

University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Python-based TinyIPFIX for IEEE 802.15.4-based Wireless Sensor Networks using ESP32 Microcontrollers

Ramon Huber

Baar, Switzerland

Student ID: 15-931-371

Supervisor: Eryk Schiller

Date of Submission: November 11, 2020

University of Zurich

Department of Informatics (IFI)

Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

The goal of this thesis is to implement TinyIPFIX, which is an application layer protocol to send data in a Wireless Sensor Network (WSN). Devices used in a WSN are often battery powered, therefore, it is important that the protocol used to send data is very efficient. The main principle of TinyIPFIX is to send data and metadata in separate messages, such that the metadata messages can be sent less often than the data messages, which decreases overhead and saves energy. The implementation is performed for the Espressif ESP32-WROOM-32D module, which becomes popular, while it is cheap and more powerful than older devices. The implementation is provided using the MicroPython programming language, which implies that the code is more readable and simple in comparison to a lower level programming language. TinyIPFIX has already been demonstrated on older devices using a low level language; the goal of this thesis is to bring TinyIPFIX to the next generation of devices and provide it as extensible software.

Das Ziel dieser Bachelorarbeit ist es TinyIPFIX, ein Protokoll der Anwendungsschicht zum Senden von Daten in einem Wireless Sensor Network (WSN), zu implementieren. Geräte die für WSNs gebraucht werden sind oft batteriebetrieben, was es wichtig macht, dass ein Protokoll um daten zu senden sehr effizient ist. Die Grundidee von TinyIPFIX ist es Daten und Metadaten in separaten Nachrichten zu schicken, sodass Metadaten-Nachrichten weniger oft gesendet werden können als Daten-Nachrichten, was zu weniger Overhead und somit zu weniger Energieverbrauch führt. Implementiert wird TinyIPFIX für das Espressif ESP32-WROOM-32D Modul, das immer mehr an popularität gewinnt und sehr günstig und besser ausgestattet ist als ältere Geräte. Als Programmiersprache wird MicroPython verwendet, was dazu führt das der Code besser lesbar und einfacher ist als wenn eine lower-level Programmiersprache verwendet würde. TinyIPFIX wurde bereits auf älteren Modulen in einer low-level Programmiersprache implementiert; das Ziel dieser Arbeit ist es nun TinyIPFIX auf die neue Generation von Geräten zu bringen und es einfacher zu machen den Code von TinyIPFIX zu erweitern.

Acknowledgments

I would like to first thank my supervisor Dr. Eryk Schiller for the support and valuable feedback I received, for the sharing of his expertise and ideas, without which this thesis would not have been possible in this form. Furthermore, I want to thank Prof. Dr. Burkhard Stiller for the possibility and support to do this thesis at the Communication Systems Group (CSG) of the University of Zurich.

I also want to thank my family and friends for their support during this time.

Contents

Abstract **i**

Acknowledgments **iii**

1 Introduction **1**

1.1 Motivation 1

1.2 Description of Work 2

1.3 Thesis Outline 2

2 Technologies and Related Work **3**

2.1 Internet of Things 3

2.2 Wireless Sensor Networks 3

2.3 Hardware 4

2.3.1 Sparkfun XBee Explorer Dongle 4

2.3.2 ESP32-WROOM-32D 4

2.3.3 XBee Pro S2C Through-Hole Module 7

2.4 UART 8

2.5 MicroPython 9

2.6 Protocols 9
2.6.1 IPFIX 9
2.6.2 TinyIPFIX 10
2.6.3 IEEE 802.15.4 18
2.7 ZeroMQ - Publish/Subscribe 18
2.8 Thonny 19

3 Design and Implementation **21**
3.1 Design 21
3.1.1 Sensor Network (TinyIPFIX) 21
3.1.2 Collector and Publish/Subscribe Network 23
3.1.3 Test Network of this Thesis 24
3.1.4 Use Cases 25
3.1.5 Class Diagram 27
3.2 Implementation 31
3.2.1 Hardware Setup 31
3.2.2 Implementation of the TinyIPFIX Protocol 40
3.2.3 Implementation of Device Related Classes 40
3.2.4 Implementation of Collector and Processing Related Classes 45

4 Evaluation **47**
4.1 Overhead 47
4.2 Transmission reliability 50
4.3 Energy consumption 52

5 Summary, Conclusions and Future Work **55**

Bibliography **57**

Abbreviations **61**

List of Figures **63**

List of Tables **65**

Listings **67**

A SuperB Schematic **69**

B TinyIPFIX Message Navigation Properties **71**

C Connecting the Pins of the Devices **73**

Chapter 1

Introduction

1.1 Motivation

Nowadays, sensors are cheap enough that it is feasible to place them everywhere. Sensors connected to each other are referred to as Wireless Sensor Networks (WSNs). Such WSNs have many influential use cases. For example, a WSN can be used to monitor a production company and increase the quality of the products. WSNs can be installed in retirement homes to monitor whether there is movement in the house or whether an inhabitant could be in trouble. We all heard of the smart home, eventually even smart cities could be realized, that for example measure the energy consumption of each household and company in order to plan energy production precisely instead of producing too much energy that is then wasted.

With all the potential that a WSN brings, it also brings challenges. One such challenge is the programming of the sensors, that are typically very constrained in their hardware. Another challenge is the fact that sensors are normally battery powered, which requires them to run very energy efficiently, because having to change the batteries often would be infeasible when there can be hundreds of sensors deployed.

To deal with those challenges, an application layer protocol has been specified. Tiny Internet Protocol Flow Information Export (TinyIPFIX) sends metadata and actual sensor

data in separate messages and therefore decreases overhead and energy cost for communication. Using TinyIPFIX in WSNs is supposed to let the devices run longer on a single battery charge, than using another protocol.

1.2 Description of Work

In this thesis, the TinyIPFIX protocol will be implemented on the Espressif ESP-WROOM-32D module. As a first step, the devices featuring the ESP-WROOM-32D module need to be made programmable by flashing the MicroPython firmware onto them. Further they need to be connected with Digi XBee devices, such that data transfer using IEEE 802.15.4 is possible between different devices featuring the ESP-WROOM-32D module. As a next step the protocol needs to be implemented on the devices. Lastly the implementation needs to be evaluated in terms of energy efficiency and overhead and it has to be shown that the protocol works in a test network.

1.3 Thesis Outline

The thesis is structured in the following way. Chapter 2 discusses the technologies used for this thesis and related work that this thesis builds upon. Furthermore, different Protocols and Hardware pieces are introduced. Chapter 3 discusses different design decisions and shows concrete examples of the implementation. Chapter 4 gives an evaluation of the implementation, while Chapter 5 summarizes the whole project and gives a prospect about future work.

Chapter 2

Technologies and Related Work

2.1 Internet of Things

The Internet-of-Things (IoT) [19] denotes things that were traditionally not connected to other devices or the Internet and are now organized as a network. An example would be a smart carpet that has sensors in it and that monitors when someone falls on it. If the sensors register that someone fell, they can send a message to some predefined persons, or inform emergency services. To this end, the carpet needs to be connected to the internet.

2.2 Wireless Sensor Networks

A Wireless Sensor Network (WSN) [18] is a network of sensors, that monitor some conditions of the environment. The data that the sensors collect is automatically and wirelessly sent to a central place without the need of a human interaction. IoT and WSNs are related topics; WSNs are often only required because of the IoT. Indeed, the example from above with the smart carpet can also be used as an example for a WSN, in the case that the sensors in the carpet are connected wirelessly and automatically send their data to a central place. The IoT in combination with WSNs has a potential to change many aspects of life. Monitoring the environment and automatically sending the data to a central place, where it can be processed, can innovate a lot of areas.

2.3 Hardware

For this work four kinds of Devices were used. Three constrained devices, namely the ESP32 DevKitC V4, the Macchina SuperB and the Digi XBee and a non-constrained Computer (running Windows 10 in this case). The ESP32 DevKitC V4 and the Macchina SuperB were used as sensors in a WSN. To connect some of the devices, a breadboard [3] was used. The ESP32 DevKitC is already compatible with a standard 2.54mm breadboard, but for the Macchina SuperB and the Digi XBee an adapter had to be used. Two kinds of adapters were used, one adapter just adapted the 2 mm spaced Pins from the XBee and the SuperB to the 2.54 mm spaced Pin holes on the breadboard. The other adapter, the Sparkfun Explorer Dongle, did the same, but additionally it also provides a Universal Serial Bus (USB) plug that can be plugged into a Computer. The second adapter made it possible to talk to those devices using a computer with a USB port.

2.3.1 Sparkfun XBee Explorer Dongle

The Sparkfun XBee Explorer Dongle [36] is the Adapter that was used to connect the SuperB and XBee to the breadboard. It features a USB Plug that can be connected to a computer. The Sparkfun XBee Explorer was important for this work because it made it possible to connect the Bee Devices to the computer and therefore program them. In the early stages of the project the Bee Devices were powered with the Sparkfun Explorer, because it was not yet clear which batteries could be used and how.

2.3.2 ESP32-WROOM-32D

The ESP32-WROOM-32D [25, 31] is a relatively new microcontroller. It is a revised version of the ESP32-WROOM-32 that was released in September 2016 by Espressif Systems. Some highlights among the features of the ESP32-WROOM-32D are

- two low-power Xtensa 32-bit microprocessors with an adjustable clock frequency between 80 MHz and 240 MHz

- 4MB of external SPI Flash
- 448 KB embedded ROM
- 520 KB embedded SRAM
- Bluetooth v4.2 BR/EDR and BLE
- 802.11 b/g/n and
- three Universal Asynchronous Receiver Transmitter (UART) interfaces

Compared to other microcontrollers on the Market the ESP32-WROOM-32D is powerful and small. Because the circuit specification is publicly available, it is easy for manufacturers to integrate the ESP32-WROOM-32D into their printed circuit boards (PCB), which explains why there are so many ESP32-WROOM-32 based PCBs around.

As mentioned above, the ESP32-WROOM Chip is open source, and therefore it is possible to program instruction sets and libraries for the chip. People have done exactly that by adjusting MicroPython to work on the chip. The MicroPython Firmware can be flashed to the ESP32-WROOM-32D Module and then the ESP32 can be programmed using MicroPython.

The choice to use ESP32 based microcontrollers for this thesis is justified by the findings of [31], namely that the ESP32-WROOM-32 has great performance properties for a good price compared to other microcontrollers on the market and especially the ESP32 DevKitC is very useful for experimenting as it is breadboard friendly and easy to set up. Another argument for the ESP32-WROOM based microcontrollers is their wide adoption and therefore big and active community.

Espressif ESP32 DevKitC V4

The ESP32 DevKitC V4 [5] features the ESP32-WROOM-32D Module and is one of the Devices that were used for this work. The ESP32 DevKitC V4 is manufactured by Espressif, just like the ESP32-WROOM-32D. It is perfectly suited to be used for

experiments as it is small, cheap, powerful and widely used [31]. Another benefit of the DevKitC V4 is that it can be directly plugged into a computer using a mini-USB cable, and can then be programmed over this connection.

For this thesis, four ESP32 DevKitC were flashed with MicroPython, because using MicroPython the development of the program should progress very fast. Compared to other languages that could be used to program the ESP32 like C or C++, MicroPython should be rather straight forward to use. Figure 2.1 shows the ESP32 DevKitC V4 device.

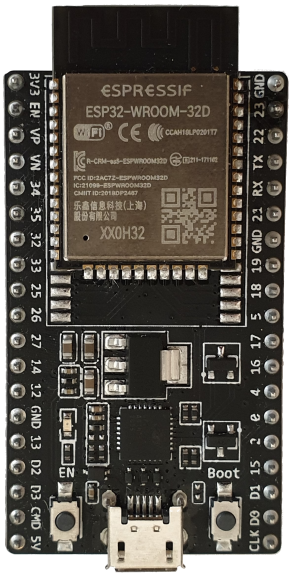


Figure 2.1: Espressif ESP32 DevKitC V4

Macchina SuperB

The SuperB [13] from Macchina also features the ESP32-WROOM-32 Module, just like the ESP32 DevKitC, therefore it has the same benefits as the DevKitC. One additional benefit that the SuperB has over the DevKitC is the even smaller size. A downside is that it does not plug into a computer as easily as the DevKitC, for this the Sparkfun Explorer has to be used.

The SuperB has the Bee socket which means that it can be used with the same expansion board as the XBee, which also has the Bee socket.

For this thesis two SuperBs were flashed with MicroPython. Figure 2.2 shows the device.



Figure 2.2: Macchina SuperB

2.3.3 XBee Pro S2C Through-Hole Module

The XBee [22] is a module that is manufactured by Digi. It is a wireless connectivity module and can be controlled via the UART interface. The XBee was used in this work because it features a 802.15.4 function set that was needed to send data between devices (i.e. between two ESP32s that are each connected to an XBee modem). IEEE 802.15.4 has the benefit of needing less energy than IEEE 802.11 which would be available on the ESP32.

In contrast to the ESP32-WROOM-32D based devices, the circuit of the XBee is not freely available and therefore working with it is much more constrained. While the ESP32 based devices can be programmed using MicroPython, the XBee can only be programmed using AT [2] and API [1] commands specified by the manufacturer. To ease the use of the XBee the manufacturer created a software called XCTU [15] that facilitates the work with the XBee. Instead of sending the AT commands via the Command Line, using XCTU it possible to do it via a nice graphical user interface (GUI). Once the XBee is configured via the GUI, data can be sent to the XBee from the ESP32 based devices over the UART interface, the XBee then just takes this data and sends it to the configured address. Figure 2.3 shows one of the XBees that was used.



Figure 2.3: Digi XBee Pro S2C

2.4 UART

Universal Asynchronous Receiver Transmitter (UART) [24, 4] is a piece of hardware that may be integrated in a microcontroller. The UART inside a device allows for the communication with other devices over a hardware connection. UART allows a full-duplex asynchronous communication. Two devices need to be connected with cables to make the communication over UART possible. There are two cables needed; one cable connects the Transmission (TX) Pin of the first device with the Reception (RX) Pin of the second and the other cable connects the RX Pin of the first device with the TX of the second. The RX Pin is the Pin, where the device expects the data of the UART connection to arrive, the TX (transmit) is used by a device to send data.

Two devices that are connected together need to agree on the configuration of their UART controllers, meaning on the configuration of the baudrate (equal to bitrate in the case of UART), the number of data bits, the number of stop bits, parity and flow control. These parameters define the structure of a UART-frame. Due to the the fact that two devices that are connected over UART do not have a common clock, UART-frames have to have this predefined structure, such that the two devices can use this structure together with the baudrate to know when a frame starts and when it ends. While nothing is sent by a device, the TX Pin is always set to one. A UART-frame has a startbit, which is always a logical zero. In this way, the receiving device knows, when the sending device is

not sending anything at the moment, meaning the signal is at one, and then the signal changes to zero for the time of one bit, then a new frame starts. After the startbit, a defined number of data-bits follow in the frame. They are then followed by a parity bit (even or odd parity) if it is defined or no parity bit. Then to end the UART-frame the defined number of stop bits is sent. The stop bits are always a logical one, such that the receiving device can then wait for a zero, which will be the startbit of the next frame. This transition from one to zero, whenever a new frame starts, makes sure that the two devices stay in synchronization with each other.

2.5 MicroPython

MicroPython [10] is a lightweight version of Python 3 and is the programming language and firmware that was used for the ESP32 microcontrollers in this work. It includes only the most important parts of the standard-library and is optimized for microcontrollers. MicroPython itself is implemented in C, which makes it fast, while the intuitive python syntax makes it easy to use.

2.6 Protocols

In this section different protocols are introduced in order to explain how data is sent between the devices in this work.

2.6.1 IPFIX

Internet Protocol Flow Information Export (IPFIX) [21, 20] is an application layer protocol. The purpose of this protocol is to send information about the traffic flow in a network. Traffic flow information can for example be necessary for administrative purposes such as billing. TinyIPFIX (the protocol used for this work) is derived from IPFIX, therefore, there are many similarities between them.

What is special about IPFIX is that it organizes data into Template and Data records. This way, the redundant meta information, which is being sent in Template messages, can be sent less often than the actual data that is sent in Data Messages. IPFIX first sends meta information in a template record set and then multiple data record sets that contain very little meta-information. Other protocols send data and meta-data in one record, which means that meta-information is sent more often than necessary.

Another characteristic of IPFIX is that it is push-based. This means that the sender sends data when it is ready and there is no method for a receiver to request data. Generally, in IPFIX, messages are only sent in one direction, which implies that there is also no way for the sender to know whether a sent packet has arrived or not. The push-based unidirectional traffic that is split up into data and template records has the advantage that it is very data and therefore energy efficient.

2.6.2 TinyIPFIX

TinyIPFIX [33] is an application layer protocol that is implemented in this thesis. It is derived from IPFIX and is optimized for constrained networks. TinyIPFIX has less overhead than IPFIX and is also push-based and unidirectional as well as it splits its messages into data and template messages. To save energy, data messages are sent more often than the template messages, which contain the information on the decoding of data messages. This is reasonable, because template messages repeat themselves and do not change often like data messages, while data messages depend on the sensor readings. Templates normally do not change on a device. For example, a device always sending a 4 byte (i.e., float type) temperature readings does not suddenly provide an 8 byte (i.e., double type) humidity reading. The reason for the retransmition of template messages is that a lost template message, which is not retransmitted, causes the following data messages not usable as they cannot be decoded.

TinyIPFIX is very data and energy efficient, which makes it a perfect fit for WSN, because sensors in a WSN are often battery powered and using TinyIPFIX will contribute to the longevity of the batteries. Figure 2.4 gives an overview of how a TinyIPFIX message is constructed.

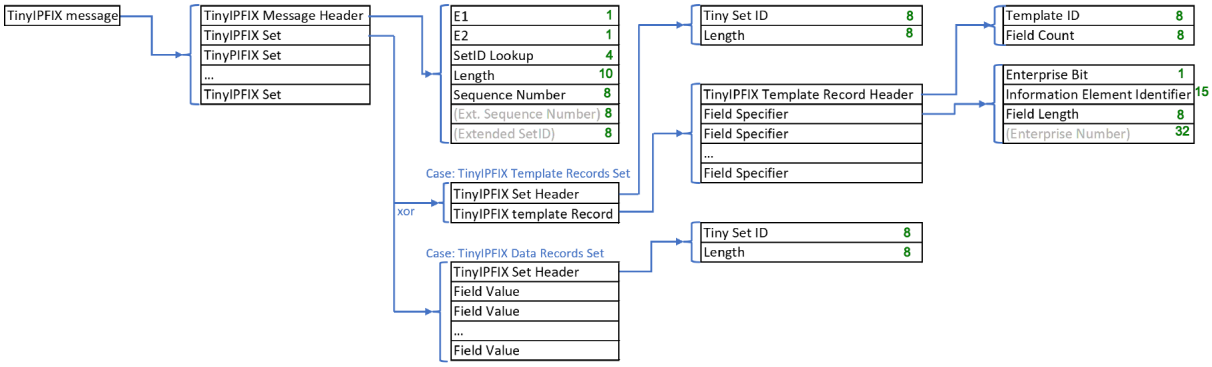


Figure 2.4: TinyIPFIX Message

The figure can be read in the following way. Each field contains an element of a TinyIPFIX message. A field with an arrow coming out of it means that everything coming after that arrow is a more in depth view of the field. For example, the leftmost field ”TinyIPFIX Message” is actually constructed of a ”TinyIPFIX Message Header” and one or more ”TinyIPFIX Set”, so sending a TinyIPFIX Message actually means sending a TinyIPFIX Header and a couple of TinyIPFIX Sets. Fields that do not have an arrow coming out of them (the ones with the green numbers) are the ones that are atomic. The green numbers indicate how many bits each of the atomic fields is long. Greyed out fields are optional and may or may not be present in a TinyIPFIX Message (more on that later). A TinyIPFIX Message has one or multiple TinyIPFIX Sets, but either all of those sets are Template Record Sets or all are Data Record Sets, different TinyIPFIX Set types cannot be present in the same message.

TinyIPFIX Message Header

The TinyIPFIX Message header is the first part of each TinyIPFIX Message. It contains the Fields E1, E2, SetID Lookup, Length and Sequence Number. Additionally, it can have either of the two or both of the fields Extended Sequence Number and Extended SetID. Figure 2.5 shows the TinyIPFIX Message Header.

The E1 field gets a value of zero or one. In the case of one, the Extended SetID field is present. In turn, the Extended SetId field is absent in the case of zero.

The E2 field also gets zero or one. Should the Value be one, the field Extended Sequence

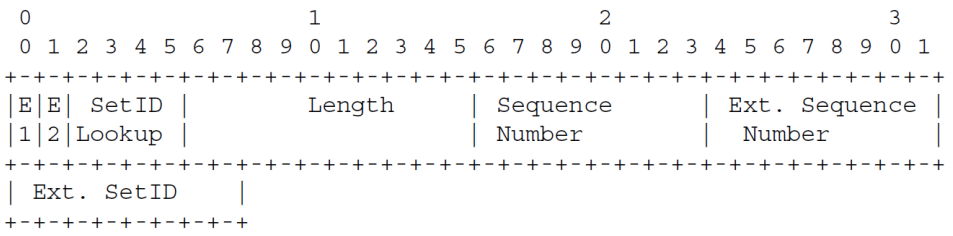


Figure 2.5: TinyIPFIX Message Header, Source: [33]

Number is present, otherwise the Extended Sequence Number field is not absent.

The field SetID Lookup is set to zero for the purpose of this thesis, as the definition is not clear in the Request for Comments (RFC) 8272 [33], which specifies TinyIPFIX.

The Length field stores the total length of the TinyIPFIX Message in bytes. The total length includes the length of the TinyIPFIX Message Header and all the Sets of the message.

The field sequence number holds the sequence number of the TinyIPFIX Message. The first message that a particular sender sends to a particular receiver has the sequence number 0. Then, with every package that the sender sends to that receiver, the sequence number is increased by one. This way, the receiver knows whether it received all messages. If one message gets lost, this shall be logged, but there is nothing that can be done about this fact. If E2 is zero, the sequence number is 8 bit long (i.e., the maximum value of the sequence number is 255). If E2 is one, the sequence number is 16 bit long, which means, it can have a maximum value of 65535. When the maximum value is reached, the sequence number restarts from zero.

The Extended SetID is never present for the purpose of this thesis, assuming E1 always zero.

Figure 2.6 shows the IPFIX Message Header. Comparing this header with the header of TinyIPFIX messages, it can first be noticed that the IPFIX header is 16 bytes long while the TinyIPFIX header is only 3 bytes long in the case of E1 = E2 = 0. The smaller header is achieved using field compression (e.g. the length field is only 10 bit instead of 16 bit), the two optional fields that can often be omitted and fields that are omitted completely

like the export time. If the export time is needed, it can be sent as a field value in the data record.

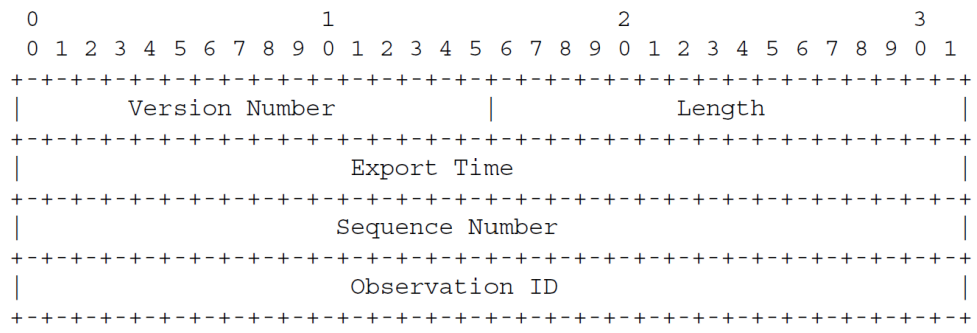


Figure 2.6: IPFIX Message Header, Source: [33]

TinyIPFIX functions with minimum overhead, which can be clearly demonstrated by comparing the IPFIX Header with the TinyIPFIX Header.

TinyIPFIX Template Records Set

After the Header, there are one or more TinyIPFIX Sets in a TinyIPFIX Message. A message can either contain only Template Record Sets or only Data Record sets, but not both at the same time.

Template Record Sets contains information on how to decode Data Record Sets.

Set Header A Template Record Set has a Set header, which contains a Tiny Set ID and a Lenght Field as seen in Figure 2.4. The Tiny Set ID is always set to two for Template Record Sets, as this is what a Set identifies as a Template Record Set. The Length Field stores the total length of the Template Record Set in bytes (including the header).

Template Record After the header follows a Template Record. Template Records define the structure and meaning of Data Record Sets. The Template Record is started with a Template Record Header (Not to be confused with the Template Record Set's Set Header from above). The Template Record Header contains a Template ID and a Field Count. The Template ID is a number between 128 and 255, which uniquely identifies

the template record (no two different template records may have the same Template ID). Data Record Sets have a Tiny Set ID, which is equal to a Template ID of a Template Record, this shows which Data Record Set is defined by which Template Record. The Field Count Field stores the number of field specifiers that are present in the Template Record. The Data Record Sets that are specified by a Template Record must also have the same number of Field Values, as the Template Record has Field Specifiers.

Field Specifier A field specifier specifies the format and meaning of a field value from a Data Record Set. Each field value of a Data Record Set has to be specified by a field specifier. Each Field Specifier has an Enterprise Bit, an Information Element Identifier a Field Length and an optional Enterprise Number. The Enterprise Bit is either zero or one, if it is zero, then the Enterprise Number is not present and the Field Specifier is a default Field Specifier defined by the Internet Assigned Numbers Authority (IANA). Using default Field Specifiers from IANA has the benefit that they are the same no matter which company uses them and where in the world they are used. If the Enterprise Bit is one, the Enterprise Number is present and the Field Specifier is a custom one defined internally in a company. The Information Element Identifier is a unique identifier for an Information element. The specification of an Information Element can then be found using the Information Element Identifier. The specification defines for example the data type and description of the Information Element. This information can then be used to make sense of the Field Values from a Data Record Set. The field length stores the length of the field value specified by the field specifier in bytes. The enterprise number is a number that is registered with IANA and uniquely identifies the company or organization. Using the Information Element Identifier the meaning of a field value can be looked up on IANA's website if the Enterprise Bit is zero. Figure 2.7 shows an extract of the Field Specifier definition from IANA. If the Enterprise Bit is one, then the Information Element Identifier and the Enterprise Number can be used to find the specification of the Field Value inside the company. For this the company has to have a file where they store in some way the specification of the Field Value.

ElementID	Name	Abstract Data Type	Data Type Semantics	Status	Description	Units
0	Reserved					
1	octetDeltaCount	unsigned64	deltaCounter	current	The number of octets since the previous report (if any) in incoming packets for this Flow at the Observation Point. The number of octets includes IP header(s) and IP payload.	octets
2	packetDeltaCount	unsigned64	deltaCounter	current	The number of incoming packets since the previous report (if any) for this Flow at the Observation Point.	packets
3	deltaFlowCount	unsigned64	deltaCounter	current	The conservative count of Original Flows contributing to this Aggregated Flow; may be distributed via any of the methods expressed by the valueDistributionMethod Information Element.	flows
4	protocolIdentifier	unsigned8	identifier	current	The value of the protocol number in the IP packet header. The protocol number identifies the IP packet payload type. Protocol numbers are defined in the IANA Protocol Numbers registry. In Internet Protocol version 4 (IPv4), this is carried in the Protocol field. In Internet Protocol version 6 (IPv6), this is carried in the Next Header field in the last extension header of the packet.	

Figure 2.7: IANA Field Specifiers, Source: [7]

TinyIPFIX Data Records Set

Set Header Data Record Sets also start with a Set Header. This header contains a Tiny Set Id and a Length field. The Tiny Set Id is between 128 and 255 (just like the Template Id is between 128 and 255) and has to correspond to a Template Id. The Length field stores the length of the Set in bytes including the header.

Field Values After the header there follow as many Field Values as are Field Specifiers in the Template Record Set that corresponds to the Data Record Set (The Template Record Set with Template Id equal to the Tiny Set Id of the Data Record Set). The meaning (e.g. is it a temperature value) and data format (e.g. unsigned16) of the Field Value can then be looked up using the Field Specifier.

Devices in TinyIPFIX

TinyIPFIX foresees three kinds of devices in a network:

- TinyIPFIX Devices (also known as End Devices),
- TinyIPFIX Concentrators,
- TinyIPFIX Collector.

TinyIPFIX Device (End Device) A TinyIPFIX Device is the simplest Device, it is measuring the environment using sensors and sends the Data to the TinyIPFIX Concentrators or the TinyIPFIX Collector. In a network most devices are typically TinyIPFIX Devices.

TinyIPFIX Concentrator In a network there can be TinyIPFIX Concentrators. The Concentrators measure the environment, just like the TinyIPFIX Devices do, but additionally, they cannot only send data (export data), but they can receive data (collect) from other devices. Concentrators aggregate the received data together with their own sensed data into a single TinyIPFIX Message. This aggregation has the effect that less messages but bigger ones are sent in the network, which saves energy and data overall, because for example there is more actual sensed data compared to message header data in a bigger message. Another important task that Concentrators fulfill is that they make it possible for TinyIPFIX Devices to send data to a Collector (by sending it to the Concentrator first) that would be out of reach without the Concentrators. This allows for bigger networks than if every TinyIPFIX Device would have to be in range of the Collector.

TinyIPFIX Collector The TinyIPFIX Collector is the only device in the network that is not a constrained device, it can for example be a normal desktop computer while all the other devices in the network are typically microcontrollers. The Collector collects all the messages that are sent to it and stores and processes them. In this work, the Publish-Subscribe pattern was chosen to process the data. Every Message that is sent by a TinyIPFIX device or a TinyIPFIX Concentrator has to arrive at the Collector at the end. The Collector does not measure something itself. There is only one Collector in a TinyIPFIX network, while there can be multiple TinyIPFIX Devices and TinyIPFIX Concentrators.

Figure 2.8 shows how a TinyIPFIX network could look like.

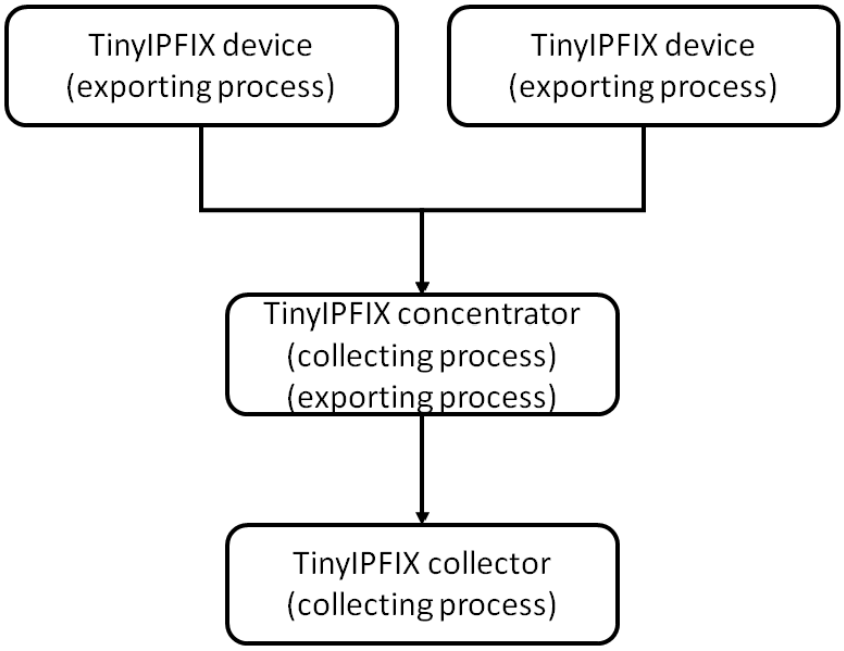


Figure 2.8: TinyIPFIX Devices

Previous implementations of TinyIPFIX

Thomas Kothmayr implemented TinyIPFIX on TinyOS 2.x using the IRIS Mote as a microcontroller [29]. He found that using TinyIPFIX in a WSN is more energy preserving than other options.

Michael Meister implemented TinyIPFIX on Contiki 2.7 using the TelosB Mote as a microcontroller [32].

Livio Sgier made an improved implementation of TinyIPFIX on Contiki and TelosB [34]. He focused on the reduction of the memory footprint, to enable the development of other programs on top of TinyIPFIX on the very memory constrained TelosB.

Livio Sgier made another project [35] where he added aggregation support for the implementation he previously worked on in [34].

The Purpose of this thesis is to bring TinyIPFIX to a newer and more capable platform, namely the ESP32. For this reason, this work makes a completely new implementation in the high-level language MicroPython, because with MicroPython much more work can be done in a shorter time than could be done in the low-level language C, that was used

in older implementations. MicroPython is much more attractive for students because it is easier to learn than C and therefore the chance that future projects will be done with TinyIPFIX and the ESP32 based devices is increased.

2.6.3 IEEE 802.15.4

IEEE 802.15.4 [17, 23] is an energy efficient standard for a Wireless Personal Area Network (WPAN) used in this work with the help of Digi XBee devices. IEEE 802.15.4 is optimized for simplicity and energy efficiency rather than for a broadband communication like IEEE 802.11 [8], which is typically recognized as Wireless Fidelity (WIFI). IEEE 802.15.4 has a range of up to a couple of meters. To achieve a longer range, nodes in a network may be organized in a multi-hop [11] fashion such that network packets are forwarded by intermediate devices until they finally arrive at the final destination. This is in contrast to a direct radio communication between the source and destination, which requires increased power. In the case of TinyIPFIX, the TinyIPFIX Concentrators are perfectly suited for the job of intermediate forwarders.

IEEE 802.15.4 also achieves low power requirements because often the devices in the network may go into an energy saving mode most of the time and wake up only if they need to send or receive data. Because of the simplicity and the energy efficiency, IEEE 802.15.4 is perfectly suited for WSNs, which is why it was chosen as the communication protocol in this work.

2.7 ZeroMQ - Publish/Subscribe

ZeroMQ [16] is a library available in Python that features an implementation of the Publish/Subscribe communication paradigm. This is a communication framework used in this work for handling messages from the WSN. The Publish/Subscribe message queue coordinates data distribution among different software components. The benefit of ZMQ is that the publisher of a message does not need to know anything about the subscribers of a message, while a subscriber may easily get messages without needing to understand how

they were created. This makes it very easy to distribute data among different components and also to add and remove components without the need of adjusting the publisher. Publishers publish their messages to different topics. The subscriber subscribed to a given topic only receives a message when she has a valid subscription to that topic.

2.8 Thonny

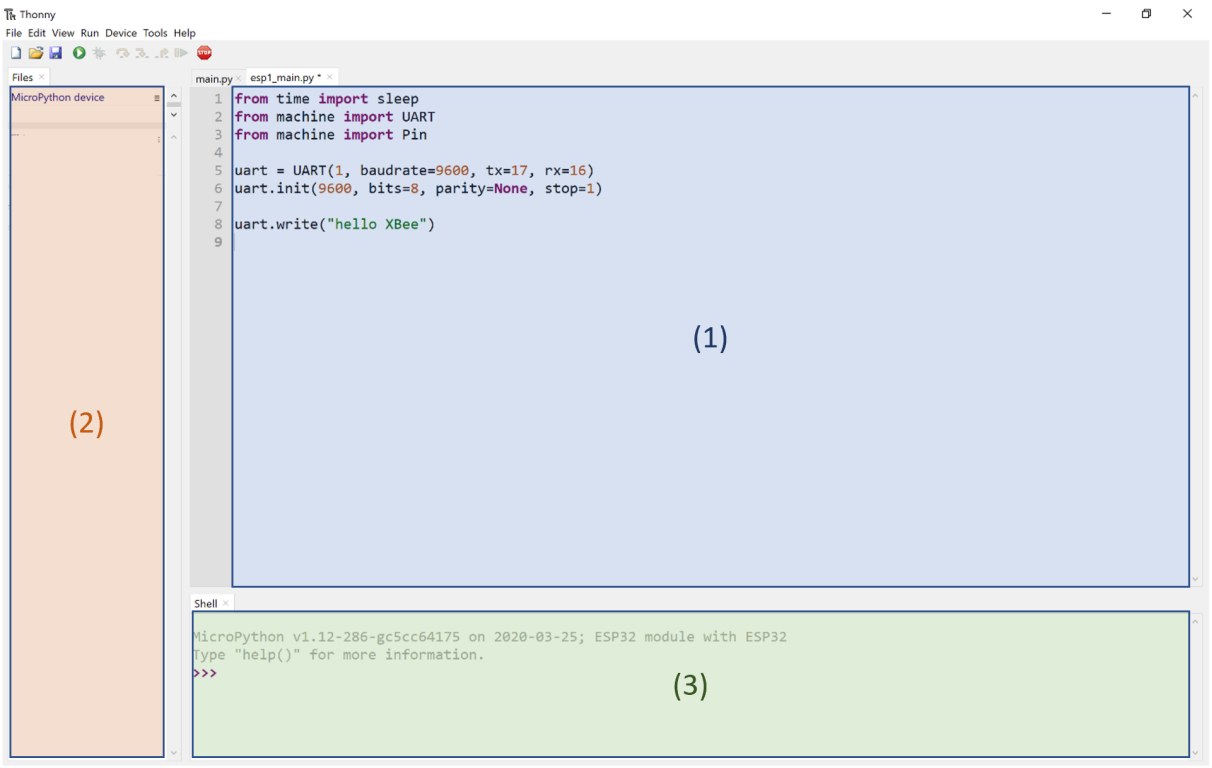


Figure 2.9: Thonny IDE

Thonny [14] is an Integrated Development Environment (IDE) that is used to program ESP32 based devices. Thonny connects to ESP32 over UART. It can either run code on ESP32, which is stored by the user in a file in the main section of Thonny, cf. Figure 2.9 (1), or it can save the content on the ESP32 device using the left panel (2). The shell resides in the bottom of the Thonny window (3). It shows the output of code executed the ESP32 device. The user may also directly provide MicroPython commands, which get immediately executed on the device.

Chapter 3

Design and Implementation

3.1 Design

In this section, the most important design decisions and the resulting architecture of the software system are discussed.

3.1.1 Sensor Network (TinyIPFIX)

Figure 3.1 gives an overview of the WSN. The End Devices measure the environment using a sensor and create TinyIPFIX Data Packets that contain the sensed data provided as payload. The created Packets are sent either towards a Concentrator or directly to the Collector. To send, an End Device (i.e., ESP32 devices) passes the data packet to the XBee device over the UART connection, which in turn sends it using the IEEE 802.15.4 protocol. The packet sent towards a Concentrator or the Collector first arrives at the corresponding XBee IEEE 802.15.4 adapter and is then forwarded over UART to the ESP32-based Concentrator or Collector node. IEEE 802.15.4 was selected as the transmission protocol, because it is a widely used standard for indoor environments and has very low power requirements.

TinyIPFIX Packets arriving at a Concentrator are aggregated with other arriving packets and TinyIPFIX messages derived on the Concentrator using its own sensor data.

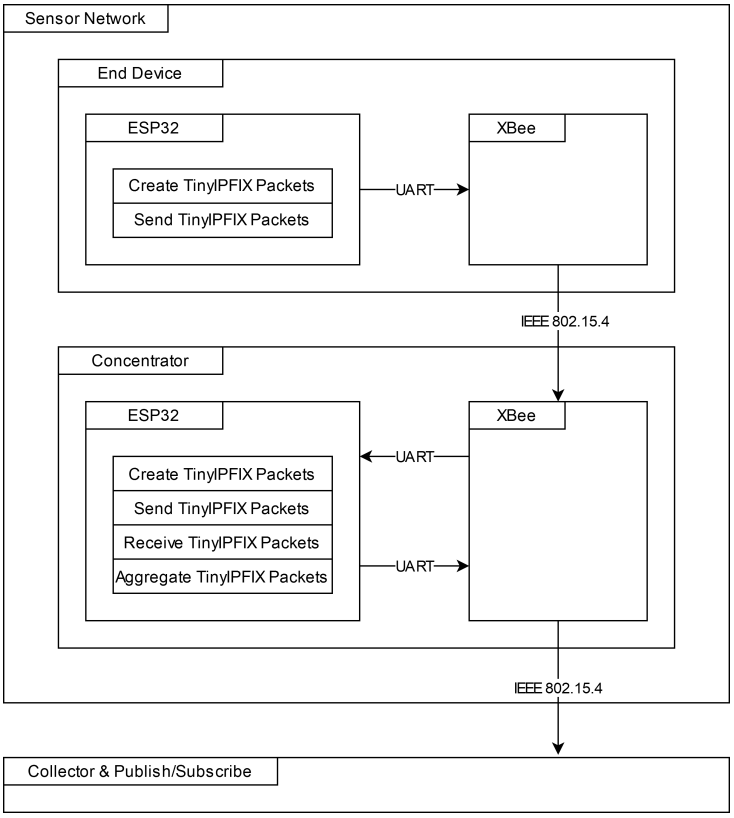


Figure 3.1: Sensor Network

Once the desired number of distinct TinyIPFIX Packets were aggregated, the aggregated Packet gets sent towards the Collector. Moreover, it is also possible to install multiple Concentrators on the way from an End Device to the Collector. To alleviate a problem of indefinitely growing packets, Concentrators have a configurable maximum packet size. If a Packet reaches the maximum size, it is then not aggregated further with any other packets, but instead, it is directly sent to either a next Concentrator or the Collector.

In Figure 3.1, one can observe that all the arrows (each representing a transmission) are always directed from End Device to Concentrator or from Concentrator to Collector but never the other way around (from Collector to Concentrator or from Concentrator to End Device). This is because this implementation of TinyIPFIX is purely push-based and does not implement a pull mechanism to immediately get the latest data. This decision has been made based on the objective to keep it as simple and energy efficient as possible.

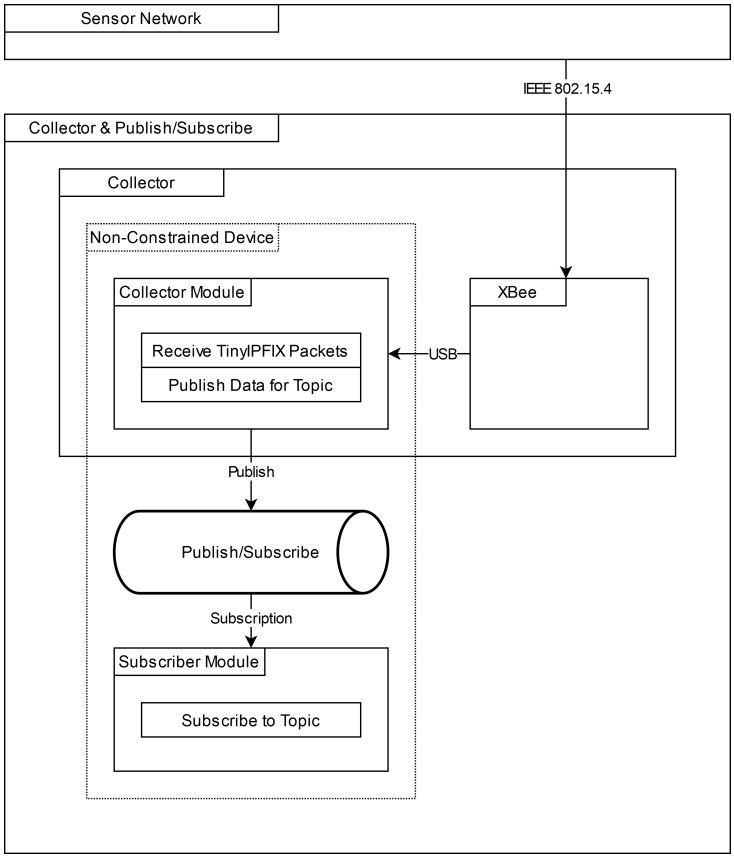


Figure 3.2: Collector and Publish/Subscribe Network

3.1.2 Collector and Publish/Subscribe Network

Figure 3.2 gives an overview of the Collector and Publish/Subscribe Network. While the Sensor Network handles the measurement of the environment and makes sure all the Packets are sent to a central place, the Collector and Publish/Subscribe Network handles the processing of the data that arrived from the sensor network. The Packets from the Sensor Network (cf. Figure 3.1) arrive at the XBee of the Collector. The XBee passes the Packet over USB to the Non-constrained Device that runs the Collector Module. Figure 3.3 shows an XBee that is connected to a Non-constrained Device which runs the Collector Module. All Packets of the whole Sensor Network have to arrive at the Collector in the end.

The Collector Module parses the received Packets. If it is a Packet containing Template Records, then the Template Records that where unknown to the Collector get stored and the ones that were already known are ignored. If it is a Packet containing Data Records,



Figure 3.3: XBee connected to a Non-constrained Device

then the Data is extracted from the Packets using the known Templates. The Data is then shared with other Modules using ZeroMQ's Publish/Subscribe mechanism, i.e., the data is published using the corresponding SetId as the Topic. Other Software Modules can then Subscribe to SetIds they are interested in and process the data further.

The choice to use publish/subscribe makes it very easy to implement a lot of future use cases easily, because a software module can get the desired data with just a few lines of code, without needing to worry about how TinyIPFIX works at all.

For this thesis two subscriber modules were made for testing purposes. One subscriber module just prints all the data to the console that it is subscribed to. The other subscriber module also receives the data for which it has a subscription and then stores it in a MySQL Database.

3.1.3 Test Network of this Thesis

The test Network that was used for this thesis is shown in figure 3.4. It is made up of four End Devices (TinyIPFIX Devices), two Concentrators and one Collector. The

two Concentrators both send their data to the Collector. Two End Devices send their data to one Concentrator, and the other two End Devices send their data to the other Concentrator.

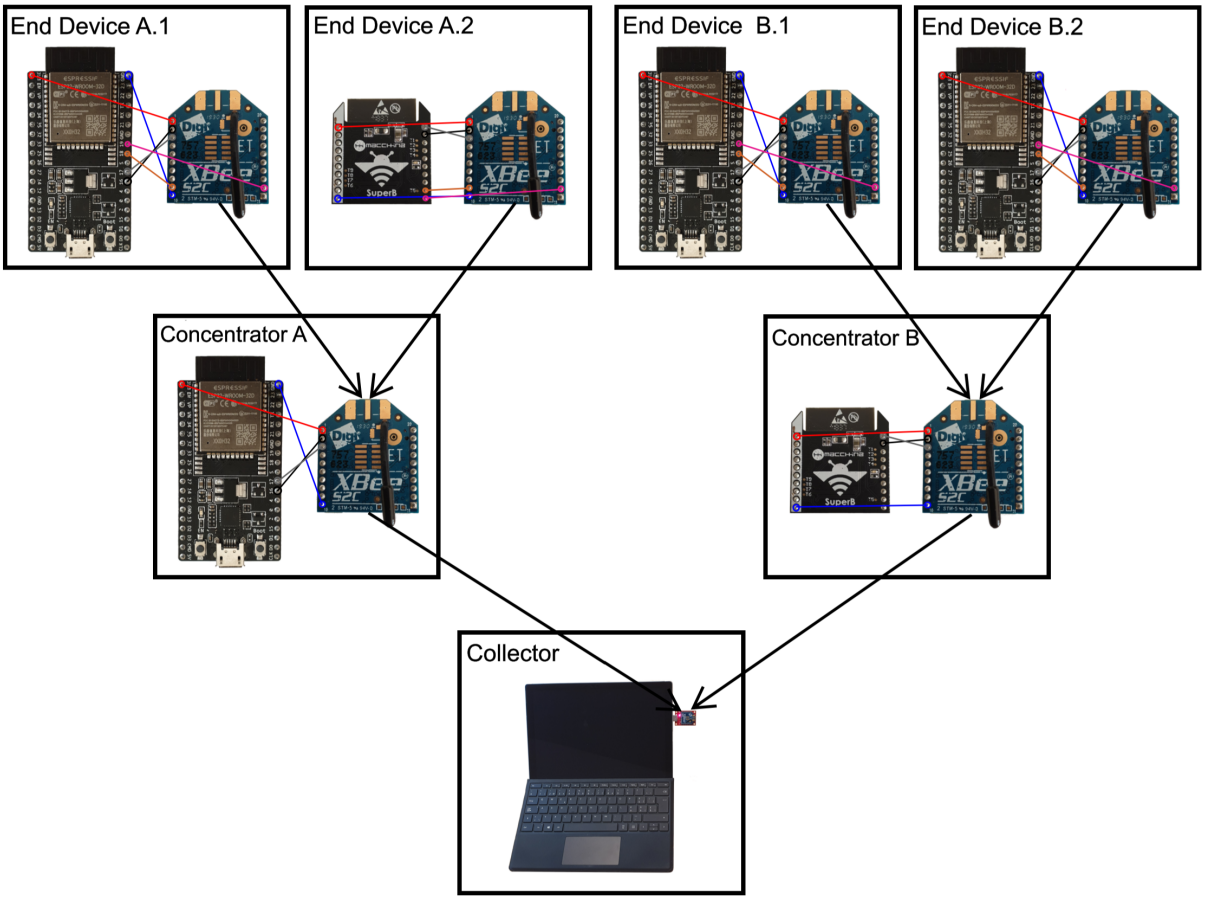


Figure 3.4: Test Network Overview

3.1.4 Use Cases

TinyIPFIX has no means of restoring a Packet that did not arrive at a receiver. For this reason it is not meant for applications where it is crucial that no data is lost. It is rather applicable for use cases where energy efficiency is more important and losing few packets is acceptable.

The two actors in Figure 3.5 are the administrator and the data consumer. On the one hand a monitoring system needs a person who sets the network and the sensors up. In a a company this can be an experienced employee, who can customize the network as the



Figure 3.5: Use Case Diagram

company needs it. In a household this can be a housemate that bought a ready to use set of TinyIPFIX devices for example inside an IoT article that has sensors in it using TinyIPFIX. On the other hand, a monitoring system has someone who is interested in the data that it produces. This can be a company, a private household or even governmental departments.

Companies need different kinds of environment monitoring for many reasons. Manufacturers of drugs sometimes need a monitoring of the temperature of their warehouse for legal reasons. An Ice Cream manufacturer might want a monitoring of temperature and humidity for quality assurance reasons.

For private households a smart carpet was developed that needs to send data from its sensors to a central place. This carpet can sense when someone fell on it and is not able to stand up. The carpet then sends an alarm to emergency services or to relatives of the owner of said carpet, which can save lives in homes where an older person lives alone. But this is only one of many examples that are already in use in private households.

Governmental departments could go even further and develop whole smart cities. Monitoring how much energy each household and company is using at the moment could innovate the energy production. A hydro power plant for example could always produce just as much energy as is actually needed instead of having to plan with big reserves. This would save a lot of otherwise wasted energy. The data could even automatically steer when a reservoir dam needs to be opened or closed. For sure, such a comprehensive monitoring can realistically only be done using cheap and energy efficient smart sensors that are connected in a network.

In all of the above cases energy efficiency and simplicity is crucial, because it makes the realization of such projects quicker and cheaper and the maintenance of the network is easier and therefore cheaper as well.

3.1.5 Class Diagram

Figure 3.6 shows the class diagram that was developed for this thesis. It will facilitate the understanding of the implementation details that follow in the next section. In the middle of the diagram are all the classes that are directly related to the TinyIPFIX Protocol, they are visually grouped in the tinyIPFIX.py group, which will later also be the name of the file that contains all those classes. On the right hand side are the classes that are related to the ESP32 devices (ESP32 DevKitC V4 and SuperB in our case) and the Sensors connected to the ESP32 devices. Those classes (the ones related to the TinyIPFIX Protocol and the ESP32 Devices) will all be implemented in MicroPython. On the left hand side of the diagram are all the classes that are related to the Collector and the processing of the TinyIPFIX messages. Those classes are implemented in regular Python, because they will run on a non-constrained device.

TinyIPFIX Protocol Related Classes

The TinyIPFIX Helper Functions Class contains many useful functions that were often needed for the implementation of the other classes. This class for example has functions that are useful when trying to manipulate single bits of a Python bytes object. For this the bytes object first needs to be converted into a string of bits, then the bits can be altered, and then the string of bits can be easily converted back into a bytes object again, using functions from the TinyIPFIX Helper Functions Class. Although this conversion from bytes to bitstring back to bytes seems complicated, it is actually much easier than manipulating single bits directly in a bytes object. Other useful functions are closer related to TinyIPFIX, for example there is a function that determines whether a TinyIPFIX message is a data or a template message. One function takes a bytearray that embodies

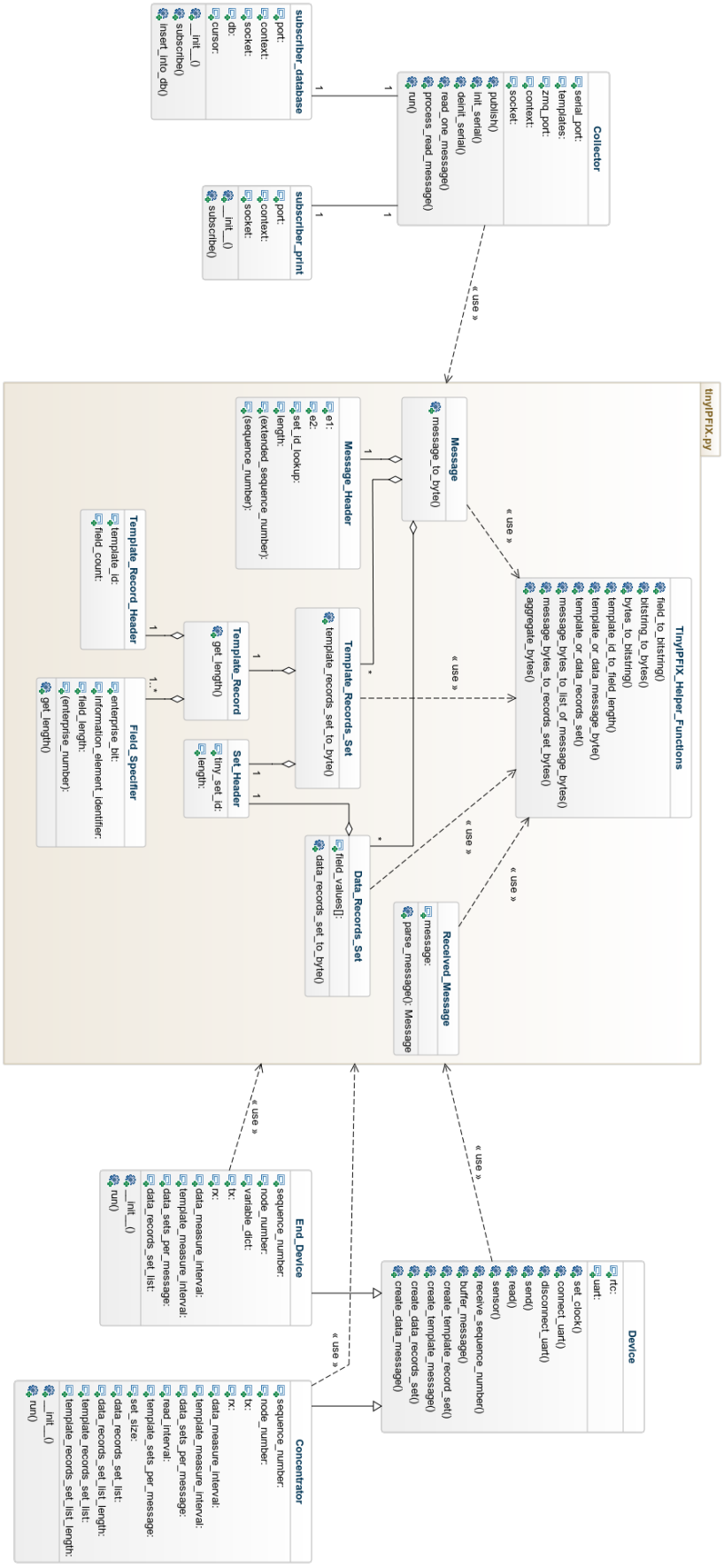


Figure 3.6: Class Diagram

multiple TinyIPFIX messages and returns a list of bytearrays that each embody one TinyIPFIX message.

The rest of the Classes each represent a part of a TinyIPFIX Message. One can see that the Message Class has no attributes at all, but it is built up with these other Classes, which makes it very easy to work with the messages. It is for example very easy to get all the Template Records of a Template Message. The Message Class has one function, which is to convert the Object instance to a bytearray object which can then be sent over the network. One interesting Class is the Received Message Class. This Class is useful when a bytearray, which embodies a TinyIPFIX message, is received. The Received Message Class needs exactly such a bytearray for construction and then has a function which returns a corresponding Message Class instance. This way the bytearray can be converted back to a Message Class at the receiver.

How a TinyIPFIX Message is built up has already been explained in the Related Work Section. For this a useful Figure (Figure 2.4) was given. In the appendix is another very useful Figure that illustrates the navigation properties of the Message Class, this Figure facilitates working with this Class, because many Micropython IDE's will not give you hints for navigation properties. See Figure B.1 in the Appendix.

Device Related Classes

The Device Class has two attributes, one is the RTC which stands for Real Time Clock. The other is the UART, which is the object that is used to send and receive data. The Device Class has many functions that are all concerned about creating, sending and receiving Messages, setting the clock or using the sensors.

The End Device Class and the Concentrator Class both inherit from the Device Class. They both have their own run function with defines what the device does. They both have the data measure interval, template measure interval and the data sets per message attributes. Those attributes are set during creation of the Class. The attributes Data and Template Measure interval define how many seconds pass between creation of a data record set or template record set respectively. The attribute data sets per message defines

how much data record sets a message should contain. For example, if the attribute is set to two, then as soon as the device created two data record sets it packs them into a message and sends them.

The end device does not need an attribute template sets per message, because they always immediately send the template message with the template record set(s) they just created. This is because they always create the same template record sets and not always different one as with the data record sets (otherwise it results in a message that repeats itself). The Concentrator although has an attribute template sets per message, because it also receives templates and does not only export its own predefined templates, which leads to different templates being sent by it.

The Concentrator has another attribute called set size, which defines the maximum number of bytes a message may have before it is sent. This ensures that messages can not get too big before they are sent and is necessary because due to the aggregation of multiple messages into one, the messages may get very big.

The tx and rx attributes present in both the End Device Class and the Concentrator Class define which Pin is used to transmit (tx) data over UART and which pin is used to receive (rx) data over UART.

The Variable Dict attribute from the End Device is used to store variables permanently, which is necessary for End Devices because they use the deepsleep mode whenever they do not need to send or create messages, and in deepsleep mode all the content of the non-permanent storage is lost.

The Node Number is a unique number that is assigned to each ESP32 Device.

The Sequence Number is an integer that is increased by one for every Message that is sent.

The Concentrator has an attribute called Data Measure Interval which defines in which interval the Concentrator should check whether there are new messages to be received.

Collector and Processing Related Classes

The Collector Class has a list of templates as an attribute that stores all the templates that are known to it. It has a serial port, that is used to talk to the XBee connected to it. Further it has a zmq port, a context and a socket that are used to publish data. To publish data it uses the publish function that uses the above mentioned attributes. The function Read One Message reads a message using the serial port and the function process read message then publishes the data if a data message was read or adds a template to the known templates if it was a template message containing a yet unknown template.

The Subscriber Database Class subscribes to published messages from the Collector Class and then stores them in a MySQL database.

The Subscriber Print Class subscribes to published messages from the Collector Class and then prints them on the console.

3.2 Implementation

This Section will explain the implementation of the TinyIPFIX Protocol and the data processing in detail. The code was also uploaded to GitHub [28].

3.2.1 Hardware Setup

The first thing that needs to be done before the start of the implementation is the setup of the hardware infrastructure such that it can be programmed and is suitable for the desired functionality.

Flashing MicroPython onto the ESP32-based devices

The ESP32-based devices do not understand MicroPython when they leave the factory. The first thing needed is, therefore, to flash them with the MicroPython firmware. There are many guides on the Internet on how to accomplish this goal. For example, [6]) provides

a large number of details, but it was still complicated to discover a method that works for a wide variety of devices. Especially, flashing a SuperB device is a challenge.

At first MicroPython has to be downloaded [9]. Then the device needs to be connected to a computer via USB. The ESP32 DevKitC V4 features a USB port on its own, for the SuperB the Sparkfun XBee Explorer Dongle was used to connect it to a computer. Then the ESP32 based devices need to be put into download mode in order for them to accept the flashing. On the ESP32 DevKitC V4 there are two buttons on the board, one of them is labeled ”EN” and one of them is labeled ”Boot”. Is the ”EN” button pressed then the device restarts. To put the ESP32 DevKitC V4 in download mode the ”Boot” button must be pressed and held, then the ”EN” button must be pressed and released, and then the ”Boot” button can be released. Putting the SuperB into download mode is a little more complicated, because the SuperB does not feature any buttons to press.

The SuperB does feature 10 pins. To understand how the SuperB can be put into download mode it is important to understand how those pins are connected with the ESP32-WROOM-32. For this the Schematic of the SuperB is useful. Figure A.1 in Appendix A is a copy of the schematic of the SuperB released by the manufacturer. To put the SuperB into download mode, one has to do a similar thing as for the ESP32 DevKitC V4. The ”IO0” Pin has to be grounded (also called pulled-low or given a value of zero), then while keeping ”IO0” grounded, apply ground and remove ground from the ”EN” pin and then remove the ground from the ”IO0” pin. Then the SuperB should be in download mode. To ground a pin, the pin labeled ”GND” may be connected to the ground lane of a breadboard using a jumper cable (this makes the ground lane actually grounded), then using another jumper cable a connection can be made between another pin slot of the same ground lane of the breadboard to the pin that should be grounded. Figure 3.7 shows a SuperB that has its power pin (labeled ”+3.3V” in the Schematic) connected to the power lane on the breadboard, its ground pin (labeled ”GND” in the Schematic) connected to the ground lane of the breadboard and its ”IO0” also connected to the ground lane on the breadboard. This means that pin ”IO0” is grounded in Figure 3.7.

On the SuperB Schematic, it can further be seen that each pin of the SuperB has a label. To find out more about the pins, the name of that label can be found in the Pin

Description section in the ESP32-WROOM-32D Datasheet [25]. For example, this can be helpful for choosing which pin to use for a connection.

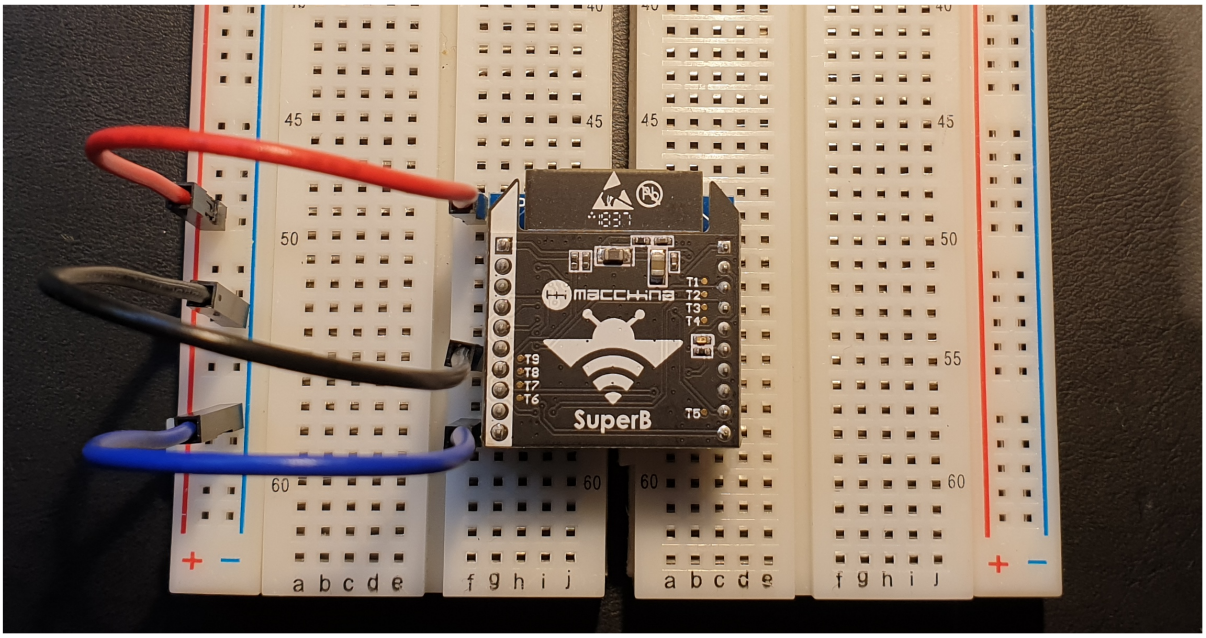


Figure 3.7: SuperB with grounded ”IO0” pin

To check whether the ESP32 based device is successfully put into download mode, a program called putty can be used for example. If putty is installed on the system, it can be used to listen to the ESP32-based device using the following command on the command line on Windows. The ”X” in ”COMX” has to be replaced with the corresponding port number of the ESP32:

```
1    putty COMX -serial -sercfg 115200,8,n,1,N
2
```

Listing 3.1: Listen to ESP32 using putty on windows

The following command can be used in the command line to list all the devices with their COM ports, to find the port number of the ESP32 on windows:

```
1    mode
```

Listing 3.2: List of COM ports on windows

To use putty to listen to the ESP32 based devices on Linux, this command can be used on the terminal; the ”X” in ”ttyUSBX” has to be replaced with the port number of the ESP32:

```
sudo putty /dev/ttyUSBX -serial -sercfg 115200,8,n,1,N
```

Listing 3.3: Listen to ESP32 using putty on Linux

Once putty is listening to the ESP32 based device, the above steps can be done to put the ESP32 into download mode and it can be observed if it is working.

Once the ESP32 based device is in download mode, the MicroPython firmware can be flashed. Note that for this purpose, putty needs to be disconnected again, such that the port is free. It is recommended to first erase the flash that is on the ESP32 based device before flashing. To erase the flash and to flash the device, a python package called esptool.py [27] is recommended. Esptool.py can be installed on the command line using the following command:

```
pip install esptool
```

Listing 3.4: Install esptool.py

To erase the flash the ESP32-based device, the following command can be used on the command line on windows; the ”X” in ”COMX” needs to be replaced with the port number of the ESP32:

```
esptool.py --port COMX erase_flash
```

Listing 3.5: Erase flash on the ESP32

To flash the ESP32 with the MicroPython firmware, the directory that contains the downloaded MicroPython firmware file has to be entered. This command can be used to change the directory and adjust to an appropriate path:

```
cd C:\Users\Username\Desktop
```

Listing 3.6: Change path in CMD

Then, use the command below to flash the firmware; the ”X” in ”COMX” has to be replaced with the port number of the ESP32 and ”esp32-idf4-20200902-v1.13.bin” has to be replaced with the filename of the MicroPython firmware file:

```
esptool.py --chip esp32 --port COMX write_flash -z 0x1000 esp32-idf4
-20200902-v1.13.bin
```

Listing 3.7: Flash MicroPython onto the ESP32

Now, the ESP32 based device may be programmed using MicroPython. MicroPython commands can be typed in using putty like above, or even better using an IDE like Thonny. In Thonny, it is not only possible to type in commands, but also to store code files on the ESP32 device. Those code files may then be re-used and are not "forgotten" by the ESP32 device like the commands are that are just typed into the console. Code that is stored on the ESP32 in the main.py file gets automatically executed every time the ESP32 device starts.

Connecting the ESP32 based device with the XBee

In this work UART was used to connect the ESP32 based device with the XBee. To this end, two cables need to connect the two devices. Additionally, two cables are used to power the XBee and to ground the XBee using the ESP32's power and ground pins. If the device should be used as an End Device, then two additional cables need to be used to control when the XBee should be awake and when it can sleep.

Figure 3.8 gives an overview of the pins that were used for this thesis. Figure 3.4 also shows how the pins of the devices were connected. In attachment C zoomed in parts of Figure 3.4 can be found.

For the Digi XBee all the marked pins need to be used the way they are labeled. For the ESP32 based devices the pins labeled power and ground also need to be used for ground and power respectively, but the other pins (TX, RX, Clear to Send (CTS)-Xbee, Wake-up-xbee) can also be replaced with other general purpose input output (GPIO) pins, this then needs to be reflected in the program code.

To connect the ESP32 based device with the XBee, the ESP32 TX pin needs to be connected to the XBee RX pin and the ESP32 RX pin needs to be connected with the XBee TX pin. The ESP32 based devices are powered and grounded via USB or via a

battery that is applied to it. To power and ground the XBee, the pins from the ESP32 can be used, for this the ESP32 power pin needs to be connected with the XBee power pin and the ESP32 ground pin needs to be connected with the XBee ground pin. If the devices should be used as an End Device, then the wake-up-xbee pin from the ESP32 based device needs to be connected with the sleep-control pin of the XBee and the CTS-XBee pin from the ESP32 based device needs to be connected with the CTS pin from the XBee.

If the Wake-up-Xbee pin transitions from high to low, then the XBee wakes up (for this the XBee has to be configured accordingly, but more on that later). The XBee then is awake, and if it is idle (does not receive data over UART in our case) for a predefined time then it goes to sleep again automatically.

The CTS-XBee pin has a low value when the XBee is woken up and ready to send data. This pin was used to check that the XBee has fully woken up, before data was sent to the device over UART.

Once above steps are done it is possible to let the ESP32 based device talk to the XBee and vice versa over UART.

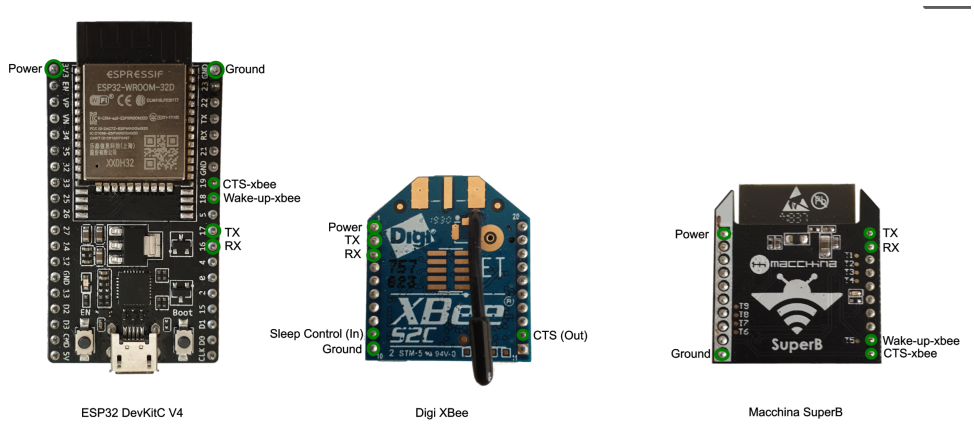


Figure 3.8: Pin Overview

Configuring the XBee's

In the previous subsection it was discussed how a ESP32 based device can be connected with an XBee, now it needs to be discussed how to configure different XBee's such that

they can talk to each other. This ultimately makes it possible that one ESP32 based device can send data to another ESP32 based device. In our case for example an End Device needs to send data to a Concentrator.

To configure the XBee's a software released by the manufacturer of the XBee, called XCTU is used. This software does nothing else than send AT commands to the XBee, but it gives a nice GUI and explanations for all options, which makes it easier to use XCTU than to send AT commands over the command line.



Figure 3.9: Digi XCTU Software

Figure 3.9 shows the GUI of XCTU when a XBee is loaded and selected. To load an XBee that is connected to the computer in XCTU, the button with the loupe symbol can be clicked or CTRL+SHIFT+D can be pressed. When the "Update" button is pressed, then the function set that the loaded XBee should run on can be chosen. For this thesis the

function set 802.15.4 TH with the firmware version 2003 was used.

The XBee network has one XBee Coordinator that manages the network and multiple XBee End Devices. For this thesis the XBee of the Collector was chosen to be the Coordinator and the rest of the XBee's are XBee End Devices. The reason for this decision is that the Coordinator of the XBee network is a single point of failure for the whole network. It is therefore desirable to be able to quickly replace a defect Coordinator and it is easier to replace a defect XBee when it is connected to a computer than if it is connected to a ESP32 that might be deployed somewhere difficult to reach. The Collector of the TinyIPFIX network is itself a single point of failure, so choosing the TinyIPFIX Collector to be the XBee Coordinator leads to one single point of failure instead of otherwise two, which is also very desirable.

Many setting can be changed for the XBee, but particularly interesting for this work are the following:

- CH (Channel) - all the devices need to use the same channel
- ID (PAN ID) - all the devices need to use the same Personal Area Network (PAN) ID; it is recommended to change the value from the default. For this work 7385 was randomly chosen for all the devices.
- DH (Destination Address High) - Represents the upper 32 bit of the destination address. The destination address defines to which address packets are sent. For this thesis, a value of zero was chosen for the Destination Address High, this in combination of choosing a value smaller than 0xFFFF for the Destination Address Low has the effect that a 16 bit address is used for transmission instead of a 64 bit one.
- DL (Destination Address Low) - Represents the lower 32 bit of the destination address or the only 16 bit of the destination address if the Destination Address High is set to zero and the Destination Address Low is set to a value smaller than 0xFFFF.

- MY (16-bit Source Address) - Represents the 16 bit source address, that is used when a 16 bit address is used for packet addressing. Other XBee's can set their Destination Address Low to the value of this MY address to send packets to this XBee.

- CE (Coordinator Enable) - Defines whether an XBee runs in Coordinator Mode or in End Device Mode.

- MM (MAC Mode) - Here it can be set whether acknowledgements for the transmission should be used or not.

- SM (Sleep Mode) - The Collector and the Concentrator XBee's are set to "No Sleep", the End Device's XBee's are set to "Cyclic Sleep with Pin Wakeup". In this mode the XBee's go to sleep when they are idle for a time, and can be woken up using a pin.

- D7 (DIO7 Configuration) - This option is set to disabled for the Collector and Concentrators XBee's and set to "CTS flow control" for the End Device's XBee's. If set to CTS flow control, then this pin can be used to find out whether the XBee is ready to send data.

Table 3.1 shows the MY (source) address and DL (destination low) address of each XBee. The DL address of the Collector's XBee is a randomly chosen unused address, because the Coordinator never sends anything. The MY address of the End Devices are not important as well, because they never receive anything. (In a future work a pull mechanism might be implemented, then the MY address of the End Devices becomes important and the DL address of the Collector's XBee can be set to 0xFFFF which makes a broadcast). One can see in the table for example that End Device A.1 sends to the address A00A, which is the MY address of Concentrator A. Compare with Figure 3.4 which visualizes which device sends to which device.

Device Name	MY	DL
Collector	C001	E99E
End Device A.1	E0A1	A00A
End Device A.2	E0A2	A00A
End Device B.1	E0B1	A00B
End Device B.2	E0B2	A00B
Concentrator A	A00A	C001
Concentrator B	A00B	C001

Table 3.1: XBee MY and DL addresses

3.2.2 Implementation of the TinyIPFIX Protocol

All the Classes that are related to the TinyIPFIX Protocol were placed in file tinyIPFIX.py. The next subsection will show what those classes do, by discussing how they are used by the devices. For further details the concrete implementation on the GitHub repository [28] can be looked at.

3.2.3 Implementation of Device Related Classes

For the implementation of the devices, a parent class named device and two child classes named Concentrator and end_device ware made. The device class handles the reading of sensor values, the connection to the XBee (reading and sending data) and the creation of TinyIPFIX messages using the sensor values. Listing 3.8 shows the functions from the device class that are used to create TinyIPFIX messages. Firstly, there are the two functions create_template_record_set and create_template_message that are responsible for creating the template message that contains meta-information about the data messages. Then the two functions responsible for the creation of data messages follow. One can see how a TinyIPFIX Message is built up using different classes that represent the different components of the message in the protocol.

```
1 def create_template_record_set(self):
2
```

```
    # create field specifier 1 - Node Number
    enterprise_bit = 1
    information_element_identifier = 1
    field_length = 2
    enterprise_number = 2882382797
    field_specifier1 = Field_Specifier(enterprise_bit,
   information_element_identifier, field_length, enterprise_number)

    # create field specifier 2 - temperature
    enterprise_bit = 1
    information_element_identifier = 2
    field_length = 4
    enterprise_number = 2882382797
    field_specifier2 = Field_Specifier(enterprise_bit,
   information_element_identifier, field_length, enterprise_number)

    # create field specifier 3 - timestamp
    enterprise_bit = 1
    information_element_identifier = 3
    field_length = 19
    enterprise_number = 2882382797
    field_specifier3 = Field_Specifier(enterprise_bit,
   information_element_identifier, field_length, enterprise_number)

    # create template record
    template_id = 129
    template_record = Template_Record(template_id, field_specifier1,
   field_specifier2, field_specifier3)

    # create template record set
    template_record_set1 = Template_Records_Set(template_record)

    return template_record_set1

def create_template_message(self, *template_record_sets):
    # create template message
    e1 = 0
    e2 = 0
```

```
37      sequence_number = self.receive_sequence_number(e2)
38      message = Message(e1, e2, sequence_number, *template_record_sets)
39      template_message_bytes = message.message_to_byte()
40
41      return template_message_bytes
42
43  def create_data_records_set(self):
44      field_value_1 = ustruct.pack('>H', self.node_number)
45      field_value_2 = ustruct.pack('>f', self.sensor_temperature())
46      field_value_3 = self.sensor_get_timestamp().encode("utf-8")
47      tiny_set_id = 129
48      data_records_set = Data_Records_Set(tiny_set_id, field_value_1,
        field_value_2, field_value_3)
49      return data_records_set
50
51  def create_data_message(self, data_records_set_list):
52      e1 = 0
53      e2 = 0
54      sequence_number = self.receive_sequence_number(e2)
55      data_message = Message(e1, e2, sequence_number, *
        data_records_set_list)
56      data_message_bytes = data_message.message_to_byte()
57      return data_message_bytes
```

Listing 3.8: Functions to create TinyIPFIX Messages from the device class

In listing 3.9 the implementation of the run function from the Concentrator class is depicted.

The class variables data_measure_interval, template_measure_interval and read_interval hold the values of how many seconds should pass between two data measures, two template creations and two UART reads respectively. The read interval should not be too big, such that no packets get lost, what is too big depends on how many devices send data to that Concentrator.

The variables data_i, template_i and read_i hold the values of in how many seconds the next data measure, template creation or UART read should happen. Initially template_i

is set to zero, this ensures that a template is sent before data is sent, which ensures that the data can be decoded at the receiver. If data_i, template_i and read_i are all greater than zero, then the device does not need to do anything and is set to sleep until one of them reaches zero. When one of them reaches zero then the device needs to for example make a data measurement.

When a measurement was done, the device checks whether it needs to send the message or whether it first can do another measurement. This is configured in variables containing the maximum size a message may reach and how many record sets may be in it. Whenever a data message should be sent and the template_record_set_list is not empty, a template message is sent first that contains all the template records sets that are currently in the template_record_set_list, this ensures that never data is sent before the corresponding template is sent that contains the information on how to decode the data.

```
def run(self):
    helper = TinyIPFIX_Helper_Functions()
    template = self.create_template_record_set()
    template_bytes = template.template_records_set_to_byte()

    data_i = self.data_measure_interval
    template_i = 0
    read_i = self.read_interval

    while True:
        sleep_interval = min(data_i, template_i, read_i)
        sleep(sleep_interval)

        template_i -= sleep_interval
        if template_i == 0:
            template_i = self.template_measure_interval
            self.template_records_set_list.append(template_bytes)
            self.template_records_set_list_length += len(template_bytes)

        data_i -= sleep_interval
        if data_i == 0:
            data_i = self.data_measure_interval
            data = self.create_data_records_set()
```

```
24              data_bytes = data.data_records_set_to_byte()
25              self.data_records_set_list.append(data_bytes)
26              self.data_records_set_list_length += len(data_bytes)
27
28          read_i -= sleep_interval
29          if read_i == 0:
30              read_i = self.read_interval
31              read = self.read()
32              if read is not None:
33                  self.buffer_message(read)
34
35          if( (len(self.data_records_set_list) >=
36              self.data_sets_per_message) or
37              (self.data_records_set_list_length >=
38              self.set_size) ):
39              if len(self.template_records_set_list) > 0:
40                  template_message_bytes = helper.aggregate_bytes(
41                      0, 0, self.receive_sequence_number(0),
42                      *self.template_records_set_list)
43                  self.send(template_message_bytes)
44                  self.template_records_set_list = []
45                  self.template_records_set_list_length = 0
46              data_message_bytes = helper.aggregate_bytes(0, 0,
47                  self.receive_sequence_number(0),
48                  *self.data_records_set_list)
49              self.send(data_message_bytes)
50              self.data_records_set_list = []
51              self.data_records_set_list_length = 0
52
53          elif( (len(self.template_records_set_list) >=
54              self.template_sets_per_message) or
55              (self.template_records_set_list_length >=
56              self.set_size) ):
57              template_message_bytes = helper.aggregate_bytes(
58                  0, 0, self.receive_sequence_number(0),
59                  *self.template_records_set_list)
60              self.send(template_message_bytes)
61              self.template_records_set_list = []
```

```
62              self.template_records_set_list_length = 0
```

Listing 3.9: Run function from the Concentrator class

The implementation of the run function from the end_device class is similar to the run function from the Concentrator class seen above. The big differences are that the end_device only creates and sends messages and does not read and aggregate messages and that the end_device class has a deepsleep functionality which requires some parameters to be stored permanently even if the device is powered off.

The main.py file that needs to be placed on the Concentrator to make it run the program is depicted in listing 3.10.

```
1  from concentrator import Concentrator
2
3  my_concentrator = Concentrator('esp32',
4      node_number = 4,
5      data_measure_interval = 60,
6      template_measure_interval = 600,
7      data_sets_per_message = 10,
8      read_interval = 2.5,
9      template_sets_per_message = 5,
10     set_size = 524287)
11
12 my_concentrator.run()
```

Listing 3.10: Main.py from the Concentrator

3.2.4 Implementation of Collector and Processing Related Classes

Three Classes were created named Collector, Subscriber_Database and Subscriber_Print. Like noted earlier, these three classes are implemented using Python as a programming language, while all the other classes of this thesis are implemented using MicroPython.

The Collector class receives and decodes all the messages from the whole network. For the decoding of the data messages it keeps a list of template messages that are needed for

the decoding. Then it publishes the messages using the SetId as the topic. For this the Publish/Subscribe mechanism from ZeroMQ is used.

The two classes Subscriber_Database and Subscriber_Print then subscribe to the topic they are interested in and receive the messages based on their subscription. The Subscriber_Database class stores all the data that it receives in a database. The Subscriber_Print class prints all the data that it receives to the console.

Chapter 4

Evaluation

For the evaluation of the implementation of TinyIPFIX a test network was made, like the one described in Subsection 3.1.3.

4.1 Overhead

In this section the overhead of a transmission in TinyIPFIX is going to be compared with the overhead of a simple type-length-value approach like [29] did in the evaluation of his implementation of TinyIPFIX. With type-length-value, a message is simply built up by having one or several times the fields type, length and value. A message could for example be type1, length1, value1, type2, length2, value2. The fields type and length have a fixed length and the field value has a variable length defined in the field length.

For comparability this work takes the assumption that the type field of the type-length-value approach can hold as many different types as TinyIPFIX can hold different information elements. TinyIPFIX can support 2^{15} IANA predefined Information elements, and each of the 2^{32} possible companies (identified by the enterprise number) can define 2^{15} information elements. Under the assumption that only one enterprise number can be used for the network, TinyIPFIX could support 2^{16} different information elements. This results in the type field having two bytes in order to also support 2^{16} different types.

Further the assumption is taken that the length field is one byte long, because that is the length of the corresponding length field in TinyIPFIX.

In type-length-value, the overhead can be calculated by the following formula, the len(field) function hereby returns the length of the field:

$$\frac{len(type) + len(length)}{len(type) + len(length) + len(value)} \tag{4.1}$$

For the type-length-value approach, the resulting formula is the following

$$Overhead = \frac{2+1}{2+1+len(value)} = \frac{3}{3+len(value)} \tag{4.2}$$

The overhead of TinyIPFIX is illustrated in figure 4.1

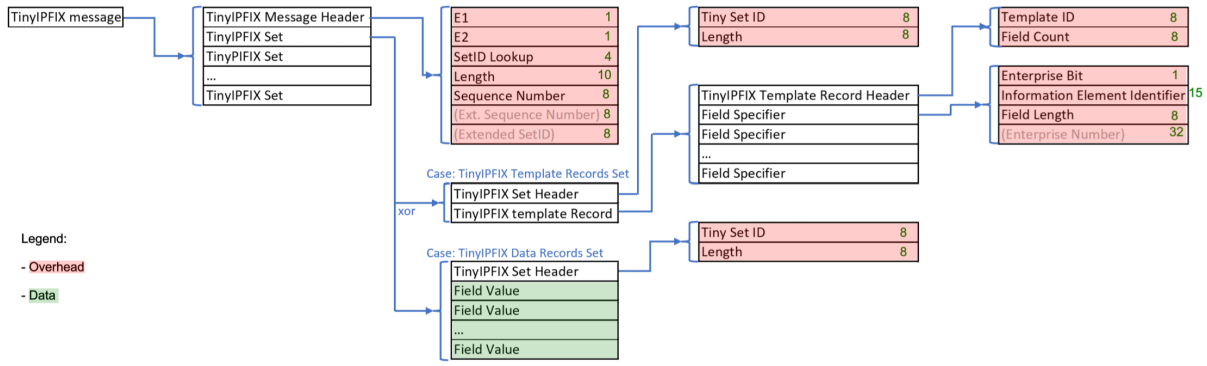


Figure 4.1: TinyIPFIX Overhead

The assumption is taken that a template message always only contains one template definition, the enterprise number is present and the extended sequence number and extended SetId are not present. Then a template message contains 7 + (number of Field Specifiers) × 7 bytes of overhead.

A data message contains 3 bytes + (number of data record sets) × 2 bytes of overhead and a variable amount of data.

For simplicity, the assumption is taken that always a 4 byte float sensor value (this could be any sensor value, for example a temperature value) and a timestamp for that sensor value need to be sent. Further the assumption is taken that 5 bytes are needed for the timestamp, which is the the storage amount that MySql uses for their Datetime data value [12].

Under the taken assumptions, a Type-Length-Value Packet has 3 bytes of overhead per value (two for the type field and one for the length field). A 4 byte value and a 5 byte value are sent. Therefore, in total 6 bytes of overhead and 9 bytes of data are sent, which results in 40% overhead.

A TinyIPFIX Packet has 21 bytes of overhead if it is a template message with two field specifiers. It has 3 + (number of data record sets) × 2 bytes of overhead and (number of data record sets) × 9 bytes of data if it is a data message. The overhead of TinyIPFIX is dependent on two remaining parameters that specify the number of data record sets per data message and how many data messages are sent until a template message is sent again.

Figure 4.2 shows the overhead of TinyIPFIX compared to the Type-length-value approach depending on the two remaining parameters of TinyIPFIX for which no assumptions were taken.

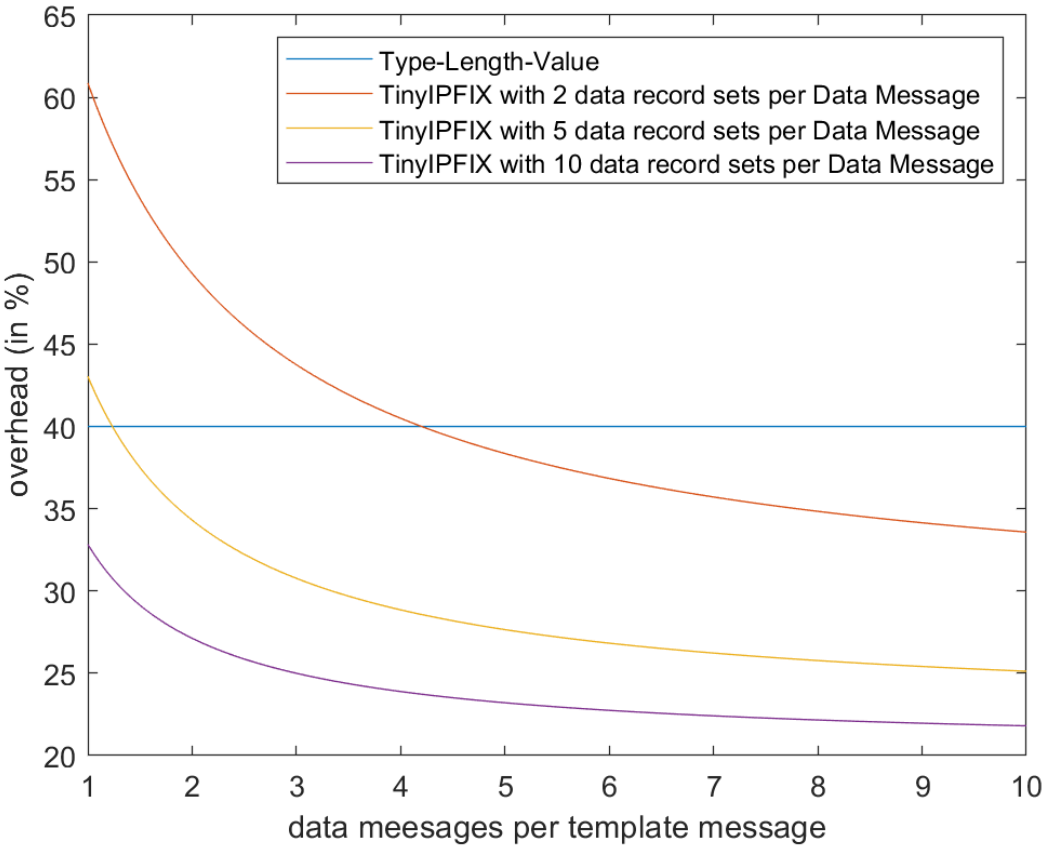


Figure 4.2: TinyIPFIX and Typle-Length-Value Overhead Comparison

It can be seen how fast TinyIPFIX can outperform Type-Length-Value. It is very reasonable that a template can be re-sent after ten data record sets or more have been sent, as this would in the very worst case lead to a loss of only ten values, which could even be decoded later once the template arrives if such a mechanism is implemented. It is also very reasonable that a TinyIPFIX data message contains two data record sets or more. It can be seen that even in the case of having only two data record sets per data message and re-sending a template for every five data messages, TinyIPFIX with an overhead of 38.4% already outperforms Type-Length-Value with an overhead of 40%. Of course also 10 data record sets per data message and a template for every 5 data messages can be sent, in this case tinyIPFIX has an overhead as small as 23.2%.

4.2 Transmission reliability

To measure the transmission reliability (i.e. how many data record sets arrive at the Collector that were being sent by an End Device or Concentrator) some measurements were done where the test network ran for an hour and it was noted how many data record sets arrived at the Collector from the End Devices and how many from the Concentrators and how many were sent by the devices in the WSN, and therefore should have arrived at the Collector. Multiple measurements were done where two parameters were altered between measurements. One parameter was the distance between devices. A measurement was done where the End Devices were placed in a distance of one meter to the Concentrators and the Concentrators in a distance of one meter to the Collector. Then further measurements were done where the distance between devices was 2, 5, 10 and in the last measurement 20 meters. The second parameter was whether the XBee's used acknowledgments or not. All the devices in the WSN were configured to create a data record set every 10 seconds and send a data message as soon as two data record sets were created. Figure 4.3 shows the results of the measurements.

Overall, it can be observed that most of the data record sets that were sent were also received. The only measurement that gave disappointing results was the one taken over five meters. Unfortunately, no clear explanation could be found. One possible reason

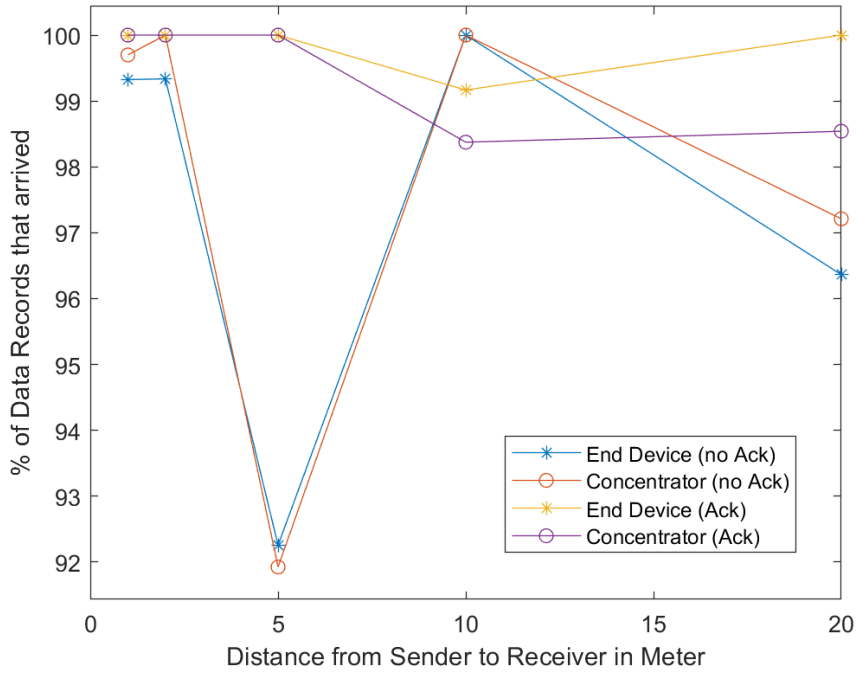


Figure 4.3: Transmission reliability

could be that some other device was sending on a similar frequency at the moment. It can also be observed that the data messages sent by the Concentrator were received more reliably, which was to be expected because the data messages of the End Devices have to be sent twice (once by the End Device and once by the Concentrator) until they reach the Collector. Furthermore, it can be observed that placing the devices in a bigger distance to each other does not greatly reduce the transmission reliability when acknowledgments are used.

These measurements also confirm that the implementation of TinyIPFIX was successful. All the data record sets were successfully created and all the ones that were received were decoded and stored in the database. The Concentrator works in the desired way as well and aggregates the defined number of messages into one message while making sure that never a data message is sent before the according template message was sent.

4.3 Energy consumption

A power meter was used to measure the energy consumption of the devices in different configurations. The power meter plugs into a USB port which powers the meter. A device can be connected to the power meter also over USB. The power meter then shows how much energy was used in milliwatt (mW) and over which period of time.

Table 4.1 shows the energy consumption different devices have in different states. In the table ESP32 refers to the ESP32 DevKitC V4 while SuperB refers to the Machhina SuperB. When the End Devices are in Deepsleep mode, the XBee of the End Device automatically enters sleep mode as well. For the sending mode the ESP32 based device was configured to send as much data as possible to the XBee over UART, and the XBee sends the data then over IEEE 801.15.4. For the receiving mode data was read in an infinite loop that was sent by a device in sending mode.

The table shows that SuperB uses less energy than the ESP32 DevKitC V4. The End Devices save very much energy compared to the Concentrators, because they can go into Deepsleep mode and put the XBee to sleep mode as well. For the Concentrators Deepsleep mode is not possible because they do not only need to send messages, but they also need to receive them, which is not possible while in Deepsleep mode. The table shows that the XBee uses very little energy while in sleep mode (ESP32 without XBee uses nearly the same amount of energy as the ESP32 End Device).

	Deepsleep	Idle	Sending	Receiving
ESP32 End Device	35 mW	190 mW	344 mW	-
SuperB End Device	20 mW	158 mW	308 mW	-
ESP32 Concentrator	-	328 mW	334 mW	390 mW
ESP32 without XBee	35 mW	192 mW	-	-
XBee without ESP32	-	168 mW	-	168 mW

Table 4.1: Energy Consumption based on Device Type and State

According to [22], the transmit current of the XBee is 120 mA, while the idle and receive current are both 32 mA, which is about the same value that was measured in this work. The voltage that the Devices were powered with while they were running was 3.3V. Although the Deepsleep mode saved very much energy, the measurements of the devices in deepsleep mode was still surprising, because the expected value was less than one milliampere. The ESP32 Series Datasheet [26], that describes the Deepsleep mode of devices that are closely related to the ESP32 DevKitC V4, notes a energy consumption of 260 microampere for the described devices in deepsleep mode. Unfortunately, no documentation that describes the energy consumption of the ESP32 DevKitC V4 in particular could be found. Hence, in a future work the implementation of the deepsleep function could be improved, such that more energy could be saved and the impact of the energy efficient protocol could be increased.

Figure 4.4 shows the energy consumption of different devices depending on how many messages they send per hour. Those measurements were taken, because they show the impact that this important configuration (messages per hour) has on energy consumption. Some devices were configured to use acknowledgments for transmission and some did not use acknowledgments. In the figure one can see that sending more messages greatly increases energy consumption for End Devices while for the Concentrators it does not make a big difference. This has to do with the fact that the End Devices are in deepsleep mode when they don't have to do anything while the Concentrators are only in idle mode, such that they don't loose any messages that are sent to them. To be more precise, one reason that sending a message has a bigger impact on the energy consumption of the End Devices than on the energy consumption of the Concentrators is that the difference of the energy consumption between the deepsleep mode and the sending mode is bigger than the difference of the energy consumption of the idle mode and the sending mode. The other reason that sending a message has a bigger impact on the energy consumption of the End Devices than on the energy consumption of the Concentrators is that the End Devices must first wake up from deepsleep and initialize themselves again when they want to send a message, which also uses some energy, while the Concentrators are already running and initialized. It can further be observed that it does not really matter whether acknowledgements were used or not. All in all, it can be observed that the energy used

mainly depends on the time that the devices are in sleep or deepsleep mode compared to the time they are awake. While the implementation of TinyIPFIX is considered a success, the configuration of the devices needs to be improved, such that they use less energy.

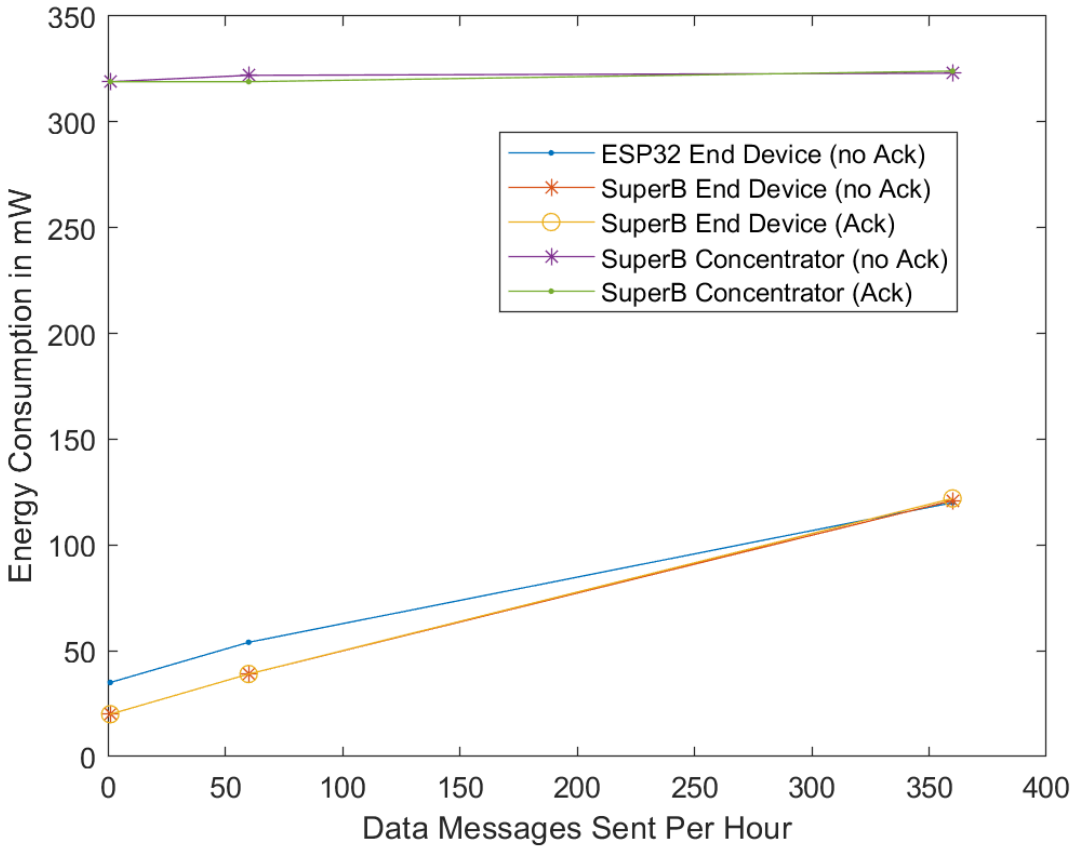


Figure 4.4: Energy Consumption based on message sending interval

Chapter 5

Summary, Conclusions and Future Work

In this thesis, TinyIPFIX was implemented for Espressif ESP-WROOM-32D. Two devices that both feature the ESP-WROOM-32D module were chosen as a hardware platform, namely the Espressif ESP32 DevKitC V4 and Machhina SuperB. The devices were prepaired for programming using MicroPython. To this end, they were flashed with an appropriate firmware. Then each ESP device was connected with a Digi XBee board, which features the IEEE 802.15.4 standard allowing for low power communication among devices.

Different design decisions had to be taken before starting the implementation and a deep understanding of the TinyIPFIX protocol was developed. Not only the core idea of TinyIPFIX, sending metadata and data in separate messages to decrease overhead and save energy, but also how each message should be built up and what meaning each bit in a message has had to be understood. Further the different kinds of devices in a TinyIPFIX network were discovered using a diagram, like the TinyIPFIX End Device and the TinyIPFIX Concentrator and it was shown how all the TinyIPFIX Messages are collected and processed by the TinyIPFIX Collector.

The implementation of TinyIPFIX was done and a link to GitHub was given where a complete implementation may be witnessed. In the thesis, the most important classes

implemented were shown and explained.

Lastly, the project was evaluated. It could be shown that TinyIPFIX has a smaller overhead than the type-length-value approach. Further it has been shown that the implementation of TinyIPFIX is working in a test network. The energy consumption of the devices running TinyIPFIX has been evaluated and the conclusion was made that the main method to reduce energy consumption is to leave them in deep sleep (ESP32 based devices) or sleep (XBee) mode for as long as possible, which is more dependent on the configuration of the devices themselves than on the protocol running on them. The configuration of the devices needs to be improved to really benefit from the smaller overhead that is reached with TinyIPFIX.

Moreover, the implementation of TinyIPFIX can be still improved. For example, it was observed in the test network that although all devices were configured to perform data measurements in the same interval, some devices made measurements a bit more often than others over a long period. Furthermore, the ergonomic of the system could be significantly improved by creating a GUI to configure the devices.

In future, different improvements might be possible. One idea could be a time synchronization across all devices in order to allow for putting the Concentrators into the deep sleep mode and wake them up on time when a message from a End Device arrives. It would also be beneficial in terms of energy consumption, when the Concentrator could be put into deep sleep already for a short time. This could greatly reduce the amount of energy the Concentrators consume. Also the End Devices could be configured to be in deep sleep even longer and the XBees could also be in the sleep mode for a longer duration. Another idea could be related to a pull mechanism, i.e., not only receive data measurements when scheduled, but also be able to request a measurement at any time.

Bibliography

[1] "Api mode in detail," https://www.digi.com/resources/documentation/Digidocs/90001942-13/concepts/c_api_mode_detailed.htm?TocPath=XBee%20API%20mode%7C____1, accessed: 2020-11-07.

[2] "The at command set," https://www.digi.com/support/knowledge-base/the-at-command-set, accessed: 2020-11-07.

[3] "Breadboard," https://en.wikipedia.org/wiki/Breadboard, accessed: 2020-11-07.

[4] "Die uart schnittstelle," http://www.mathe-mit-methode.com/schlaufuchs_web/elektrotechnik/mikrocontroller_lernmaterial/microcontroller_allgemein/mikrocontroller_ext_hardware/mikrocontroller_uart.html, accessed: 2020-09-04.

[5] "Esp32-devkitc v4 getting started guide," https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html, accessed: 2020-11-07.

[6] "Flashing micropython firmware with esptool.py on esp32 and esp8266," https://randomnerdtutorials.com/flashing-micropython-firmware-esptool-py-esp32-esp8266/, accessed: 2020-10-05.

[7] "Iana ip flow information export (ipfix) entities," https://www.iana.org/assignments/ipfix/ipfix.xhtml, accessed: 2020-09-05.

[8] "Ieee 802.11-2016," https://standards.ieee.org/standard/802_11-2016.html, accessed: 2020-11-07.

[9] "Micropython downloads," https://micropython.org/download, accessed: 2020-10-30.

[10] "Micropython homepage," https://micropython.org/, accessed: 2020-09-04.

[11] "Multihop network," https://www.sciencedirect.com/topics/engineering/multihop-network, accessed: 2020-11-07.

[12] "Mysql data type storage requirements," https://dev.mysql.com/doc/refman/5.6/en/storage-requirements.html#data-types-storage-reqs-date-time, accessed: 2020-09-29.

[13] "Superb," https://www.macchina.cc/catalog/m2-accessories/superb, accessed: 2020-11-07.

[14] "Thonny," https://thonny.org/, accessed: 2020-09-08.

[15] "Xctu," https://www.digi.com/products/embedded-systems/digi-xbee/digi-xbee-tools/xctu, accessed: 2020-11-07.

[16] "Zeromq messaging patterns - publish/subscribe," https://learning-0mq-with-pyzmq.readthedocs.io/en/latest/pyzmq/patterns/pubsub.html, accessed: 2020-10-05.

[17] "Ieee standard for low-rate wireless networks," *IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015)*, pp. 1–800, 2020.

[18] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer networks*, vol. 38, no. 4, pp. 393–422, 2002.

[19] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[20] B. Claise and B. Trammell, "Information model for ip flow information export (ipfix)," Internet Requests for Comments, RFC Editor, RFC 7012, 09 2013. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7012.txt

[21] B. Claise, B. Trammell, and P. Aitken, "Specification of the ip flow information export (ipfix) protocol for the exchange of flow information," Internet Requests for Comments, RFC Editor, RFC 7011, 09 2013. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7011.txt

[22] *XBee/XBee-PRO S2C Zigbee RF Module User Guide*, Digi International, 01 2020, aG.

[23] S. C. Ergen, "Zigbee/ieee 802.15. 4 summary," *UC Berkeley, September*, vol. 10, no. 17, p. 11, 2004.

[24] Espressif Systems, "Uart," https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/uart.html, accessed: 2020-09-04.

[25] *ESP32-WROOM-32D & ESP32-WROOM-32U Datasheet*, Espressif Systems, 09 2019, v1.9.

[26] *ESP32 Series Datasheet*, Espressif Systems, 04 2020, v3.4. https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[27] A. Fredrik Ahlberg, "esptool 2.8," https://pypi.org/project/esptool/, accessed: 2020-09-21.

[28] R. Huber, "TinyIPFIX for ESP32, GitHub repository," https://github.com/ramonhuber/TinyIPFIX-for-ESP32, Oct. 2020.

[29] T. Kothmayr, "Data collection in wireless sensor networks for autonomic home networking," *Bachelor Thesis, Department of Computer Science, Technische Universität München*, 2010.

[30] Macchina LLC, "Superb esp32 breakout - overview," https://docs.macchina.cc/superb-docs/hardware, 12 2017, revision: R0, Accessed: 2020-09-20.

[31] A. Maier, A. Sharp, and Y. Vagapov, "Comparative analysis and practical implementation of the esp32 microcontroller module for the internet of things," in *2017 Internet Technologies and Applications (ITA)*. IEEE, 2017, pp. 143–148.

[32] M. Meister, "Data gathering in wireless sensor networks using ipfix under contiki," 2015.

[33] C. Schmitt, B. Stiller, and B. Trammell, "Tinyipfix for smart meters in constrained networks," Internet Requests for Comments, RFC Editor, RFC 8272, 11 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8272.txt

[34] L. Sgier, "Optimization of tinyipfix implementation in contiki and realtime visualization of data," *Software Project, Communication Systems Group, Department of Informatics, University of Zurich, Zurich, Switzerland*, 2016.

[35] ——, "Tinyipfix aggregation in contiki," 2017.

[36] *XBee-Explorer-Dongle*, SparkFun Electronics, 09 2015, v23.

Abbreviations

PCB	Printed Circuit Board
GUI	Graphical User Interface
UART	Universal Asynchronous Receiver Transmitter
IANA	Internet Assigned Numbers Authority
IDE	Integrated Development Environment
RTC	Real Time Clock
GPIO	General Purpose Input Output
WSN	Wireless Sensor Network
USB	Universal Serial Bus
IPFIX	Internet Protocol Flow Information Export
TinyIPFIX	Tiny Internet Protocol Flow Information Export
PAN	Personal Area Network
WPAN	Wireless Personal Area Network
CTS	Clear to send
mW	Milliwatt
RFC	Request for Comments
IoT	Internet of Things

List of Figures

2.1 Espressif ESP32 DevKitC V4 6

2.2 Macchina SuperB 7

2.3 Digi XBee Pro S2C 8

2.4 TinyIPFIX Message 11

2.5 TinyIPFIX Message Header, Source: [33] 12

2.6 IPFIX Message Header, Source: [33] 13

2.7 IANA Field Specifiers, Source: [7] 15

2.8 TinyIPFIX Devices 17

2.9 Thonny IDE 19

3.1 Sensor Network 22

3.2 Collector and Publish/Subscribe Network 23

3.3 XBee connected to a Non-constrained Device 24

3.4 Test Network Overview 25

3.5 Use Case Diagram 26

3.6 Class Diagram 28

3.7 SuperB with grounded “IO0” pin 33

3.8 Pin Overview 36

3.9 Digi XCTU Software 37

4.1 TinyIPFIX Overhead 48

4.2 TinyIPFIX and Typle-Length-Value Overhead Comparison 49

4.3 Transmission reliability 51

4.4 Energy Consumption based on message sending interval 54

A.1 SuperB Schematic, Source: [30] 70

B.1 TinyIPFIX Message Navigation Properties 72

C.1 Connecting the ESP32 DevKitC V4 with the XBee (End Device) 73

C.2 Connecting the SuperB with the XBee (End Device) 74

C.3 Connecting the ESP32 DevKitC V4 with the XBee (Concentrator) 75

C.4 Connecting the SuperB with the XBee (Concentrator) 75

List of Tables

3.1 XBee MY and DL addresses 40

4.1 Energy Consumption based on Device Type and State 52

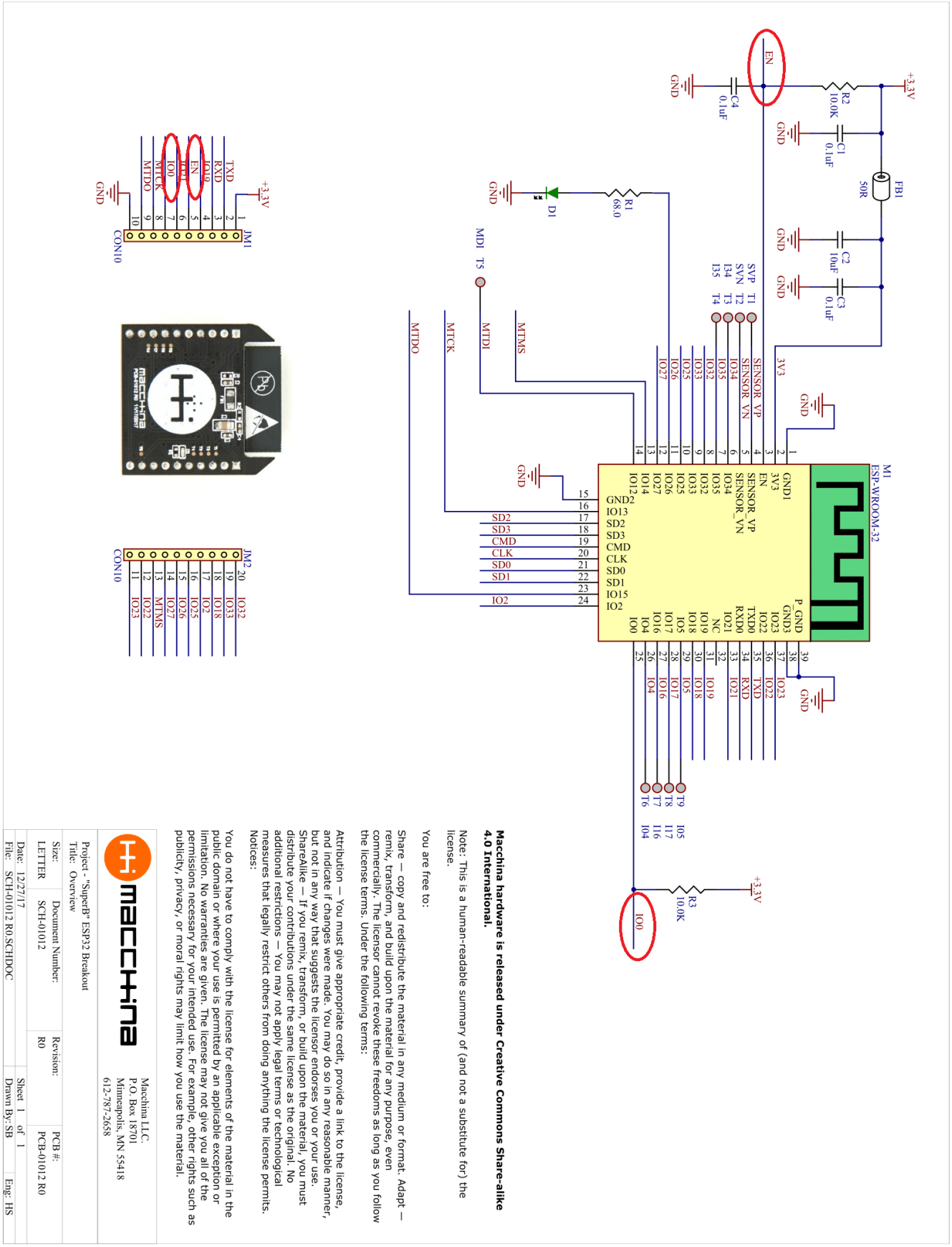
Listings

3.1 Listen to ESP32 using putty on windows 33
3.2 List of COM ports on windows 33
3.3 Listen to ESP32 using putty on Linux 34
3.4 Install esptool.py 34
3.5 Erase flash on the ESP32 34
3.6 Change path in CMD 34
3.7 Flash MicroPython onto the ESP32 35
3.8 Functions to create TinyIPFIX Messages from the device class 40
3.9 Run function from the Concentrator class 43
3.10 Main.py from the Concentrator 45
B.1 Navigation Example 71

Appendix A

SuperB Schematic

Figure A.1 shows the schematic of the Macchina SuperB.



Macchina hardware is released under Creative Commons Share-alike 4.0 International.

Note: This is a human-readable summary of (and not a substitute for) the license.

You are free to:

Share — copy and redistribute the material in any medium or format. Adapt — remix, transform, and build upon the material for any purpose, even commercially. The licensor cannot revoke these freedoms as long as you follow the license terms. Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation. No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

Macchina LLC.
P.O. Box 18701
Minneapolis, MN 55418
612-787-2658

Project: "SuperB" ESP32 Breakout			
Title: Overview			
Size: LETTER	Document Number: SCH-01012	Revision: R0	PCB #: PCB-01012 R0
Date: 12/27/17			Sheet 1 of 1
File: SCH-01012 R0.SCHDOC			Drawn By:: SB Eng: HS

Figure A.1: SuperB Schematic, Source: [30]

Appendix B

TinyIPFIX Message Navigation Properties

Figure B.1 shows the navigation properties of the Message Class. This Figure should facilitate working with Message objects, because many MicroPython IDE's do not have hints for navigation properties. To print all the tiny set id's of a Data Message one could use this code for example:

```
for record_set in message.records_sets:
    print(record_set.set_header.tiny_set_id)
```

Listing B.1: Navigation Example

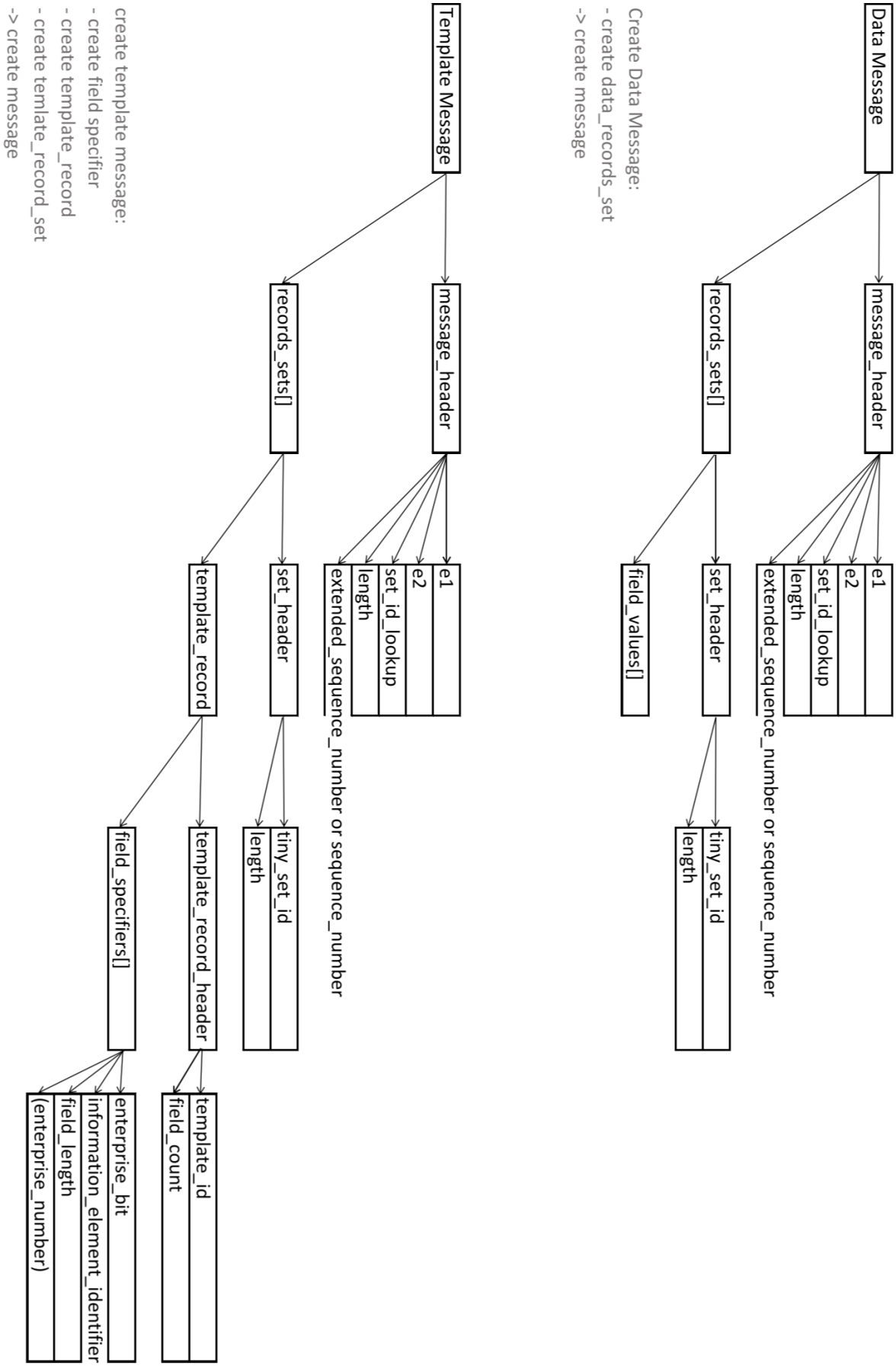


Figure B.1: TinyIPFIX Message Navigation Properties

Appendix C

Connecting the Pins of the Devices

Figure C.1 shows how the pins of a ESP32 DevKitC V4 need to be connected to the pins of an XBee, when they are used as an End Device.

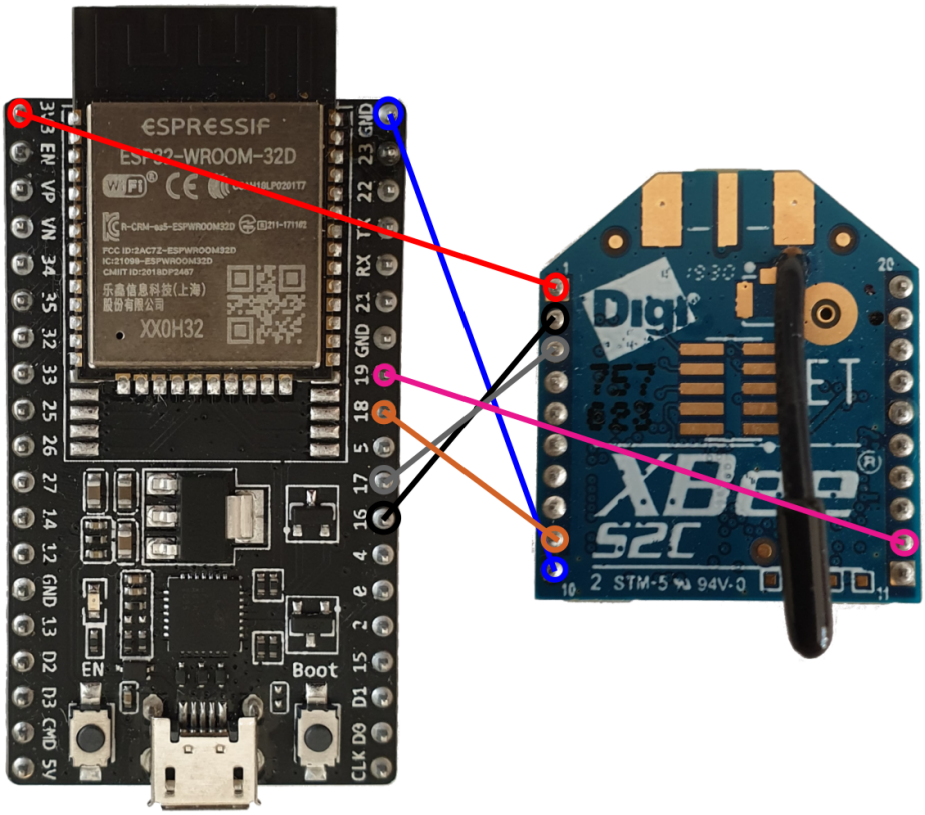


Figure C.1: Connecting the ESP32 DevKitC V4 with the XBee (End Device)

Figure C.2 shows how the pins of a SuperB need to be connected to the pins of an XBee,

when they are used as an End Device.

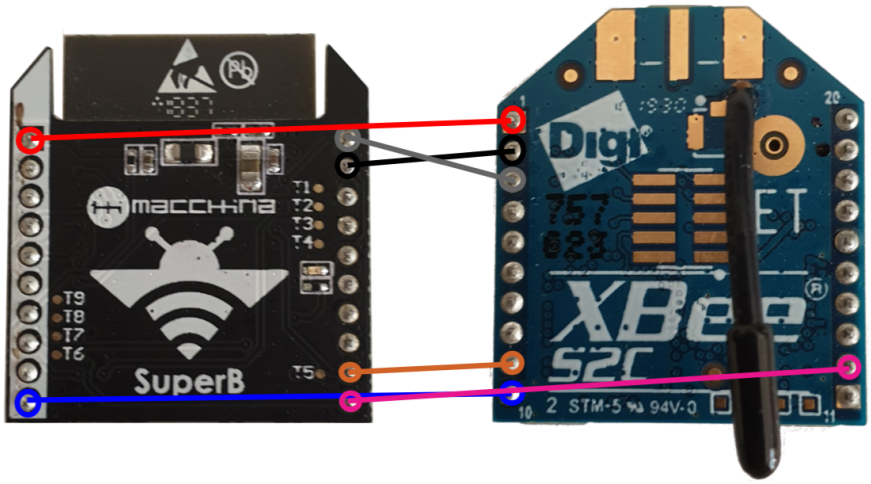


Figure C.2: Connecting the SuperB with the XBee (End Device)

Figure C.3 shows how the pins of a ESP32 DevKitC V4 need to be connected to the pins of an XBee, when they are used as a Concentrator.

Figure C.4 shows how the pins of a SuperB need to be connected to the pins of an XBee, when they are used as a Concentrator.

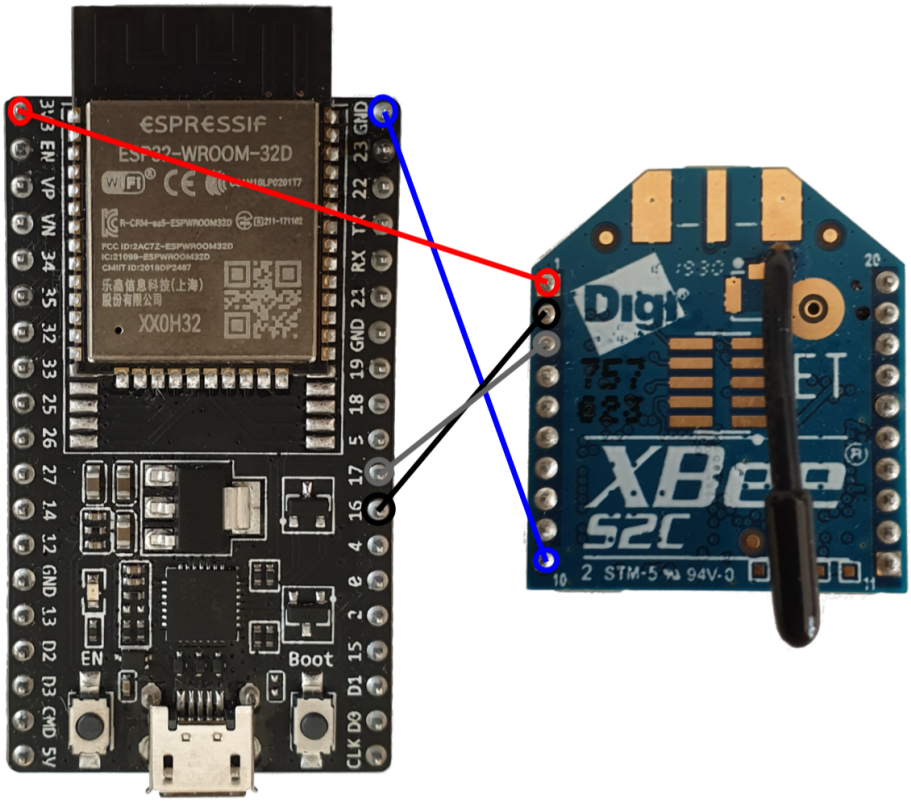


Figure C.3: Connecting the ESP32 DevKitC V4 with the XBee (Concentrator)

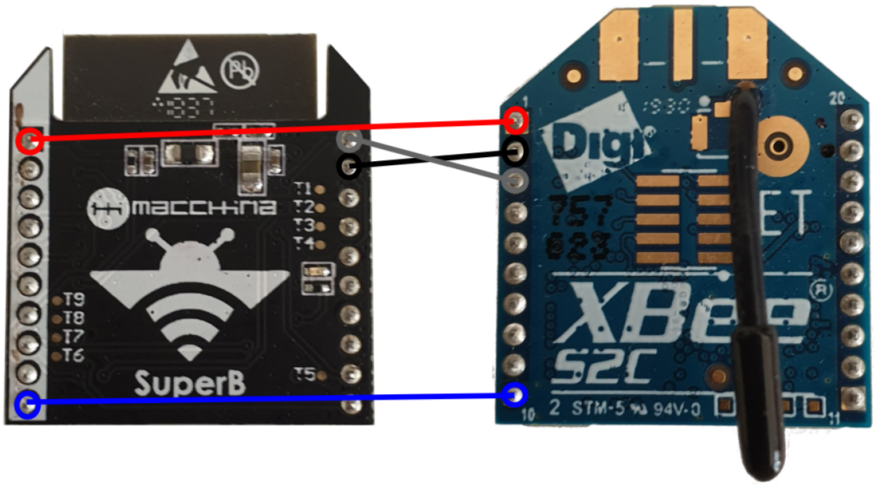


Figure C.4: Connecting the SuperB with the XBee (Concentrator)