Submitted in fulfillment of the requirements for the degree of
Master of Science in Computer Science:
Computer Networks and Distributed Systems

Elaborate Energy Consumption Modelling for OpenWSN

Bruno Van de Velde

Adviser: Jeroen Famaey

Supervisor: Glenn Daneels

Esteban Municio

Faculty of Science
2017

Contents

Acknowledgements

I would like to thank Glenn Daneels and Esteban Municio for their guidance and feedback on the thesis. I would also like to thank Glenn Ergeerts for his help with the measurement setup. Finally, I would like to thank Prof. Famaey for the opportunity of writing this thesis.

Summary

A network consisting of devices running on batteries only has a limited lifetime. In order to extend how long the network can remain operational, the energy consumption of the devices in the network has to be minimized. Time slotted channel hopping (TSCH) is a reliable and ultra-low power medium access control technology that can be used for such networks.

In this thesis an energy consumption model for IEEE802.15.4e TSCH networks is provided, focussing on devices running the OpenWSN firmware. By identifying all network-related CPU state changes, our model provides a good representation for the device behavior and can be used to accurately predict the energy consumption when the model is used in a simulation.

Experimental verification of our model showed that the consumptions calculated by our model differs less than 1% compared to the measured consumptions. This difference includes measurement inaccuracies and the variations of the guard time. Only the remaining part of the difference is due to the simplifications that have to be made to be able to make a model from of a complex real-life situation.

Our model provides accurate energy consumption predications and is therefore suitable for 6TiSCH simulations with OpenWSN.

We also contributed a driver for the CC1200 radio chip on the OpenUSB hardware to the OpenWSN firmware project, which allows experimenting on the sub-1GHz band.

List of Abbreviations

6top 6TiSCH Operation Sublayer

ACK acknowledgement

AEM Advanced Energy Monitor

AGT Acknowledgment Guard Time

ASN Absolute Slot Number

IoT Internet of Things

MAC Medium Access Control

PGT Packet Guard Time

SFD Start of Frame Delimiter

SoC System on Chip

SWO Serial Wire Output

TSCH Time-Slotted Channel Hopping

WSN Wireless Sensor Networks

List of Tables

List of Figures

Elaborate Energy Consumption Modelling for OpenWSN

Bruno Van de Velde
University of Antwerp
IDLab,
Departement of Mathematics and Computer Science

Glenn Daneels
University of Antwerp
IDLab,
Departement of Mathematics and Computer Science

Esteban Municio
University of Antwerp
IDLab,
Departement of Mathematics and Computer Science

Jeroen Famaey
University of Antwerp
IDLab,
Departement of Mathematics and Computer Science

***Abstract*—In this paper an energy consumption model for IEEE802.15.4e TSCH networks is provided, focussing on devices running the OpenWSN firmware. By identifying all network-related CPU state changes, our model provides a good representation for the device behavior and can be used to accurately predict the energy consumption when the model is used in a simulation. Experimental verififcation of our model showed that the consumptions calculated by our model differs less than 1% compared to the measured consumptions. This difference includes measurement inaccuracies and the variations of the guard time. Only the remaining part of the difference is due to the simplifications that have to be made to be able to make a model from of a complex real-life situation. Our model provides accurate energy consumption predications and is therefore suitable for 6TiSCH simulations with OpenWSN.**

I. Introduction

As the popularity of the Internet of Things (IoT) grows, Wireless Sensor Networks (WSN) are becoming more popular and this leads to many challenges [1].

One major challenge is minimizing the energy consumption of the devices in the network (which are referred to as "motes"). Whether the military is monitoring an area to detect enemy intrusion or whether the concentration of dangerous gases in measured in industrial plants, the goal is to have motes that can run for many years on a small battery. It is easy to see that reducing the power consumption has a lot of benefits. The battery will last longer so it takes longer before the motes or their batteries have to be replaced. Alternatively the lifetime could be kept the same but the mote could be made even smaller as less space is required for the battery. There are also cases where it is hard or infeasible to replace the motes once their battery has run out, e.g., if "smart dust" [2] is deployed over an entire region.

One thing we can do to reduce the consumption is having a more energy-efficient Medium Access Control (MAC) protocol. If the mote has to be listening on its radio the whole time then it will consume a lot more power than when it only wakes up at the time another mote is transmitting. We therefore need a MAC protocol that reduces the time where the radio is active and increases the amount of time during which the mote can sleep. Many such protocols were developed over time [3], one of them being the MAC layer from IEEE 802.15.4 [4].

The IEEE 802.15.4e MAC amendment to the existing IEEE 802.15.4 standard enhances and adds functionalities to this MAC layer [5]. One of the newly added modes that was designed for low-power devices is Time-Slotted Channel Hopping (TSCH). This mode is mostly suited for multihop mesh networks. The time-slotted access makes the latency bounded and predictable and provides motes with a guaranteed bandwidth. By using multiple channels the capacity of the network can be increased and the channel hopping improves the reliability. TSCH networks can achieve 99.999% reliability [6] while providing a deterministic performance and energy consumption.

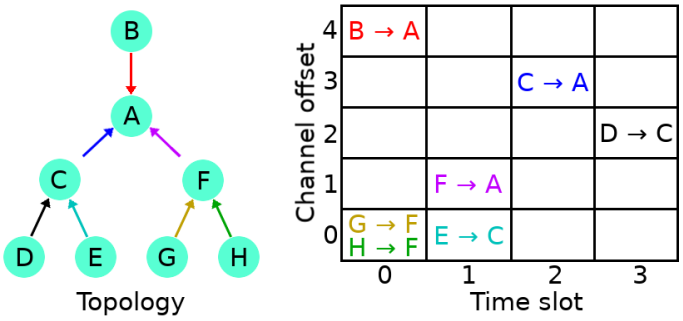

Fig. 1. Slot frame schedule example

A. Time-Slotted Channel Hopping

In TSCH networks, time is divided into time slots. Each slot provides enough time to transmit a MAC frame of the maximum size followed by an optional acknowledgement (ACK) frame indicating that the MAC frame was successfully received. During every time slot multiple channels can be used simultaneously, leading to a 2-dimensional grid of cells called a slot frame.

Fig. 1 shows an example of a slot frame with 5 channels and 4 time slots, for 8 motes labelled A to H. Each cell in the grid represents a specific time slot and channel offset in which a directed communication between motes can be assigned. These assigned cells can either be dedicated to a single transmitter, or they can be shared between multiple motes (like $G \rightarrow F$ and $H \rightarrow F$ in the example). A shared cell can be useful for sporadic or unpredictable traffic.

The slot frames are continuously repeated over time. The cells in the slot frame can however still be updated dynamically, so not every slot frame has to be identical. The schedule

of the slot frames is synchronized across all motes, so they know in which slot to transmit, receive or sleep.

TSCH also uses channel hopping to combat multi-path fading and external interference [7]. This channel hopping depends on the Absolute Slot Number (ASN) and the amount of channels. If the slot frame size (amount of slots in a single slot frame) is not a prime number, it may therefore be possible that not every frequency is used. The exact frequency to communicate on is determined by $frequency = F((ASN + channelOffset) \mod nrOfChannels)$ where F is a lookup table containing the set of available channels.

In this paper we only focused on TSCH in IEEE 802.15.4e [8], but the research is also applicable to other protocols using TSCH such as WirelessHART and ISA100.11a [9].

B. The importance of energy models

To develop MAC protocols we somehow need to measure how well the protocol performs. We need a way to predict the lifetime of the mote or be able to compare different MAC protocols with each other. It is infeasible to perform large scale tests with thousands of motes or to perform tests that last several years. Instead we need an energy model that we can use to simulate a network of any size. Having a realistic energy model is crucial to optimize the energy consumption of MAC protocols. Such models can also help with improving the topology by showing which nodes are overloaded and will run out of battery power before the others. The more accurate the model, the more useful the results of the simulation will be to predict real world consequences.

C. Contributions

We created an elaborate energy consumption model for accurately predicting the energy consumption of motes running OpenWSN firmware and forming a 6TiSCH network. Although based on existing research, we built the model from the ground up. Using the most recent firmware version and new hardware, our model is more up-to-date than the existing ones. The model takes more different states and types of time slots into account, allowing to more accurately predict the energy consumption.

This model can be combined with the existing OpenSim simulator in OpenWSN. This allows simulations to be run that predict or analyze the energy consumption. Although the python code of the model is written, this integration is left for further research.

We also contributed a driver for the CC1200 radio chip on the OpenUSB hardware to the OpenWSN firmware project. This was necessary for us to use the hardware, but will also allow others to make use of the hardware for other research. Unlike the other supported radio chips which operate on the 2.4 GHz band, the CC1200 operates on the sub-1GHz band.

D. Paper organization

We start by providing an overview of what we used in this research and point to similar research in Section II. We elaborate on our model in Section III and compare it with experimental measurements in Section IV to show how accurate the model is. Our conclusions and future work can be found in Section V.

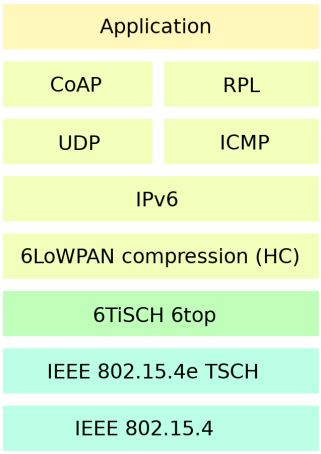

Fig. 2. 6TiSCH stack

II. Background and related work

In this section we provide an overview of the hardware, software and protocols that we have used.

A. OpenMote hardware

We worked with the OpenMote, a modular open-hardware ecosystem designed for the Industrial IoT [10]. The platform was developed at UC Berkeley and was designed to efficiently implement IoT standards such as the upcoming IETF 6TiSCH.

The OpenMote-CC2538 is the core of the OpenMote hardware ecosystem. It is the most important component, the other components (e.g., the OpenBattery) can be considered as extensions to it. It features a TI CC2538 SoC that consists out of a 32 MHz microcontroller with 32 kB of RAM available and an IEEE 802.15.4-compliant 2.4 GHz radio.

The OpenUSB version that we worked with has a CC1200 radio chip. Unlike the CC2538 which contains a 2.4 GHz radio, the CC1200 is a radio transceiver that operates on the sub-1GHz band. This allows long-range communication between the motes. The OpenUSB Rev.D used in this paper was a preliminary board and is thus not identical to the OpenUSB that is being sold and shipped, but they are equivalent.

B. 6TiSCH

6TiSCH is a protocol stack that tries to standardize IPv6 on top of the TSCH mode of IEEE 802.15.4e [11]. Above the IEEE 802.15.4e TSCH layer sits the 6TiSCH Operation Sublayer (6top) that allows a scheduling entity to manage the TSCH schedule of the network. This layer is necessary to let other IETF standards such as 6LoWPAN [12], RPL [13] and CoAP [14] work together with IEEE 802.15.4e TSCH. The protocol stack is shown in Fig. 2.

6top enables distributed scheduling in 6TiSCH networks [15]. Motes can negotiate to add or delete TSCH cells in the slot frame of their neighbor. For example, a mote can request additional cells to its parent mote when its traffic demand increases. Cells that are reserved like this are called "soft cells". 6top also supports centralized schedulers that can

allocate "hard cells" which cannot be dynamically reallocated by 6top itself.

1) TSCH Slot Template: 6top allows changing slot types to indicate that the mote should transmit, listen or put its radio to sleep. For IEEE 802.15.4e we identified 7 different types of time slots:

- TxDataRxAck: The mote sends a frame during this time slot and receives an ACK when the data has been received successfully.
- TxData: The mote sends a frame during this time slot but does not expect an ACK (e.g., broadcast or multicast frames such as DIO messages).
- RxDataTxAck: The mote listens and receives a frame in this time slot and replies with an ACK to indicate that it successfully received the frame.
- RxData: The mote listens and receives a frame in this time slot but no ACK is send (e.g., broadcast or multicast frames).
- RxIdle: The mote listens but does not receive a frame in this time slot.
- Sleep: The mote does not transmit or receive during this time slot.
- TxDataRxAckMissing: The mote sends a frame and expects and ACK, but no ACK is received. This could be caused by a collision of the data frame.

2) OpenWSN: OpenWSN is an open-source project that implements the 6TiSCH standard [16]. This means that it provides a complete protocol stack based on IoT standards such as 6LoWPAN, RPL and CoAP. The hierarchical design of the project makes it relatively easy to port the project to new hardware platforms. Hardware drivers for the most common motes are already available inside the OpenWSN project itself.

Next to the firmware, they also provide useful software, such as the OpenVisualizer. Although the main use of the OpenVisualizer project is to connect the OpenWSN network to the internet, it also provides the ability to monitor the network. The tool shows the internal state (neighbor table, scheduling table, packet queue, etc.) of all the motes that are physically connected to the computer running the OpenVisualizer. It also has the ability to run simulated motes and to debug the communication with Wireshark [17].

C. Energy models

As modelling the energy consumption is an important subject, many papers have already been written about it. Several of which have also dealt with the consumption of TSCH networks.

Some works focused on specific features in TSCH. De Guglielmo et al. analyzed the IEEE 802.15.4e TSCH CSMA-CA algorithm that is used in shared time slots [18]. Papadopoulos et al. investigated the impact of the guard time in TSCH [19]. They made the guard time smaller when motes are closer to their sink and concluded that it resulted in significant savings in energy consumption without compromising the reliability of the network.

Other works such as Juc et al. compared the TSCH and DSME modes of 802.15.4e [20]. They found that the energy consumption of TSCH tends to lie higher compared to DSME mode. This is due to the large fixed guard time in TSCH and because DSME can aggregate multiple ACKs and transmit a single group ACK.

Finally, X. Vilajosana et al. presented an energy model for TSCH networks, making use of OpenWSN for their experimental validation [21]. The values from the model were compared against measurements on GINA and OpenMote-STM32 platforms.

Our research followed the same lines as the last mentioned paper, but with several differences and improvements. OpenWSN is continuously updated and the current firmware is different than the version from 2013. By using the OpenMote-CC2538 and a preliminary OpenUSB Rev.D board, we also used state-of-the-art hardware. Instead of looking at two different platforms, we focused on a single platform and studied the differences between using a 2.4 GHz and a sub-1GHz radio. We also explicitly looked at the difference in power consumption between using a System on Chip (SoC) and the case with a separate microcontroller and radio chip. All the steps to reach the model are explained in detail, making this research reproducible. This way, the model can still be used for different types of hardware by simply changing some of the measured values.

III. Model

In this section we introduce the model by showing the different states in each time slot type and introduce the formula to calculate the consumption of each slot.

A. Time slots

Fig. 3 presents a general overview of the activity of a transmitter mote during a TxDataRxAck time slot and a receiver mote during a RxDataTxAck time slot.

Our model divides each time slot into different states. Some of the states seen in Fig. 3 consist of a part where the CPU is active and a part where the CPU is sleeping, which is seen as two different states in our model. The state of the radio in our model only changes at moments when the CPU state changes. This is of course a simplification as in the real world the radio state changes a little before or after this moment, typically while the CPU is active.

We only discuss the TxDataRxAck time slot in full detail. The other slots are similar and we only talk about the differences with the TxDataRxAck slot.

1) TxDataRxAck: Table I and Fig. 4 illustrate the different states in a TxDataRxAck slot and what the CPU and radio states are during each moment. In the figure, the CPU has two states (Sleep and Active) while the radio has three states (Sleep, Idle and Active). The radio being active refers to it either being in Listen, Rx or Tx mode.

At the beginning of each time slot, the CPU wakes up and performs the tasks required for any slot. This includes incrementing the ASN and scheduling the next state depending

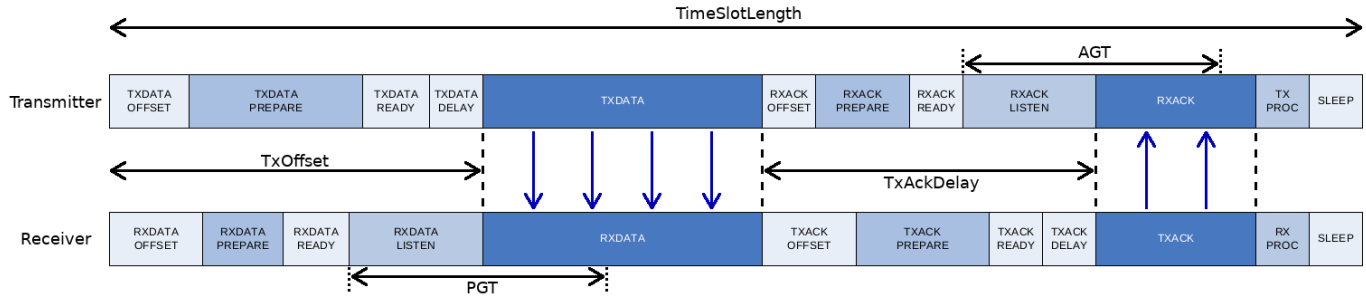

Fig. 3. General states in TxDataRxAck and RxDataTxAck time slots

Fig. 4. States in a TxDataRxAck slot

TABLE I. STATES IN A TXDATARXACK SLOT

State in slot	CPU State	Radio State
TxDataOffsetStart	Active	Sleep
TxDataOffset	Sleep	Sleep
TxDataPrepare	Active	Idle
TxDataReady	Sleep	Idle
TxDataDelayStart	Active	Idle
TxDataDelay	Sleep	Tx
TxDataStart	Active	Tx
TxData	Sleep	Tx
RxAckOffsetStart	Active	Sleep
RxAckOffset	Sleep	Sleep
RxAckPrepare	Active	Idle
RxAckReady	Sleep	Idle
RxAckListenStart	Active	Idle
RxAckListen	Sleep	Listen
RxAckStart	Active	Rx
RxAck	Sleep	Rx
TxProc	Active	Idle
Sleep	Sleep	Sleep

on the type of the slot. The CPU then sleeps again during TxDataOffset until the moment the radio is needed.

During TxDataPrepare, the radio wakes up, the channel is set and the bytes to transmit are loaded into the radio. The duration of this state is variable mainly because the time to load the bytes depend on the frame size. Since this state always starts at the same offset and has a variable duration, there is some time left between the TxDataPrepare and the actual transmission. During this TxDataReady state, the radio is in Idle mode while waiting until it is time to transmit. To minimize the energy consumption of the mote, the duration of the TxDataReady state should thus be as small as possible.

The first byte behind the Start of Frame Delimiter (SFD) has to be transmitted exactly $TxOffset$ ms after the start of the time slot. In order to do so, the time required to switch the radio from Idle to Tx mode has to be taken into account. The duration of the TxDataDelay equals the time between the Tx command being sent to the radio and the moment the SFD has been transmitted.

After the RxAckOffset that follows where the mote sleeps, the RxAckPrepare state then prepares the radio again by waking it up and setting the correct channel. Any time less than the maximum duration of RxAckPrepare is then spent in the RxAckReady state.

The ACK is transmitted exactly $TxAckDelay$ ms after the end of the TxData state. Because the clocks of the transmitting and receiving mote may not be perfectly in sync, the ACK might arrive slightly earlier or later than expected. The radio is thus turned on at the start of the RxAckListen instead of just in time for the data. If no ACK is received during the Acknowledgment Guard Time (AGT) period, the mote turns off the radio and considers the transmission failed. The duration of the AGT is defined as 1000 μs in OpenWSN. When the clocks between the motes are perfectly in sync, the RxAckListen state has a duration of $AGT/2$ plus the time to change the radio from Idle mode to Rx mode (which is considered to be instantaneous in OpenWSN).

During the TxProc state, the ACK is read from the radio and the transmission is considered successfully when the ACK is valid. The mote also synchronizes its clock based on the offset between $TxAckDelay$ and the actual data reception time, if the ACK came from its parent in the network graph. For the remaining part of the time slot, both the CPU and radio are in Sleep mode.

2) RxDataTxAck: This time slot can be considered the opposite of the TxDataRxAck. The states to handle the data in TxDataRxAck are found in handling the ACK in RxDataTxAck and vice versa.

The guard time for the data is however larger than the AGT that is used for ACKs. The Packet Guard Time (PGT) determines how long the radio listens for the data before the radio is turned off. When no data is received during the PGT period, we classify the time slot as RxIdle instead of RxDataTxAck. In OpenWSN, the PGT is defined as 2600 μs.

3) TxData and RxData: When no ACKs are required (e.g., for broadcasts), only the first half of the time slot is used. During the TxData and RxData slots, the mote sleeps once the data has been transmitted or received.

4) RxIdle: When the transmitter has no data to send, the slot that could have been a TxDataRxAck becomes a Sleep

slot. But on the receiver side a different type of slot is needed to represent the behavior of the mote. The RxIdle slot occurs when the receiver expects data but does not receive anything. This behavior is not an error, it simply means that a slot was reserved but the transmitter did not have any data to send at that moment.

5) Sleep: In time slots where no data has to be transmitted or received, the mote can sleep during the whole slot. The mote only briefly wakes up at the start of the slot to e.g., increment the ASN.

6) TxDataRxAckMissing: There are many error states in OpenWSN. The code would get in such error state when e.g., the radio remains active too long or when the prepare state lasts longer than the maximum allowed time. It is unlikely that the code would end up in most of these error states unless when there is a configuration issue. There is however one error state which is likely to occur eventually: a missing ACK. In the TxDataRxAck slot, data is transmitted and an ACK is received. In the slot that we refer to as TxDataRxAckMissing, the ACK is expected but not received.

B. Building the model

With all states per time slot identified and with the consumptions and durations during each state measured, the model can be built.

The consumption of a time slot is given by the following formula. The resulting consumption is expressed as the charge drawn from the battery, in Coulombs.

$$SlotCons = \sum_{State \in Slot} duration(State) * current(State)$$

If we apply this for all slot types and shorten the notation of the terms then we find the formula below which calculates the charge drawn for each different time slot.

$$\forall Slot \in SlotTypes : Q_{Slot} = \sum_{State \in Slot} \Delta t_{State} * I_{State}$$

The unit of the duration is ms while the unit of the current is mA. This means that the unit of the resulting charge is μC.

The model with consumptions of each slot type could be used in a simulation where the consumption is increased each slot as the simulation runs. To calculate the consumption over time, having a model of the entire slot frame is more practical. We did not build such a model ourselves but instead relied on the existing model described in the "Slot-Frame Energy Consumption Modeling" from the "A Realistic Energy Consumption Model for TSCH Networks" paper [21]. The charge drawn per slot in their calculations can be replaced by the measurements performed in this paper and the formula above.

C. Support for different hardware

Since the model has so many inputs, in order to use it for different hardware, many numbers might have to be adapted. This is necessary if a highly accurate simulation is needed with such hardware. The model can however be easily simplified to be used with different hardware more quickly, at the cost of some accuracy. By setting the duration of short states to 0, the model becomes easier to change as only the states that have the most impact on the consumption will have to be updated. Alternatively the duration can be estimated instead of measured as most durations are very similar to the ones we measured with the OpenMote hardware. The consumption of CPU and radio also do not have to be measured, the values could be taken from the datasheet. The result is a slightly less accurate model, but no or few measurements have to be made to be able to use the model to simulate any hardware.

IV. Results

In this section we discuss the results of our measurements and experimentally verify the accuracy of the model.

A. State durations

We measured the duration of each state in every time slot where the CPU is active. The durations in which the CPU is sleeping could then be trivially calculated.

States do not always have the exact same duration for a variety of reasons. There could be multiple code branches (different execution paths), the packet size could influence it or the duration of an operation can simply be variable (e.g., waking up the CC1200 chip). Multiple measurements had to be made to find a single duration that could be associated with the state.

Changing the CC1200 mode from Sleep to Idle takes between 246 and 343 μs, which causes every state where the radio wakes up to have a variable duration. It only required a few measurements to find that the median for waking up is 268 μs. However, we decided to use the average value instead of the median because it would result in slightly more accurate energy consumption prediction. To avoid being susceptible to outliers, we measured the wakeup time over ten thousand times and found an average of 273 μs.

For states with multiple code branches, the average duration was also taken. Since these are states where the radio is not active, these small variations on the duration only have a small impact on the total slot consumption.

To measure states where packets are loaded to and read from the radio, we measured the duration before and after the radio is accessed. We then separately measured the communication with the radio for different packet sizes (0 to 125 bytes with steps of 25 bytes). We performed linear interpolation on the measured durations to come up with a formula that works for all packet sizes.

The duration of transmitting and receiving also depends on the packet size. Since the radio has a baud rate of 250 kbps, the time to transmit one bit is 4 μs, which makes the time to transmit a byte 32 μs. We thus simply have to multiple the amount of transmitted bytes with 32 μs to find the duration. The PHY header byte and 2 bytes CRC also have to be included as they are sent with the packet. We measured the time between the start of frame and end of frame interrupts to verify that we can use this calculation and found that on average the error was only 0.13% with our measurements.

TABLE II. CONSUMPTION OF DIFFERENT DEVICE STATES

CPU state	Radio state	Consumption (in mA)	
		CC2538	CC1200
Active	Sleep	18.5253	18.5977
Active	Idle	18.5253	21.0067
Active	Listen	36.0883	43.3729
Active	Rx	32.1613	57.3220
Active	Tx	36.1228	59.3448
Sleep	Sleep	12.1690	12.4005
Sleep	Idle	12.1690	15.0322
Sleep	Listen	29.6143	38.2895
Sleep	Rx	25.5274	50.7769
Sleep	Tx	29.6779	53.6732

To model the guard time we assumed that the clocks are in sync. The packet thus always arrives exactly in the center of guard interval in our model.

B. Device state consumption

We measured the consumption of the OpenMote-CC2538 while connected to the OpenUSB during different device states. Since the CPU and radio are the two components responsible for the majority of the energy consumption, these device states are all combinations between CPU and radio modes. Instead of measuring the consumption of the CPU and radio separately, we measured the consumption of the entire device. The result is that any energy consumption not related to the CPU or radio (e.g. SPI or timers) are measured as part of the CPU usage. This allows a slightly more accurate energy prediction compared to models that ignore the other components.

Table II shows the consumption of different device states. The values for the Tx state were measured when the transmit power of the radio was set to 0 dBm.

The CC2538 has an identical consumption when the radio is in Sleep or Idle state because the CC2538 consists of both the CPU and radio and the radio itself does not have a separate Sleep state. Instead it has a single "Off" state of which we use the consumption for both the Sleep and Idle states.

The CPU usage while sleeping is very high. This is caused because the OpenMote-CC2538 code in OpenWSN currently uses the lowest possible sleep mode. When using the CC2538 radio, the consumption dropped to 1.6420 mA when entering deep sleep (SYS_CTRL_PM_2 specifically). This is still high because the OpenUSB is still consuming power. Since the OpenWSN version that uses the CC2538 does not access the CC1200 at all, the CC1200 chip is still in Idle mode which is where this extra consumption is coming from. When the OpenMote-CC2538 is not connected to the OpenUSB (which contained this CC1200 chip), the measured consumption during deep sleep drops to only 0.0317 mA. With future updates to OpenWSN it is thus expected that the consumption of the CPU in Sleep state will be more comparable to that of different hardware.

If the CPU is put in deep sleep when the CC1200 radio is used, the consumption drops to 0.7611 mA. The consumption could still be lower if the radio went into a deeper sleep mode as well (SLEEP instead of XOFF).

TABLE III. MEASURED AND CALCULATED CONSUMPTION FOR EACH SLOT TYPE, IN μC

Slot type	Measured		Calculated	
	CC2538	CC1200	CC2538	CC1200
TxDataRxAck	283.34	446.72	284.60	445.17
RxDataTxAck	287.41	458.68	286.22	457.78
TxData	262.07	386.76	262.78	388.01
RxData	265.39	399.98	263.09	397.01
RxIdle	229.61	260.97	229.33	261.15
Sleep	184.19	183.63	182.90	186.36
TxDataRxAckMissing	280.06	417.35	279.89	418.85

C. Slot consumption

With the duration and consumption of each state we could calculate the consumption for each type of slot. We also measured the consumption of entire slots to experimentally verify the correctness of our calculated values in our model. Table III shows the values we measured and calculated for each of the slot types for both radios. We configured both radios to have a transmit power of 0 dBm and sent packets of 127 bytes (the maximum packet size when including the CRC bytes).

As seen in Table III, the difference between the measured and calculated values is relatively small. The main contributors to these differences are small measurement errors and the variations in guard time duration. In the measured data, the guard time is a little smaller or larger than in the calculated data, which assumes perfectly synchronized clocks. On average the difference is only 1.3 μC or 0.457%, which shows that our model provides a good representation for real-life scenarios.

Fig. 5 shows the current during a TxDataRxAck time slot according to both the model and the measurements. The peaks on the graphs do not perfectly match, because the model simplified certain states. The radio state may be changed while the CPU is active, causing the CPU and radio to be active at the same time, while the model might only consider the radio as active once the CPU goes to sleep. This results in a peak in the measured time slot where there is no peak in the model.

D. Slot frame consumption

We also compared the consumption calculated by our model with measurements for entire slot frames.

We set up a network of three motes with a fixed topology, as shown in Fig. 6. The leaf mote was configured to send one packet of 127 bytes (including CRC) every 2 seconds. Slot frames consisted of 51 time slots of which one was configured for data from the leaf to the relay mote and one for data from the relay to the root mote. The first time slot in the slot frame is reserved for advertisements and used to send Enhanced Beacon frames. We did not look at the broadcasted beacons and DAO packets that were send every 30 and 60 seconds, we only focussed on the packets coming from the leaf mote. The first time slot in each slot frame is thus considered to be of type RxIdle.

Since beacons and similar packets were ignored, the consumption values may not offer a good prediction for how long the battery of the mote will last. The goal of the comparison of slot frame consumption was however to verify how accurate the model is by calculating and measuring similar circumstances. It is thus not important that the scenario with

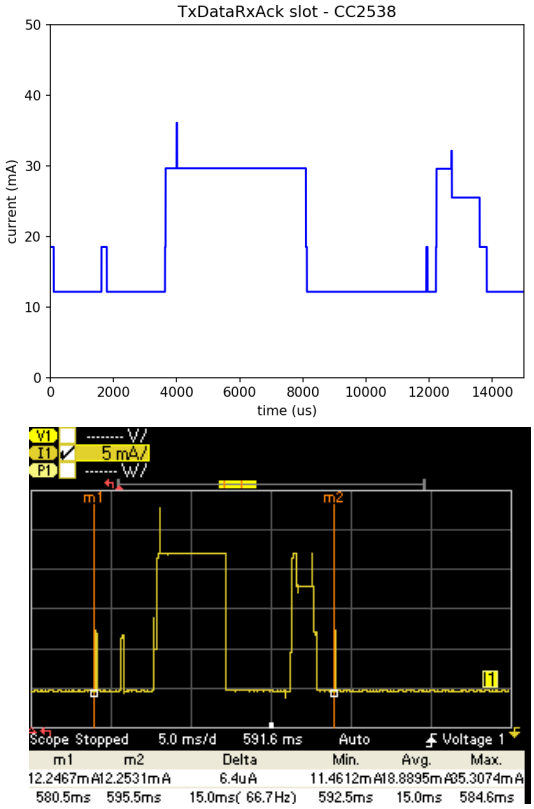

Fig. 5. Comparison between calculated and measured TxDataRxAck time slot when using the CC2538 radio

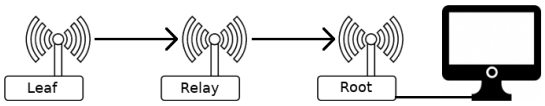

Fig. 6. Topology used while comparing the consumption of a slot frame

only 3 motes is less likely and that the activity in a slot frame is simplified.

The slot frame of the leaf mote always consists of 1 RxIdle slot and at least 49 Sleep slots. The final slot will be either TxDataRxAck when there is data to send or another Sleep slot. Since there are 51 slots in a slot frame and every time slot lasts 15 ms, the duration of each slot frame is 765 ms. Since a packet is send every 2 seconds, the consumption of the leaf slot frame can be considered as follows:

$$Q_{leaf} = Q_{RxIdle} + 49 * Q_{Sleep} + (0.3825 * Q_{TxDataRxAck} + 0.6175 * Q_{Sleep})$$

The slot frame of the relay mote can be determined in a similar way. There are RxIdle and Sleep slots when no packet is received, while there are RxDataTxAck and TxDataRxAck slots when a packet was received and forwarded. The first time slot is also of type RxIdle and the remaining 48 slots are always Sleep slots. The formula for the consumption of the slot frame of the relay mote thus becomes:

$$Q_{relay} = Q_{RxIdle} + 48 * Q_{Sleep} + (0.3825 * (Q_{RxDataTxAck} + Q_{TxDataRxAck}) + 0.6175 * (Q_{RxIdle} + Q_{Sleep}))$$

TABLE IV. MEASURED AND CALCULATED SLOT FRAME CONSUMPTION, IN μC

	Measured		Calculated	
Mote type	CC2538	CC1200	CC2538	CC1200
Leaf	9499.80	9580.50	9413.23	9678.14
Relay	9543.75	9742.71	9481.42	9828.15

We will not focus on the consumption of the root mote. It is connected to the computer and serves as a gateway to the internet, therefore the mote typically does not run on batteries. Serial communication cannot be disabled on this mote and the measured consumption would therefore not fully correspond to our model.

We measured the consumption of the slot frames for the leaf and relay mote and compared them with the values calculated through the model using the above formulas. The result can be found in Table IV. On average the error between the calculated and measured values are below 1% as expected. An error of around 66.3 μC is normal based on our average error of 1.3 μC that would occur in each of the 51 time slots. The error seems to be slightly higher here however, mainly because the majority of time slots were Sleep slots where the error per time slot was above average.

The comparison again shows that our model is quite good. For example, the difference between the measured and calculated consumptions for the Leaf mote for the CC2538 radio is only 0.91%. However, we have to be careful with using these numbers to show how accurate the model is exactly. Since the difference is small, the measurement errors will become relatively large. A measurement error of only 0.1% on both the measured slot frame consumption and the consumptions on which the calculated values are based could already cause the difference to be 0.71% instead of 0.91%.

V. CONCLUSION

We have proposed a new energy model for time slots in OpenWSN that takes all network-related CPU state changes into account. We have experimentally verified that when inputting the durations and consumptions of the OpenMote hardware, the calculated energy consumption by the model is very close to the (measured) consumption of this hardware. In our setup the error between the calculated and measured energy consumption was less than 1%. This makes our model suitable for use in simulations. The ability to calculate the consumption based on the packet size also improves upon existing models that only provide the consumption of an entire time slot with a packet of maximum size.

We also contributed a driver for the CC1200 radio chip on the OpenUSB hardware to the OpenWSN firmware project, which allows experimenting on the sub-1GHz band.

Our model provides accurate energy consumption predictions and is therefore suitable for 6TiSCH simulations with OpenWSN.

A. Future work

Once the OpenMote-CC2538 platform code is updated to allow deep sleep, the accuracy of our model can be examined over a longer duration, as a minor error on the Sleep slot would no longer lead to a large total error. The consumption of the CC1200 chip could also be reduced in the future by going into SLEEP mode instead of XOFF mode.

Ultimately, the goal of the model is to be used in a simulation. It should thus be integrated into the OpenSim part of the OpenVisualizer to be of real use. Another place where the model can be used is in the 6TiSCH simulator [22]. If the sleep consumption issues are fixed then the energy model used in the simulator could be replaced by the model described in this paper. The energy consumption found by the simulation will then take the packet size into account and be more accurate.

REFERENCES

[1] D. Christin, A. Reinhardt, P. S. Mogre, R. Steinmetz *et al.*, "Wireless sensor networks and the internet of things: selected challenges," *Proceedings of the 8th GI/ITG KuVS Fachgespräch Drahtlose sensornetze*, pp. 31–34, 2009.

[2] J. M. Kahn, R. H. Katz, and K. S. J. Pister, "Emerging challenges: Mobile networking for "smart dust"," *Journal of Communications and Networks*, vol. 2, no. 3, pp. 188–196, Sept 2000.

[3] P. Huang, L. Xiao, S. Soltani, M. W. Mutka, and N. Xi, "The evolution of mac protocols in wireless sensor networks: A survey," *IEEE Communications Surveys Tutorials*, vol. 15, no. 1, pp. 101–120, First 2013.

[4] "Ieee standard for local and metropolitan area networks–part 15.4: Low-rate wireless personal area networks (lr-wpans)," *IEEE Std 802.15.4-2011 (Revision of IEEE Std 802.15.4-2006)*, pp. 1–314, Sept 2011.

[5] "Ieee standard for local and metropolitan area networks–part 15.4: Low-rate wireless personal area networks (lr-wpans) amendment 1: Mac sublayer," *IEEE Std 802.15.4e-2012 (Amendment to IEEE Std 802.15.4-2011)*, pp. 1–225, April 2012.

[6] L. Doherty, W. Lindsay, and J. Simon, "Channel-specific wireless sensor network path data," in *2007 16th International Conference on Computer Communications and Networks*, Aug 2007, pp. 89–94.

[7] T. Watteyne, A. Mehta, and K. Pister, "Reliability through frequency diversity: why channel hopping makes sense," in *Proceedings of the 6th ACM symposium on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks*. ACM, 2009, pp. 116–123.

[8] T. Watteyne, M. Palattella, and L. Grieco, "Using ieee 802.15.4e time-slotted channel hopping (tsch) in the internet of things (iot): Problem statement," Tech. Rep., 2015.

[9] S. Petersen and S. Carlsen, "Wirelesshart versus isa100.11a: The format war hits the factory floor," *IEEE Industrial Electronics Magazine*, vol. 5, no. 4, pp. 23–34, Dec 2011.

[10] X. Vilajosana, P. Tuset, T. Watteyne, and K. Pister, "Openmote: open-source prototyping platform for the industrial iot," in *International Conference on Ad Hoc Networks*. Springer, 2015, pp. 211–222.

[11] D. Dujovne, T. Watteyne, X. Vilajosana, and P. Thubert, "6tisch: deterministic ip-enabled industrial internet (of things)," *IEEE Communications Magazine*, vol. 52, no. 12, pp. 36–41, December 2014.

[12] J. Hui and P. Thubert, "Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks," RFC 6282, Tech. Rep. 6282, Sep. 2011. [Online]. Available: https://rfc-editor.org/rfc/rfc6282.txt

[13] A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks," RFC 6550, Tech. Rep. 6550, Mar. 2012. [Online]. Available: https://rfc-editor.org/rfc/rfc6550.txt

[14] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," RFC 7252, Tech. Rep. 7252, Jun. 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7252.txt

[15] Q. Wang, X. Vilajosana, and T. Watteyne, "6tsch operation sublayer (6top)," 2013.

[16] T. Watteyne, X. Vilajosana, B. Kerkez, F. Chraim, K. Weekly, Q. Wang, S. Glaser, and K. Pister, "Openwsn: a standards-based low-power wireless development environment," *Transactions on Emerging Telecommunications Technologies*, vol. 23, no. 5, pp. 480–493, 2012.

[17] Wireshark Foundation, "Wireshark," https://www.wireshark.org.

[18] D. D. Guglielmo, B. A. Nahas, S. Duquennoy, T. Voigt, and G. Anastasi, "Analysis and experimental evaluation of ieee 802.15.4e tsch csma-ca algorithm," *IEEE Transactions on Vehicular Technology*, vol. 66, no. 2, pp. 1573–1588, Feb 2017.

[19] G. Z. Papadopoulos, A. Mavromatis, X. Fafoutis, N. Montavont, R. Piechocki, T. Tryfonas, and G. Oikonomou, "Guard time optimisation and adaptation for energy efficient multi-hop tsch networks," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, Dec 2016, pp. 301–306.

[20] I. Juc, O. Alphand, R. Guizzetti, M. Favre, and A. Duda, "Energy consumption and performance of ieee 802.15.4e tsch and dsme," in *2016 IEEE Wireless Communications and Networking Conference*, April 2016, pp. 1–7.

[21] X. Vilajosana, Q. Wang, F. Chraim, T. Watteyne, T. Chang, and K. S. J. Pister, "A realistic energy consumption model for tsch networks," *IEEE Sensors Journal*, vol. 14, no. 2, pp. 482–489, Feb 2014.

[22] T. Watteyne, K. Muraoka, N. Accettura, and X. Vilajosana, "The 6tisch simulator," https://bitbucket.org/6tisch/simulator, 2017.

Chapter 1

Introduction

As the popularity of the Internet of Things (IoT) grows, Wireless Sensor Networks (WSN) are becoming more popular and this leads to many challenges [1].

One major challenge is minimizing the energy consumption of the devices in the network (which are referred to as "motes"). Whether the military is monitoring an area to detect enemy intrusion or whether the concentration of dangerous gases in measured in industrial plants, the goal is to have motes that can run for many years on a small battery. It is easy to see that reducing the power consumption has a lot of benefits. The battery will last longer so it takes longer before the motes or their batteries have to be replaced. Alternatively the lifetime could be kept the same but the mote could be made even smaller as less space is required for the battery. There are also cases where it is hard or infeasible to replace the motes once their battery has run out, e.g., if "smart dust" [2] is deployed over an entire region.

One thing we can do to reduce the consumption is having a more energy-efficient Medium Access Control (MAC) protocol. If the mote has to be listening on its radio the whole time then it will consume a lot more power than when it only wakes up at the time another mote is transmitting. We

therefore need a MAC protocol that reduces the time where the radio is active and increases the amount of time during which the mote can sleep. Many such protocols were developed over time [3], one of them being the MAC layer from IEEE 802.15.4 [4].

The IEEE 802.15.4e MAC amendment to the existing IEEE 802.15.4 standard enhances and adds functionalities to the MAC layer [5]. One of the newly added modes that was designed for low-power devices is Time-Slotted Channel Hopping (TSCH). This mode is mostly suited for multi-hop mesh networks. The time-slotted access makes the latency bounded and predictable and provides motes with a guaranteed bandwidth. By using multiple channels the capacity of the network can be increased and the channel hopping improves the reliability. TSCH networks can achieve 99.999% reliability [6] while providing a deterministic performance and energy consumption.

1.1 Time-Slotted Channel Hopping

In TSCH networks, time is divided into time slots. Each slot provides enough time to transmit a MAC frame of the maximum size followed by an optional acknowledgement (ACK) frame indicating that the MAC frame was successfully received. During every time slot multiple channels can be used simultaneously, leading to a 2-dimensional grid of cells called a slot frame.

Figure 1.1 shows an example of a slot frame with 5 channels and 4 time slots, for a network with a topology displayed in Figure 1.2. Each cell in the grid represents a specific time slot and channel offset in which a directed communication between motes can be assigned. These assigned cells can either be dedicated to a single transmitter, or they can be shared between multiple motes (like $G \rightarrow F$ and $H \rightarrow F$ in the example). A shared cell can be useful for sporadic or unpredictable traffic.

The slot frames are continuously repeated over time as displayed in Figure 1.3. The cells in the slot frame can however still be updated dynamically, so not every slot frame has to be identical. The schedule of the slot frames is synchronized across all motes, so they know in which slot to transmit, receive

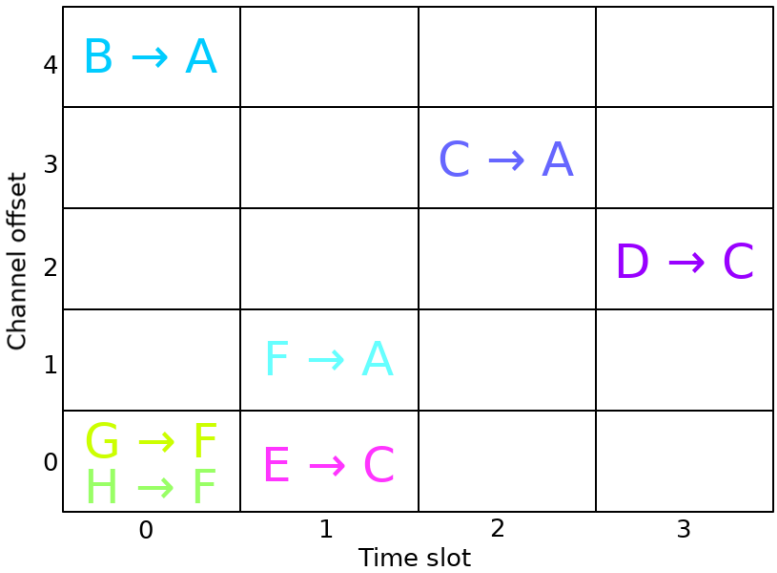

Figure 1.1: *Slot frame schedule example*

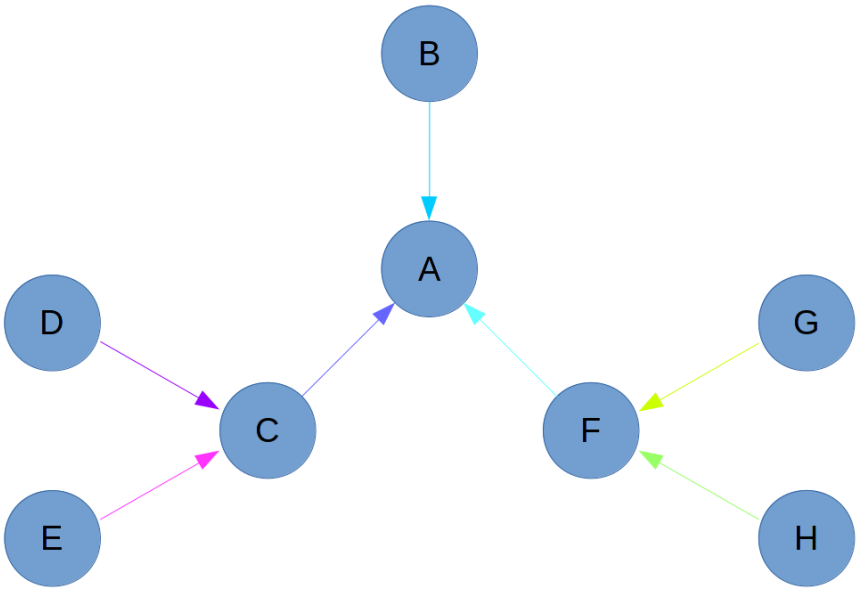

Figure 1.2: *Topology of example network*

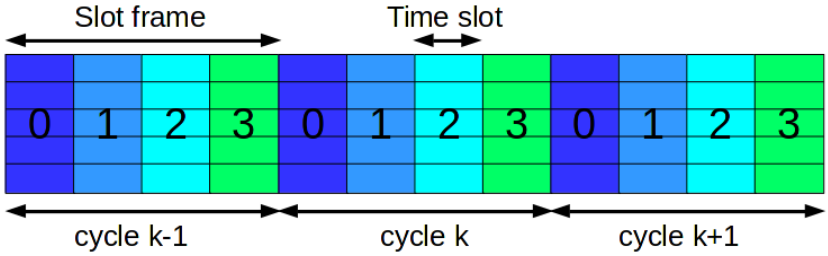

Figure 1.3: *Slot frames are repeated*

or sleep.

TSCH also uses channel hopping to combat multi-path fading and external interference [7]. This channel hopping depends on the Absolute Slot Number (ASN) and the amount of channels. If the slot frame size (amount of slots in a single slot frame) is not a prime number, it may therefore be possible that not every frequency is used. The exact frequency to communicate on is determined by the following function where F is a lookup table containing the set of available channels:

$$frequency = F((ASN + channelOffset) \mod nrOfChannels)$$

In this thesis we only focused on TSCH in IEEE 802.15.4e [8], but the research is also applicable to other protocols using TSCH such as WirelessHART and ISA100.11a [9].

1.2 The importance of energy models

To develop MAC protocols we somehow need to measure how well the protocol performs. We need a way to predict the lifetime of the mote or be able to compare different MAC protocols with each other. It is infeasible to perform large scale tests with thousands of motes or to perform tests that last several years. Instead we need an energy model that we can use to simulate a network of any size. Having a realistic energy model is crucial to optimize the energy consumption of MAC protocols. Such models can also help with improving the topology by showing which nodes are overloaded and will run out of battery power before the others. The more accurate the model, the more useful the results of the simulation will be to predict real world consequences.

1.3 Contributions

We created an **elaborate energy consumption model for accurately predicting the energy consumption of motes running OpenWSN**

firmware and forming a 6TiSCH network. Although based on existing research, we built the model from the ground up. Using the most recent firmware version and new hardware, our model is more up-to-date than the existing ones. The model takes more different states and types of time slots into account, allowing to more accurately predict the energy consumption.

This model can be combined with the existing OpenSim simulator in OpenWSN. This allows simulations to be run that predict or analyze the energy consumption. Although the **python code of the model** is written, this integration is left for further research.

We also contributed a driver for the CC1200 radio chip on the OpenUSB hardware to the OpenWSN firmware project. This was necessary for us to use the hardware, but will also allow others to make use of the hardware for other research. Unlike the other supported radio chips which operate on the 2.4 GHz band, the CC1200 operates on the sub-1GHz band.

1.4 Thesis organization

We start by providing an overview of what we used in this research and point to similar research in Chapter 2. The setup and preparations for our experiments are described in Chapter 3. We elaborate on our model in Chapter 4 and compare it with experimental measurements in Chapter 5 to show how accurate the model is. Our conclusions and future work can be found in Chapter 6.

The appendices provide extra information and data that was left out of the thesis. The thesis does not go over all time slot types in detail, but for all types we provide a list of all states and their duration in Appendix A and Appendix B. All figures comparing our time slot model with our experimental measurements are provided in Appendix C.

Chapter 2

Background and related work

In this chapter we provide an overview of the hardware, software and protocols that we have used.

We start by describing the hardware used in our measurements, then we introduce OpenWSN and the protocols it is based on and finally we talk about the existing models and the difference with our model.

2.1 OpenMote hardware

We worked with the OpenMote, a modular open-hardware ecosystem designed for the Industrial IoT [10]. The platform was developed at UC Berkeley and was designed to efficiently implement IoT standards such as the upcoming IETF 6TiSCH.

The OpenMote hardware ecosystem consists of four parts, the OpenMote-CC2538, OpenBattery, OpenBase and OpenUSB, shown in Figure 2.2.

2.1.1 OpenMote-CC2538

The OpenMote-CC2538 is the core of the OpenMote hardware ecosystem. It is the most important component, the other components (e.g., the OpenBattery) can be considered as extensions to it. It features a TI CC2538 SoC

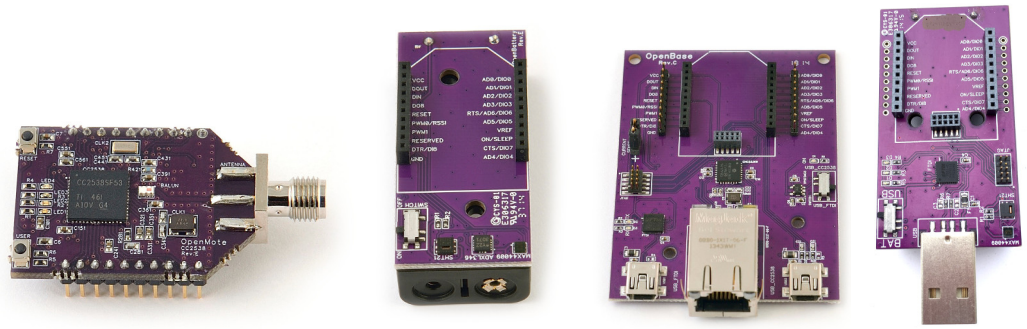

Figure 2.1: *OpenMote hardware ecosystem. From left to right: OpenMote-CC2538, OpenBattery, OpenBase, OpenUSB*

that consists out of a 32 MHz microcontroller with 32 kB of RAM available and an IEEE 802.15.4-compliant 2.4 GHz radio.

The form factor and pin-out were chosen to be the same as other popular low-power wireless board, such as the XBee and WaspMote, meaning that the OpenMote-CC2538 can be used with accessories built for those boards.

2.1.2 OpenBattery

The OpenBattery extension allows the OpenMote-CC2538 to be powered by two AAA batteries instead of having it powered by a USB. The battery pack also includes sensors that the OpenMote-CC2538 can access: a temperature/humidity sensor (SHT21), a 3-axis accelerometer (ADXL346) and a light sensor (MAX44009). The board also contains an on/off switch so that you can turn the device off when you are not using it.

2.1.3 OpenBase

While the OpenMote-CC2538 can be placed on an OpenBattery to run, it has to be connected to a computer in order to flash it. The OpenBase offers this functionality and also provides a way to debug the program. The OpenBase is connected to the computer with a Mini USB through which you can flash the OpenMote-CC2538 that is placed on the OpenBase. It also serves as serial input and output for the flashed program. The OpenBase also includes a

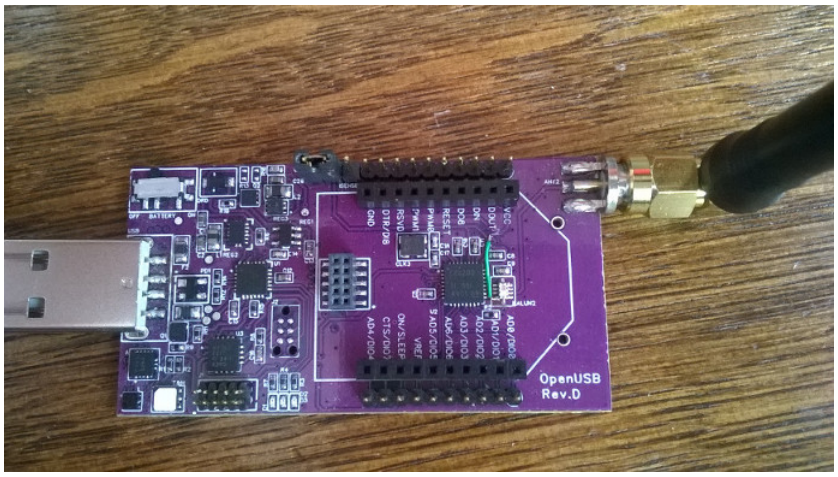

Figure 2.2: *OpenUSB Rev.D*

JTAG interface, allowing debugging the program by placing breakpoints and inspecting variables. Finally, there is also an Ethernet interface that allows the program to be connected to the internet without needing a computer.

2.1.4 OpenUSB

The OpenUSB is still in development. Originally it provided the same functionality as the OpenBattery, but also with a male USB connector and a JTAG interface. When an OpenMote-CC2538 is connected to an OpenUSB, it acts in the same way as the less powerful TelosB mote.

The OpenUSB version that we worked with also has a CC1200 radio chip. Unlike the CC2538 which contains a 2.4 GHz radio, the CC1200 is a radio transceiver that operates on the sub-1GHz band. This allows long-range communication between the motes. The maximum distance for communication with the CC1200 heavily depends on the configuration of radio parameters and is estimated to go from 292 meter up to 14978 meter in line-of-sight in the 868 MHz band [11].

The OpenUSB Rev.D used in this thesis (see Figure 2.2) was a preliminary board and is thus not identical to the OpenUSB that is being sold and shipped, but they are equivalent.

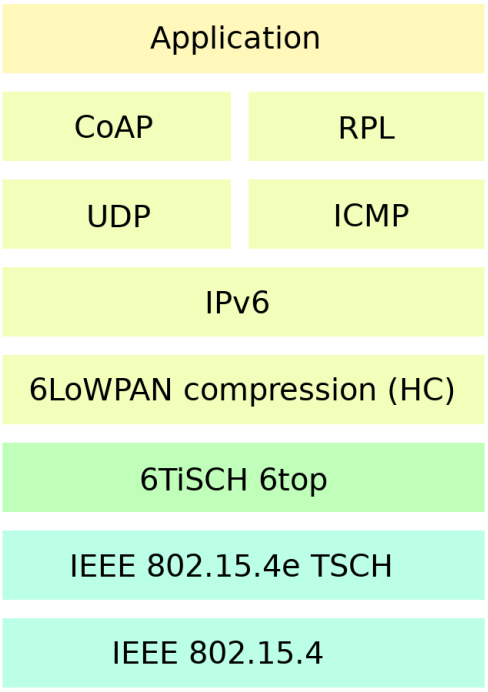

Figure 2.3: *6TiSCH stack*

2.1.5 OpenMote+

A newer board is currently being developed in the OpenMote ecosystem. Similar to the combination of the OpenMote-CC2538 and OpenUSB, the OpenMote+ will offer a dual-radio interface for short- and long-range communication on the 2.4 GHz and sub-1GHz bands [12].

2.2 6TiSCH

6TiSCH is a protocol stack that tries to standardize IPv6 on top of the TSCH mode of IEEE 802.15.4e [13]. Above the IEEE 802.15.4e TSCH layer sits the 6TiSCH Operation Sublayer (6top) that allows a scheduling entity to manage the TSCH schedule of the network. This layer is necessary to let other IETF standards such as 6LoWPAN [14], RPL [15] and CoAP [16] work together with IEEE 802.15.4e TSCH. The protocol stack is shown in Figure 2.3.

6top enables distributed scheduling in 6TiSCH networks [17]. Motes can

negotiate to add or delete TSCH cells in the slot frame of their neighbor. For example, a mote can request additional cells to its parent mote when its traffic demand increases. Cells that are reserved like this are called "soft cells". 6top also supports centralized schedulers that can allocate "hard cells" which cannot be dynamically reallocated by 6top itself.

2.2.1 TSCH Slot Template

6top allows changing slot types to indicate that the mote should transmit, listen or put its radio to sleep. For IEEE 802.15.4e we identified 7 different types of time slots:

- TxDataRxAck: The mote sends a frame during this time slot and receives an ACK when the data has been received successfully.
- TxData: The mote sends a frame during this time slot but does not expect an ACK (e.g., broadcast or multicast frames such as DIO messages).
- RxDataTxAck: The mote listens and receives a frame in this time slot and replies with an ACK to indicate that it successfully received the frame.
- RxData: The mote listens and receives a frame in this time slot but no ACK is send (e.g., broadcast or multicast frames).
- RxIdle: The mote listens but does not receive a frame in this time slot.
- Sleep: The mote does not transmit or receive during this time slot.
- TxDataRxAckMissing: The mote sends a frame and expects and ACK, but no ACK is received. This could be caused by a collision of the data frame.

2.2.2 OpenWSN

OpenWSN is an open-source project that implements the 6TiSCH standard [18]. This means that it provides a complete protocol stack based on IoT

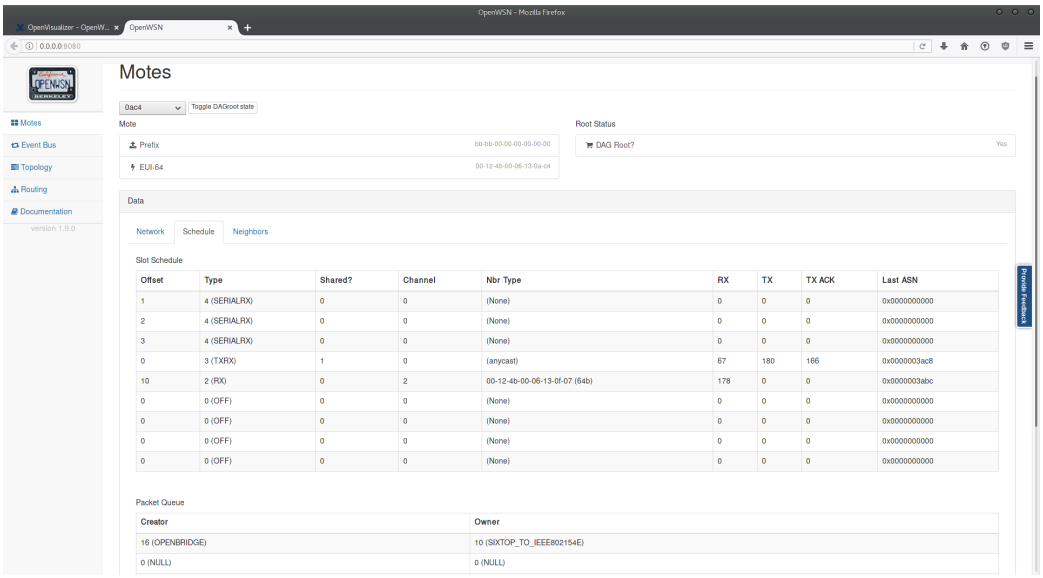

Figure 2.4: *OpenVisualizer web view*

standards such as 6LoWPAN, RPL and CoAP. The hierarchical design of the project makes it relatively easy to port the project to new hardware platforms. Hardware drivers for the most common motes are already available inside the OpenWSN project itself.

Next to the firmware, they also provide useful software, such as the OpenVisualizer shown in Figure 2.4. Although the main use of the OpenVisualizer project is to connect the OpenWSN network to the internet, it also provides the ability to monitor the network. The tool shows the internal state (neighbor table, scheduling table, packet queue, etc.) of all the motes that are physically connected to the computer running the OpenVisualizer. It also has the ability to run simulated motes and to debug the communication with Wireshark [19].

2.3 Energy models

As modelling the energy consumption is an important subject, many papers have already been written about it. Several of which have also dealt with the consumption of TSCH networks.

Some works focused on specific features in TSCH. De Guglielmo et al. ana-

lyzed the IEEE 802.15.4e TSCH CSMA-CA algorithm that is used in shared time slots [20]. Papadopoulos et al. investigated the impact of the guard time in TSCH [21]. They made the guard time smaller when motes are closer to their sink and concluded that it resulted in significant savings in energy consumption without compromising the reliability of the network.

Other works such as Juc et al. compared the TSCH and DSME modes of 802.15.4e [22]. They found that the energy consumption of TSCH tends to lie higher compared to DSME mode. This is due to the large fixed guard time in TSCH and because DSME can aggregate multiple ACKs and transmit a single group ACK.

Finally, X. Vilajosana et al. presented an energy model for TSCH networks, making use of OpenWSN for their experimental validation [23]. The values from the model were compared against measurements on GINA and OpenMote-STM32 platforms.

Our research followed the same lines as the last mentioned paper, but with several differences and improvements. OpenWSN is continuously updated and the current firmware is different than the version from 2013. By using the OpenMote-CC2538 and a preliminary OpenUSB Rev.D board, we also used state-of-the-art hardware. Instead of looking at two different platforms, we focused on a single platform and studied the differences between using a 2.4 GHz and a sub-1GHz radio. We also explicitly looked at the difference in power consumption between using a System on Chip (SoC) and the case with a separate microcontroller and radio chip. All the steps to reach the model are explained in detail, making this research reproducible. This way, the model can still be used for different types of hardware by simply changing some of the measured values.

Chapter 3

Methodology

In this chapter we explain the changes made to the OpenWSN firmware and the setup used to perform the measurements.

Two kind of measurements have to be performed to find the energy consumption during a time slot:

- For each device state (combination of cpu and radio state) we need to measure the energy consumption of the mote (Section 3.2).
- We also need to measure the duration of each of these device states in the time slot (Section 3.3).

3.1 Firmware changes

Setting an output pin high or low changes the power consumption so we decided to disable the code that toggled debug pins and leds while doing the energy measurements. We also disabled the serial communication because even when the OpenUSB is not connected to a computer, the code would still try to output data which caused an increase in power consumption.

Other than these small adaptions we did not need to change anything to use the 2.4 GHz radio. The sub-1GHz radio on the other hand required a lot

more work as there were no working drivers yet for the CC1200 radio chip on the OpenUSB.

3.1.1 CC1200 driver development

Two drivers for the CC1200 already existed:

- The first implementation [24] was created for the CC1200 Development Kit (CC1200DK) hardware, it however provided little code that could be reused. Other than the command strobes that are sent to the CC1200 chip to change the radio state, the code is platform specific and irrelevant for the OpenUSB hardware. Although all functions were implemented and this code was merged into the OpenWSN development branch, the code could not be compiled yet.
- The second implementation [25] was written for the OpenUSB platform, but it was never finished. We decided to use this driver implementation as a starting point because a large part of the needed code was already there. The existing code however contained several bugs which we also had to fix. These issues ranged from small typos and stub functions[1] that caused the code to hang, to parts that we rewrote entirely.

Handling interrupts

One part we had to implement was handling the radio interrupts. There are two interrupts from the radio that we need to handle, one at the start and one at the end of a frame that is being received or transmitted. We configured the radio such that one of the GPIO pins would become high at the start of a frame and would become low again at the end of a frame. The exact moments of these interrupts is shown in Figure 3.1. The packet in the IEEE 802.15.4 frame structure will first transmit a 32 bits preamble sequence followed by 8

[1] A stub function is a method that exists but is not yet implemented. It can be an empty function, return a fixed value or perform any other temporary operation that substitutes the yet-to-be-developed code.

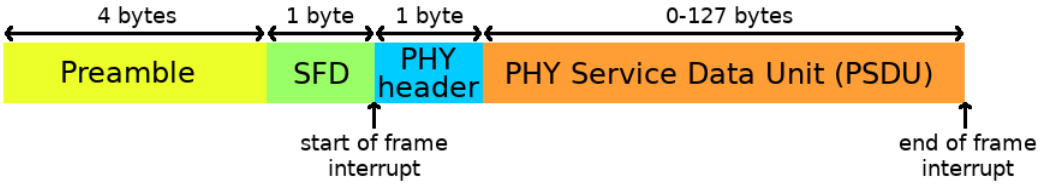

Figure 3.1: *Interrupt timing in IEEE 802.15.4 frame*

bits Start of Frame Delimiter (SFD). Once the SFD is transmitted, the start of frame interrupt is triggered. The 8 bits physical header (containing the size of the frame) and the frame itself are then transmitted. Once all bytes have left the radio, the end of frame interrupt is triggered.

Getting the timing right

Timing is very important because one mote has to receive at the exact moment the other mote is transmitting. Even with a large guard time, if the transmitter starts too late or too early, the receiver may still get the packet but incorrectly believe that it is out of sync. Some timing related constants thus have to be updated when using a different radio.

One constant that has to be updated is the time it takes to go from IDLE to TX mode, as this can be different in each radio. We need this duration so that the "go" signal can be given at the right time in order to transmit the first byte exactly when expected by the receiver.

Other constants that needed to be reviewed are the maximum time to copy from and to the radio. While using the CC2538 just requires the bytes to be written to a register in memory, the bytes have to be send over SPI when using the CC1200, which is significantly slower.

A decision also had to be made about the data rate. Using the 50 kbps found in both the existing code and the code for the CC1200DK platform would mean that the slot length cannot remain 15 ms. Transmitting a packet of 127 bytes would already take longer than the entire slot length. Many more timing constants would thus have to be adapted. We opted for the alternative and boosted the CC1200 data rate to 250 kbps by changing the radio register settings. This is the same data rate used by the CC2538 radio and it thus

Property	Value
Modulation format	2-FSK
Symbol rate	250 ksps
Deviation	124.816895 kHz
RX filter BW	833.333333 kHz

Table 3.1: *CC1200 radio configuration used to achieve a bit rate of 250 kbps*

State	Consumption	Condition
IDLE	1.5 mA	Clock running, system waiting with no radio activity
XOFF	180 μA	Crystal oscillator disabled
SLEEP	0.12 / 0.5 μA	Low-power RC oscillator off / running

Table 3.2: *CC1200 power modes when not receiving or transmitting*

takes the same amount of time to transmit the frame. The settings changed on the radio are shown in Table 3.1.

Trying to sleep

There are three different modes for when the CC1200 is inactive: IDLE, XOFF and SLEEP (listed in decreasing order of energy consumption). The CC1200 automatically goes to IDLE state when not transmitting or receiving. Going to the XOFF state turns off the crystal oscillator. The SLEEP state is the lowest power mode possible but will not retain all register values. Table 3.2 shows the theoretical energy consumption of these states [26].

We encountered several issues when attempting to put the CC1200 in SLEEP mode to save power.

When a packet is received, the OpenWSN firmware first turned off the radio before trying to read the packet from the radio. Since the CC1200 flushes the RX FIFO buffer when going to SLEEP mode, reading the packet failed. With the CC2538 radio this is not a problem because it only has a single OFF mode which does not flush the RX buffer. However, for using the CC1200 radio it is necessary to turn the radio off after reading the packet instead of before.

The code to save energy taken from the CC1200DK platform did not work properly. The authors attempted to first put the radio into XOFF mode to disable the crystal oscillator and afterwards put the radio into SLEEP mode. Due to the way the command strobes are passed to the radio, the device however ended up in IDLE mode instead. The radio should be put in SLEEP mode directly, which also disables its crystal oscillator.

Putting the radio into SLEEP mode increased the energy consumption by almost 10 mA in our measurements. We therefore decided to use the XOFF mode instead to save power.

3.2 Measuring energy consumption

We used two setups to measure the energy consumption. The first method was only used for the CC2538 radio and for a limited amount of device states (we had minimized the amount of device states by assuming the cpu would always be sleeping while the radio is active). Once we found that the first setup could not fully measure the energy consumption of the CC1200 radio, we decided to perform all measurements in a different way.

3.2.1 Gecko setup

We started by doing measurements using the EFM32GG-STK3700 Giant Gecko Starter Kit from Silicon Labs. The Gecko contains an Advanced Energy Monitor (AEM) which allows tracking its current consumption. It is capable of measuring current in the range of 0.1 μA to 50 mA with an accuracy of 0.1 mA for currents above 250 μA and an accuracy of 1 μA for currents below 250 μA.

The Gecko was connected to the OpenUSB by connecting the VMCU/VCC and GND pins as displayed in Figure 3.2. With this setup, the OpenUSB is powered through the Gecko instead of via a USB. The Gecko is connected by USB to a computer where the Energy Profiler in Simplicity Studio displays the energy consumption.

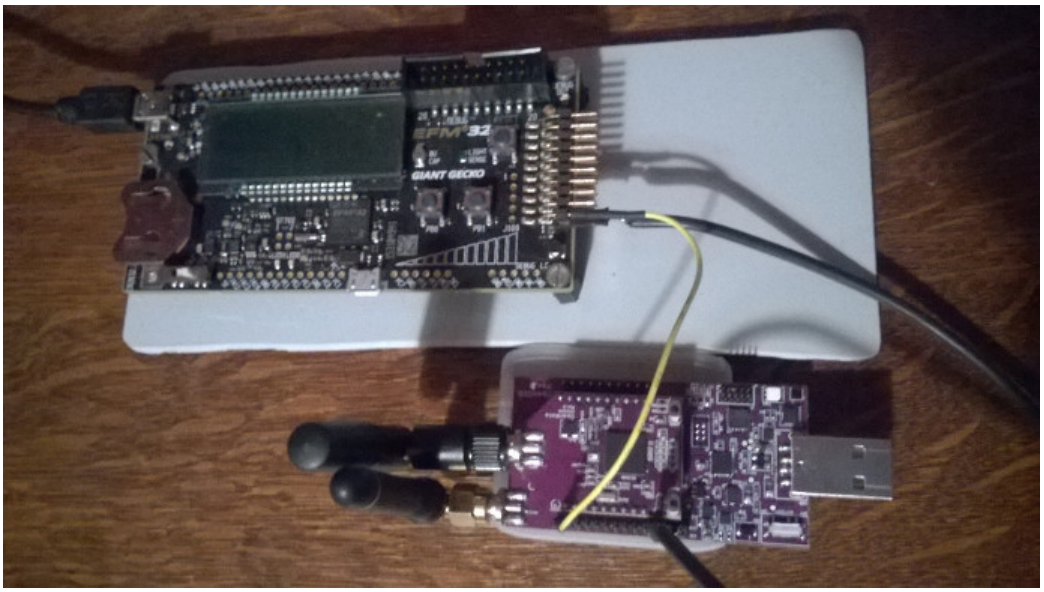

Figure 3.2: *Giant Gecko connected to the OpenUSB*

Since the AEM measures the entire energy consumption including the consumption of the Gecko, we needed to flash a program on the Gecko that consumed minimal power. The flashed program put the Gecko into EM4[2], the lowest possible energy mode, which has a theoretical consumption of only 20 nA. Since the AEM measures in steps of 0.1 μA, the consumption of the Gecko no longer affected the measured consumption.

A downside of using the Gecko to measure the energy consumption is that the AEM is limited to 6250 current samples per second. This means that there can only be one sample every 160 μs or 93.75 samples per 15 ms time slot. This is enough to estimate the energy consumption but peaks in the consumption may not be fully visible.

Once we started performing measurements while using the CC1200 radio, we found that it could not be measured with the Gecko as the consumption exceeded 50 mA. We needed a different setup to continue.

[2]The program actually waited 3 seconds before going into EM4. Putting the Gecko in EM4 at the very beginning of the program can lock up the device and prevent it from being flashed again using the normal method.

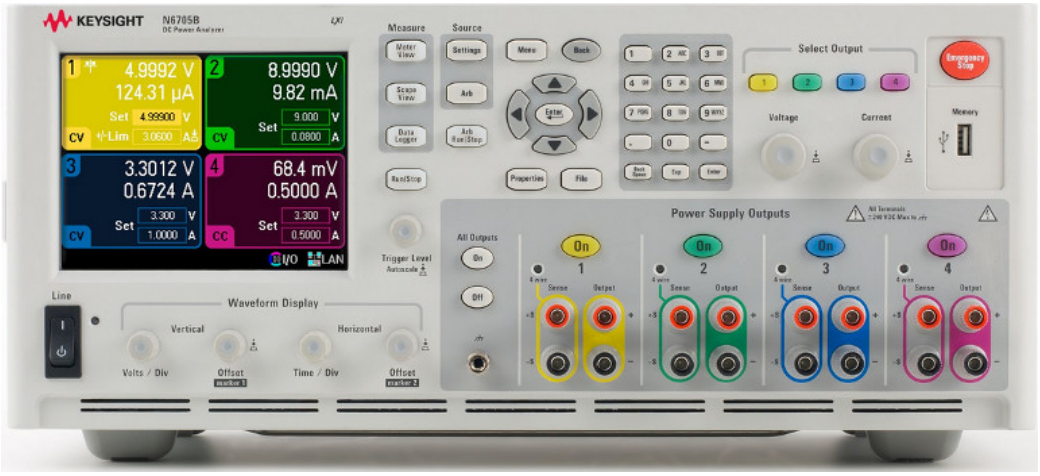

Figure 3.3: *N6705B DC Power Analyzer*

3.2.2 Actual setup

To perform the measurements we used a N6705B DC Power Analyzer, shown in Figure 3.3. The OpenUSB was connected on its VCC and GND pins to one of the power supply outputs of the N6705B, which was configured to provide an input voltage of 3.3 V.

Although this device is more accurate than the Giant Gecko, the main reason to use it was because the consumption of the CC1200 radio would fall within the measurable range. Although a better accuracy of the measurement tool increases the accuracy at which predictions about the battery lifetime can be made, we believe that for this thesis the exact numbers are less relevant. The measured numbers were used to experimentally verify how good the model is, but will have no use when the model is going to be used for different hardware. We therefore did not set up the 4-wire sensing on the N6705B which can provide ammeter accuracy up to 0.025% [27], but instead used the 2-wire mode without trying to achieve an excellent accuracy.

Where possible we used the average consumption over a period of around 400 ms. Some states like RX and TX however only last for as long as the radio takes to send all the bytes. Since packets of 127 bytes (the maximum packet size) were being used, the average was taken over only 3-4 ms for these states.

Using this setup, we redid the measurements with the CC2538 and also performed extra measurements. We no longer assumed a relationship between the CPU and radio states.

3.2.3 Code

The code below was used to put the mote in the right device state so that we could measure the energy consumption of each state. Because the code is slightly different for each state, not all lines of code have to be in the program at the same time. Code that is not always present is placed in comments and was uncommented in the situations where they were needed.

```
#include "radio.h"
#include "bsp_timer.h"
#include <source/sys_ctrl.h>
#ifndef USING_CC1200
  #include "cc2538rf.h"
#endif

void endFrame() {
  radio_rfOff();
}

int mote_main() {
  board_init();
  radio_setEndFrameCb(endFrame);

  uint8_t packet[127];
  radio_loadPacket(packet, 127);

  // Set the TX power of the radio
#ifdef USING_CC1200
  // 0 dBm:
  //        cc1200_set_tx_power(0);
  // 14 dBm:
```

```
  //          cc1200_set_tx_power(14);
#else
  // 0 dBm:
  //          HWREG(RFCORE_XREG_TXPOWER) = 0xBC;
  // 3 dBm:
  //          HWREG(RFCORE_XREG_TXPOWER) = 0xD5;
#endif

  // Set the state of the radio
  // Listen/Rx:
  //          radio_rxEnable();
  //          radio_rxNow();
  // Tx:
  //          radio_txEnable();
  //          radio_txNow();
#ifdef USING_CC1200
  // Idle:
  //          cc1200_idle();
  // Sleep:
  //          radio_rfOff();
#else
  // Off:
  //          radio_rfOff();
#endif

  while (true)
  {
    // Set the state of the CPU
    // Active:
    //     /* empty while loop */
    // Sleep:
    //     SysCtrlPowerModeSet(SYS_CTRL_PM_NOACTION);
    //     SysCtrlSleep();
    // DeepSleep:
```

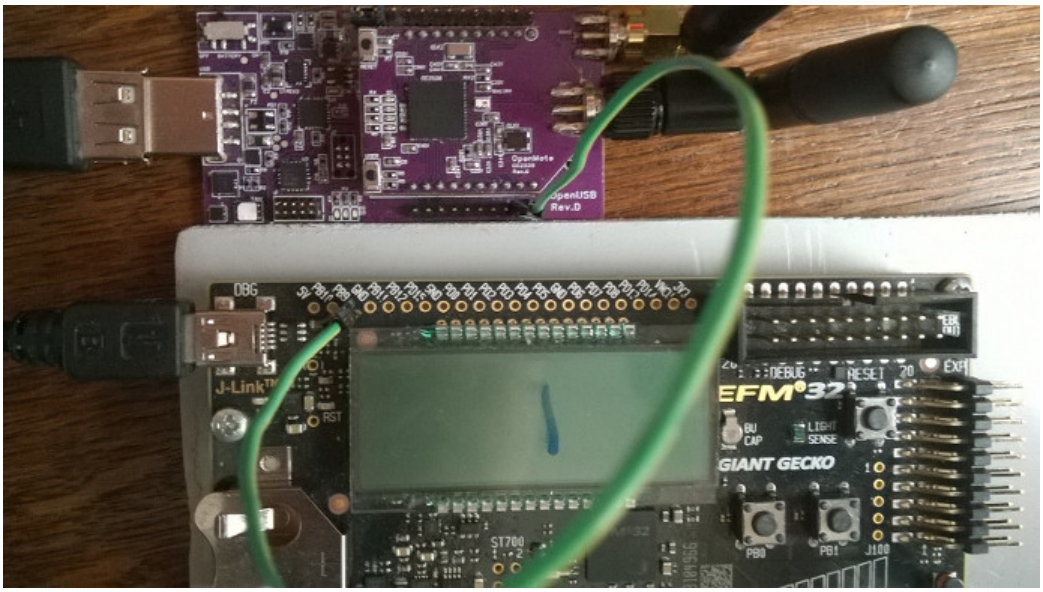

Figure 3.4: *Giant Gecko connected to the OpenUSB to measure durations*

```
        //      SysCtrlPowerModeSet(SYS_CTRL_PM_2);
        //      SysCtrlDeepSleep();
    }
}
```

3.3 Measuring state durations

For the timing measurements we again used the Giant Gecko. There exists an example for the Gecko that shows on the display how many seconds the PB0 button has been pressed. This has been our starting point.

Since the PB0 button is mapped to pin PB9, one of the OpenUSB pins has been connected to pin PB9 on the Gecko. The program could then measure how long this pin was made low. On the OpenUSB pin PD2 was used because this debug pin is not in use by the CC1200 chip. The connection between the OpenUSB and Gecko is shown in Figure 3.4.

Instead of printing the time on the display, the durations were sent through the Serial Wire Output (SWO) interface. The output was then printed in the console in Simplicity Studio on the connected computer. This allowed post-

processing of the durations such as calculating the average and deviations. We also changed the accuracy of the reported durations from milliseconds to microseconds.

3.3.1 Accuracy

To get this accuracy we used a timer with a frequency of 14 MHz. Although this should result in an accuracy around 0.1 μs (since there are 14 ticks per μs), all values have been rounded to the nearest μs. Testing however showed that there is some variance that sometimes makes the result less accurate than 1 μs. For short measurements (e.g., 50 μs), the measured durations varies less than 1 μs. For longer durations the error is larger as the variation scales linearly with the duration. When the duration should be 125 ms then the deviation is already a bit larger than 200 μs.

However, we still believe that the tool used for the measurement was accurate enough. We did not make long measurements. Each slot is only 15 ms long and we only need to measure individual parts of this slot which are all less than 4 ms. Although the variations seen here are still 8 μs between the lowest and highest value, the testing also showed that the results are equally spread. The average always lies at the center of the interval and by doing enough measurements we are confident that the final result is still accurate up to 1 μs.

3.3.2 Code

The code running on the OpenMote-CC2538 simply has to make the PD2 pin low at the start of the measurement and high at the end. The pin is also made high directly before the start of the measurement. This is done to avoid wrong measurements where the code path does not reach the end each time it passes the start. The durations in these cases will be much higher than the expected value (e.g., 15 ms when the code is executed every slot), so these high values are filtered out on the computer.

The code below is what was placed around the part of the project for which the duration was to be measured.

```
// End previous measurement if not stopped yet
gpio_on(BSP_PIND_PORT, BSP_PIND_2);

// Start a new measurement
gpio_off(BSP_PIND_PORT, BSP_PIND_2);

// Execute code of which we want to know the duration

// End the measurement
gpio_on(BSP_PIND_PORT, BSP_PIND_2);
```

Chapter 4

Model

In this chapter we introduce the model by showing the different states in each time slot type and introduce the formula to calculate the consumption of each slot.

4.1 Time slots

Figure 4.1 presents a general overview of the activity of a transmitter mote during a TxDataRxAck time slot and a receiver mote during a RxDataTxAck time slot.

Our model divides each time slot into different states. Some of the states seen in Figure 4.1 consist of a part where the CPU is active and a part where the CPU is sleeping, which is seen as two different states in our model. The state of the radio in our model only changes at moments when the CPU

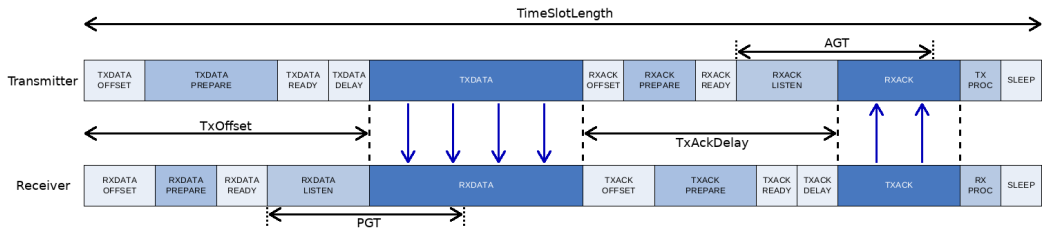

Figure 4.1: *General states in TxDataRxAck and RxDataTxAck time slots*

State in slot	CPU State	Radio State
TxDataOffsetStart	Active	Sleep
TxDataOffset	Sleep	Sleep
TxDataPrepare	Active	Idle
TxDataReady	Sleep	Idle
TxDataDelayStart	Active	Idle
TxDataDelay	Sleep	Tx
TxDataStart	Active	Tx
TxData	Sleep	Tx
RxAckOffsetStart	Active	Sleep
RxAckOffset	Sleep	Sleep
RxAckPrepare	Active	Idle
RxAckReady	Sleep	Idle
RxAckListenStart	Active	Idle
RxAckListen	Sleep	Listen
RxAckStart	Active	Rx
RxAck	Sleep	Rx
TxProc	Active	Idle
Sleep	Sleep	Sleep

Table 4.1: *States in a TxDataRxAck slot*

state changes. This is of course a simplification as in the real world the radio state changes a little before or after this moment, typically while the CPU is active.

We only discuss the TxDataRxAck time slot in full detail. The other slots are similar and we only talk about the differences with the TxDataRxAck slot. Tables containing all states in each time slot type are provided in Appendix A.

4.1.1 TxDataRxAck

Table 4.1 and Figure 4.2 illustrate the different states in a TxDataRxAck slot and what the CPU and radio states are during each moment. In the figure,

Figure 4.2: *States in a TxDataRxAck slot*

the CPU has two states (Sleep and Active) while the radio has three states (Sleep, Idle and Active). The radio being active refers to it either being in Listen, Rx or Tx mode.

At the beginning of each time slot, the CPU wakes up and performs the tasks required for any slot. This includes incrementing the ASN and scheduling the next state depending on the type of the slot. The CPU then sleeps again during TxDataOffset until the moment the radio is needed.

During TxDataPrepare, the radio wakes up, the channel is set and the bytes to transmit are loaded into the radio. The duration of this state is variable mainly because the time to load the bytes depend on the frame size. Since this state always starts at the same offset and has a variable duration, there is some time left between the TxDataPrepare and the actual transmission. During this TxDataReady state, the radio is in Idle mode while waiting until it is time to transmit. To minimize the energy consumption of the mote, the duration of the TxDataReady state should thus be as small as possible.

The first byte behind the SFD has to be transmitted exactly $TxOffset$ ms after the start of the time slot. In order to do so, the time required to switch the radio from Idle to Tx mode has to be taken into account. The duration of the TxDataDelay equals the time between the Tx command being sent to the radio and the moment the SFD has been transmitted.

After the RxAckOffset that follows where the mote sleeps, the RxAckPrepare state then prepares the radio again by waking it up and setting the correct channel. Any time less than the maximum duration of RxAckPrepare is then spent in the RxAckReady state.

The ACK is transmitted exactly $TxAckDelay$ ms after the end of the TxData state. Because the clocks of the transmitting and receiving mote may not be perfectly in sync, the ACK might arrive slightly earlier or later than expected. The radio is thus turned on at the start of the RxAckListen instead

Figure 4.3: *States in a RxDataTxAck slot*

of just in time for the data. If no ACK is received during the Acknowledgment Guard Time (AGT) period, the mote turns off the radio and considers the transmission failed. The duration of the AGT is defined as 1000 μs in OpenWSN. When the clocks between the motes are perfectly in sync, the RxAckListen state has a duration of $AGT/2$ plus the time to change the radio from Idle mode to Rx mode (which is considered to be instantaneous in OpenWSN).

During the TxProc state, the ACK is read from the radio and the transmission is considered successfully when the ACK is valid. The mote also synchronizes its clock based on the offset between $TxAckDelay$ and the actual data reception time, if the ACK came from its parent in the network graph. For the remaining part of the time slot, both the CPU and radio are in Sleep mode.

4.1.2 RxDataTxAck

Figure 4.3 illustrates the different states in a RxDataTxAck slot. This time slot can be considered the opposite of the TxDataRxAck. The states to handle the data in TxDataRxAck are found in handling the ACK in RxDataTxAck and vice versa.

The guard time for the data is however larger than the AGT that is used for ACKs. The Packet Guard Time (PGT) determines how long the radio listens for the data before the radio is turned off. When no data is received during the PGT period, we classify the time slot as RxIdle instead of RxDataTxAck. In OpenWSN, the PGT is defined as 2600 μs.

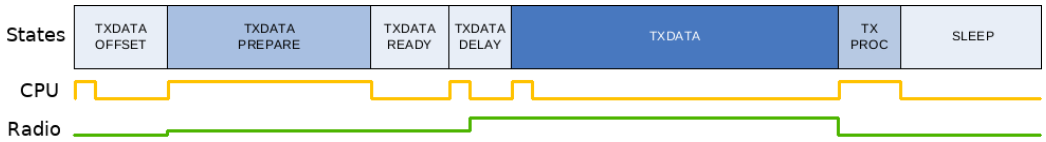

Figure 4.4: *States in a TxData slot*

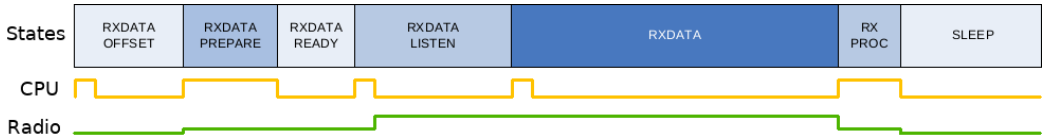

Figure 4.5: *States in a RxData slot*

4.1.3 TxData and RxData

When no ACKs are required (e.g., for broadcasts), only the first half of the time slot is used. During the TxData and RxData slots, the mote sleeps once the data has been transmitted or received. The states in the TxData and RxData slots are shown in Figure 4.4 and Figure 4.5.

4.1.4 RxIdle

When the transmitter has no data to send, the slot that could have been a TxDataRxAck becomes a Sleep slot. But on the receiver side a different type of slot is needed to represent the behavior of the mote. The RxIdle slot occurs when the receiver expects data but does not receive anything. This behavior is not an error, it simply means that a slot was reserved but the transmitter did not have any data to send at that moment. Figure 4.6 shows the states in an RxIdle slot.

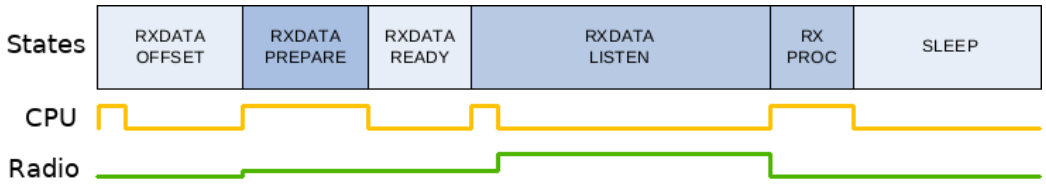

Figure 4.6: *States in a RxIdle slot*

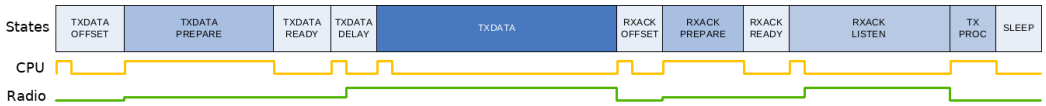

Figure 4.7: *States in a TxDataRxAckMissing slot*

4.1.5 Sleep

In time slots where no data has to be transmitted or received, the mote can sleep during the whole slot. The mote only briefly wakes up at the start of the slot to e.g., increment the ASN.

4.1.6 TxDataRxAckMissing

There are many error states in OpenWSN. The code would get in such error state when e.g., the radio remains active too long or when the prepare state lasts longer than the maximum allowed time. It is unlikely that the code would end up in most of these error states unless when there is a configuration issue. There is however one error state which is likely to occur eventually: a missing ACK. In the TxDataRxAck slot, data is transmitted and an ACK is received. In the slot that we refer to as TxDataRxAckMissing, the ACK is expected but not received. The different states during this TxDataRxAckMissing slot are shown in Figure 4.7.

4.2 Building the model

With all states per time slot identified and with the consumptions and durations during each state measured, the model can be built.

The consumption of a time slot is given by the following formula. The resulting consumption is expressed as the charge drawn from the battery, in Coulombs.

$$SlotConsumption = \sum_{State \in Slot} duration(State) * current(State)$$

If we apply this for all slot types and shorten the notation of the terms then we find the formula below which calculates the charge drawn for each different time slot.

$$\forall Slot \in SlotTypes : Q_{Slot} = \sum_{State \in Slot} \Delta t_{State} * I_{State}$$

The unit of the duration is ms while the unit of the current is mA. This means that the unit of the resulting charge is μC.

The model with consumptions of each slot type could be used in a simulation where the consumption is increased each slot as the simulation runs. To calculate the consumption over time, having a model of the entire slot frame is more practical. We did not build such a model ourselves but instead relied on the existing model described in the "Slot-Frame Energy Consumption Modeling" from the "A Realistic Energy Consumption Model for TSCH Networks" paper [23]. The charge drawn per slot in their calculations can be replaced by the measurements performed in this thesis and the formula above.

Our slot model contains the information for any packet size. Formulas containing $(NBSent/MaxPktSz)*Q_{slot}$ in the slot-frame model (where NBSent is the amount of bytes sent) could be replaced by just Q_{slot_NBSent} where Q_{slot_NBSent} is the slot consumption calculated using the number of bytes sent. This is unlike Q_{slot}, which was calculated with packets of maximum size in that work.

4.3 Support for different hardware

Since the model has so many inputs, in order to use it for different hardware, many numbers might have to be adapted. This is necessary if a highly accurate simulation is needed with such hardware. The model can however be easily simplified to be used with different hardware more quickly, at the cost of some accuracy. By setting the duration of short states to 0, the model becomes easier to change as only the states that have the most impact on the consumption will have to be updated. Alternatively the duration can

be estimated instead of measured as most durations are very similar to the ones we measured with the OpenMote hardware. The consumption of CPU and radio also do not have to be measured, the values could be taken from the datasheet. The result is a slightly less accurate model, but no or few measurements have to be made to be able to use the model to simulate any hardware.

Chapter 5

Results

In this chapter we discuss the results of our measurements and experimentally verify the accuracy of the model.

5.1 State durations

Using the setup explained in Section 3.3, we measured the duration of each state in every time slot where the CPU is active. The durations in which the CPU is sleeping could then be trivially calculated. All measured and calculated values can be found the tables in Appendix B.

States do not always have the exact same duration for a variety of reasons. There could be multiple code branches (different execution paths), the packet size could influence it or the duration of an operation can simply be variable (e.g., waking up the CC1200 chip). Multiple measurements had to be made to find a single duration that could be associated with the state.

Changing the CC1200 mode from Sleep to Idle takes between 246 and 343 μs, which causes every state where the radio wakes up to have a variable duration. It only required a few measurements to find that the median for waking up is 268 μs. However, we decided to use the average value instead of the median because it would result in slightly more accurate energy consumption

prediction. To avoid being susceptible to outliers, we measured the wakeup time over ten thousand times and found an average of 273 μs.

For states with multiple code branches, the average duration was also taken. Since these are states where the radio is not active, these small variations on the duration only have a small impact on the total slot consumption.

To measure states where packets are loaded to and read from the radio, we measured the duration before and after the radio is accessed. We then separately measured the communication with the radio for different packet sizes (0 to 125 bytes with steps of 25 bytes). We performed linear interpolation on the measured durations to come up with a formula that works for all packet sizes.

The duration of transmitting and receiving also depends on the packet size. Since the radio has a baud rate of 250 kbps, the time to transmit one bit is 4 μs, which makes the time to transmit a byte 32 μs. We thus simply have to multiple the amount of transmitted bytes with 32 μs to find the duration. The PHY header byte and 2 bytes CRC also have to be included as they are sent with the packet. We measured the time between the start of frame and end of frame interrupts to verify that we can use this calculation and found that on average the error was only 0.13% with our measurements.

To model the guard time we assumed that the clocks are in sync. The packet thus always arrives exactly in the center of guard interval in our model.

5.2 Device state consumption

Using the setup mentioned in Section 3.2.2, we measured the consumption of the OpenMote-CC2538 while connected to the OpenUSB during different device states. Since the CPU and radio are the two components responsible for the majority of the energy consumption, these device states are all combinations between CPU and radio modes. Instead of measuring the consumption of the CPU and radio separately, we measured the consumption of the entire device. The result is that any energy consumption not related to the CPU or radio (e.g., SPI or timers) are measured as part of the CPU

CPU state	Radio state	Consumption (in mA)
Active	Off	18.5253
Active	Listen	36.0883
Active	Rx	32.1613
Active	Tx	36.1228
Sleep	Off	12.1690
Sleep	Listen	29.6143
Sleep	Rx	25.5274
Sleep	Tx	29.6779

Table 5.1: *Consumption of different device states when using the CC2538 radio*

usage. This allows a slightly more accurate energy prediction compared to models that ignore the other components.

5.2.1 2.4 GHz radio

Table 5.1 shows the consumption of different device states when the CC2538 radio is being used. The values for the Tx state were measured when the transmit power of the radio was set to 0 dBm. When the transmit power is set to 3 dBm (the current default in OpenWSN), the consumption of the Tx states becomes 37.9312 and 31.4720 mA (for the CPU in Active and Sleep state respectively).

Unlike shown earlier in Table 4.1, there are no Sleep and Idle radio states here, only an Off state. This is because the CC2538 consists of both the CPU and radio and the radio does not have a separate Sleep state. In this case, our model which works with Sleep and Idle states will consider the consumptions of both states to equal the consumption of the Off state.

As expected, the difference in consumption between the CPU being active or sleeping is almost identical for all radio states. On average the CPU consumes 6.4737 mA more when active.

The CPU usage while sleeping is very high. This is caused because the OpenMote-CC2538 code in OpenWSN currently uses the lowest possible sleep

CPU state	Radio state	Consumption (in mA)
Active	Sleep	18.5977
Active	Idle	21.0067
Active	Listen	43.3729
Active	Rx	57.3220
Active	Tx	59.3448
Sleep	Sleep	12.4005
Sleep	Idle	15.0322
Sleep	Listen	38.2895
Sleep	Rx	50.7769
Sleep	Tx	53.6732

Table 5.2: *Consumption of different device states when using the CC1200 radio*

mode. The consumption dropped to 1.6420 mA when entering deep sleep (SYS_CTRL_PM_2 specifically). This is still high because the OpenUSB is still consuming power. Since the OpenWSN version that uses the CC2538 does not access the CC1200 at all, the CC1200 chip is still in Idle mode which is where this extra consumption is coming from. When the OpenMote-CC2538 is not connected to the OpenUSB (which contained this CC1200 chip), the measured consumption during deep sleep drops to only 0.0317 mA. With future updates to OpenWSN it is thus expected that the consumption of the CPU in Sleep state will be more comparable to that of different hardware.

5.2.2 sub-1GHz radio

Table 5.2 shows the consumption of different device states when the CC1200 radio is being used. The values for the Tx state were measured when the transmit power of the radio was set to 0 dBm. When the transmit power is set to 14 dBm (the current default in OpenWSN), the consumption of the Tx states become 102.7338 and 96.6123 mA (for the CPU in Active and Sleep state respectively).

The difference between the CPU in Active and Sleep mode vary more in these measurements compared to when using the CC2538 radio. The difference is

on average 5.89436 mA in the measurements with the CC1200 radio instead of being around 6.4737 mA as expected. One reason is that the CC1200 consumption was not always the same every time we measured the consumption, leading to larger errors between measurements. Another reason is that due to the SPI peripheral and certain pins that have to be kept high or low, the difference between the CPU being in Active and Sleep mode can simply be different when using the CC2538 or CC1200 radio.

When both the CPU and Radio are in Sleep state, the consumption is still high because the CPU still has a high energy consumption. If the CPU is put in deep sleep, the consumption drops to 0.7611 mA. The consumption could still be lower if the radio went into a deeper sleep mode as well (SLEEP instead of XOFF).

5.3 Slot consumption

With the duration and consumption of each state we could calculate the consumption for each type of slot. We also measured the consumption of entire slots to experimentally verify the correctness of our calculated values in our model. Table 5.3 shows the values we measured and calculated for each of the slot types for both radios. We configured both radios to have a transmit power of 0 dBm and sent packets of 127 bytes (the maximum packet size when including the CRC bytes).

As seen in Table 5.3, the difference between the measured and calculated values is relatively small. The main contributors to these differences are small measurement errors and the variations in guard time duration. In the measured data, the guard time is a little smaller or larger than in the calculated data, which assumes perfectly synchronized clocks. On average the difference is only 1.3 μC or 0.457%, which shows that our model provides a good representation for real-life scenarios.

Figure 5.1 shows the current during a TxDataRxAck time slot according to both the model and the measurements. Comparisons for all time slot types can be found in Appendix C. The peaks on the graphs do not perfectly

Slot type	Measured		Calculated	
	CC2538	**CC1200**	**CC2538**	**CC1200**
TxDataRxAck	283.34	446.72	284.60	445.17
RxDataTxAck	287.41	458.68	286.22	457.78
TxData	262.07	386.76	262.78	388.01
RxData	265.39	399.98	263.09	397.01
RxIdle	229.61	260.97	229.33	261.15
Sleep	184.19	183.63	182.90	186.36
TxDataRxAckMissing	280.06	417.35	279.89	418.85

Table 5.3: *Measured and calculated consumption for each slot type, in μC*

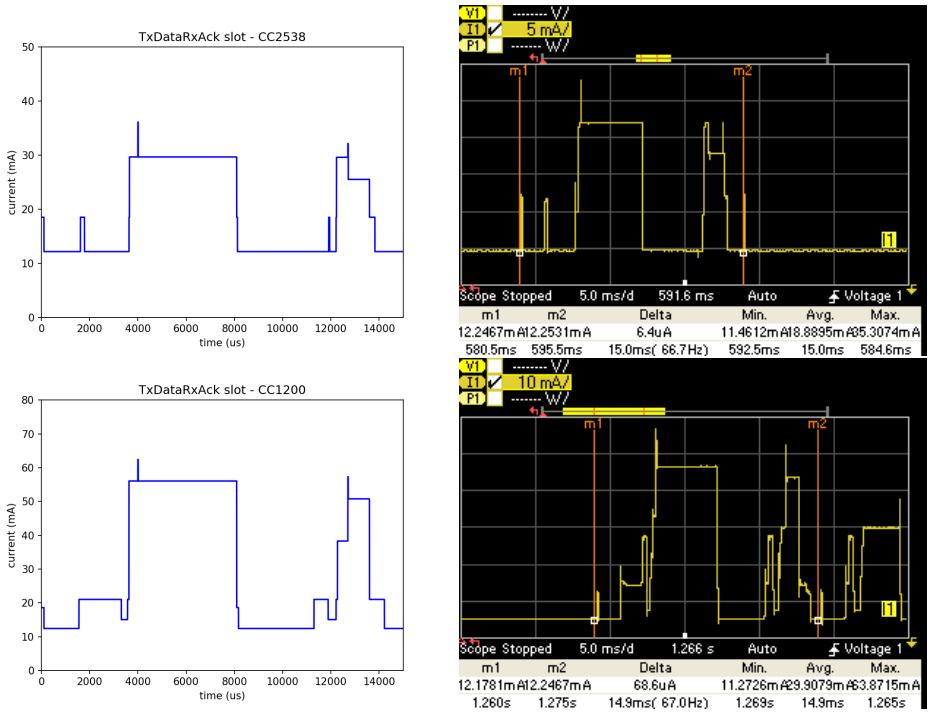

Figure 5.1: *Comparison between calculated (left) and measured (right) TxDataRxAck time slot when using the CC2538 (top) and CC1200 (bottom) radios*

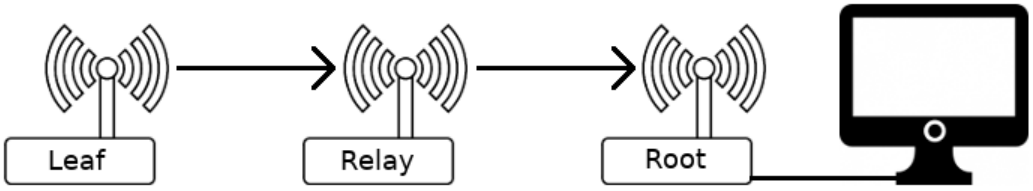

Figure 5.2: *Topology used while comparing the consumption of a slot frame*

match, because the model simplified certain states. The radio state may be changed while the CPU is active, causing the CPU and radio to be active at the same time, while the model might only consider the radio as active once the CPU goes to sleep. This results in a peak in the measured time slot where there is no peak in the model. There is only one state that was not modelled at all. In the images of the CC1200 radio there are peaks at the end of the TxDataPrepare and RxAckPrepare states. These are caused because the radio is manually put in Calibration mode. Something like this does not explicitly happen with each radio and is subject to change (e.g., the CPU does not always have to remain active during this state), so we decided not to model it and assume the radio is still in Idle mode.

5.4 Slot frame consumption

We also compared the consumption calculated by our model with measurements for entire slot frames.

We set up a network of three motes with a fixed topology, as shown in figure 5.2. The leaf mote was configured to send one packet of 127 bytes (including CRC) every 2 seconds. Slot frames consisted of 51 time slots of which one was configured for data from the leaf to the relay mote and one for data from the relay to the root mote. The first time slot in the slot frame is reserved for advertisements and used to send Enhanced Beacon frames. We did not look at the broadcasted beacons and DAO packets that were send every 30 and 60 seconds, we only focussed on the packets coming from the leaf mote. The first time slot in each slot frame is thus considered to be of type RxIdle.

Since beacons and similar packets were ignored, the consumption values may

not offer a good prediction for how long the battery of the mote will last. The goal of the comparison of slot frame consumption was however to verify how accurate the model is by calculating and measuring similar circumstances. It is thus not important that the scenario with only 3 motes is less likely and that the activity in a slot frame is simplified.

The slot frame of the leaf mote always consists of 1 RxIdle slot and at least 49 Sleep slots. The final slot will be either TxDataRxAck when there is data to send or another Sleep slot. Since there are 51 slots in a slot frame and every time slot lasts 15 ms, the duration of each slot frame is 765 ms. Since a packet is send every 2 seconds, the consumption of the leaf slot frame can be considered as follows:

$$Q_{leaf} = Q_{RxIdle} + 49 * Q_{Sleep} + (0.3825 * Q_{TxDataRxAck} + 0.6175 * Q_{Sleep})$$

The slot frame of the relay mote can be determined in a similar way. There are RxIdle and Sleep slots when no packet is received, while there are RxDataTxAck and TxDataRxAck slots when a packet was received and forwarded. The first time slot is also of type RxIdle and the remaining 48 slots are always Sleep slots. The formula for the consumption of the slot frame of the relay mote thus becomes:

$$Q_{relay} = Q_{RxIdle} + 48 * Q_{Sleep} + Q_{other}$$
$$Q_{other} = (0.3825 * (Q_{RxDataTxAck} + Q_{TxDataRxAck}) + 0.6175 * (Q_{RxIdle} + Q_{Sleep}))$$

The root mote only has a single RX slot. Its consumption can thus be represented by the following formula:

$$Q_{root} = Q_{RxIdle} + 49 * Q_{Sleep} + (0.3825 * Q_{RxDataTxAck} + 0.6175 * Q_{RxIdle})$$

We will however not focus on the consumption of the root mote. It is connected to the computer and serves as a gateway to the internet, therefore the mote typically does not run on batteries. Serial communication cannot be disabled on this mote and the measured consumption would therefore not fully correspond to our model.

We measured the consumption of the slot frames for the leaf and relay mote and compared them with the values calculated through the model using the

Mote type	**Measured**		**Calculated**	
	CC2538	**CC1200**	**CC2538**	**CC1200**
Leaf	9499.80	9580.50	9413.23	9678.14
Relay	9543.75	9742.71	9481.42	9828.15

Table 5.4: *Measured and calculated slot frame consumption, in μC*

above formulas. The result can be found in Table 5.4. On average the error between the calculated and measured values are below 1% as expected. An error of around 66.3 μC is normal based on our average error of 1.3 μC that would occur in each of the 51 time slots. The error seems to be slightly higher here however, mainly because the majority of time slots were Sleep slots where the error per time slot was above average.

The comparison again shows that our model is quite good. For example, the difference between the measured and calculated consumptions for the Leaf mote for the CC2538 radio is only 0.91%. However, we have to be careful with using these numbers to show how accurate the model is exactly. Since the difference is small, the measurement errors will become relatively large. A measurement error of only 0.1% on both the measured slot frame consumption and the consumptions on which the calculated values are based, could already cause the difference to be 0.71% instead of 0.91%.

Chapter 6

Conclusion

We have proposed a new energy model for time slots in OpenWSN that takes all network-related CPU state changes into account. We have experimentally verified that when inputting the durations and consumptions of the OpenMote hardware, the calculated energy consumption by the model is very close to the (measured) consumption of this hardware. In our setup the error between the calculated and measured energy consumption was less than 1%. This makes our model suitable for use in simulations. The ability to calculate the consumption based on the packet size also improves upon existing models that only provide the consumption of an entire time slot with a packet of maximum size.

We also contributed a driver for the CC1200 radio chip on the OpenUSB hardware to the OpenWSN firmware project, which allows experimenting on the sub-1GHz band.

Our model provides accurate energy consumption predictions and is therefore suitable for 6TiSCH simulations with OpenWSN.

6.1 Future work

Experimentally verifying the accuracy of our model over a long duration is limited because the consumption while sleeping is so high. This is some-

thing that should be looked at. Once the OpenMote-CC2538 platform code is updated to allow deep sleep, the consumption for different states in the model can be measured again. Comparisons can then be made between measuring the consumption over a certain amount of time and calculating the consumption with the model over the same duration.

The consumption of the CC1200 chip could also be reduced in the future by going into SLEEP mode instead of XOFF mode. The CPU can also be put in Sleep mode in almost all cases during radio calibration. These changes will reduce the energy consumption of the OpenUSB so that it becomes more useful to use this hardware.

Ultimately, the goal of the model is to be used in a simulation. It should thus be integrated into the OpenSim part of the OpenVisualizer to be of real use. The model code that calculates the consumption for the occurring time slot is trivial, but it has to be integrated with the OpenSim code to be called at the right moment. A simple implementation would keep a counter on how much energy has been consumed since the start of the simulation, the counter would be increased for each time slot that passes. The next step would be to simulate a battery. Instead of just keeping track of the drawn charge, the value is subtracted from the total battery capacity. When the battery gets empty, the mote would be removed from the simulation.

Another place where the model can be used is in the 6TiSCH simulator [28]. If the sleep consumption issues are fixed then the energy model used in the simulator could be replaced by the model described in this thesis. The energy consumption found by the simulation will then take the packet size into account and be more accurate.

Bibliography

[1] D. Christin, A. Reinhardt, P. S. Mogre, R. Steinmetz, et al. Wireless sensor networks and the internet of things: selected challenges. *Proceedings of the 8th GI/ITG KuVS Fachgespräch Drahtlose sensornetze*, pages 31–34, 2009.

[2] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Emerging challenges: Mobile networking for "smart dust". *Journal of Communications and Networks*, 2(3):188–196, Sept 2000.

[3] P. Huang, L. Xiao, S. Soltani, M. W. Mutka, and N. Xi. The evolution of mac protocols in wireless sensor networks: A survey. *IEEE Communications Surveys Tutorials*, 15(1):101–120, First 2013.

[4] Ieee standard for local and metropolitan area networks–part 15.4: Low-rate wireless personal area networks (lr-wpans). *IEEE Std 802.15.4-2011 (Revision of IEEE Std 802.15.4-2006)*, pages 1–314, Sept 2011.

[5] Ieee standard for local and metropolitan area networks–part 15.4: Low-rate wireless personal area networks (lr-wpans) amendment 1: Mac sublayer. *IEEE Std 802.15.4e-2012 (Amendment to IEEE Std 802.15.4-2011)*, pages 1–225, April 2012.

[6] L. Doherty, W. Lindsay, and J. Simon. Channel-specific wireless sensor network path data. In *2007 16th International Conference on Computer Communications and Networks*, pages 89–94, Aug 2007.

[7] T. Watteyne, A. Mehta, and K. Pister. Reliability through frequency diversity: why channel hopping makes sense. In *Proceedings of the 6th*

ACM symposium on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks, pages 116–123. ACM, 2009.

[8] T. Watteyne, M. Palattella, and L. Grieco. Using ieee 802.15.4e time-slotted channel hopping (tsch) in the internet of things (iot): Problem statement. Technical report, 2015.

[9] S. Petersen and S. Carlsen. Wirelesshart versus isa100.11a: The format war hits the factory floor. *IEEE Industrial Electronics Magazine*, 5(4):23–34, Dec 2011.

[10] X. Vilajosana, P. Tuset, T. Watteyne, and K. Pister. Openmote: open-source prototyping platform for the industrial iot. In *International Conference on Ad Hoc Networks*, pages 211–222. Springer, 2015.

[11] Texas Instruments. Achieving Optimum Radio Range. `http://www.ti.com/lit/an/swra479/swra479.pdf`, March 2015.

[12] P. Tuset, X. Vilajosana, and T. Watteyne. Openmote+: a range-agile multi-radio mote. In *EWSN*, pages 333–334, 2016.

[13] D. Dujovne, T. Watteyne, X. Vilajosana, and P. Thubert. 6tisch: deterministic ip-enabled industrial internet (of things). *IEEE Communications Magazine*, 52(12):36–41, December 2014.

[14] J. Hui and P. Thubert. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. Technical Report 6282, September 2011.

[15] A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, JP. Vasseur, and R. Alexander. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. Technical Report 6550, March 2012.

[16] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). Technical Report 7252, June 2014.

[17] Q. Wang, X. Vilajosana, and T. Watteyne. 6tsch operation sublayer (6top). 2013.

[18] T. Watteyne, X. Vilajosana, B. Kerkez, F. Chraim, K. Weekly, Q. Wang, S. Glaser, and K. Pister. Openwsn: a standards-based low-power wireless development environment. *Transactions on Emerging Telecommunications Technologies*, 23(5):480–493, 2012.

[19] Wireshark Foundation. Wireshark. `https://www.wireshark.org`.

[20] D. De Guglielmo, B. Al Nahas, S. Duquennoy, T. Voigt, and G. Anastasi. Analysis and experimental evaluation of ieee 802.15.4e tsch csma-ca algorithm. *IEEE Transactions on Vehicular Technology*, 66(2):1573–1588, Feb 2017.

[21] G. Z. Papadopoulos, A. Mavromatis, X. Fafoutis, N. Montavont, R. Piechocki, T. Tryfonas, and G. Oikonomou. Guard time optimisation and adaptation for energy efficient multi-hop tsch networks. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pages 301–306, Dec 2016.

[22] I. Juc, O. Alphand, R. Guizzetti, M. Favre, and A. Duda. Energy consumption and performance of ieee 802.15.4e tsch and dsme. In *2016 IEEE Wireless Communications and Networking Conference*, pages 1–7, April 2016.

[23] X. Vilajosana, Q. Wang, F. Chraim, T. Watteyne, T. Chang, and K. S. J. Pister. A realistic energy consumption model for tsch networks. *IEEE Sensors Journal*, 14(2):482–489, Feb 2014.

[24] J. M. Munoz. jmmunoz86/openwsn-fw develop_fw-527 branch. `https://github.com/jmmunoz86/openwsn-fw/tree/develop_FW-527`.

[25] P. Tuset. Openmote/openwsn-fw develop_fw-493 branch. `https://github.com/OpenMote/openwsn-fw/tree/develop_FW-493`.

[26] Texas Instruments. CC1200 Low-Power, High-Performance RF Transceiver. `http://www.ti.com/lit/ds/symlink/cc1200.pdf`, July 2013.

[27] Keysight Technologies. N6700 modular power system family data sheet. `http://literature.cdn.keysight.com/litweb/pdf/5989-6319EN.pdf`, 2016.

[28] T. Watteyne, K. Muraoka, N. Accettura, and X. Vilajosana. The 6tisch simulator. `https://bitbucket.org/6tisch/simulator`, 2017.

Appendices

Appendix A

Time slot states

The tables in this appendix show the different states for each type of time slot and in which state the CPU and radio are in at that time.

State in slot	**CPU State**	**Radio State**
SleepStart	Active	Sleep
Sleep	Sleep	Sleep

Table A.1: *States in a Sleep slot*

State in slot	CPU State	Radio State
RxDataOffsetStart	Active	Sleep
RxDataOffset	Sleep	Sleep
RxDataPrepare	Active	Idle
RxDataReady	Sleep	Idle
RxDataListenStart	Active	Idle
RxDataListen	Sleep	Listen
RxProc	Active	Sleep
Sleep	Sleep	Sleep

Table A.2: *States in a RxIdle slot*

State in slot	CPU State	Radio State
RxDataOffsetStart	Active	Sleep
RxDataOffset	Sleep	Sleep
RxDataPrepare	Active	Idle
RxDataReady	Sleep	Idle
RxDataListenStart	Active	Idle
RxDataListen	Sleep	Listen
RxDataStart	Active	Rx
RxData	Sleep	Rx
TxAckOffsetStart	Active	Idle
TxAckOffset	Sleep	Sleep
TxAckPrepare	Active	Idle
TxAckReady	Sleep	Idle
TxAckDelayStart	Active	Idle
TxAckDelay	Sleep	Tx
TxAckStart	Active	Tx
TxAck	Sleep	Tx
RxProc	Active	Sleep
Sleep	Sleep	Sleep

Table A.3: *States in a RxDataTxAck slot*

State in slot	CPU State	Radio State
TxDataOffsetStart	Active	Sleep
TxDataOffset	Sleep	Sleep
TxDataPrepare	Active	Idle
TxDataReady	Sleep	Idle
TxDataDelayStart	Active	Idle
TxDataDelay	Sleep	Tx
TxDataStart	Active	Tx
TxData	Sleep	Tx
RxAckOffsetStart	Active	Sleep
RxAckOffset	Sleep	Sleep
RxAckPrepare	Active	Idle
RxAckReady	Sleep	Idle
RxAckListenStart	Active	Idle
RxAckListen	Sleep	Listen
RxAckStart	Active	Rx
RxAck	Sleep	Rx
TxProc	Active	Idle
Sleep	Sleep	Sleep

Table A.4: *States in a TxDataRxAck slot*

State in slot	CPU State	Radio State
TxDataOffsetStart	Active	Sleep
TxDataOffset	Sleep	Sleep
TxDataPrepare	Active	Idle
TxDataReady	Sleep	Idle
TxDataDelayStart	Active	Idle
TxDataDelay	Sleep	Tx
TxDataStart	Active	Tx
TxData	Sleep	Tx
TxProc	Active	Sleep
Sleep	Sleep	Sleep

Table A.5: *States in a TxData slot*

State in slot	CPU State	Radio State
RxDataOffsetStart	Active	Sleep
RxDataOffset	Sleep	Sleep
RxDataPrepare	Active	Idle
RxDataReady	Sleep	Idle
RxDataListenStart	Active	Idle
RxDataListen	Sleep	Listen
RxDataStart	Active	Rx
RxData	Sleep	Rx
RxProc	Active	Idle
Sleep	Sleep	Sleep

Table A.6: *States in a RxData slot*

State in slot	CPU State	Radio State
TxDataOffsetStart	Active	Sleep
TxDataOffset	Sleep	Sleep
TxDataPrepare	Active	Idle
TxDataReady	Sleep	Idle
TxDataDelayStart	Active	Idle
TxDataDelay	Sleep	Tx
TxDataStart	Active	Tx
TxData	Sleep	Tx
RxAckOffsetStart	Active	Sleep
RxAckOffset	Sleep	Sleep
RxAckPrepare	Active	Idle
RxAckReady	Sleep	Idle
RxAckListenStart	Active	Idle
RxAckListen	Sleep	Listen
TxProc	Active	Sleep
Sleep	Sleep	Sleep

Table A.7: *States in a TxDataRxAckMissing slot*

Appendix B

Duration of states in time slot

The tables in this appendix show the duration of every state for each time slot type. The durations of the states where the cpu is active were measured or are an average of multiple measured values. The time spent sleeping however depends on the active duration and some timing constants. The timing constants used in OpenWSN are provided and explained in Table B.1.

Constant	Duration		Description
	CC2538	CC1200	
DelayTx	366	427	Time between instructing radio to transmit and SFD leaving the radio.
DelayRx	0	0	Time between instructing radio to listen and radio being ready to receive.
MaxTxDataPrepare	2014	2014	Maximum allowed duration of the TxDataPrepare state.
MaxRxDataPrepare	1007	1007	Maximum allowed duration of the RxDataPrepare state.
MaxTxAckPrepare	671	1007	Maximum allowed duration of the TxAckPrepare state.
MaxRxAckPrepare	305	915	Maximum allowed duration of the RxAckPrepare state.
TsLongGT	1300		Half of the PGT. Time between radio starting to listen and expected data reception.
TsShortGT	500		Half of the AGT. Time between radio starting to listen and expected ACK reception.
TsTxOffset	4000		Time between start of time slot and the SFD leaving the radio when transmitting the data.
TsTxAckDelay	4606		Time between end of data transmission and the SFD leaving the radio when transmitting the ACK.
TsSlotDuration	15000		Duration of each time slot.

Table B.1: *Timing constants used in OpenWSN, in μs*

State in slot	Duration (CC2538 radio)	Duration (CC1200 radio)
SleepStart	57	57
Sleep	14943	14943

Table B.2: *Duration of states in the Sleep slot, in μs*

State in slot	Duration (CC2538 radio)	Duration (CC1200 radio)
RxDataOffsetStart	126	126
RxDataOffset	1567	1567
RxDataPrepare	38	676
RxDataReady	969	331
RxDataListenStart	17	58
RxDataListen	2583	2542
RxProc	25	118
Sleep	9675	9582

Table B.3: *Duration of states in the RxIdle slot, in μs*

State in slot	Duration (CC2538 radio)	Duration (CC1200 radio)
RxDataOffsetStart	126	126
RxDataOffset	1567	1567
RxDataPrepare	38	676
RxDataReady	969	331
RxDataListenStart	17	58
RxDataListen	1283	1242
RxDataStart	17	15
RxData	$(3 + pktSize) * 32 - 17$	$(3 + pktSize) * 32 - 15$
TxAckOffsetStart	$126 + (pktSize * 0.91)$	$362 + (pktSize * 8.439)$
TxAckOffset	$3443 - (pktSize * 0.91)$	$2810 - (pktSize * 8.439)$
TxAckPrepare	153	930
TxAckReady	518	77
TxAckDelayStart	17	58
TxAckDelay	349	369
TxAckStart	16	15
TxAck	880	881
RxProc	94	135
Sleep	$5308 - (pktSize * 32)$	$5267 - (pktSize * 32)$

Table B.4: *Duration of states in the RxDataTxAck slot, in μs*

State in slot	Duration (CC2538 radio)	Duration (CC1200 radio)
TxDataOffsetStart	105	105
TxDataOffset	1515	1454
TxDataPrepare	$60 + (pktSize * 0.875)$	$738 + (pktSize * 8.152)$
TxDataReady	$1954 - (pktSize * 0.875)$	$1276 - (pktSize * 8.152)$
TxDataDelayStart	17	58
TxDataDelay	349	369
TxDataStart	16	16
TxData	$(3 + pktSize) * 32 - 16$	$(3 + pktSize) * 32 - 16$
RxAckOffsetStart	32	75
RxAckOffset	3769	3116
RxAckPrepare	38	587
RxAckReady	267	328
RxAckListenStart	17	58
RxAckListen	483	442
RxAckStart	16	15
RxAck	880	881
TxProc	225	619
Sleep	$5177 - (pktSize * 32)$	$4783 - (pktSize * 32)$

Table B.5: *Duration of states in the TxDataRxAck slot, in μs*

State in slot	Duration (CC2538 radio)	Duration (CC1200 radio)
TxDataOffsetStart	105	105
TxDataOffset	1515	1454
TxDataPrepare	$60 + (pktSize * 0.875)$	$738 + (pktSize * 8.152)$
TxDataReady	$1954 - (pktSize * 0.875)$	$1276 - (pktSize * 8.152)$
TxDataDelayStart	17	58
TxDataDelay	349	369
TxDataStart	16	16
TxData	$(3 + pktSize) * 32 - 16$	$(3 + pktSize) * 32 - 16$
TxProc	72	109
Sleep	$10832 - (pktSize * 32)$	$10795 - (pktSize * 32)$

Table B.6: *Duration of states in the TxData slot, in μs*

State in slot	Duration (CC2538 radio)	Duration (CC1200 radio)
RxDataOffsetStart	126	126
RxDataOffset	1567	1567
RxDataPrepare	38	676
RxDataReady	969	331
RxDataListenStart	17	58
RxDataListen	1283	1242
RxDataStart	17	15
RxData	$(3 + pktSize) * 32 - 17$	$(3 + pktSize) * 32 - 15$
RxProc	$198 + (pktSize * 0.91)$	$488 + (pktSize * 8.439)$
Sleep	$10706 - (pktSize * 31.09)$	$10416 - (pktSize * 23.561)$

Table B.7: *Duration of states in the RxData slot, in μs*

State in slot	Duration (CC2538 radio)	Duration (CC1200 radio)
TxDataOffsetStart	105	105
TxDataOffset	1515	1454
TxDataPrepare	$60 + (pktSize * 0.875)$	$738 + (pktSize * 8.152)$
TxDataReady	$1954 - (pktSize * 0.875)$	$1276 - (pktSize * 8.152)$
TxDataDelayStart	17	58
TxDataDelay	349	369
TxDataStart	16	16
TxData	$(3 + pktSize) * 32 - 16$	$(3 + pktSize) * 32 - 16$
RxAckOffsetStart	32	75
RxAckOffset	3769	3116
RxAckPrepare	38	587
RxAckReady	267	328
RxAckListenStart	17	58
RxAckListen	983	942
TxProc	44	137
Sleep	$5754 - (pktSize * 32)$	$5661 - (pktSize * 32)$

Table B.8: *Duration of states in the TxDataRxAckMissing slot, in μs*

Appendix C

Time slot comparion: model vs. measurement

The figures displayed in this appendix show the consumption over time according to the model and what we measured. Each time slot is compared for both radios.

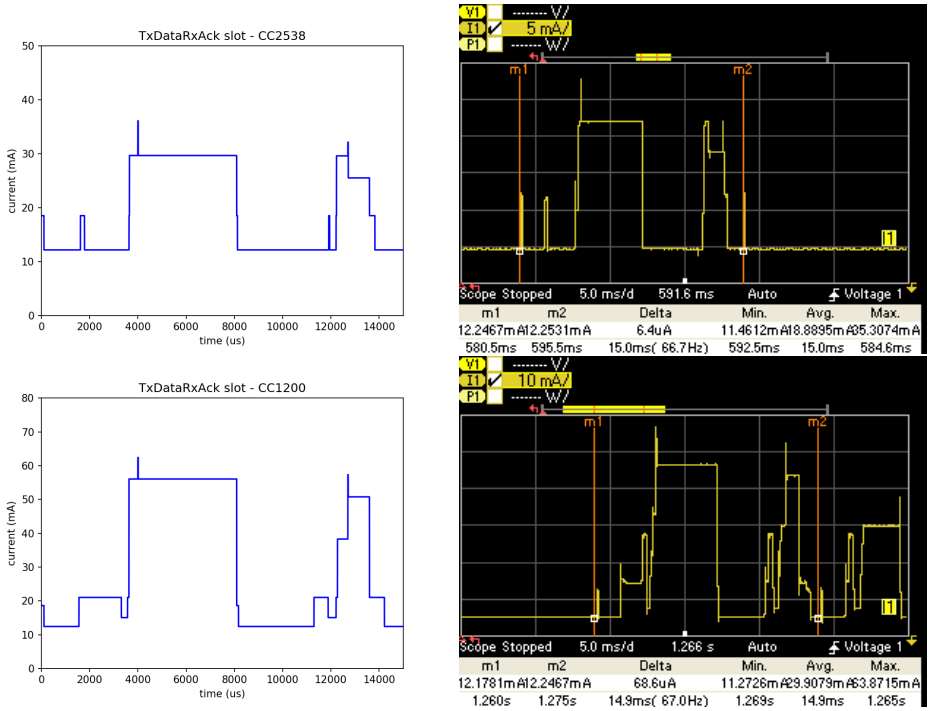

Figure C.1: *Comparison between calculated (left) and measured (right) TxDataRxAck time slot when using the CC2538 (top) and CC1200 (bottom) radios*

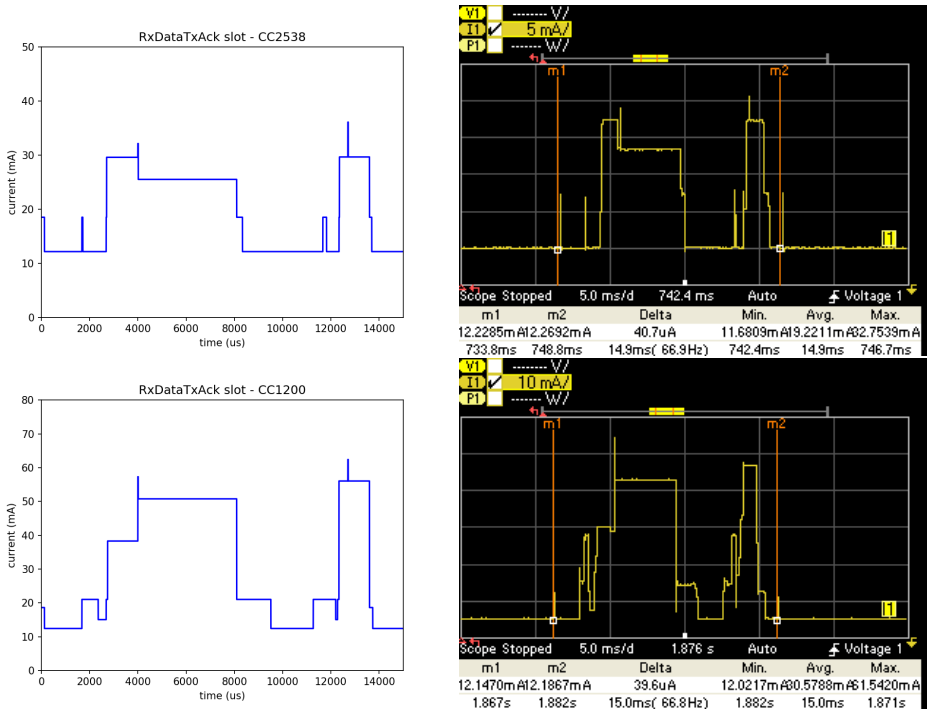

Figure C.2: *Comparison between calculated (left) and measured (right) RxDataTxAck time slot when using the CC2538 (top) and CC1200 (bottom) radios*

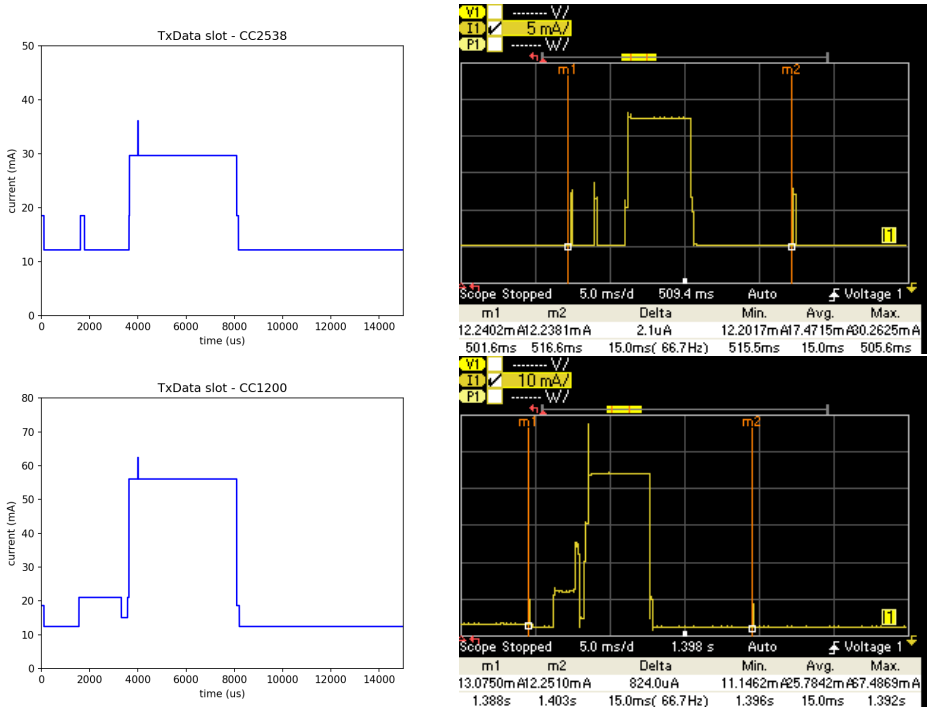

Figure C.3: *Comparison between calculated (left) and measured (right) TxData time slot when using the CC2538 (top) and CC1200 (bottom) radios*

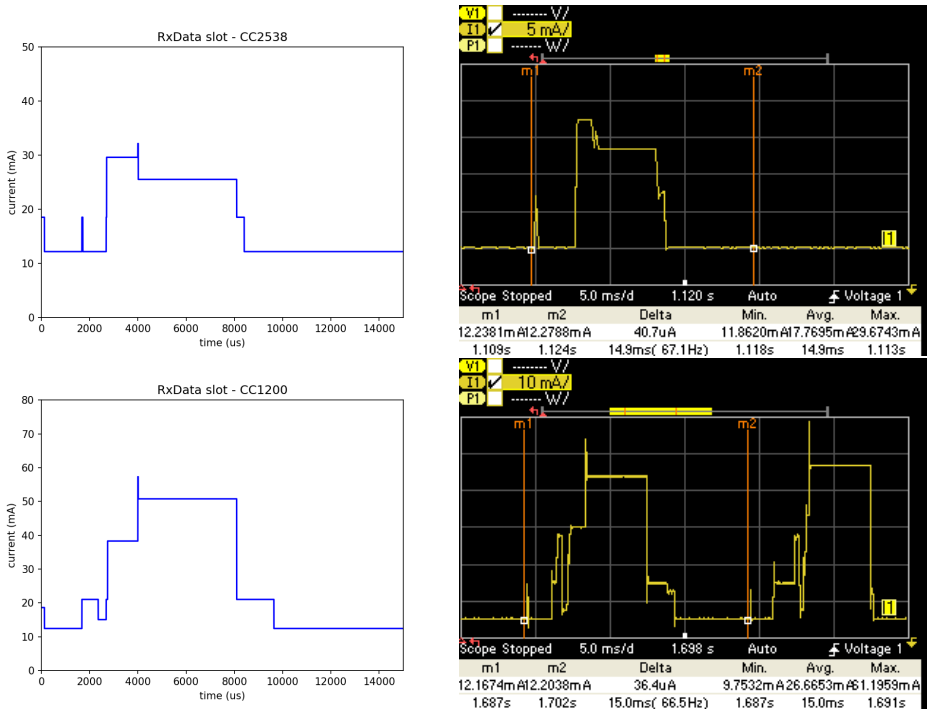

Figure C.4: *Comparison between calculated (left) and measured (right) RxData time slot when using the CC2538 (top) and CC1200 (bottom) radios*

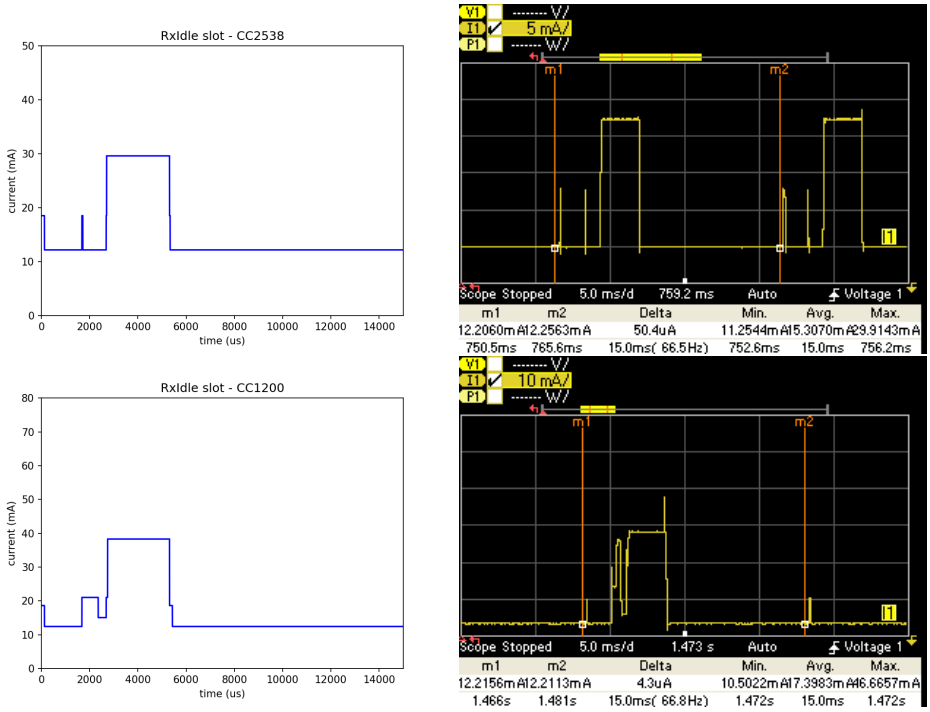

Figure C.5: *Comparison between calculated (left) and measured (right) RxIdle time slot when using the CC2538 (top) and CC1200 (bottom) radios*

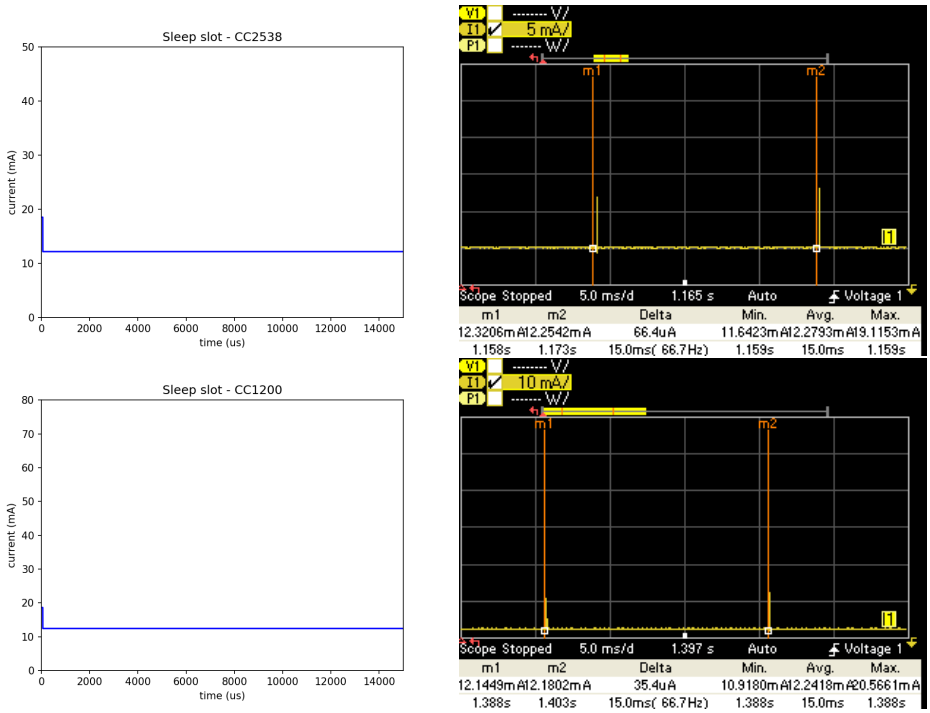

Figure C.6: *Comparison between calculated (left) and measured (right) Sleep time slot when using the CC2538 (top) and CC1200 (bottom) radios*