FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Evaluation of Operating Systems for Embedded Low-Power Wireless Systems

Contiki, Riot OS, Zephyr

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Technische Informatik

eingereicht von

Theodor Mittermair
Matrikelnummer 1426389

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Wolfgang Kastner, Dipl.-Ing. Dr.techn.
Mitwirkung: Stefan Seifried, Dipl.-Ing., BSc.
Philipp Raich, Dipl.-Ing., BSc.

Wien, 14. März 2019

Theodor Mittermair	Wolfgang Kastner

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Evaluation of Operating Systems for Embedded Low-Power Wireless Systems

Contiki, Riot OS, Zephyr

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Engineering

by

Theodor Mittermair
Registration Number 1426389

to the Faculty of Informatics

at the TU Wien

Advisor: Wolfgang Kastner, Dipl.-Ing. Dr.techn.
Assistance: Stefan Seifried, Dipl.-Ing., BSc.
Philipp Raich, Dipl.-Ing., BSc.

Vienna, 14th March, 2019

Theodor Mittermair | Wolfgang Kastner

Erklärung zur Verfassung der Arbeit

Theodor Mittermair
Scheibenweg 2, 2211 Pillichsdorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 14. März 2019

Theodor Mittermair

Danksagung

Zuallererst möchte ich meiner Familie danken, die meine absurden Essens- & Schlafenszeiten ertragen hat.

Ein Danke gilt auch meinem ersten Betreuer, Stefan Seifried, welcher mir diese zwar sehr umfangreiche, aber auch lehrreiche Arbeit übergeben hat, sowie meinem zweiten Betreuer, Philipp Raich, der mich trotz großer Verzögerungen motiviert hat nicht aufzugeben.

Desweiteren möchte ich mich bei meinem Studienkollegen David Kaufmann für Unterstützung bei der Fehlersuche im Kontext von IPv6 bedanken.

Und zum Schluss ein Danke an die Hersteller jenes koffeinhaltigen Getränks in dem ich viele meiner Sorgen dieser Arbeit ertrunken habe.

Acknowledgements

To begin with, a very large thank you to my family, which had to endure my absurd food and bedtimes.

Additional thanks go to my first supervisor, Stefan Seifried, who gave me this comprehensive but also instructive work, as well as to my second supervisor, Philipp Raich, who kept motivating me even though my work was often delayed.

Furthermore i'd like to thank my colleague David Kaufmann, who supported me with his knowledge on IPv6 when debugging.

And finally, thanks to the manufacturer of that caffeinated drink I drowned so many sorrows in.

Kurzfassung

Diese Arbeit evaluiert die Funktionalität der drei Betriebssysteme **Zephyr** [urld], **Riot OS** [urlc] und **Contiki** [urlb] durch exemplarische Beispielanwendungen orientiert an IoT-Geräten. Dazu zählen Textausgabe an eine serielle Konsole, Umschalten von GPIOs sowie Senden von Nachrichten über ein Wireless-Netzwerk. Der Fokus dieser Arbeit liegt mehr auf allgemeinen Aussagen über gewählte Kriterien als Aussagen über spezifische Anwendungen. Es soll keine Antwort auf die Frage „Was ist das Beste?“ gefunden werden (was im Allgemeinen auch nicht möglich ist, da die Realität solch eine allgemeine Aussage nicht zulässt), vielmehr soll es Hinweise für die Richtung tiefergehender Untersuchungen hinsichtlich speziellerer Anwendungen geben.

Abstract

This paper evaluates the three operating systems **Zephyr** [urld], **Riot OS** [urlc], **Contiki** [urlb] by examining various functionality using sample applications with IoT in mind. This includes external interfacing with a serial console, toggling GPIOs and wireless networking. The evaluation is focused more around general than specific use cases and attempts to make general statements on evaluation criteria. This paper does not and is not intended to find a definite answer to the question „Which is the best?“ (which in general is impossible to answer, since reality always has trade-offs), instead it should provide hints for the direction of further, more in-depth, application and use case specific research.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Home-automation sensors, security cameras, garage doors, watches, fridges, washing machines, rice cookers, toasters; in some regard they have more in common than it might seem at first glance. All of these embedded systems have representatives in the vastly growing field of Internet of Things (IoT), are comprised of a rising number of hardware and software components, growing in complexity, and are often unconsciously encountered in each of our daily lives.

In the area of hardware and software, it is not unusual for technology to change very rapidly. While microcontroller hardware is improved and equipped with additional features (such as built-in wireless communication peripherals), microcontroller operating systems are continuously improved and adapted as well (e.g., hardware support, new software features). As a result, the natural question arises whether or not a combination of hardware and software, forming an embedded system, is the correct choice for an application.

Motivated by the constant desire for technological improvement as well as continuous development, this work aims to (re-)evaluate and compare similar software projects used for IoT products regarding their features and performance.

Continuous development on software projects often leads to legacy code, hindering further development, reducing usability or even performance. However, software projects which are alive for an extended period may profit from experience that the developers have made with it (e.g., the reason being the way features are implemented). In comparison to that, a younger software project might have less legacy code and might learn from other projects failures/success and improve upon it, but also allows to repeat old or introduce new mistakes. It also fans out possible choices, on the one hand, a positive argument for constructive competition, on the other hand, splitting resources (i.e., time

and effort of developers) for improvement between existing software projects. Due to this ever-changing nature of technology, it is necessary to reevaluate existing solutions, as well as reviewing new and upcoming ones.

Therefore this work will consider the operating systems **Riot OS** [urlc] (developed since 2008') and **Contiki** [urlb] (first release in 2003) as well-known, and **Zephyr** [urld] as a relatively new software project (public presentation 2016), with a focus on IoT relevant performance metrics. Being motivated by IoT applications, metrics included are networking performance, power consumption and resource usage (i.e., Random Access Memory (RAM) / Read Only Memory (ROM) / Central Processing Unit (CPU)). Other works [WSS09][TA09] also evaluate IPC (inter process communication) and similar metrics (e.g., performance of semaphores, mailboxes, messages, events), which is considered out-of-scope for this work.

That being said, generally, it is hard or even impossible to provide a definitive answer to what is best since there are too many variables and there is most likely no one-fits-all solution. Therefore, it makes sense to give the results of multiple, generalized comparisons which can be mapped to different use cases, serving as a base for further research and product design decisions.

1.2 Evaluation Criteria

The operating systems are to be evaluated in regard to the following criteria:

- Functional Criteria
  - Networking
    - Delay / Latency
    - Throughput
    - Reliability
  - Resources
    - Memory Usage
    - CPU Usage
    - Power Consumption
  - Security
    - Memory Protection
- Non Functional Criteria
  - Support for Concurrency
  - Modularisation
  - Hardware Support

Networking has different aspects of different importance defined by the application of an embedded system. For example, to an application including a sensor reading the room temperature it might be relatively irrelevant how long exactly it takes to transmit a temperature value to a central station, as long as it reaches its destination. In that case reliability is of greater importance than throughput (low overall data volume) or delay (data use is often deferred, e.g., room temperature control). Another example is the transmission of image data from a live camera, where high throughput and small delay is desired to transmit the frames in (close to real-)time, whereas reliability is non-critical (e.g., the loss of some frames).

The *resource usage* of an operating system is an important factor in the design of an embedded system. Lower resource usage enables cheaper designs and longer battery life. However, in reality, trade-offs have to be made between (program-)space, (run-)time and cost for the design of an embedded system. For that reason, the *power usage*, *RAM and ROM requirements* as well as *CPU usage* are considered in this work.

This work also includes evaluation on GPIO capabilities. One of the reasons to use an operating system in conjunction with an embedded system is the abstraction of hardware provided. Therefore, interaction with GPIOs usually is performed via the functionality provided by the operating system. Some applications make use of Pulse Width Modulation (PWM) or want to use communication protocols such as Universal (Serial) Asynchronous Receiver Transmitter (U(S)ART), Inter-Integrated Circuit (I2C), Serial Peripheral Interface (SPI). If no dedicated hardware is available, such functionality can be implemented using digital GPIOs for emulation. Another aspect is the interaction with external components (i.e., sensors, actuators), which often makes use of asynchronous notification signals (i.e., external interrupts). An application designer and/or application software developer is likely to want to receive such information promptly. Depending on the implementation, the performance can vary, implying some interesting metrics. For aforementioned reasons, this work introduces the following metrics:

- *maximum toggling frequency*, serving as an indicator for how fast a software generated PWM could be
- *minimum on-/off-times*, serving as an indicator for the possible speed a software emulated protocol could reach considering a protocol's timing constraints.
- *input to output delay*, serving as an indicator for response time to asynchronous external signals.

Specific applications (especially those working with untrusted user input) profit from additional *security* features, including but not limited to, public accessible devices or devices reachable through networking. Apart from deliberate attacks on embedded systems, such features can also protect against unintentional faults such as mistakes made during design and/or implementation of an embedded system. A possible preventive measure is to use memory access privileges, which in case of an application software that

would introduce memory corruption, could prevent crashing the operating system. It has to be noted that for some of the mentioned functionality an operating system might require hardware support (e.g., virtual memory) and therefore might not be applicable to all hardware and software combinations.

Non-functional criteria can ease development and maintenance of an embedded system. *Modularisation* and *Concurrency* often work hand in hand, allowing multiple independent modules to run parallel with minimal influence on each other, makes software design more comfortable. Support for additional *hardware* provided by an operating system allows for rapid development of prototypes and standardized, controlled interaction within the operating systems environment. Of course, an operating system should be *stable*, which means that it does not produce unrecoverable errors by itself. Otherwise most other criteria would be violated from the start.

1.3 Problem Statement

The microcontroller operating systems **Zephyr** [urld], **Riot OS** [urlc] and **Contiki** [urlb] need to be compared to each other according to the criteria (see section 1.2): **network behavior** (delay, throughput, reliability), **resource usage** (CPU time, memory usage, power consumption), **security** (memory protection), **support for concurrency**, **modularisation**, **hardware support**.

The comparison aims to be based on prototypical application software running on a representative hardware platform (see section 4.1) to provide benchmark results where possible. Minimal application software is to be implemented, this should be understood as not including anything additionally beyond what a certain use case requires, but does not imply additional attempts to exclude possibly removable operating system specific parts. If possible, existing sample code provided by the software projects around the operating systems should be reused. Attempts to optimize the implementation based on the underlying operating system should not be made, because for general use case scenarios it is assumed that the application software developer does not want to spend time with that task.

1.4 Aim of the Work

This work aims to be an initial effort in creating application independent, generalized approach to compare operating systems running on microcontrollers.

Such comparison could yield benefits for operating system developers, application developers as well as embedded system designers, including, but not limited to:

- qualitative and quantitative improvements to the operating systems as a result of identified weaknesses
- ease of hardware and software component selection during project planing

- greater success in prototype development due to easier requirement evaluation
- educational purposes in academical use

In the possible case that such a general comparison does not seem to be viable, in parts or as a whole, it should provide reasoning for future work.

Otherwise, the results of the comparison should show if and which of the considered operating systems has significant advantages or drawbacks compared to their competitors, with a focus on IoT and wireless networks.

1.5 Structure of the Work

In Chapter 1 "Introduction" the reader is introduced to this work, including characteristics of the evaluation criteria and the hardware platform used as test environment. Chapter 2 "State of the Art" discusses existing comparisons in general and in relation to the operating systems to be compared. Reasoning to why and how certain criteria are evaluated as well as relevant scientific aspects are explained in Chapter 3 "Methodology". Chapter 4 "Methods" introduces the methods used for the experiments described in Chapter 5 "Implementation". Results of aforementioned experiments are presented and discussed in Chapter 6 "Results" to be summed up in Chapter 7 "Conclusion" with IoT, future research and academical use in mind.

CHAPTER 2

State of the Art

One of the most common metrics that can be found in related work are static analysis of RAM and ROM usage (e.g., [WSS09, Table 1], [BHG+13, Table 1]). This is due to the fact that nearly every project around an embedded system has to deal with these constraints when selecting hardware as well as software components and is therefore also part of this work.

[TA09] makes observations on various time related behavior of operating system level behavior (e.g., task switch time, message passing). Provided metrics are very important but are already coupled more tightly to implementation details, which might be unknown at the time of component selection. While it makes sense to include especially inter-process communication in a generalized comparison such as this work aims to create, it is considered out of scope due to the focus of this work on IoT and respectively wireless networks.

Slightly different timing related behavior is studied in [AC09] with a focus on real-time. While real-time is a field of study on its own, their test setup using an oscilloscope is very similar to what is used for this work and will be explained in section 3.3. The criteria *Latency* and *Worst Case Response Time* evaluated are in their nature quite similar to what has been described in section 1.2 for GPIO capabilities.

While work has been done evaluating both, specific aspects of microcontroller operating systems (e.g., wireless networking performance [FG13][LDMM+05], stability and dependability [VDKGGT15]), as well as certain microcontroller operating systems in depth (e.g., RIOT OS [BHG+13], Contiki [DGV04]), it does not appear that a lot of direct comparison between microcontroller operating systems themselves, independent from application, exist.

CHAPTER 3

Methodology

3.1 Quantitative Criteria

Quantitative criteria are evaluated by example and/or performing experiments in a constructed scenario close to real world conditions. While the the experiments themselves might not be full applications, they serve as proof of functionality and provide a guiding information.

In this work, quantitative evaluation criteria can be split into two categories:

1. those that can be evaluated completely deterministic using known methods and available (software-)tools.
2. those that must be carried out as a real world experiments (i.e., possibly dependent on the surrounding environment) allowing for observation by external measurement instruments.

Regarding the first category, the results generated by available tools are assumed to have no uncertainty. Therefore, such results are either directly copied as reported or where necessary filtered and accumulated.

In the second category, uncertainty can be introduced by several sources (e.g., finite precision of measurement instruments, influence of a measurement method on the observed system, methodical errors, rounding errors). Where possible with reasonable expenditure, steps are taken to reduce the error as much as possible and described for reproducability.

Results of experiments which are expected to vary due to mostly unpredictable surroundings (i.e., network performance metrics) are sampled multiple times in the same environment to suppress random errors and edge cases. The number of repeated measurements has been selected as n=20 by rule of thumb to not break the scope of this work.

Resulting data is presented numerically as minimum, maximum and median (to reject one-time-off errors) as well as graphically, using box-plots[1] to hint at result distribution or bar-plot for comparison.

If not stated explicitly otherwise, all numeric values that are calculated or provided by a measurement instrument at a higher precision are rounded to 2 decimal places.

3.2 Qualitative Criteria

Qualitative criteria are mostly evaluated on the information and documentation provided by the software project themselves. Assuming that the software projects do not present themselves better than they actually are, that should be the most qualified source of information. If necessary, as either clarification or for additional information, the source code of the software project can be used. Additionally, other work conducted in the past may provide a base to extend on as well as an indicator for recent developments.

3.3 Indirect Measurements

Some of the evaluation criteria can not be measured directly (on the test platform itself), or would introduce an additional error. These problems are also hinted at in [AC09] and corresponding references. A very simple possibility would be to use the serial connection that probably most of the operating systems provide for log messages, but would also introduce additional uncertainty (buffering and delays on both sides). Therefore, such criteria are mapped to an equivalent time measurement using GPIOs providing external signals (see section 4.2), allowing the calculation of the original performance metric. Additional errors introduced by this approach should be neglect-able in direct comparisons.

[1] `http://www.physics.csbsju.edu/stats/box2.html`

CHAPTER 4

Methods

4.1 Hardware Platform

This work evaluates the three operating systems **Zephyr** [urld], **Riot OS** [urlc] and **Contiki** [urlb] by examining various functionality and typical use cases by example and conducting experiments on real hardware.

The hardware used is the *CC2538 Development Kit* [urla] produced by *Texas Instruments*[1]. Some of its features are:

- CC2538EM (evaluation module with microcontroller)
  - ARM Cortex M3 based CPU
  - 32 MHz crystal oscillator
  - 2.4 GHz RF transceiver with IEEE 802.15.4 support
  - 512 KB FLASH memory (ROM)
  - 32 KB SRAM memory (RAM)
  - Encryption Hardware Support
  - U(S)ART, SPI, I2C
- SmartRF06 (peripheral board)
  - XDS 100v3 Programming and Debugging Interface
  - 5 Buttons
  - 4 LEDs

[1] `http://www.ti.com/`

- LCD
- SD Card Slot

Especially the IEEE 802.15.4 RF transceiver is essential to this evaluation since it is used as a wireless network interface.

4.2 Time Difference Measurement

A 2-channel Digital Storage Oscilloscope (DSO) is used to record the state transitions (i.e., transitions from low to high or vica versa) of up to two digital signals. The DSO can then be used to measure the time difference between either consecutive state transitions of one signal (figure 4.1 (left)) or between corresponding state transitions of two signals (figure 4.1 (right)).

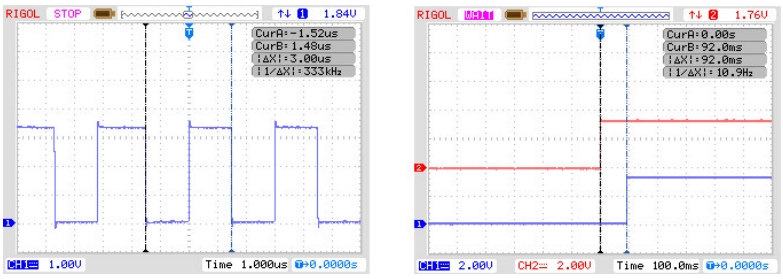

Figure 4.1: A time difference measurement performed on one signal (left) or two signals (right)

4.3 Power Consumption

A multitude of possibilities to measure the power consumption of a device with different measurement instruments exist. Depending on the capabilities of the selected methods and measurement instruments, the accuracy of the result can vary. The following subsections describe two different possibilities with their advantages and drawbacks.

4.3.1 USB-Meter

Since the testing hardware (SmartRF06 Board, see section 4.1) can be supplied via Universal Serial Bus (USB) connectors, a relatively cheap USB-Power-Meter (available for less than 10€ in online stores) is a convenient option to obtain a measurement, but should be considered more as a rough estimate. A drawback is, that in the absence of an exact product-name or specification, no data-sheet stating the internals (i.e., measurement circuitry used, method and accuracy of the measurement) could be found at the time of writing. Due to the low price compared to (qualitative feature-) comparable measurement instruments, it appears reasonable to expect that the electrical power and charge values

recorded are only rough estimations calculated from voltage and current (using discrete time step integration).

Figure 4.2 shows the USB-Power-Meter that is used in this work. The interface shows the electrical voltage in volt, electrical current in ampere, electrical power in watt and electrical charge that went through the device in milliampere hours.

Figure 4.2: The USB-Power-Meter in operation

4.3.2 Oscilloscope, Shunt

Using a 2-channel DSO in combination with a shunt (a low value high precision resistor) in the devices supply path allows to measure both, the supply voltage and voltage present at the shunt (see figure 4.3). Subsequently, the current flowing through the shunt can be calculated. By doing that, continuous measurement data can be recorded and used to calculate power consumption (using discrete time step integration). Accuracy is now closely coupled to the quality of the measurement equipment (i.e., the resolution of the DSO and precision of the shunt). Compared to cheaper Analog Digital Conversion (ADC) equipment (e.g., Multimeter, USB-Meter as described in section 4.3.1) a typical DSO is expected to yield better results. An undesired side effect is the measurement error introduced by the additional resistor in the devices supply path, slightly limiting the current flowing. Assuming an unaffected current of less than 10 mA at 5V supply voltage results in an estimation of the electrical load to be greater than 500Ω. A shunt smaller than 1Ω would introduce an error of less than 0.2%, small enough not to influence the results of this work relevantly.

Additionally, when making these two measurements at the same time (i.e., voltage and current) one of the two measurement instruments introduces a measurement error to the

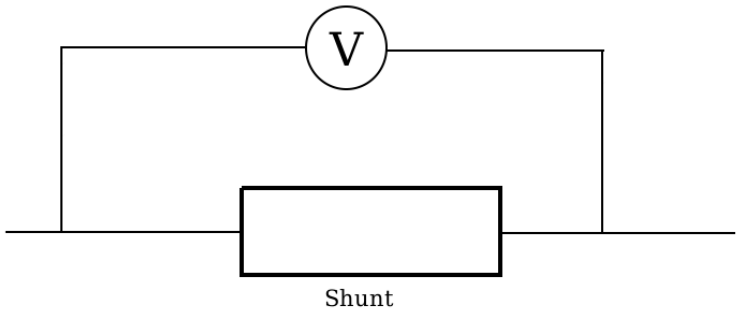

Figure 4.3: A shunt with a voltage meter attached

other measurement. This happens due to non-ideal measurement instruments (i.e., non-zero internal resistance of ampere-meter, non-infinite internal resistance of volt-meter). Figure 4.4 shows two possible measurement setups where the one of the measurements is influenced by the other.

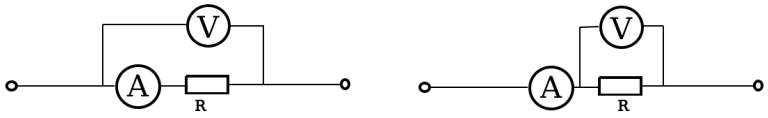

Figure 4.4: Comparison between current correct measurement (left) and voltage correct measurement (right)

For this work, it makes sense to set up the measurement in a way that the current measurement is not influenced by the voltage measurement as depicted in figure 4.5. The voltage is expected to be regulated by the power supply and therefore more stable, while current peaks are expected as result of varying modes of operation (e.g., use of sleep modes, wireless activity).

The testing hardware already features circuitry to ease the current measurement, including a shunt and measurement amplification. This circuitry is shown in figure 4.6 and has the added benefit of almost eliminating the previously described influence of the second measurement instrument, replacing it with a small error independent of the measurement instrument. Said error could be calculated using the equivalent parallel resistance, but is also neglected as insignificantly small for the purpose of this work.

4.4 Memory Usage

Many operating systems that target embedded systems do not provide dynamic memory management at all or avoid it as much as possible. Reasons for this are at least partially the very constrained amount of resources, harder predictable runtime behaviour and

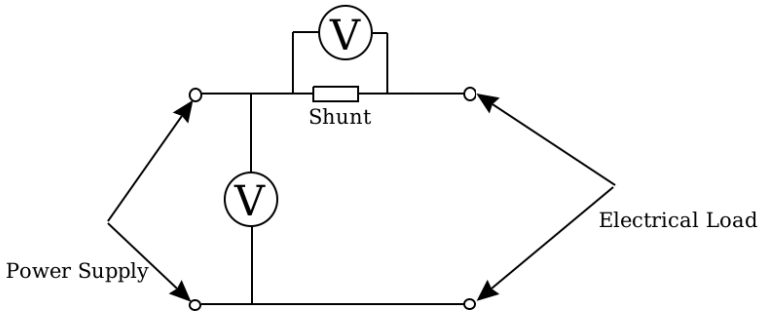

Figure 4.5: Schematic of current correct measurement setup

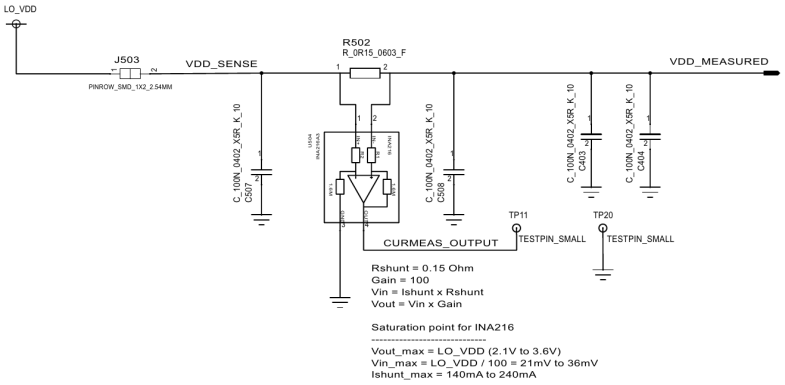

Figure 4.6: Schematic of the current sense circuitry provided by the SmartRF06. (Source: [72s17])

the additional trouble to prevent or analyze programming mistakes. Therefore, often memory blocks larger than absolutely required (e.g., message buffer(s) of maximum message length) or larger than required at all times (e.g., preallocated message queue(s) of fixed size) are allocated statically for a very specific purpose. This allows this work to derive both, the ROM and RAM usage, directly from result of the compiler. Otherwise, a run time evaluation, using (additional) debugging hardware or accepting overhead to perform the measurement, would be necessary.

Information about the ROM and RAM usage can be extracted from the compiled binary (which includes both, operating system as well as application), using available

software tools. For example, this can be done by using the command *arm-none-eabi-size programm.elf* on a computer running Linux where the required software packages are installed.

As shown in figure 4.7, the column *text* (i.e., executable program) lists the content that will be placed in ROM, where as both *data* (i.e., initialized program data) and *bss* (i.e., uninitialized program data) occupy space in RAM.

```
   text    data     bss     dec     hex filename
  52310    3360   28744   84414   149be zephyr.elf
```

Figure 4.7: Exemplary output of the "size" tool

While the hardware only differentiates between two types of memories, ROM and RAM, software could differentiate to greater detail (e.g., kernel vs. application, per driver/module/application, initialized vs. uninitialized). For the sake of generalization and simplicity, this work focuses on ROM and RAM, with RAM subdivided into initialized and uninitialized memory. The subdivision of RAM serves the idea to hint at buffer allocation mechanisms already mentioned.

CHAPTER 5

Implementation

5.1 Software Versions & Configuration

Zephyr [urld]
used https://github.com/KillerLink/zephyr/tree/e0bd9e264dd17a5dc4fb
based on https://github.com/zephyrproject-rtos/zephyr/tree/26c9c7480ec0a379809e

Riot OS [urlc]
used https://github.com/KillerLink/RIOT/tree/96885577e487d4c22ce7
based on https://github.com/RIOT-OS/RIOT/tree/1d693403b65231b6784a

Contiki [urlb]
used https://github.com/KillerLink/contiki/tree/8679b37c3d80444b3b06
based on https://github.com/contiki-os/contiki/tree/32b5b17f674232867c22

GCC: **8.2.1 20180831**
arm-none-eabi-gcc: **8.2.0**

To compare the operating system **Zephyr** [urld] in this work, preliminary implementation work, which has not yet been fully published had to be done. Both, modifications and experiment implementations are published via git repositories listed above[1].

Working with a congested network (i.e., sending data as fast as possible) showed network stability issues in **Riot OS** [urlc]. These issues are presumably caused due to loosing IEEE 802.15.4 frames at the receiver, coming in at a very high rate. For this reason

[1] commit hashes are shortened to 20 characters but should still be unique

the link layer driver had been modified from flushing the hardware receive buffer after reception of a complete IEEE 802.15.4 frame to keeping the remaining data. Additionally, to prevent buffer underflows, it now waits for data to be available instead of expecting it to be already there. Some of these problems are also hinted at in various issues in the repository[2].

To exclude as much external influence as possible, this work uses a direct connection between 2 communicating nodes, therefore there are no routers, no routing protocols and no auto-configuration. For this purpose, the devices were configured to have the following Internet Protocol Version 6 (IPV6) link-local addresses:

- Server (Receiver): fd0b:ad46:74e5:de4c::1
- Client (Transmitter): fd0b:ad46:74e5:de4c::2

5.2 Networking

For all network experiments described below, the decision to send data in only one direction as opposed to request-response based communications was intentionally and serves the purpose to exclude unnecessary interactions between the application software and operating system from polluting the measurement with noise (e.g., additional system-calls, waiting on the availability of memory buffers).

Additionally, it has been tried to keep code in places that would influence the results short (e.g., in between return of a send/receive function and the generation of the corresponding external signal, when generating the signals using direct memory access to the General Purpose Input Output (GPIO) peripheral).

This work uses User Datagram Protocol (UDP) because of the following reasons:

- UDP is a well-known, widely used protocol that can be used as a common functionality provided by all comparison candidates. Many other protocols can be implemented on top of UDP, including TCP-over-UDP[3][LBS] if Transfer Control Protocol (TCP) is not natively supported but required.
- UDP eases the modeling of experiments where data transmission would be too large for single packets on the link layer of the OSI Network Model. [4]
- UDP has lower resource requirements (i.e., RAM, CPU time) compared to TCP, which might be important depending on the available hardware resources.

[2] `https://github.com/RIOT-OS/RIOT/projects/5`
[3] `https://tools.ietf.org/id/draft-baset-tsvwg-tcp-over-udp-01.html`
[4] If the required fragmentation capabilities are implemented.

- Measuring reliability from the view point of the application only makes sense in the context where data loss is not already prevented by a lower layer in the OSI Network Model, like TCP would.

The implementations use the functionality provided by the different operating systems that is closest to the concept of Unix-sockets, if that choice is available. Also, if applicable, the Media Access Control (MAC) is chosen to be be as simple as possible (i.e., sending data exactly once if possible, opposed to sending the same data multiple times to increase the chance of successful transmission). These decisions aim to create experiments that can be implemented less dependent on the underlying operating system and are more comparable to future experiments.

When choosing the amount of data sent per packet, one needs to consider possible implications. On the one hand, small packets introduce a relatively large overhead (e.g., addressing, packet frames), that can influence throughput negatively. On the other hand, large packets reduce that overhead, but might need to be split into fragments to be transmitted over the link-layer, relying on intermediate network-layers to handle this, possibly having a negative impact on latency and/or reliability. For aforementioned reasons, 2 sets of measurement series are taken, with either 50 or 1000 bytes per UDP packet.

5.2.1 Latency

Measuring the **latency** is done by sending a fixed amount of data repeatedly from one device to another, measuring the delay from start of transmission to completing reception. The transmitting device raises an external signal just before the application issues the call to start sending data, while the receiving device raises an external signal immediately after the program execution returns from the call to receiving data. The time difference between those 2 signals is measured as described in section 4.2.

5.2.2 Throughput

The **throughput** can be observed by measuring how much data can be sent in a fixed amount of time or sending a fixed amount of data and measuring the time it took. For both cases, the amount of transmitted data can be divided by the time taken to obtain an indicator for throughput. This work will use fixed amount of data, purposely neglecting lost data by continuous transmission/reception until the desired amount of data has been received. Assuming data is continuously transmitted and received, an external signal can be generated whenever the chosen amount of data (for 1000 Byte packets: $TOTAL\_DATA = 100 kilobyte$, for 50 Byte packets: $TOTAL\_DATA = 50 kilobyte$) has been sent / received. The time difference between two consecutive signals can be measured, for both, receiver and transmitter, as described in section 4.2. The throughput is then calculated from the receivers' measurement $\Delta t_{receiver}$ as shown in equation (5.1).

$$throughput_{kilobyte/s} = \frac{TOTAL\_DATA}{\Delta t_{receiver}} \tag{5.1}$$

5.2.3 Reliability

Evaluation of network communication **reliability** can be done by sending a fixed amount of data from one device to an other and asses how much of it was either lost or successfully transmitted. There are more intricate methods (e.g., using identifiable packets) that could provide a better result, but were not chosen due to higher complexity and possible impact on the result (e.g., missing a packet because a buffer ran over while the previous packet is checked).

The receiver transmits a fixed amount of data (TOTAL_DATA = $100 kilobyte$) once, while the receiver reports all data received (e.g., using the serial interface). Data transmission is done without special handling of data loss on layers below the transport layer of the OSI Network Model from within the application implementation. Here, the implementation of this experiment relies on the operating systems' capabilities to provide textual output, reporting the amount of received data. Using the output of the receiver after the transmitter has completed sending TOTAL_DATA bytes, a value for $Bytes_{received}$ can be obtained. From the difference between the amount of data sent and received, the data loss (in percent) can be inferred as shown in equation (5.2).

$$loss_{percent} = (1 - \frac{Bytes_{received}}{\text{TOTAL\_DATA}}) * 100 \tag{5.2}$$

5.3 Resource Usage

5.3.1 Memory Usage

Because this experiment's metrics depend on the specific application, 3 different scenarios are selected to be representative of expected typical uses:

- **GPIO**: includes reading and writing from and to GPIOs. The implementation for the *input to output delay* experiment described in section 5.4.3 is used.
- **serial**: includes writing text over a U(S)ART. Implemented as an application that writes "Hello World!" approximately every second.
- **networking**: includes transmission of data over a wireless network. The implementation for the *network throughput* (server/receiver side) experiment described in section 5.2.2 is used.

Results can be obtained as described in section 4.4.

5.3.2 CPU Usage

Comparing the **CPU usage** of an operating system is possible by estimating how much time is spent in the operating system itself as opposed to the application software. Of course this depends on the application software, since there are tasks that involve interaction with the operating system as well as tasks that can run independently. While the former are very specific to implementation and application, only the latter will be evaluated here as a more generalized approach, simulating a best-case scenario. Evaluating this can be done by raising an external signal, advancing a simple counter in the user application until it reaches a certain value, and finally lowering the external signal again. The time difference is measured as described in section 4.2. By comparing the measured time to the theoretical time it would take, the difference can be used to indicate how much time was not spent in the user application (i.e., the operating system). Reference times can be either obtained summing the execution times of the assembly instructions used in the loop that increases the counter or running the same loop on the microcontroller without an underlying operating system.

5.3.3 Power Consumption

The **power consumption** of a device can be measured as described in section 4.3. Since this metric also depends on the application, the same scenarios described in section 5.3.1 are tested as representatives.

5.4 GPIO Capabilities

5.4.1 Maximum Toggling Frequency

Evaluating the **maximum toggling frequency** of a GPIO can be done by configuring it as an output, toggling it as fast as possible and measuring the frequency of the generated signal with a DSO. Section 4.2 describes the measurement, figure 4.1 (left) shows a corresponding example.

Some operating systems provide a dedicated "toggle" command for GPIOs, which possibly makes use of hardware support for toggling the state of a GPIO configured as output. If no such functionality is provided, the dedicated "on" and "off" commands must be used, as described in section 5.4.2. In this case, the maximum toggle frequency can be calculated as shown in equation (5.3) instead.

$$f_{maximum\_toggling} = \frac{1}{t_{min\_on} + t_{min\_off}} \tag{5.3}$$

5.4.2 Minimum On-/Off-Times

The **minimum on-/off-times** t_{min_on} and t_{min_off} can be measured using a DSO on a GPIO configured as output, alternating its state via dedicated "on" and "off" commands

provided by the operating system. Section 4.2 describes the measurement.

5.4.3 Input to Output Delay

The **input to output delay** can be observed with a DSO on a GPIO configured as output and changing its value when the external interrupt is handled by the application software. Section 4.2 describes the measurement, figure 4.1 (right) shows an example.

CHAPTER 6

Results

6.1 GPIO Capabilities

The results presented in table 6.1 are obtained from the experiment described in section 5.4.

Zephyr [urld] did not provide any direct digital output toggle capabilities, which is why the maximum toggle frequency is calculated from the sum of the minimal on-/off-times as described in section 5.4.1. The explanation to the resulting lower frequency from already higher minimum on-/off-times is that the current implementation is a wrapper to the Texas Instruments HAL (Hardware Abstraction Library), adding another layer of indirection.

Riot OS [urlc] unexpectedly exhibited different on-/off-times when using digital write functionality, even though toggling had an even mark-space ratio. This could not yet be explained.

Contiki [urlb] has shown significant longer input to output delay, most likely because it uses threads and events, adding additional overhead from scheduling and event generation. Since the difference in performance is this large, a setup mistake or failure in experiment design is deemed to be possible, but could not be determined clearly.

GPIO	Zephyr	Riot OS	Contiki
maximum toggle frequency	(223 kHz)	379 kHz	362 kHz
minimal on-time	2.24 us	1.88 us	1.48 us
minimal off-time	2.24 us	1.36 us	1.48 us
input to output delay	6.4 us	7.2 us	92.0 ms

Table 6.1: Measurement results for GPIO capabilities

As a reference, the CC2538 User's Guide [71c13] states that the microcontroller is capable of "Fast toggle capable of a change every two clock cycles". Using the 32MHz oscillator as system clock, this would theoretically result in 16MHz maximum toggling frequency when implemented as device specific instructions. Because the flash is limited to 16MHz, this is only true when executing from RAM or possible with good prefetching performance.

6.2 Resource Usage

6.2.1 CPU Usage

This experiment has proven to be difficult to implement as would have been described in section 5.3.2 which also has led to questioning its design.

```
1 RAMFUNC void asm_loop2(void) {
2     EVAL_LED1_OUTPUT;                          //prepare external signal
3     EVAL_LED1_OFF;                             //external signal off
4     while (1) {
5         __asm__ volatile (
6             "MOV r1, #1<<22 \n\t"              //load loop counter
7             ".align 8\n\t"                     //alignment (possible prefetching improvement)
8             ".ASMLOOP2:"                       //loop head
9             "nop\n\t"
10            "CBZ r1, .ASMLOOP2DONE\n\t"        //conditional branch on counter zero
11            "nop\n\t"
12            "SUB r1, r1, #1\n\t"               //subtract one
13            "B .ASMLOOP2\n\t"                  //branch to loop head
14            ".ASMLOOP2DONE:"                   //loop end
15            "nop\n\t"
16            : /* No outputs */
17            : /* No inputs */
18            : "cc", "r1"                       //clobber status register and r1
19        );
20        EVAL_LED1_TOGGLE;                      //external signal generation
21    }
22 }
23
```

Figure 6.1: Exemplary assembler delay loop

Figure 6.1 shows the delay loop written in assembly to implement what has been described in section 5.3.2. To obtain a reference time value, this loop was executed from RAM in a baremetal environment (i.e., no operating system, no interrupts). While testing and developing this implementation, a few problems became apparent:

- Some limitations to this test are implied by the hardware. The *CC2538 Development Kit* [urla] used in this work features a CPU running at 32MHz, but its execution speed from flash is limited to 16MHz without prefetching, increasing its platform dependency.
- If prefetching would be used to circumvent the limitation, the result would be hard to predict. While the influences of prefetching could be calculated in an environment without interrupts, it seems to be unfeasible when interrupt could occur frequently.
- When executing from RAM (which is not available to all targets) to circumvent both, flash limitations as well as prefetching in-determinism, branch prediction seems to slow down the loop. Documentation [70c07, Table 18-1] states that for

memory usage [bytes]		Zephyr	Riot OS	Contiki
GPIO	.text (ROM)	12002	8600	43534
	.data (RAM)	1712	144	481
	.bss (RAM)	4720	2860	12591
serial	.text (ROM)	14242	10072	43636
	.data (RAM)	1716	136	465
	.bss (RAM)	4696	2660	12607
networking	.text (ROM)	52362	44568	39700
	.data (RAM)	3360	176	440
	.bss (RAM)	22880	17640	28980

Table 6.2: Memory usage per application, grouped by use (executable, initialized, uninitialized)

the hardware used in this work, any branch requires a fixed amount of cycles plus a value P, ranging from 1 to 3 inclusive, depending on the pipeline as well as alignment and width of (branch-) target instruction. Tests in the baremetal environment lead to the conclusion that in this very specific setup P would be always 3, but no documentation or authoritative confirmation to this assumption was found. It also hints at problems with generalization and portability of this approach.

- Implementing execution from RAM would not be possible on all platforms (due to lack of hardware support). Even if it is possible on the *CC2538 Development Kit* [urla], doing so would most likely require unreasonable modifications to the current build system (i.e., compiler flags, linker scripts) to ensure that the delay loop is compiled equally in all systems and placed in / executed from RAM.

Additionally, **Contiki** [urlb] handles multitasking (see section 6.5) very differently. While preemptive scheduling is supported by means of a library, by default multitasking is cooperative. As a result, the delay loop would just block the CPU.

In conclusion, the experiment design has been found to be inapplicable and unsuitable. A possible alternative could be to use a specific workload and define constraints that would require the application to interact with the operating system in a limited and defined way that allows for comparison.

6.2.2 Memory Usage

The results presented in table 6.2 are obtained from the experiment described in section 5.3.1.

For the ROM usage, **Zephyr** [urld] and **Riot OS** [urlc] perform very similar, with the later using less in all three scenarios. As to be expected, the ROM usage is larger for the networking scenario, due to additional functionality required compared to the other two.

Power Usage		Zephyr	Riot OS	Contiki
USB Meter [W]	GPIO	0,395	0,395	0,354
	Networking	0,503	0,503	0,503
Shunt, Amplifier, DSO [mW]	GPIO	2,93	2,60	0,24
	Networking	9,11	9,76	9,11

Table 6.3: Power consumption per application

Contiki [urlb] uses a much larger amount of ROM for the GPIO and serial scenario, because it seems to always include the networking functionality. This is very likely a configuration error and attempts to improve the situation have been made (i.e., explicitly trying to exclude certain parts, adding optimizations), but led to almost the same result or a failing build process. The drop in ROM usage for **Contiki** [urlb] in the networking scenario probably is a result of explicitly selecting interchangeable functionality different from default.

Regarding the RAM usage, the results differ a lot over both, different scenarios as well as operating systems.

Most notably here is the difference for initialized and uninitialized memory of **Zephyr** [urld] in comparison to the other two. While the results are not obviously wrong, it might be possible that **Riot OS** [urlc] and **Zephyr** [urld] have customized their build process in a way that interferes with the *size* tool. A possible explanation is that the linker script and startup code have been customized.

If the sum of initialized and uninitialized data is observed, **Riot OS** [urlc] used the least RAM in all three scenarios.

Also note the large jump in use of RAM for uninitialized data in the network scenario, caused at least partially by buffers reserved for the required functionality. As already mentioned in section 4.4, the size of these buffers is defined at compile time and possibly larger than ever needed, depending on the application and surrounding environment.

6.2.3 Power Consumption

The results presented in table 6.3 are obtained from the experiment described in section 5.3.3.

The first thing to note is the different unit scale used for the USB-Power-Meter and oscilloscope. This is due to the fact that the USB-Power-Meter, as visible in figure 4.2, reports the power consumption in watt, while the measurements taken with the DSO have been recorded in millivolt and resulted in calculations of milliwatt. Observing this, as well as the order of magnitude the recorded results differ, is enough of an indicator that the USB-Power-Meter is absolutely unsuitable for this type of measurements. Further reasons are that the USB-Power-Meter is unable to measure only the microcontroller alone, but measures even before the voltage regulator and including any peripherals that

are included on the *CC2538 Development Kit* [urla]. Therefore the following observations will be made based on the results obtained from the DSO.

As to be expected, measurements with active networking have increased power consumption compared to the GPIO application due to the use of the RF hardware. While Riot OS does have a slightly higher power consumption in this case, it does not seem to be significant and is most likely a measurement error, introduced by recording a singular value (see figure 6.2) (opposed to obtaining an average over time by means of numeric integration).

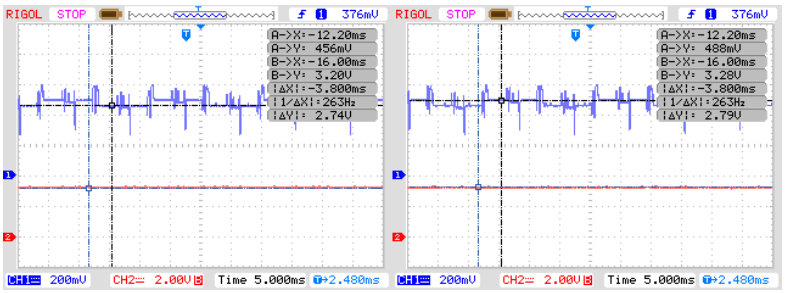

Figure 6.2: Power consumption measurement using a DSO connected to a shunt over a measurement amplifier circuit (see figure 4.6). Measurement errors: underestimation (left), overestimation (right)

An interesting fact is that **Contiki** [urlb] exhibits vastly lower power consumption in the GPIO application than the other two. It appears that **Contiki** [urlb] makes use of some power saving features (e.g., CPU sleep modes) by default. According to respective documentation, it seems that equivalent features should be supported in both, **Riot OS** [urlc] as well as **Zephyr** [urld], but are either not implemented or enabled.

6.3 Networking

Following remarks apply to all of the networking experiments.

The network setup described in section 5.2 was difficult to implement in **Riot OS** [urlc] and **Contiki** [urlb], because both generally seem to assume that router, routing protocols and auto-configuration should be used. Due to these assumptions, there was few to no documentation or examples regarding this configuration, requiring significant amount of trial and error to archive the desired results. **Zephyr** [urld] in contrast to that allowed simple configuration of the desired IPV6 addresses through its configuration system.

The results in the following sections are presented in multiple ways to allow for better interpretation:

- tabular for readability and exact results,
- side by side of

– either a combined box- or bar-plot, allowing for direct comparison
– a set of separate box-plots, highlighting the sample distribution

Also consider that these experiments have been carried out under very favorable conditions (i.e., low distance, low interference). While this is very important and should definitely not be neglected for real world deployments, it has been considered out of scope of this work, which is more focused on the maximum achievable performance under good conditions. A considerable amount of work exists that focuses on reliable low power transmission for IoT applications.

Out of interest and given opportunity an additional experiment was conducted, where measurements were taken using timestamps attached to serial output of the device, as briefly mentioned in section 3.3. The timestamps were created on the computer, using a tool called *ts* from a software package called *moreutils*. Using the command *ts "[%.s]"* allows prefixing of the input with system local timestamps, which in theory could be used to make relative measurements.

Figure 6.3 shows 2 sets of measurements taken per operating system for throughput, using 1000 bytes per packet and 100KB total, as described in section 5.2. The numerical results, compared to the ones obtained using external signaling and a DSO (see section 6.3.2, section 6.3.2), are very similar and appear to be good indicators. Upon closer inspection, the distribution of measurement results visible in figure 6.3 hints at some problems. Measurements taken for **Zephyr** [urld] show a very small interquartile range, with some very atypical outliers, all of which sharing almost the same value. Similar cues can be found for **Contiki** [urlb], even though the interquartile range is much larger, it almost matches the range from minimum to maximum. Additionally, the median is located close to the maximum instead of somewhere in the middle. Most likely due to buffering, only a few discrete values have been sampled for at least some of the operating systems. It can be concluded that this approach (i.e., attaching timestamps to serial textual output) is particularly susceptible to errors of this kind. Furthermore, it has implications on possible experiment designs that would use a serial connection to a computer (e.g., to emulate a network interface) which would then perform measurements.

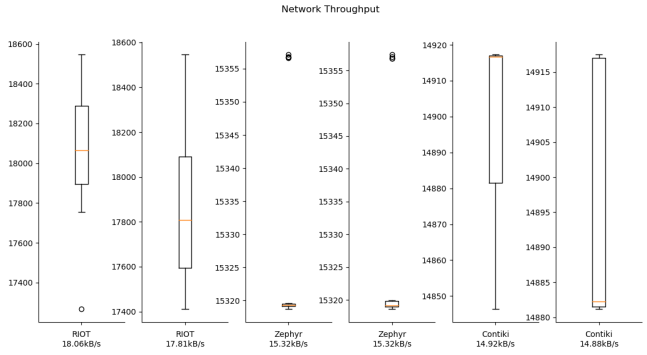

Figure 6.3: Network throughput measured using timestamps generated on serial output

6.3.1 Latency

The results presented in table 6.4, figure 6.4 and figure 6.5 are obtained from the experiment described in section 5.2.1.

As to be expected, the latency is shorter for smaller packets. Neglecting the exact amount of overhead, effectively assuming that both fit in a single IEEE802.15.4 frame, the latency of a 100 byte UDP packet would be the same as for a 50 byte UDP packet. Conducting a thought experiment, the 10 times larger 1000 byte UDP packet should take (at least) 10 times as long, which seems to hold true.

While the total difference in results seems rather small, an interesting observation can be made comparing the 2 different experiment configurations with another. **Zephyr** [urld] had the best result for smaller packets, but the worst for larger ones. Almost the inverse seems to happen for **Riot OS** [urlc], which is close to the worst result, obtained from **Contiki** [urlb] for smaller packets, but has significantly better results for larger packets than both of its competitors.

The obtained results can be compared with the theoretical analysis provided by [LDMM+05, Table3, 16 bit address, no ack] and appear to be plausible.

Latency [ms]	50 byte packets			1000 byte packets		
1 Meter @ 0 dbm	Zephyr	Riot OS	Contiki	Zephyr	Riot OS	Contiki
minimum	5,16	6,36	6,92	68,40	55,20	68,00
maximum	5,20	6,52	6,92	68,80	55,60	68,00
median	5,16	6,36	6,92	68,80	55,20	68,00

Table 6.4: Network latency measurement results, combined

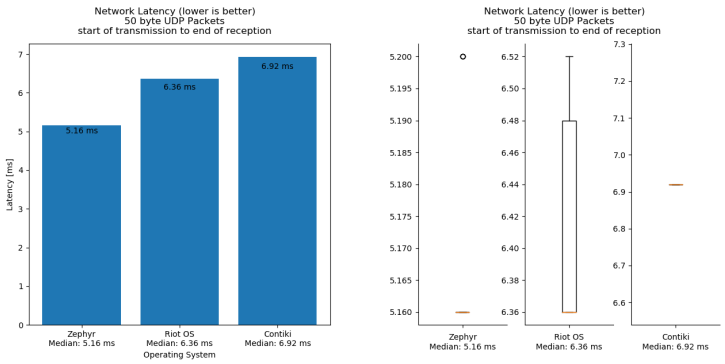

Figure 6.4: Network latency, 50 byte packets, comparison (left) and distribution (right)

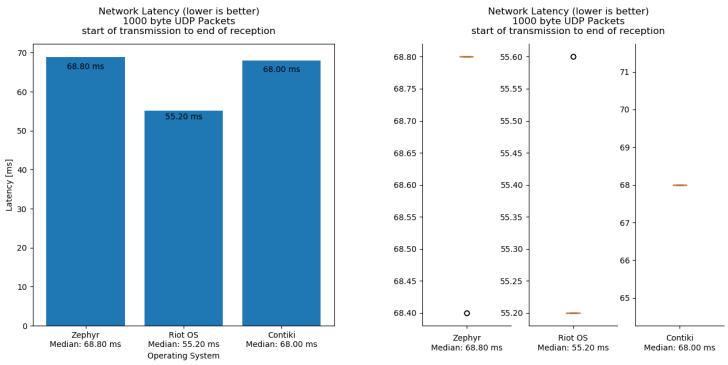

Figure 6.5: Network latency, 1000 byte packets, comparison (left) and distribution (right)

6.3.2 Throughput

The results presented in table 6.5, figure 6.6 and figure 6.7 are obtained from the experiment described in section 5.2.2.

As to be expected, the throughput for smaller packets is worse than for larger packets, due to the overhead introduced by IPV6 and UDP. The 50 byte UDP packets perform particularly bad due to the inefficient use of IEEE802.15.4 frames, which can be up to 125 bytes large, whereas the 1000 byte UDP packets fill these frames completely except the last fragment.

In [LDMM+05, Table2, 16 bit address, no ack], the theoretical maximum throughput of raw data was determined to be $151596bps == 18.9495kilobyte/s$. The result obtained for 1000 byte UDP packets almost reaches these numbers, with **Contiki** [urlb] reaching 76.04%, **Zephyr** [urld] 80.42% and **Riot OS** [urlc] 92.56%. The difference is most likely due to protocol overhead and unreliable data transmission.

Throughput [kilobyte/s] 1 Meter @ 0 dbm	50 byte packets 50 kilobyte total			1000 byte packets 100 kilobyte total		
	Zephyr	Riot OS	Contiki	Zephyr	Riot OS	Contiki
minimum	8,20	8,20	7,90	14,12	15,53	14,20
maximum	8,28	9,26	7,96	15,24	18,25	14,79
median	8,30	9,20	7,90	15,24	17,54	14,41

Table 6.5: Network throughput measurement results, combined

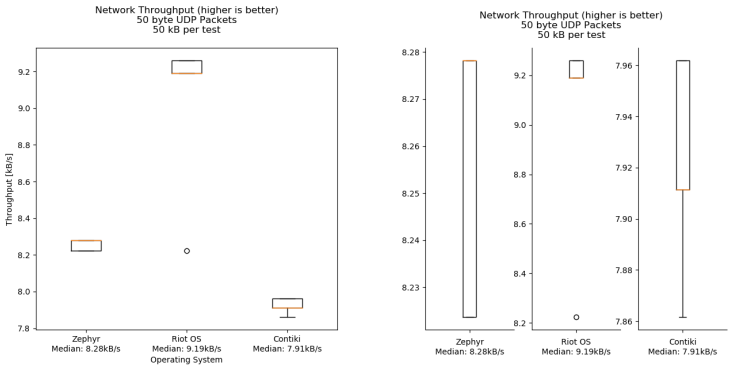

Figure 6.6: Network throughput, 50 byte packets, comparison (left) and distribution (right)

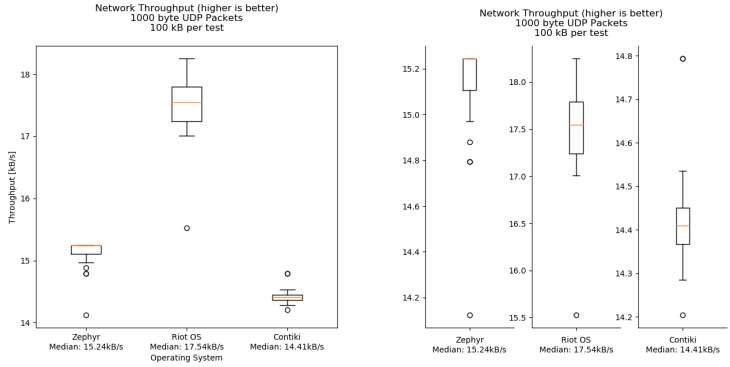

Figure 6.7: Network throughput, 1000 byte packets, comparison (left) and distribution (right)

6.3.3 Reliability

The results presented in table 6.6, figure 6.8 and figure 6.9 are obtained from the experiment described in section 5.2.3.

As to be expected, using larger UDP packets yields greater loss, because the entire UDP packet is lost if a single fragment is lost.

Assuming p is the probability to lose any IEEE802.15.4 frame, the probability q to not loose any of n frames can be calculated as $(1 - ((1 - p)^n)$. This formula can be used to linear interpolate the results obtained from 50 byte UDP packets ($p \leftarrow 0.0065$) to 1000 byte UDP packets ($n \leftarrow 11$). The calculated results are very similar to the experimental results for 1000 byte UDP packets for **Zephyr** [urld] and **Riot OS** [urlc], but differ for **Contiki** [urlb], having higher loss rate in the experiments performed.

Surprisingly, the best and the worst result, for both 50 and 1000 byte UDP packets, differ by approximately factor 2.

Loss [%] 1 Meter @ 0 dbm	50 byte packets 100 kilobyte total			1000 byte packets 100 kilobyte total		
	Zephyr	Riot OS	Contiki	Zephyr	Riot OS	Contiki
minimum	0,60	0,10	1,20	4,00	6,00	16,00
maximum	3,55	1,25	1,50	37,00	12,00	28,00
median	0,65	0,95	1,25	9,00	8,00	19,00

Table 6.6: Network loss measurement results, combined

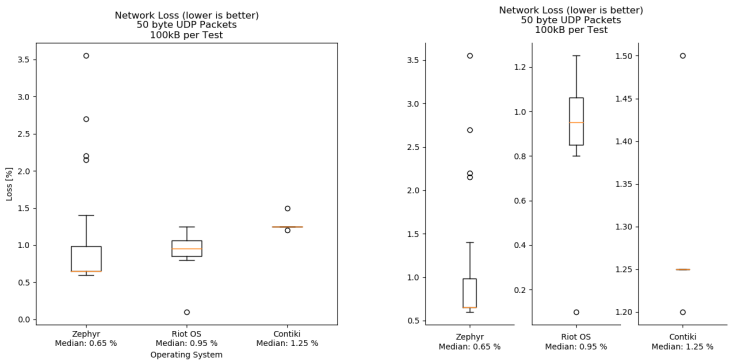

Figure 6.8: Network loss, 50 byte packets, comparison (left) and distribution (right)

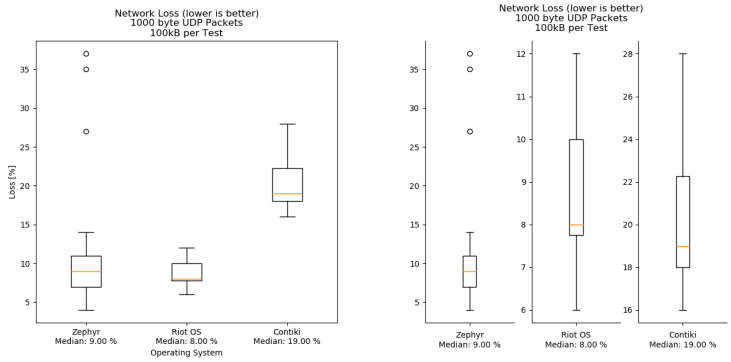

Figure 6.9: Network loss, 1000 byte packets, comparison (left) and distribution (right)

6.4 Security - Memory Protection

Zephyr [urld] provides an option to split application memory and kernel memory, as well as support for a Memory Protection Unit (MPU) on specific CPUs.

Riot OS [urlc] does not seem to have special support for memory protection at the

moment, but there has been preliminary work[1] for supporting MPUs.

According to [DGV04], **Contiki** [urlb] is specifically not designed to support memory protection, though source code shows that some CPUs have specific implementations that seem provide relevant features.

6.5 Multitasking

In the context of this work and considering the typical microprocessor available in current hardware, multitasking is understood as the ability to define multiple entry points to application code that can run interleaved depending on certain conditions (e.g., input data being available).

All of the operating systems compared support a form of multitasking, however there are important differences.

Zephyr [urld] and **Riot OS** [urlc] provide the widely known and used concepts of threads and typical synchronization primitives (e.g., semaphores, mutexes). Both of the aforementioned operating systems are capable of cooperative as well as preemptive multitasking.

In contrast to that, **Contiki** [urlb] uses *Protothreads*[2] [DSVA06] as their primary concept of multitasking. This has implications on how application code needs to be written, but can have advantages on severely resource constrained systems (i.e., less CPU and RAM overhead). However, an auxiliary library[3] provides functionality for the more typical concept of threads.

6.6 Modularisation & Hardware Support

Due to the very different structure and dimensions of modularisation described it is difficult to make a general comparison. For this reason, the general structure and quantity are described loosely for each of the compared operating systems below. Please note that the numbers below are extracted from the software repositories by counting indicators (e.g., directories, specific strings) and are very likely to be inexact or even wrong due to irregularities in project structures.

Zephyr [urld] structures modules in its software repository by CPUs, boards, and drivers. CPUs are grouped by architecture and family. Common drivers are grouped by their purpose (e.g., GPIO, I2C, SPI, PWM). At the time of writing, there were 7 architectures[4], with a total of 81 CPUs / CPU-families, 134 boards and 177 drivers in 37 subgroups. This

[1] `https://github.com/RIOT-OS/RIOT/issues/3284`
[2] `http://contiki.sourceforge.net/docs/2.6/a01802.html`
[3] `https://github.com/contiki-os/contiki/wiki/Multithreading`
[4] arc, arm, nios2, riscv32, x86, x86_64, xtensa, (posix was not included here, since it rather describes an interface than a CPU)

information has been obtained by counting the respective directories or configuration files[5]. Configuration of an application is managed via Kconfig[6], which has different front-ends available.

Riot OS [urlc] is structured by CPU-families, boards and drivers. At the time of writing, there were 49 CPUs from 5 architectures[7], 143 boards and 100 drivers. This information has been obtained by counting the respective directories or configuration files[8]. Configuration of an application is managed via variables declared in a Makefile, split into modules to be included and configuration options (identified by documentation).

Contiki [urlb] is structured by a unclear mix of CPU-families and architectures, platforms and add-on devices. At the time of writing, there were a not exactly enumerable amount of CPUs (estimated 30+) from 7 architectures[9], 64 platforms and 9 devices. This information has been obtained by counting the respective directories or configuration files[10]. Configuration of an application is managed via setting variables or pre-processor directives in the Makefile or an included c-header file.

From the experience gained implementing several of the experiments within the operating systems, the more declarative approach used in **Zephyr** [urld] was very convenient in terms of configuration and dependency management. The included configuration tool (available using *"make menuconfig"*) listed all possible configurations options and manages dependencies. Opposed to that, the current configuration, available options or functional interdependence was often unclear when working with **Riot OS** [urlc] and **Contiki** [urlb].

Furthermore, during the work on **Zephyr** [urld], it has shown that the strong modularisation approach is conducive to platform development, reusability and combinability. In contrast to that, in **Contiki** [urlb] there were problems when trying to configure some options that had default values and were not meant to be changed. For both, **Riot OS** [urlc] and **Contiki** [urlb] the available configuration capabilities and options had to be extracted from either source or separately available documentation.

[5] *Kconfig.soc* for CPUs, *Kconfig.board* for boards, *Kconfig.** for drivers
[6] `https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt`
[7] arm, xtensa, x86, x86_64, 8051 variant
[8] directories for CPUs and boards, *Makefiles* for drivers
[9] avr, arm, 6502, 8051, MIPS M4K, x86, x86_64
[10] directories and Makefiles for CPUs, Makefiles for platforms, directories for devices

CHAPTER 7

Conclusion

This work showed differences and similarities in the three operating systems **Zephyr** [urld], **Riot OS** [urlc] and **Contiki** [urlb] in regard to various generalized performance criteria with a focus on wireless networks. Both, characteristics relevant to the selection of software as well as problems with some evaluation criteria and methods have become apparent.

In regard to the evaluation goals described in section 1.4, the comparatively young operating system **Zephyr** [urld] has shown to be mostly on par with its competitors for quantitative criteria, especially showing advantages for qualitative criteria (e.g., configuration, modularisation).

Riot OS [urlc] has performed very well in network performance, with some drawbacks in configuration and application implementation.

While **Contiki** [urlb] as the oldest competitor has a very rich and stable environment (e.g., network simulator, development virtual machine), it had small drawbacks in many areas (e.g., worst network performance, unclear modularisation structure, no first class support for platform independent interrupt handling).

The most important fact to take away from this work is: *general* comparison of operating systems is hard. This is due to a multitude of reasons, some of which are:

- Difference in (typical) use-case (e.g., hardware platforms, applications). Likely due to the age difference of the competitors, they seem to focus on slightly different hardware and applications.
- Difference in usage (e.g., configuration, modularisation, writing an application). This makes it hard to configure the competitors as "comparable". Experiment results might be straight out false if the configuration is incomplete or wrong.

- Difference in (default) configuration (e.g., network setup, buffer sizes, power saving). Certain defaults, originating in assumptions on use cases, can influence several evaluation criteria. Especially for networking performance, there are many different configurations that work, but perform very differently depending on various conditions such as the surrounding environment, the hardware platform and application.
- Difference in design (e.g., memory allocation, multitasking, modularisation). The way a functionality is intended to be used, described by an interface, can make a huge difference. For example, functionality to send data could either „consume“ or „copy“ given input. The earlier can be faster and more efficient by avoiding unnecessary copies, while the later might increase ease of use by providing an additional layer of abstraction.

While it does not seem impossible to find, document and conduct general comparisons, it needs even greater effort. Also, the criteria, methodology and methods should be discussed by a larger group, to avoid pitfalls and achieve a broader view.

This work concludes that while general comparisons give very good hints at whether or not an operating system **can** suite specific needs, application specific work is still very much required to find out if it actually **does**.

Glossary

application Possible or intended purpose of an embedded system (e.g., a motor speed controller), including the therefore relevant hardware (e.g., motor) and software (e.g., controller software). 1–4, 7, 9, 15, 16, 19–21, 25–28, 33–37

application software Software component (possibly running within an operating system) implementing the application. Also referred to as **application task**, **user task**, **user software** or **user program**. 3, 4, 18, 21, 22

compiler Software to convert textual program representation into machine executable representation. 15

embedded system A conglomerate of hardware and software components tightly coupled to its application. 1, 3, 4, 7, 14, 37

GPIO Multipurpose hardware that can be used as either input or output. Present in many microcontrollers. 3, 7, 10, 21, 22, 26, 37

Makefile Software configuration stored in a file, describing various steps to generate desired output. 34

mark-space ratio Ratio between the two states of an alternating digital signal. 23

microcontroller An integrated circuit that includes a processing unit (i.e., a microprocessor) as well as peripherals such as GPIOs or application specific hardware interfaces. 1, 4, 7, 11, 21, 24, 26, 37

microprocessor An integrated circuit that usually provides only processing capabilities and communication interfaces. 33, 37

network Set of components, either physically or virtually connected, able to exchange information with each other. 5, 7, 18, 20, 35

operating system Software that provides common services and functionality as an environment for other software to be built and run on. 1–5, 7, 10, 11, 14, 15, 17–22, 25, 26, 28, 33–38

OSI Network Model Characterization and standardization of communication functions in 7 layers. 18–20

real-time Terminology commonly used in the context of operating systems used to describe that a certain functionality (implemented in hardware or software) complies with defined timing constraints (as opposed to the popular misconception and misuse of the term for *something that is fast*). 7

Acronyms

ADC Analog Digital Conversion 13

CPU Central Processing Unit 2, 3, 18, 21, 24, 25, 27, 32–34

DSO Digital Storage Oscilloscope 12, 13, 21, 22, 26–28

GPIO General Purpose Input Output 18, 20, 21, 23, 27, 33

I2C Inter-Integrated Circuit 3, 11, 33

IoT Internet of Things 1, 2, 5, 7, 28

IPV6 Internet Protocol Version 6 18, 27, 30

MAC Media Access Control 19

MPU Memory Protection Unit 32, 33

PWM Pulse Width Modulation 3, 33

RAM Random Access Memory 2, 3, 11, 15, 16, 18, 26, 33

ROM Read Only Memory 2, 3, 11, 15, 16, 25, 26

SPI Serial Peripheral Interface 3, 11, 33

TCP Transfer Control Protocol 18, 19

U(S)ART Universal (Serial) Asynchronous Receiver Transmitter 3, 11, 20

UDP User Datagram Protocol 18, 19, 29–31

USB Universal Serial Bus 12

Bibliography

[70c07] Cortex M3 Technical Reference Manual. Technical report, ARM Infocenter, 6 2007.

[71c13] CC2538 System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee®/ZigBee IP® Applications - User's Guide. Technical reference, Texas Instruments, 5 2013.

[72s17] SmartRF06 Evaluation Board (EVM) - User's Guide. Technical reference, Texas Instruments, 5 2017.

[AC09] Rafael Vidal Aroca and Glauco Caurin. A real time operating systems (RTOS) comparison. *Sao Carlos, Brasil*, 2009.

[BHG+13] Emmanuel Baccelli, Oliver Hahm, Mesut Gunes, Matthias Wahlisch, and Thomas Schmidt. RIOT OS: Towards an OS for the Internet of Things. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference*, pages 79–80. IEEE, 2013.

[DGV04] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference*, pages 455–462. IEEE, 2004.

[DSVA06] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42. Acm, 2006.

[FG13] Hossam Mahmoud Ahmad Fahmy and Salma Ghoneim. Performance comparison of wireless networks over IPv6 and IPv4 under several operating systems. In *Electronics, Circuits, and Systems (ICECS), 2013 IEEE 20th International Conference*, pages 670–673. IEEE, 2013.

[LBS] Cheng-Han Lee, Salman Abdul Baset, and Henning Schulzrinne. TCP over UDP.

[LDMM+05] Benoît Latré, Pieter De Mil, Ingrid Moerman, Niek Van Dierdonck, Bart Dhoedt, and Piet Demeester. Maximum throughput and minimum delay in IEEE 802.15.4. In *International Conference on Mobile Ad-Hoc and Sensor Networks*, pages 866–876. Springer, 2005.

[TA09] Su Lim Tan and Tran Nguyen Bao Anh. Real-time operating system (RTOS) for small (16-bit) microcontroller. In *2009 IEEE 13th International Symposium on Consumer Electronics*, pages 1007–1011, May 2009.

[urla] CC2538DK. `http://www.ti.com/tool/cc2538dk/`. Accessed: 14.07.2017.

[urlb] Contiki. `http://www.contiki-os.org/`. Accessed: 13.07.2017.

[urlc] RIOT. `https://riot-os.org/`. Accessed: 13.07.2017.

[urld] Zephyr Project. `https://www.zephyrproject.org/`. Accessed: 13.07.2017.

[VDKGGT15] Erik Van Der Kouwe, Cristiano Giuffriday, Razvan Ghituletez, and Andrew Stuart Tanenbaum. A methodology to efficiently compare operating system stability. In *High Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium*, pages 93–100. IEEE, 2015.

[WSS09] Heiko Will, Kaspar Schleiser, and Jochen Schiller. A real-time kernel for wireless sensor networks employed in rescue scenarios. In *Local Computer Networks, 2009. LCN 2009. IEEE 34th Conference*, pages 834–841. IEEE, 2009.