MONTAN
UNIVERSITÄT
LEOBEN

Chair of Automation

Master's Thesis

Research and Development of a Telemetry System for Condition Monitoring of Machines Using Sub-GHz Frequency Bands

Philipp Peterseil, BSc

September 2019

Abstract

This thesis presents the conception, implementation and test of a condition monitoring system using sub-GHz frequency bands. The results from performance testing on a complete prototype system are also presented. The Wireless M-Bus network protocol has been selected and implemented; this provides a standardized and established interface.

The battery powered smart sensor was designed as an embedded system featuring a low-power wireless microcontroller together with an accelerometer. An Industrial PC (IPC), with IEC 61131 certification, was selected as the edge device. This device includes a Wireless M-Bus extension module.

The range was evaluated using omnidirectional aerials. At a transmission power of 14 dBm, corresponding to a current consumption of 6.5 mA, ranges up to several hundred meters were achieved; there is, however, a dependency on obstacles that are projecting into the fresnel zone.

The main influence in long-term mean current-consumption and therefore life of battery, was identified as the current consumed in *low power deep sleep* mode. Assuming a typical application measuring 1600 samples at a rate of 3200 Hz once each hour and transmitting data at an interval of six hours leads to a life of battery estimation of about two years.

These results confirm that sub-GHz frequency bands are well-suited for condition monitoring purposes with moderate data rate demands.

Zusammenfassung

In dieser Arbeit wird Konzept, Umsetzung und Evaluierung eines Systems zur Zustandsüberwachung vorgestellt, welches auf den Sub-GHz Frequenzbändern basiert. Darüber hinaus werden die an einem vollständigen Prototypen durchgeführten Leistungstests präsentiert. Das Wireless M-Bus Netzwerkprotokoll wurde ausgewählt, da es eine genormte, etablierte Schnittstelle zur Datenübertragung bietet.

Der Smart Sensor vereint einen Beschleunigungssensor und einen energieeffizienten Mikrocontroller mit Netzwerkunterstützung zu einem batteriebetriebenen eingebetteten System. Ein IEC 61131 konformer Industrie PC (IPC) wurde als Edge Device verwendet. Dieser beinhaltet ein Wireless M-Bus Erweiterungsmodul.

Die Evaluierung der Reichweite wurde mit omnidirektionalen Antennen durchgeführt. Bei einer Sendeleistung von 14 dBm, entsprechend einer Stromaufnahme von 6.5 mA, konnten Reichweiten von einigen hundert Metern, abhängig von Hindernissen in der Fresnel'schen Zone, erreicht werden.

Der Stromverbrauch im Ruhezustand des Smart Sensors wurde als Haupteinfluss der Batteriebetriebszeit identifiziert. Unter Annahme eines stündlichen Messzykluses mit 1600 Datenpunkten, einer Messfrequenz von 3200 Hz, sowie einer gesammelten Datenübertragung in Intervallen von sechs Stunden, konnten Batteriebetriebszeiten von bis zu ca. zwei Jahren geschätzt werden.

Diese Ergebnisse bestätigen die Eignung der Sub-GHz Frequenzbänder für Anwendungen in der Zustandsüberwachung, sofern moderate Anforderungen an die Übertragungsgeschwindigkeit vorliegen.

MONTANUNIVERSITÄT LEOBEN
www.unileoben.ac.at

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt, und mich auch sonst keiner unerlaubten Hilfsmittel bedient habe.

Ich erkläre, dass ich die Richtlinien des Senats der Montanuniversität Leoben zu "Gute wissenschaftliche Praxis" gelesen, verstanden und befolgt habe.

Weiters erkläre ich, dass die elektronische und gedruckte Version der eingereichten wissenschaftlichen Abschlussarbeit formal und inhaltlich identisch sind.

Datum 17.09.2019

Unterschrift Verfasser/in
Philipp, Peterseil
Matrikelnummer: 01535141

Acknowledgements

I dedicate this thesis to those who encouraged and supported me during my studies at the University of Leoben in any way.

I would like to thank Univ.-Prof. Paul O'Leary who supervised this thesis for his scientific and technical advices. He aroused my interest in science many times by guiding me to fascinating insights into how e. g. speed of light constraints the performance of computer processors stepwise.

I would like to thank Michael Habacher from eSENSEial Data Science GmbH for supporting me with the required hardware and for sharing his experience with me.

Furthermore I would like to thank my girlfriend Alina as well as my family and friends for standing by my side during this thesis.

Contents

Part I Background methods

Part II Research & Development

Part III Assessment

Part IV Appendix

List of Figures

List of Tables

Acronyms

AES	Advanced Encryption Standard
BLE	Bluetooth Low Energy
CBC	Cipher Block Chaining
CPFSK	Continuous-Phase Frequency Shift Keying
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSV	Comma Separated Value
CTR	Counter
DRAM	Dynamic Random Access Memory
ECB	Electronic Codebook
EEPROM	Electrically Erasable Programmable Read-Only Memory
FSK	Frequency Shift Keying
GFSK	Gaussian Frequency Shift Keying
GPIO	General Purpose Input/Output
GSM	Global System for Mobile Communications
I^{2}C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IIoT	Industrial Internet of Things
IP	Internet Protocol
IPC	Industrial PC
ISM	Industrial Scientific Medical
ISO	International Organisation for Standardization
IV	Initialisation Vector

MCU	Microcontroller Unit
OSI	Open Systems Interconnection
PCB	Printed Circuit Board
PWM	Phase Width Modulation
RAM	Random Access Memory
RF	Radio Frequency
ROM	Read-Only Memory
RSA	Rivest-Shamir-Adleman
RSSI	Received Signal Strength Indicator
RTC	Real Time Clock
SMD	Surface Mounted Device
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
UART	Universal Asynchronous Receiver/Transmitter
XML	Extensive Markup Language
XOR	Exclusive Or

Notation

Mathematical notation

0b00001010	Binary number 0b00001010 $= 10 =$ 0x0A
0x0A	Hexadecimal number 0x0$A = 10 =$ 0b00001010
$\underline{c}$	Complex number c
i	Imaginary unit
A	Matrix A
$\boldsymbol{a}$	Vector $\boldsymbol{a}$
$a \ll b$	Shifts bits of a to the left by b digits
$a \gg b$	Shifts bits of a to the right by b digits
$a \bullet b$	Special multiplication, see (4.11)
$\mathsf{A} \oplus \mathsf{B}$	Element-wise, bitwise XOR operation
$\deg[\mathrm{y}(x)]$	Degree of the polynomial $\mathrm{y}(x)$
$\mathrm{R}_{\mathrm{g}(x)}[\mathrm{h}(x)]$	Remainder of the polynomial long division $\frac{\mathrm{h}(x)}{\mathrm{g}(x)}$

Citation style

An example sentence [`<source>`, `<page(s)>`].

The square bracket within a sentence indicates that the the specified page(s) `<page(s)>` of the source `<source>` were used as reference for the particular sentence. The specification of page numbers is optional.

An example paragraph. [`<source>`, `<page(s)>`]

The square bracket after the last sentence of a paragraph indicates that the the specified page(s) `<page(s)>` of the source `<source>` were used as reference for the whole paragraph. The specification of page numbers is optional.

No direct quotations were used.

1 Introduction

Condition monitoring is a central part of the Industry 4.0 concept. Compared to the conventional condition assessment, which is conducted on a regular basis, it offers continuous data that can be used to autonomously determine when human intervention is actually required. Methods of data science, e. g. artificial intelligence, can be used to estimate when a specific component of the monitored system is going to break down. This can avoid critical damage and therefore long downtime while enabling to use the components to their full extent. Beyond that, the knowledge gained from condition monitoring can be used in claims management as well as to improve further designs. This concept relies on sensors that continuously provide measuring data of relevant parameters.

In monitoring large systems, transmission of data is a challenge. Industry 4.0 knows two contrary concepts to deal with it, cloud computing and edge computing. The architecture of Industrial Internet of Things (IIoT) networks can be described according to Fig. 1.1. Devices, that can be e. g. smart meters, are connected to an edge gateway. In highly flexible implementations wireless connections are used. Gateway and cloud communicate via the Internet. Cloud computing refers to collecting and analyzing sensor data at remote data centers while edge computing is a decentralized approach where data is processed at edge level. Cloud computing offers unprecedented computing resources, with the drawback that it relies on a internet connection that is capable of transmitting data produced by hundreds of sensors. On the contrary, edge computing is conducted distributed, as physically close to the sensors as possible. This eliminates the Internet as a strict dependency and thereby improves reliability. In combined approaches the Internet is used to efficiently distribute computation results of edge computing, which can be significantly lower sized. Fundamental for suchlike concepts are smart sensors that deliver enough computational power for their specific task.

Smart sensors are embedded systems consisting of a sensor, a Microcontroller Unit (MCU) and an interface for connecting to other devices as well as to the Internet via an edge gateway. In many cases they have to withstand vibration, dirt and even splash-water. By implementing battery powered sensors with wireless communication, the smart sensor can be infused in resin to become a self-contained system that is isolated from the environment. This also prevents cable breakage, which is a common failure mechanism. If an existing machine is upgraded with a condition monitoring system, the installation of cables for power supply and communication is no longer required. Due to latest generations of high tech Microcontroller Units (MCUs), battery powered smart meters can be designed to operate for over 3 years with just a coin cell battery. Therefore one of the main obstacles is to implement a robust wireless data transmission method that is suitable for harsh industrial environments.

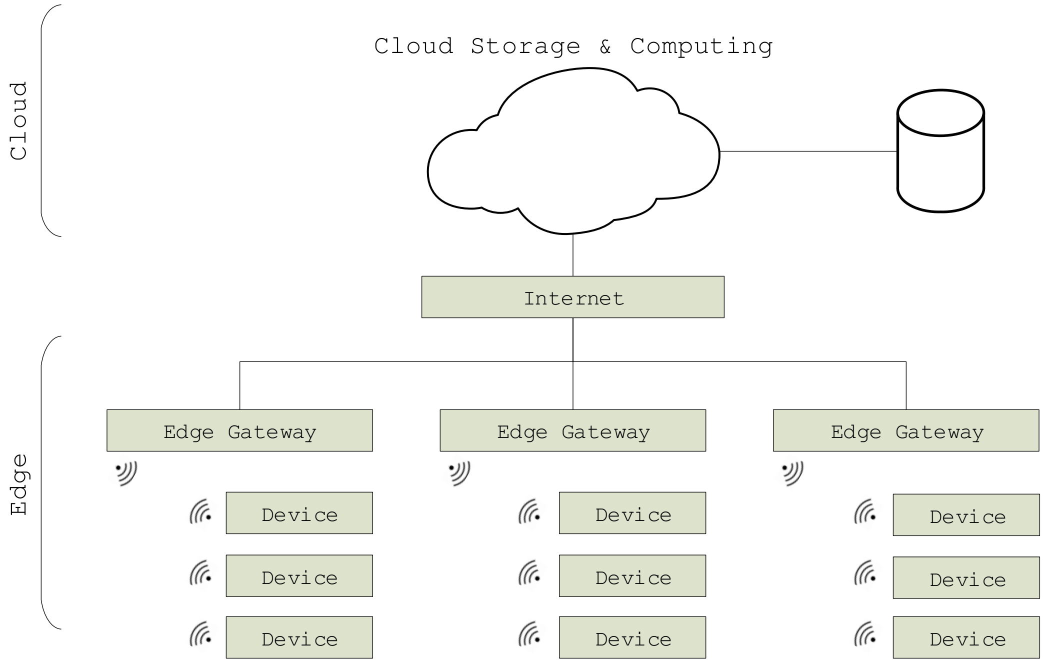


Fig. 1.1. Architecture of an Industrial Internet of Things (IIoT) network.

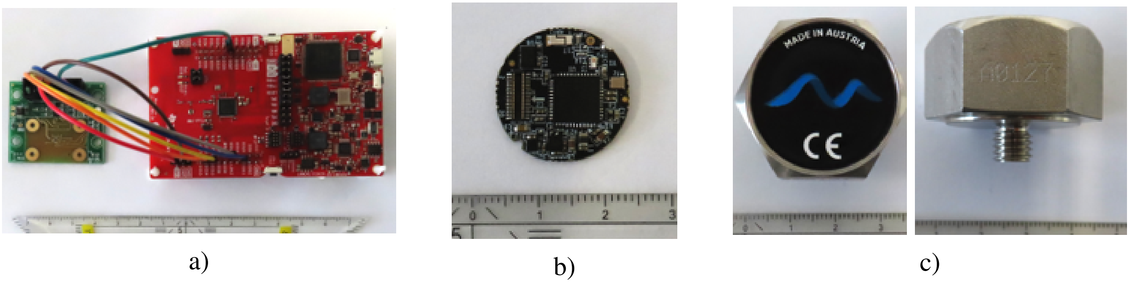
1.1 State of the Art

At present many smart sensors such as the sensor screw in Fig. 1.2c use the 2.4 GHz frequency bands, e. g. Bluetooth Low Energy (BLE), for transmission. The experience gathered during launching them shows that troubles can rise when such sensors interfere with other devices in the same frequency band. This happens e. g. in environments with many BLE devices. In industrial environments also noise levels are higher. In monitoring large plants, another constraint is the limitation in transmission range.

Due to the lower number of devices using sub-GHz frequency bands, it is speculated that an appropriate sub-GHz protocol might better suit the needs of an IIoT smart sensor. From a theoretical point of view also higher transmission ranges should be possible. It is well-known that the transmission rate using sub-GHz is slower, which would not be a problem in many cases.

1.2 Objectives

The application of sub-GHz frequency bands in condition monitoring tasks, especially in harsh industrial environments, shall be investigated. For any specific application, a literature research



a) b) c)

Fig. 1.2. The development of smart sensor's hardware takes place in three stages. The first prototype often utilizes development boards that are supplied by the microprocessor unit's producer, see Fig. 1.2a. After the proof of concept, circuit boards are designed, see Fig. 1.2b, which are used for further firmware development and are installed to the product at the stage of manufacturing, see Fig. 1.2c. Image courtesy of eSENSEial Data Science GmbH.

should be carried out to examine benefits and drawbacks of sub-GHz and which protocol best fits the requirements. In this application it became evident that the Wireless M-Bus sub-GHz network protocol is the most suitable, see Chapter 5. This includes a market research to clarify for which protocol industry-suited modules are available. Afterwards the hard-, firm-, and software required to implement a prototype wireless condition monitoring system, including a sub-GHz network stack, is developed that acts as a proof of concept. The prototype is based on a development board of the CC1352 MCU of Texas Instruments, see Fig. 1.2a. It enables testing of wireless sensors, e.g. vibration sensors, with respect to their range and stability of transmission, as well as their current consumption.

Part I

Background methods

2 Embedded Systems Architecture

In a world where more and more machines and also everyday items are getting linked to the Internet, MCUs for highly diverse fields of application are available. They are the heart of every embedded system, that is, an electronic system with all hardware components embedded on a Printed Circuit Board (PCB) [20, p. 93]. Some microcontroller families are focussed on efficiency, while others may prioritize performance or connectivity. Selecting an MCU for a project is always a compromise. For this thesis the Texas Instruments CC1352P was chosen. It features SubGHz support, an ARM Cortex-M4 Central Processing Unit (CPU) with floating point unit and a fair current consumption.

This chapter is discusses the architecture of this particular device and the most important components that can be found in almost every MCU. Furthermore the interfacing between an MCU and a sensor shall be investigated.

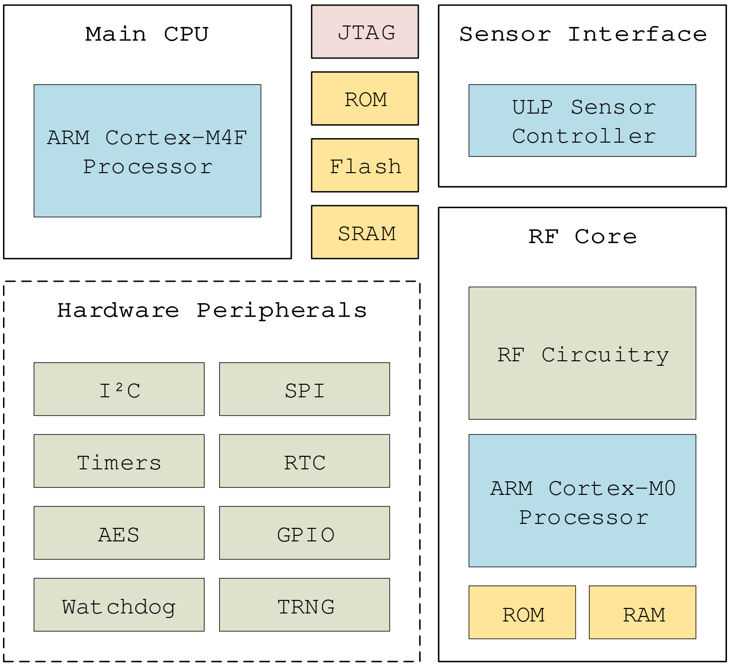


Fig. 2.1. Architecture of the Texas Instruments CC1352P. Compare to [4].

The main modules of the Texas Instruments CC1352P can be identified as: the main CPU; the Radio Frequency (RF) core and the sensor interface. Each of these elements has it's own dedicated CPU. See Figure 2.1.

Each module is connected to the hardware peripherals and they partly share Read-Only Memory (ROM), flash memory and Static Random Access Memory (SRAM).

2.1 Central Processing Unit

A processor is responsible for the actual execution of a program. More precisely, a processor cyclically fetches, decodes and executes the machine code instructions of a program that is stored in any kind of memory. The instruction set thereby depends on the specific CPU. [20, pp. 156–157]

A program that is written in a high-level programming language such as C++ has to be translated to machine code instructions suitable for the specific device before it can be executed. This process is called compilation and is explained in detail in Chapter 7.4.

The Texas Instruments CC1352P contains three CPUs. The high-performance main CPU is running user applications that can be very computationally intensive. However, it has a relative high current consumption in active mode. In contrast, the sensor controller is an ultra low power processor that is intended to monitor sensor values and alert the main CPU, which can be in a *low power deep sleep* mode at the moment, if required. The sensor controller shares the peripherals with the main CPU and can only take over very basic tasks. The RF core is interfacing with RF circuitry and provides a command-based application programming interface for autonomously handling various radio tasks to the main CPU. [5, pp. 77 84–85 1900]

2.2 Memory

This section should give a brief overview to the different memory types that are used in the Texas Instruments CC1352P and what their purpose is.

2.2.1 Flash

Flash, also referred to as *flash electrically erasable programmable read-only memory*, is responsible for storing the application firmware and constants. A program can be executed whether directly from flash or copied to the Random Access Memory (RAM) at first and then be executed from there.

Flash is a non-volatile memory, that is, data is not lost when the memory is not powered. Flash memory can be erased in blocks a limited number of times.

External flash memories that can be connected to the microcontroller via e. g. Serial Peripheral Interface (SPI) can be used to extend storage capacity.

2.2.2 Random Access Memory (RAM)

The RAM is used to store the programs stack and heap, as well as global or static variables. Local variables are allocated on the stack and therefore also stored in the RAM.

Both of the most common types, SRAM and Dynamic Random Access Memory (DRAM), are volatile memories.

The Texas Instruments CC1352P implements a SRAM, which is the fastest type of RAM. [20, p. 180]

2.2.3 Read Only Memory (ROM)

A read-only memory that is preprogrammed by the manufacturer can be used to provide e. g. bootloaders, operating systems and network stacks. Once programmed their content can't be altered anymore. This can be achieved by etching data into the chip during production. Memory of this type is called mask ROM. [20, p. 178]

2.3 Peripherals

Peripherals in a MCU are extending the core functionalities that the CPU provides together with various types of memory. The quantity and types of peripheral devices vary even within the same MCU families. Peripheral hardware that is utilized within the implementation of this project's hard- and firmware shall be introduced.

2.3.1 General Purpose Input and Output (GPIO)

General Purpose Input/Output (GPIO) pins are programmable digital inputs or outputs. Configured as inputs they can be used to read in logic levels or trigger interrupts of the CPU. As Outputs GPIO pins can control external logic and drive low-current circuits. For switching of higher currents the GPIO pin usually drives a transistor. Most GPIO units have selectable pull-up or pull-down resistors integrated.

GPIO pins share functionality with other peripheral hardware. Therefore the programmer can select if a particular pin shall be a GPIO pin or e. g. a pin that refers to a communication interface.

A notable feature of the Texas Instruments CC1352P is the multiplexer that enables routing of all peripherals signals to arbitrary pins. [5, p. 84]

2.3.2 Timer

Timers are basically counters that are periodically incremented by a clock. The clock source and prescalers are configurable. This enables not only precise timings but also fully autonomous generation of phase-width modulated signals in a specific mode.

Interrupting the CPU for a number of events can be configurated.

2.3.3 Real Time Clock (RTC)

A Real Time Clock (RTC) module is a special type of timer that is usually driven from a separate 32.768 Hz low-frequency crystal oscillator. Together with a 2^{15} prescaler, a clock cycle of *exactly* 1 s is achieved. The separate quartz oscillator in addition allows to turn off the main oscillator while the CPU is waiting for an external interrupt to save energy. In most microcontrollers that are equipped with a RTC, the module is intended to be battery backed-up by a coin-cell battery to keep time even if the main circuit is not powered.

2.3.4 Serial Peripheral Interface (SPI)

SPI is a synchronous interface that uses a common clock cycle for bidirectional transmission. SPI is a very fast interface and is often used to program microcontrollers together with bootloaders, to connect external memory or to communicate with sensors.

A SPI module can connect more than one external peripheral to the MCU, see Figure 2.2. Therefore the external peripheral that is currently communicated with, has to be selected by means of issuing the slave select (`SS`) line.

2.3.5 Inter-Integrated Circuit (I^{2}C)

Inter-Integrated Circuit (I^{2}C) is an interface that can connect more than 100 devices. Each device can be selected by means of an I^{2}C address that is unique within the network. I^{2}C only requires two wires, a bidirectional data line and a clock signal. It is used for sensors that get along with medium transfer rates.

2.3.6 Advanced Encryption Standard (AES) Crypto Accelerator

The Advanced Encryption Standard (AES) crypto accelerator is a module that is designed to efficiently perform operations that are required for encrypting and decrypting. The AES is further discussed in Chapter 4.2.

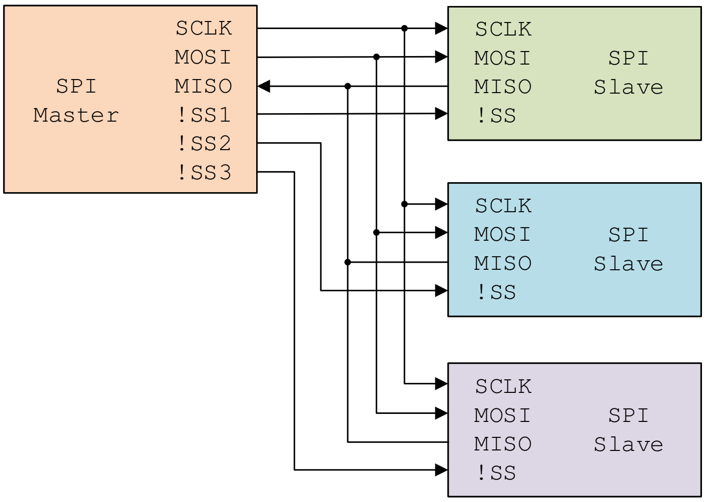


Fig. 2.2. SPI connecting a master device with three slaves. The slave select line is usually inverted. An inverted line is indicated by an exclamation mark, e. g. `!SS`. Compare to [21, p. 21].

2.4 Flash Programming

Flash programming is the process of transferring an executable program to the non-volatile storage of an MCU. This requires programming hardware which is embedded in development boards usually. Most programming adapters are connected to the PC via USB. A common interface that connects the programming adapter to the microcontroller is called JTAG. JTAG can be used to flash a device as well as to debug firmware.

Another state of the art method used for firmware updates is *over-the-air upgrade*. Over-the-air upgrade uses a wireless protocol to transmit the executable file to the MCU. A firmware update process then is conducted.

2.5 Interfacing a Sensor

Digital external peripherals, in particular sensors, are usually physically connected to the MCU via SPI, I^{2}C, Universal Asynchronous Receiver/Transmitter (UART) or a parallel interface. Additionally some sensors provide programmable interrupt outputs which are routed to GPIO pins of the MCU.

Digital sensors generally provide a register based setup. Each register has a unique address and a specific purpose. To achieve the desired configuration, to start and stop sampling etc., registers are set consecutively according to the device's datasheet. Once a measurment is triggered, the

MCU has to poll a status register that signals if the conversion is finished recurrently or wait for a specified time. As an alternative the MCU can proceed with it's task until the sensor triggers an interrupt, signaling that the conversion is finished. This is the desired method if the sensor provides it.

3 Wireless Data Transmission

Development of wireless data transmission methods is a demanding task that involves a hardware and a software part. A general design reference covering both is the Open Systems Interconnection (OSI) model[1]. It's key concepts shall be introduced in this chapter. It especially points out how a network can be arranged in principle. Furthermore the frequency shift keying method shall be introduced as an important modulation method for physical wireless data transmission.

3.1 Open Systems Interconnection Model

The OSI reference model is describing seven formal layers in which networking is separated. Each of the layers has a particular task that it is responsible for. The layers are numbered from one to seven, beginning with the physical layer which is the closest to the hardware. With an increasing number, the abstractedness of the layer rises. In general, the lower four layers are about *transferring* data from A to B, while the upper ones are concerned with the application and user interaction. The various layers and their responsibilities are listed in Tab. 3.1. Note that not every system implements all layers. [16, pp. 148–154].

#	Layer	Responsibilities
7	Application Layer	User Application Services, ...
6	Presentation Layer	Compression, Encryption, ...
5	Session Layer	Session Management, ...
4	Transport Layer	Connections, Acknowledgments, Retransmissions, ...
3	Network Layer	Logical Addressing, Fragmentation, ...
2	Data Link Layer	Error Detection and Handling, Addressing, ...
1	Physical Layer	Hardware Specifications, Encoding, ...

Tab. 3.1. Open Systems Interconnection Model (OSI) layers. Compare to [16, pp. 184–185].

A basic principle in the OSI model is that every layer is communicating with it's equal on the other side. Communication on the same layer is represented by a protocol. The link between them is logically for all layers except the physical layer. Communication therefore utilizes lower layers consecutively until the data can finally be *transmitted* at the physical layer. On the receiver side this procedure is happening the other way round. [16, pp. 156–157]

[1] The Open Systems Interconnection (OSI) model is also known as International Organization for Standardization (ISO) model, since it is defined in an ISO standard.

Since they rely on each other, adjacent layers within a particular host have to communicate with each other. Therefore each layer has to define an interface for intercommunication between them. Well-defined interfaces enable to exchange implementations of particular layers. [16, p. 154]

Another important concept of the OSI model is encapsulation. For transmission, each protocol is represented by a *protocol data unit* that consists of protocol headers and data. The protocol data unit of one layer becomes the *service data* of the next layer below, which again forms a protocol data unit together with it's headers, and so forth. Thereby application data is encapsulated by more and more protocol headers downwards the network stack. [16, p. 159]

3.2 Frequency Shift Keying

Frequency Shift Keying (FSK) is a method to modulate the frequency of a carrier signal with respect to a digital input signal. It can be found in wireless data transmission standards like Bluetooth or Global System for Mobile Communications (GSM) and belongs to the physical layer. The remarks in this chapter are illustrated in Figure 3.1.

Aiming at binary FSK, depending on the digital input the carrier frequency f_0 is either shifted to the lower or to the upper side by the frequency deviation Δf. A digital “0” translates to a baseband level of -1 and a digital “1” to a baseband level of $+1$, leading to a frequency shift of $-1 \cdot \Delta f$ respectively $+1 \cdot \Delta f$. [8, pp. 58–60]

The easiest way of modulating the signal would be switching between two independent oscillators that are set up to $f_0 - \Delta f$ and $f_0 + \Delta f$. Due to the fact that the oscillators are not synchronized discontinuities will occur. This leads to high band width demands. A better approach regarding band with usage is Continuous-Phase Frequency Shift Keying (CPFSK). Thereby the frequency shifts also happen instantly but without discontinuities in phase. Even better band with characteristics can be expected by Gaussian Frequency Shift Keying (GFSK). GFSK uses a gaussian filter to smooth transitions of the baseband level which leads to a frequency shift that is continuous in the frequency domain as well as in the time domain. Note that a low band width is desired in order to keep sideband power as low as possible. Furthermore it reduces crosstalk. [37]

Demodulation can be implemented by two sharp bandpass filters that are tuned to $f_0 - \Delta f$ respectively $f_0 + \Delta f$, both followed by envelope detectors. If the signal of the $f_0 - \Delta f$ envelope detector is higher than that of the $f_0 + \Delta f$ envelope detector a digital “0” is output, otherwise a digital “1”. This can be achieved by a comparator. [8, pp. 60–61]

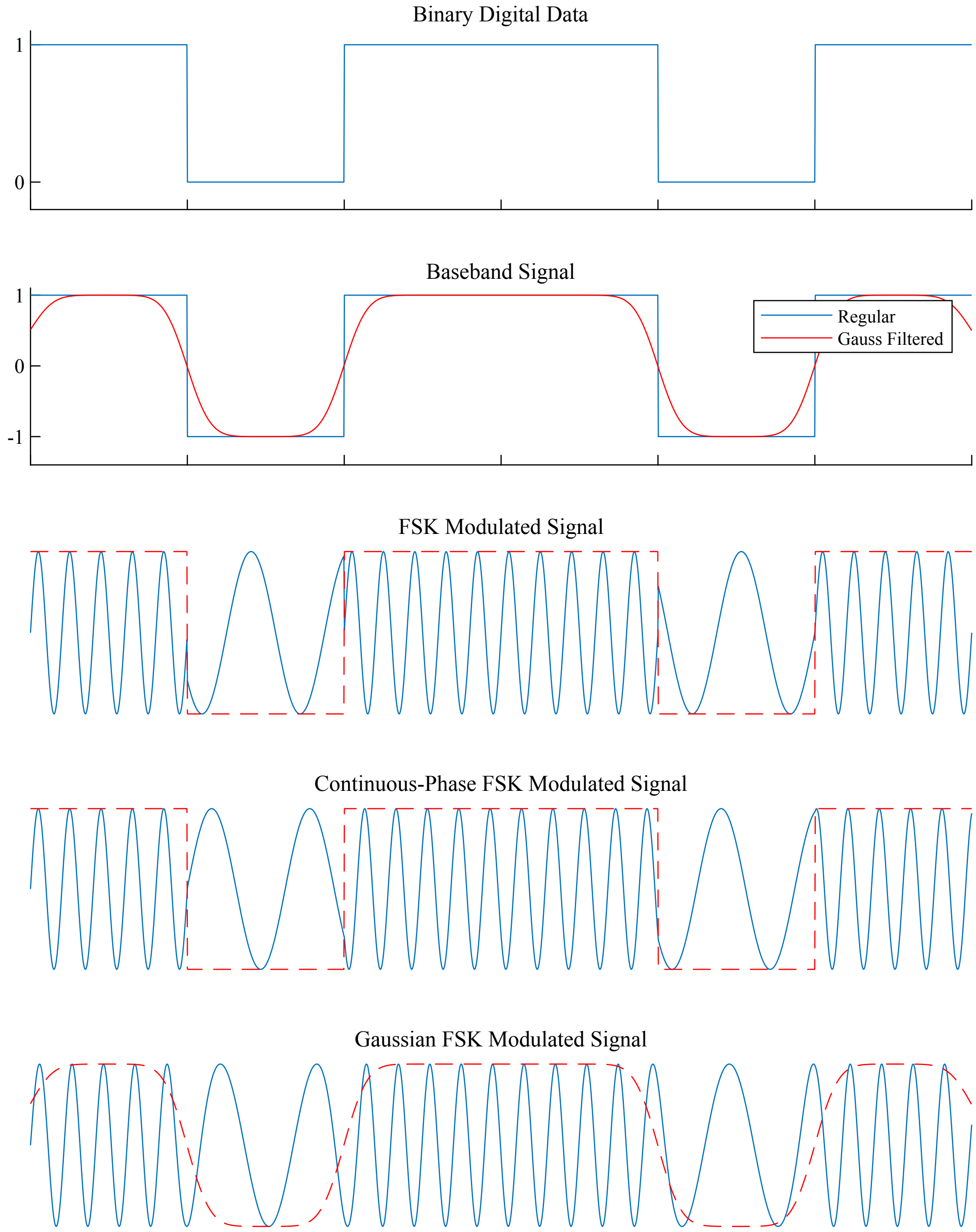


Fig. 3.1. Different Frequency Shift Keying (FSK) methods distinguish by their behaviour in transition. Needless to say, all plots refer to the time domain.

4 Cyclic Coding and Encryption

Transmission of data requires an error detection mechanism that corrupted data can be handled properly. A common error detection mechanism is the Cyclic Redundancy Check (CRC), which appends a checksum to the data. Especially in industrial environments it is recommended to use encrypted communication. A convenient algorithm is the AES. Both can be implemented in hardware as well as in software. In the following sections they shall be discussed from a mathematical point of view.

4.1 Cyclic Redundancy Check (CRC)

Network transmissions can be compared to spoken conversations. Say your opposite noticed you speaking, then he or she would subconsciously check your questions plausibility to decide if he or she has understood *a correct* plausible sentence and whether give you an answer or ask for repetition. In digital data transmission a simple method to check if a message was received correctly is the CRC[1]. Based on a failed CRC check a retransmission would probably be initiated.

CRC checks are based on polynomial long division of polynomials defined in $\mathbb{F}_2[x]$. Therefore specific finite field arithmetics have to be applied when adding or multiplying coefficients, see (4.1). Notice that adding and multiplying of coefficients within $\mathbb{F}_2[x]$ matches the definitions of Exclusive Or (XOR) and logical AND respectively. Subtractions and divisions are implicitly defined as additive and multiplicative inverses. [19, p. 74]

$$\begin{array}{c|cc} + & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array} \qquad \begin{array}{c|cc} \cdot & 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 0 & 1 \end{array} \tag{4.1}$$

Each coefficient of the data polynomial $\mathrm{d}(x)$ represents one bit of the data. To calculate the checksum of the data polynomial, at first a generator polynomial $\mathrm{g}(x)$ has to be defined. It's degree shall be denoted with $w = \deg[\mathrm{g}(x)]$. The CRC checksum is defined as the remainder of a polynomial long division by the generator polynomial. The remainder then is subtracted from the dividend, which makes it an integer multiple of $\mathrm{g}(x)$ and a further polynomial long division by $\mathrm{g}(x)$, the CRC check, would be able without remainder. Consider the dividend would

[1] Cyclic Redundancy Check (CRC) is a very often used term that refers to cyclic codes that are used for error detection. [19, p. 147]

be the unmodified data polynomial $\mathrm{d}(x)$, then subtracting the remainder would modify the data. Therefore the polynomial long division by $\mathrm{g}(x)$ is applied to an extended data polynomial $x^w\mathrm{d}(x)$. This polynomial contains the same information than the data polynomial but reserves space for the remainder, in other words the checksum. Denoting the remainder of a polynomial long division $\frac{\mathrm{h}(x)}{\mathrm{g}(x)}$ as $\mathrm{R}_{\mathrm{g}(x)}[\mathrm{h}(x)]$, the calculation of the polynomial representation of the data including the checksum can be written in the following form, see (4.2). In this representation the residual is added. An addition is exactly the same as a subtraction in $\mathbb{F}_2$, since $\{x = -x \,|\, x \in \mathbb{F}_2\}$. [19, pp. 147–148]

$$\mathrm{c}(x) = x^w\mathrm{d}(x) + \underbrace{\mathrm{R}_{\mathrm{g}(x)}[x^w\mathrm{d}(x)]}_{\text{Checksum}} \tag{4.2}$$

Since the remainder is subtracted from the extended data polynomial, $\mathrm{c}(x)$ is a multiple of $\mathrm{g}(x)$, if the transmission is free of errors. Thus, for a successful transmission, the remainder of the polynomial long division $\frac{\mathrm{c}(x)}{\mathrm{g}(x)}$ has to be 0, see (4.3). This is the criteria of the CRC check.

$$0 = \mathrm{R}_{\mathrm{g}(x)}[\mathrm{c}(x)] \tag{4.3}$$

The algorithm can be fully implemented representing coefficients of polynomials as bits and interpreting the place of the bit as the monomials degree. Additions have to be replaced with XORs and multiplications with logical ANDs. This enables very economic implementations. The difference in the calculation shall be demonstrated by an example. The CRC checksum of the data stream 0b00001100 using the generator polynomial $\mathrm{g}(x) = 1x^3 + 0x^2 + 1x^1 + 1x^0$ shall be calculated. [19, pp. 148–149]

At first the polynomial long division method is applied to calculate the checksum.

$$\begin{aligned} \mathrm{g}(x) &= 1x^3 + 0x^2 + 1x^1 + 1x^0 \\ \mathrm{d}(x) &= 1x^3 + 1x^2 + 0x^1 + 0x^0 \\ w &= \deg[\mathrm{g}(x)] = 3 \\ x^w\mathrm{d}(x) &= 1x^6 + 1x^5 + 0x^4 + 0x^3 + 0x^2 + 0x^1 + 0x^0 \end{aligned} \tag{4.4}$$

$$\begin{array}{rrrrrrrl} 1x^6 & +1x^5 & +0x^4 & +0x^3 & +0x^2 & +0x^1 & +0x^0 & : \mathrm{g}(x) = x^3 + x^2 + x \\ -1x^6 & -0x^5 & -1x^4 & -1x^3 & & & & \\ & 1x^5 & 1x^4 & 1x^3 & +0x^2 & & & \\ & -1x^5 & -0x^4 & -1x^3 & -1x^2 & & & \\ & & +1x^4 & 0x^3 & +1x^2 & +0x^1 & & \\ & & -1x^4 & -0x^3 & -1x^2 & -1x^1 & & \\ \hline & & & & & 1x^1 & +0x^0 & \qquad = \mathrm{r}(x) \end{array} \tag{4.5}$$

$$\begin{aligned}
\mathrm{r}(x) &= \mathrm{R}_{\mathrm{g}(x)}[x^w \mathrm{d}(x)] = 1x^1 + 0x^0 \\
\mathrm{c}(x) &= x^w \mathrm{d}(x) + \mathrm{r}(x) = 1x^6 + 1x^5 + 0x^4 + 0x^3 + 0x^2 + 1x^1 + 0x^0
\end{aligned} \tag{4.6}$$

To check if the data was received correctly, (4.3) is used.

$$\begin{array}{rrrrrrrl}
1x^6 & +1x^5 & +0x^4 & +0x^3 & +0x^2 & +1x^1 & +0x^0 & : \mathrm{g}(x) = x^3 + x^2 + x \\
-1x^6 & -0x^5 & -1x^4 & -1x^3 & & & & \\
 & 1x^5 & 1x^4 & 1x^3 & +0x^2 & & & \\
 & -1x^5 & -0x^4 & -1x^3 & -1x^2 & & & \\
 & & +1x^4 & 0x^3 & +1x^2 & +1x^1 & & \\
 & & -1x^4 & -0x^3 & -1x^2 & -1x^1 & & \\
\hline
 & & & & & & 0 & \rightarrow \text{OK}
\end{array} \tag{4.7}$$

The same calculation as (4.5) can be carried out easier using XOR in the bit representation.

$$\begin{array}{cccccccc}
 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
\oplus & 1 & 0 & 1 & 1 & & & \\
 & & 1 & 1 & 1 & 0 & & \\
\oplus & & 1 & 0 & 1 & 1 & & \\
 & & & 1 & 0 & 1 & 0 & \\
\oplus & & & 1 & 0 & 1 & 1 & \\
\hline
 & & & & & & 1 & 0
\end{array} \tag{4.8}$$

The CRC check is conducted analogue to (4.3).

$$\begin{array}{cccccccc}
 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
\oplus & 1 & 0 & 1 & 1 & & & \\
 & & 1 & 1 & 1 & 0 & & \\
\oplus & & 1 & 0 & 1 & 1 & & \\
 & & & 1 & 0 & 1 & 1 & \\
\oplus & & & 1 & 0 & 1 & 1 & \\
\hline
\text{OK} & & & & & & & 0
\end{array} \tag{4.9}$$

Some CRC implementations invert the data stream before calculation, invert the checksum or extend the data polynomial in a different way.

Notice that a CRC check is not a definitive method to find out if the transmission was free of errors.

4.2 Advanced Encryption Standard (AES)

Secure transmission of confidential data via public networks requires an appropriate encryption. One of the most important encryption algorithms that are considered to be secure at present [29, pp. 145–146] is called AES. It is a special case of the Rijndael cipher that was developed by Vincent Rijmen and Joan Daemen. AES basically makes use of a *private* key to transform the plaintext to the ciphertext. [29, pp. 137–138]

In particular, AES is a symmetric block cipher with a block size of 128 bit and key lengths of 128, 192 or 256 bit [29, p. 138]. Symmetric methods use the same key for encryption and decryption, while asymmetric methods like Rivest-Shamir-Adleman (RSA) use different keys for encrypting and decrypting a message [3, pp. 76–77]. In data transfer asymmetric methods, also known as public key ciphers, are often used to enable the secure exchange of a symmetric key. Applying RSA, every client has a private and public key pair. The public key, as the name implies, is available for everyone, while the private key is only known by the client owning it. If the public key is used to encrypt data, the corresponding private key has to be used for decryption and vice versa. To *begin* a secure data transfer between two clients, e. g. peer A and peer B, who do not have any information about each other, at first they have to exchange their public keys. For this a secure connection is not required, since public keys are not sensible. To secure a message, peer A now uses peer B's public key to encrypt it and transmits it to peer B. To decrypt the message, peer B's private key is required. Since only peer B itself knows it, only peer B can decrypt the message. For exactly the same scenario a symmetric method could not be applied, because peer A and peer B would both have to know the same (secret) key. Symmetric methods like AES are less computation-intensive. Additionally, they can be implemented in hardware easier. Therefore, an economic secure connection can be achieved by using RSA to exchange a secret AES key and then using AES for further communication. [3, pp. 133–134 137]

Block ciphers in general map plaintext blocks with a *fixed* length to ciphertext blocks with the same length. To encrypt an arbitrary length plaintext, a mode of operation has to be selected. Usually the plaintext has to be padded in order that it's length is an integer multiple of the block length to enable splitting into block length sized portions. In contrast, stream ciphers map plaintexts with an arbitrary length to ciphertexts. [3, pp. 68–70 77]

The AES algorithm can be described according to Fig. 4.1. For encryption and decryption, the method performs several transformation rounds. Depending on the key length, the number of rounds n is determined. For key lengths of 128, 192 or 256 bit, respectively 10, 12 or 14 rounds are used. For n rounds $(n+1)$ subkeys are required, which are derived from the main key in the `KeyPreparation` operation. The so called state matrix T, to which all cryptographic operations are applied consecutively as well as the subkeys K_i, that are derived from the main key K are both represented by 4x4-matrices. Given a plaintext block $\boldsymbol{p} = \begin{pmatrix} T_0 & T_1 & \dots & T_{15} \end{pmatrix}^\mathrm{T}$ and a main key $\boldsymbol{k} = \begin{pmatrix} k_0 & k_1 & \dots & k_{4L-1} \end{pmatrix}^\mathrm{T}$ with length L in 4-byte words, the initial state matrix T and the 4xL-main key matrix K are assembled according to (4.10). [29, pp. 138–142]

$$\mathsf{T} = \begin{bmatrix} T_{00} & T_{04} & T_{08} & T_{12} \\ T_{01} & T_{05} & T_{09} & T_{13} \\ T_{02} & T_{06} & T_{10} & T_{14} \\ T_{03} & T_{07} & T_{11} & T_{15} \end{bmatrix} \quad \mathsf{K} = \begin{bmatrix} k_{00} & k_{04} & \dots & k_{4L-4} \\ k_{01} & k_{05} & \dots & k_{4L-3} \\ k_{02} & k_{06} & \dots & k_{4L-2} \\ k_{03} & k_{07} & \dots & k_{4L-1} \end{bmatrix} \tag{4.10}$$

For this chapter additions shall be interpreted as bitwise XORs that are denoted as $\mathsf{A} \oplus \mathsf{B}$. Multiplications shall be defined according to (4.11) and denoted with e. g. $2 \bullet a$. It is sufficient to define multiplications with 1, 2 and 3. Matrix multiplications are carried out as usual, using the addition and multiplication definitions from above respectively. Note the difference between rotation and shifting. Every cell of the state matrix consists of 1 B data which has 8 digits in binary representation. Therefore $0b01000000 \ll 1 = 0b10000000$, $0b10000000 \ll 1 = 0b00000000 \neq 0b00000001$.

$$\begin{aligned} 1 \bullet a &= a \\ 2 \bullet a &= \begin{cases} (a \ll 1), & a < 128 \\ (a \ll 1) \oplus 0b00011011, & a \geq 128 \end{cases} \\ 3 \bullet a &= (2 \bullet a) \oplus a \end{aligned} \tag{4.11}$$

The cipher uses four operations that are performed consecutively. `AddRoundKey` adds the particular subkey $\mathsf{K_i}$ to the state matrix T, which is the plaintext at the initial application, see (4.12). [29, p. 141]

$$\texttt{AddRoundKey}\,\mathsf{T} = \mathsf{T} \oplus \mathsf{K_i} \tag{4.12}$$

`SubBytes` substitutes each byte of the state matrix with a corresponding byte from a substitution box, which maps each character uniquely to another, see appendix A.1. [29, pp. 139–140]

`ShiftRows`, (4.13), rotates each row of the temporary buffer left by the number of bytes equal to the row's position in the matrix, starting with 0 for the first row. [29, p. 140]

$$\texttt{ShiftRows}\,\mathsf{T} = \begin{bmatrix} T_{00} & T_{04} & T_{08} & T_{12} \\ T_{05} & T_{09} & T_{13} & T_{01} \\ T_{10} & T_{14} & T_{02} & T_{06} \\ T_{15} & T_{03} & T_{07} & T_{11} \end{bmatrix} \tag{4.13}$$

`MixColumns`, (4.14), is a matrix operation that basically mixes columns and is responsible for diffusion. [29, pp. 140–141]

$$\texttt{MixColumns}\,\mathsf{T} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \bullet \mathsf{T} \tag{4.14}$$

The `KeyPreparation` can be described according to Fig. 4.2. It is initialized with the column vectors of the main key K, (4.15), in iteration 0.

$$\mathsf{K} = \begin{pmatrix} \boldsymbol{k}_0 & \boldsymbol{k}_1 & \dots & \boldsymbol{k}_{L-1} \end{pmatrix} \tag{4.15}$$

Since groups of four adjacent vectors $\boldsymbol{k_s}$ describe a subkey $\mathsf{K_i}$, $(n+1)$ subkeys are required and every iteration generates L vectors, $m = \text{ceil}\,(4 \cdot (n+1)/L)$ iterations have to be carried out.

The operations used for preparing the subkeys were already explained, except `RCon(i)`, which is the round constant and is defined in (4.16).

$$\begin{aligned} \texttt{RCon(0)} &= \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix}^{\mathrm{T}} \\ \texttt{RCon(i)} &= \; 2 \bullet \texttt{RCon(i-1)}, \quad i \geq 1 \quad i \in \mathbb{N} \end{aligned} \tag{4.16}$$

Subkeys then can be extracted corresponding to (4.17).

$$K_i = \begin{pmatrix} \boldsymbol{k}_{4i} & \boldsymbol{k}_{4i+1} & \boldsymbol{k}_{4i+2} & \boldsymbol{k}_{4i+3} \end{pmatrix}, \quad 0 \leq i \leq n \tag{4.17}$$

AES maps a 128 bit ($\hat{=}$ 16 B $\hat{=}$ 16 characters) fixed-length plaintext to a 128 bit ciphertext. To actually apply the cipher to a plaintext with an arbitrary length, a mode of operation has to be introduced. The simplest one is Electronic Codebook (ECB), where the plaintext is padded to a length that is an integer multiple of 16 B. The padded plaintext is then split up to 16 B blocks which are encrypted independently. [3, p. 69]

A more sophisticated mode of operation is Cipher Block Chaining (CBC), see Fig. 4.3. CBC requires a 16 B Initialisation Vector (IV) that can be chosen randomly but should not be used a second time. As discussed for ECB, CBC also has to be padded. The IV is added (XOR) to the first block of the plaintext. The result is encrypted and becomes to the first block of the ciphertext. Beginning from the second block of the plaintext, the previous ciphertext block is used instead of the IV. [29, pp. 71–74]

Another cipher mode of operation is Counter (CTR). CTR generates cipher blocks by consecutively encrypting the value of a counter and incrementing it afterwards. These cipher blocks are concatenated and from a key stream with a length that is an arbitrary integer multiple of 16 B. For encryption of e. g. a 19 B plaintext, the first 19 B of the key stream are simply added (XOR) to the plaintext to translate it to the ciphertext. While CBC is decrypted by applying the same operations

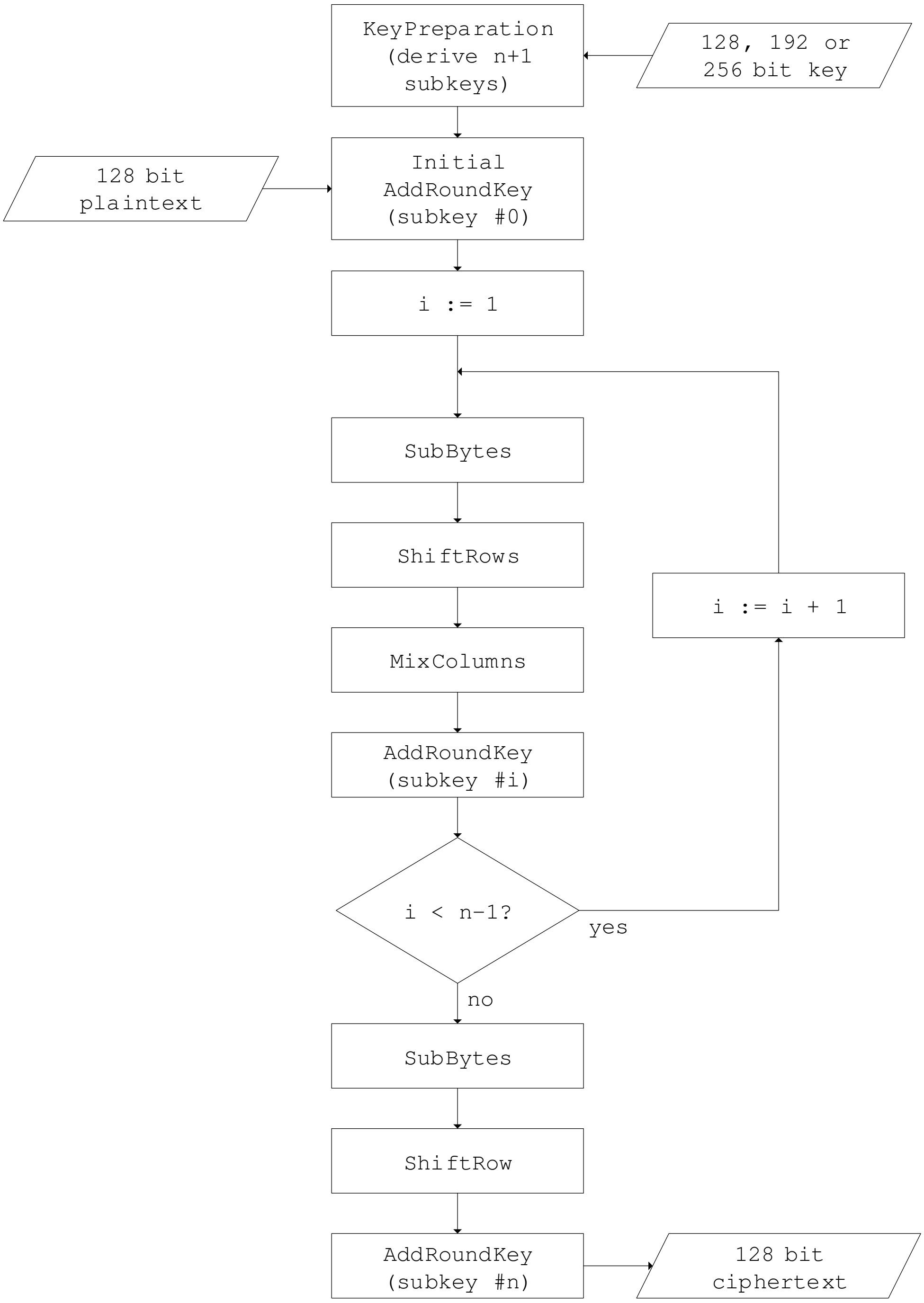


Fig. 4.1. The Advanced Encryption Standard (AES) ciphering procedure uses n transformation rounds and $(n+1)$ subkeys to encrypt the plaintext. [29, pp. 138–141]

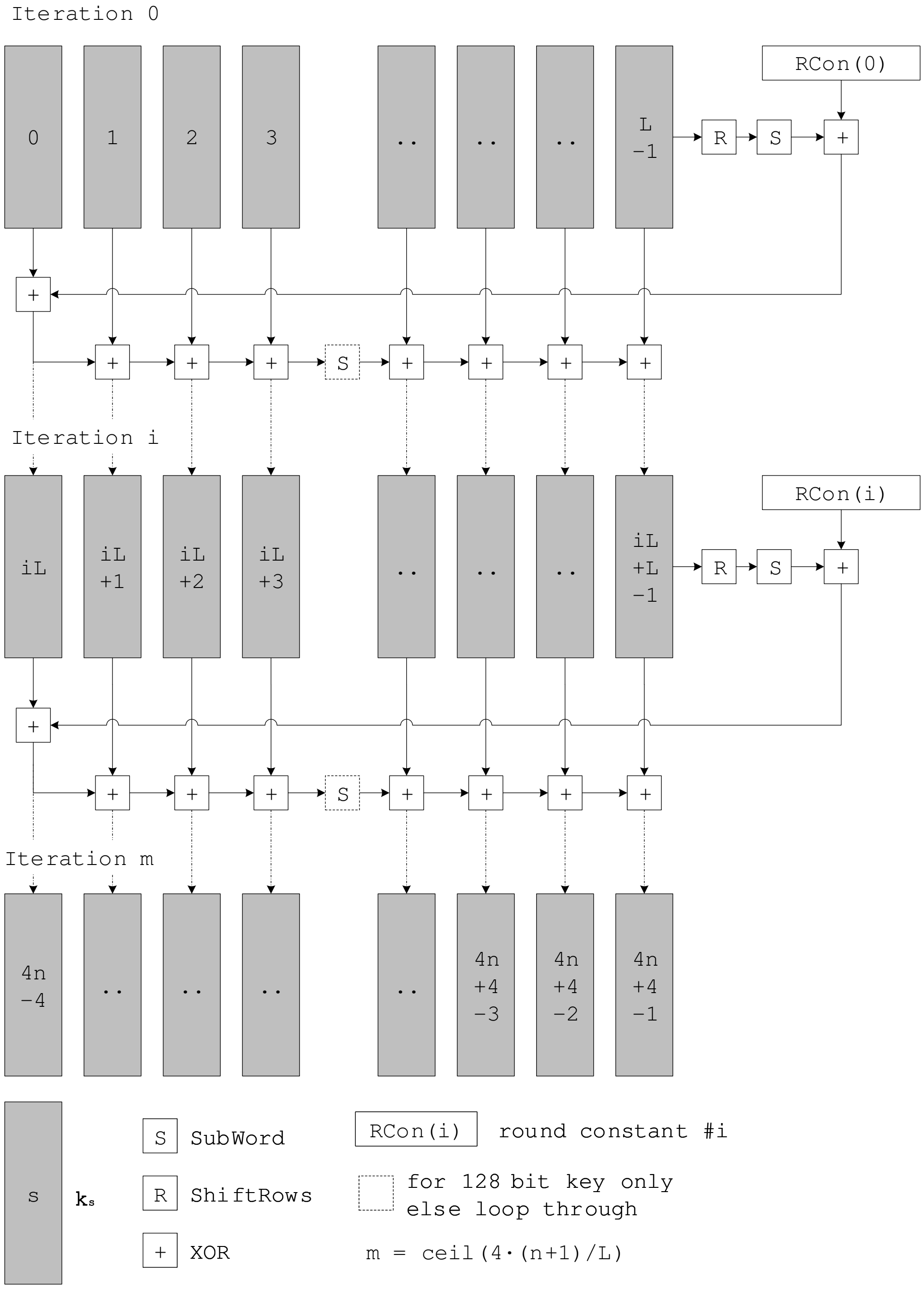


Fig. 4.2. The Advanced Encryption Standard (AES) subkey preparation algorithm for 128, 192 as well as for 256 bit key length is the first step in the encryption algorithm.

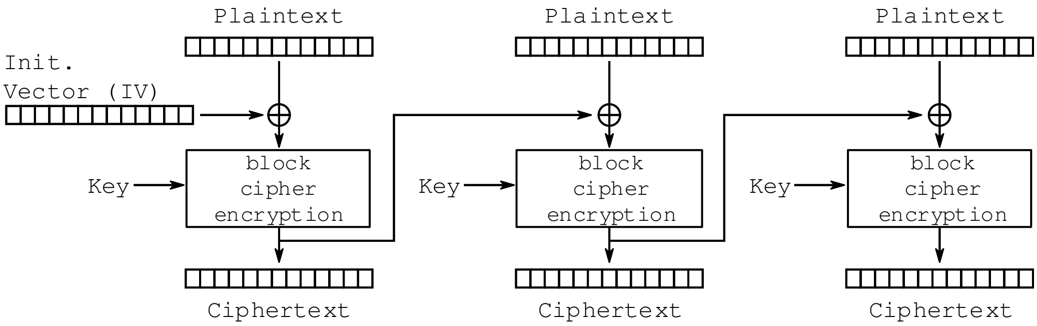


Fig. 4.3. The Advanced Encryption Standard (AES) cipher block chaining mode of operation is used for plaintexts that are longer than 128 bit. [3, p. 73][36]

required for encryption in reverse order, CTR is encrypted and decrypted by exactly the same algorithm. [22, p. 99]

5 Sub-GHz Frequency Band

Sub-GHz refers to frequency bands below 1 GHz in general. The term is vague defined, nevertheless most often Industrial Scientific Medical (ISM) frequency bands or bands that are allocated to so called short-range devices are meant. Especially those frequency bands that do not require a licence are relevant for IIoT tasks.

Popular protocols, such as IEEE 802.15.4 and Wireless M-Bus use the 868 MHz to 870 MHz band in Europe and the 902 MHz to 928 MHz band in USA. Limits for 868 MHz to 870 MHz sub-bands are given in Tab. 5.1. [9, pp. 277–313]

The aim of this chapter is to show why the sub-GHz frequency band is attractive for IIoT applications in general and which protocol best suits the needs in condition monitoring.

5.1 Benefits and Drawbacks

In this section benefits and drawbacks of the sub-GHz frequency band compared to the 2.4 GHz band with respect to data rate, range and current consumption are elaborated. In particular, BLE is a strong competitor of sub-GHz standards, because it also offers reasonable data rates paired with low power consumption. This comparison should be carried out from a general point of view that is not limited to a certain protocol.

Lower Frequency	Upper Frequency	Power Limit	Max. Duty Cycle[1]
MHz	MHz	mW	%
868.0	868.6	25	1.0
868.7	869.2	25	0.1
869.4	869.65	500	10.0
869.7	870.0	5	100.0

Tab. 5.1. The tabulated frequency bands [7], amongst others, can be used for short-range devices in Europe without licence. They are applied in IEEE 802.15.4 and Wireless M-Bus.

[1] The duty cycle limit refers to active transmission time and must be obtained within a span of 1 h at any time [7].

5.1.1 Transmission Range

A wireless transmission between a transmitting antenna and a receiving antenna in free space can be modeled by the Friis equation, see (5.2). In this equation P_t and P_r represent power fed into the transmitting antenna respectively power available at the output of the receiving antenna. G_t and G_r are power gain factors of the transmitting and receiving antennas. λ is the wavelength of the radio wave and d the distance between the antennas. [28, pp. 183–185][10]

The Friis transmission equation,

$$\frac{P_r}{P_t} = G_r G_t \left(\frac{\lambda}{4\pi d}\right)^2, \tag{5.1}$$

is valid for distances $d \gg 2a^2/\lambda$, with the largest linear dimension of either of the antennas a. It omits absorption effects of the transmission medium as well as effects of the ground. [10]

The wavelength λ can be expressed as the velocity of light c divided by the frequency f, yielding

$$\frac{P_r}{P_t} = G_r G_t \left(\frac{c}{4\pi d f}\right)^2. \tag{5.2}$$

Based on the Friis transmission equation, a more realistic model for transmission, the link budget, can be derived, see (5.4). It includes a more *realistic* path loss L_p based on a factor n that can be found in [24], see (5.3).

$$L_p = \left(\frac{\lambda}{4\pi}\right)^2 \cdot \left(\frac{1}{d}\right)^n \tag{5.3}$$

Furthermore transmit-chain losses L_t and receive-chain losses L_r at connectors and cables as well as a margin L_m for fluctuations of n are considered.

$$P_r = \frac{P_t \cdot G_t \cdot G_r}{L_p \cdot L_t \cdot L_r \cdot L_m} \tag{5.4}$$

Atmel Corp. [24] recommends a minimum margin L_m of 15 dB. Equation (5.4) is also often formulated in absolute levels.

Note that the greatest portion of power is transmitted within an ellipsoidal shaped region between sender and receiver, the first Fresnel zone. Therefore this region should be kept free from obstacles. The maximum radius of the ellipsoid is proportional to $\sqrt{\lambda}$. [38]

The Friis transmission equation (5.2) can be used to derive the ratio between the power at the transmitter and at the receiver. It is proportional to $1/f^2$ for a given distance d. Thus, the lower the frequency, the lower is it's attenuation for a certain distance. BLE operating at 2.4 GHz for example, compared to a sub-GHz technique operating at 868 MHz, would require a transmission power that is $(2400/868)^2 \approx 7.6$ times higher in order to reach the same distance, assuming an equal sensitivity level and antenna gain factors of 1. Using equal transmission powers for both would lead to a $(2400/868) \approx 2.8$ times higher transmission range for sub-GHz. Transmission power for BLE is limited with 100 mW [7], while transmission power limits for sub-GHz depending on the frequency band can be found in Tab. 5.1, e. g. 5 mW to 500 mW.

Another significant parameter for transmission range is the receiver sensitivity. It characterizes the minimum power that is required at the receiver to distinct data from noise. Receiver sensitivity decreases with data rate. This relation can be evaluated for the Texas Instruments CC1352P dual-band MCU in [4] for example and also for STMicroelectronics devices in [30] and [31]. Considering the Texas Instruments device, only the slowest BLE mode working with data rates of $125\,\mathrm{kbit\,s^{-1}}$ achieves a sensitivity that is comparable to sub-GHz, while the fastest BLE mode allowing data rates of $2\,\mathrm{Mbit\,s^{-1}}$ requires more than 20 times the power to barely receive a packet.

Transmission range is a major benefit of sub-GHz technology.

5.1.2 Data rate

Data rate in general has a strong dependence on the specified protocol and the used mode. As a typical limit [9] estimates data rates of $100\,\mathrm{kbit\,s^{-1}}$ for protocols in the sub-GHz ISM band, while BLE enables data rates up to $2\,\mathrm{Mbit\,s^{-1}}$. Another drawback of sub-GHz is that the duty cycle is limited within most bands, see Tab. 5.1, while it is not for BLE. This limits the amount of data that can be transmitted at a time.

Assuming a transmission with a carrier frequency of 868.95 MHz, the maximum duty cycle is limited to 1 % per hour, that is 3.6 s. With a data rate of $100\,\mathrm{kbit\,s^{-1}}$ per hour only $100\mathrm{kbit\,s^{-1}} \cdot 3.6\,\mathrm{s} = 360\mathrm{kbit} = 45\,\mathrm{kB}$ can be transmitted.

5.1.3 Current consumption

The current consumption for an equal effective radiated power is higher for higher frequencies. The higher losses are caused by faster switching in radio circuitry. Texas Instruments e. g. specifies transmission current of the CC1352P MCU for 20 dBm with 63 mA at 915 MHz and 85 mA at 2.4 GHz[2] [4].

5.1.4 Interference

A benefit of using sub-GHz bands is that there are only a few devices operating on it at present, thus, interferences are rare. This can be a huge issue with 2.4 GHz technologies since many devices, such as Wi-Fi routers, Bluetooth and microwaves are using it. In environments where a lot of people have enabled Bluetooth on their mobile phone, e. g. at exhibitions, reliability of the connection can get lost. Crosstalk with mobile telephone systems is also possible.

[2] The current measure of the 915 MHz frequency transmission refers to a supply current of 3 V, while the current measure of 2.4 GHz refers to 3.3 V.

5.1.5 Conclusion

There are many aspects that have to be taken into account for selecting an appropriate low-power wireless transmission technology. The most important ones are transmission range, data rate, current consumption and interference which all together decide the amount of energy that is required for transmission of a particular block of data. Lower transmission power for same distances together with outstanding sensitivity as well as fair current consumption are the key features of sub-GHz technologies. Since smart sensors have capabilities to do computations on their own and components such as bearings do not have to be monitored permanently, data rate is not the limiting factor in many cases and therefor sub-GHz perfectly suits the needs. If a faster data rate is required or the duty cycle limit can't be satisfied BLE might be the network standard of choice.

5.2 Avaliable Standards

Sub-GHz networking has recently become popular. There are many protocols already available; however, there are hardly any devices on the market using them. Therefore, it is difficult to say which of them will take hold.

Furthermore there are a lot of proprietary protocols that shall not be dealt with in this thesis, since they would very likely limit hardware selection to the particular manufacturer.

5.2.1 IEEE 802.15.4 PHY/MAC

The IEEE 802.15.4 network standard defines a physical and a media access control layer, the latter being a sublayer of the data link layer in the OSI model (see Chapter 3.1). It is a packet-based approach that basically supports point-to-point or star topology. Furthermore AES-128 encryption is supported and 2^{16} nodes can be addressed. IEEE 802.15.4 is not limited to sub-GHz, it can also be used with 2.4 GHz frequency bands. [9, pp. 277–279]

IEEE 802.15.4 is used as the basis for higher level wireless standards, such as Thread, 6LoWPAN, Zigbee and various proprietary protocols.

6LoWPAN

6LoWPAN is a low-power *wireless private area network* that enables transmission of IPv6 protocols over IEEE 802.15.4 networks. It adds network and transport layers on top of the IEEE 802.15.4 stack and specifies data rates of 20 kbit s^{-1} for the 868 MHz frequency band. [14]

The introduction of Internet Protocol (IP) to smart sensor allows to use existing well-proven infrastructure as well as already existing tools for management and diagnostics. It allows connecting IIoT devices to the Internet. [14]

To reduce configuration overhead to a minimum, a stateless address auto configuration method is specified. [15]

Thread

Thread is an open standard for Connected Home applications, featuring IP-based networking. It uses 6LoWPAN which in turn is based on IEEE 802.15.4. [33, p. 27]

Though being based on the IEEE 802.15.4 physical layer, a physical interface conforming to the 2450 MHz specifications of IEEE 802.15.4 has to be supported. Therefore at the moment it can't be used for a sub-GHz physical layer. [33, p. 44]

It is likely that further versions of Thread will support sub-GHz technology.

Zigbee

Zigbee is a network standard for industrial monitoring, home networks and wireless remote control. It is developed by the Zigbee Alliance and requires a licence for sale of commercial products. [9, pp. 309–311]

Zigbee is based on IEEE 802.15.4 and supports the 868 MHz frequency band with a data rate of $20\,\mathrm{kbit\,s^{-1}}$. [9, pp. 309–311]

5.2.2 LoRaWAN

The term LoRaWAN is an abbreviation for *long range wide area network*. It is specified by the LoRa Alliance and uses a proprietary modulation technique called LoRa. [17]

Using LoRaWAN, the smart sensor communicates with an arbitrary LoRa gateway that is in range which passes the data to a LoRaWAN server via the Internet. The server processes the data and manages devices. [17]

LoRaWAN enables data rates up to $50\,\mathrm{kbit\,s^{-1}}$ and reaches very high transmission ranges up to 40 km [39]. LoRaWAN is intended for the use in *smart cities*.

5.2.3 Wireless M-Bus

M-Bus is an abbreviation for *meter bus* and is used for remote reading of gas, water and electricity meters. Wireless M-Bus is the radio variant of M-Bus and is specified in the ÖNORM EN 13757-4 standard, see [22]. [32, p. 10]

Wireless M-Bus allows data rates of up to 66.67 $\mathrm{kbit\,s^{-1}}$, depending on the mode used [22].

5.3 Selection of a Sub-GHz Standard

Especially for the use in IIoT applications, a standard that has already taken hold shall be selected. In this case the decision was made for Wireless M-Bus.

Major reasons for selecting Wireless M-Bus were the availability of industry-suited components on the edge gateway side as well as availability of sensors that use the standard. It's already in use for e. g. drive-by readouts of electricity meters.

Wireless M-Bus is licence-free and defined in a European Standard, which indicates a well-proven standard.

Additionally, compared to standards based on IEEE 802.15.4 for example, the implementation of Wireless M-Bus is simpler, offers higher data rates and is also designed for low power consumption.

6 Wireless M-Bus

Very basic features of Wireless M-Bus and reasons why it was selected were already discussed in Chapter 5. In this chapter the network standard and it's features shall be discussed more in detail, especially the mode T which was chosen to be used in this thesis.

In general, the specification of Wireless M-Bus differentiates between a *meter*[1] and a *other* that it communicates with.

6.1 Modes

Depending on the application, an appropriate Wireless M-Bus mode has to be selected, see Tab. 6.1. Modes are named by a letter together with a number. While the letter is an abbreviation for the intended purpose, the number "1" or "2" differentiates between a unidirectional[2] or bidirectional transmission mode respectively.

The mode S is for use with stationary devices. In this case it is usually sufficient to transmit in intervals reaching from minutes to hours due to the fact that the other device can be assumed to be in range. In contrast, mode T is intended for frequent transmissions in order to enable driving or walking by other devices to capture data. Thus, transmission intervals are in the scale of seconds. [22, p. 12]

Mode	Frequency	Deviation	Encoding	Chip rate	Data rate	Max. Duty Cycle
	MHz	kHz		$\text{kbit}\,\text{s}^{-1}$	$\text{kbit}\,\text{s}^{-1}$	%
S1	868.3	50.0	M	32.768	16.38	0.02
S2	868.3	50.0	M	32.768	16.38	1
T1	868.95	50.0	3-of-6	100.0	66.67	0.1
T2-TX	868.95	50.0	3-of-6	100.0	66.67	0.1
T2-RX	868.3	50.0	M	32.768	16.38	0.1
R2	868.33	6	M	4.8	2.4	1

Tab. 6.1. Properties of some of the Wireless M-Bus modes defined in [22]. In this table "M" denotes manchester encoding.

[1] A meter refers to a measurement-meter not a unit of length.

[2] Unidirectional refers to a *meter* that can transmit but not receive.

In order to reduce power in bidirectional modes, usually communication begins with a periodical message from the sensor. This allows the sensor to enter a *low power deep sleep* mode instead of permanently waiting to receive commands, while the other is assumed to be reachable at any time. For transmissions from the other device to the meter, the former has to wait for one of the latter's periodical messages. After sending a periodical message, the meter is waiting to receive a command from the other device for a defined time span. The other can use this time frame to send it's command. [22, p. 12]

To allow sending commands from the other device to the sensor with reasonable delays, the mode T2 is appropriate.

6.2 Coding

Depending on the Wireless M-Bus mode either manchester or 3-of-6 encoding is used. Coding is used to occupy a narrower baseband width with a direct component that is vanishing. It avoids long sequences of “0” or “1” which leads to a more reliable connection. [22, p. 19]

6.2.1 Manchester

To manchester encode a stream of bits, each bit that is “0” is represented by the sequence “10” and each bit that is “1” by a sequence “01”.

6.2.2 3-of-6 Constant-Weight Code

A more efficient encoding technique than manchester is the 3-of-6 constant-weight code. 3-of-6 refers to the fact that in the encoded representation each word consists of three “1” and three “0” bits. [22, pp. 22–23]

To encode, each 4-bit nibble of data is replaced by a corresponding 6-bit word, see Tab. A.2.

6.3 Network Layers

In this section the Wireless M-Bus layers that are required for networking, such as the data link layer and the transport layer are presented. Note that Wireless M-Bus does not define all seven layers of the OSI-Model.

C-Field	Name	Direction	Function	Confirmation
0x0	SND-NKE	To meter	Link reset after communication	-
0x3	SND-UD	To meter	Send user data	ACK/NACK/ RSP-UD
0x4	SND-NR	From meter	Send data without request (no reply)	-
0x*B*	REQ-UD2	To meter	Request user data	ACK/RSP-UD

Tab. 6.2. C-Field codes, amongst others, used in primary stations. Compare table to [22, p. 41].

C-Field	Name	Direction	Function	Initiated by
0x0	ACK	Both directions	Acknowledge reception	SND-UD
0x1	NACK	From meter	Not acknowledge packet if it is invalid	SND-UD
0x8	RSP-UD	From meter	Response of data after request	REQ-UD2

Tab. 6.3. C-Field codes, amongst others, used in secondary stations. Compare table to [22, p. 42].

6.3.1 Physical Layer

The physical Wireless M-Bus layer makes use of FSK or GFSK modulation (see Chapter 3.2) with the carrier frequency and frequency deviation defined in Tab. 6.1. Before an actual datagram, a synchronization pattern and a preamble corresponding to the selected mode is transmitted.

6.3.2 Data Link Layer

Beginning with the data link layer, a Wireless M-Bus packet is arranged in blocks. The data link layer is always the first block, see Tab. 6.4. The second block can be an extended link layer or a transport layer and is arranged according to Tab. 6.5. If the second layer exceeds the length limit or further layers follow, the structure defined in Tab. 6.6 is used. It is distinguished between the primary and secondary station, while the primary station is the station that initialized the bidirectional communication. Usually this is the other device.

The data link layer defines the overall length L of the packet, including CRC-Fields but not the length field itself. In the C-Field the function of the message is specified, which determines e. g. if a confirmation is required, see tables 6.2 and 6.3. Different codes are used in primary and secondary stations. Fields M and A are representing a manufacturer ID and the address of the device respectively. The CRC-Field contains the CRC checksum[3] of the particular block, see Chapter 4.1. [22, pp. 38–43]

[3] The generator polynomial used in Wireless M-Bus CRC-Checks is $x^{16}+x^{13}+x^{12}+x^{11}+x^{10}+x^{8}+x^{6}+x^{5}+x^{2}+1$. [22, p. 80]

L-Field	C-Field	M-Field	A-Field	CRC-Field
1 B	1 B	2 B	6 B	2 B

Tab. 6.4. Frame format A of the first Wireless M-Bus block in a packet. [22, p. 39]

CI-Field	Data Field	CRC-Field
1 B	(((L-9) modulo 16)-1) byte if it is the last block; else 15 B	2 B

Tab. 6.5. Frame format A of the second Wireless M-Bus block in a packet. [22, p. 39]

All further layers are represented by one or more additional blocks, see Tab. 6.6. The CI field specifies the type of the following protocol and therefore how the following bytes have to be interpreted.

6.3.3 Transport Layer

The transport layer defines the address of the meter that is communicated to and a status. A configuration field defines if AES-128 CBC encryption is used, see Chapter 4.2, and defines the meter's receiving behaviour. The transport layer defines an access counter that is used to avoid receiving a message twice and to assign responses to particular requests. If a request is sent and therefore a bidirectional communication initiated, a so called frequent access cycle is started. In frequent access cycle, meter and other both repeat their previous message as long as it is not acknowledged by a new command or terminated. A timeout is used additionally to end the frequent access cycle if one of the peers does not correctly terminate the connection. An example for a bidirectional communication is given in Fig. 6.1.

6.3.4 Application Layer

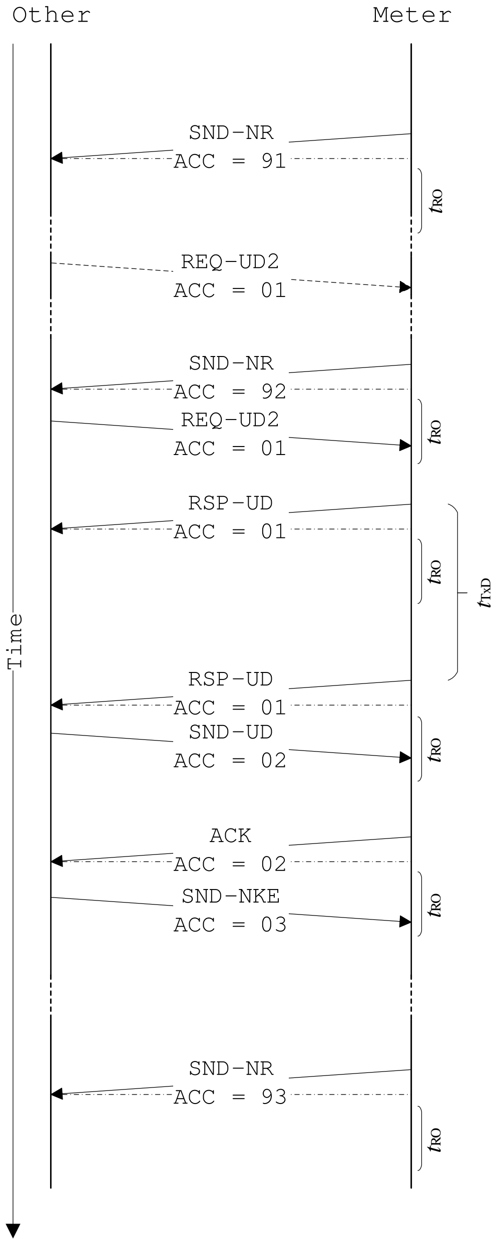
The application layer handles actual data that shall be transmitted.

6.4 Evaluation of a Packet

Since Wireless M-Bus uses encoding and encryption, manual parsing of transmitted data is very expensive. To simplify parsing of Wireless M-Bus packets, a tool was developed that can be used

Data Field	CRC-Field
((L-9) modulo 16) byte if it is the last block; else 16 B	2 B

Tab. 6.6. Frame format A of optional Wireless M-Bus blocks in a packet. [22, p. 39]



The meter periodically sends it‘s status to provide a reception time frame t_{RO}.

The other tries to request data, but fails because the meter is not receiving at the moment.

The meter sends a periodical message, while other uses the reception time frame to request data again.

The meter now receives the request and responds with data. It enters the *frequent access cycle*. The other does not send it‘s next message (or link reset) within the reception time frame because it is busy with processing data.

Being in frequent access cycle, the meter repeats the last message until it is acknowledged by a new message or a link reset. Repeated messages are delayed by t_{TxD}. Other sends it‘s next command.

The meter acknowledges the command and the other resets the link which also terminates the frequent access cycle.

The meter continues with periodically sending status.

Fig. 6.1. Bidirectional Wireless M-Bus communication. Compare to [22, p. 88].

to encode and decode data with either manchester or 3-of-6 constant-weight code. Furthermore the tool can generate CRC checksums as well as encrypt or decrypt data using AES-128 with CBC mode of operation, see Figure 6.2.

MBusTool was implemented using C++ with Qt, therefore many methods could be reused in the firmware implementation.

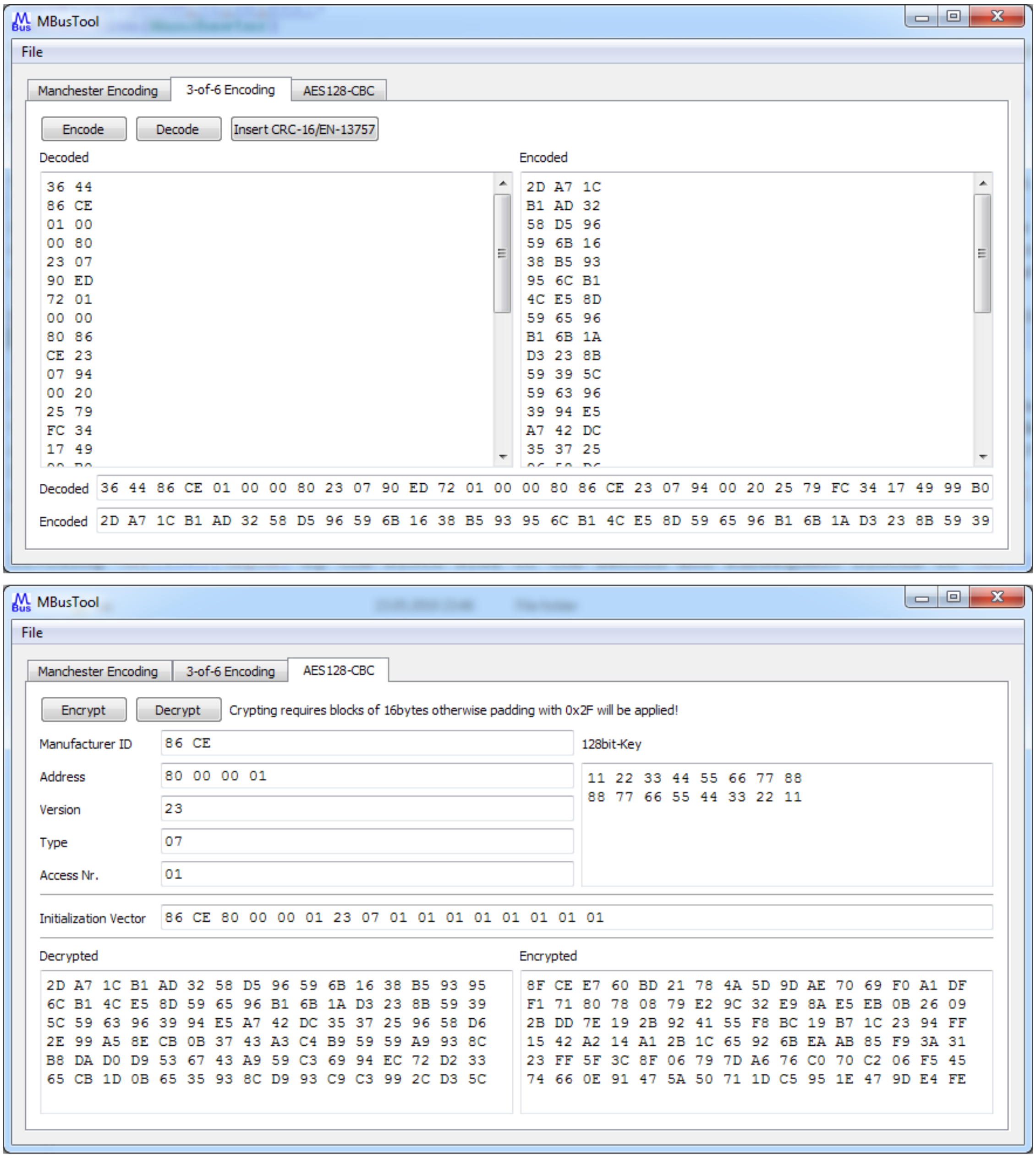
6.5 Effective Data Rate

Data rates that are specified in Tab. 6.1 refer to the transmission rate that does not take protocol headers and CRC checksums in account that reduce the rate at which user data can effectively be transmitted. The effective data rate shall be calculated for the Wireless M-Bus mode T2 for a datagram that is sent from the meter to the other device. The buffer of the transmitting MCU is assumed to have a size of 255 B. Due to the fact that 3-of-6 constant-weight coding is applied, effectively $\frac{2}{3} \cdot 255\,\mathrm{B} = 170\,\mathrm{B}$ of unencoded data can be transmitted. Subtracting the length of the data link layer of 12 B including the CRC checksum yields 158 B remaining.

Dividing 158 B by the block size of the second and subsequent blocks of 18 B including CRC checksums yields 8 blocks with a remainder of 14 B. The bytes left considering coding, data link layer size and CRC fields can be obtained by $8 \cdot 16\,\mathrm{B} + 12\,\mathrm{B} = 140\,\mathrm{B}$.

Subtracting the size of the transport layer, 5 B, and subtracting an assumed 1 B for a CI field and another 1 B for length of application data leads to 133 B remaining. Thus, the chip rate is reduced by a factor of $\frac{133}{255}$. This leads to an effective data rate of $\frac{133}{255} 100\,\mathrm{kbit\,s^{-1}} \approx 50\,\mathrm{kbit\,s^{-1}} = 6.25\,\mathrm{kB\,s^{-1}}$.

The calculation assumed that the buffer of the MCU is fully used and therefore the calculated effective data rate is only reached in the best case.



Decoded

```
36 44
86 CE
01 00
00 80
23 07
90 ED
72 01
00 00
80 86
CE 23
07 94
00 20
25 79
FC 34
17 49
```

Encoded

```
2D A7 1C
B1 AD 32
58 D5 96
59 6B 16
38 B5 93
95 6C B1
4C E5 8D
59 65 96
B1 6B 1A
D3 23 8B
59 39 5C
59 63 96
39 94 E5
A7 42 DC
35 37 25
```

Decoded `36 44 86 CE 01 00 00 80 23 07 90 ED 72 01 00 00 80 86 CE 23 07 94 00 20 25 79 FC 34 17 49 99 B0`

Encoded `2D A7 1C B1 AD 32 58 D5 96 59 6B 16 38 B5 93 95 6C B1 4C E5 8D 59 65 96 B1 6B 1A D3 23 8B 59 39`

Crypting requires blocks of 16bytes otherwise padding with 0x2F will be applied!

Manufacturer ID `86 CE`

Address `80 00 00 01`

Version `23`

Type `07`

Access Nr. `01`

128bit-Key

```
11 22 33 44 55 66 77 88
88 77 66 55 44 33 22 11
```

Initialization Vector `86 CE 80 00 00 01 23 07 01 01 01 01 01 01 01 01`

Decrypted

```
2D A7 1C B1 AD 32 58 D5 96 59 6B 16 38 B5 93 95
6C B1 4C E5 8D 59 65 96 B1 6B 1A D3 23 8B 59 39
5C 59 63 96 39 94 E5 A7 42 DC 35 37 25 96 58 D6
2E 99 A5 8E CB 0B 37 43 A3 C4 B9 59 59 A9 93 8C
B8 DA D0 D9 53 67 43 A9 59 C3 69 94 EC 72 D2 33
65 CB 1D 0B 65 35 93 8C D9 93 C9 C3 99 2C D3 5C
```

Encrypted

```
8F CE E7 60 BD 21 78 4A 5D 9D AE 70 69 F0 A1 DF
F1 71 80 78 08 79 E2 9C 32 E9 8A E5 EB 0B 26 09
2B DD 7E 19 2B 92 41 55 F8 BC 19 B7 1C 23 94 FF
15 42 A2 14 A1 2B 1C 65 92 6B EA AB 85 F9 3A 31
23 FF 5F 3C 8F 06 79 7D A6 76 C0 70 C2 06 F5 45
74 66 0E 91 47 5A 50 71 1D C5 95 1E 47 9D E4 FE
```

Fig. 6.2. MBusTool was developed as a convenience during the implementation of Wireless M-Bus. It features encoding and decoding, encryption and decryption using AES-128 CBC as well as CRC checksum calculation.

7 Programming Fundamentals

This chapter explains the basics of object-oriented programming and the way code is documented in this thesis. It should also deal with patterns, such as singleton and publish-subscribe, which often lead to very tidy and functional object-oriented code. Furthermore the concept of multithreading and the procedure of translating source code to machine code shall be introduced.

All remarks here refer to the C++ programming language, although being written as general as possible.

7.1 Object-Oriented Programming

While in traditional procedural programming the algorithm was the central difficulty, object-oriented programming has a strong focus on data. In object-oriented programming complex data structures and algorithms are broken down into simpler *classes*, which contribute to fulfilling the overall task. [2, pp. 11–12]

7.1.1 Classes

Objects, from a programmers point of view, are instances of classes. Classes consist of variables and functions, which are called attributes and methods respectively in this context. State and behaviour, thus, form a unit in object-oriented programming. [41, p. 265]

In class diagrams, classes are represented by a box where the first compartment contains the name of the class. The second compartment consists of the attributes and the third contains method declarations. For instances of the class the first compartment is underlined and contains the instances name and the class name separated by a colon. Methods and static attributes are not quoted again, since they are the same for all instances. [6]

Note the difference between a class and an instance or object of a class. A class is a *general* definition of the features that are represented, while an instance of it contains *specific* values representing a concrete object, see Figure 7.1. A class can be considered as a scheme according to which objects are constructed. Objects are created by calling a constructor method, which has the same name as the class. It is responsible for initializing the object. Before deletion, a so called destructor is called, which de-initializes the object. This could include e. g. deallocation of memory. The destructor has the same name as the class but with a leading ~. [41, pp. 285–295]

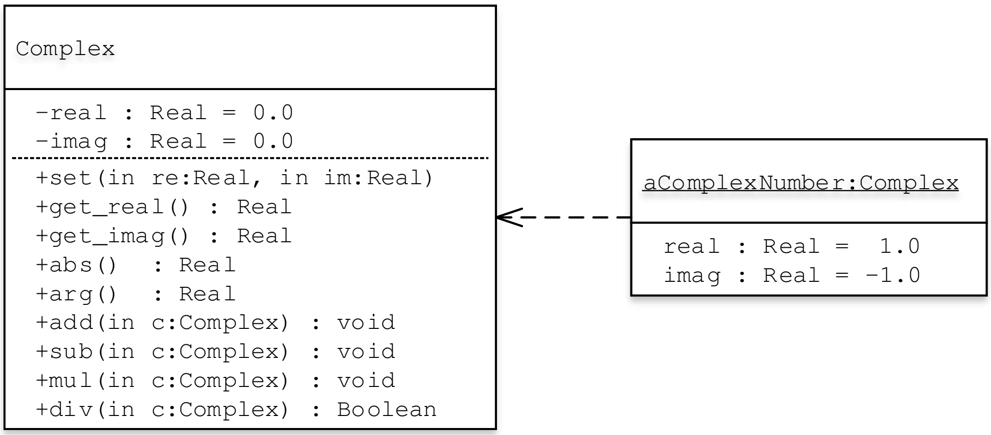


Fig. 7.1. A class representing complex numbers in general could consist of two private real number attributes, `real` and `imag`, which are both initialized with 0.0 on instantiation. Furthermore there are some public methods like `add`, `sub`, `mul` and `div` defined. `aComplexNumber` represents an instance of it. The instance represents a specific number, in this case $1 - \mathbf{i}1$, to which other `Complex` objects can be e. g. added or subtracted.

A class is defining visibility of data and functions. The keywords *private*, *protected* and *public* are used to specify if the particular attribute or method is visible from within the class only, also from within an inherited class or from everywhere. In class diagrams the specifiers are denoted by -, # and + respectively at the left side of the attribute's or method's name within the class box, see Figure 7.1. [41, pp. 272–273 407 845]

7.1.2 Static Attributes and Methods

In some cases attributes are required, which shall be *the same* for all instances, so called static attributes. They could be used e. g. to count the number of instances of a class, see Figure 7.2. Static attributes are analogue to global variables in procedural programming. To access a private static attribute, a member function that returns the attribute's value is required. Since the attribute is independent from an object, a get-method that is also independent from an object should be used. These methods are called static methods. Static attributes and static methods are underlined in class diagrams. [41, pp. 343–349]

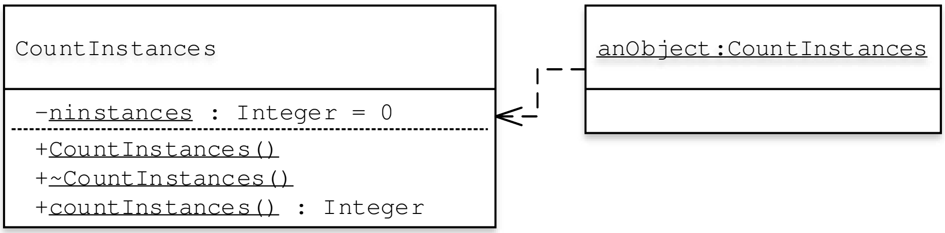


Fig. 7.2. `CountInstances` counts the number of its instantiated objects. The constructor `CountInstances()` implements incrementation of the static attribute `ninstances`, while the destructor `~CountInstances()` decrements it. The number of instances can be retrieved by calling the static method `countInstances()`.

7.1.3 Inheritance and Abstract Classes

A very efficient mechanism in object-oriented programming is inheritance. Derived classes inherit all attributes and methods of the base class, while only those that are marked as public or protected are visible for them. The base class is a more general representation of an object, while the derived class is an extension to a more specific object. This has the advantage that the common part of similar classes can be reused and only has to be written and debugged once. An example is depicted in Figure 7.3. Inheritance from more than one class as well as deriving from a derived class is possible. In class diagrams, inheritance is represented by a solid line with a triangle at the base class end. Note that every object from type of the derived class *is* an object from type of the base class, but not vice versa. [41, pp. 392–393 849]

Behaviour of methods that are defined in the base class can be overwritten in the derived class if they are originally declared virtual. Overwriting can be applied for each inherited class separately independent to other derived classes. Dynamic linkage takes care that the method that is overwriting the base class's definition is even used when the derived class object is casted to base class type. The fact that calling equal methods of classes pretending to be of the same (base class) type can trigger different implementations is called polymorphism. This feature is used e. g. to extend classes that are defined in pre-compiled libraries [41, p. 413]. Overwriting a virtual method is represented in class diagrams by simply defining a method in the derived class that exactly matches the definition of a base class's method. [41, pp. 414–420]

A common interface can be conceived by a base class that declares virtual dummy methods for the required functionality. Derived classes then overwrite the dummy methods with their specific implementation. To avoid the need for dummy methods, abstract methods can be used. Abstract methods are virtually declared but not defined, thus, a class containing one or more abstract methods cannot be instantiated. Such classes are called abstract classes. Classes that derive from an abstract class have to define every abstract method, else they are also abstract. Abstract class's

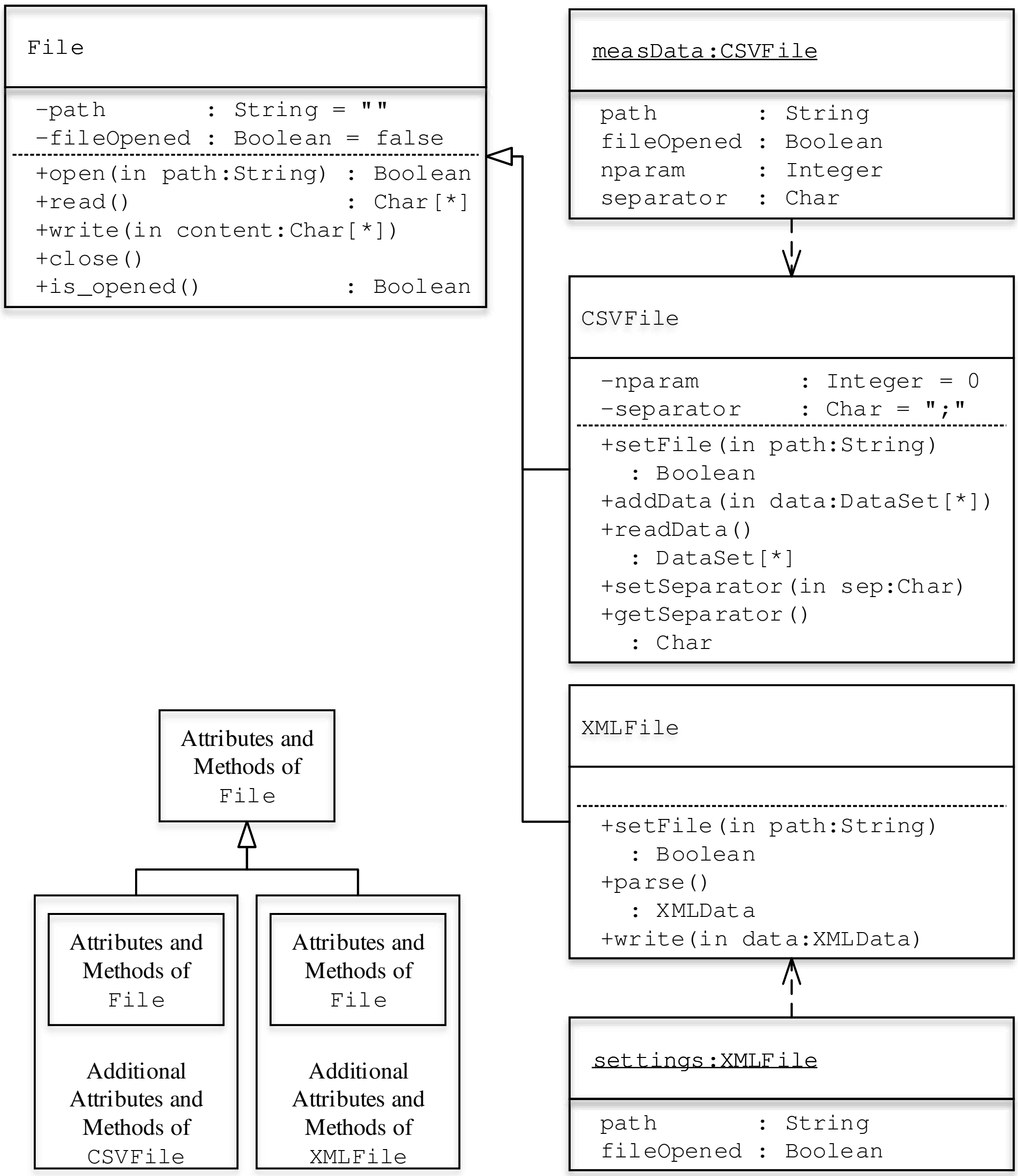


Fig. 7.3. The class `File` implements a very simple interface to handle files. The classes `CSVFile` and `XMLFile`, which also represent files in general, inherit from `File`. They extend the simple file interface by methods for easier handling of Comma Separated Value (CSV) or Extensive Markup Language (XML) files. It is assumed that `DataSet` is a class describing one line of data of a CSV file and that XMLData is a class that can store an XML tree.

names and abstract methods are print italic in class diagrams. [41, pp. 435–438 852]

7.2 Programming Patterns

Programming patterns are arrangements that can be implemented to solve certain design problems. Using them has the benefit that they are well-known and therefore constraints as well as consequences can be read up in e. g. [11].

7.2.1 Singleton

The singleton pattern allows classes with only a single instance, to which it provides global access. This behaviour is often desired for classes that represent global settings. Only one global settings object is required, which should be easily accessible from everywhere in the application, see 7.4. [11, p. 127]

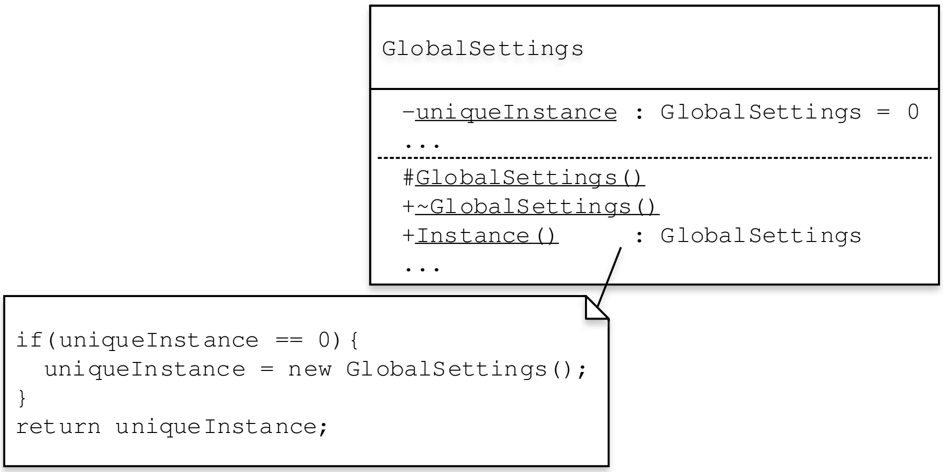
The singleton pattern declares the class's constructor protected that it can't be instantiated from outside the class. A static method `Instance()` is defined that constructs and stores the instance in a static attribute at the first call. It returns a reference to the instance. [11, pp. 128–130]

Note that the class's constructor is declared as protected and not as private, which means that it is possible to derive from it. This enables instantiating the attribute referring to the sole instance with an object of subclass type. [11, p. 130]

7.2.2 Publish-Subscribe

The publish-subscribe pattern is used to notify all subscribed objects that the publishing one changed it's state. To achieve this behaviour usually classes `Publisher` and `Subscriber` are defined. `Subscriber` is an abstract class that only has one abstract method `update(in p:Publisher)`. All subscribers derive from it and implement the update method according to the desired behaviour. `Publisher` maintains a list of references to subscribers, that can be attached or detached arbitrarily at any time. A method `notify()` iterates through the list and calls each subscriber's update method, without having to know of which subclass type of `Subscriber` the objects exactly are. It passes a reference to the publisher that the subscriber can distinct which one is notifying. Publishing classes are derived from `Publisher`. An example is illustrated in Figure 7.5. [11, pp. 293–303]

Commonly subscribers maintain a reference to the publisher. Thus, subscribers are able to put requests to the publisher, that can in turn trigger the publisher to notify *all* of it's subscribers. This leads to a *behaviour* like individual subscribers would know about each other, which is not necessary with the publish-subscribe pattern. [11, pp. 293–294]


GlobalSettings
-uniqueInstance : GlobalSettings = 0
...
#GlobalSettings()
+~GlobalSettings()
+Instance() : GlobalSettings
...

```
if(uniqueInstance == 0){
  uniqueInstance = new GlobalSettings();
}
return uniqueInstance;
```

Fig. 7.4. The class `GlobalSettings` applies the singleton pattern: It's constructor is protected that it can't be instantiated from outside. The sole instance is constructed at the first call to the static method `Instance()` and is stored in the static attribute `uniqueInstance`. Calling `Instance()` returns the object.

7.2.3 Smart Pointer

Depending on the case of application of an array, it is possible that it's size is not known at compile time. For example in processing sensor data the number of measurements, and therefore the size of the buffer collecting the samples, depends on the certain setup. Therefore the compiler can't pre-reserve the required space on the stack for local buffers or in the data or bss sections for global ones. Of course it would be possible to use a buffer with a fixed size that is capable of storing the largest possible number of samples. This would not be very efficient by means of memory usage. To solve this problem dynamically allocated memory is required.

Dynamically allocated memory can be requested at runtime and is limited. It is located in the heap. In C++ dynamically allocated memory always has to be deallocated, else applications would in most cases consume more and more heap memory while running. This is not experienced until there is no more heap memory available. An error message would be triggered then, if implemented, otherwise a program crash is caused. This phenomenon is called memory leak. Since Java for example employs a garbage collection, dynamic allocated memory is deallocated automatically. This avoids memory leaks generally.

In C++ it is also possible to achieve a similar behaviour by using smart pointers. The C++ standard library basically provides some implementations of smart pointers, while especially in MCU environments they are not always available.

A simple smart pointer that is similar to the `shared_ptr` from the standard library can be

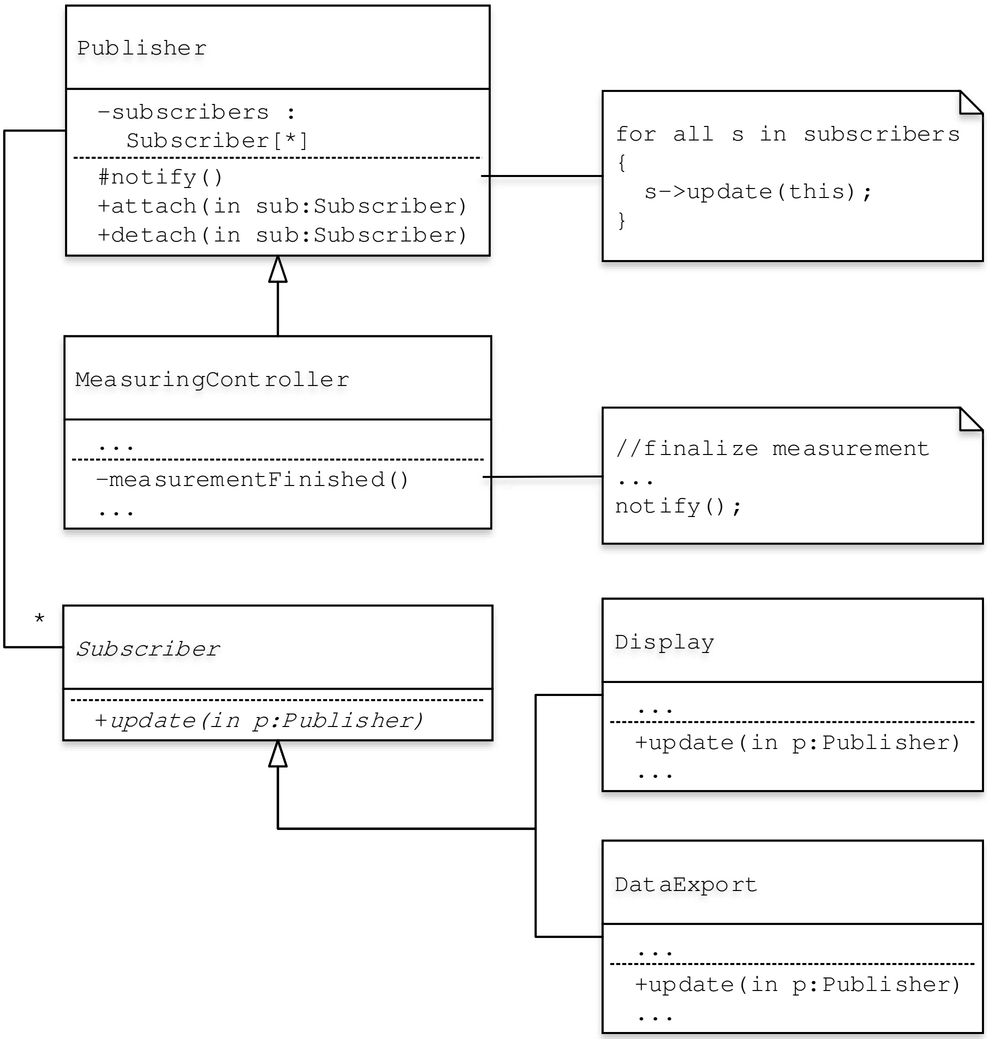


Fig. 7.5. The classes `Display` and `DataExport` shall be notified when `MeasuringController` has finished a new measurement. Therefore the measuring controller subclasses `Publisher`. `Display` and `DataExport` are derived from `Subscriber` and implement `update(...)` according to their desired behaviour. Objects of `Display` and `DataExport` are attached to the measuring controller object, which calls `notify()` when a new measurement is finished. This subsequently notifies the `Display` and `DataExport` objects.

implemented according to Figure 7.6. A class that is holding a reference to the maintained object is used. Additionally it has a reference to an object of a class `RefCount` consisting of a counter variable which can be incremented and decremented. `RefCount` counts the number of smart pointer objects that exist and correspond to the same maintained object reference. In the destructor of the smart pointer the counter value is checked. If the smart pointer object being deleted is the last reference to the maintained object, the object is also deallocated. Otherwise the counting object is decremented. It is required to define the smart pointer's copy mechanism in order that it increments the counting object after copying the references to the maintained object and the `RefCount` object. Note that there exist as many `RefCount` objects as *different* objects managed by smart pointers. Constructing more than one smart pointer object from the same dynamic allocated memory object is not allowed while copying the smart pointer object is applicable. Compare to [25].

Smart pointers enable assigning a scope to dynamic allocated memory and therefore allow treating it like a regular variable. The overhead that is caused by the usage of smart pointers of course is much higher than maintaining the memory manually. An implementation of a smart pointer class can be found in Chapter B.1.

7.2.4 Doubly Linked List

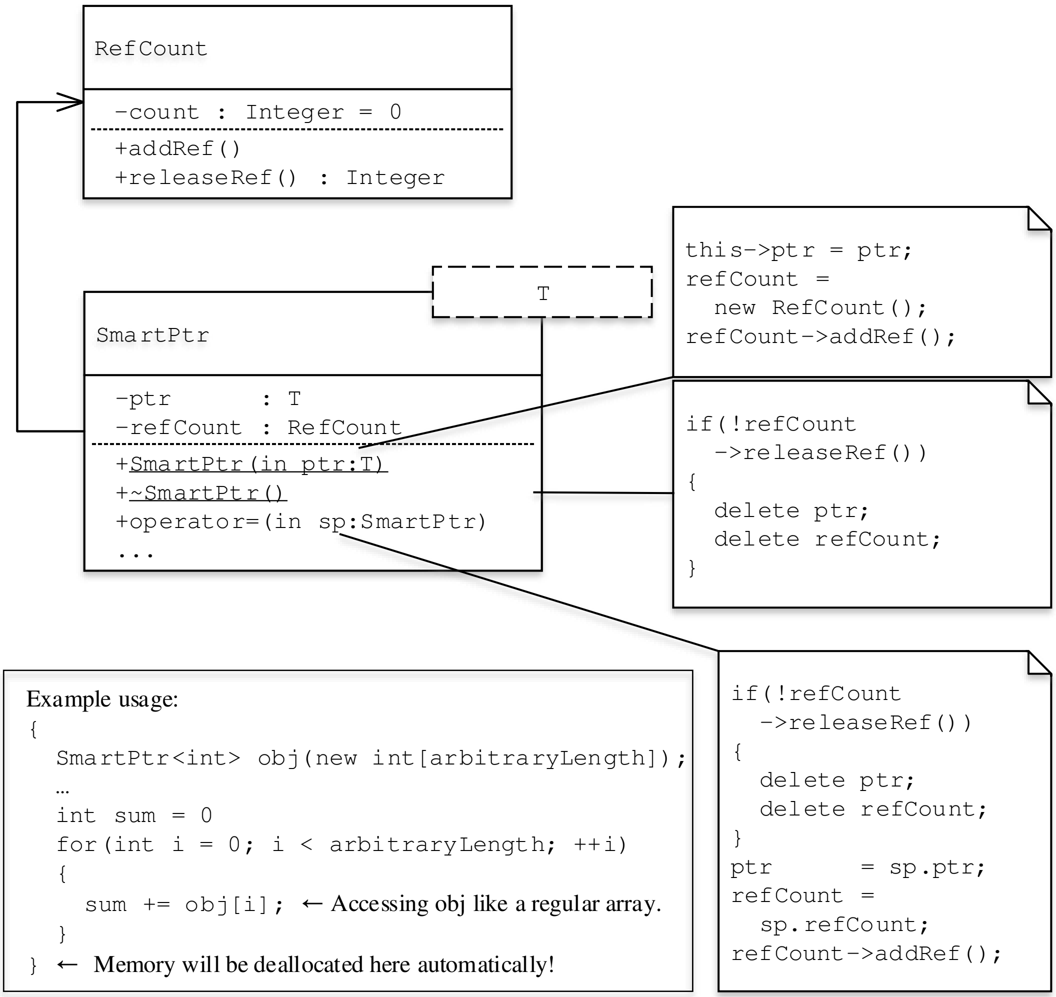
In Section 7.2.3 dynamically allocated memory was introduced as a way to create arbitrary sized arrays. Their size indeed can be chosen arbitrarily at the time of construction, but it can not be changed afterwards. A concept that allows appending, sorting and removing of an arbitrary number of elements is called doubly linked list[1].

In doubly linked lists each element, next to it's actual data, maintains references to the next and the previous list elements which are of the same type, see Figure 7.7. Usually implementations provide a class that has a reference to the first list element and feature functionality to access the elements, as well as to append elements to the list or remove elements from it.

During the firmware part of this thesis a very special implementation of doubly linked lists was developed, that avoids dynamically allocated memory usage and does not require runtime type information. It has the drawback that each element can only belong to one list at a time. Figure 7.8 shows how it is implemented.

Each class that should enable it's objects to be in such a list has to be derived from `DequeElement`. This effectively adds attributes for maintaining the references to the next and the previous list element as well as methods to get them. A list itself is represented by a template class `Deque` whereby `T` denotes the type of the subclassed list element. Since the class `Deque` is a *friend* of `DequeElement` it is allowed to access private attributes and members of `DequeElement`. This allows the list object to maintain the list element object's references within them.

[1] The C++ standard library of course offers containers like `std::vector` or `std::deque`, but they are neither suitable nor available for most microcontroller units.



```
this->ptr = ptr;
refCount =
  new RefCount();
refCount->addRef();
```

```
if(!refCount
  ->releaseRef())
{
  delete ptr;
  delete refCount;
}
```

```
if(!refCount
  ->releaseRef())
{
  delete ptr;
  delete refCount;
}
ptr      = sp.ptr;
refCount =
  sp.refCount;
refCount->addRef();
```

Example usage:

```
{
  SmartPtr<int> obj(new int[arbitraryLength]);
  ...
  int sum = 0
  for(int i = 0; i < arbitraryLength; ++i)
  {
    sum += obj[i]; ← Accessing obj like a regular array.
  }
} ← Memory will be deallocated here automatically!
```

Fig. 7.6. A smart pointer class according to Chapter 7.2.3. The smart pointer class `SmartPtr` is implemented as a template class that it can be used for arbitrary data types. A `RefCount` object stores how many smart pointer objects referring to the same object exist. The methods `addRef()` and `releaseRef()` increment respectively decrement the counting variable `count`.

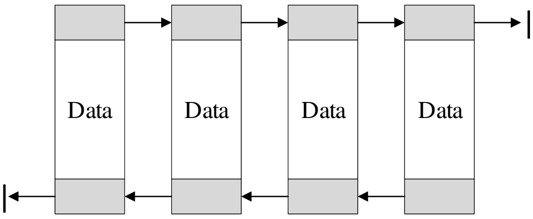


Fig. 7.7. In a doubly linked list each element has references to the previous respectively the next list element. Compare to [41, p. 442].

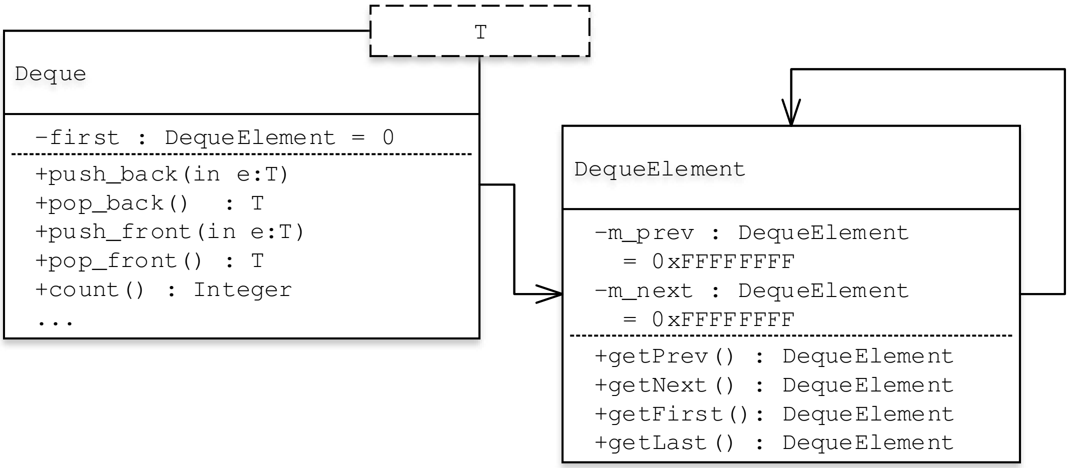


Fig. 7.8. Implementation of a doubly linked list suitable for microcontroller units.

By only allowing to append elements of the particular type `T`, and not of `DequeElement` (which would enable to append *all* classes that subclassed `DequeElement`), it is obvious that all involved references, which are stored as `DequeElement` references, are also of the specific type `T`. This prevents requiring runtime type information, which is usually used if a particular object type should be identified from a reference of base class type.

An implementation of a doubly linked list can be found in Chapter B.2.

7.3 Concurrency

Concurrency in computer systems refers to performing multiple independent activities in parallel. While a single core system can only run one task at a time it can switch between two or more tasks, creating the illusion they happen concurrently. Processors with multiple cores allow genuine concurrency. The difference is illustrated in Figure 7.9. Notice that this is a simplified representation. Task switches in single core machines involve saving the CPU state and reloading the state for the task that is switched to. These delays can not be neglected in overall performance. Although two or more CPU cores in a typical personal computer enable genuine concurrency, task switching is required to handle more tasks than the hardware can run in parallel. Keep in mind that a personal computer is typically running some background tasks as well as user tasks, where each of them possibly employs multithreading in addition. The so-called scheduler that implements task switching is a core part of the operating system. A notably simple implementation of a scheduler is discussed in [27]. Concurrency is used in order to separate concerns as well as to increase performance. [40, pp. 5–6 10]

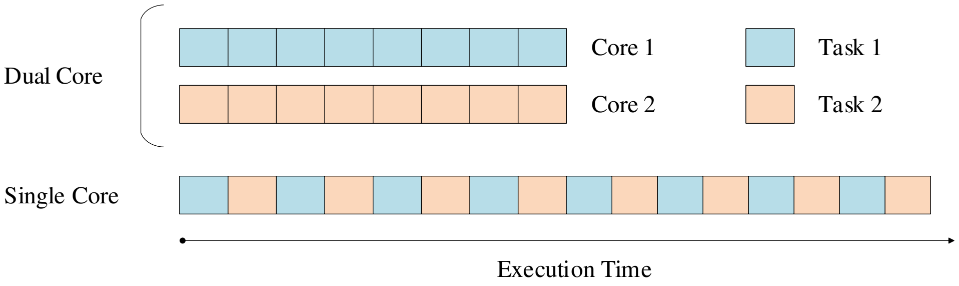


Fig. 7.9. Concurrency on single vs. dual core computers: While the dual core machine supports genuine concurrency, the single core utilizes task switching. Compare to [40, p. 6].

An example for genuine concurrency in order to separate concerns is given in Chapter 2, where a separate processor is used to handle radio commands.

7.3.1 Multitasking vs. Multithreading

The two common approaches to utilize concurrency are multitasking and multithreading. Multitasking is based on completely separate processes that communicate via operating system's resources. This approach is easier to handle for the programmer, since the tasks do not share resources and communication between them is based on higher-level communication mechanisms. The simplification in intercommunication is at the cost of resources. [40, pp. 7–8]

Multithreading is based on processes that operate in the same shared memory space, so called threads. This means that the same global variables can be accessed from each thread. While the intercommunication between threads is very fast, the programmer has to ensure that structures of data seen by threads are consistent at any time. Without protection an inconsistency could occur when one thread is reading a memory location that is currently being written by another. Common synchronizing mechanisms that can avoid such conflicts are called mutex and semaphores. [40, pp. 9–10]

7.3.2 Synchronization

To avoid e. g. corruption of data on concurrent access from multiple threads, synchronization is required. A commonly used synchronization primitive is called mutex, which is a composition of **mut**ual **ex**clusion. A mutex is *locked* at the beginning of a critical section and is *unlocked* at the end. Other threads can not enter the critical section at the same time since the call to the mutex's locking method blocks until it is unlocked again by the thread that locked it initially. In the meantime the locking thread owns the resource.

A semaphore is a more general approach to control access to limited resources. Semaphores use two atomic operations, *post* and *pend*, which both can be called from arbitrary threads at any time. *Post* increments the value of a counter, while *pend* waits for the counter value to exceed 0, then it decrements it and returns. If a counter with a maximum value of 1 is used, it is spoken of a binary semaphore. The intention of semaphores is to notify another thread that a resource is available again. A binary semaphore can also be used like a mutex.

Both mechanisms have to be supported by the processor.

7.4 Compilation & Linking

In this chapter many programming concepts were already presented. Now the gap between writing a program in the high level language C++ and flashing an executable to an MCU should be closed. The whole procedure is depicted in Figure 7.10.

Programming itself takes place in an Integrated Development Environment (IDE), which is basically a text editor. Header (`.h`) and source files (`.cpp`) contribute to the overall software solution. The headers contain declarations that *declare* which symbols are going be *defined* in a source file. Most often the symbols defined in a source file are declared in a header file with the same name. Both source files and header files include header files, while mutual inclusion is not allowed.

The first step in translating source code to an executable is preprocessing. During preprocessing macros are expanded and precompiler-switches are evaluated.

Subsequently, source files are going to be compiled. Compilation is the process of translating source code into machine code. This is conducted on a source file per source file basis. Processing the source files consecutively is possible because the compiler does not need to know how other source files define particular symbols. It is sufficient to know their declaration, which is achieved through inclusion of the header files corresponding to relevant other source files. For each source file an object file containing machine code is created during compilation.

The last step is linking, which puts object files as well as precompiled external libraries, if used, together to the executable file. Note that in this step machine code parts of actual implementations made in any of the source files are linked to each other. Thus, changing a source file requires only recompilation of the particular file and relinking. The same principles apply for precompiled external libraries. Relevant header files declaring library symbols have to be included. Precompiled library files containing the implementations have to be included in the linking process. [41, pp. 31–32]

When compiling source code for another platform, e. g. firmware for a MCU compiled on a PC, it is spoken about cross-compilation.

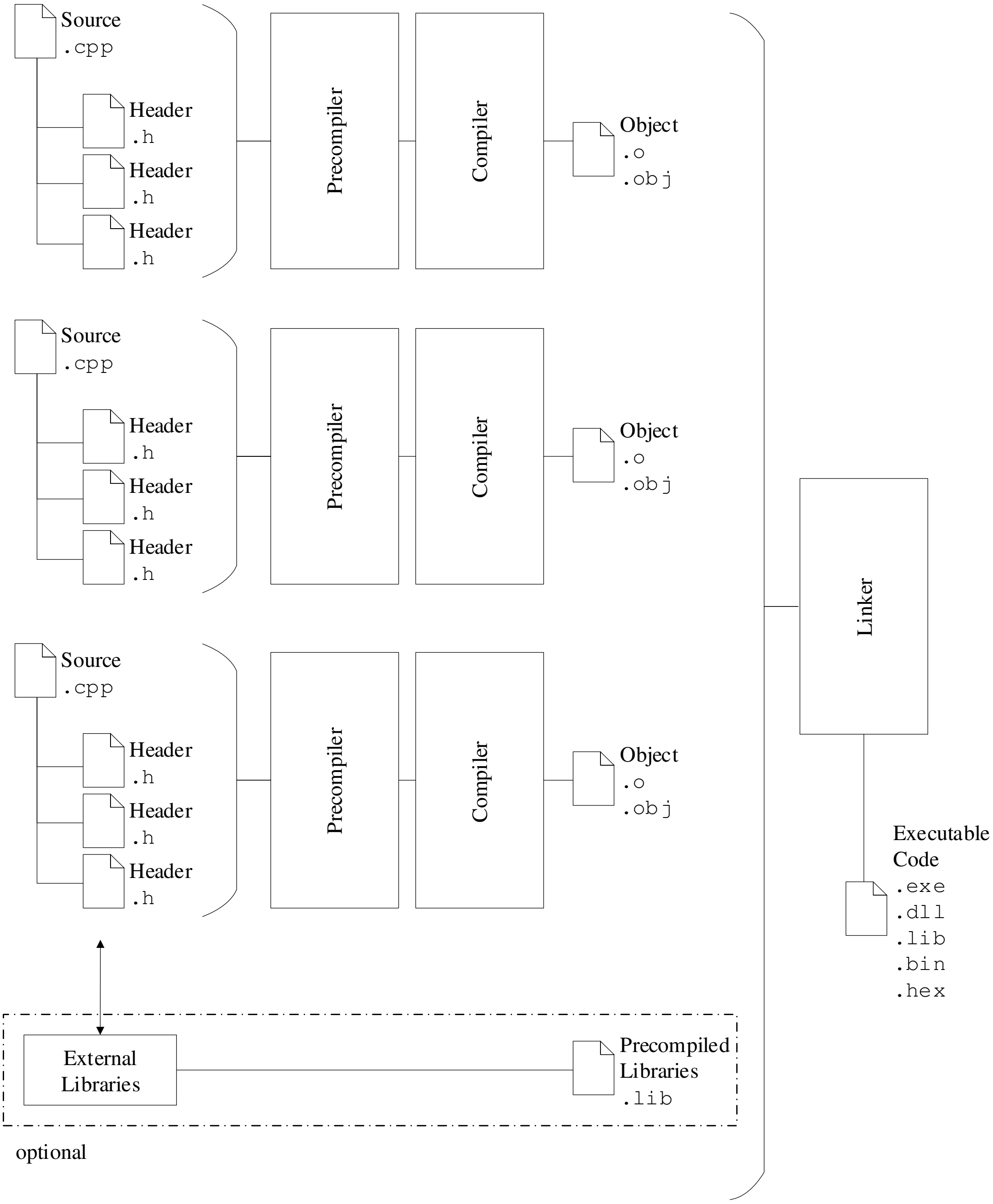


Fig. 7.10. Procedure of translating source code to an executable file. This involves precompilation, compilation and linking.

Part II

Research & Development

8 Hardware

In this chapter the hardware components that were used to develop a basic Wireless M-Bus condition monitoring system are explained. The system is split into a smart sensor and an edge gateway part. Additionally a proprietary current measurement circuit that allows wide current ranges is introduced.

8.1 Smart Sensor Development Setup

To actually start developing a smart sensor at first a MCU has to be selected, which is the heart of the system. As already mentioned in Chapter 2, it was decided to use the Texas Instruments CC1352P dual-band microcontroller in this thesis. The system's sensing element has been selected to be a Kionix KX222-1054 acceleration sensor. Development boards of the MCU as well as evaluation boards of the sensor are used, in which the necessary basic circuitry is already built up together with the particular chip. The main features of those components are as follows.

- Texas Instruments LaunchPad^{TM} kit with CC1352P MCU
    1. Basic Texas Instruments CC1352P circuit
    2. PCB antenna for 868 MHz and 2.4 GHz
    3. TI XDS110 JTAG debug probe embedded
    4. Macronix MX25R8035F 1 MB external flash connected via SPI
    5. Two programmable LEDs
    6. Two programmable buttons
    7. SPI, I^{2}C and UART interfaces
- Kionix KX222-1054 tri-axis accelerometer evaluation board
    1. Basic Kionix KX222-1054 circuit
    2. Selectable range ±8 g, ±16 g and ±32 g
    3. Selectable sample rate up to 25.6 kHz
    4. SPI and I^{2}C interface

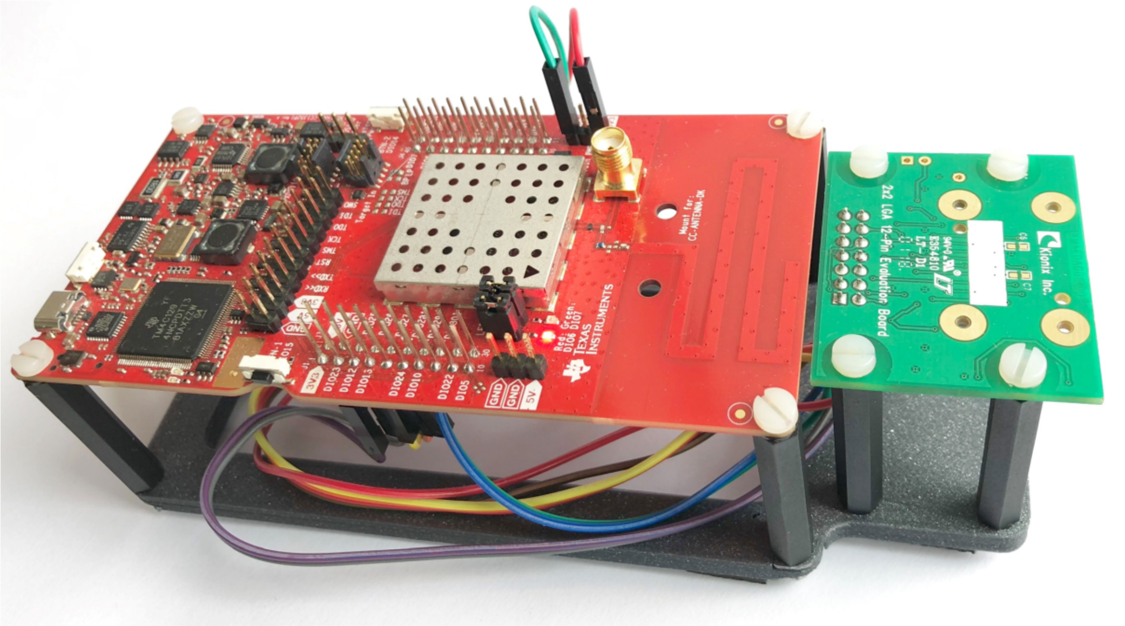


Fig. 8.1. The hardware setup consists of a TI LaunchPad$^{\mathrm{TM}}$ development board and a Kionix KX222-1054 evaluation board. They are both mounted on a 3D printed frame.

The MCU supports sub-GHz as well as 2.4 GHz networking, which can be used together e. g. to benefit from sub-GHz technology in an IIoT application while using a BLE interface for over-the-air upgrade. The device has an outstanding receiver sensitivity for sub-GHz modulation methods.

The ARM Cortex-M4 CPU with floating point unit provides enough computational power to actually perform computations, such as downsampling directly on the smart sensor, thus, reducing the amount of data that has to be transmitted to the edge device.

The sensor hardware development setup is made up of the MCU development board and the sensor evaluation board which are both mounted to a 3D printed frame that keeps the PCBs in position, see Fig. 8.1. The architecture of the MCU development board as well as the connection to the sensor via SPI is illustrated in Fig. 8.2. SPI is used due to high data rate demands. Using I^{2}C would not be beneficial.

The setup is usually powered by the USB connection to the developer's PC, but can also be powered from a separate power supply or a battery. Especially for measuring current consumption of the hardware it is necessary to disconnect the USB connection and remove all jumpers that connect the JTAG programmer to the MCU.

The sensor evaluation board basically features a LED that indicates if it is powered. Since the overall current consumption during deep sleep periods of the microcontroller is much less than the LED current, the LED had to be disabled. This was achieved by desoldering the series resistor.

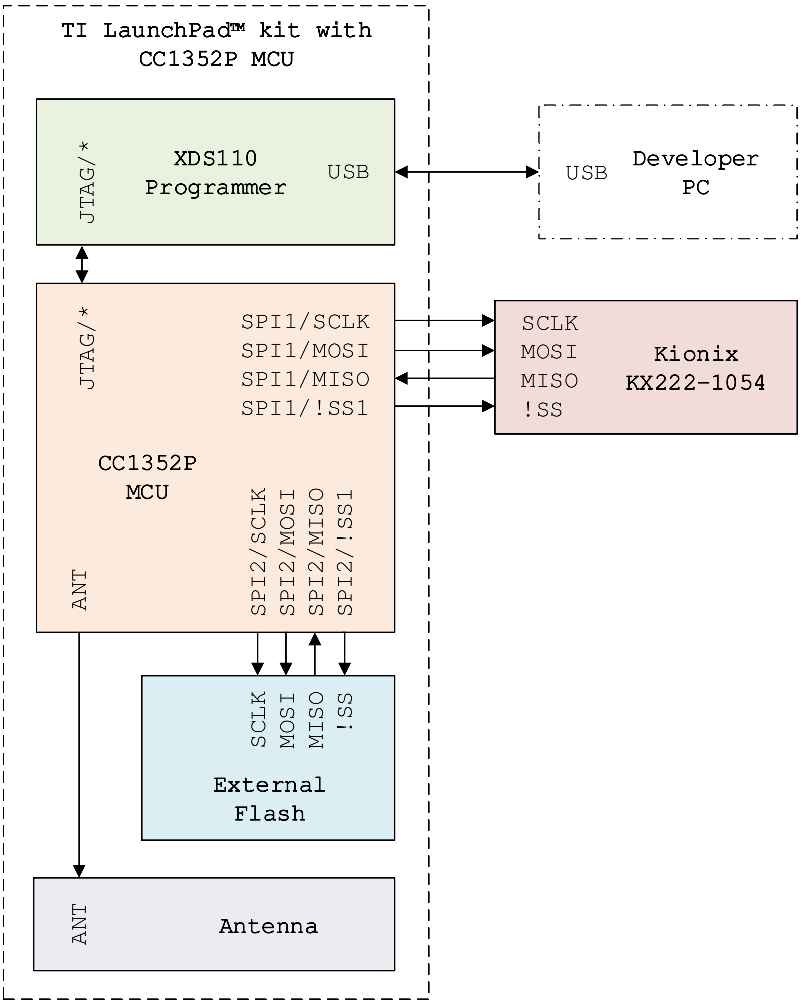


Fig. 8.2. Block diagram of the sensor development setup that was used in this thesis.

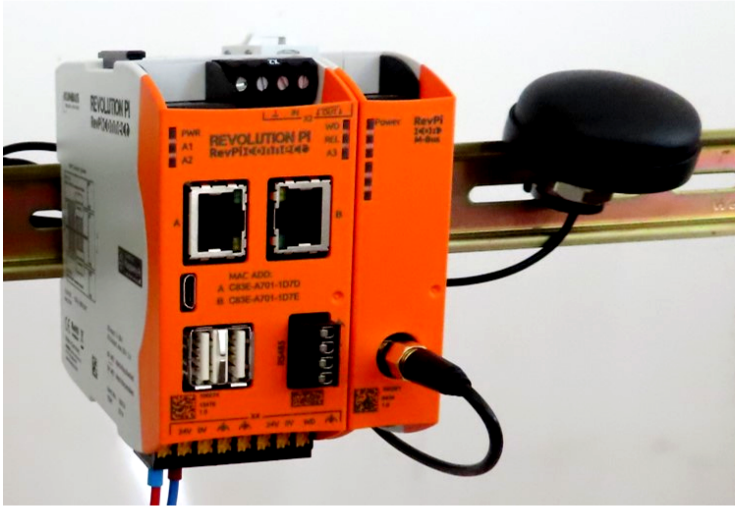


Fig. 8.3. The edge gateway setup consists of a RevolutionPi Connect paired with Wireless M-Bus module and an antenna.

8.2 Edge Device Setup

The edge device is based on an IEC 61131 certified Industrial PC (IPC), the Kunbus RevPi Connect. It is utilized with a quad-core 1.2 GHz CPU, 1 GB RAM and 4 GB storage. Furthermore it has two Ethernet interfaces that can be used to connect e. g. to an automation network and the corporate network simultaneously [26]. The device is running the operating system Raspbian with a real time patch.

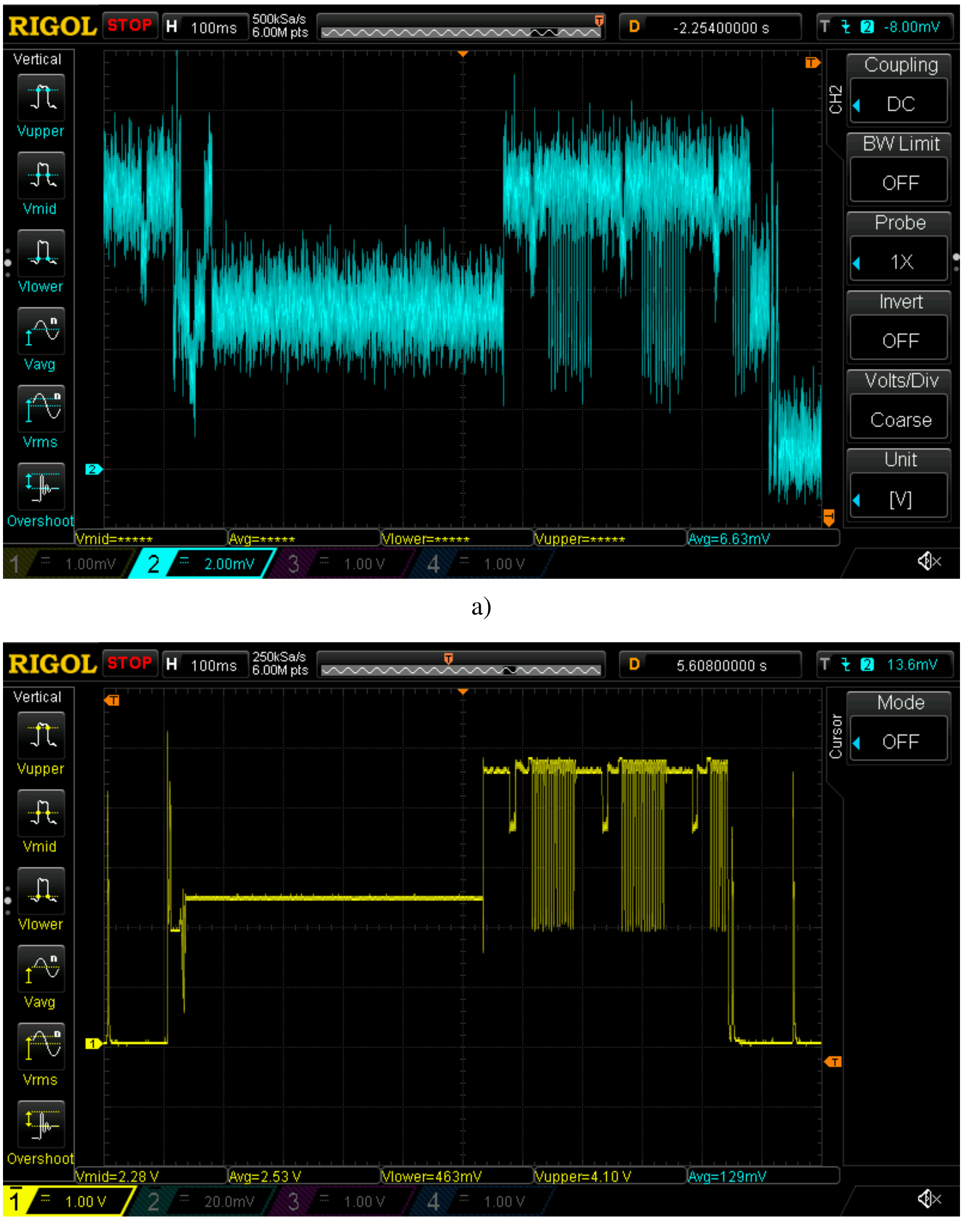
The RevPi Connect was chosen because a Wireless M-Bus extension module for it is available, the RevPi Con M-Bus module. The module is connected via a proprietary *connect bridge* interface and is communicated with via a serial port. It requires an external antenna to be attached.

The system has to be supplied by an external 24 V power supply and is mounted on a DIN rail. The whole setup is depicted in Fig. 8.3.

8.3 Current Measurement Circuit

The first current measurement approach used a measuring device that has three ranges 1 nA to 1000 nA, 1 µA to 1000 µA and 1 mA to 1000 mA. Unfortunately the current consumption of the sensor development setup was close to the lower limit of a particular range in many cases, which caused very noisy measurements, see Fig. 8.4a.

Measuring current consumption of the sensor development setup is a difficult concern since the current varies from about 1 µA to approximately 65 mA, depending on the task that is currently



a)

H 100ms
500kSa/s 6.00M pts
D -2.25400000 s
T 2 -8.00mV
Coupling DC
BW Limit OFF
Probe 1X
Invert OFF
Volts/Div Coarse
Unit [V]
Vmid=***** Avg=***** Vlower=***** Vupper=***** Avg=6.63mV
1 1.00mV 2 2.00mV 3 1.00 V 4 1.00 V

b)

H 100ms
250kSa/s 6.00M pts
D 5.60800000 s
T 2 13.6mV
Mode OFF
Vmid=2.28 V Avg=2.53 V Vlower=463mV Vupper=4.10 V Avg=129mV
1 1.00 V 2 20.0mV 3 1.00 V 4 1.00 V

Fig. 8.4. The same measurement was conducted using a measuring device with inappropriate ranges in Fig. 8.4a and with the developed circuit in Fig. 8.4b.

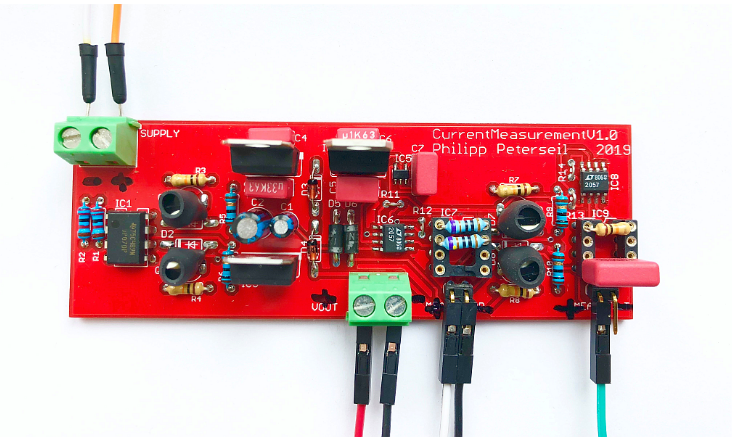


Fig. 8.5. After testing the breadboard prototype, a PCB version of the current measurement circuit was built.

performed. In order to avoid buying an appropriate expensive current measuring device a relative simple circuit of operational amplifiers was invented that was then used for current sensing. Fig. 8.4 compares the circuit to the measuring device used in a first approach.

The current measurement circuit prototype was built on a solderless breadboard. After some improvements, a robust PCB version was built that simplified mounting of components in Surface Mounted Device (SMD) packages. It also reduces noise due to shorter wiring.

8.3.1 Schematic

The schematic of the current measurement circuit is explained in figures 8.6 and 8.7a - 8.7c.

The measurement itself is based on a transimpedance amplifier [13, p. 184] that utilizes a push-pull amplifier [13, pp. 91–94] as a power stage. The feedback resistor that determines the transimpedance is implemented as an IC socket in which arbitrary resistors can be plugged. This enables to vary sensitivity.

The transimpedance amplifier is followed by another IC socket that enables e. g. to plug in a low-pass filter. The low-pass filtered signal is put out via a voltage follower. This stage can be used to determine mean current consumption.

Note that it measures the current of the external circuit together with the quiescent current of the voltage regulator `IC5`. To minimize the effect of the regulator, a special type with a very low quiescent current $< 500\,\text{nA}$ was used.

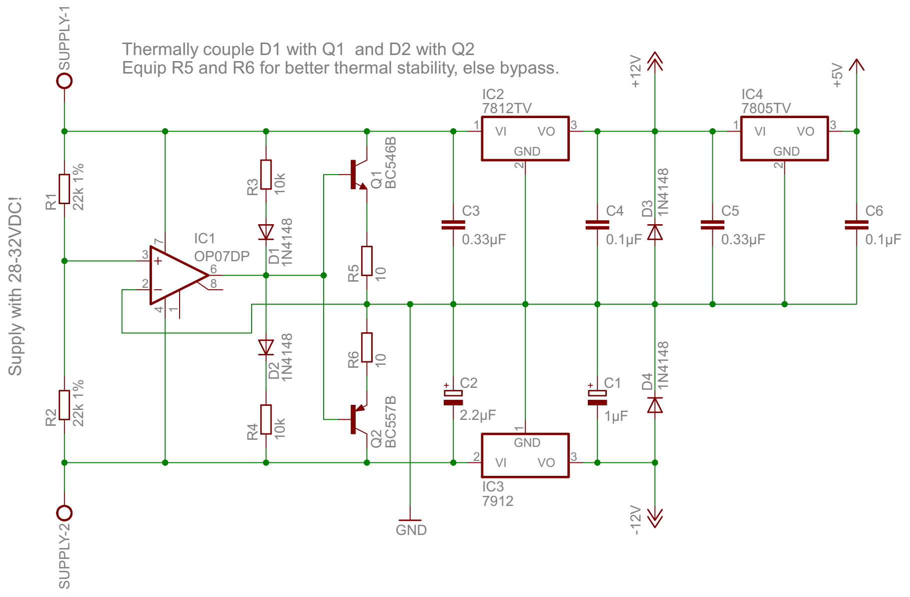
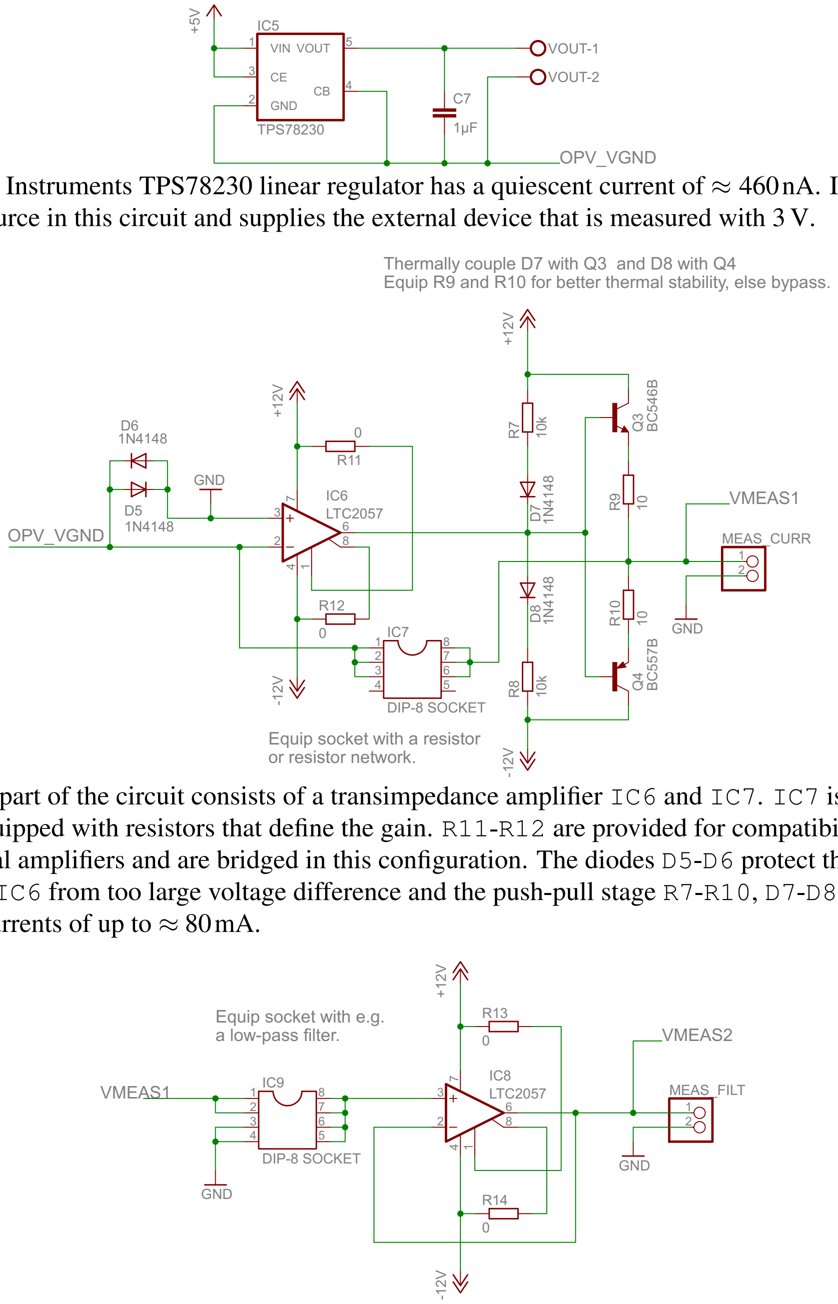


Fig. 8.6. The current measuring circuit requires a sophisticated power supply, which is powered from 28 V to 32 V. R1 and R2 build a voltage divider, that is a reference for the center voltage. The operational amplifier IC1, together with the push-pull amplifier R3-R6, D1-D2 and Q1-Q2 facilitates current source and sink capabilities at center voltage level. The linear voltage regulators IC2, IC3 and IC4 are responsible for supplying 12 V, −12 V and 5 V respectively. C1−C6 stabilize the regulators.



a) The Texas Instruments TPS78230 linear regulator has a quiescent current of ≈ 460 nA. It acts as a current source in this circuit and supplies the external device that is measured with 3 V.

b) The main part of the circuit consists of a transimpedance amplifier `IC6` and `IC7`. `IC7` is a socket that can be equipped with resistors that define the gain. `R11`-`R12` are provided for compatibility to other operational amplifiers and are bridged in this configuration. The diodes `D5`-`D6` protect the operational amplifier `IC6` from too large voltage difference and the push-pull stage `R7`-`R10`, `D7`-`D8` and `Q3`-`Q4` enables currents of up to ≈ 80 mA.

c) The socket `IC9` is intended to be equipped with an arbitrary low-pass filter that is stabilized by the operational amplifier voltage-follower `IC8`. `R13`-`R14` are provided for compatibility to other operational amplifiers and are bridged in this configuration.

Fig. 8.7. Explanation of the current measuring circuit's schematics.

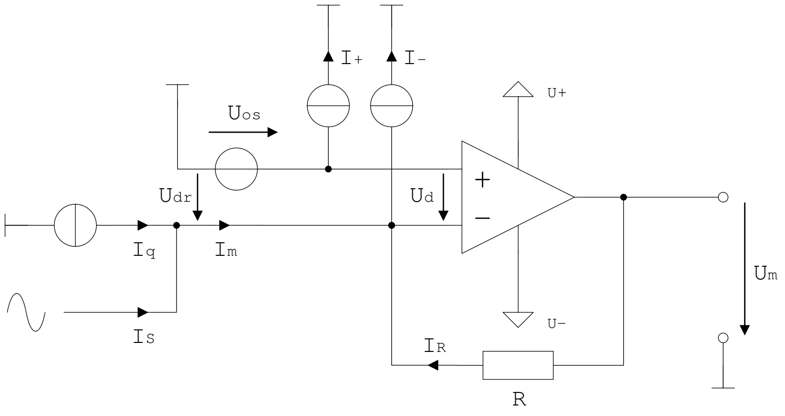


Fig. 8.8. The transimpedance amplifier circuit that is using a real operational amplifier.

8.3.2 Measuring Error Estimation

In this section the transfer function of the transimpedance amplifier, which is the main module of the circuit is derived including deviations from the ideal operational amplifier. Afterwards, the transfer function together with specified characteristics of a particular operational amplifier is used to estimate the accuracy of measurements.

The schematic of the transimpedance amplifier with a real operational amplifier is given in Fig. 8.8. It includes influences of offset voltage, bias and offset currents as well as the quiescent current of the voltage regulator `IC5` in Fig. 8.7a, which is assumed to be constant.

The circuit can be split up in a part that deals with constant current and voltage sources and a part dealing with the dynamic current that is to be measured according to Helmholtz's law, see Fig. 8.9.

The measured current

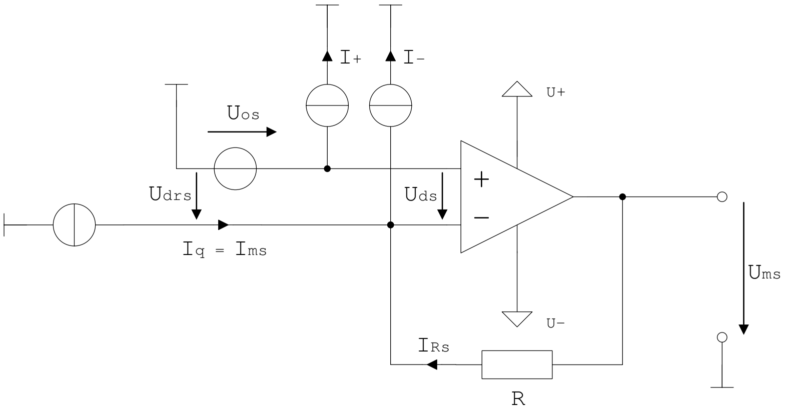
$$\underline{I}_m = I_q + \underline{I}_s, \tag{8.1}$$

with respect to the quiescent current of the voltage regulator `IC5` in Fig. 8.7a denoted as I_q and the current of the external smart sensor to be measured $\underline{I}_s$.

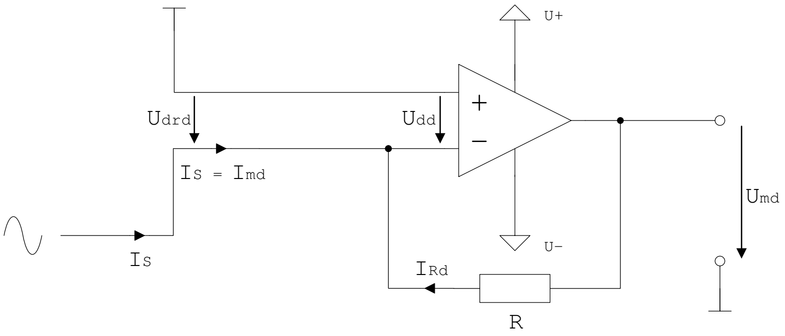
Kirchhoff's voltage law yields

$$0 = \underline{U}_d + U_{os} - \underline{U}_{dr}, \tag{8.2}$$

with respect to the difference voltage of the ideal operational amplifier $\underline{U}_d$, the offset voltage of the operational amplifier U_{os} and the difference voltage of the real operational amplifier $\underline{U}_{dr}$.



a) Static part of the circuit



b) Dynamic part of the circuit

Fig. 8.9. Superposition of static and dynamic circuit parts depicted in figures 8.9a and 8.9b yields the overall circuit depicted in Fig. 8.8.

The complex gain $\underline{G}$ of an operational amplifier with respect to the frequency f can be formulated as

$$\underline{G} = \frac{G_0}{1 + \mathrm{i}(f/f_c)}, \tag{8.3}$$

with respect to the open-loop gain G_0 and the critical frequency f_c. The critical frequency of an operational amplifier $f_c = f_0/G_0$, with respect to the gain-bandwidth product f_0 and the open-loop gain G_0. [34, p. 539]

The definition of the gain yields

$$\underline{U}_m = \underline{U}_d \cdot \underline{G} = (\underline{U}_{dr} - U_{os}) \cdot \underline{G}, \tag{8.4}$$

using the definition of the offset voltage [34, pp. 530–532].

The bias current of a real operational amplifier is defined as

$$I_b = \frac{1}{2} \cdot (I_+ + I_-) \tag{8.5}$$

and the offset current of a real operational amplifier is defined as

$$I_{os} = I_+ - I_-, \tag{8.6}$$

with respect to the current flowing into the non-inverting input I_+, respectively into the inverting input I_- of the amplifier. [34, pp. 532–534][23]

Kirchhoff's current law at the inverting input yields

$$\underline{I}_m + \underline{I}_R - I_- = 0. \tag{8.7}$$

Using (8.5) and (8.6) to eliminate I_+ yields

$$I_- = I_b - \frac{I_{os}}{2}. \tag{8.8}$$

Using (8.8) and (8.7) to eliminate I_- yields

$$0 = \underline{I}_m + \underline{I}_R - I_b + \frac{I_{os}}{2}. \tag{8.9}$$

Kirchhoff's voltage law can be used to derive

$$0 = \underline{U}_m + \underline{U}_{dr} - \underline{I}_R \cdot R, \tag{8.10}$$

with respect to the feedback resistor R and the output voltage U_m.

Using (8.4) and (8.10) to eliminate $\underline{U}_{dr}$ yields

$$\frac{\underline{U}_m}{\underline{G}} + U_{os} = (I_b - \frac{I_{os}}{2} - \underline{I}_m) \cdot R - \underline{U}_m, \tag{8.11}$$

which can be rearranged to achieve

$$\underline{U}_m = \frac{1}{1+1/\underline{G}}\left(\left(I_b - \frac{I_{os}}{2} - \underline{I}_m\right)\cdot R + U_{os}\right). \tag{8.12}$$

The operational amplifiers gain $\underline{G}$ depends on the frequency. Therefore, (8.12) shall be separately evaluated for the static part, see Fig. 8.9a, and for the dynamic part, see Fig. 8.9b. Note that static parts of voltages and currents are indexed with an additional “s”, while dynamic parts are indexed with an additional “d”.

For the static part

$$\underline{I}_{ms} = I_q \tag{8.13}$$

and

$$\underline{G}(f=0) = G_0 \tag{8.14}$$

as well as

$$G_0 \gg 1 \tag{8.15}$$

is assumed yielding the static part of the output voltage

$$\begin{aligned} U_{ms} &= \frac{1}{1+1/G_0}\left(\left(I_b - \frac{I_{os}}{2} - I_q\right)\cdot R + U_{os}\right) \\ &\approx \left(I_b - \frac{I_{os}}{2} - I_q\right)\cdot R + U_{os}. \end{aligned} \tag{8.16}$$

For the dynamic part, all static voltage sources are replaced by bridges and all constant current sources replaced by breaks, yielding

$$\underline{U}_{md} = -\frac{1}{1+1/\underline{G}}\cdot R\cdot \underline{I}_s. \tag{8.17}$$

The magnitude of $\underline{U}_{md}$ can be written as

$$|\underline{U}_{md}| = \frac{G_0\cdot f_0}{\sqrt{(f\cdot G_0)^2 + (f_0\cdot(G_0+1))^2)}}\cdot R\cdot I_s. \tag{8.18}$$

The composition of (8.17) and (8.16) yields the equation describing the overall system

$$\underline{U}_m = \left(I_b - \frac{I_{os}}{2} - I_q\right)\cdot R + U_{os} - \frac{1}{1+1/\underline{G}}\cdot R\cdot \underline{I}_s. \tag{8.19}$$

It shall be corrected by the voltage, that is caused by the quiescent current of the voltage regulator

$$U_q = -I_q\cdot R_0, \tag{8.20}$$

Symbol	Min.	Nom.	Max.	Unit	Device
R_0		500		Ω	
R	499.5		500.5	Ω	0.1 %
I_q		500		nA	TPS78230 [35]
G_0	0.8	31.6		10^6	LTC2057 [18]
f_0		1.5		MHz	LTC2057 [18]
U_{os}			4	µV	LTC2057 [18]
I_b			300	pA	LTC2057 [18]
I_{os}			460	pA	LTC2057 [18]

Tab. 8.1. The tabulated values are characteristic for the components used in the current measuring circuit, see Chapter 8.3.1.

yielding the voltage corresponding to the measured current of the external sensor

$$\begin{aligned} \underline{U}_s &= \underline{U}_m - U_q \\ &= \underbrace{\left(I_b - \frac{I_{os}}{2} - I_q\right) \cdot R + U_{os} + I_q \cdot R_0}_{F_a} - \underbrace{\frac{1}{1+1/\underline{G}} \cdot R}_{\underline{A} \cdot R_0} \cdot \underline{I}_s, \end{aligned} \tag{8.21}$$

with respect to the nominal value R_0 of the feedback resistor R, the absolute measurement error F_a and the attenuation $\underline{A}$. An arbitrary signal can be described analogue with a term for each spectral component.

The attenuation $\underline{A}$ can be used to calculate the relative error

$$F_r = |\underline{A}| - 1. \tag{8.22}$$

The relative error F_r is composed of the relative error of the feedback resistor and the attenuation caused by demands on the operational amplifiers gain, which can't be met for higher frequencies. At the unity-gain frequency f_0 the signal is attenuated by -3 dB corresponding to ≈ -29.3 %.

Evaluating the absolute error according to (8.21), together with worst case values from Tab. 8.1, leads to a lower limit of the absolute error of -3.65 µV and to an upper limit of 4.38 µV. The worst-case relative error with respect to the frequency is depicted in Fig. 8.10.

8.3.3 Phase compensation

In Chapter 8.3.2 equations describing the transimpedance amplifier were derived, which take imperfections of the operational amplifier, such as offset voltage and limited unity-gain frequencies into account. Another impact on the dynamic behaviour of a transimpedance amplifier is caused by the capacitance C_i at the input, see Fig. 8.11. Note that the capacitance C_i represents all capacitive

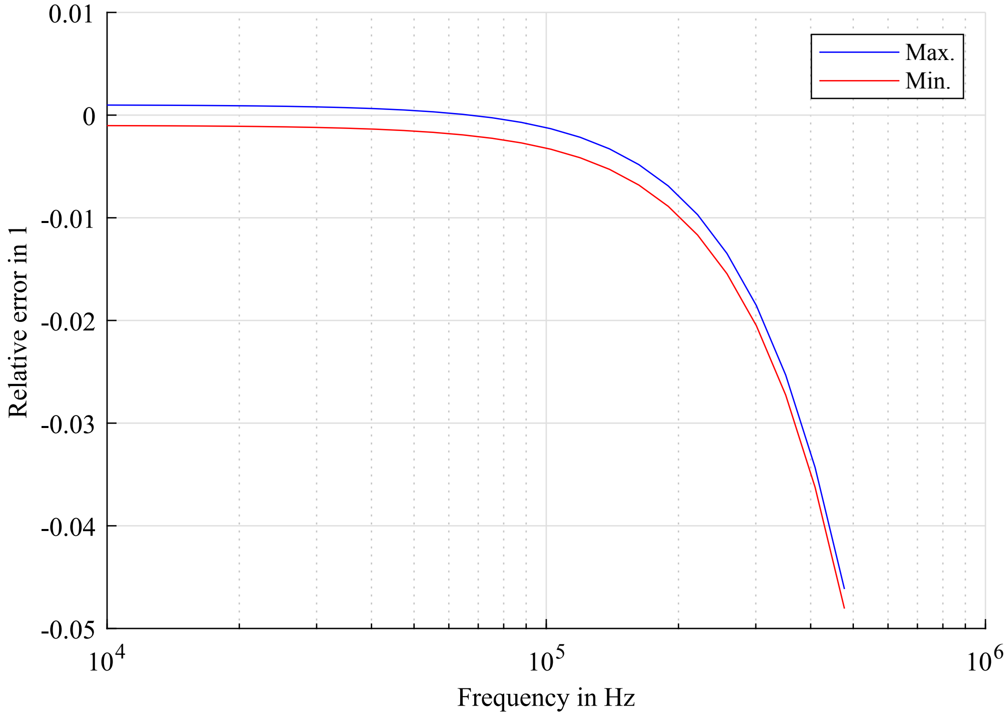


Fig. 8.10. Relative error of a transimpedance amplifier with characteristics specified in Tab. 8.1.

effects at the input, such as the operational amplifier's input capacitance or the capacitance of protection diodes.

Together with the feedback resistor R, the input capacitance C_i acts as a low-pass filter in the feedback. The additional phase-shift that is caused by the low-pass filter can lead to resonances for high frequencies. More precisely, the 90° phase-shift of the operational amplifier itself, the 90° of the low-pass filter and the 180° caused by the negative feedback can add up to 360°, yielding a positive feedback, see Fig. 8.12. [12, p. 41]

The resonance peak at least affects noise which can be enough for instable operation. The effect should be considered if the critical frequency of the feedback low-pass filter

$$f_{gi} = 1/(2\pi R C_i) \tag{8.23}$$

is lower than the unity-gain frequency f_0 of the operational amplifier.

To compensate the instability, an additional capacitor C_f can be placed in parallel to the feedback resistor R. Taking effects of the capacities into account, (8.18) slightly changes to

$$\underline{U}_{md} = -\frac{1}{1+\frac{1}{\underline{G}\cdot\underline{\beta}}} \cdot R \cdot \underline{I}_s, \tag{8.24}$$

with respect to

$$\underline{\beta} = \frac{1 + \mathrm{i}2\pi f \cdot RC_f}{1 + \mathrm{i}2\pi f \cdot R\left(C_i + C_f\right)}, \tag{8.25}$$

see [12, pp. 39–43].

That the transimpedance amplifier circuit works properly, the condition

$$|\underline{G}| \cdot \left|\underline{\beta}\right| \gg 1, \tag{8.26}$$

or rearranged

$$|\underline{G}| \gg \frac{1}{\left|\underline{\beta}\right|}, \tag{8.27}$$

has to be fulfilled. Therefore $1/\left|\underline{\beta}\right|$ is called gain demand.

The resonance peak occurs at the frequency

$$f_r = \sqrt{f_{gi} \cdot f_0}, \tag{8.28}$$

where the operational amplifier's gain $|\underline{G}|$ and the gain demand $1/\left|\underline{\beta}\right|$ are equal. [12, p. 41]

A simple design equation for the compensation capacitor C_f is given by

$$C_f = \sqrt{\frac{C_i}{2\pi \cdot R \cdot f_0}}, \tag{8.29}$$

for $C_i \gg C_f$. [12, p. 49]

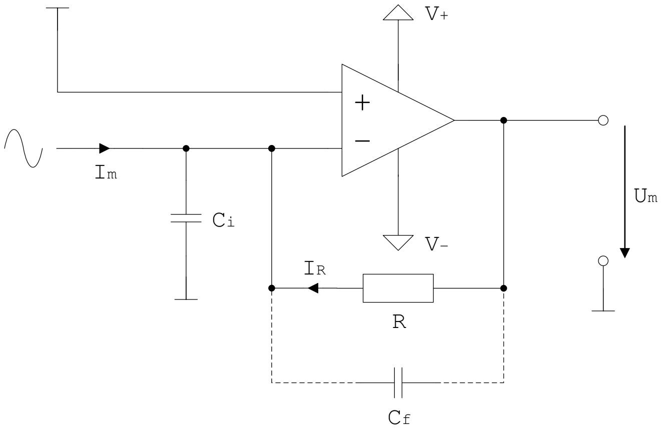


Fig. 8.11. Phase compensation can be achieved by placing a capacitor in parallel to the feedback resistor. Compare to [12, p. 42].

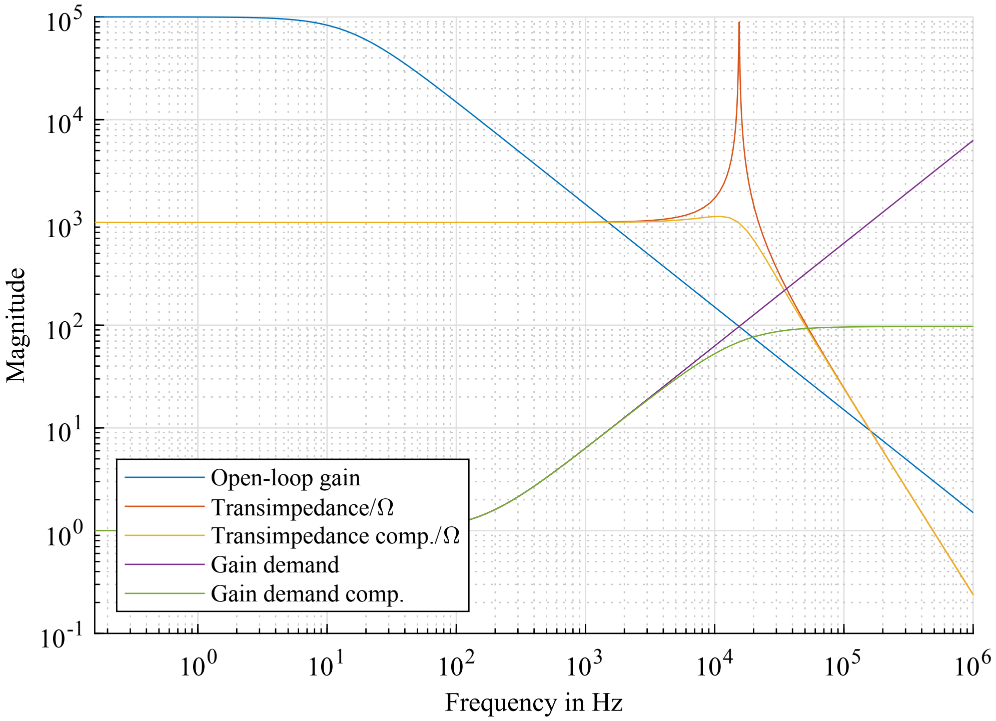


Fig. 8.12. A resonance peak can be caused by capacitance at the input of a transimpedance amplifier. Compare to [12, p. 40].

9 Firmware

This chapter documents the firmware that is running on the smart sensor. At first the main threads in which the application was split up are explained. Afterwards the classes implementing most of the features are discussed. The firmware is implemented in object oriented C++ using Texas Instrument's TI-RTOS operating system.

9.1 Program Flow

The firmware is split up in two threads to separate the concerns of networking and measuring, see Fig. 9.1.

The network thread at first sets up the network stack. This includes allocating a buffer memory, setting up the device's Wireless M-Bus address as well as the AES-128 encryption key. The thread then waits for the network semaphore to be available, which is cyclically posted by a timer interrupt (external). Then it checks if a new dataset is available. In the case that a new dataset is available in the external flash memory, a reference to it is stored. In the other case the latest dataset is chunked and transmitted part by part until a new dataset is available.

The measuring thread is initialized by allocating temporary buffers, setting up the time module to facilitate creation of timestamps and preparing the sensor. A particular sensor is derived from an abstract sensor class, while the `MeasScheduler` class allows to handle an arbitrary number of sensors by means of triggering their measurement method on a regular basis. Therefore the sensor objects have to be passed to the measurement scheduler `MeasScheduler`.

The scheduler basically iterates over the sensors to determine which is the next to be triggered. Then it sets up a timer generating an interrupt at the time, when the next sensor shall be triggered. The timer interrupt posts the semaphore that the measuring thread is waiting for after starting the timer. Then the measurement of the particular sensor is triggered, which can include data processing, depending on the sensor. The datasets are stored on the external flash. Finally, the next measurement is scheduled.

A sensor is basically implemented as a class derived from `AbstractSensor`. The concrete sensor class defines how a measurement is conducted. Each `AbstractSensor` has a `DataManager` object that is responsible for storage of measuring data. Each `DataManager` in turn has a list of `DSPAbstractComputation` objects, each defining a particular algorithm, e. g.

downsampling. After acquiring, the measuring data are processed and then written to non-volatile memory.

The threads intersect at the DataManager object that is responsible for reading and writing data to the external flash memory. In order to synchronize the threads, a mutex is used to provide exclusive access to the DataManager object. To check which sensor was the last triggered and if a new dataset is available, the thread safe DataExchange class is used.

9.2 Implementation

This chapter is documenting the implementation of the main features of the smart sensor firmware, such as the Wireless M-Bus stack, the measurement scheduler and the memory management.

It was decided to document object-oriented code by means of class diagrams together with a verbal explanation of the usage.

9.2.1 Wireless M-Bus Stack

The Wireless M-Bus stack is based on the MBusRFLayer class that implements encoding and the interfacing with the Texas Instruments CC1352P's radio core. The class can be considered as the physical layer together with the radio core and the RF circuitry. It passes the encoded datagram to the radio core, which is responsible for interfacing the RF circuitry and therefore actual transmission. At the moment packets can be transmitted but not received.

The class MBusRFLayer, see Fig. 9.2, contains a Deque, see Fig. 7.8, that maintains the consecutive layers, which have to be derived from MBusAbstractLayer. The MBusAbstractLayer base class provides functionality to network layers to communicate with each other by means of events, allowing to exchange objects. It defines virtual methods data(...) and count() which are used by MBusRFLayer to copy data from the layer's buffer to the transmission buffer. The method coversSubsequentLayers() returns true if the layer's buffer includes data from all subsequent layers, which then do not have to be considered anymore. This is the case for the transport layer, if it is configured to encrypt all subsequent layers with AES-128 CBC (see Chapter 4.2). A layer's buffer is defined as an array if it's size is known at compilation time, otherwise a smart pointer, see Chapter 7.2.3 is used.

The data link layer for example is implemented according to Fig. 9.3. Further layers, such as the transport layer as well as the user data layer are defined in a similar way.

To send a Wireless M-Bus datagram, at first objects of all layers, as well as an object of MBusRFLayer have to be defined. Then the layers have to be pushed on the network stack using MBusRFLayer's method pushLayer(...). Before sending any message, the static method Initialize() has to be called to prepare the radio core. To initiate transmission MBusRFLayer's method send(...), with the used coding scheme as an argument, has to be called.

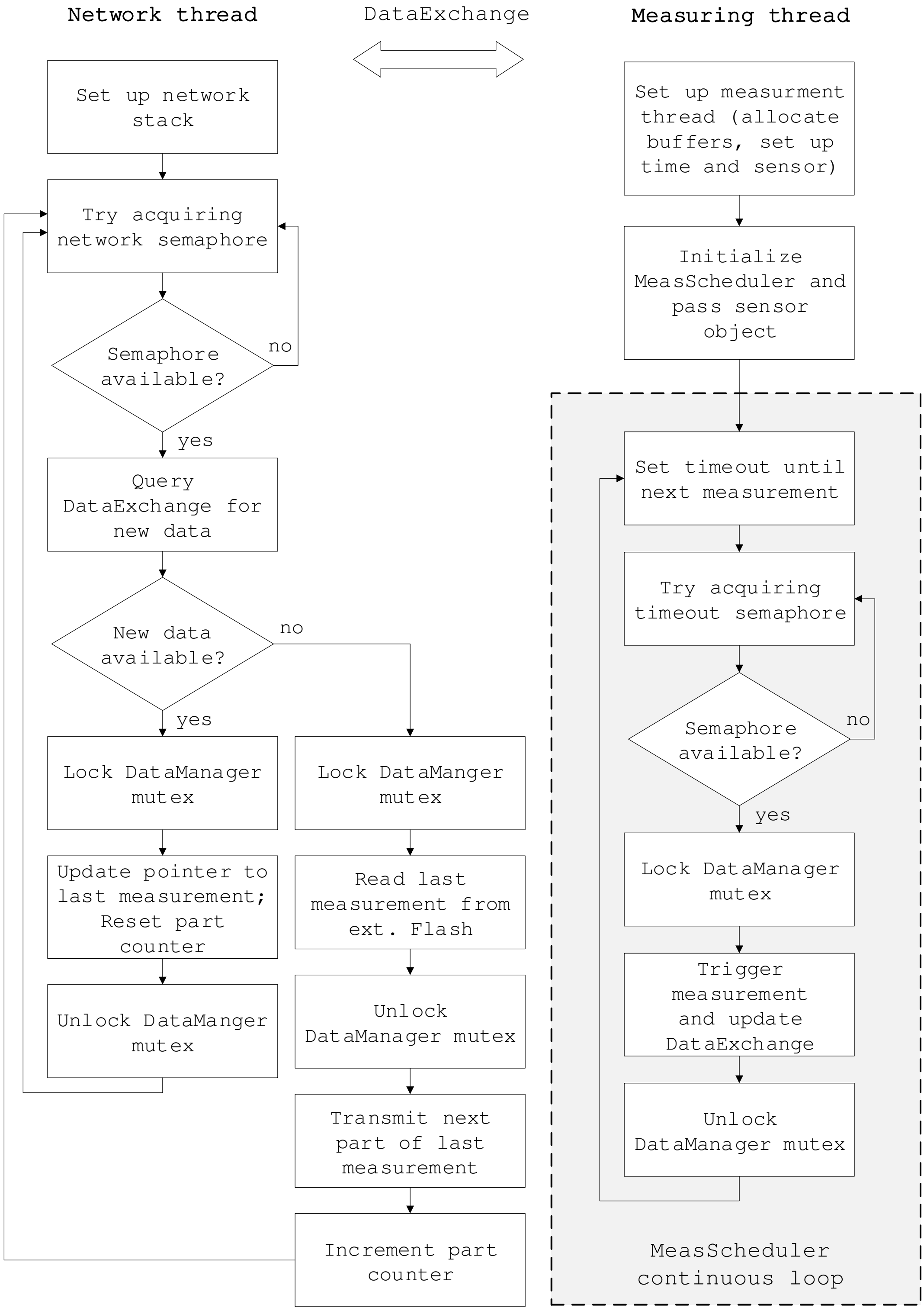

Network thread
DataExchange
Measuring thread
Set up network stack
Try acquiring network semaphore
Semaphore available?
no
yes
Query DataExchange for new data
New data available?
no
yes
Lock DataManager mutex
Update pointer to last measurement; Reset part counter
Unlock DataManger mutex
Lock DataManger mutex
Read last measurement from ext. Flash
Unlock DataManager mutex
Transmit next part of last measurement
Increment part counter
Set up measurment thread (allocate buffers, set up time and sensor)
Initialize MeasScheduler and pass sensor object
Set timeout until next measurement
Try acquiring timeout semaphore
Semaphore available?
no
yes
Lock DataManager mutex
Trigger measurement and update DataExchange
Unlock DataManager mutex
MeasScheduler continuous loop

Fig. 9.1. The firmware is split up in two threads. The network thread is responsible for transmission of datasets, while the measuring thread periodically triggers measurements.

The method at first iterates over all network layers and calls their transmissionPending() method to notify them to prepare for transmission. Then the prepareBuffer(...) method copies data from the layer's buffer to the transmission buffer with respect to the Wireless M-Bus block structure, see tabulations 6.4, 6.5 and 6.6. This operation includes generation of CRC checksums (Chapter 4.1). Finally the transmission buffer is encoded using the specified scheme and passed to the RF core.

9.2.2 Measurement Scheduler

The singleton class MeasScheduler, see Fig. 9.4, is responsible for scheduling measurements. It maintains a list of sensors that it is responsible for.

At first, sensor objects have to be created and added to a list, in this case to a Deque<AbstractSensor>, see Fig. 7.8. The list then has to be passed to the MeasScheduler by the method setSensorList(...).

Before usage, the MeasScheduler has to be initiated by initializeScheduler(). This sets up the clock module and the semaphore that are used together for timing.

Then runContinuousLoop() is called. The method is an endless loop that cyclically reschedules measurements, waits until the next measurement has to be started and triggers it. Rescheduling iterates over all AbstractSensors and uses the timeLastMeasurement and periodMeasurement properties, which define when the last measurement was triggered and the period, respectively, to identify the sensor that has to be triggered next. The time until the next measurement is calculated and then a timer is set up to generate an interrupt at that time. In the meantime, the task is waiting for the semaphore that is to be posted by the timer interrupt.

Finally, the measurement is triggered and the DataExchange objects are notified, that the particular sensor has acquired new data. Then the cycle starts again with rescheduling.

9.2.3 General Sensor Interface

The class AbstractSensor, see Fig. 9.4, provides a general interface to time and trigger measurements. Each concrete sensor is represented by a class derived from AbstractSensor. It utilizes or implements a driver for actual communication with the sensor. This interface allows the MeasScheduler to handle a number of different sensors.

9.2.4 Memory Management

The abstract class `DataManager`, see Fig. 9.5, is defining an interface for temporary and permanently[1] storing measuring data. Since data can be stored permanently on e. g. an internal flash memory, an external flash memory or an Electrically Erasable Programmable Read-Only Memory (EEPROM), it was decided to use an abstract class for storage. In this case an external flash memory was used and therefore the abstract class was implemented for an external flash.

To use the `DataManager`, at first the type of data and header size of the application have to be specified in the constructor. The class internally uses byte arrays for storage and takes care of type conversion. Then `AbstractComputation` type objects can be added to the `DataManager` object. The specified computations are applied to the measuring data before permanently saving them. The `DataManager` object is then passed to the `AbstractSensor` that uses it. The `AbstractSensor` can now use the methods `addHeaderData(...)` and `addData(...)` to add header data, respectively, measuring data. If it is finished, it has to call `save()` to trigger performance of computations and then permanently save the data.

Flash memory can only be deleted in blocks. More precisely, each bit can separately be toggled from “1” to “0” while resetting bits from “0” to “1” is only feasible in large blocks. Furthermore a mechanism is required that enables to delimit particular datasets in memory.

It was decided to further segment each block of the external flash memory according to Fig. 9.6. Each segment has a header specifying if it is the first segment representing a particular dataset and if the segment is valid, see Tab. 9.1. The segment header is followed by an index that is indicating blocks that belong to the same dataset. If the segment is the first one representing a particular dataset, the length of the dataset follows. This allows to check if all segments of a particular dataset are available and valid. The remaining bytes of the segment are used for actual data.

If a dataset is written to the external flash as a number of segments, at first a header stating that the segment is currently being written is used. After the whole dataset has been written, the distinctive bit changing the status from *is being written* to *valid* is cleared. This enables detection of corrupted data caused by a reset during the writing operation.

The class `DataManager` can easily be adapted to a wear leveling enabled permanent storage for settings.

9.2.5 Auxiliary Classes

Sensor data as well as networking requires time of day. Time is maintained within a RTC module that is often battery backed up. To simplify generation of a timestamp or to set up time, the singleton class `TimeMod` was implemented, see Fig. 9.7. The method `getTime(...)` can be used to create a timestamp, while `setTime(...)` is used for setup.

[1] Note that permanent means as long as the allocated memory is not full. Then it is overwritten in a circular buffer manner.

Segment header			
Bit 3	Bit 2	Bit 1	Bit 0
H	S2	S1	S0

H	Definition
0	Second or subsequent segment (no length field)
1	First segment

S2-S0	Definition
111	Segment is empty
110	Segment is being written
100	Segment is valid
000	Segment is invalid

Tab. 9.1. Header used for storage of a segment in external flash memory.

In a battery supplied system it is important to provide methods for monitoring the battery level. Therefore the singleton class `BatteryTempMod` was implemented. See Fig. 9.8. It allows to query supply voltage and battery level via `getVoltage()` and `getBatteryPercent()` respectively. Furthermore the interface can be used to read the device temperature acquired by an internal temperature sensor via `getTemerature()`.

9.2.6 Over the Air Download

Over the air download was implemented according to a Texas Instruments code sample using their Bluetooth stack together with a bootloader. The over the air download functionality is implemented in the application firmware, while the bootloader is a stand-alone firmware that is responsible for checking available firmware during the boot process and branching to a specific one. It is not modified by an application firmware update.

Over the air download was only used in early versions of the firmware, since the 868 MHz and 2.4 GHz PCB dual-band antenna then was replaced by an external antenna only supporting 868 MHz.

9.3 Low power deep sleep

Entering low power deep sleep mode when idle is initiated by the Texas Instruments TI-RTOS operating system. Each driver used maintains power constraints that the operating system has to consider when choosing a low-power mode. The mode with the lowest power consumption that allows timers as wake-up sources is the standby mode. It is usually used when both threads wait on their particular semaphores to be posted by timers.

9.4 Extendability

In this firmware, close attention was given to extendability. Usage of abstract classes for sensors, the type of non-volatile memory and computations offers a very flexible solution. For example, without changing everything else, a different sensor could be used by simply implementing `AbstractSensor` for it. Due to the fact that `MeasScheduler` is capable of handling a list of `AbstractSensors`, actually a number of sensors can be used simultaneously without any firmware modification. Any desired computation can be achieved by simply deriving from `AbstractComputation` and appending the object to the `DataManager` instance that is responsible for the particular sensor.

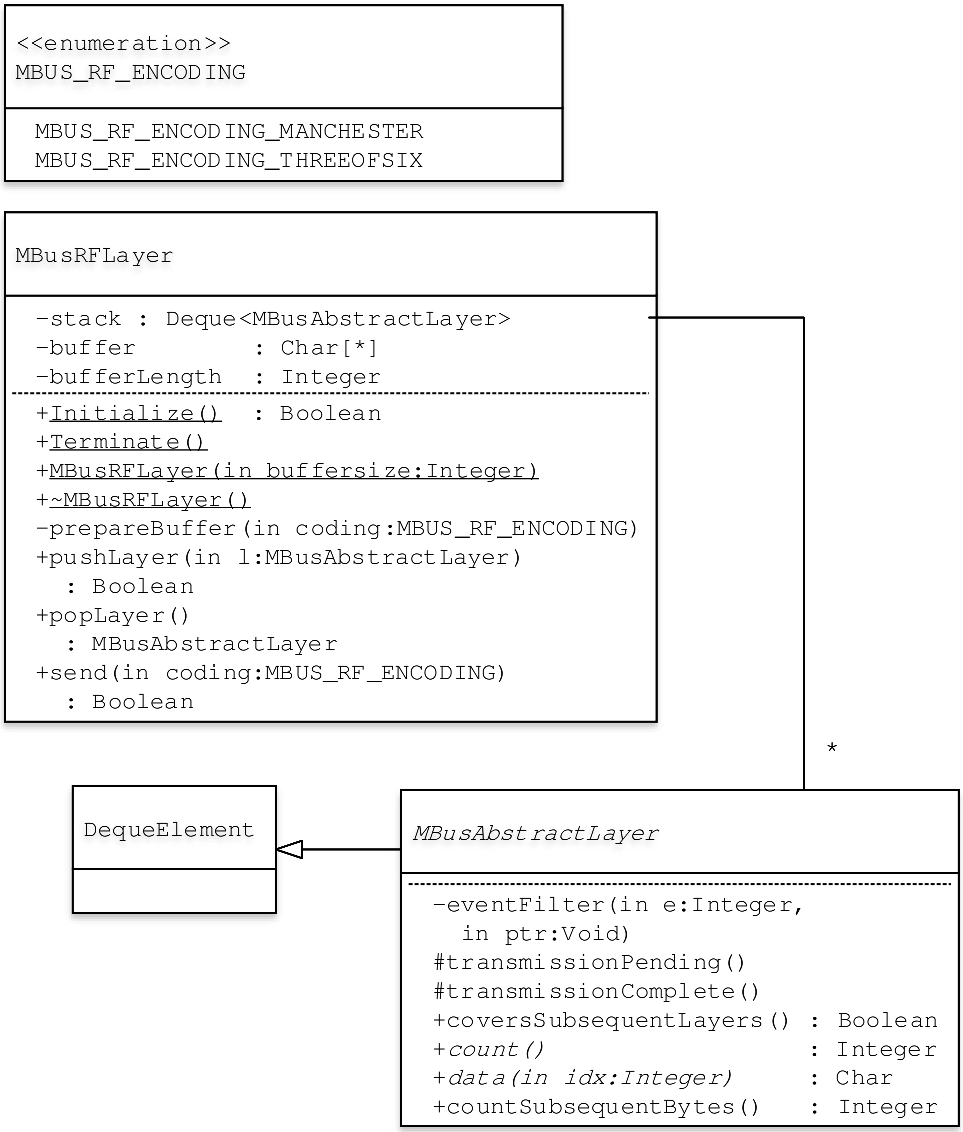

<<enumeration>>
MBUS_RF_ENCODING
MBUS_RF_ENCODING_MANCHESTER
MBUS_RF_ENCODING_THREEOFSIX
MBusRFLayer
-stack : Deque<MBusAbstractLayer>
-buffer : Char[*]
-bufferLength : Integer
+Initialize() : Boolean
+Terminate()
+MBusRFLayer(in buffersize:Integer)
+~MBusRFLayer()
-prepareBuffer(in coding:MBUS_RF_ENCODING)
+pushLayer(in l:MBusAbstractLayer)
: Boolean
+popLayer()
: MBusAbstractLayer
+send(in coding:MBUS_RF_ENCODING)
: Boolean
*
DequeElement
MBusAbstractLayer
-eventFilter(in e:Integer,
in ptr:Void)
#transmissionPending()
#transmissionComplete()
+coversSubsequentLayers() : Boolean
+count() : Integer
+data(in idx:Integer) : Char
+countSubsequentBytes() : Integer

Fig. 9.2. The class `MBusRFLayer` is defining the lowest layer in the network stack, the physical layer. The `MBusAbstractLayer` base class provides functionality to network layers to communicate with each other.

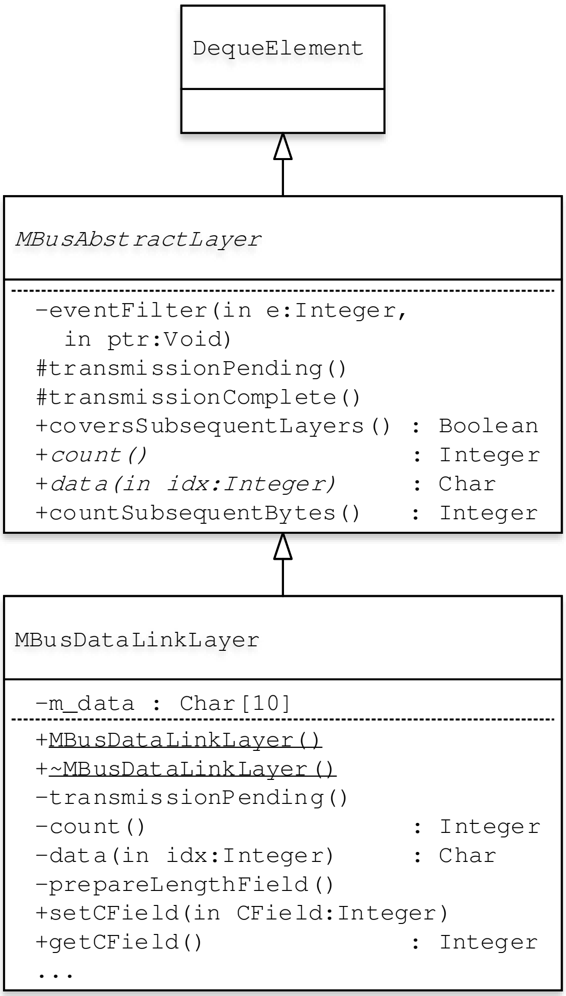


Fig. 9.3. The class `MBusDataLinkLayer` implements the data link layer.

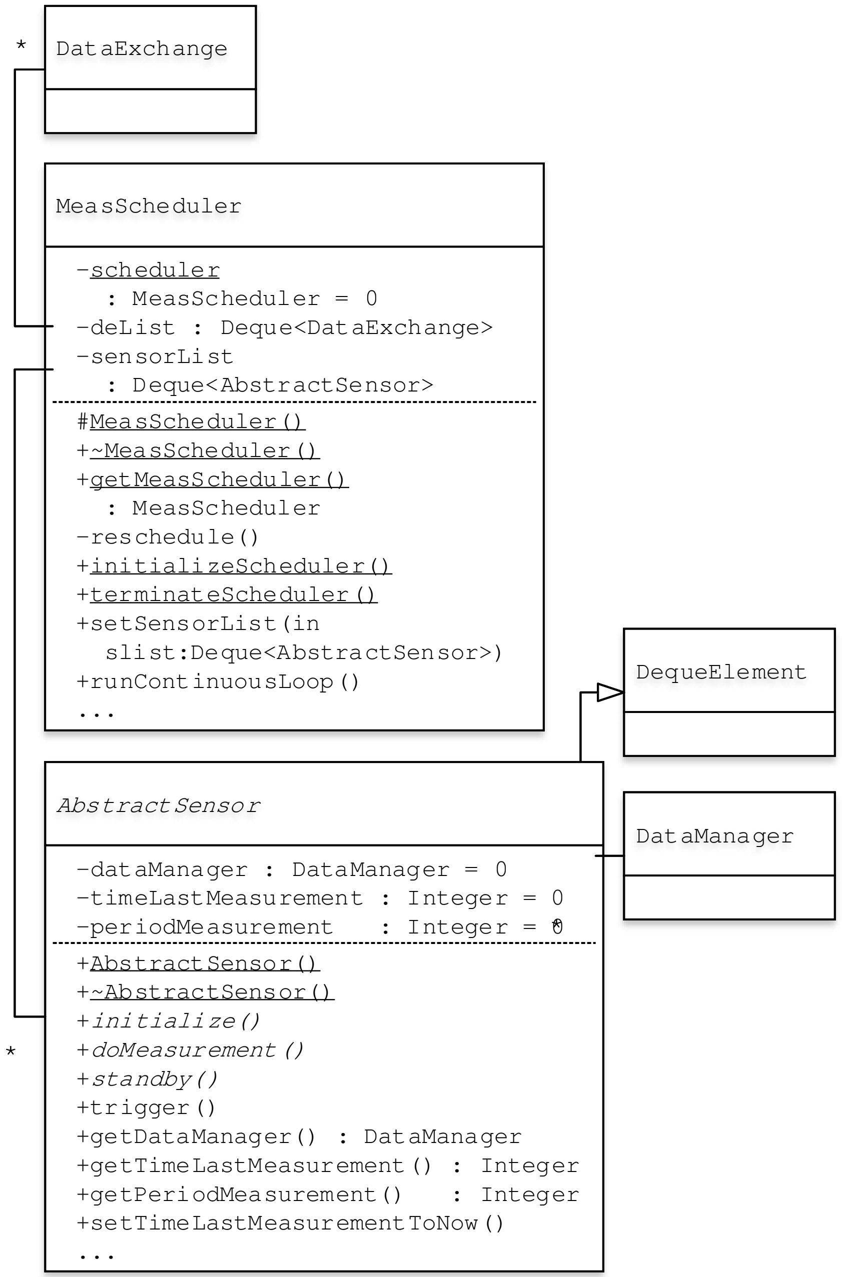


Fig. 9.4. The class `MeasScheduler` is responsible for cyclically triggering sensors's measurement routine on time.

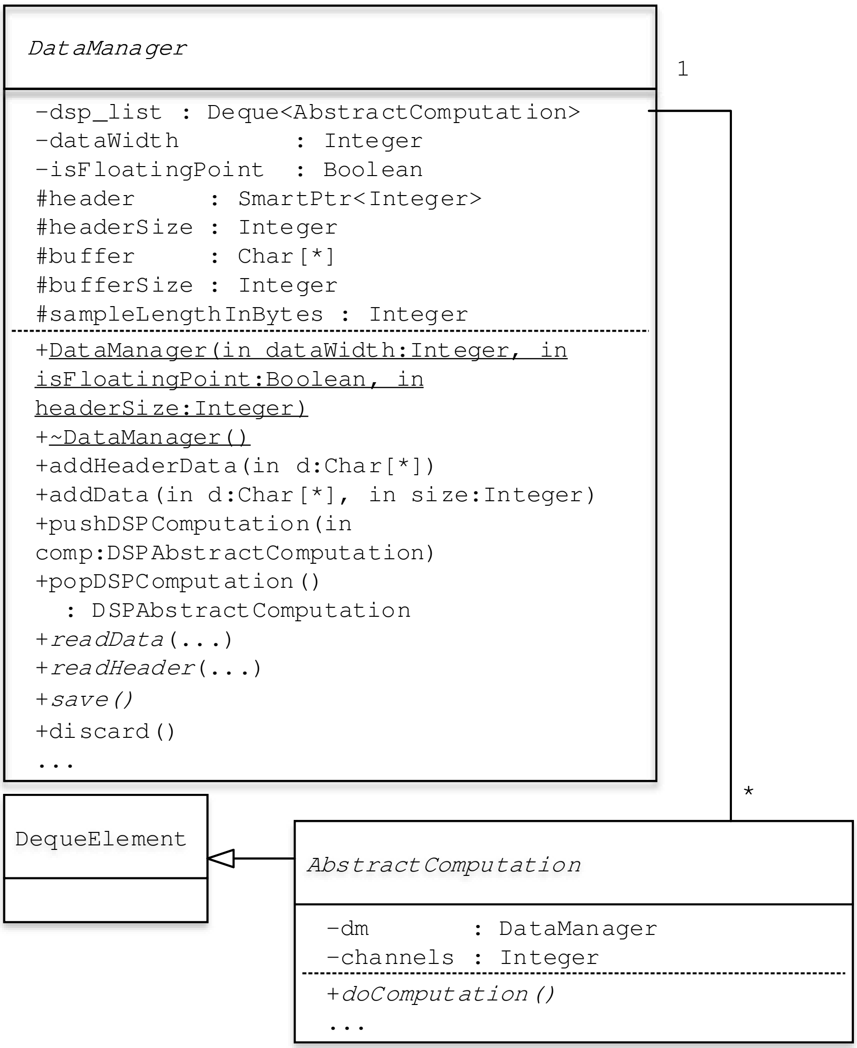

DataManager
1
-dsp_list : Deque<AbstractComputation>
-dataWidth : Integer
-isFloatingPoint : Boolean
#header : SmartPtr<Integer>
#headerSize : Integer
#buffer : Char[*]
#bufferSize : Integer
#sampleLengthInBytes : Integer
+DataManager(in dataWidth:Integer, in isFloatingPoint:Boolean, in headerSize:Integer)
+~DataManager()
+addHeaderData(in d:Char[*])
+addData(in d:Char[*], in size:Integer)
+pushDSPComputation(in comp:DSPAbstractComputation)
+popDSPComputation() : DSPAbstractComputation
+readData(...)
+readHeader(...)
+save()
+discard()
...
*
DequeElement
AbstractComputation
-dm : DataManager
-channels : Integer
+doComputation()
...

Fig. 9.5. The class `DataManager` is responsible for temporary buffering of measuring data, as well as for processing and permanently saving it.

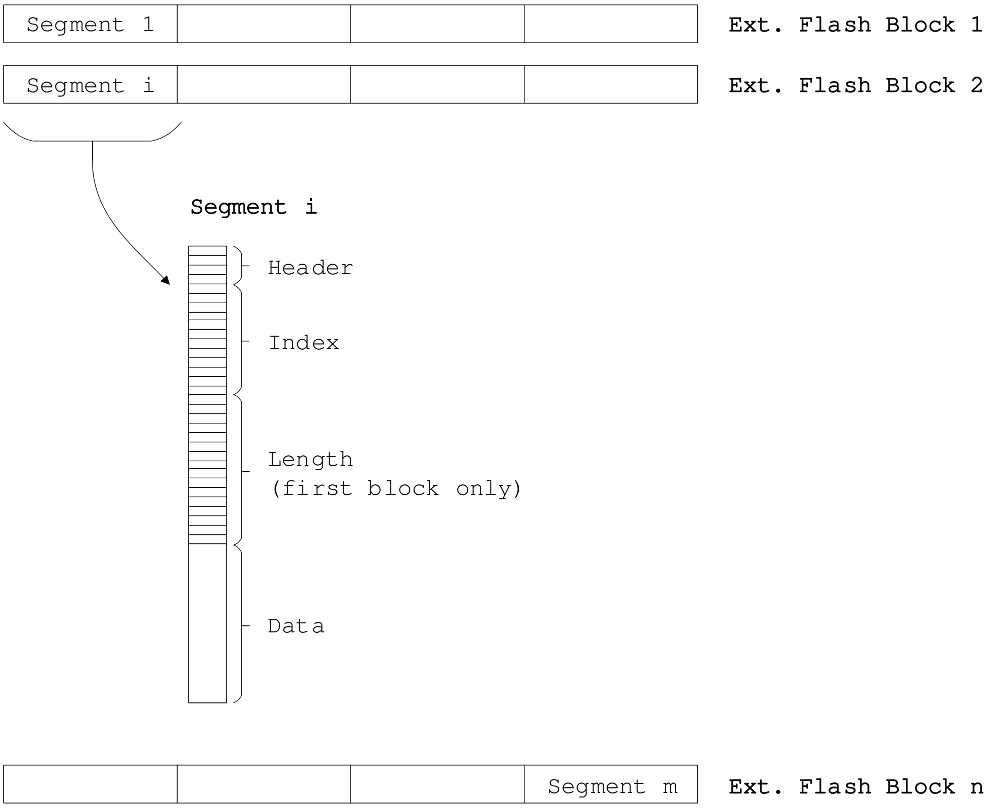
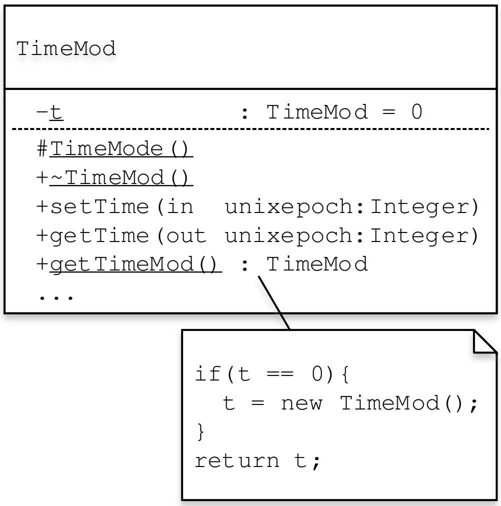
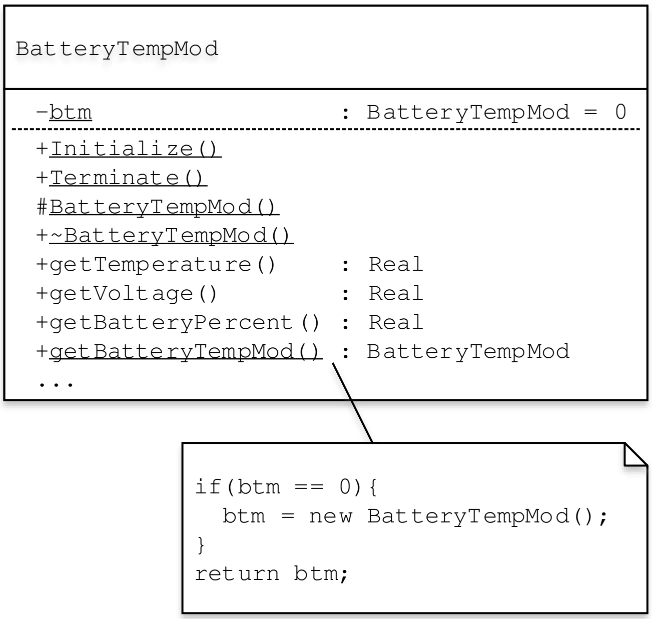


Fig. 9.6. The external flash is arranged in segments.



```
if(t == 0){
  t = new TimeMod();
}
return t;
```

Fig. 9.7. The class `TimeMod` is a simple interface to get and set system time.



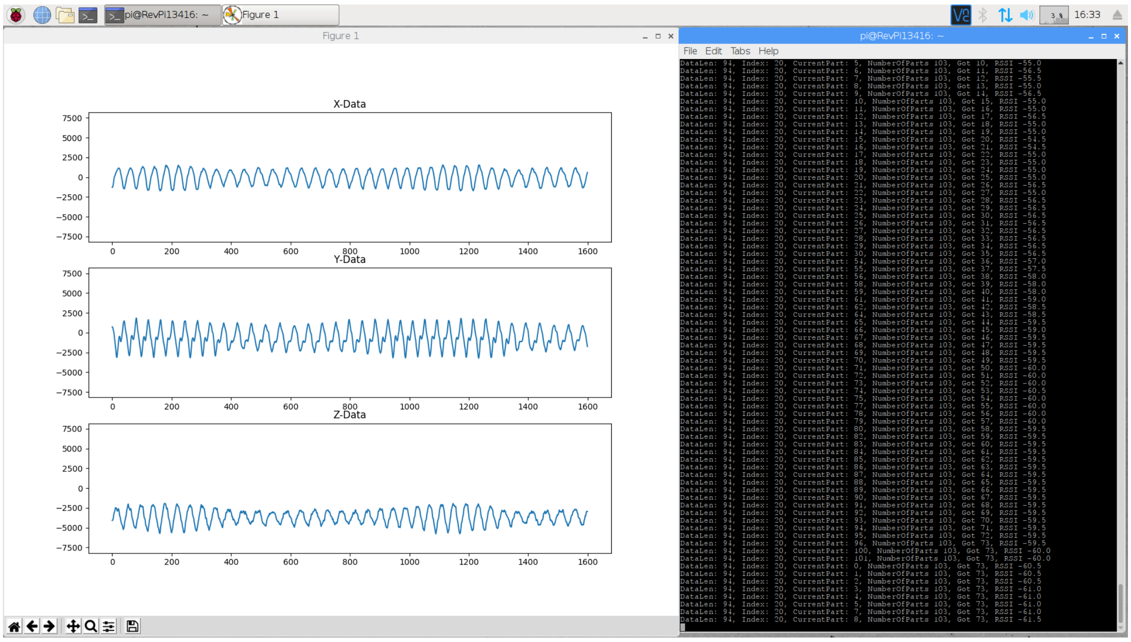
```
if(btm == 0){
  btm = new BatteryTempMod();
}
return btm;
```

Fig. 9.8. The class `BatteryTempMod` implements an interface to query power supply voltage, battery level and the temperature of the device.

10 Software

The software developed for the IPC is written in Python 3 and is intended for demonstration as well as development purposes, see Fig. 10.1.

It is capable of plotting three dimensional vibration data. Furthermore it displays the number of packets the current dataset consists of and how many of them have already been received. Additionally, Received Signal Strength Indicator (RSSI) values are printed out which correspond to the received power. The RSSI value of each part belonging to a certain dataset is written to a Comma Separated Value (CSV) file. Note that the manufacturer of the edge gateway's Wireless M-Bus module defines the RSSI value as twice the received power in dBm, see [1]. The conversion to dBm is handled by the software.



```
DataLen: 94, Index: 20, CurrentPart: 5, NumberOfParts 103, Got 10, RSSI -55.0
DataLen: 94, Index: 20, CurrentPart: 6, NumberOfParts 103, Got 11, RSSI -56.5
DataLen: 94, Index: 20, CurrentPart: 7, NumberOfParts 103, Got 12, RSSI -55.5
DataLen: 94, Index: 20, CurrentPart: 8, NumberOfParts 103, Got 13, RSSI -55.0
DataLen: 94, Index: 20, CurrentPart: 9, NumberOfParts 103, Got 14, RSSI -56.5
DataLen: 94, Index: 20, CurrentPart: 10, NumberOfParts 103, Got 15, RSSI -55.0
DataLen: 94, Index: 20, CurrentPart: 11, NumberOfParts 103, Got 16, RSSI -55.0
DataLen: 94, Index: 20, CurrentPart: 12, NumberOfParts 103, Got 17, RSSI -56.5
DataLen: 94, Index: 20, CurrentPart: 13, NumberOfParts 103, Got 18, RSSI -55.0
DataLen: 94, Index: 20, CurrentPart: 14, NumberOfParts 103, Got 19, RSSI -55.0
DataLen: 94, Index: 20, CurrentPart: 15, NumberOfParts 103, Got 20, RSSI -54.5
DataLen: 94, Index: 20, CurrentPart: 16, NumberOfParts 103, Got 21, RSSI -54.5
DataLen: 94, Index: 20, CurrentPart: 17, NumberOfParts 103, Got 22, RSSI -55.0
DataLen: 94, Index: 20, CurrentPart: 18, NumberOfParts 103, Got 23, RSSI -55.0
DataLen: 94, Index: 20, CurrentPart: 19, NumberOfParts 103, Got 24, RSSI -55.0
DataLen: 94, Index: 20, CurrentPart: 20, NumberOfParts 103, Got 25, RSSI -55.0
DataLen: 94, Index: 20, CurrentPart: 21, NumberOfParts 103, Got 26, RSSI -56.5
DataLen: 94, Index: 20, CurrentPart: 22, NumberOfParts 103, Got 27, RSSI -55.0
DataLen: 94, Index: 20, CurrentPart: 23, NumberOfParts 103, Got 28, RSSI -56.5
DataLen: 94, Index: 20, CurrentPart: 24, NumberOfParts 103, Got 29, RSSI -56.5
DataLen: 94, Index: 20, CurrentPart: 25, NumberOfParts 103, Got 30, RSSI -56.5
DataLen: 94, Index: 20, CurrentPart: 26, NumberOfParts 103, Got 31, RSSI -56.5
DataLen: 94, Index: 20, CurrentPart: 27, NumberOfParts 103, Got 32, RSSI -56.5
DataLen: 94, Index: 20, CurrentPart: 28, NumberOfParts 103, Got 33, RSSI -56.5
DataLen: 94, Index: 20, CurrentPart: 29, NumberOfParts 103, Got 34, RSSI -56.5
DataLen: 94, Index: 20, CurrentPart: 30, NumberOfParts 103, Got 35, RSSI -56.5
DataLen: 94, Index: 20, CurrentPart: 54, NumberOfParts 103, Got 36, RSSI -57.0
DataLen: 94, Index: 20, CurrentPart: 55, NumberOfParts 103, Got 37, RSSI -57.5
DataLen: 94, Index: 20, CurrentPart: 56, NumberOfParts 103, Got 38, RSSI -58.0
DataLen: 94, Index: 20, CurrentPart: 58, NumberOfParts 103, Got 39, RSSI -58.0
DataLen: 94, Index: 20, CurrentPart: 59, NumberOfParts 103, Got 40, RSSI -58.0
DataLen: 94, Index: 20, CurrentPart: 61, NumberOfParts 103, Got 41, RSSI -59.0
DataLen: 94, Index: 20, CurrentPart: 62, NumberOfParts 103, Got 42, RSSI -58.5
DataLen: 94, Index: 20, CurrentPart: 64, NumberOfParts 103, Got 43, RSSI -58.5
DataLen: 94, Index: 20, CurrentPart: 65, NumberOfParts 103, Got 44, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 66, NumberOfParts 103, Got 45, RSSI -59.0
DataLen: 94, Index: 20, CurrentPart: 67, NumberOfParts 103, Got 46, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 68, NumberOfParts 103, Got 47, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 69, NumberOfParts 103, Got 48, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 70, NumberOfParts 103, Got 49, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 71, NumberOfParts 103, Got 50, RSSI -60.0
DataLen: 94, Index: 20, CurrentPart: 72, NumberOfParts 103, Got 51, RSSI -60.0
DataLen: 94, Index: 20, CurrentPart: 73, NumberOfParts 103, Got 52, RSSI -60.0
DataLen: 94, Index: 20, CurrentPart: 74, NumberOfParts 103, Got 53, RSSI -60.5
DataLen: 94, Index: 20, CurrentPart: 75, NumberOfParts 103, Got 54, RSSI -60.0
DataLen: 94, Index: 20, CurrentPart: 77, NumberOfParts 103, Got 55, RSSI -60.0
DataLen: 94, Index: 20, CurrentPart: 78, NumberOfParts 103, Got 56, RSSI -60.0
DataLen: 94, Index: 20, CurrentPart: 79, NumberOfParts 103, Got 57, RSSI -60.0
DataLen: 94, Index: 20, CurrentPart: 80, NumberOfParts 103, Got 58, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 82, NumberOfParts 103, Got 59, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 83, NumberOfParts 103, Got 60, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 84, NumberOfParts 103, Got 61, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 85, NumberOfParts 103, Got 62, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 86, NumberOfParts 103, Got 63, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 87, NumberOfParts 103, Got 64, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 88, NumberOfParts 103, Got 65, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 89, NumberOfParts 103, Got 66, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 90, NumberOfParts 103, Got 67, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 91, NumberOfParts 103, Got 68, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 92, NumberOfParts 103, Got 69, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 93, NumberOfParts 103, Got 70, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 94, NumberOfParts 103, Got 71, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 95, NumberOfParts 103, Got 72, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 96, NumberOfParts 103, Got 73, RSSI -59.5
DataLen: 94, Index: 20, CurrentPart: 100, NumberOfParts 103, Got 73, RSSI -60.0
DataLen: 94, Index: 20, CurrentPart: 101, NumberOfParts 103, Got 73, RSSI -60.0
DataLen: 94, Index: 20, CurrentPart: 0, NumberOfParts 103, Got 73, RSSI -60.5
DataLen: 94, Index: 20, CurrentPart: 1, NumberOfParts 103, Got 73, RSSI -60.5
DataLen: 94, Index: 20, CurrentPart: 2, NumberOfParts 103, Got 73, RSSI -60.5
DataLen: 94, Index: 20, CurrentPart: 3, NumberOfParts 103, Got 73, RSSI -61.0
DataLen: 94, Index: 20, CurrentPart: 4, NumberOfParts 103, Got 73, RSSI -61.0
DataLen: 94, Index: 20, CurrentPart: 5, NumberOfParts 103, Got 73, RSSI -61.0
DataLen: 94, Index: 20, CurrentPart: 7, NumberOfParts 103, Got 73, RSSI -61.0
DataLen: 94, Index: 20, CurrentPart: 8, NumberOfParts 103, Got 73, RSSI -61.5
```

Fig. 10.1. The software is capable of plotting vibration data as well as displaying and logging received datagrams.

10.1 Interfacing the Wireless M-Bus module

The Wireless M-Bus module is connected to the IPC via a proprietary *connect bridge* interface. The interface emulates a serial connection to the device. The communication protocol is pure binary. It does not represent any command as human readable text.

During debugging, the application `minicom` was used to display incoming data, while

```
printf "[message]" > /dev/ttyConBrdige
```

was used to send commands to the device.

The script discussed in this chapter assumes that the following Wireless M-Bus module settings are already set up.

- Matching serial port settings
- Master (*other*) network role
- T1 or T2 M-Bus mode
- Binding of the meter to the module (*other*) or disabled filtering
- RSSI mode enabled
- Data interface with start and stop byte

10.2 Program Flow

The main routine of the program is described in Fig 10.2. It makes use of the `readDataSet()` function that returns the last completely received set of acceleration data acquired by the sensor.

The function `readDataSet()`, see Fig. 10.3, is responsible for collecting chunks in which the dataset is split for transmission and reassembles them. It utilizes the `readPart()` function for actual reception of Wireless M-Bus datagrams.

The function `readPart()`, see Fig 10.4, is used to identify Wireless M-Bus packets that are passed by the Wireless M-Bus module via the serial interface. It uses start and stop bytes to identify Wireless M-Bus packets, together with the length information defined in the datagram's data link layer. This is necessary, since the Wireless M-Bus module separates datagrams only by a short time delay. Evaluating the length of the datagram avoids misidentification of packets as far as possible. Note that a Wireless M-Bus datagram can also contain characters equal to the start and end bytes.

[1] This functionality is defined within `readDataSet()`.

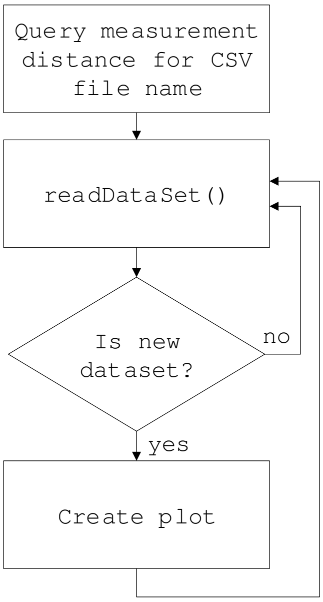


Fig. 10.2. The main program starts with querying the distance between meter and other. It is used for naming the CSV file[1]that collects RSSI values. Then `readDataSet()` is called, which is listening for sensor data and returns a complete dataset. If the dataset is different from the current plot, a replot is issued.

10.3 Integration in Other Projects

While this demonstration software only implements basic features, the functions `readPart()` and `readDataSet()` can also be used in more comprehensive applications.

It is highly recommended to implement a bidirectional communication for industrial applications to facilitate retransmission of lost datagrams according to Fig 6.1.

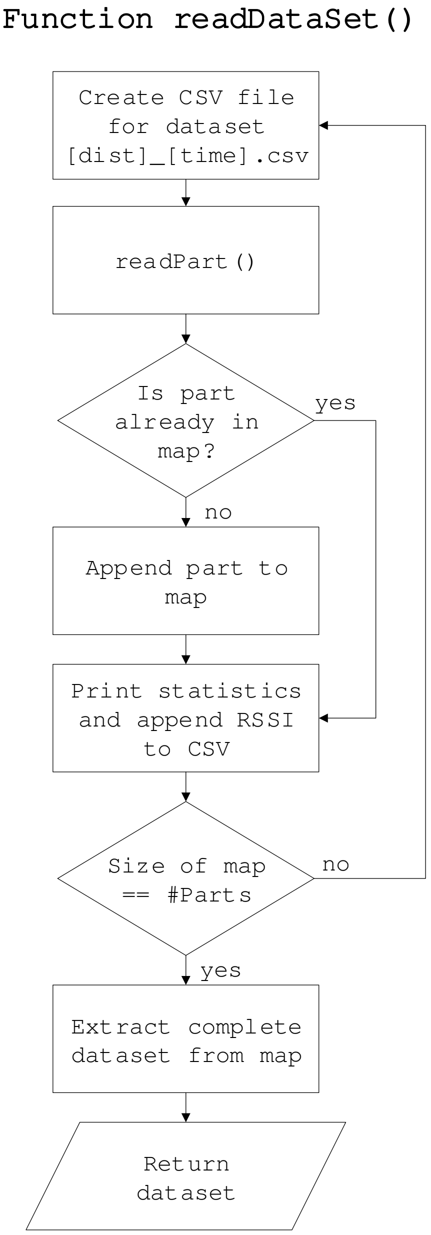


Fig. 10.3. The function `readDataSet()` recurrently calls `readPart()` to gradually get all chunks of a dataset. The chunks are collected in a map that assigns the chunk's data to it's part number. After a new part was received, the new statistics of received parts is printed to the console and it's RSSI value is appended to a CSV file logging the transmission of the current dataset. When all chunks of the dataset were received, in other words, the size of the map is equal to the number of parts in which the dataset was divided, the complete dataset is assembled and returned.

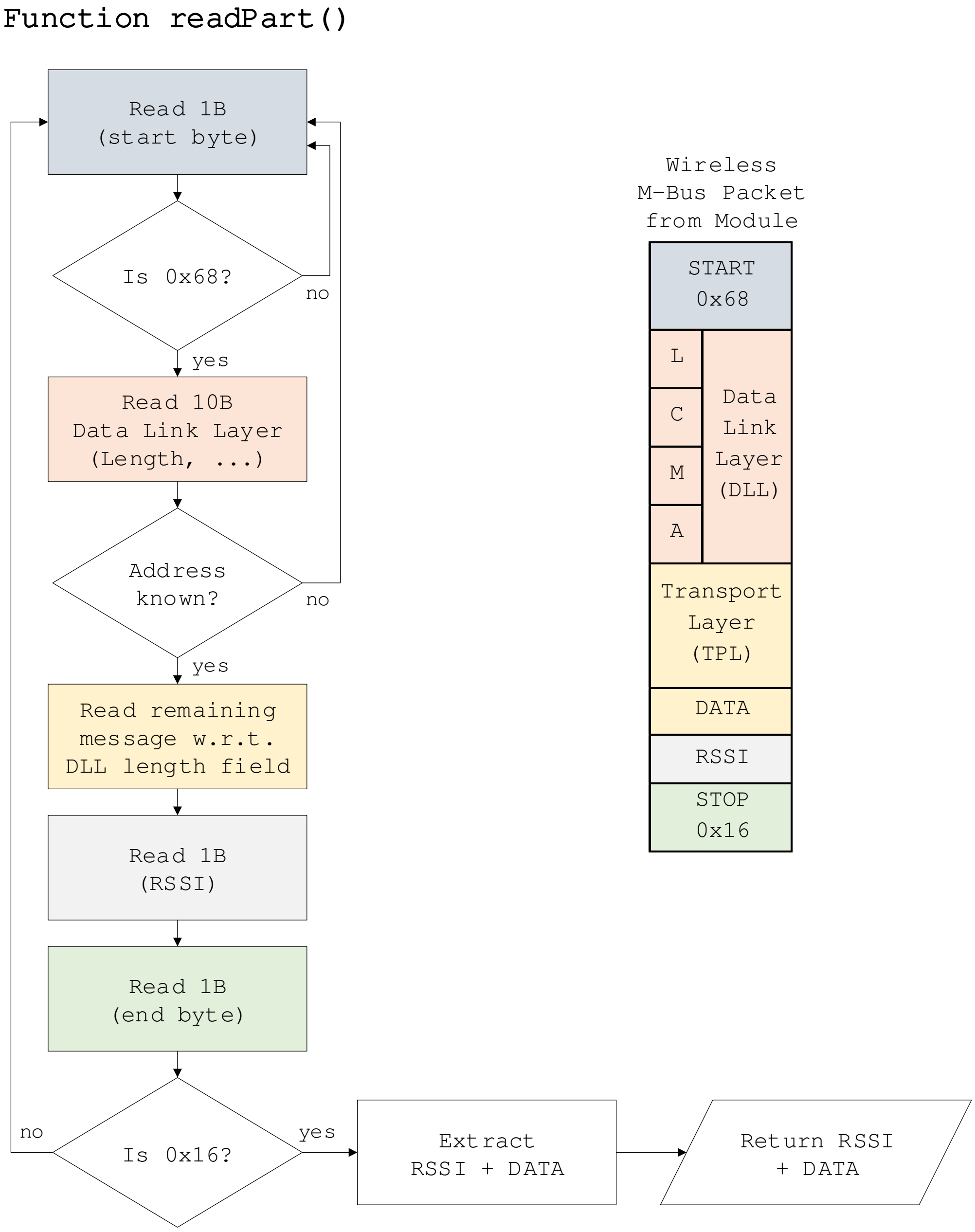


Fig. 10.4. The function `readPart()` handles the communication with the Wireless M-Bus module via a serial interface. At first, it consecutively reads single bytes from the serial interface until the start byte 0x68 is encountered. The function then assumes that the next 10 B are the data link layer and checks if the address of the sending meter is known. In that case the remaining message with respect to the data link layer's length field is read, as well as the following RSSI value. As a last check for a valid message the end byte is validated. The function returns the RSSI value and the transmitted chunk of the measuring data.

Part III

Assessment

11 Evaluation

In this chapter the developed condition monitoring system is evaluated in terms of transmission range and the smart sensor's current consumption.

11.1 Range

The transmission range of the prototype system was evaluated for 5 dBm and 14 dBm transmission power.

11.1.1 Experimental Setup

To evaluate transmission range, the prototype setup, depicted in figures 8.1 and 8.3, was used together with omnidirectional aerials.

The firmware was adapted to enable a triggered sending of exactly 1000 packets, each with a size of 20 bytes not including headers. The software running on the edge device logged the RSSI values for all correctly received packages to a CSV file. In this manner the number of incorrectly received packets can be determined. The number of lines of the CSV file divided by the total amount of 1000 sent packets yields the ratio of correctly received packets. Logging the RSSI value for each received packet at a given range allows the estimation of an interval for the RSSI value at a given distance. Neither error correction codes nor retransmissions were used, so that the fundamental transmission properties could be evaluated.

The transmission range was measured in horizontal direction by a GPS device.

At first, the transmission range was estimated by (5.4) and (5.3) together with the following assumptions:

1. receiver sensitivity $P_r = 101\,\mathrm{dBm}$,
2. transmission power $P_t = 5\,\mathrm{dBm}$ respectively $P_t = 14\,\mathrm{dBm}$,
3. antenna gain factors G_t and G_r that compensate transmit and receive chain losses L_t and L_r,
4. link margin of $L_m = 15\,\mathrm{dBm}$ as recommended in [24] and
5. a factor n of 2.5 according to [24], which was assessed at an open field 1.5 m above the ground.

This yields distances of 117 m for 5 dBm and 275 m for 14 dBm at a received signal strength of −86 dBm. Reliable transmissions should be expectable in the calculated range. Error correction can be used to extend this range.

11.1.2 Results

The results of the range evaluation are depicted in figures 11.1 and 11.2. With a transmission power of 5 dBm ≈ 3 mW ranges of over 150 m could be achieved, while a transmission power of 14 dBm ≈ 25 mW allowed transmission ranges of over 400 m. The locations that were chosen for the measurements avoided objects projecting into the Fresnel zone as good as was possible.

The omnidirectional characteristic of the antenna is valid approximately around a horizontal plane. At short measurement distances there was a significant vertical displacement between the devices; such that the near field effects in the Z-direction caused errors.

Differences considering the 5 dBm and 14 dBm tests in RSSI fluctuation as well as packet loss at equal RSSI levels are assumed to be effected by different interferences at the different locations.

Differences compared to the estimated transmission ranges are assumed to be caused by a conservative n factor.

11.2 Current Consumption

In this chapter the current consumption of the smart sensor is evaluated for the same setup using 5 dBm respectively 14 dBm transmission power. A typical application defined by:

1. 3200 Hz sample frequency,
2. 500 ms sample time,
3. 6 B per sample,
4. one measurement per hour,
5. a collected transmission of measurements at six-hourly intervals,
6. 950 mA h coin cell battery supply and
7. 4 s advertising interval

is assumed.

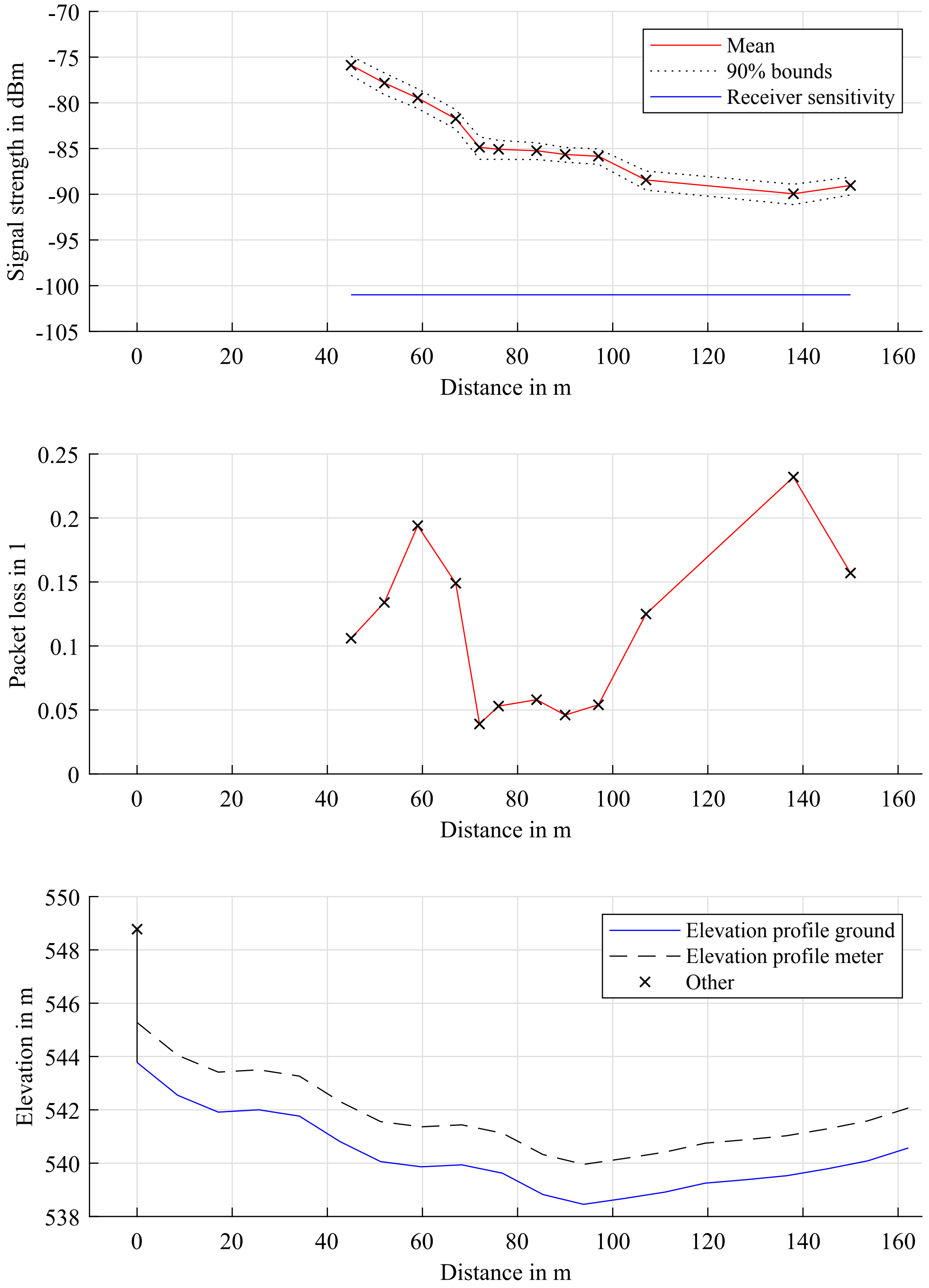


Fig. 11.1. The transmission range was evaluated for a transmission power of 5 dBm. Elevation data powered by Google.

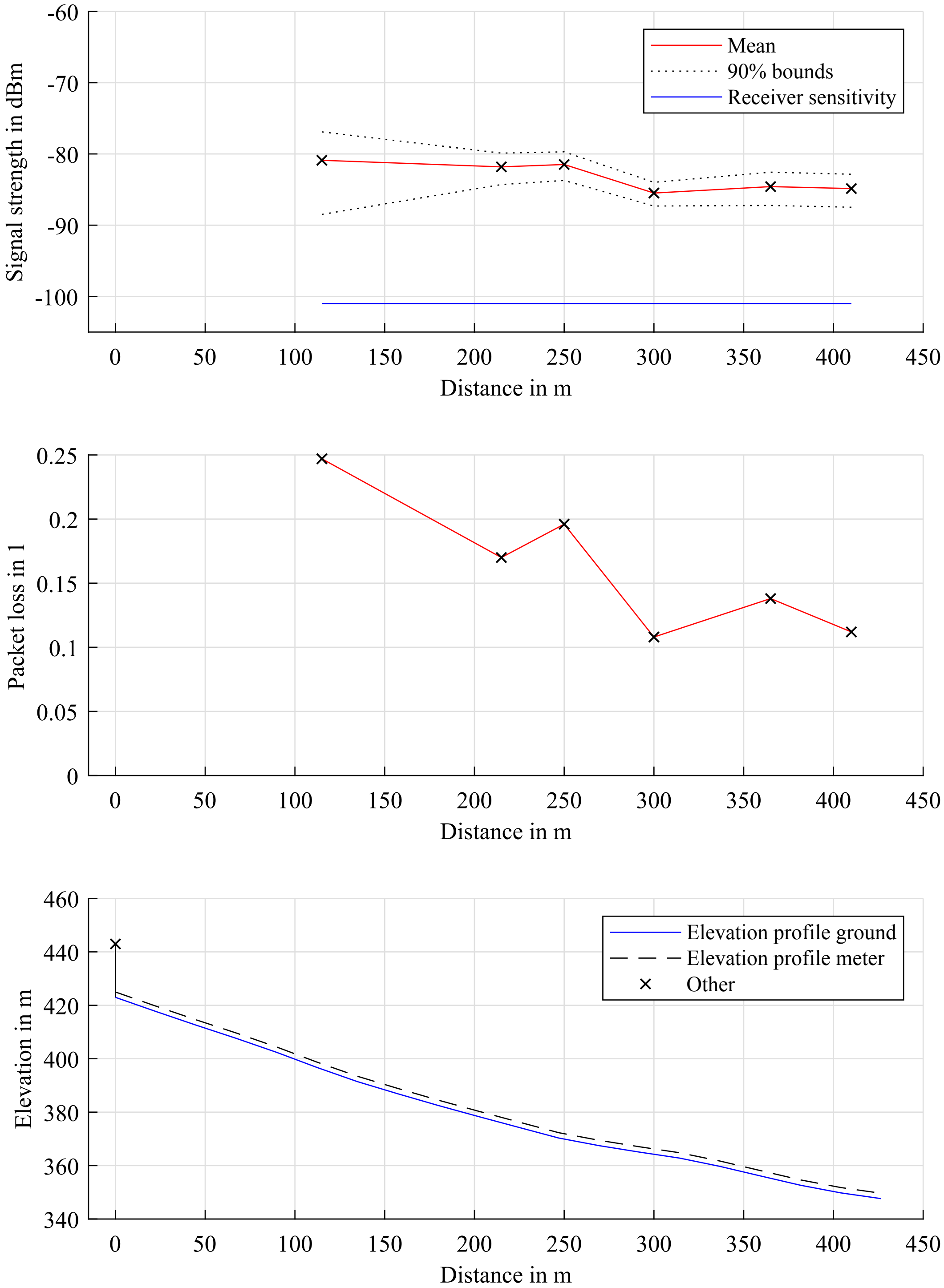


Fig. 11.2. The transmission range was evaluated for a transmission power of 14 dBm. Elevation data powered by Google.

11.2.1 Experimental Setup

To measure currents, the circuit explained in Chapter 8.3 was used together with a Rigol DS1054Z oscilloscope. The current measuring circuit was supplied by a power supply and the jumpers of the Texas Instruments CC1352P development board were configured in order to detach the debugger and level shifters. Additionally, the LED from the accelerometer's evaluation board was removed. Note that the influence of the voltage regulator's quiescent current is neglected.

11.2.2 Results

In the oscilloscope graphs, see figures 11.3 to 11.5, the yellow line represents the inverted, unfiltered output of the current measurement circuit. The blue line represents the inverted filtered output using a low-pass filter with $R_f = 10\,\mathrm{k\Omega}$ and $C_f = 1\,\mathrm{\mu F}$ yielding a critical frequency

$$f_c = 1/\left(2\pi R_f C_f\right) = 15.9\,\mathrm{Hz}. \tag{11.1}$$

Firstly the standby current of the MCU was evaluated. It is smaller than 1 µA and can therefore be neglected, see Fig. 11.3a.

According to Fig. 11.3b the current consumption for the sampling process can be estimated to be $2.1\,\mathrm{V}/500\,\Omega = 4.2\,\mathrm{mA}$ lasting for 0.5 s. The current consumption for saving the dataset is approximately $5.2\,\mathrm{V}/500\,\Omega = 10.4\,\mathrm{mA}$ and lasts approximately 440 ms. The energy used for sampling and saving of data per day

$$E_s = 24 \cdot (4.2\,\mathrm{mA} \cdot 0.5\,\mathrm{s} + 10.4\,\mathrm{mA} \cdot 440\,\mathrm{ms}) = 0.16\,\mathrm{mA\,s}. \tag{11.2}$$

According to Fig. 11.4b the average current consumption for transmission with a power of 5 dBm is about $2.94\,\mathrm{V}/500\,\Omega \cdot 12 \cdot 5\,\mathrm{ms}/100\,\mathrm{ms} = 3.53\,\mathrm{mA}$, while Fig. 11.5b estimates a current consumption of $1.09\,\mathrm{V}/100\,\Omega \cdot 12 \cdot 5\,\mathrm{ms}/100\,\mathrm{ms} = 6.54\,\mathrm{mA}$ for transmitting with 14 dBm. It is assumed that transmission is chunked in blocks of 100 B data sent with a period of 100 ms. This leads to an effective transmission rate of $1000\,\mathrm{B\,s^{-1}}$. The energy used for transmission per day for 5 dBm

$$E_{t5} = 4 \cdot \left(3.53\,\mathrm{mA} \cdot 6 \cdot 6\,\mathrm{B} \cdot 3200\,\mathrm{Hz} \cdot 0.5\,\mathrm{s}/1000\,\mathrm{B\,s^{-1}}\right) = 813.3\,\mathrm{mA\,s}, \tag{11.3}$$

respectively for 14 dBm

$$E_{t14} = 4 \cdot \left(6.54\,\mathrm{mA} \cdot 6 \cdot 6\,\mathrm{B} \cdot 3200\,\mathrm{Hz} \cdot 0.5\,\mathrm{s}/1000\,\mathrm{B\,s^{-1}}\right) = 1507\,\mathrm{mA\,s}. \tag{11.4}$$

According to Fig. 11.4a the average current consumption for advertising with a transmission power of 5 dBm is about $3.91\,\mathrm{V}/500\,\Omega \cdot 12 \cdot 2\,\mathrm{ms}/4\,\mathrm{s} = 46.9\,\mathrm{\mu A}$, while Fig. 11.5a estimates a current consumption of $1.23\,\mathrm{V}/100\,\Omega \cdot 12 \cdot 2\,\mathrm{ms}/4\,\mathrm{s} = 73.8\,\mathrm{\mu A}$ for advertising with 14 dBm. For advertising 20 B user data are estimated. The time left per day for advertising is calculated by

$24 \cdot 60 \cdot 60\,\mathrm{s} - 4 \cdot 6 \cdot 6\,\mathrm{B} \cdot 3200\,\mathrm{Hz} \cdot 0.5\,\mathrm{s}/1000\,\mathrm{B\,s^{-1}} - 24 \cdot (0.5\,\mathrm{s} + 0.44\,\mathrm{s}) = 86\,147\,\mathrm{s}$. This leads to an energy per day for advertising with 5 dBm

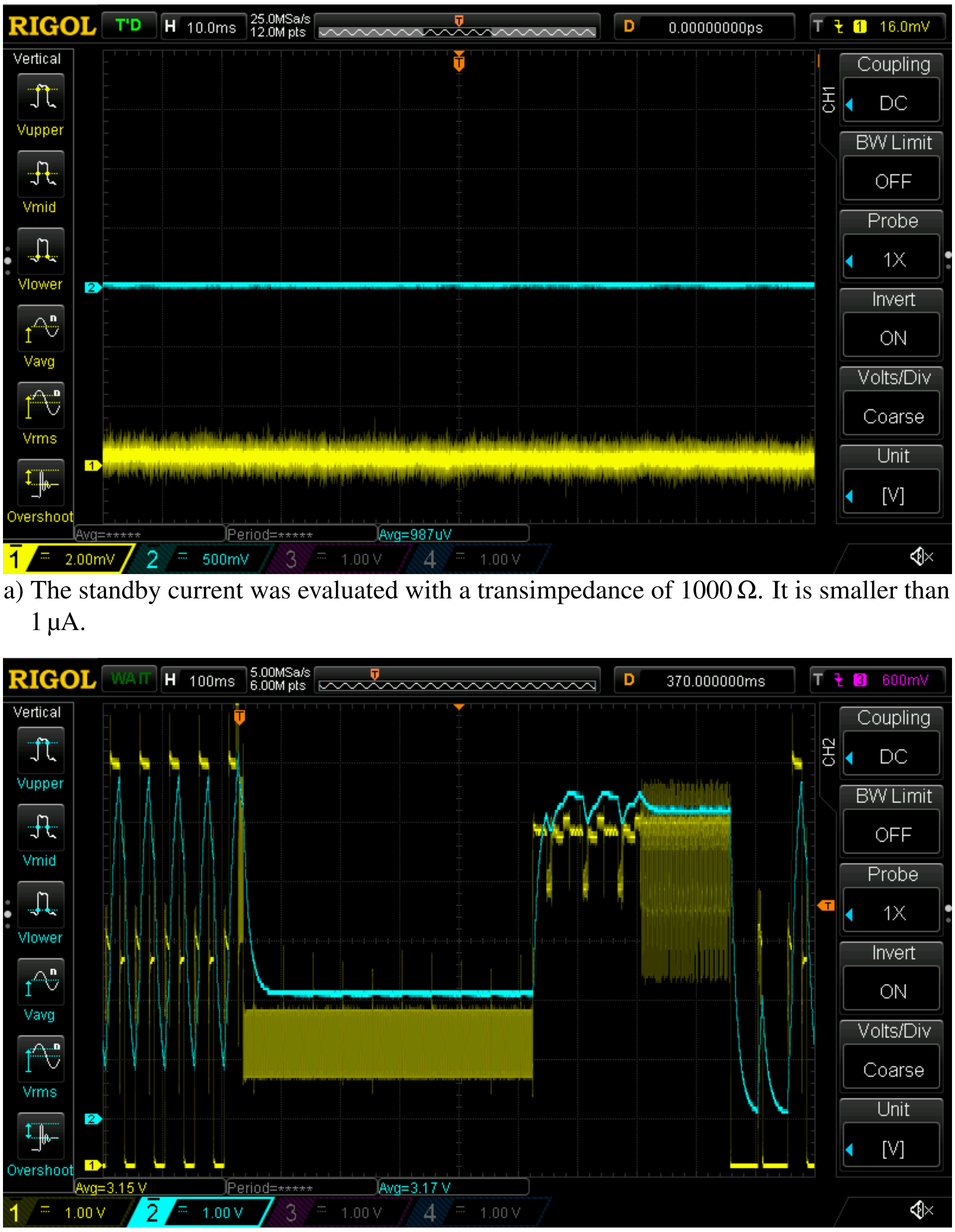
$$E_{adv5} = 46.9\,\mu\mathrm{A} \cdot 86\,147\,\mathrm{s} = 4040\,\mathrm{mA\,s}, \tag{11.5}$$

respectively for 14 dBm

$$E_{adv14} = 73.8\,\mu\mathrm{A} \cdot 86\,147\,\mathrm{s} = 6358\,\mathrm{mA\,s}. \tag{11.6}$$

This yields a total energy of $4853\,\mathrm{mA\,s} \widehat{=} 1.348\,\mathrm{mA\,h}$ per day for 5 dBm and $7865\,\mathrm{mA\,s} \widehat{=} 2.185\,\mathrm{mA\,h}$ for 14 dBm. With a 950 mA h coin cell this leads to a battery life of 705 days for 5 dBm respectively 435 days for 14 dBm.

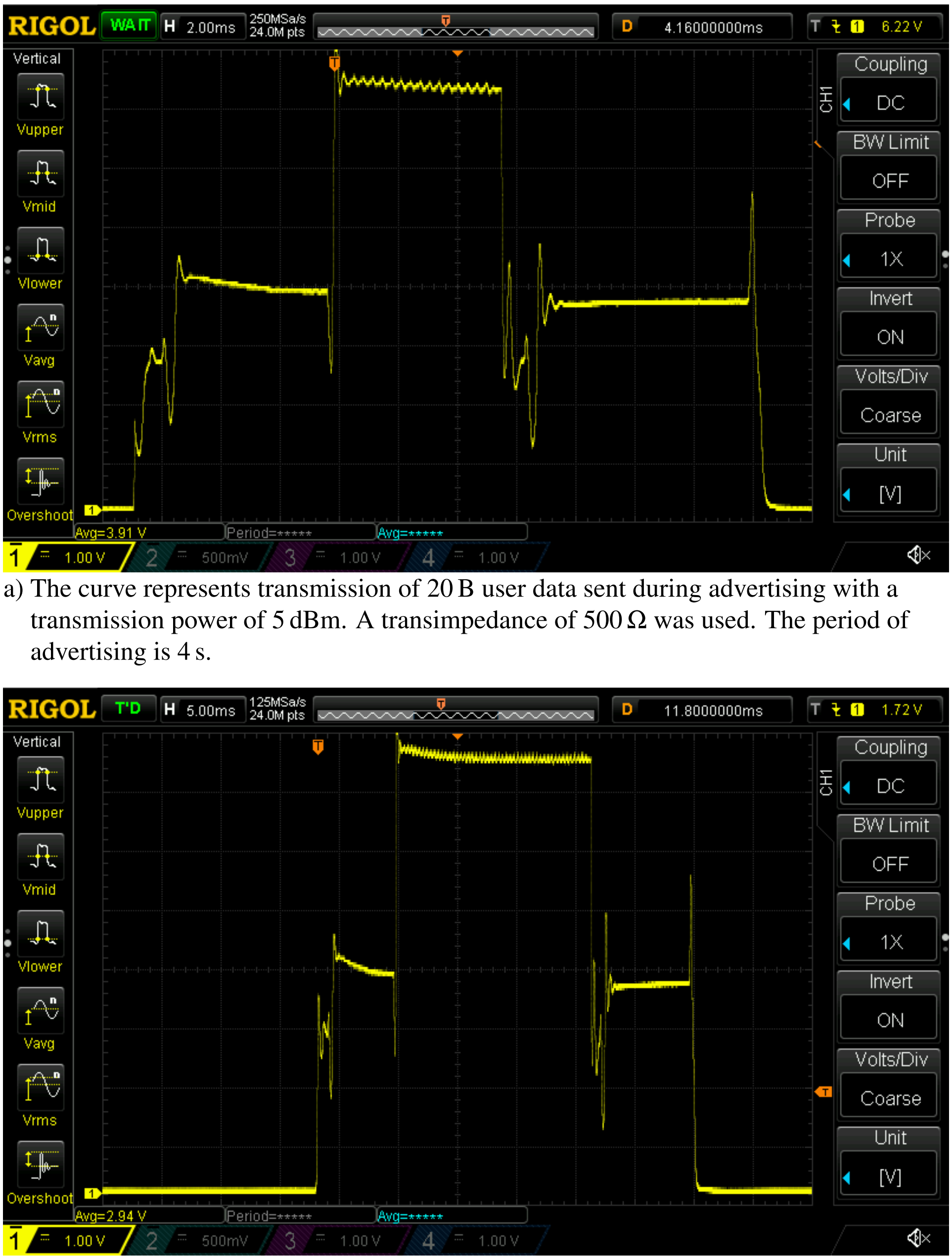
The energy used for advertising can be identified to be the largest influence on life of battery.



a) The standby current was evaluated with a transimpedance of 1000 Ω. It is smaller than 1 μA.

b) The current consumption during sampling and saving was acquired with a transimpedance of 500 Ω. Sampling can be observed from −380 ms to 120 ms, while erasing and writing of the external flash is happening from 120 ms to 460 ms

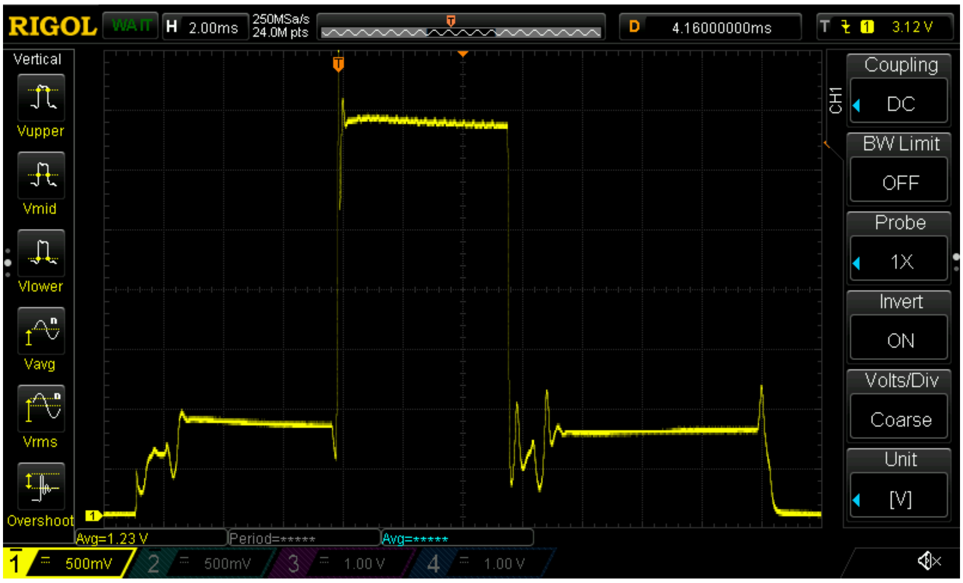
Fig. 11.3. Evaluation of the current consumption in standby as well as in a measuring cycle.



a) The curve represents transmission of 20 B user data sent during advertising with a transmission power of 5 dBm. A transimpedance of 500 Ω was used. The period of advertising is 4 s.

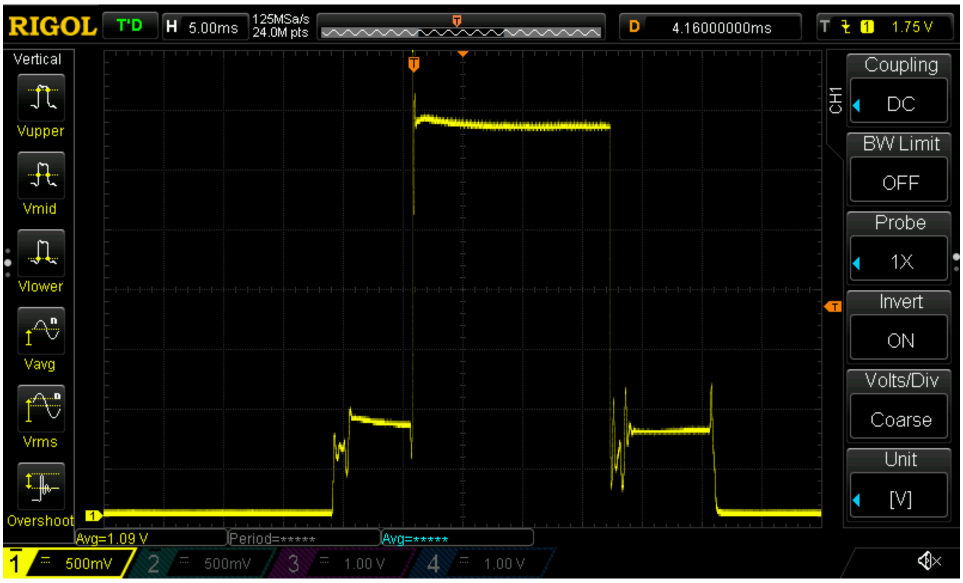
b) The curve represents transmission of 100 B user data with a transmission power of 5 dBm. A transimpedance of 500 Ω was used. The period of transmitting a 100 B block is 100 ms.

Fig. 11.4. Evaluation of the current consumption with a transmission power of 5 dBm.



H 2.00ms
250MSa/s
24.0M pts
D 4.16000000ms
T 1 3.12 V
Coupling DC
BW Limit OFF
Probe 1X
Invert ON
Volts/Div Coarse
Unit [V]
Avg=1.23 V
Period=*****
Avg=*****
1 500mV
2 500mV
3 1.00 V
4 1.00 V

a) The curve represents transmission of 20 B user data sent during advertising with a transmission power of 14 dBm. A transimpedance of 100 Ω was used. The period of advertising is 4 s.



H 5.00ms
125MSa/s
24.0M pts
D 4.16000000ms
T 1 1.75 V
Coupling DC
BW Limit OFF
Probe 1X
Invert ON
Volts/Div Coarse
Unit [V]
Avg=1.09 V
Period=*****
Avg=*****
1 500mV
2 500mV
3 1.00 V
4 1.00 V

b) The curve represents transmission of 100 B user data with a transmission power of 14 dBm. A transimpedance of 100 Ω was used. The period of transmitting a 100 B block is 100 ms.

Fig. 11.5. Evaluation of the current consumption with a transmission power of 14 dBm.

12 Conclusion

This thesis presents the conception, prototype implementation and testing of a condition monitoring system that is using sub-GHz frequency bands.

The Friis transmission equation is introduced to discuss the benefits of sub-GHz frequency bands. Together with antenna gain factors it considers path loss in vacuum and yields that the attenuation is decreasing for increasing wave-lengths. This enables equal transmission ranges while consuming less power compared to 2.4 GHz technologies as well as avoiding interference with Bluetooth, Wi-Fi, mobile phones and microwaves.

Especially in industrial environments, where a line of sight is not given in many cases, the wider Fresnel zone compared to 2.4 GHz allows to surround obstacles.

Wireless M-Bus was selected as network protocol due to standardization. It allows further power savings by being optimized for low-power meters. The sensor setup, consisting of a Texas Instruments CC1352P microcontroller and a Kionix KX222-1054 accelerometer, allows conducting some less expensive computations for data preparation, e. g. downsampling. The RevolutionPi IPC in contrast allows more computationally expensive computations.

The firmware is written in C++ using abstract classes in many cases; this simplifies the implementation of e. g. other types of sensors or arbitrary computations that have to be conducted after sampling. This leaded to very flexible code that can be easily adapted for other use cases.

The range was evaluated using omnidirectional aerials. Ranges of over 400 m at a current consumption of 14 mA during actual transmission were achieved. Packet loss ratios indicate that a forward error correction mechanism could improve the system's performance. The trend of the RSSI values shows that the correct alignment of aerials is very important.

The current consumption of the smart sensor was evaluated for all operating states. An estimation for a typical measurement yields that the smart sensor can be powered for approximately two years using only a single coin-cell battery. For the evaluated setup, the energy consumed by advertising has the largest influence on the life of battery.

The developed prototype system provides a suitable basis for a new commercial condition monitoring system offering low-power paired with a very good transmission range. However, usage in commercial systems requires the implementation of a bidirectional communication. The channel from the other device to the meter is necessary to ensure a reliable data exchange and/or remote configuration. In addition, a more general driver implementation that allows simple integration in edge software would be preferred. It can be based on the software presented in this thesis.

Part IV

Appendix

A Tables

A.1 Advanced Encryption Standard (AES) S-Box

0x00–0x07	0x63	0x7C	0x77	0x7B	0xF2	0x6B	0x6F	0xC5
0x10–0x17	0xCA	0x82	0xC9	0x7D	0xFA	0x59	0x47	0xF0
0x20–0x27	0xB7	0xFD	0x93	0x26	0x36	0x3F	0xF7	0xCC
0x30–0x37	0x04	0xC7	0x23	0xC3	0x18	0x96	0x05	0x9A
0x40–0x47	0x09	0x83	0x2C	0x1A	0x1B	0x6E	0x5A	0xA0
0x50–0x57	0x53	0xD1	0x00	0xED	0x20	0xFC	0xB1	0x5B
0x60–0x67	0xD0	0xEF	0xAA	0xFB	0x43	0x4D	0x33	0x85
0x70–0x77	0x51	0xA3	0x40	0x8F	0x92	0x9D	0x38	0xF5
0x80–0x87	0xCD	0x0C	0x13	0xEC	0x5F	0x97	0x44	0x17
0x90–0x97	0x60	0x81	0x4F	0xDC	0x22	0x2A	0x90	0x88
0xA0–0xA7	0xE0	0x32	0x3A	0x0A	0x49	0x06	0x24	0x5C
0xB0–0xB7	0xE7	0xC8	0x37	0x6D	0x8D	0xD5	0x4E	0xA9
0xC0–0xC7	0xBA	0x78	0x25	0x2E	0x1C	0xA6	0xB4	0xC6
0xD0–0xD7	0x70	0x3E	0xB5	0x66	0x48	0x03	0xF6	0x0E
0xE0–0xE7	0xE1	0xF8	0x98	0x11	0x69	0xD9	0x8E	0x94
0xF0–0xF7	0x8C	0xA1	0x89	0x0D	0xBF	0xE6	0x42	0x68
0x08–0x0F	0x30	0x01	0x67	0x2B	0xFE	0xD7	0xAB	0x76
0x18–0x1F	0xAD	0xD4	0xA2	0xAF	0x9C	0xA4	0x72	0xC0
0x28–0x2F	0x34	0xA5	0xE5	0xF1	0x71	0xD8	0x31	0x15
0x38–0x3F	0x07	0x12	0x80	0xE2	0xEB	0x27	0xB2	0x75
0x48–0x4F	0x52	0x3B	0xD6	0xB3	0x29	0xE3	0x2F	0x84
0x58–0x5F	0x6A	0xCB	0xBE	0x39	0x4A	0x4C	0x58	0xCF
0x68–0x6F	0x45	0xF9	0x02	0x7F	0x50	0x3C	0x9F	0xA8
0x78–0x7F	0xBC	0xB6	0xDA	0x21	0x10	0xFF	0xF3	0xD2
0x88–0x8F	0xC4	0xA7	0x7E	0x3D	0x64	0x5D	0x19	0x73
0x98–0x9F	0x46	0xEE	0xB8	0x14	0xDE	0x5E	0x0B	0xDB
0xA8–0xAF	0xC2	0xD3	0xAC	0x62	0x91	0x95	0xE4	0x79
0xB8–0xBF	0x6C	0x56	0xF4	0xEA	0x65	0x7A	0xAE	0x08
0xC8–0xCF	0xE8	0xDD	0x74	0x1F	0x4B	0xBD	0x8B	0x8A
0xD8–0xDF	0x61	0x35	0x57	0xB9	0x86	0xC1	0x1D	0x9E
0xE8–0xEF	0x9B	0x1E	0x87	0xE9	0xCE	0x55	0x28	0xDF
0xF8–0xFF	0x41	0x99	0x2D	0x0F	0xB0	0x54	0xBB	0x16

Tab. A.1. Advanced Encryption Standard (AES) S-Box used in `SubBytes`. Compare table to [29, pp. 139–140].

A.2 3-of-6 Constant Weight Code

Unencoded	Encoded
0b0000	0b010110
0b0001	0b001101
0b0010	0b001110
0b0011	0b001011
0b0100	0b011100
0b0101	0b011001
0b0110	0b011010
0b0111	0b010011
0b1000	0b101100
0b1001	0b100101
0b1010	0b100110
0b1011	0b100011
0b1100	0b110100
0b1101	0b110001
0b1110	0b110010
0b1111	0b101001

Tab. A.2. Conversion table for encoding or decoding data with 3-of-6 constant-weight code. Compare table to [22, p. 23].

B Implementations

B.1 Smart Pointer

```
#ifndef AUXILIARY_SMARTPTR_H_
#define AUXILIARY_SMARTPTR_H_

#include <stdint.h>

class RefCount
{
private:
    uint8_t count;
public:
    void addRef();
    uint8_t releaseRef();

    RefCount();
    ~RefCount();

};

template <typename T>
class SmartPtr
{
private:
    T* ptr;
    RefCount* refCount;

public:
    SmartPtr()
        : ptr(0), refCount(0)
    {
        refCount = new RefCount();
        refCount->addRef(); // only assignment is possible on default constructed object; assignment releases ref --> size should be 0 now, therefore it has to be 1 at init
    }
    SmartPtr(T* ptr)
        : ptr(ptr), refCount(0)
    {
        refCount = new RefCount();
        refCount->addRef();
    }
    SmartPtr(const SmartPtr<T>& sp)
        : ptr(sp.ptr), refCount(sp.refCount)
    {
        refCount->addRef();
    }

    T& operator*()  const {return *ptr;}
    T* operator->() const {return ptr;}

    SmartPtr<T>& operator= (const SmartPtr<T>& sp)
    {
        if(this != &sp){
            if(!refCount->releaseRef()){
                if(ptr) delete ptr;
                delete refCount;
            }

            ptr = sp.ptr;
            refCount = sp.refCount;
            refCount->addRef();
        }
        return *this;
    }

    T& operator[](uint8_t idx) const {return ptr[idx];}

    void reset(){
        if(!refCount->releaseRef()){
            if(ptr) delete ptr;
            delete refCount;
        }

        ptr = 0;
        refCount = new RefCount();
        refCount->addRef();

    }

    T* get() const {return ptr;}

    ~SmartPtr()
    {
        if(!refCount->releaseRef()){
            if(ptr) delete ptr;
            delete refCount;
        }
    }
};

#endif /* AUXILIARY_SMARTPTR_H_ */
```

Listing B.1. `SmartPtr.h`: Compare implementation to [25].

```
#include "Auxiliary/SmartPtr.h"

RefCount::RefCount()
    : count(0)
{

}
RefCount::~RefCount(){

}

void RefCount::addRef(){
    count++;
}
uint8_t RefCount::releaseRef(){
    return --count;
}
```

Listing B.2. `SmartPtr.cpp`: Compare implementation to [25].

B.2 Doubly Linked List

```
#ifndef MBUS_DEQUE_H_
#define MBUS_DEQUE_H_

#include <stdint.h>
#include "DequeElement.h"

template <class X>
class Deque{
    DequeElement* first = 0;
private:
    Deque(X l){
        if(l == 0) return;
        DequeElement* f = l;
        while(f->m_prev != reinterpret_cast<DequeElement*>(0)
                && f->m_prev !=
                  reinterpret_cast<DequeElement*>(0xFFFFFFFF))
            f = f->m_prev;
        first = f;
        return;
    }

public:
    static void postEventToList(X l, const unsigned int& event){
        const Deque tmp(l);
        tmp.postEvent(event);
    }

    uint8_t count() const {
        if(!first) return 0;

        uint8_t ct = 1;
        DequeElement* next = first;
        while(next->m_next){
            ++ct;
            next = next->m_next;
        }
        return ct;
    }

    X operator[](uint8_t idx) const {
        if(!first) return 0;
        DequeElement* next = first;
        while(idx--){
            next = next->m_next;
            if(!next) return 0;
        }
        return static_cast<X>(next);
    }

    X end() const {
        if(!first) return 0;
        DequeElement* next = first;
        while(next->m_next){
            next = next->m_next;
        }
        return static_cast<X>(next);
    }

    X begin(){
        if(!first) return static_cast<X>(0);
        return static_cast<X>(first);
    }

    bool push_back(X el){
        //updateList();
        if(el->m_next != reinterpret_cast<DequeElement*>(0xFFFFFFFF))
```

```
            return false; //if it is already in a list -> return false

        el->m_next = 0; // is in list, but last element
        el->m_prev = end();
        if(first == 0){
            first = el;
        }
        else{
            end()->m_next = el;
        }
        return true;
    }
    X pop_back(){
        //updateList();
        uint8_t ct = count();
        X ret = static_cast<X>(0);
        if(!ct) return 0;
        if(ct == 1){
            ret = reinterpret_cast<X>(first);
            first->m_next =
              reinterpret_cast<DequeElement*>(0xFFFFFFFF);
            first->m_prev =
              reinterpret_cast<DequeElement*>(0xFFFFFFFF);
            first = 0;
        }
        else{
            DequeElement* le = (*this)[ct-2];
            ret = reinterpret_cast<X>(le->m_next);
            le->m_next->m_next =
              reinterpret_cast<DequeElement*>(0xFFFFFFFF);
            le->m_next->m_prev =
              reinterpret_cast<DequeElement*>(0xFFFFFFFF);
            le->m_next = 0;
        }
        return ret;
    }
    bool push_front(X el){
        //updateList();
        if(el->m_prev != reinterpret_cast<DequeElement*>(0xFFFFFFFF))
            return false; //if it is already in a list -> return false

        el->m_prev = 0; // is in list, but first element
        el->m_next = begin();
        if(first != 0) begin()->m_prev = el;
        first = el;
        return true;
    }
    X pop_front(){
        //updateList();
        X ret = static_cast<X>(0);
        if(!first) return 0;
        if(count() == 1){
            ret = reinterpret_cast<X>(first);
            first->m_next =
              reinterpret_cast<DequeElement*>(0xFFFFFFFF);
            first->m_prev =
              reinterpret_cast<DequeElement*>(0xFFFFFFFF);
            first = 0;
        }
        else{
            DequeElement* le = begin();
            ret = reinterpret_cast<X>(le);
            le->m_next->m_prev = reinterpret_cast<DequeElement*>(0);
            first = le->m_next;
            le->m_next = reinterpret_cast<DequeElement*>(0xFFFFFFFF);
            le->m_prev = reinterpret_cast<DequeElement*>(0xFFFFFFFF);
        }
        return ret;
    }
    void postEvent(const unsigned int& e, void* ptr) const{
        uint8_t ct = count();
        while(ct--){
            DequeElement* l = (*this)[ct];
            l->eventFilter(e, ptr);
        }
    }
    void postEvent(const unsigned int& e) const{
        postEvent(e, NULL);
    }

    Deque(){}

    ~Deque(){}
};

#endif /* MBUS_DEQUE_H_ */
```

Listing B.3. `Deque.h`

```
#ifndef DequeElement_H_
#define DequeElement_H_


class DequeElement{
private:
    DequeElement* m_prev = reinterpret_cast<DequeElement*>(0xFFFFFFFF)
   ;
```

```
    DequeElement* m_next = reinterpret_cast<DequeElement*>(0xFFFFFFFF)
   ;
    template <class X>
    friend class Deque;

    virtual void eventFilter(const unsigned int& e, void* ptr);

public:
    DequeElement* getNext() const {return m_next;}
    DequeElement* getPrev() const {return m_prev;}
    DequeElement* getFirst() const;
    DequeElement* getLast() const;
};

#endif /* DequeElement_H_ */
```

Listing B.4. `DequeElement.h`

```
#include "Auxiliary/DequeElement.h"

void DequeElement::eventFilter(const unsigned int& e, void* ptr){

}

DequeElement* DequeElement::getFirst() const{
    DequeElement* l = const_cast<DequeElement*>(this);
    while(l->getPrev()){
        l = l->getPrev();
    }
    return l;
}
DequeElement* DequeElement::getLast() const{
    DequeElement* l = const_cast<DequeElement*>(this);
    while(l->getNext()){
        l = l->getNext();
    }
    return l;
}
```

Listing B.5. `DequeElement.cpp`

Bibliography

[1] *[W-MBUS] How is the RSSI byte value transformed into an attenuation/signal strength value?* Radiocrafts AS. URL: https://radiocrafts.com/kb/w-mbus-rssi-byte-value-transformed-attenuation-signal-strength-value/ (visited on 2019-10-01).

[2] Günther Blaschek. *Object-oriented programming. With prototypes / Günther Blaschek.* Berlin: Springer-Verlag, 1994. ISBN: 3-540-56469-1.

[3] Johannes Buchmann. *Einführung in die Kryptographie*. 4th ed. Springer-Lehrbuch. Berlin and Heidelberg: Springer, 2008. ISBN: 978-3-540-74451-1.

[4] *CC1352P SimpleLink^{TM} High-Performance Dual-Band Wireless MCU With Integrated Power Amplifier*. Version July. Texas Instruments Inc., 2019.

[5] *CC13x2, CC26x2 SimpleLink^{TM} Wireless MCU Technical Reference Manual*. Version May. Texas Instruments Inc., 2019.

[6] Michael Jesse Chonoles and James A. Schardt. *UML 2 for dummies*. –For dummies. New York: [Great Britain] : Wiley, 2003. ISBN: 9780764526145.

[7] Europäische Union. *2011/829/EU: Durchführungsbeschluss der Kommission vom 8. Dezember 2011 zur Änderung der Entscheidung 2006/771/EG zur Harmonisierung der Frequenznutzung durch Geräte mit geringer Reichweite*. 2011-12-13.

[8] Saleh Faruque. *Radio frequency modulation made easy*. SpringerBriefs in electrical and computer engineering, 2191-8120. Switzerland: Springer, 2017. ISBN: 978-3-319-41200-9.

[9] Louis E. Frenzel. *Handbook of serial communications interfaces. A comprehensive compendium of serial digital input/output (I/O) standards*. Amsterdam: Newnes, 2015. ISBN: 9780128006290.

[10] H. T. Friis. "A Note on a Simple Transmission Formula". In: *Proceedings of the IRE* 34.5 (1946), pp. 254–256. ISSN: 0096-8390. DOI: 10.1109/JRPROC.1946.234568.

[11] Erich Gamma et al. *Design patterns. Elements of reusable object-oriented software*. Addison-Wesley professional computing series. Reading, Mass. and Wokingham: Addison-Wesley, 1995. ISBN: 0201633612.

[12] Jerald G. Graeme. *Photodiode amplifiers. Op amp solutions / Jerald G. Graeme*. New York and London: McGraw-Hill, 1996. ISBN: 0-07-024247-x.

[13] Paul Horowitz and Winfield Hill. *The Art of Electronics*. 2nd ed. Cambridge: Cambridge University Press, 1989. ISBN: 0521370957.

[14] Internet Engineering Task Force, ed. *IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs)*. Informational. Version August. 2007. URL: `https://tools.ietf.org/pdf/rfc4919.pdf` (visited on 2019-09-07).

[15] Internet Engineering Task Force, ed. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. Proposed Standard. Version September. 2007. URL: `https://tools.ietf.org/pdf/rfc4944.pdf` (visited on 2019-09-07).

[16] Charles M. Kozierok. *The TCPIP guide. A comprehensive, illustrated Internet protocols reference*. San Francisco: No Starch Press, 2005. lxxiv, 1539. ISBN: 9781593270476.

[17] *LoRaWAN^{TM} What is it? A technical overview of LoRa® and LoRaWAN^{TM}*. Technical Marketing Workgroup 1.0. LoRa Alliance, November 2015. URL: `https://lora-alliance.org/sites/default/files/2018-04/what-is-lorawan.pdf` (visited on 2019-09-07).

[18] *LTC2057/LTC2057HV. High Voltage, Low Noise Zero-Drift Operational Amplifier*. Datasheet. Linear Technology Corp., 2013.

[19] Todd K. Moon. *Error correction coding. Mathematical methods and algorithms*. Hoboken, N.J.: Wiley Interscience, 2005. ISBN: 9780471648000.

[20] Tammy Noergaard. *Embedded systems architecture. A comprehensive guide for engineers and programmers*. Second edition. Amsterdam: Elsevier/Newnes, 2013. xv, 653 pages. ISBN: 978-0-12-382196-6.

[21] Donald Norris. *Programming with STM32. Getting started with the Nucleo Board and CC++*. Norris, Donald (author.) New York: McGraw-Hill Education, 2018. xiii, 290 pages. ISBN: 978-1-260-03131-7.

[22] ÖNORM, ed. *Communication systems for meters*. Draft Standard. Vienna: Austrian Standards Institute, 2017-12-01.

[23] *Op Amp Input Bias Current*. MT-038 Tutorial. Analog Devices Inc., October 2008.

[24] *Range Calculation for 300MHz to 1000MHz Communication Systems*. APPLICATION NOTE. Atmel Corporation, July 2015.

[25] Madhu Raykar. *Implementing a simple smart pointer in C++*. URL: `https://www.codeproject.com/Articles/15351/Implementing-a-simple-smart-pointer-in-c` (visited on 2019-08-28).

[26] *RevPi Connect*. Kunbus GmbH. URL: `https://revolution.kunbus.de/shop/de/revpi-connect` (visited on 2019-09-09).

[27] Miro Samek and Robert Ward. “Build a Super Simple Tasker”. In: *Embedded Systems Design* July (2006).

[28] Sergei Alexander Schelkunoff and Harald Trap Friis. *Antennas. Theory and Practice*. New York: Wiley, 1952.

[29] Klaus Schmeh. *Kryptografie. Verfahren, Protokolle, Infrastrukturen*. Heidelberg: dpunkt, 2016. 900 Seiten in 1 Teil. ISBN: 978-3-86490-356-4.

[30] *SPIRIT1. Low data rate, low power sub-1GHz transceiver*. Datasheet - production data. Version October. STMicroelectronics N.V., 2016.

[31] *STM32WB55xx. Multiprotocol wireless 32-bit MCU Arm®-based Cortex®-M4 with FPU, Bluetooth® 5 and 802.15.4 radio solution*. STM32WB55xx. Version February. STMicroelectronics N.V., 2019.

[32] *Telit Wireless M-Bus 2013 Part 4 User Guide*. Version Rev.14 – 2016-01-11. Telit Wireless Solutions Inc., 2013.

[33] Thread Group, ed. *Thread Specification*. Version 1.1.1. 2017-02-13.

[34] U. Tietze and Ch. Schenk. *Halbleiter-Schaltungstechnik*. 11., völlig neu bearb. und erw. Aufl. Berlin: Springer, 1999. XXVI, 1421 Seiten. ISBN: 3-540-64192-0.

[35] *TPS782 500-nA IQ,150-mA, Ultra-Low Quiescent Current Low-Dropout Linear Regulator*. Datasheet. Texas Instruments Inc., January 2015.

[36] Wikipedia, ed. *Cipher Block Chaining Mode*. URL: `https://de.wikipedia.org/w/index.php?oldid=188262852` (visited on 2019-08-13).

[37] Wikipedia, ed. *Frequency-shift keying*. 14.08.2019. URL: `https://en.wikipedia.org/w/index.php?oldid=899829355` (visited on 2019-08-30).

[38] Wikipedia, ed. *Fresnel zone*. 26.09.2019. URL: `https://en.wikipedia.org/w/index.php?oldid=914219634` (visited on 2019-10-01).

[39] Wikipedia, ed. *Long Range Wide Area Network*. 29.08.2019. URL: `https://de.wikipedia.org/w/index.php?oldid=191802014` (visited on 2019-09-07).

[40] Anthony Williams. *C++ concurrency in action. Practical multithreading / Anthony Williams*. Shelter Island, N.Y.: Manning, 2012. ISBN: 9781933988771.

[41] Jürgen Wolf. *C++ von A bis Z. Das umfassende Handbuch*. 2. aktualisierte Aufl., 2. Nachdr. Galileo Computing. Bonn: Galileo Press, 2011. 1247 Seiten. ISBN: 978-3-8362-1429-2.