANT(ID_CHAN1);
      ANT_delayMs(2000);
      period = decreasePeriod(period, STOP_SINGLE);
    }
  }
}
/* Experiment 5: Burst Data Transfer between two nodes */
void doExperiment5(char deviceType) {
  if (deviceType == 'm') {
    printf("Burst mode not working! Please use ANTWareII!\n");
  } else {
    uint32_t count = 0;
```

```
    openChannel(ID_CHAN1, FREQ_CHAN1, STD_FREQ, false);
    printAndWait(ANT_BROADCAST_DATA, ID_CHAN1);
    while(true) {
      s = ANT_RecvPacket_Blockfree(buffer, BC_ANT_BUFFSIZE);
      if (s == RS_PACKET_COMPLETE) {
        if (buffer[2] == ANT_BURST_DATA) {
          count++;
          if ((buffer[3] & 0xE0)== 0x00) {
            clock_gettime(CLOCK_MONOTONIC, &tstart);
            count = 1;
          } else if ((buffer[3] & 0x80) > 0) {
            clock_gettime(CLOCK_MONOTONIC, &tend);
            double total_t = (tend.tv_sec - tstart.tv_sec);
            total_t += (tend.tv_nsec - tstart.tv_nsec) / 1000000000.0;
            count--;
            printf("Received %d burst packets in %f s ==> %f\n",
                   count, total_t, count * 8 / total_t);
          }
        }
      }
    }
    closeANT(ID_CHAN1);
  }
}
/*  Experiment 6 - Communication Distance  */
void doExperiment6(char deviceType) {
  int pSetting = ANT_TRANSMIT_POWER_MINUS_20DBM;
  if (deviceType == 'm') {
    for(; pSetting <= ANT_TRANSMIT_POWER_0DBM; ++pSetting) {
      setTransmitPower(pSetting);
      openChannel(ID_CHAN1, FREQ_CHAN1, STD_FREQ, true);
      ANT_SendBroadcastData(ID_CHAN1, data);
      printf("Started broadcast! Press return to exit!\n");
      getchar();
      closeANT(ID_CHAN1);
    }
  } else if (deviceType == 's') {
      float distance = 0.0f;
    for(; pSetting <= ANT_TRANSMIT_POWER_0DBM; ++pSetting) {
      setTransmitPower(pSetting);
      ANT_delayMs(200);
      distanceState_t state = DS_INCRESING;
```

```
      printf("distance = %f\n", distance);
      while (true) {
        getchar();
        openChannel(ID_CHAN1, FREQ_CHAN1, STD_FREQ, false);
        while(true) {
          s = ANT_RecvPacket_Blockfree(buffer, BC_ANT_BUFFSIZE);
          if (s == RS_PACKET_COMPLETE) {
            if (buffer[2] == ANT_CHANNEL_EVENT &&
                buffer[5] == ANT_MESSAGE_EVENT_RX_SEARCH_TIMEOUT) {
              if (state == DS_DECREASING) {
                printf("No connection (current range %fm\n!",distance);
                printf("Decrease distance by 0.4m and press return!");
              }else {
                state = DS_DECREASING;
                printf("Lost connection (current range %fm)!", distance);
                printf(" Decrease distance by 0.4m and press return!");
              }
              distance -= .4f;
              break;
            } else if (buffer[2] == ANT_BROADCAST_DATA) {
              if (state == DS_DECREASING) {
                printf("Found connection (current range %fm)!\n",
                       distance);
                state = DS_DONE;
              } else {
                printf("Still in range (current %fm)\n!", distance);
                printf("Increase distance by 0.4m and press return!");
                distance += .4f;
              }
              break;
            }
          }
        }
        closeANT(ID_CHAN1);
        if (state == DS_DONE) {
          break;
        }
      }
    }
  }
}
```

Source code added to the ANT C library[9]:

```
typedef enum {
  RS_WAITING_FOR_SYNC = 0,
  RS_WAITING_FOR_MSGLENGTH,
  RS_WAITING_FOR_MSGTYPE,
  RS_WAITING_FOR_PAYLOAD,
  RS_WAITING_FOR_CHECKSUM,
  RS_PACKET_COMPLETE,
  RS_ERROR
} ReceiverState_t;

uint32_t ANT_RecvPacket_Blockfree(uint8_t *buffer,
                                  uint32_t size) {
  static ReceiverState_t state = RS_WAITING_FOR_SYNC;
  static uint8_t payloadLeft;
  static uint8_t mesg_len;
  static uint8_t checksum;
  static uint8_t* revcBufferPos = BC_ANT_recvBuffer;

  if (state == RS_PACKET_COMPLETE) {
    state = RS_WAITING_FOR_SYNC;
    revcBufferPos = BC_ANT_recvBuffer;
  }

  uint8_t c;
  if (antUARTreceivedChar()) {
    antUARTgetChar((char*) &c);
  } else {
    return state;
  }
  switch (state) {
    case RS_WAITING_FOR_SYNC:
      if (c == ANT_SYNCBYTE) {
        *revcBufferPos++ = c;
        checksum = ANT_SYNCBYTE;
        state = RS_WAITING_FOR_MSGLENGTH;
      }
      break;

    case RS_WAITING_FOR_MSGLENGTH:
      *revcBufferPos++ = c;
      mesg_len = c;
```

```
      checksum ^= c;
      payloadLeft = c;
      state = RS_WAITING_FOR_MSGTYPE;
      break;

    case RS_WAITING_FOR_MSGTYPE:
      *revcBufferPos++ = c;
      checksum ^= c;
      state = RS_WAITING_FOR_PAYLOAD;
      break;

    case RS_WAITING_FOR_PAYLOAD:
      *revcBufferPos++ = c;
      checksum ^= c;
      payloadLeft--;
      if (payloadLeft == 0) {
        state = RS_WAITING_FOR_CHECKSUM;
      }
      break;

    case RS_WAITING_FOR_CHECKSUM:
      *revcBufferPos++ = c;
      if (c == checksum) {
        if ((uint32_t) (mesg_len + 4) > size) {
          state = RS_ERROR;
          error("Receiver Buffer is not big enough! \
                 size : %d (need %d)!", size, mesg_len + 4);
        } else {
          state = RS_PACKET_COMPLETE;
          memcpy(buffer, BC_ANT_recvBuffer, mesg_len + 4);
        }
      } else {
        state = RS_ERROR;
      }
      break;

    default:
      state = RS_ERROR;
      break;
  }
  return state;
}
```

```
uint32_t ANT_SendBurstTransfer(
    uint8_t Channel,
    uint8_t* BurstData,
    uint32_t packetCount) {
  static uint8_t seq[3] = {0x20, 0x40, 0x60};
  if (packetCount == 1) {
    ANT_SendAcknowledgedData(Channel, BurstData);
    return 0;
  } else {
    uint32_t i;
    uint8_t *pos = BurstData;
    uint8_t channelSeq = Channel;
    for (i = 0; i < packetCount - 1; i++) {
      ANT_SendBurstTransferPacket(channelSeq, pos);
      pos += 8;
      channelSeq = Channel;
      channelSeq |= seq[i % 3];
    }
    channelSeq |= (1 << 7);
    ANT_SendBurstTransferPacket(channelSeq, pos);
  }
  return 0;
}
```

Bibliography

[1] Hugues Smeets, Tim Meurer, Chia-Yen Shih, and Pedro José Marrón. Demonstration abstract: A lightweight, portable device with integrated usb-host support for reprogramming wireless sensor nodes. In *Information Processing in Sensor Networks, IPSN-14 Proceedings of the 13th International Symposium on*, pages 333–334. IEEE, 2014.

[2] Dynastream Innovations Inc. ANTAP1MxIB RF Transceiver Module. `http://www.thisisant.com/resources/ant-ap1-transceiver-module-datasheet/`. [Online; accessed 27-September-2015].

[3] Dynastream Innovations Inc. ANT Message Protocol and Usage. `http://www.thisisant.com/resources/ant-message-protocol-and-usage/`. [Online; accessed 27-September-2015].

[4] Robert Lim, Federico Ferrari, Marco Zimmerling, Christoph Walser, Philipp Sommer, and Jan Beutel. Flocklab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems. In *Information Processing in Sensor Networks (IPSN), 2013 ACM/IEEE International Conference on*, pages 153–165. IEEE, 2013.

[5] Olof Rensfelt, Frederik Hermans, Christofer Ferm, Per Gunningberg, and Lars-Ake Larzon. Sensei-uu: a nomadic sensor network testbed supporting mobile nodes. In *the proc. of the 4th ACM International Workshop on Wireless Networks Testbeds, Experimental Evaluation, and Characterization*, 2010.

[6] Thomas Moser and Lukas Karrer. *The EventCollector Concept*. PhD thesis, MS thesis, ETH Zürich, Distributed Systems Group, 2001.

[7] Oliver Kasten and Marc Langheinrich. First experiences with bluetooth in the smart-its distributed sensor network. In *Workshop on Ubiquitous Computing and Communications, PACT*, volume 1, 2001.

[8] Dynastream Innovations Inc. ANT Windows Library Package with source code. `http://www.thisisant.com/resources/ant-windows-library-package-with-source-code/`. [Online; accessed 27-September-2015].

[9] Inc. BeatCraft. ANT/Library for PIC24F. `http://labs.beatcraft.com/en/index.php?ANT%2FLibrary%20for%20PIC24F`. [Online; accessed 27-September-2015].

[10] Christopher E. Strangio. The RS232 STANDARD - A Tutorial with Signal Names and Definitions. `http://www.camiresearch.com/Data_Com_Basics/RS232_standard.html`, 2015. [Online; accessed 27-September-2015].

[11] Dynastream Innovations Inc. ANT Channel Search and Background Scanning Channel. `http://www.thisisant.com/resources/an11-ant-channel-search-and-background-scanning-channel/`. [Online; accessed 27-September-2015].

[12] Dynastream Innovations Inc. Burst Transfers. `http://www.thisisant.com/resources/an04-burst-transfers/`. [Online; accessed 27-September-2015].

[13] Dynastream Innovations Inc. ANTware II. `http://www.thisisant.com/developer/resources/software-tools/`. [Online; accessed 27-September-2015].

[14] Dynastream Innovations Inc. AP1 Module product page. `http://www.thisisant.com/developer/components/ap1-module/`. [Online; accessed 27-September-2015].

[15] Dynastream Innovations Inc. AP2 RF Transceiver Module. `http://www.thisisant.com/resources/ant-ap2-transceiver-module-datasheet/`. [Online; accessed 27-September-2015].

[16] Hugues Smeets, Chia-Yen Shih, Marco Zuniga, Tobias Hagemeier, and Pedro Josè Marrón. Trainsense: a novel infrastructure to support mobility in wireless sensor networks. In *Wireless Sensor Networks*, pages 18–33. Springer, 2013.

Erklärung

German

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

English

I hereby declare that I have written this Bachelor thesis independently, using no other than the specified sources and resources, and that all quotations have been indicated.

Essen, September 27, 2015
(Place, Date)

Michael Krane