Telematics
Computer Systems

Freie Universität Berlin

Computer Systems and Telematics — Distributed, Embedded Systems

Diploma Thesis

Virtualization of the RIOT Operating System

Ludwig Ortmann

Matr. 3914103

Supervisor: Dr. Emmanuel Baccelli
Assisting Supervisor: Prof. Dr.-Ing. Jochen Schiller

Institute of Computer Science, Freie Universität Berlin, Germany

March 2, 2015

I hereby declare to have written this thesis on my own. I have used no other literature and resources than the ones referenced. All text passages that are literal or logical copies from other publications have been marked accordingly. All figures and pictures have been created by me or their sources are referenced accordingly. This thesis has not been submitted in the same or a similar version to any other examination board.

Berlin, March 2, 2015

(Ludwig Ortmann)

Abstract

Abstract

Software developers in the growing field of the Internet of Things face many hurdles which arise from the limitations of embedded systems and wireless networking. The employment of hardware and network virtualization promises to allow developers to test and debug hardware independent code without being affected by these limitations. This thesis presents *RIOT native*, a hardware and network emulation implementation for the *RIOT* operating system, which enables developers to compile and run *RIOT* as a process in their host operating system. Running the operating system as a process allows for the use of debugging tools and techniques only available on desktop computers otherwise, the integration of common network analysis tools, and the emulation of arbitrary network topologies. By enabling the use of these tools and techniques for the development of software for distributed embedded systems, the hurdles they impose on the development process are significantly reduced.

Contents

List of Figures

List of Tables

Listings

Acronyms

6LoWPAN IPv6 over Low power Wireless Personal Area Networks. 2, 78

ADC analog-to-digital converter. 79

API application programming interface. ix, 5, 14, 16, 31, 33–36, 38, 39, 42–44, 46, 48, 49, 54, 57, 58, 73, 79

ATA AT Attachment. 27

BLE Bluetooth low energy. 75

CASE Computer Aided Software Engineering. 3, 4, 16, 21, 79

CPU central processing unit. 5, 8, 14, 15, 18, 20, 23, 25, 26, 33–35, 39, 52, 63–66, 71, 74, 75

DSL domain specific language. 3, 16

FIFO First In First Out. 38

FPU floating point unit. 74

GCC GNU Compiler Collection. 39, 50, 54, 67

GDB GNU Debugger. ix, 23, 59, 71, 77

GPIO general-purpose input/output. ix, xi, 48, 49, 67, 79

HIL Hardware in the Loop. 21

I^{2}C Inter-Integrated Circuit. 14, 79

IETF Internet Engineering Task Force. 7

IoT Internet of Things. v, vii, 2–9, 11, 13, 19, 24, 25, 27, 45, 58, 60, 73, 75, 79

IP Internet protocol. 2

ISA instruction set architecture. 26, 27, 57, 74

ISR interrupt service routine. 35, 38

JTAG Joint Test Action Group. 20

JVM Java Virtual Machine. 26

KiB kibibytes. 7

LED light emitting diode. 9, 34

LGPL GNU Lesser Public License. 6

MCU microcontroller unit. 20, 79

MMU memory management unit. 74

MTU maximum transfer unit. 75

NIC network interface controller. 27

OS operating system. v, 1, 6–9, 11, 14, 20, 23–27, 29, 31, 33, 34, 37, 39, 42, 43, 49, 54, 57, 58, 65, 78, 79

OSI Open Systems Interconnect. 42

OSS open-source software. 4, 31

PWM Pulse Width Modulation. 79

QEMU Quick EMUlator. 26

RAM random access memory. 7, 15, 18, 25, 65, 66, 73

ROM read only memory. 15, 18, 20

RTC realtime clock. ix, 34, 47, 48

SIL Software in the Loop. 21

SMLC Smart Manufacturing Leadership Coalition. 1

SPI Serial Peripheral Interface. 79

SUT system under test. 17, 21

TCP Transmission Control Protocol. 9, 76, 79

TDD test driven development. 29

UART Universal asynchronous receiver/transmitter. 9, 33, 34, 45–47, 78

UmL User-mode Linux. 26

USART universal synchronous/asynchronous receiver/transmitter. 45, 79

VGA video graphics adapter. 27

CHAPTER 1

Introduction

This thesis is about the virtualization of an operating system for embedded systems (*RIOT*) as a means of facilitating the problem of developing software for embedded systems in general and the Internet of Things in particular.

1.1 Motivation

1.1.1 The Rise of Embedded Systems

As of this writing computers have penetrated almost all aspects of everyday life. More and more we use them effortlessly, without even being aware that we are in fact interfacing with a computer. Furthermore, not a day passes, without someone somewhere creating a new smart object. Be it some hacker in a garage who equips her pet with a geospatially enabled camera to see what it does at night, or an engineer who builds a wristwatch that interfaces with the world wide web to provide its user with up-to-date weather forecasts. During the last couple of years the significance of networked, embedded computers has risen from research object or toy to an industry transforming technology. Examples of this are the German *Industrie 4.0* [1] or the American *Smart Manufacturing Leadership Coalition* [2].

1.1.2 The Complementary Worlds of Traditional and Embedded Systems

There are many differences between todays desktop, notebook or server computers and embedded systems. Size, energy consumption, and hardware costs being the driving factors, embedded systems usually have one thing in common: constrained hardware [3]. Despite the hardware industry's trend to downsize components, reduce costs, and gain performance per power in general, a less capable hardware platform will most likely be cheaper, more energy efficient, and smaller in the foreseeable future meaning Moore's law does not apply.

These low end devices can typically not run a generic operating systems (OSs) for less constrained hardware like for example *Linux*. Therefore, embedded systems need special software to cater to their reduced performance and capacity. Also, these devices will typically be much more useful when interconnected and especially when connected to the Internet.

The need for networking exists, because most things we want these smart objects to do depend on events they can not detect themselves (as the aforementioned weather forecast), or because we want them to report what they detect to a more powerful system (for example a supercomputer that creates weather forecasts). Also, in tune with the rise of *cloud computing*, lack of performance can be tackled by offloading work intensive tasks to some remote system.

1.1.3 The Internet of Things

The term *Internet* denotes the world wide interconnected network of computer networks that utilize the "Internet protocol suite". This family of communication protocols provides end-to-end communication for applications and handles every aspect of data transportation, including routing of data units. In order to become a useful part of the Internet, embedded devices need to be able to transparently communicate over Internet protocol (IP) [4]. Embedded systems are generally not powerful enough to run the traditional IP suite. Also, embedded devices typically communicate over wireless data links and often make use of a mesh communication structure to provide access to devices over a larger area. This means, that the devices need to use highly specialized routing protocols in order to cater to the peculiarities of the medium and the low power profile of the device. To accommodate this, there are three options: implementing only the relevant parts of the needed protocols, squashing and tuning them to the needs of the application at hand, adapting and tailoring the protocols instead and use gateways at the borders of the Internet that translate between the protocol dialects, or implementing some silo solution which translates between IP and the embedded device. All approaches are being actively pursued with software stacks uIP [5] as an example for the former, IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) [6] for an adaption layer, and ZigBee [7] for a silo solution. The whole package consisting of embedded devices and specialized software, especially communication protocols, that live at the "fringes" of the internet is called the IoT[1] [8].

1.1.4 The Caveats of Software Development for the IoT

Software development for the IoT has some unique properties.

Hardware

Due to the use of embedded hardware, the typical development cycle of writing and debugging code running on constrained hardware is depending on specialized software and hardware. While it is possible to use standard compilers and even debuggers, the functionality is often limited[2]. Also, additional hardware is necessary for the debugger to interface with the target. Furthermore, deployment (i.e. uploading of the compiled code to the target), is a task that is typically time consuming on an embedded device. Finally, the target hardware might not even exist, or is still undergoing changes, because the development of

[1] There are different definitions for the term *Internet of Things*. The one given above is used throughout this thesis.

[2] Compare: section 2.5.3 on page 20

software and hardware happen in parallel and depend on each other. In projects where this is the case, it is also likely that only a limited number of devices are available for testing during longer periods of the development process.

Wireless Networking

As IoT hardware usually employs wireless communication, all the difficulties that arise from the communication medium make debugging of network protocols even harder. In order to eavesdrop on the communication between two devices, a third device is needed. This might be some specialized hardware that has been created for this particular purpose, or another target device running software written specifically for that purpose. In either case, additional hardware and software needs to be bought and/or developed and maintained. On top of that, the properties of the medium, like increased packet loss[3], also affect the eavesdropping device.

Structured analysis of larger wireless scenarios in a testbed is especially expensive due to the need of space and maintenance. Even if every device is attached to a host system, so that it can be programmed in a manner that is not too error prone[4], unplugged, crashed, broken, or missing devices will need to be physically examined. This, along with the maintenance of the host systems (which are error prone as well), leads to many resources spent on maintaining the infrastructure.

All of these issues make the development of software for the IoT tedious and error prone in aspects that do not arise for the development of regular desktop or server software.

1.1.5 The Future of Embedded Software Development

In some industries, software development for embedded systems is already done with the help of *Computer Aided Software Engineering (CASE) tools*. These tools enable developers to define models for both, the system and the data it should process. The application implementation is then "derived" from these definitions[5].

Another approach is to define *domain specific languages*. These programming languages have a grammar that is tailored for a specific task at a high level, e.g. sensing their environment and communicating those data to remote systems. For these languages, all the difficulties that arise from general purpose languages such as C, interfacing with sensor drivers, using network protocols et cetera are shifted to the implementor of the domain specific language (DSL).

While both approaches will certainly be helpful at least for certain markets, there still remains the task of implementing the building blocks which are used by DSLs and CASE

[3] compared to wired communication media like Ethernet

[4] The alternative to programming a locally attached device is programming *over the air*. The target device can run special software that allows uploading a new firmware image, flash it to the permanent storage and reboot into it. This has two disadvantages: On one hand it needs extra memory and maintenance, on the other hand if the new firmware has a bug, another update might not be possible because the device becomes unresponsive or the update mechanism itself is broken.

[5] CASE tools also have support for other useful tasks, like defining and tracing requirements, verifying the model, testing the implementation and so on.

tools to generate code for the specification. Therefore, some developers will always have to cope with the problems outlined above.

Furthermore, while these tools are expensive and CASE tools also have high training costs, the applications, for example in the automotive industry, that are built with them have the highest standards of reliability. Therefore the high development costs are justified. This is not always the case for IoT applications. Smaller companies and startups might not be able to make the investments necessary to use these tools.

Finally, it is not certain whether these tools will ever be available as open-source software (OSS).

1.1.6 The Need for Open Source Software

The subject of free and open source software has been a controversy since its inception. However, the success of *Linux* shows that concerns regarding maturity of OSS are basically prejudices. The growing economy around *Linux* also shows that viable OSS business models exist. Concerns regarding security of OSS platforms have been refuted by the numerous security flaws that are discovered in proprietary commercial software on a daily basis. At the end of the day, it does not seem like there is much of difference between OSS and proprietary software per se. However, this would ignore the defining aspect of both worlds, that is open versus proprietary.

The possibility to study source code enables trust. In order for a revolutionary technology like the IoT to gain traction, trust is needed. Trust, however, is a scarce resource at the moment. In the aftermath of the politically and privately driven privacy disasters of recent, the idea of putting internet enabled devices into everyday objects is often associated with the fear of an omnipresent surveillance machine. Therefore, without trust, there will be no IoT, at least not on a conscious and voluntary basis.

To conclude: in order for the IoT to happen trust is needed, and OSS has the major benefit of providing trust.

1.1.7 The Problem at Hand

Software development for the IoT presents many challenges that arise from limitations of embedded devices and particularities of wireless communication. In order to increase developers effectiveness, a means to reduce these hurdles is needed.

The problems specific to IoT development do not arise in traditional software development[6] due to both, the reliability and availability of the target hardware[7], and the availability of considerably more advanced debugging software. The availability of such debugging software is dependent on powerful hardware and certain hardware debugging features, as well as the software environment and therefore generally not feasible on embedded hardware. When it comes to network applications, the challenges lie largely in the communication medium, and

[6] Usually there are also different requirements, especially for reliability, but these are generally independent from the problems outlined above.

[7] Software for desktop or server systems can typically be tested on the same machine that it is developed on.

can only be addressed by elaborate testbeds.

1.1.8 Compatibility of Software for Embedded with Desktop Systems

Looking more closely at the software that is being developed for embedded devices, only part of the code is actually depending on specific hardware features[8]. Development of more energy conserving communication protocols for example is not depending on a battery being used to power the device, nor does a smaller code size depend on a less powerful central processing unit (CPU). Many parts of a typical IoT software stack can run on a regular desktop computer as well as on an embedded device. The only parts that actually depend on the target device are device drivers. All that is needed for any development that does not directly interact with hardware is the software environment, i.e. the same application programming interface (API), that will be present on the target device.

The same applies to network applications. While network interface drivers rely on hardware, anything that comes on top of them does not. As long as the payload that is written to the device arrives at some destination, it does not really matter if it travelled through air or over wires, and how it has been encoded in the meantime. The medium does have certain characteristics such as data rate, probability of packet loss and leads to the use of different topologies. As these characteristics are typically "worse" on IoT devices compared to desktop computers, and protocols that work on "bad" links also work on "good" links, but not necessarily vice versa, the use of IoT protocols on desktop computers is generally possible.

1.1.9 Facilitating the Problems of Software Development for Embedded Systems

In order to alleviate the problems with embedded software development and gain the advantages of traditional software development tools, the implementation of a virtual hardware platform that enables the developer to test and debug embedded applications in much the same manner as traditional software is proposed.

In [9], the concept of *dual-targeting* is outlined as a means to isolate software from hardware development[9]. The authors added macros[10] to their source code to deactivate any functionality that would actually communicate with the hardware[11] and were thus able to run the application on their desktop computers.

Running an application on a traditional development computer system instead of on the targeted embedded device brings all the benefits of contemporary software development to the development process for IoT applications. Standard tools for debugging, profiling and analyzing software can be used to assist with the development of applications and large parts of the software environment.

[8] Figure 4.1 on page 59 suggests a system dependent to independent code ratio of between 1 : 4 and 1 : 10 for any given platform.

[9] The ramifications of this approach are explored in more depth in [10].

[10] A macro is a command that is evaluated when the source code is translated into binary form. Using macros, it is possible to change the resulting program during compilation without having to modify the source code.

[11] When data was to be read from the hardware they inserted dummy data instead.

For the IoT, it is essential to be able to debug network applications. In order to debug network protocols on embedded devices it is necessary to employ specialized tools. On one hand, some form of network environment setup is needed for experimentation with network topologies, on the other hand a dedicated traffic analyzer[12] is needed. Virtual networking enables both, scalable testbeds without additional costs for space and maintenance requirements, and the seamless use of standard traffic analyzers like *Wireshark* [11].

As contemporary desktop computers are orders of magnitudes more powerful than IoT devices, a single development machine can be used to host vast amounts of IoT application instances. Additionally the network properties can be tuned to specific needs and reliable monitoring is already available as the traffic does not even leave the host machine. Finally, virtual networking allows for reproducible results because the environment can be controlled[13].

1.1.10 The Software Environment

One of the biggest achievements of software engineering is *reuse*, i.e. the identification and separation of code that is used in more than one place. Reuse is enabled by abstraction and generalization which in itself already leads to software with higher modularity, cleaner interfaces and better maintainability. A result of reuse is often an increased code size for a particular module compared to a custom implementation, as the reusable module often has a larger scope than what is needed by any particular software that uses it.

There are primarily two kinds of software that are written primarily for reuse: Software that is written only for reuse and is capable of running on its own is called a library. Libraries are complemented by operating systems, which also serve as a reusable abstraction of functionality, i.e. hardware access, resource sharing and so on, but are able to run on their own. Indeed, today's software applications are typically depending on an OS, and are not actually able to run on a bare computer.

For microcontrollers, the situation is a little different today. While software libraries are also used, off the shelf OS' are not the standard. The reason for this lies in the relatively low complexity of the applications, high dependency on hardware, and the overhead that comes with an OS. Nonetheless, a couple of OS' for microcontrollers have been developed in the recent years. Examples include *Contiki* [12], *TinyOS* [13], and *FreeRTOS* [14].

Along with the IoT, the relevance of OS' for microcontrollers is also increasing. In order to be able to communicate with the Internet, a certain amount of software, namely a network stack, is needed. As network stacks are a fairly complex pieces of software, reimplementing them over and over again is not feasible. Also, networked applications can easily become more complex due to their asynchronous distributed nature.

RIOT [15] is an OS for the IoT that caters to both, research and engineering needs which is distributed under the *GNU Lesser Public License* [16]. Research is an important factor in the relatively young and quickly evolving field of IoT. In fact *RIOT* has primarily been developed as a research platform at Freie Universität Berlin, INRIA Saclay [17] and HAW

[12] A traffic analyzer has the ability to decode network protocols. It can help find errors in both, the online packet format, and the communication pattern.

[13] Compare section 2.8 on page 28

Hamburg. *RIOT*'s roots go back to an industry product though, and its features can be traced back to these roots. Comprised of a modular micro kernel and a scheduler with real time properties, it is one of the smallest and fastest systems in the market. Due to its use as a research tool, new internet protocols are continuously being developed on it. This makes *RIOT* attractive, not only for research on top of this, but also for companies looking for readily available software stacks in the emerging IoT market and is why it was chosen as a base for this thesis.

1.2 Terminology

As outlined in the motivation, *RIOT* aims at computers powerful enough to run a tiny OS, yet not powerful enough to run one like Linux or Windows. In [3, section 3] the Internet Engineering Task Force (IETF) defines classes for "constrained devices". This thesis is written with "Class 1" and "Class 2" devices in mind, that is devices ranging from 10 to 50 kibibytes (KiB) of random access memory (RAM) and 100 to 500 KiB of flash memory. Throughout this thesis I use the words "embedded system" to denote such a device.

The term 'traditional' or 'regular operating system' denotes operating systems such as Linux, OS X or Windows, 'traditional' or 'regular computer system' denotes a computer system that is powerful enough to run this kind of software.

I write of 'traditional' or 'regular software development' and 'software development for embedded systems' to refer the software development processes for 'regular computer systems' that run 'regular operating systems', and 'embedded devices' that can not run 'regular operating systems' respectively.

The development processes for embedded systems that are powerful enough to run 'regular operating systems' might fall into one category or another depending on the project at hand. This is not a concern for this thesis however.

The reasons for this divide are analyzed in section 2.3 on page 15. The bottom line is that class 1 constrained devices are not only constrained in memory and processing power which hinders certain development tools and processes, but also offer only limited debugging features.

When talking about software, I use the following terms based on [18]:

- Behavior:
  A systems internal and external changes in state given an initial internal and external state. For real-time systems, the time it takes to change from one state to another is part of the behavior[14].
- Correctness:
  The systems adherence to intended behavior.
- Defect:
  An error in the systems implementation that can cause a failure.
- Error:
  The violation of a systems correctness.

[14] Time can be regarded as an external state.

- Failure:
  The visible manifestation of an error.
- Infection:
  The result of a defect in the running system, possibly leading to a failure.
- Specification:
  A written version of the intended behavior of a system.

1.3 Goals for this Thesis

Following the observations made in section 1.1, the following goals can be defined to help mitigate the particularities embedded systems have on software development:

- *Virtualization Support*
  By providing a virtual development board, costs for acquisition and maintenance of hardware are reduced.
- *Development Tools Support*
  The virtual platform should enable development tools that are not available on embedded devices and allow for regular use of tools which only offer limited functionality on embedded devices.
- *Speedup of Development Processes*
  The virtual platform should be quicker to use than embedded systems. Waiting times for hardware should be eliminated.
- *Virtual Network Analysis and Testbed Support*
  In order to enable structured testing of network protocols, support for various virtual networks should be implemented. For debugging purposes, the use of network analysis tools should be enabled.

The outcome of this thesis will not only help increase research productivity but also raise *RIOT*'s attractiveness for industry and private use by lowering the hurdles of development and increasing the usefulness for research.

1.4 Contribution

A hardware virtualizer called *native* has been added to the *RIOT* OS. It allows for the compilation and execution of applications based on *RIOT* as user processes in Linux, FreeBSD and Mac OS X. This virtual platform does not only enable development for and of *RIOT* without the need for actual hardware, but also helps mitigating the problems of debugging on IoT devices.

1.4.1 The Emulator Platform

In order to minimize execution overhead and to speed up the development process, the native platform has been implemented as call level emulator. It provides a board and CPU

featuring timers, a Universal asynchronous receiver/transmitter (UART), network interface, and example sensor and actuator implementations in the form of an energy meter and real-time clock, and light emitting diodes (LEDs). To support large-scale experimentation, virtual *RIOT* applications can be run as daemons. The virtual UART is accessible via Transmission Control Protocol (TCP) and UNIX sockets in the host OS.

1.4.2 A Development Methodology for the IoT

Based on experiences with software development for the IoT and traditional systems, a methodology has been formalized. It addresses the problems specific to the IoT software development domain in open source projects.

1.4.3 Improving the Development Process

By eliminating the need to copy images to flash memory, speedups of between 4 and 60 compared to *msb-430* and *chronos* have been achieved.

By adding support for the use of *Valgrind's* [19] memory debugger *memcheck*, performance profilers like *gprof* [20], and compiler options like *stack smashing protection* the development process has been significantly improved.

Furthermore, the method of running hardware independent code natively on a desktop computer allows for taking better advantage of the debugger.

1.4.4 Virtual Networking Support

Support for network emulation helps overcome typical problems of testing network protocols with wireless hardware. By making use of *tap* devices the use of existing tools has been enabled.

Debugging of network protocols is possible through the use of tools such as *Wireshark* [11]. The possibility to use the *DES-Virt* [21] network emulation framework allows for structured testing of arbitrary network topologies with well-defined characteristics.

1.4.5 In-Process Preemptive Threading

While preemptive scheduling is a natural part of contemporary OS', it is not available for threads within user processes. Due to the preemptive nature of *RIOT*'s scheduler, a method for preemptive threading in the user space had to be designed.

CHAPTER 2

Background and Related Work

In this chapter, the background and goals for the thesis that were outlined in 1.3 are analyzed in more depth. First, the structure and goals of the open source project *RIOT* are analyzed to shape the environment in which this thesis is developed. From this, some basic requirements for the implementation of this thesis are derived.

One of the project goals is high software quality and ease of use. The next section analyzes this goal and outlines existing methods and technologies that are used to achieve it.

Next, existing tools and technologies are introduced and examined according to the project goals and their role in software development for embedded systems. An attempt is made to also touch on technologies and methods that are not part of this thesis in order to help define the scope.

Finally, some general considerations for the implementation architecture are undertaken.

2.1 The Open Source Project

RIOT is an open source project consisting of a growing number of contributers who collaborate to write an OS for the IoT.

2.1.1 Community

The open source aspect of the project has several implications for the development process. For one, the contributors are a heterogeneous and growing group. According to the project history[1], about 54 people have contributed to *RIOT* since the project was created in September 2010. For 2014, [22] counts a monthly average of 16.6 contributors.

The contributors come from various backgrounds, such as *university and research institutes*, *commercial enterprises*, but there are also *private persons*. The individuals reside in several

[1] The output of the command git shortlog −−no−merges −s has been cleaned of duplicates and apparently nonsensical designations. The actual number of contributors may differ but is expected to be close.

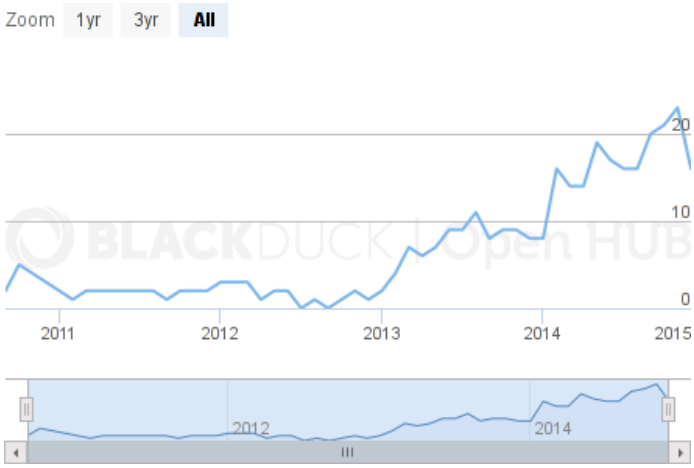

Figure 2.1.: Number of *RIOT* contributors per month
Source: [22] (Copyright 2015 Black Duck Software, Inc., Creative Commons Attribution 3.0 Unported License ([23]))

different countries spanning the entire globe. As of this writing, the highest concentrations of project members are at *Freie Universität Berlin*, *HAW Hamburg*, and *INRIA*, all of which use *RIOT* in research contexts. The distribution can be credited to the project's roots in the *FireKernel* [24] which was developed at *Freie Universität Berlin*. and led to the initiation of *RIOT* in cooperation with *INRIA* as part of the *SAFEST* research project [25].

Project members communicate via the collaboration platform GitHub [26], mailing lists [27], internet relay chat [28], video conferences, and during physical meetings in locations with a high number of contributors.

2.1.2 Project Goals

The projects website mentions several of its goals [15], including *ease of access* and *code quality*, as well as *thorough testing*. Another important aspect is *free software*. While *RIOT* itself is licensed under the *LGPL 2.1* license, all of the tools and libraries needed for its development and maintenance are open source as well. This fact reflects the tendency of the *open source movement* to prefer open to proprietary software[2]. The importance of an open ecosystem is also stressed in the project's vision declaration [29].

2.1.3 Development Process

To make sure the goal of high software quality is met, the project has adopted a specific development process [30]. Part of this process is a peer review system based on *Pull Requests* [31] that aims to make sure, every contribution is reviewed by at least one person in addition to the author. Additionally, a test suite is automatically executed which makes sure the project always compiles and does not break existing tests.

[2] Notable exceptions include the collaboration platform GitHub which is free to use for open source projects but is not open source software itself, and the proprietary video conferencing software *PlaceCam* which is free to use for *RIOT* due to sponsoring by its authors.

2.1.4 Implications

The community structure and the project's goals lead to certain requirements for methodologies and technologies used within the project. For one, any tools need to be free software, not only because of a belief in the benefits of openness [32], but also because of the costs of commercial software. Due to the projects structure and size, it can not easily provide access to commercial tools for all its members, especially because of a relatively high fluctuation. Additionally, one can expect free software developers to be familiar with the tools commonly used in free software projects.

When first appearing in the community, contributors typically have access to none or only very few of the supported devices. While this sometimes changes over time, no contributors can be expected to have access to a majority of the supported devices. The typical *RIOT* contributor does not even have access to at least one device of all supported architectures[3]. As of this writing, *RIOT* supports 34 different boards, and this number will most likely steadily increase over time. This leads to a situation where no individual developer can test all possible devices locally, and no two developers can be expected to even have one device in common.

In order for an open source IoT project such as *RIOT* to achieve their goals, it is necessary to address the physical and financial restrictions of both, hardware and software availability. A versatile, readily available platform for testing and development is needed as a reference.

In the case of *RIOT*, an important goal is high software quality. In order to allow for rigorous testing of software, the project can not rely on hardware alone because of both, technical limitations of IoT hardware (compare section 2.5) and the availability problem.

2.2 Software Quality

To address the project's goal of high quality and well tested code, it is necessary to define what this means. In general, software is tested for functionality and performance. Code quality usually refers to the structure and style of the source code. Together, these properties describe common functional and non-functional goals of software quality as outlined below.

For the context of this thesis, certain classes of problems which can be tackled with software development tools have been identified.

2.2.1 Quality Characteristics

In order to meet the functional and non-functional requirements of the project, certain code characteristics need to be focussed on.

The following list is by no means exhaustive, it merely covers certain properties that have been identified as generic and useful in the context of *RIOT*. They are based on [33] and tailored for this thesis.

- Functionality

[3] The term architecture in this context distinguishes between the processor families *ARM7*, *Cortex M0*, *Cortex M3*, *Cortex M4*, *MSP430*, and *AVR8*.

  - Network Interoperability – not part of [33], but gravely important for this project
  - Circuit Interoperability – same as above
- Reliability
- Efficiency
- Maintainability
  - Syntactic Correctness
- Portability – this goal is at the core of the project and needs no further attention
- Usability – same as above

In order to fulfil the project goals, each of these characteristics needs to be met.

2.2.2 Functionality

Functional properties of software describe its intended purpose, the function it should perform. For an OS such as *RIOT*, which itself is a building block for other applications, the specification is limited to its APIs[4].

Network Interoperability

Verifying network protocol implementations is different from verifying functional correctness because the specification is usually completely external to the project which implements it, highly complex, and not easy to test without external tools. Errors in byte order for example will only become a problem when communicating with a system that uses a different byte order locally. Also, especially in the case of wireless communication, problems of the medium may lead to inconclusive results as described in section 2.5.3.

Circuit Interoperability

Quite analogous to network interoperability, protocols for communication between the CPU and peripheral devices like Inter-Integrated Circuit (I^2C) must be verified to work with third party tools.

2.2.3 Reliability

When a certain program might perform its intended task well a couple of times, but fails at some point, it is unreliable. The reasons for such failures are principally various, but in this context only defects in the program semantic are of interest. They result from failures in input sanitation or bounds checking, use of uninitialized variables, neglected synchronization, access to invalid memory, or deadlocks in threaded applications among

[4] Although *RIOT* has no specification in the meaning of the word as used in the embedded system industry (describing timings, electrical properties, all possible inputs/outputs, ...), there usually is documentation that describes what each function does, i.e. a functional specification of the APIs.

others. Because the target hardware generally lacks memory protection, it is usually not possible to recover from memory access failures on embedded systems.

2.2.4 Efficiency

Although it is not an outspoken goal of *RIOT* to excel in speed, for example of network throughput or memory usage, the targeted domain of microcontrollers dictates certain bounds for these metrics. Because it is usually considered expensive, minimizing hardware is one of the main concerns for embedded systems. Due to this, it is necessary to limit the usage of RAM, read only memory (ROM), and CPU.

2.2.5 Maintainability

As source code is mostly not written once and then forgotten, it is important to make sure it can be understood by other human beings. Modularity, comments, and stylistic coherency play important roles in this regard.

Syntactic Correctness

When writing software manually it is possible to write syntactically incorrect code. Usually this becomes obvious immediately because the program can not be compiled. However, a modular architecture where only a subset of the existing code is ever translated at one time, syntax errors can slip through because they are never compiled. As *RIOT* is highly modular, syntactic correctness is non-trivial to achieve.

2.2.6 Assuring Software Quality

In order to fulfil the above list of source code characteristics, it would be foolish to rely on human prudence and skill alone. Luckily, various tools and methods have been developed which can help in the strive for high code quality.

2.3 Development Tools for Quality Assurance

In order to ensure quality, there are two principal approaches: *constructive* and *analytical* quality assurance. For both approaches, tools exist that help in their execution. In this section, both approaches are introduced and evaluated for their fitness in the context of an open source project such as *RIOT*.

Constructive tools are used to generate code from a specification. As long as the specification is sound, the product should be free of errors. Analytical tools on the other hand work on an existing product (given either in source or binary form) and help to check it for certain properties. While the former ensure that the implementation is free from *defects*[5], the latter can help in both, determining *failures* and finding the *defect* that causes them.

[5] Obviously, constructive tools can themselves have defects in which case they might produce erroneous code.

2.3.1 Constructive

At the time of this writing, there are two kinds of constructive technologies: *domain specific languages* and *code generators* that come with *Computer Aided Software Engineering* tools. While the use of such tools to generate code is a promising approach to eliminate many sources of errors, it is not feasible for a project such as *RIOT*. One reason is that these tools are not currently available as free software. Also, their use requires special training while *RIOT* aims to be accessible to the wider range of software developers familiar with C. An open source project such as *RIOT* can neither finance licenses for commercial tools, nor can it offer training for people who want to contribute. Furthermore, the hurdle introduced by such tools would discourage many potential contributors and basically prevents one-time contributors.

2.3.2 Conclusion

Due to the lack of constructive software development tools implemented as free software, they are not a topic of this thesis. In contrast to constructive tools, there exists a wide variety of open source tools for analytical quality assurance. In the following section, a closer look is taken at existing analytical software development tools.

2.4 Analytical Software Development Tools

In contrast to constructive software development tools, analytical tools are used on existing software. Obviously, this has the disadvantage of requiring a human to write potentially erroneous software first. As of today, it is standard practice however, to write code by hand, and most developers are acquainted with this process.

In order to see which tools and technologies exist and what they are used for, the commonly known technologies for each of the quality properties motivated in section **??** are given.

2.4.1 Functionality

For the verification of software there are basically two methodologies: *automated structured testing* and formal methods, notably *model checking*. While testing is the most common methodology today, it can typically not be used to prove the correctness of a system. Testing usually only shows that a system is correct for a very limited amount of inputs. Model checking on the other hand determines the correctness of a system by verifying it for every possible input.

In contrast to testing, model checking is a far more complex method. Because of this, model checking, if used at all, is only applied to parts of a system.

Code coverage tools will help in determining the completeness of a test suite. A popular method for achieving maximum code coverage is test driven development, where developers write tests first and the implementation last.

To identify misuse of APIs and "code smells", i.e. bad coding patterns, linters exist. They come with sets of rules which identify for example failures to initialize variables that are

passed to functions which are known to read them.

Network Interoperability

Again, testing is the most important method for obtaining this goal. However, unit tests can usually not be employed for this property due to the complexity of the task. Also, it is necessary to make sure the messages are actually sent out at the electric level. Instead, integration tests and load tests are employed.

In addition to tests *network analyzers* can be used to check the syntactic correctness of the network traffic. By analyzing traffic patterns, semantic problems can also be spotted more easily.

Circuit Interoperability

Like network protocols, communication protocols for peripheral devices can not be tested properly. *Logic analyzers*, like network analyzers, are specialized tools that know certain protocols. They help spot malformed messages and allow for analysis of the communication pattern.

In contrast to network analyzers, they need to be connected to the system under test (SUT) electronically.

2.4.2 Reliability

Memory debuggers are tools which track how memory is accessed and incorporate knowledge about either, the semantic of the memory, i.e. which memory location belongs to which identifiers, or they track the memory's history and context to make sure every memory read and write is possible.

Linters and other static analyzers like compilers will also try to make sure no undefined behavior can happen in that they analyze the source code for invalid memory access. This only has a limited accuracy because it is possible that for example memory offsets are part of the input to the program, i.e. not part of the source code and therefore unknown to the static analyzer.

Fuzzers try to fill the gap between formal analysis and testing by feeding (partly) randomized input to applications over an interface of choice. Interfaces in this context are network interfaces, function calls, command line parameters etc.

Most importantly, reliability can only be achieved if functional correctness is maintained for all possible inputs. This means that *testing* needs to take care of corner cases or *formal verification* needs to be performed.

2.4.3 Efficiency

There are different tools for each of the resources which can be optimized for.

Performance profilers and code coverage tools identify source code lines that are executed most often, i.e. use most of the CPU time. *Memory profilers* track the usage of RAM and determine source code lines that lead to higher memory consumption. *Object file analyzers* can identify how much space each symbol needs, so they can be used to identify ROM usage.

Additionally, *compilers* offer function instrumentation, allowing for custom profiling by developers.

2.4.4 Maintainability

Code metric analyzers are tools that analyze source code for statistically to determine whether they are correspond to known proportions for good modularization, documentation, cohesion, etc.

By checking for bad coding patterns *linters* also help in preventing unmaintainable code.

Syntactic Correctness

To check the syntactic correctness of software, it is generally enough to translate it because *compilers* need to know the correct syntax of the languages they translate.

Due to a complex structure with conditional translation such as is the case for *RIOT* and other highly modular software systems, special care needs to be taken of covering every line of code.

One possibility is identifying each macro creating a build matrix making sure all possible combinations are compiled at least once.

This is a task that can also be fulfilled by a *code coverage* program. Their purpose is to identify which lines are executed and how often if at all. In reverse, by filtering out lines which are not executed at all, it is possible to identify ones that still need translation. Building from there, one can make sure they will be translated (and executed).

2.4.5 Methods and Tools

To summarize, this is the list of quality characteristics and their respective methods and tools:

- Functional Correctness:
  Linter, Automated Structured Testing, Model Checking
  - Network Interoperability:
    Automated Structured Testing, Network Analyzer
  - Circuit Interoperability:
    Automated Structured Testing, Logic Analyzer
- Reliability:
  Linter, Memory Debugger, Fuzzer, Automated Structured Testing
- Efficiency:
  Memory Profiler, Performance Profiler, Object file analyzer

- Maintainability:
  Linter, Code Metric Analyzer
  - Syntactic Correctness:
    Compiler

2.4.6 Preliminary Conclusion

In order to achieve its goals, and open source IoT project like *RIOT* needs support for most if not all of the above tools and methods.

2.5 Tools for IoT Analysis

In order to figure out what tools are missing for IoT software development, the tools that have been identified as necessary are examined for their fitness for embedded software development.

Analytical tools can be categorized into *static* and *dynamic* tools. Additionally, there are formal methods, notably *model checking*, which can also be applied to existing code.

2.5.1 Model Checking

Model checking is generally not done with complete system implementations. To reduce the number of possible states, it is typical for a model checker to only analyze part of a system. Furthermore the models are often generated or written in a dedicated language only for the purpose of running the model checker. There are model checkers that can analyze C code as well [34], however the limitations for model checking in general also apply to them. In any case, a model checker would not just run the whole software on the target hardware, but build a verifier for selected parts of the code and run this on a powerful desktop computer.

While model checking is entirely possible for embedded systems, it is very intensive in work and training.

2.5.2 Static Analysis

The following tools perform static analysis:

- Compiler
- Linter
- Code Metric Analyzer
- Object File Analyzer

As static analysis tools do not need to inspect the software in execution, it does not really matter whether they are used for regular software or embedded software - source code is source code. One notable exception lies in the languages these tools analyze. In contrast to application software code, system software code usually has some portion of code that

needs to be written in assembly language. At the time of this writing, there are no static analyzers for assembly code. In this regard, system software tends to have higher amount of code that can not be subjected to static analysis.

2.5.3 Dynamic Analysis

The remaining tools fall into the category of dynamic analyzers. They are:

- Debugger
- Memory Debugger
- Memory Profiler
- Performance Profiler
- Automated Structured Tests
- Network Analyzer
- Logic Analyzer

Although not listed before, the *debugger* has been included here because it is usually very helpful in finding defects once an error has been found.

Debugger

In order to use a debugger with an embedded device, specialized hardware is necessary. Because embedded devices are not that powerful and capable, the debugger is split into a front-end running on a desktop computer, and a Joint Test Action Group (JTAG) or similar interface to connect to the *debug port* on the target microcontroller unit (MCU). While modern development boards sometimes provide integrated debugging facilities, it is still more common that additional (expensive) devices are needed for that purpose. Debugging embedded devices also often has qualitative deficits compared to debugging software on a regular desktop computer [35]: In order to set breakpoints, a debugger needs to either change the running software, or rely on CPU support for breakpoints. Because embedded CPUs tend to have only few breakpoints and the software might be in ROM, only a limited number or even no breakpoints might be available. The same goes for watchpoints, which may either be available in software or hardware. Depending on this, breakpoints and watchpoints might not work at all or be too slow to use.

Memory Debugger, Memory Profiler, and Performance Profiler

The family of automatic profiling, and analysis tools is principally not available on embedded systems, because they need a host OS, powerful CPU, and larger quantities of memory to work. Compare section 2.6.2 for reference.

It is possible to implement custom performance or memory profilers for embedded devices. When using the −finstrument−functions compiler flag, the compiler calls a set of user provided functions every time it enters or leaves a function. The analysis function receives the address of the instrumented function as an argument. By logging time, memory usage, or power consumption, along with the function that was called, basic profiling can be

achieved even on constrained devices, provided the overhead is not too great to prevent regular operation

Automated Structured Tests

While automated testing is possible with embedded hardware, it does not scale well. As shown in section 4.3.2, deployment . This is due to a high linear overhead for deploying the compiled tests (as shown in section 4.3.2) to the devices and their relatively slow execution. Parallelization is costly due to hardware costs and maintenance.

An additional difficulty is found in the task of embedded systems which typically includes reading writing analogue or logic values on the *MCU* pins in a timely manner. Due to this, the only viable method to systematically test the runtime behavior of an embedded device is *Hardware in the Loop testing*[6]. Hardware in the Loop (HIL) testing means, that the SUT is connected to a specialized test device that interfaces with the system by controlling the inputs and measuring time and outputs. This test system is running tests by checking if the outputs are set to the correct values depending on the inputs within the defined time constraints. In order to determine the inputs, the environment of the SUT is simulated based on models. The models characteristics and the correct outputs will typically be derived from the specifications using CASE tools. HIL testing has the advantage of being very reliable and the disadvantage of being very costly in hardware as well as time.

Network Analyzer

When looking at network analysis for embedded systems, one needs to differentiate between wireless and wired networks. For Ethernet based networks, it is possible to simply use a desktop computer in the usual fashion. In the case of wireless networks, it becomes much more difficult.

The testing of wireless network functionality is problematic because the environment is hard to control. The quality of the medium is more or less unpredictable and prone to fluctuations. Due to this, test results are unreliable or even unreproducible. With larger testbeds, this characteristic becomes ever more relevant. This conflicts with the necessity of structured testing to be repeatable.

Regarding packet capturing, it is necessary to add dedicated capturing devices. While there are specialized hardware network traffic analyzers for wireless communication, it is also possible to use the target hardware for this task. For software based capturing, the capturing software needs to be written (and tested). If available at all, off-the-shelf-hardware is usually rather expensive.

Logic Analyzer

For semi-manual analysis of a SUTs state on the signal level, there exist *logic analyzers*. By sampling the currents of arbitrary lines on the system and deriving patterns from it,

[6]There also exist other *in the loop* techniques (e.g. *model in the loop*, *Software in the Loop*, *processor in the loop*), all of which share the simulated environment as a defining characteristic.

the logic analyzer can "see" what the system really does. The ability to interpret e.g. bus protocols makes them invaluable when debugging drivers etc.

Although they are expensive, logic analyzers allow for achieving circuit interoperability relatively easy.

2.5.4 Conclusion

When it comes to embedded devices, one can often answer whether a certain type of analysis tool is readily available merely by looking whether they are static or dynamic tools. Figure 2.2 shows which tools are available with or without additional hardware on embedded systems and on regular workstations.

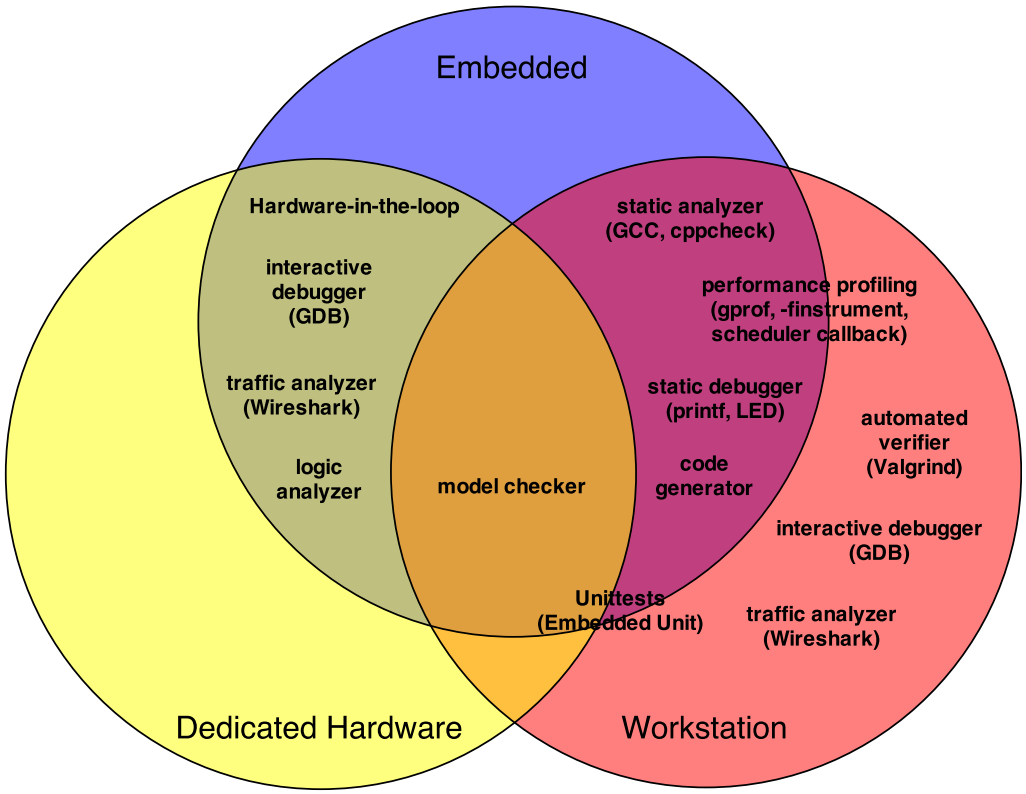

Figure 2.2.: Availability of Software Development Tools.
Tools that are not clearly in one category or another are overlapping category borders.

Due to this segregation, whole classes of problems can not be analyzed in an effective manner on embedded systems. This is one of the reasons why developing software for these devices is often considered hard. The need for runtime analysis arises from the problem classes that they are used for: the determination of how a system actually behaves under certain inputs.

The reason why runtime analysis is often not feasible on embedded hardware is the limitation of the devices. On one hand there is too much overhead involved in the gathering of information, on the other hand the tools themselves are too large to even fit on such

constrained devices. Also, due to their complexity, the runtime overhead due to live evaluation is too high for them to be used effectively. Finally, the target might lack hardware features required for analysis (e.g. breakpoints), or the data which needs to be analysed is not accessible by the system (e.g. wireless network traffic).

2.6 Tools for Desktop Systems

In this section, a closer look is taken at the concrete tools that are available on desktop systems only, or differ significantly when used on desktop hardware.

2.6.1 GDB – Debugger

The GNU Debugger [36] is the de-facto standard debugger for regular software as well as embedded devices [35, Chapter 16]. When run on desktop hardware, it offers a wide range of advanced features like virtually unlimited conditional breakpoints, catchpoints, tracepoints and watchpoints, as well as the ability to record and replay execution.

2.6.2 Valgrind memcheck – Memory Debugger

The Valgrind memory debugger *memcheck* is a specialised tool for investigating defects that arise from misuse of memory. As this is one of the most common errors in software written in C, a good memory debugger is invaluable. Valgrind[37] supervises every memory access and checks its validity. Both reading as well as writing accesses are monitored. When an invalid memory location is accessed, Valgrind prints out a warning as shown in figure 2.3. It can also stop the process, start the GDB debugger and attach it to its process. To achieve this, Valgrind dynamically recompiles the program under surveillance to an intermediate language. The memcheck tool then inserts monitoring code around instructions in order to maintain a log of all memory operations. Valgrind runs on Linux and OS X on x86 as well as Linux on ARM and other architectures[7]. As Valgrind needs to run in a host OS, it is not possible to use it to check the OS on an embedded device.

2.6.3 gprof and Valgrind – Memory and Performance Profiling

The gprof profiler [20] was a de-facto standard tool when optimizing the performance of software. While it was ground breaking in the area and is still in use today, it has certain shortcomings.

One popular alternative are the Valgrind tools cachegrind [38] and callgrind [38]. While cachegrind is a specialized for analyzing performance with regard to CPU caches it also yields general profiling information. The more advanced callgrind, which builds on cachegrind, but additionally provides callgraphs. As they are Valgrind tools the same limitation as for *memcheck* arise - all of these tools need to be run inside an existing OS.

[7] http://valgrind.org/info/platforms.html

```
...
==3679== Conditional jump or move depends on uninitialised value(s)
==3679==    at 0x402DEDF: bcmp (in
/usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==3679==    by 0x805246F: ipv6_net_if_addr_match (ip.c:523)
==3679==    by 0x805223A: ipv6_net_if_add_addr (ip.c:460)
==3679==    by 0x80559C7: sixlowpan_lowpan_init_interface
(lowpan.c:1737)
==3679==    by 0x805B839: rpl_init (rpl.c:212)
==3679==    by 0x804E816: rpl_udp_init (rpl.c:70)
==3679==    by 0x804F218: handle_input_line (shell.c:163)
==3679==    by 0x804F365: shell_run (shell.c:220)
==3679==    by 0x804E031: main (main.c:55)
==3679==  Uninitialised value was created by a stack allocation
==3679==    at 0x805590F: sixlowpan_lowpan_init_interface
(lowpan.c:1709)
...
```

Figure 2.3.: memcheck output for the use of an uninitialised memory location

2.6.4 Structured Automated Tests

In contrast to constrained devices, running high numbers of tests is not a problem on desktop hardware.

2.6.5 Wireshark - Network Analyzer

Thanks to the OS' capabilities and an abundance of processing power, the *Wireshark* network analyzer can be used to capture and evaluate network traffic directly on the monitoring system.

2.6.6 Conclusion

The restrictions which prevent the use of most dynamic analysis tools on embedded devices don't apply to desktop systems. By enabling the use of development tools only available on desktop systems for analysis of software for IoT devices, an important gap can be bridged.

2.7 System Virtualization

In section 1.1.9 the claim that virtualization would help bring the benefits of traditional software development to the IoT was made. By enabling the use of desktop hardware to analyze and debug IoT software, the limitations of software development for constrained devices are facilitated.

In itself, virtualization can provide benefits, namely configurability and reduced hardware costs. Of course, the extent to which a specific virtualization platform may be controlled

is defined by each platform, but in general it is possible and relatively easy to modify the parameters of a software process, while it is relatively difficult and expensive to do this for hardware. Hardware costs can be reduced when one host system can share its resources to run several virtual instances.

There are several aspects to resource sharing, the foremost being that when a software system does not need the full processing power of the underlying hardware, another software system may take a share of the hardware resources, i.e. CPU, RAM and so on. This principle is founded on the observation, that processing power on modern computers is vast and the prime example for virtualization are modern OS'[8][39, Chapter 2] which abstract from the underlying hardware to allow for seemingly parallel execution of tasks. In order for the virtualization to be effective it must not use too much processing power itself in order to achieve its goal. One reason why this overhead may be introduced is to increase the realism of the virtual environment. This leads to the two main classes (or techniques) of virtualizers, namely *simulators* and *emulators*. While *simulators* provide an environment that has the same interface and the same behavior as the original, *emulators* only mimic the interface and may drastically simplify the degree to which the behavior is identical to the real system. The degree to which the environment is modelled also has influence over the controllability of the system. Depending on the task, it is quite possible that a simulator will exhaust the resources of a typical desktop computer. One example for this are weather forecast simulations which typically use the worlds most powerful data centers to simulate the complex atmospheric effects of the whole earth.

In order to decide which virtualization technique best suites the requirements for the virtual machine that should be implemented for *RIOT*, a detailed analysis of both, the requirements and the techniques needs to done.

Due to the fact that IoT hardware is significantly less powerful than regular computers per definition, it follows that a time sharing virtualization approach should be able to run a high number of virtual IoT systems in parallel given a moderate overhead. As one of the goals for this thesis is to enable testbed virtualization with many nodes, a highly realistic simulation is probably not feasible.

2.7.1 System Abstraction Levels

Virtualization can happen at different levels. In this section, different abstraction levels are explained with the goal of identifying defining properties.

As mentioned before, virtualization can happen at different levels. An operating system for example defines an abstraction layer for hardware. An application that is written against this abstraction layer could be compiled to run on a different OS that implements the same interface without changing the application code. The POSIX specification [40] is a prominent example of such an abstraction layer.

[8] An OS is not a virtualizer in the sense that it mimics an environment, it merely realizes one form of virtualization.

Call Level Virtualization

Given this approach, it is possible to write an OS virtualizer which implements the abstraction layer but runs as an application within a host OS itself. The User-mode Linux (UmL) virtualizer is an example of this approach, as described in [41, p. 18].

This form of virtualization happens at the system call level. Function calls that would usually invoke some function of the OS are called in the virtualizer instead. The virtualizer in turn might just call the same function in its host OS.

Hardware Level Virtualization

A different approach, which is typical for simulators, is to virtualize the hardware itself. To achieve this, the virtualizer will "execute"[9] the machine code of a given software. As a computer consists of more than the CPU however, and most software is intended to interact with the real world in one way or another, an interface to some other hardware needs to be implemented as well.

Here, two possible paths fork: One is to mimic the behavior of existing hardware, i.e. mocking some standard hardware device, thus being compatible with existing implementations. The other is to define a new hardware device which needs to be explicitly programmed for. Throwing compatibility with existing devices over board has the advantage of possibly getting rid of burdens which only made sense for real hardware, and the disadvantage of requiring users to make their software compatible with this new hardware platform. The Java Virtual Machine (JVM) [42] is a prominent example of a virtual hardware platform that defines its own interface, while QEMU [43] is an example for the former.

Call Level versus Hardware Level Virtualization

There are different implications that arise from these techniques. While the call level virtualization requires an application to be executable on the host system, a hardware level virtualization allows the execution of arbitrary software. The downside of hardware virtualization is that the virtualizer itself is far more complex and the execution has a much higher overhead as it can not execute on the hardware directly.

Part of this downside can be overcome to a large extent by employing *hardware assisted virtualization*. This means that instead of interpreting (or recompiling) the machine code, it can be executed on the host CPU directly with active help from that CPU. While this technique is widely employed and has enabled cloud computing [44] to a large extent, it still requires the software to be in a compatible binary format. This is not a problem because x86 is the predominant instruction set architecture (ISA) in the desktop and internet server industry where this kind of virtualization is usually used. Also, there is a certain set of standard hardware which is sufficient to emulate in order for the typical software system to

[9] "Execution" in this context means that the simulator will either interpret the machine code, i.e. read one instruction after another and use a lookup table of sorts to decide what to do, or it can recompile the code into either the native machine language, modifying it where necessary, or some optimized virtual machine code. The latter is done by e.g. Quick EMUlator (QEMU) and Valgrind, the former is done e.g. by Rosetta.

be virtualized[10].

Hardware for the IoT however is much more heterogeneous in both, ISAs and peripheral hardware. Also, the interface to peripheral hardware is not as standardized as on the x86 platform.

2.7.2 Emulation versus Simulation

When talking about virtualization, it is important to not only define at which system level the virtualization happens, but also how accurate the virtualized system is modelled. When the virtual system behaves exactly like the modelled system, this is called simulation. If the system is only modelled at a functional level, it is called emulation.

Both, simulation and emulation have different use cases. The strengths of each technique is illustrated in figure 2.4.

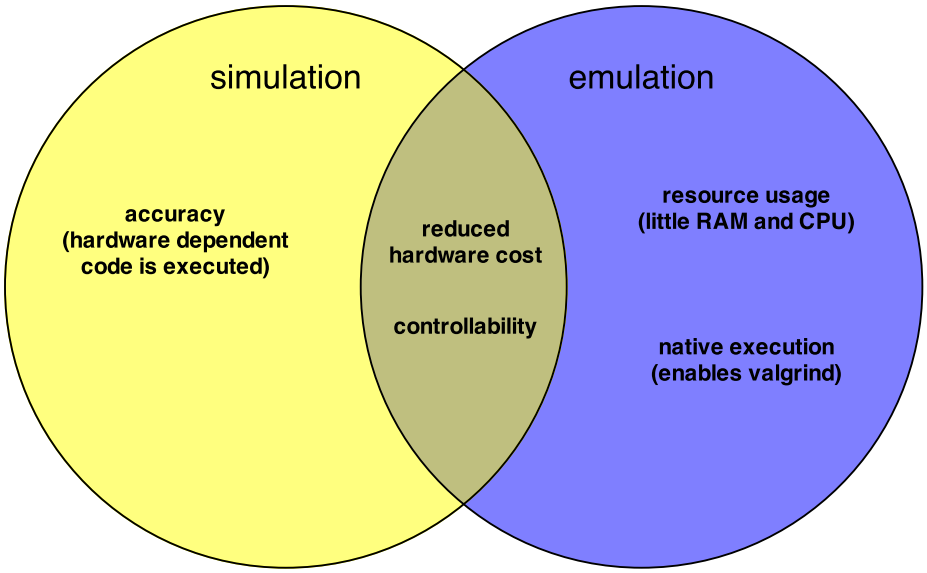

Figure 2.4.: Strenghts of virtualization techniques

2.7.3 Conclusion

The virtual platform that is proposed in this thesis is intended to be used as both, a virtual hardware platform for software analysis, and as a tool for network virtualization.

In order to facilitate larger network scenarios, it should be easy on resources. This is a point in favour of emulation already.

To be most useful as a software analysis tool, it needs to be to run as a process of a host OS. This means, the use of hardware level emulators like *QEMU* is not feasible. It also means, that no hardware simulator can be used.

[10] QEMU for example implements two network interface controllers (NICs), two kind of video graphics adapters (VGAs) and two kinds of AT Attachment (ATA) interfaces on the x86 architecture.

2.8 Network Virtualization

To circumvent the problems that come with physical testbeds as given below, the virtual platform should provide a means for deterministic testing and debugging of network communication.

The problems to be solved are:

- cost and usability of the tools
- scalability of the testbed
- controllability of the environment

Virtual networking can solve two problems. On one hand it creates a controllable, reproducible environment that can be used to test how network protocols perform in specific situations. On the other hand, it enables monitoring of the traffic, so that a developer can check if the protocol behaves as expected.

2.8.1 Network Simulation versus Emulation

Like for system virtualization, there are two kinds of tools: simulators and emulators. Simulators try to provide a realistic and accurate simulation of a real network including the particularities of the medium. Emulators just provide the functionality and optionally some properties (for example packet loss, reordering or delay) of a network.

Network Simulators

Network simulators are highly specialized tools. This can make them hard to use due to steep learning curves and the general problem at hand - defining a sensible scenario. Compare [45] for an analysis of how complicated network simulators are. Due to the high level of realism, a network simulation generally also has a high demand for resources.

The evaluation of test results is not trivial either. Large network simulations can generate vast amounts of data. The handling and analysis of the data requires resources and work. Due to this, a network simulator demands for dedicated hardware.

Network Emulators

If precise timings or realistic modelling of the medium properties are not important, a network emulation can often suffice for developing, debugging, testing and evaluating network protocols. Depending on the task they can even be preferable because they can scale better due to their lower resource demand and because they are easier to use.

Conclusion

Due to the focus on quality assurance and alleviating embedded software development, support of network emulation to facilitate network virtualization seems more urgent than network simulation. As also described in chapter 5, network emulation serves the task of

development, testing and debugging, while testbeds and network simulators serve different purposes.

2.8.2 TAP Networking

All contemporary OS' come with or have third party support for TAP devices[11], i.e. virtual Ethernet devices[12] [46]. These devices can be connected using virtual switches and controlled using the respective firewalls. Also, as they act like regular network devices, the traffic can be monitored with any tool built for that purpose [47], i.e. pcap file export is not necessary.

At Freie Universität Berlin, a framework for network emulation based on *TAP* devices with the *Linux* tool *netem* [48] has been developed for use in the *DES-Testbed* [49]. This framework can be used to defined arbitrary network topologies with well defined properties like loss and bandwidth as mentioned in [50, Virtualization Using DES-Virt].

2.9 Productivity

As *RIOT* has a focus on ease of development (usability) and well tested code, one can not but think of test-driven development. In order be productive in a test-driven development process, the speed of the test framework is key to productivity.

2.9.1 Waiting Destroys Productivity

In Peopleware [51, Chapter 11], the influence of interruptions on a developers concentration is outlined. The long and short of it is that to really get work done the utmost concentration is needed and any ever so short interruption will cost a developer about 15 minutes of time to get back into the flow.

Waiting for the ...

While there appears to be no scientific literature on this topic[13], it can be argued that waiting for the compiler can constitute a form of interruption. Based on the thin literature, a 15 second period is sufficient to break a developers concentration. This is partly due to boredom itself, but most often a result of switching to a different activity to pass the time. In the *test*

[11] TAP stands for network tap.

[12] There is also support for TUN devices which are based on UDP.

[13] There is some non-scientific literature on that topic, though:
a few seconds: `https://web.archive.org/web/20131217102615/http://www.componentowl.com/blog/zen-coder-vs-distraction-junkie/`
15 seconds: `https://web.archive.org/web/20140814235947/http://www.joelonsoftware.com/articles/fog0000000043.html`
mentions papers: `https://web.archive.org/web/20140717021753/http://www.nytimes.com/2007/03/25/business/25multi.html`
counter argument: `https://web.archive.org/web/20140816175252/http://staging.embedded.com/design/prototyping-and-development/4008900/3/Getting-disciplined-about-embedded-software-development-Part-2--The-Seven-Step-Plan`

driven development (TDD) model, the problems that arise from longer compilation times are magnified as the development process is centered around recompilation and execution.

Implications for Embedded Software Development

In contrast to traditional systems, compilation time tends to be shorter for constrained devices due to the fact that the applications are much smaller. Therefore, compile time is usually not such a problem for embedded systems. Deployment, i.e. the installation of the application into the systems flash memory, on the other hand tends to be a lengthy operation for these devices. It is typical for embedded systems to have deployment timings in tens of seconds[14]

2.9.2 Test-Driven Development

In order to support a workflow where a developer wants to test a new piece of code, by compiling, deploying and running it, these steps should finish in under 15 seconds in order not to break concentration. While this is difficult to achieve on embedded devices, a virtual platform should be able to make this deadline.

Implications for the Virtual Platform

The compilation of an application for a virtual platform should not need substantially different time than for a physical platform. Deployment should not take any significant time for the virtual platform in contrast to the often slow transfer to an embedded device. Finally, the execution of an application, especially applications like test cases, could take significantly less time on a virtual platform, if it is not artificially slowed down, because the development systems is significantly faster than the embedded system.

Implications for the Virtual Network

Networked applications have additional properties for deployment and execution. In this case, deployment could happen in parallel for embedded devices, so that this duration stays more or less constant. Also, the execution of applications is absolutely parallel for embedded devices, as each instance has its own physical device. A virtualizer in contrast will typically run many instances on one machine, so deployment (if any), and execution are serialized. Therefore, even a very small time penalty for the deployment and execution of the application can lead to a significant delay, resulting in loss of the developers concentration.

2.9.3 Conclusion

In conclusion, the duration for any of the compilation, deployment, and execution steps of a virtual platform should be as short as possible. Ideally, these steps would be carried out in less than 15 seconds in total[15], even for hundreds of virtual machines.

[14] Compare 4.3.2

[15] The time limit should not be surpassed on a contemporary development system.

2.10 The *native* Platform

As motivated in section 2.1, *RIOT* is in need of OSS methods and technologies to achieve its goal of high software quality, and a reference platform. In the introductory chapter **??**, the implementation of a virtual platform as a solution for this need was already anticipated. Section 2.2, it was observed that there is a technological gap in development tools for embedded systems in that (almost) no dynamic analysis tools exist. This leads to a whole class of potential defects that can not be found and analyzed as effectively as possible on desktop hardware. Based on this observation, the possibilities for system and network virtualization were analyzed in section 2.7. In section 2.8, the same has been done for network virtualization.

Based on the various requirements collected in this chapter, the important design decisions for the virtual platform can be made:

- the virtual platform should provide call-level API hardware emulation within *RIOT*
- the virtual platform should act as a regular process within the host OS
- the virtual platform will emply *TAP*-based networking
- the *DES-Virt* framework will be used for network emulation
- support for various dynamic analysis tools needs to be provided in a user-friendly manner
- both, the virtual platform as well as the virtual network should be very fast

2.10.1 Related Work

The use of call-level API emulation for system development is nothing new. One prominent example of this approach is the *Contiki native* emulation platform [52], which does for *Contiki* what the *RIOT native* emulation platform should do for *RIOT*. However, *Contiki* employs a cooperative threading model and is thus free to use existing library functions for implementing its threading API. Furthermore, in contrast to *RIOT*, *Contiki* comes with a *network simulation* tool called *Cooja* [53]. Another difference to *Contiki native* is the support for analysis tools and safety checking in API implementations[16]. Last but not least, the *RIOT native* platform strives to support as many driver interfaces as possible including physical hardware to increase code coverage, while the *Contiki native* platform appears to confide in supporting some sensors.

[16] *native* tries to detect and warn about the misuse of APIs where possible.

CHAPTER 3

Design and Implementation of the *native* Platform

Virtualization of the *RIOT* kernel is realized by adding a new "hardware" architecture to *RIOT*, namely "native"[1]. Compiling *RIOT* for this architecture on a supported system[2] produces a binary that can be run as a user process. The most common hardware elements found on the kind of systems *RIOT* runs on are being emulated at the call level interface in the native port and make use of system calls to realize their functionality.

Since it is one task of the OS to abstract from the underlying hardware, porting one OS to run as a process in another OS is basically the implementation of an abstraction layer between the two OS'. The main difficulty is the fact that *RIOT* has its own concept of processes and its own scheduler, which is why the OS' support for multithreading could not be utilized for *RIOT*.

3.1 Overview

The *native* platform 3.1 consists of several conceptual parts. Following *RIOT*'s hardware modularization, *native* implements both, a CPU and board. Board and CPU have different capabilities. As several boards could use the same CPU and complement the CPU's capabilities, some non-essential functionality has been grouped into the board. This grouping is also represented in the file hierarchy.

Applications written for *RIOT* make calls not only to the *RIOT* core APIs, but also to optional interfaces that come with *RIOT* in the form of system libraries (like cryptographic functions or network protocols) and hardware drivers (like UART or network interfaces), which define their own hardware abstraction.

For *native*, all interfaces (threading, interrupt handling, timers, power management, and synchronization) defined by the core module have been implemented. In addition to that,

[1] The name was initially chosen because Contiki calls its equivalent port "native" [52]. 'native' in this context means that the native toolchain and execution environment of the host system are used.

[2] Linux, FreeBSD and OS X on x86 and Linux on ARM compatible hardware are supported as of now.

the most important driver (LED, energy meter, RTC, random, and UART) and system (transceiver, and config) interfaces are provided. The implementations follow the separation found in other board/CPU at the time of their writing as sketched in 3.1.

Finally, a standard C library is expected to be available, not only by applications but also by the *RIOT* core, driver and system modules. In the case of native, the C library is provided by the host OS. Because some of the C library functionality provides implicit hardware access by performing system calls, part of the API is implemented as part of the *native* platform.

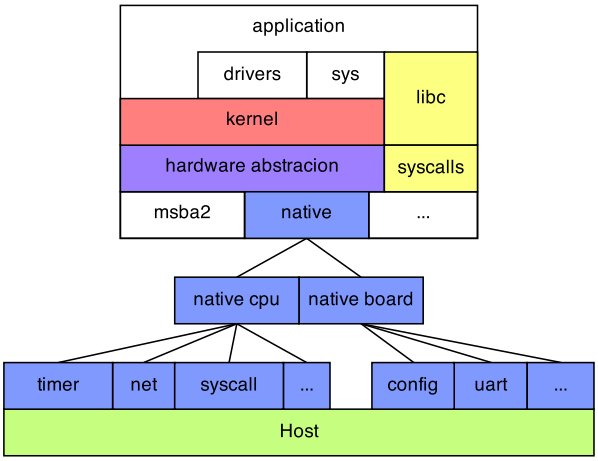

Figure 3.1.: *RIOT* native architecture overview

Regarding the implementation, *native* is located between *RIOT*'s hardware API and the host system API as seen in 3.2. The host functionality is provided by either standardized *POSIX* interfaces or by system specific "native APIs". For example, *POSIX* does not provide an API for virtual network interface creation, so that the implementation of this task looks different for *Linux* and *Mac OS X*. As *RIOT*'s threading model does not match the threading model specified by any of the existing host APIs, the most fitting glsAPI, i.e. *ucontext*, is complemented by some CPU specific code.

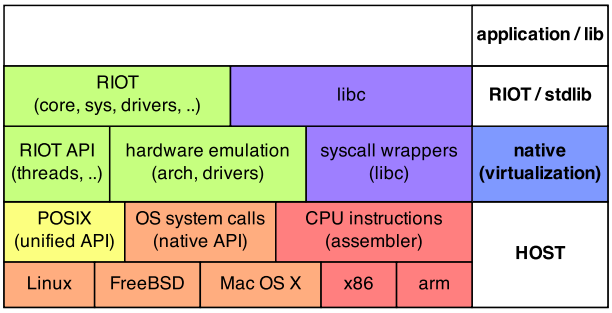

Figure 3.2.: *RIOT* native implementation overview

3.2 Threads

In computing, multithreading denotes the concept of several concurrent execution paths within one process. In *RIOT*, there is only ever one process, but it can have several threads.

3.2.1 The Thread Concept

A process is a program in execution. A program is a sequence of instructions. A program is in execution when a processor executes its instructions. Any non-trivial program contains instructions to read from and write to memory. The term "memory" can mean different things depending on the computer architecture and its role. Conceptually, there are two different kinds: data and instruction memory.

Contemporary CPU's have a so called "program counter" that points to the memory location of the next instruction. Additionally, there are a couple of so called registers that hold the data that is currently being worked with. The combination of program counter and registers make up the execution state.

To enable more accessible programming techniques and cope with blocking hardware interaction, it is possible to run a process in several threads. Threads share a processor and memory with another. To switch threads, the old thread's execution state is saved to memory and the new thread's state is loaded.

Contemporary CPU's have special instructions to read and write their state to a memory region called "stack". To support this, the "stack pointer" register points to the last element that has been saved to the stack.

A context switch in this case is generally implemented by first writing the execution state to stack, pointing the stack pointer to the new context's stack memory location, and loading the execution state.

It is also possible for threads to be switched involuntarily. Certain hardware events can trigger so called "interrupts". When the CPU is interrupted, it sets the program counter to a predefined memory location that contains the so called "interrupt service routine", "ISR" for short. The ISR can contain instructions to save the context of the previously running thread and restore a different thread's context after the interrupt has been handled. In order to support this, contemporary CPU's will save all or part of the execution state to the stack when the interrupt happens and have special instructions to load the state from stack.

3.2.2 Threads in RIOT

The *RIOT* threading API [54][31,32] requires a user who wishes to start a new thread to provide for the memory the thread should use as a stack. Typically this is done by letting the compiler allocate a static chunk of memory as shown in figure 3.3.

The thread_create implementation only contains some architecture independent functionality and calls the function thread_stack_init(stack, size , function) that is implemented for each architecture.

```
1  static char radio_stack_buffer[RADIO_STACK_SIZE];
2  void *radio(void *arg); /* defined elsewhere */
3
4  int main()
5  {
6      kernel_pid_t radio_pid = thread_create(
7          radio_stack_buffer,          /* stack memory */
8          sizeof(radio_stack_buffer), /* size of stack */
9          PRIORITY_MAIN - 2,           /* thread priority */
10         CREATE_STACKTEST,            /* creation flags */
11         radio,                       /* thread function */
12         NULL,                        /* argument for the
13                                         thread function */
14         "radio");                    /* thread name */
15 }
```

Figure 3.3.: Creating a thread in RIOT

3.2.3 POSIX ucontext API

POSIX defines a set of functions for user space context switching in the POSIX.1-2001 standard[3]. As they are defined in the ucontext.h header, they are being referred to as the ucontext API throughout this document. The functions are shown in figure 3.4.

```
1  void makecontext(ucontext_t *ucp, void (*func)(), int argc,
       ...);
2  int swapcontext(ucontext_t *oucp, ucontext_t *ucp);
3  int getcontext(ucontext_t *ucp);
4  int setcontext(const ucontext_t *ucp);
```

Figure 3.4.: POSIX ucontext API

3.2.4 Threads in *Native*

During the life cycle of a thread makecontext is used to initialize a new thread, setcontext is used to activate it, swapcontext is used to switch between two threads, and getcontext is used to save the current state.

[3] It has since been removed from the standard again in POSIX.1-2008. As of now it is still supported in Linux, FreeBSD and OS X. The semantic of the implementation has changed since it was first standardized due to implementation difficulties.

3.2.5 Alternatives to *ucontext*

The *setjmp* function family could probably also be used to create threads for use with the *RIOT* scheduler. In [55], the author describes how the *GNU Portable Threads* implementation makes use of this function set to enable cooperative threading. Due to the relatively complex (compared to the *ucontext* function family) setup phase needed for the creation of threads with *setjmp*, this approach has not been pursued.

3.3 Interrupts

Interrupts, as the name says, interrupt the regular execution – typically an interrupt service routine is called to address each specific interrupt. In order to prevent a sensitive operation from being interrupted, interrupts can also be temporarily disabled.

3.3.1 Interrupt Concepts

Modern OS' allow for some of the events that typically involve a hardware interrupt to be received as signals though. Also, they allow processes to create signals as a means of communication. The *RIOT* native port therefore uses signals to emulate hardware interrupts.

In principal there are two kinds of interrupts that need to be supported by any OS: hardware and software interrupts. While user processes can generate hardware interrupts, for example via a division by zero, they cannot directly receive these interrupts. Interrupt servicing is one of the core functions of an OS and should not be accessible to user processes for a variety of reasons, foremost reliability and security. Software interrupts are conceptually different from hardware interrupts in that they are created by software events rather than hardware events. What they do have in common though is the instrumentation of the interrupt hardware as a means of communication. Therefore the same restrictions that apply to hardware interrupts apply to software interrupts as well, namely it does not allow their arbitrary usage for user processes.

3.3.2 POSIX Signals

Interrupts are available to user processes as signals in a POSIX environment. They are differentiated by a "signal" number which is passed to the "signal handler" whenever a signal occurs. The signal(sig, *func) system call registers a function func to be called to handle the signal sig. This signal handler will then be called asynchronously whenever the signal occurs. When the signal handler returns, control is returned to the context which was active at the time the signal occurred.

One major drawback in the POSIX specification of signals is, that signal handlers can only call a limited set of system calls safely[4]. One system call that is not safe happens to be the context switching call. As *RIOT* supports asynchronous context switches which are triggered by interrupts, it needs to switch contexts in the interrupt service routine. The

[4] man 7 signal lists all safe functions

significance of this is that, in order to be able to support the system calls *RIOT* needs, one can not service the interrupt directly from within the signal handler.

3.3.3 *Native* Interrupts

In order to implement interrupts, *native* uses the POSIX signal API to install signal handlers. System facilities like file descriptors and clocks are configured to generate signals which interrupt the regular execution and lead to the invocation of a general purpose signal handler.

Asynchronous Context Switching

The native signal handler processes the signal number to differentiate which virtual hardware triggered the interrupt, and modifies the return context so control is passed to an interrupt service routine (ISR) context on return. This is necessary to work around the POSIX limitation of not allowing context switches from within the signal handler context. Figure 3.5 illustrates the context changes involved in signal handling.

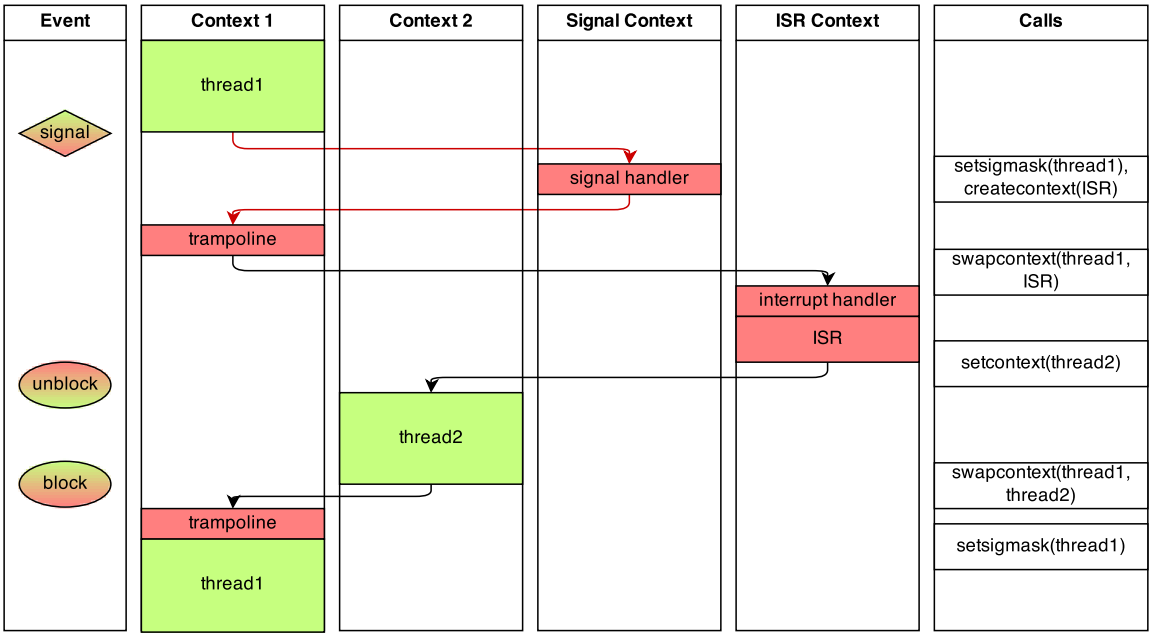

Figure 3.5.: *RIOT* native thread switching
Whenever a signal occurs, the host OS performs a context switch to a signal handler (transition marked in red). As it is not possible to safely return to a different context from the signal handler, a trampoline is used instead. After execution of the trampoline, *swapcontext* can be used in the regular manner to switch threads. From *RIOT*'s perspective, the execution of the signal handler is transparent and the ISR is executed in the same order as on other platforms.

The signal handler saves the signal by writing it to a First In First Out (FIFO) data structure. Then, it makes sure it is safe to switch contexts, i.e. no system call is currently being executed. This is necessary because it is not possible to return to an interrupted

system call from userspace. If it is safe to switch contexts, it installs a trampoline function to be called when the signal handler returns execution to the context that ran when the signal occurred. In order to prevent further interruptions during the critical transition, it modifies the context's signal mask to block all signals. Upon return to this context, the trampoline code saves the context on the current thread's stack and switches to a signal handler context. The interrupt service routine which handles the interrupt for *RIOT* is then called from within this context depending on the signal that has been received. Figure **??** illustrates how the stack contents of an interrupted context change during interrupt handling.

In the case that it is not safe to switch the context asynchronously, because a system call is currently executing, the system call can finish. In order to realize the context switch in this situation, every system call is wrapped with a function that checks whether an interrupt occurred in the meantime, and synchronously switches contexts if this is the case.

To modify the execution in case of an asynchronous context switch, an undocumented feature of the signal handler API is used. Figure 3.6 shows this for x86 Linux.

```
_native_saved_eip =
    ((ucontext_t *)context)->uc_mcontext.gregs[REG_EIP];
((ucontext_t *)context)->uc_mcontext.gregs[REG_EIP] =
    (unsigned int)&_native_sig_leave_tramp;
```

Figure 3.6.: saving and modifying the program counter in the Linux signal handler

The signal handler receives not only the signal that led to it being called, but also a pointer to the process state which was stored on the stack by the OS' interrupt handler. By accessing this data structure, it is possible to change the program counter on the stack. The changing of program counter and stack pointer is shown in round boxes in **??**. Although the register names are different for Linux, FreeBSD and OS X, all systems provide this type of access.

Implementation Details

The trampoline function is implemented in assembly. The need for assembly is due to the need for a function that does not modify the CPU state. While there is a naked attribute on some architectures, *GNU Compiler Collection (GCC)* does not implement it on the *x86* architecture [56].

In order to allow for easier maintenance of the various signals the *native* port uses, two functions, **int** register_interrupt(**int** sig, **void** (*handler)(**void**)) and **int** unregister_interrupt(**int** sig) have been implemented. These functions make an internal note of which handler to call when the signal occurs, and install a global signal handler _native_isr_entry(...) for the signal.

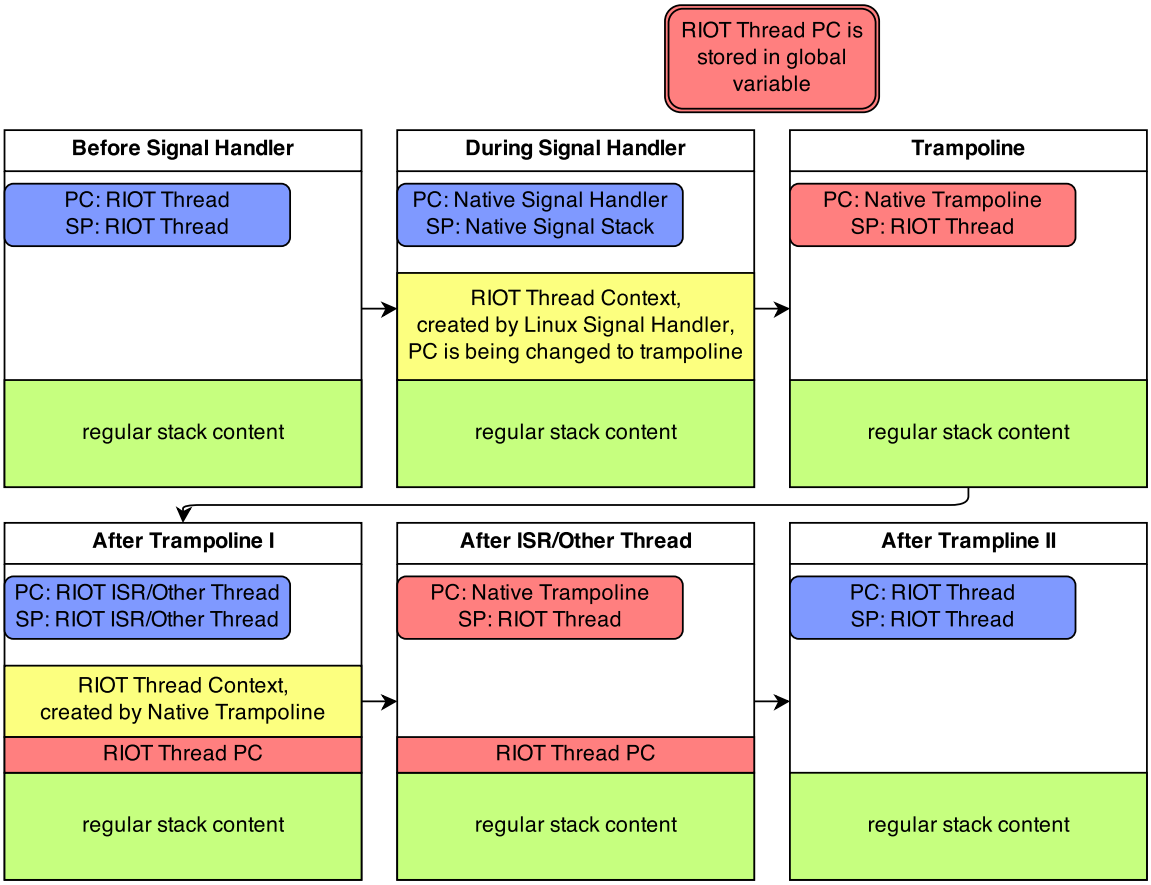

Figure 3.7.: *RIOT* thread stack states during interrupts
First, the signal handler is executed regularly, but changes the return program counter. Then, the original program counter and context are stored on stack. When reentering the thread, the original context is restored and the program counter is jumped to.

3.4 Hardware Timers

A typical hardware timer has two functionalities:

- record the passage of time and provide the current point in time
- trigger interrupts when certain points in time has been reached

3.4.1 Timers in *RIOT*

In *RIOT*, timer functionalities are made available to the system by the implementation of two functions[5]:

- **unsigned long** hwtimer_arch_now(**void**);
- **void** hwtimer_arch_set_absolute(**unsigned long** value, **short** timer);

3.4.2 Timers in *Native*

hwtimer_arch_now

POSIX defines several clocks to get the current process time. The process time counts the time since a process has been started and is therefore analogues to a hardware timer which counts the time since it has been activated.

int clock_gettime(CLOCK_MONOTONIC, ...) [57] has been chosen to implement hwtimer_arch_now because it is available on most platforms.

hwtimer_arch_set_absolute

POSIX defines the system call setitimer (...) [58] to install a timer which triggers a signal when the given point in time is reached. The native port uses this system call to implement the hardware timer interface.

There is only one timer available to user processes in POSIX[6], which is why several virtual timer channels are being multiplexed onto this "hardware" timer channel.

3.5 Virtual Networking

To enable network communication between *RIOT* processes, support for virtual networking has been added in the form of the "nativenet" module. The emulated transceiver uses a *TAP* interface to provide Ethernet based networking via the *RIOT transceiver* module.

[5] There is also **void** hwtimer_arch_set(**unsigned long** offset, **short** timer), but the underlying mechanism is the same.

[6] setitimer is defined in POSIX.1-2001 but has been made obsolete by POSIX.1-2008. Due to the alternative's (timer_settime) dependence on librt, setitimer is being used nonetheless.

3.5.1 TAP Networking

A TAP interface[7] is a virtual network interface that is available in Linux, FreeBSD and OS X[8]. TAP devices can be configured like regular network interfaces and added to a virtual switch which is called "bridge" in Linux [46]. When several TAP interfaces are connected to the same bridge, they behave like regular Ethernet interfaces that are connected by a network switch. The immediate advantage of using TAP interfaces over implementing a network emulator from scratch is that it is readily available. Other advantages arise from the fact that TAP interfaces are treated like regular network devices by the host OS so their use allows for using existing software for network analysis and control.

3.5.2 TAP networking in native

The network interface was written against *RIOT*'s *transceiver* API [59] whose purpose it is to provide unified access to various typical network interfaces ("transceivers") operating at the data link layer, i.e. Open Systems Interconnect (OSI) layer two [60].

For embedded transceivers, this means that it is possible to configure a network address, wireless channel and PAN ID, and to send and receive data. These transceivers can typically work in a *monitor* mode in which the transceiver receives and delivers all packets, not just the ones that match its address. To cater for all of these options, the *native* interface does not use the Ethernet device transparently. Instead, a minimal link layer protocol is implemented on top of Ethernet, thus tunneling one layer two protocol over another.

The "*nativenet* protocol" is rather simple. Its header format and position within a network packet is depicted in 3.8.

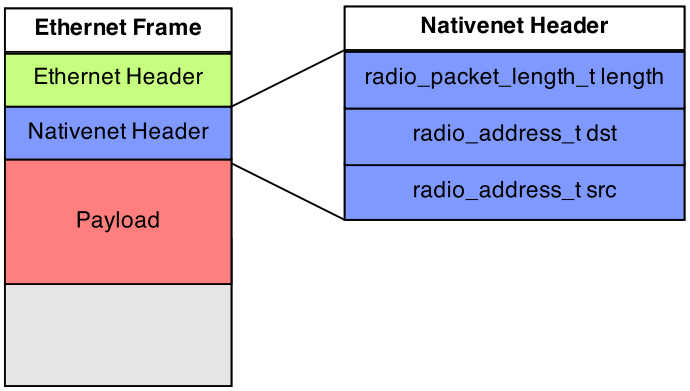

Figure 3.8.: The Nativenet Header

The implementation of the intermediate layer two protocol is necessary to enable changing link-layer addresses of the interfaces. At least in Linux, it is not possible to change a TAP interface's MAC address after it has been created.

[7] TAP stands for network tap.

[8] OS X does not come with TAP support, but the *tuntaposx* software package includes an open source implementation of a TAP kernel module.

Because *nativenet* uses its own addresses which are not recognized by the host OS, broadcasting is used on the Ethernet layer.

3.5.3 Nativenet Implementation

Internally, the *native* network module is structured into three parts: an interface to the *transceiver* API and integration with the *transceiver* module, a *nativenet* API which defines the *RIOT* interface of the network module, and a TAP network API which is used for internal modularization[9]. The resulting network implementation is shown in 3.9.

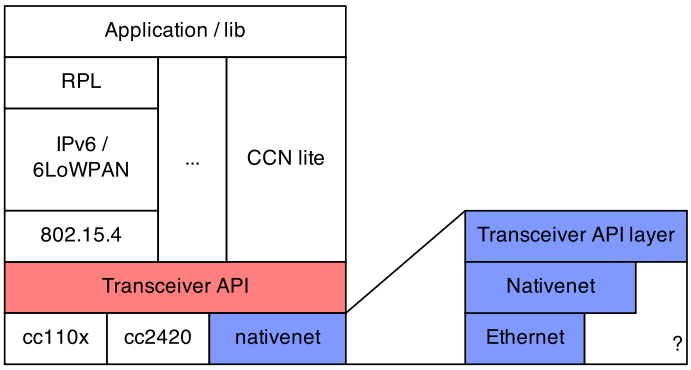

Figure 3.9.: Nativenet Integration in *RIOT*'s Network Architecture
The *nativenet* transceiver is a regular transceiver from *RIOT*'s point of view. Albeit not currently used, the internal structure of the nativenet transceiver allows for different lower level interfaces. They would take the place of the question mark in the lower right corner.

3.5.4 Configurability

The *nativenet* module can be configured within certain bounds in order to help with debugging of the network stack.

Because the 1500 byte MTU of Ethernet is untypical for low-power networks, it is possible to reduce the packet size of the virtual interface. In order to determine the maximum size of a packet that can be sent, the macro NATIVE_MAX_DATA_LENGTH is checked. The default value is defined to make use of a complete Ethernet frame. To conform with the 802.15.4 definition, a value of 255 is chosen instead when the "sixlowpan" module is being built.

The virtual network module itself has been implemented to be easily extendible. While only the tap interface has been implemented as of now, adding support for a different *network layer* only requires the implementation of a few functions. Most of the transceiver module API is handled completely in the nativenet abstraction layer. The functions that need to work with the actual network interface (sending, receiving, initializing) require

[9] This description is for the original implementation. The module has been updated since to allow for integration with *RIOT*'s next network API. As of now, it contains an additional abstraction which corresponds to this newer API.

the implementation of corresponding functions to communicate with the lower layer. For interfaces that would be used more transparently (an 802.15.4 dongle for example), a callback architecture for the changing of channel, PAN ID and address has been provided.

3.6 Traffic Analysis Support

One intended side effect of using TAP interfaces is that existing software can be used to monitor and investigate the network traffic between *RIOT* processes. The *pcap* API [61] can be used for traffic monitoring and tools such as *Wireshark* [11] and others [47] can be used for analysis. In order to help with analysis, a plugin has been written for Wireshark that parses the *nativenet* network layer. A screen capture of Wireshark parsing a *nativenet* packet is shown in figure 3.10.

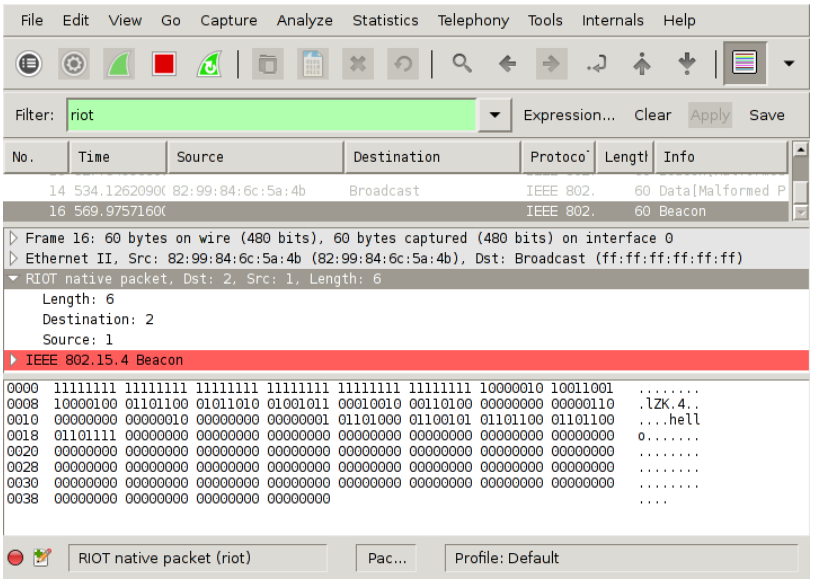

Figure 3.10.: Screen capture of Wireshark using the dissector for nativenet packets

3.7 Virtual Testbed Support

Another consequence of using TAP devices for network emulation is that network traffic can be manipulated by the host's kernel with their respective filtering mechanisms. Linux' *netem* facilities are especially useful, as they allow for statistical network modelling. The interaction of related technologies is shown in figure 3.11.

Although rather complex in use, the *tc* command allows for the definition of various rules to shape network traffic. While it is also possible to define duplication and reordering and delay of packets, the following parameters are more interesting for the emulation of wireless networks:

- Loss - drops packets according to several random distributions

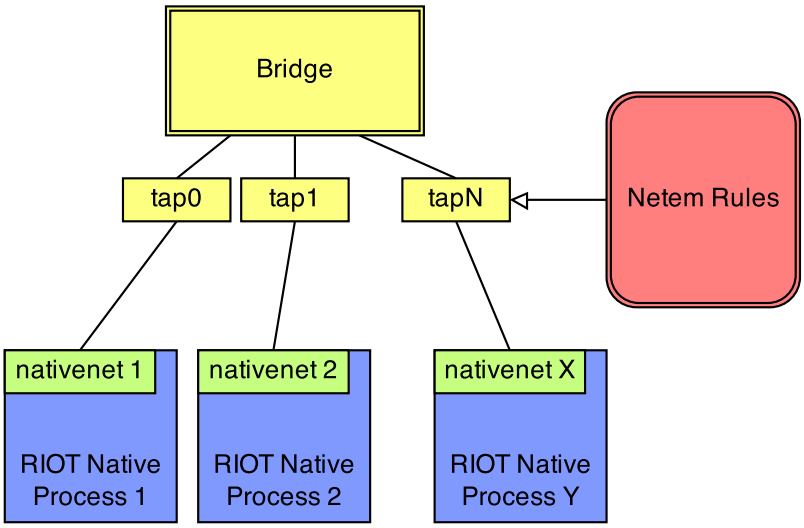

Figure 3.11.: A virtual network of *nativenet* transceivers

- Corruption - inserts errors into packets
- Rate - limits the transmission rate

3.7.1 *DES-Virt*

The *DES-Virt* framework [21] [62] leverages *netem*'s functionality and allows the definition of various network topologies via XML files. In addition to *tc*, it makes use of the *ebtables* tool to disable forwarding of packets between devices completely, thus creating a basic topology. An example for a resulting network architecture and the technologies involved is shown in 3.12.

3.8 UART

A UART is a serial input/output device. Several UARTs may be available on a system, one of which is typically used as a character device to enable text based communication between the IoT device and a desktop computer[10].

3.8.1 UARTs in *RIOT*

The "*uart0*" driver interface in *RIOT* is used to provide abstract access to a default UART hardware device. It is comprised of uart0_readc and uart0_putc functions among others.

RIOT has a shell module that is used to provide interactive command access to the system.

[10] In fact, most IoT devices nowadays have universal synchronous/asynchronous receiver/transmitters (USARTs) which can also communicate in a synchronous manner, the terminology tends to ignore this, though. This is also the case with *RIOT* where the UART driver is used for UARTs and USARTs alike.

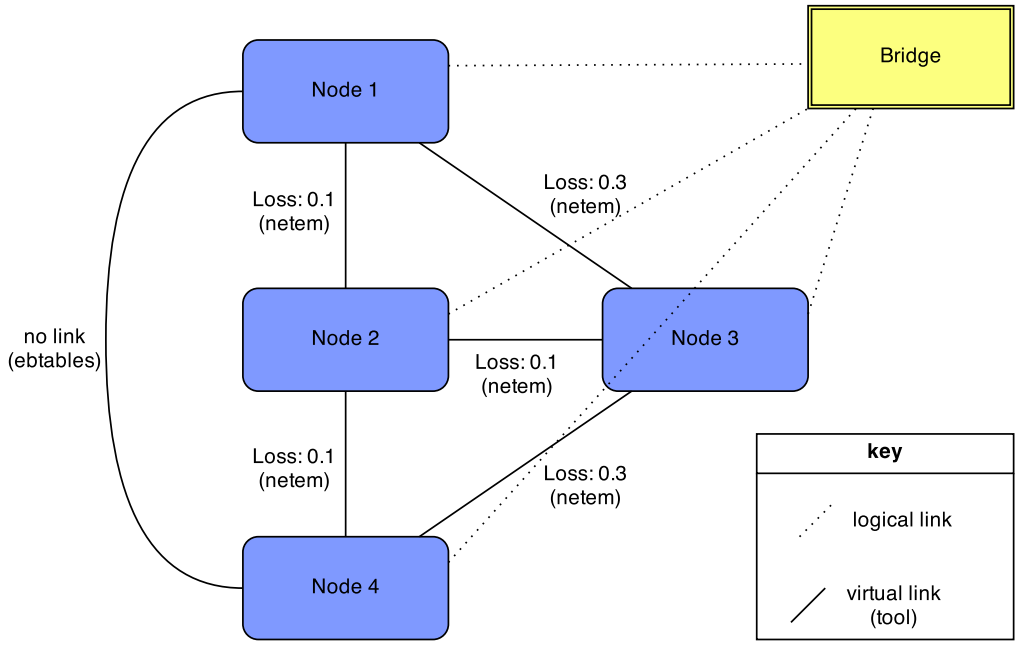

Figure 3.12.: A virtual testbed topology as defined by *DES-Virt*

The shell has to be initialized with input and output functions - these are typically the uart0_readc and uart0_putc functions.

Additionally, the *stdio* family of functions is used throughout *RIOT*. The standard libraries of each toolchain transparently write *stdout* output to a default UART which - in a software development scenario - is typically the same UART used by the shell via *RIOT*'s *uart0* driver.

3.8.2 stdio in C and POSIX

The C API defines the *stdio* API which provides every process with one input, and two output file handles: *stdin*, *stdout*, and *stderr*. Usually, these communicate with the controlling terminal, i.e. the terminal that started the process. The *stdio* family of functions like *printf* write to *stdout* when not explicitly invoked with a different output identifier.

POSIX *stdio* Redirection

It is possible to assign a different file descriptor to *stdin*, *stdout* and *stderr*[11] with the POSIX *dup2* system call. By doing so, *stdio* communication can be redirected to arbitrary file handles.

[11] The same goes for any other file descriptor.

POSIX Pipes

One special file implementation in POSIX is the *pipe*, which is created with the *pipe* system call. It creates tuple of file descriptors where everything that is written to second is available for reading on the first.

3.8.3 *uart0* in *Native*

The native *uart0* input module has been designed to cater for different use cases: Per default all input is read from the stdin file descriptor that the controlling terminal has set up. A screen capture of a terminal displaying *RIOT native uart0* output is shown in 3.13.

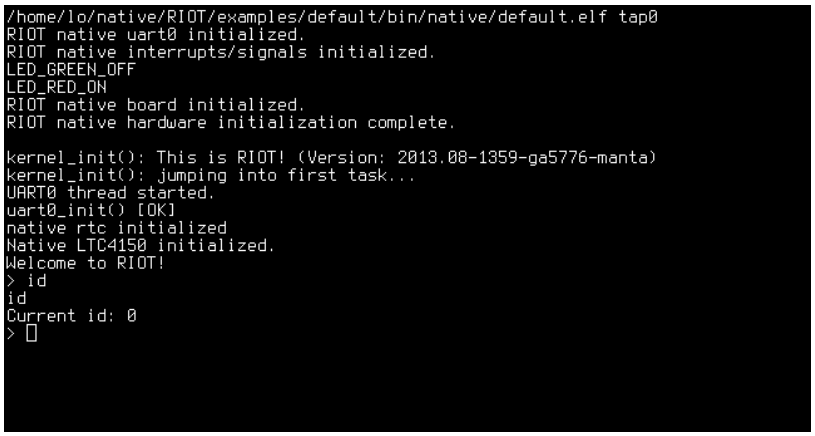

Figure 3.13.: Screen capture of a terminal emulator communicating with the native *uart0* module

When a *RIOT* process is daemonized, i.e. detached from the controlling terminal, the stdin file descriptor is internally set up to read from a *pipe*. It is now possible to attach to *uart0* by connecting to a socket (either TCP or UNIX) of the *RIOT* process. When the socket is connected, the stdin file descriptor is changed to read from this socket instead. Once the socket is closed, it is changed to read from the pipe again.

The reason for this is, that the remaining UART driver does not need to differentiate whether it is connected to a file descriptor or not. The output part works in an analogous fashion and discards output while not connected to via a socket when in daemon mode.

As *RIOT* not only handles input and output in the *uart0* driver, but also uses the printf family of functions for output, the *uart0* module is divided into an input/output redirection part for the *uart0* driver and an output part that works without driver interaction. The latter part is always included even if the *uart0* module is not explicitly built.

3.9 RTC

In contrast to a timer as outlined in section 3.4, an RTC provides wall-clock time and date. These devices differ from timers not only in their representation of time (usually they have a lower resolution but a greater range), but also in their mode of operation. Where timers

will also reset on a system reset and may be deactivated in case no timer event is scheduled, RTCs are generally always on in order to keep the clock in sync with the rest of the world. This is usually the case even if the system is powered down otherwise.

3.9.1 RTC in *Native*

The RTC driver uses POSIX system calls to to implement the *RIOT* rtc driver[12].

Although the API has been implemented, the alarm feature is not functional as there is no POSIX interface which models RTC closely. It is possible to implement this functionality using the timer interface described in section 3.4, but as the alarm feature was not used by any *RIOT* application at the time of this writing, it has been neglected.

3.10 GPIO

In order to allow for the development of drivers and applications using GPIO on *native*, *RIOT*'s interface has been implemented[13]. There are two separate implementations of the interface to cater for different purposes: A *virtual* GPIO interface, that can be used for build-testing and application development without access to actual hardware, and a *sysfs* interface that allows for access to physical GPIO devices in Linux. The *sysfs* interface is useful on platforms like the *Raspberry Pi* [63], which are built to allow for rapid development of embedded applications using Linux.

3.10.1 *RIOT* GPIO API

The low-level API for GPIO access in *RIOT* is defined in drivers/include/periph/gpio.h and consists of definitions for GPIO pin names, configuration, and read and write access function declarations. A low-level driver for the peripheral GPIO API needs to implement these functions, and specify which pin names exist.

3.10.2 Linux *sysfs* GPIO

Linux's *sysfs* GPIO interface [64] allows for portable access to hardware GPIO pins by userspace applications. The design is rather simple: By writing the number of a specific GPIO pin to a control file, that pin is made accessible to the user. After this initial step, the pin is available in the *sysfs* pseudo file system and can be configured and used for reading and writing. Figure 3.14 is an exemplary listing demonstrating this procedure.

[12] The gettimeofday functions which is used to obtain the system time conforms to POSIX.1-2001 and has been marked obsolete in POSIX.1-2008. As the alternative clock_gettime requires linking librt, the obsolete interface has been chosen nonetheless.

[13] It is available as a pull request at `https://github.com/RIOT-OS/RIOT/pull/1737`. Merging is not planned prior to the completion of the missing features outlined below.

```
1 # echo "23" > /sys/class/gpio/export
2 # echo "out" > /sys/class/gpio/gpio23/direction
3 # echo "1" > /sys/class/gpio/gpio23/value
4 # echo "0" > /sys/class/gpio/gpio23/value
5 # echo "5" > /sys/class/gpio/export
6 # echo "in" > /sys/class/gpio/gpio5/direction
7 # cat /sys/class/gpio/gpio5/direction
8 1
```

Figure 3.14.: Using *sysfs* GPIO devices in Linux.

3.10.3 GPIOin *Native*

A virtual GPIO interface has been implemented as a stub only. It implements the complete API, but only saves states and settings in a data structure. It does not have any possibilities for external control.

Additionally, the *Linux sysfs* GPIO API is supported. It does not offer interrupt driven operation, but reading and writing synchronously is possible. In order to use it, file permissions need to be set manually in the host OS. To keep the *native* board independent from the host OS, the application can specify how many GPIO pins should be made available.

Interrupt driver operation has not been implemented because it requires the implementation of an abstraction layer for file interrupts first. While such a layer is under development[14], the GPIO API is already usable without it and served as a (successful) test vehicle for peripheral interface interoperability.

3.11 Valgrind memcheck

The *memcheck*[15]tool which is part of the *Valgrind* framework (compare sections 2.6.2 and 2.6.3) does not require any changes to the program under investigation in general. It primarily needs debugging symbols in order for messages to be more accessible to humans. The implementation details of the *RIOT* native port are different from regular programs insofar as it implements its own threading mechanism. While Valgrind does have wrappers for threading libraries, it can not cope with the ucontext API, nor with the trampoline used for asynchronous context switching. While support for the ucontext API could be added to Valgrind any time, this is not the case for the trampoline. This is due to the fact, that only the author of the program knows how sane behavior is defined in this case as a result of the arbitrary stack usage.

The implication of Valgrind not being prepared for the threading method is that the location of the stack memory is unknown to it. From Valgrind's perspective, the process under investigation suddenly starts accessing stack memory in places that are not meant to be accessed as stack memory.

[14] The work in progress is available at `https://github.com/LudwigOrtmann/RIOT/tree/native_fildes`.
[15] The terms *memcheck* and *Valgrind* are used in a synonymous fashion because *memcheck* is the default tool *Valgrind* uses when started without parameters.

In order to enable programs that use their own threading implementation to be analyzed, the Valgrind headers contain the macro VALGRIND_STACK_REGISTER(start, end) which, when run in a Valgrind process, tells it to interpret a certain memory region[16] as stack memory. The manual mentions, that:

> Warning: Unfortunately, this client request is unreliable and best avoided. [65]

As testing did not reveal any false positives[17]and Valgrind's results did help spot several erroneous memory uses, it is assumed the macro indeed works as intended.

Another method would be to pass an option to Valgrind that specifies the stack size of the program under evaluation, i.e. the whole memory area *RIOT* uses for stacks. Specifying the memory area manually on each run is error prone as well and highly uncomfortable. A method to automate the passing of this option has not been evaluated because the macro seems to work as intended.

3.12 Stack Smashing Protection

In order to complement *Valgrind memcheck* in cases where it does not invalid stack memory accesses (compare section 2.6.2), the *stack smashing detection* feature of *GCC* can be used. To make it easily accessible, the build system has been configured to automatically add the −fstack−protector−all compiler flag in the *native* board's *Makefile* when the −DDEVELHELP compiler flag is given. Now, when an invalid memory access as shown in figure 3.15 is detected, the program terminates with some debug output as shown in figure 3.16 and creates a core dump.

```
1 void foo(void)
2 {
3     char bar[4] = "123\0";
4     int food = 4;
5
6     bar[food] = 'x';
7
8     printf("%s;%d\n", bar, food);
9 }
```

Figure 3.15.: Example of an invalid memory write that Valgrind does not detect.

[16] any location between *start* and *end* where "start" denotes the lower and "end" the upper bound

[17] Only one instance of seemingly correct behavior is indicated by *memcheck* as of now. Compare section A.3 for reference.

```
123;4
*** stack smashing detected ***: .../spp-example.elf terminated
======= Backtrace: =========
/usr/lib32/libc.so.6(+0x6c469)[0xf75a7469]
/usr/lib32/libc.so.6(__fortify_fail+0x37)[0xf7636877]
/usr/lib32/libc.so.6(+0xfb83a)[0xf763683a]
.../spp-example.elf[0x804d13a]
.../spp-example.elf[0x804d197]
.../spp-example.elf[0x8049392]
/usr/lib32/libc.so.6(makecontext+0x4b)[0xf7578a7b]
======= Memory map: ========
...
```

Figure 3.16.: Abbreviated example output of glibc stack smashing protection in action.

3.13 Profiling

Profilers work without the need for any changes in the code. In order to increase the usability, targets for two popular open source profilers have been added to the build system.

3.13.1 gprof

The gprof profiler works by linking the gprof runtime library into the application. This is achieved by giving the −pg parameter to the compiler as well as the linker. In order to obtain human readable results, the −g parameter must be given to the compiler so that debugging symbols are generated.

The all−gprof target can be used to do this. As the build system does not know about compile flags, it is necessary to remove the existing compilation products beforehand as shown in figure 3.17.

```
$ make clean
...
$ make all-gprof
...
```

Figure 3.17.: Compilation of the application for profiling with gprof

The application can now be started in the regular manner[18]. This results in the creation of a file named "gmon.out.PID"[19] in the current working directory. To show the profiling results, the *gprof* command line program is used. A build target which automatically evaluates the latest profile has been added for convenience. Figure 3.18 shows how to use it.

[18] A term−gprof target has been added for consistency.
[19] The file suffix is the process ID.

```
$ make eval-gprof
gprof /.../RIOT/tests/test_shell/bin/native/test_shell.elf gmon.out.24683
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00       65     0.00     0.00  _native_syscall_enter
  0.00      0.00     0.00       65     0.00     0.00  _native_syscall_leave
...
```

Figure 3.18.: gprof invocation and partial output

3.13.2 *cachegrind*

The *cachegrind* tool of the Valgrind framework has no needs despite debugging symbols for human readable output. An all−cachegrind target which enables debugging symbols has been added for convenience. Figure 3.19 shows how to build an application for profiling with *cachegrind*.

```
$ make clean
...
$ make all-cachegrind
...
```

Figure 3.19.: Compilation of the application for profiling with *cachegrind*

To generate the profile, the *cachegrind* tool is used. The term−cachegrind target can be used to to do this as show in figure 3.20.

As the primary goal of the *cachegrind* tool is profiling of CPU cache misses, it prints out this data when the application terminates. The profiling data is saved in a file named "cachegrind.out.PID"[20] which can be analyzed using the *cg_ annotate* command line program as shown in figure 3.21.

3.14 Ramifications of Using Native Libraries

The *RIOT* native port is compiled more or less like any native application on the host system. This has the primary advantage of not needing to maintain any third party standard libraries. As a side effect, *RIOT* and its applications are also linked against the system environment.

[20] The file suffix is the process ID.

```
$ make term-cachegrind
valgrind --tool=cachegrind /.../RIOT/tests/test_shell/bin/native/test_shell.elf
==26079== Cachegrind, a cache and branch-prediction profiler
...
==26079== I   refs:      343,683
==26079== I1  misses:      1,068
==26079== LLi misses:      1,047
==26079== I1  miss rate:    0.31%
==26079== LLi miss rate:    0.30%
==26079==
==26079== D   refs:      185,260  (145,840 rd   + 39,420 wr)
==26079== D1  misses:      2,935  (  2,152 rd   +    783 wr)
==26079== LLd misses:      2,461  (  1,718 rd   +    743 wr)
==26079== D1  miss rate:     1.5% (    1.4%     +    1.9%  )
==26079== LLd miss rate:     1.3% (    1.1%     +    1.8%  )
==26079==
==26079== LL refs:         4,003  (  3,220 rd   +    783 wr)
==26079== LL misses:       3,508  (  2,765 rd   +    743 wr)
==26079== LL miss rate:      0.6% (    0.5%     +    1.8%  )
```

Figure 3.20.: *cachegrind* invocation and partial output

```
$ make eval-cachegrind
cg_annotate cachegrind.out.26079
...
--------------------------------------------------------------------------------
     Ir  I1mr  ILmr      Dr  D1mr  DLmr     Dw D1mw DLmw
--------------------------------------------------------------------------------
343,728 1,074 1,053 145,830 2,128 1,715 39,422  777  741  PROGRAM TOTALS

--------------------------------------------------------------------------------
    Ir I1mr ILmr     Dr D1mr DLmr    Dw D1mw DLmw  file:function
--------------------------------------------------------------------------------
68,334   10   10 30,573  644  565   936    3    0  ???:_dl_addr
57,850   10   10 41,198    2    1 8,323  516  516  /.../core/thread.c:thread_create
53,833   15   14 20,947  245  188 8,425    5    3  ???:do_lookup_x
30,800   11   11  7,339  135  117 4,239    5    4  ???:_dl_lookup_symbol_x
27,040   46   46 10,176  524  492 2,400    3    3  ???:_dl_relocate_object
22,863    3    3  8,128   64   37     2    0    0  ???:strcmp
 ...
```

Figure 3.21.: cg_annotate invocation and partial output

3.14.1 Host Transparency

Due to the fact that the *native* port does not hide the underlying OS from the rest of *RIOT*, some confusion can arise. It is possible to call any function of the underlying system directly from within a *RIOT* application. This can have advantages, but it also has some disadvantages. For example, it is possible to use any library that is installed in the host OS transparently without importing it into *RIOT* first. This can be seen as an advantage, because it makes experimentation easy. The obvious disadvantage is, that one might accidentally call functions that are not part of *RIOT* in native, so the same application will not be able to run on a different platform.

This can be particularly annoying, when implementing some API for *RIOT* that is also part of the host system. The implementation of the POSIX layer and C++ support in *RIOT* for example made it necessary to change details of the native platform in order for the implementations to coexist. In order for *RIOT*'s defines not to override the system defines, the native platforms code is translated with a limited header path inclusion (only *core* paths are included). Then, in order to allow for local definitions of the host APIs native uses, the native platform makes all the system calls available under a different name as shown in figure 3.22.

```
1 *(void **)(&real_getpid) = dlsym(RTLD_NEXT, "getpid");
```

Figure 3.22.: Example system call wrapper to allow native to access *getpid* and *RIOT* to define it differently.

Finally, it is possible to do system calls, either directly or indirectly via some library function. Due to asynchronously context switches, system calls need to be guarded. If an application developer calls such a system function, it can lead to undefined behavior.

Alternatives to Native Libraries

The best solution to safeguard against this would be the use of a non-native C library for linking *RIOT* and the application, and provide native system calls only through an API defined by the *native* implementation. This would have the additional benefit of allowing for use of the same C library implementation on *native* and embedded platforms. Such a solution has not been implemented but provides an interesting topic for follow-up work on the *native* platform.

3.14.2 OS X

During the time the native port was implemented and tested, several particularities of the Mac OS X platform led to problems that did not arise on the other platforms[21]. While GCC has the ability to specify which header files should be given preference, the clang that ships with OS X not only omits such an option, it furthermore includes all sorts of system headers in the compiler itself. This leads to a situation where it is not possible to override some

[21] Linux and FreeBSD

system definitions (e.g. POSIX threads) and not others (e.g. stdio), because one can only either include all system headers or not. Due to this unruly behavior, it is often necessary to take special care of this system.

CHAPTER 4

Evaluation of Functionality, Performance and Impact of the *native* Platform

In this chapter, the *native* platform is evaluated in regard to the goals set out in chapter 2.

First and foremost, it should provide a virtual platform which enables faster development by alleviating problems associated with software development for embedded systems. These problems include availability and usefulness of analysis and debugging tools, reproducibility of results, especially in the domain of wireless networking, and cost of ownership.

The virtual platform should also provide a reference "hardware" which can be used as a common ground by developers in the community.

4.1 Functional Analysis

The implementation of the virtualization has two facets: compatibility with the existing OS, i.e. *RIOT*, and support for development tools. While the main development effort went into implementing *RIOT*'s APIs, some work was also necessary to enable the use of development tools. This section summarizes and evaluates which and how the goals for this thesis are met in the implementation.

4.1.1 Virtualization

The primary goal of this thesis was the virtualization of the OS. No embedded hardware is needed to run *native RIOT* processes, insofar this has been achieved. However, a small function written in assembler is needed for each architecture that should run *RIOT*. This could be written for all architectures without much effort, however it will probably not be necessary in the foreseeable future as x86 and ARM will most likely stay the predominant ISAs for development systems. Another constraint is, that the Windows OS is not supported. To overcome this, developers running Windows as the native OS can use a virtual Linux instance to run *native RIOT* processes.

As for the virtual platform, a typical embedded board has been implemented. It features

virtual timers, networking, energy meter and very rudimentary flash memory. The result is a hardware platform that can be used to run any application that uses *RIOT*'s APIs.

Because the *native* port is just another target board from the OS' point of view, no changes are required to an application in order to use it. The virtualization happens at the call level and is only API compliant. This means that no specific hardware features are simulated, but merely an equivalent output for an input is being generated.

A *native RIOT* process behaves equivalent to the same application running on embedded hardware with regard to execution order given that events occur in the same order. This means that a *native* process can have an equivalent execution as one running on an IoT device.

4.1.2 Network Virtualization

By employing Linux' sophisticated network emulation engine and firewall rules, virtual topologies can be created on top of the logical topology.

Using virtual *RIOT* instances as *routers* in *DES-Virt*, somewhat realistic as well as ideal network scenarios can be created for testing and research. This happens completely transparent to the *RIOT* network stack. Therefore the findings obtained through such experiments are immediately available for verification in a testbed with nodes. This is a huge advantage over the use of traditional simulators, where two software stacks have to be maintained - one in the product and one in the simulator.

4.2 Support for Development Tools

The development tools outlined in section 2.5.3 have been enabled.

4.2.1 Coverage

To quantify the usefulness of the virtual platform for testing *RIOT*, a rough estimate for potential coverage is in order.

Figure 4.1 show the amount of source code lines in *RIOT* in proportion for different parts of the system. The chart compares the average size of a platform to the system core, which is always used in full, and to the total amount of all drivers and system modules. It is assumed that all system libraries are principally able to be tested with *native*. In the best case, i.e. all drivers could be tested with *native*, only 6.23% of the code an average platform could potentially run can not be tested with *native*. In the worst case, i.e. no drivers can be tested with *native*, 25.25% of the code an average platform could potentially run can not be tested with *native*.

While it must be noted that the assumptions about the ability to test system and driver code are rough, it is very likely that the true amount of source code lines that can be tested with *native* lies between 74.75% and 93.77% for the average platform.

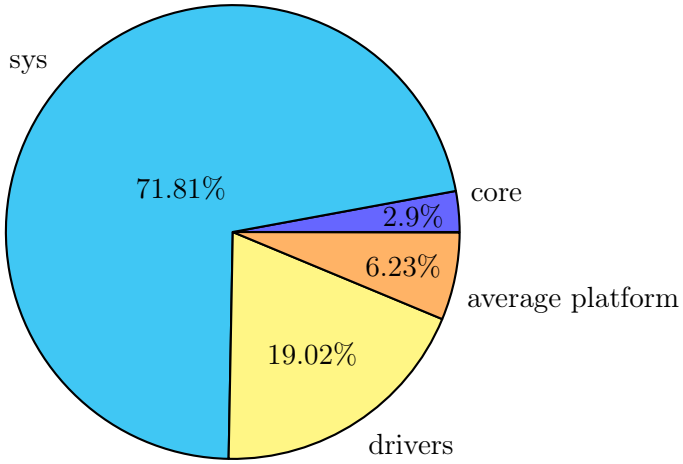

Figure 4.1.: Number of Lines per Category in RIOT

4.2.2 GDB

The use of GDB is possible without any restrictions. The standard *make* target "debug" can be used to run the debugger including required arguments effortlessly. In contrast to embedded hardware, a *native* GDB session is faster, cheaper and possibly more capable. Depending on the embedded platform's respective debugger and debugger software interface, which happens to provide a GDB server for most platforms, a *native* GDB session on a modern *x86* desktop computer supports all features of GDB like record and replay. Even when the same features are available on the target platform, their use is often too slow to be effective.

These differences make the debugger more valuable in the case of problems that are not specific to the hardware platform in question. Not only is time saved, it can even be used more efficiently. Finally the cost for dedicated hardware can be reduced. In a development team that would usually need one debugger per developer, the possibility to debug hardware unrelated issues natively will make sharing of a few debuggers possible. As the statistics show that most software is hardware independent, the impact of this one benefit of the *native* platform alone should not be underestimated.

4.2.3 Valgrind Memcheck

While the benefits of *native* GDB feature support over embedded debugging can be significant already, the Valgrind tool *Memcheck* is simply unavailable otherwise. It's use with the *native* port is straightforward and its findings have proved invaluable may a time already. Special *make* targets "all-valgrind" and "term-valgrind" have been provided to increase usability.

4.2.4 Cachegrind and gprof

The use of the execution profilers Cachegrind and gprof is straightforward, special *make* targets[1] are provided to make their use more accessible. Using these tools for runtime analysis can help find performance bottlenecks. Although the available resources, especially regarding processing power, will make absolute timings meaningless, relating the amount of time spent in different functions to each other might be useful. One unresolved issue with these tools is, that a significant amount of time is spent in the *native* abstraction layer when looking at typical IoT applications. While it should be possible to exclude these parts from the profile, the task has not been looked into further.

4.2.5 Wireshark

Thanks to the use of tap devices for networking[2], traffic can be monitored in real-time. Due to the fact that *Wireshark* has parsers for many protocols, it is possible to test new implementations against it. This has already proven useful to find errors in *RIOT*'s network stack, as Wireshark highlights malformed packets or checksum failures. While it is possible to monitor real traffic as well, the advantage of being able to monitor the traffic of network protocols immediately without the need to implement (and debug) traffic monitors early on in the development process might be substantial. In the case of new protocols, Wireshark can be extended with "dissectors" [66, Chapter 11.1]. As it is also possible to use Wireshark as an analyzer for captures from physical networks [67] the results from emulation and real hardware can be compared directly.

4.3 Performance of the *native* Platform

As execution speed is important for usability, the following sections evaluate how *native* relates to other boards in terms of execution and deployment. To estimate the possible size of testbeds for a given host system, *native*'s resource requirements are analyzed as well.

4.3.1 Execution Time and Memory

One reason for using a *native* call level implementation over support for an emulator such as QEMU was minimizing overhead. In this section an attempt to measure and relate relevant metrics is made.

One big issue is memory consumption. The less memory one instance uses, the more instances can be run in parallel.

Memory consumption was derived by measuring the amount of free space left after starting a large number of instances. No other applications were run in parallel so as to get a relatively good approximation. The scripts used to perform the measurements can be found in section A.

[1] "all-cachegrind", "term-cachegrind" and "eval-cachegrind" as well as "all-gprof", "term-gprof" and "eval-gprof"

[2] This is in contrast to a home brew network layer like Contiki's cooja where one needs to export the pcap traces

In order to minimize overhead, QEMU is started without a GUI and with a telnet serial interface. Using any other serial interface would require a running counterpart to connect to and therefore increase resource usage[3]. The amount of memory used for the virtual machine (2MB) was the minimum which allowed the machine to start, a lower amount would lead to a crash during boot.

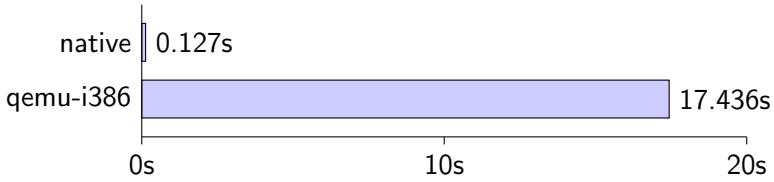

Figure 4.2.: startup time for 100 hello-world instances

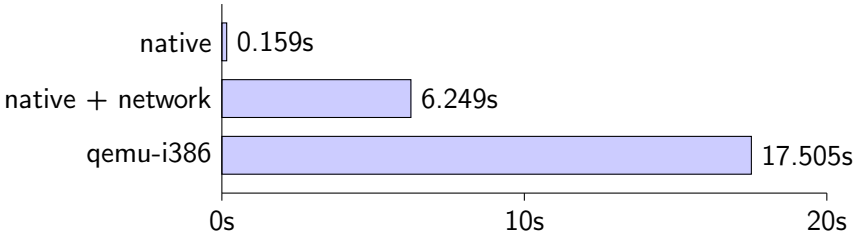

Figure 4.3.: startup time for 100 default instances

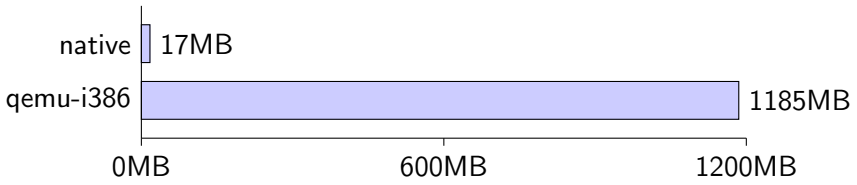

Figure 4.4.: memory consumption for 100 hello-world instances

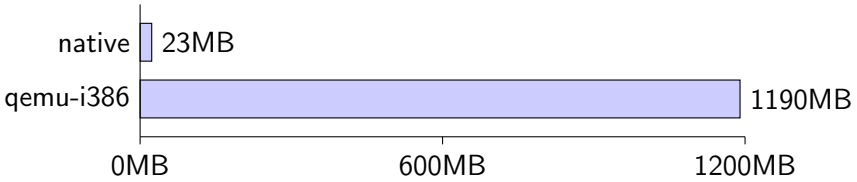

Figure 4.5.: memory consumption for 100 default instances

Figure 4.3 relates startup times for *native* and *QEMU* instances. It shows, that *native* instances start over 100 times faster than their *QEMU* counterparts. Also, the fraction of a second needed to start 100 *native* instances is barely noticeable, while *QEMU* needs over 17 seconds. It is noteworthy, that this exceeds the 15 second barrier after which developers start loosing their concentration as outlined in section 2.9.

Finally, the time needed for network setup (which would be needed by both platforms if *QEMU* had network support as it employs the same network technology), is about 40 times

[3] The *TCP* and *UDP* serial interface implementations for example need to connect to a server, and the stdio implementation wouldn't start when detached from a terminal.

slower than *native* starts. However, the network setup is needed only once after a host system reboot and can be reused infinitely afterwards.

Figure 4.5 relates the memory consumption of *native* and *QEMU*. It shows a static overhead of about 12 MB per instance for *QEMU* compared to *native*. The memory required for the actual application seems analogous for both virtualization solution with an about 230 KB per instance for the default application.

4.3.2 Compile And Deploy Time

Measuring timings was done with the *time* utility as shown in section A.2[4]. Figures 4.6 4.7 list results for some representative platforms and applications.

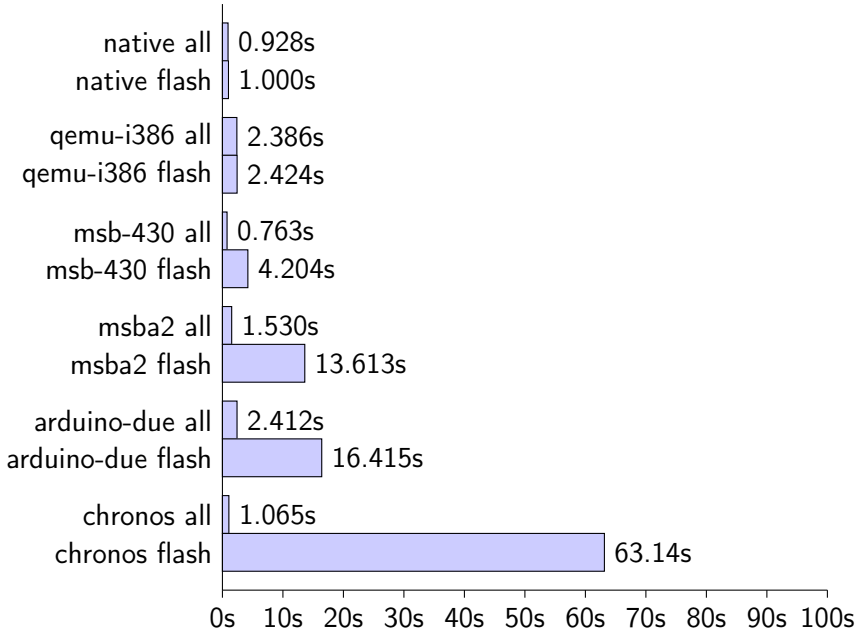

Figure 4.6.: timings for compiling and flashing the hello-world application

The timings themselves are not revealing anything that would not have been suspected. Building lasts comparable amounts of times on all boards, while flashing does not take any time on *native* and *qemu-i386*. This is due to the fact, that neither of these platforms copy the binary.

The discrepancies between build times have mainly to do with the fact that different platforms have different hardware support. For example, the *native* board builds faster than the *qemu-i386* board as all hardware interaction is abstracted at the call level. The QEMU

[4] At the time of this writing, the build system is always checking every path in the source tree for changes before flashing, as the "flash" target depends on the "all" target. While it would have been possible to run the flash utility manually, thus circumventing the extra delay imposed by the build system, that is not usually done by a developer. In any case, the difference between the "all" and "flash" timings is exactly the time a manual flash would yield.

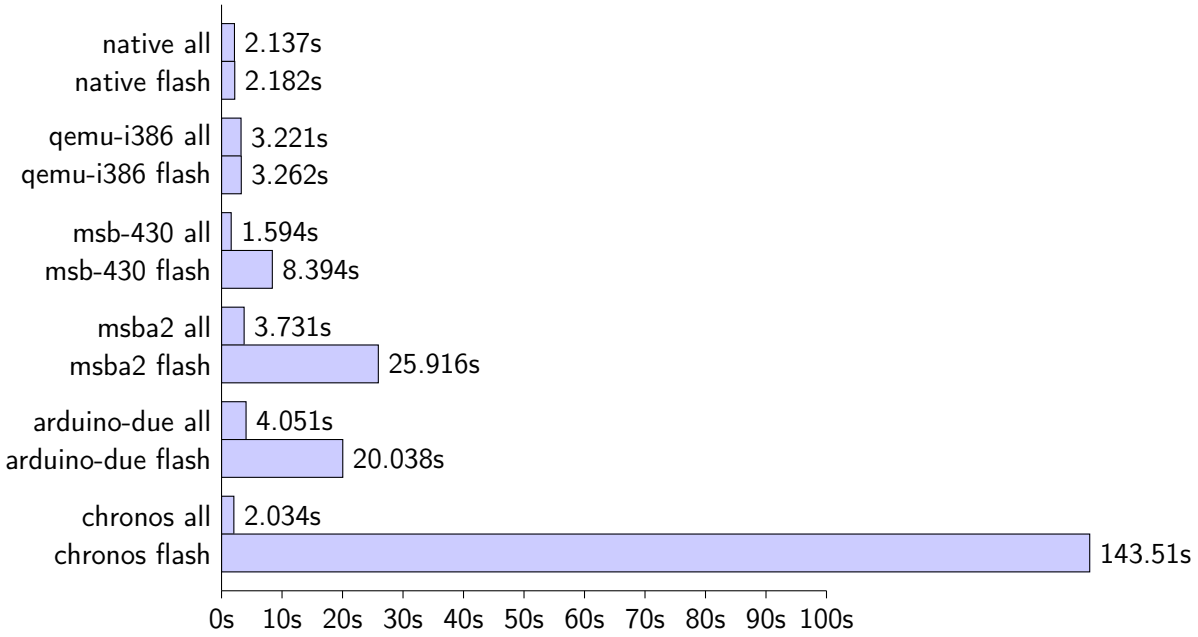

Figure 4.7.: timings for compiling and flashing the default application

board on the other hand implements complete drivers including the likes of PCI which is not even emulated on the *native* platform. The obvious conclusion here is that a simpler platform is quicker to build, and the *native* platform strives to be as simple as possible. Other than that, the tools fair comparably when it comes to build timings.

The impact of not needing to flash at all varies with the target board and the size of the application. Although one could easily miss the difference between *native* and *msb-430* for the *hello-world* application, the time it takes to flash the default application onto the *chronos* board is dramatic in contrast.

While the absolute time savings of the virtual boards are not that large compared to most physical platforms, it is time that a developer will only spend waiting. Again, it is noteworthy that for the default application, most physical platforms exceed the 15 second barrier after which developers loose their concentration as outlined in section 2.9.

4.3.3 Runtime Behavior

In order to determine the relative speed of the *native* CPU compared to an embedded system, *native* and *msba2* platforms were chose to perform a task with comparable computational cost in the same duration of wall-clock time.

The test was created in two phases: In the first phase, the maximum packet transfer rate on the *msba2* for a given payload size was determined so that a packet delivery ratio of 1 was guaranteed. Then, a certain number of packets were transfered using this speed and the CPU utilization (figures 4.8 4.9) was measured on the sending as well as on the receiving side. The second phase was done on both, the *msba2* as well as the *native* board.

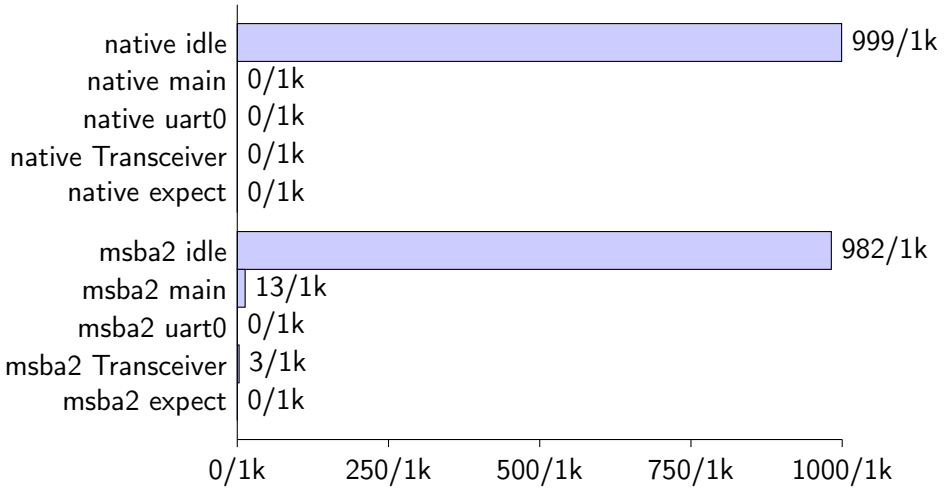

Figure 4.8.: sender runtime

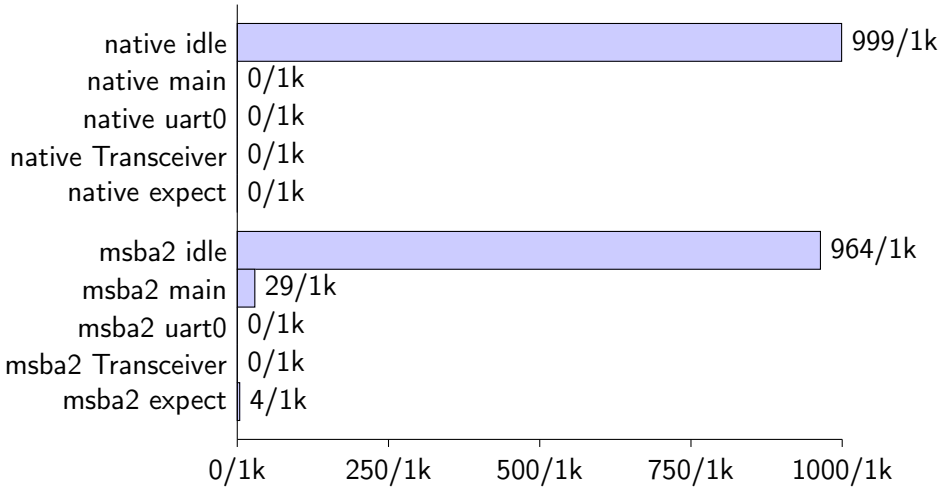

Figure 4.9.: recipient runtime

This test shows, that *native* and *msba2* fare identically, lest for the CPU utilization. They both have about the same increase in context switches (figures 4.10 4.11) during the tests. As *native* has no means of throttling the processor time an instance gets, and the test machines processor is far more powerful than the *msba2*'s, each instance sees significantly less CPU utilization.

When looking at the runtime of a *native* instance it is noteworthy that the idle process does not actually use any resources on the host machine. To look at this from a different angel: the time spent in the idle process is time that could be used by other virtual instances to run one of their non-idle processes. Also, it should be noted that the way the runtime is accounted for is not very precise. All the time spent servicing an interrupt (or signal in the case of *native*) is accounted for in the thread that ran before the interrupt occurred. As the idle thread is the thread which runs the most, it is also the one the gets most of the ISR time.

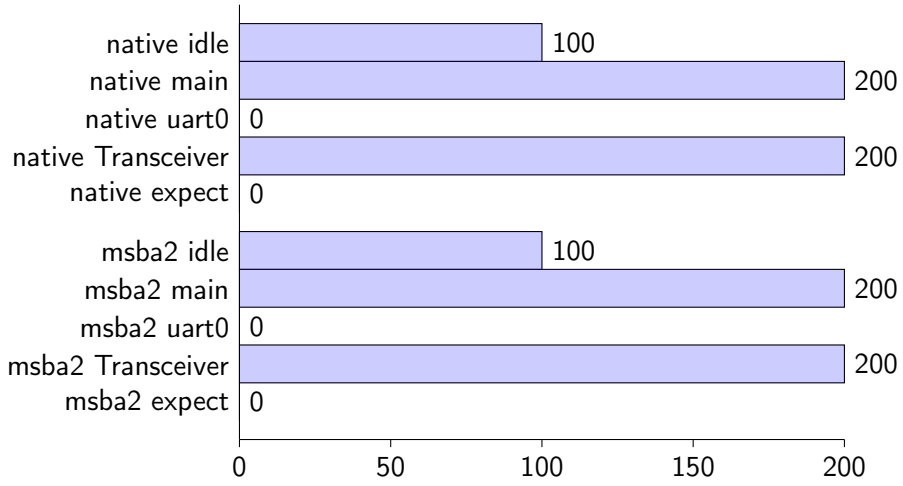

Figure 4.10.: sender context switches

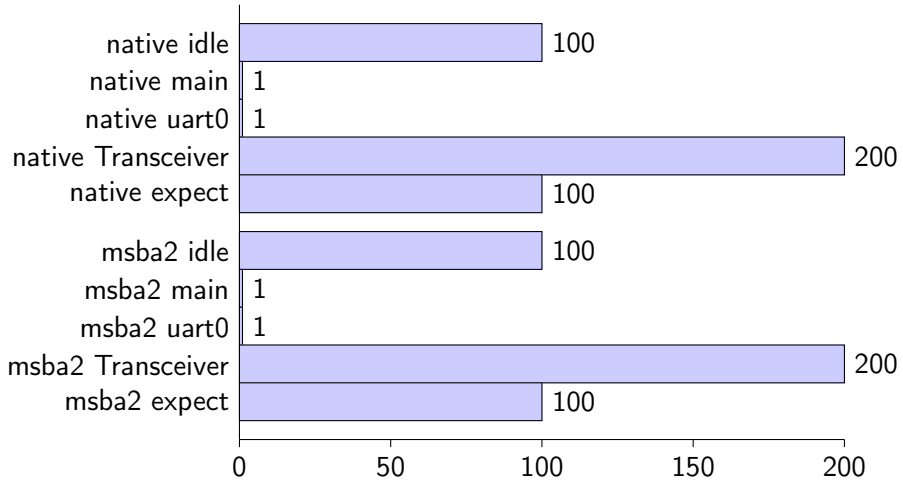

Figure 4.11.: recipient context switches

4.3.4 Testbed Size

In order to determine the maximum size of a virtual testbed, it is not sufficient to calculate how much memory a node needs. Each node needs some processor time when it executes. In this context, executing means that the node is not executing *RIOT*'s idle thread, i.e. waiting for a signal. Therefore it is necessary to factor in the actual application that is being run. An application that is executing in a busy loop or sending data continuously would use all the available computing resources of one CPU core of the host system. On the other hand, an application that just waits for an event uses no resources but memory[5].

More important than CPU and RAM however, is the limit the host OS puts on its virtual switch implementation. *Linux* for example has a statically defined maximum amount of

[5] This is not entirely true in case *RIOT*'s *vtimer* module is used. At the time of this writing, this module has a *longterm timer* that fires every half hour. In this regard, a very large number of virtual instances could end up using a significant amount of processing power... also, if enabled, the *LTC* driver fires continuously.

1024 interfaces per bridge. Changing this limit would require patching and recompilation of the kernel. As 1024 *native* instances of the default application would only require about 235 MB of RAM and no CPU when idling, this is a limit that will be met long before the host system is maxed out performance-wise.

4.4 User Feedback

After the *native* platform had been in use for a while, interviews were led with a representative selection of four active members of the community. Three out of the four members were using the *native* platform.

The single developer not using it was only involved with writing drivers and could not use the platform for its lack of physical hardware support.

4.4.1 Use Cases

native was used in the following scenarios by the interviewees:

- checking how an implementation behaves: 1/3
- fallback when hardware broken: 1/3
- implementing network protocols: 2/3
- implementing system and core modules: 1/3
- implementing applications: 1/3

4.4.2 Tool Utilisation

The interviewees were asked about their utilization of the various tools which are enabled by *native*:

- DES-Virt: 2/3
- GDB: 3/3
- Valgrind: 3/3
- Wireshark: 2/3

The developer not using *DES-Virt* and *Wireshar* did not work on any network related code.

4.4.3 Problems

When asked about problems, the interviewees reported the following problems with *native*:

- random crashes: 1/3
- Mac OS X specific problems: 1/3
- too few warnings: 1/3
- no support for thread analysis: 1/3

4.4.4 Highligths

When asked about the strengths of the *native* platform, the following features were mentioned:

- no need for flashing/hardware: 2/3
- Wireshark: 2/3

4.5 Problems

There are a couple of issues with the current state of the implementation:

- On Mac OS X, Valgrind Memcheck does not work in combination with *nativenet*.
- Also on Mac OS X, the *nativenet* module fails to work after transmitting of one packet in each direction.
- The *reboot* command fails (i.e. the process terminates with an error) for certain UART configurations.
- When a floating point operation is interrupted, the process terminates with an error.
- The transparency of the native C library can require tricks for includes that exist in *RIOT* (in particular in the POSIX components) and in the host system.

Adding support for a dedicated C library is probably relatively straightforward and will resolve any issues with includes for POSIX etc. and conflicting implementations. When using the dedicated C library, for example *newlib*, it is possible to instruct the compiler to not use any system includes during the compilation and linking stages. This would also make it impossible to accidentally use some external library that is not available in *RIOT*.

In particular, the floating point problem could probably be avoided by doing floating point operations in software instead. Although a compiler switch to not use hardware support for floating point operations exists, this still requires the use of an external library as neither GCC nor clang have built-in support for software floating point operations[6]. It would probably make most sense to resolve this issue along with adding a dedicated C library.

Support for the reboot command in the UART module is under development as a side effect of the implementation for GPIO interrupt support mentioned in section 3.10.3. The same might be true for the Mac OS X transceiver problem which dwarfs in contrast to the other OS X problem. Also, as there is no proper support for testbed virtualization in OS X, the usefulness of support effort for that platform is questionable in general.

The Mac OS X Valgrind problem is apparently very specific to the application as web searches did not yield any related results at the time of the problems manifestation.

[6] GCC has such a library, but it is not built by default.

CHAPTER 5

Development Methodology

The *native* platform has been added to *RIOT* with the general goal of improving the development process. This section summarizes recommendations for the development process and particularities of the native platform.

Development of both, applications for *RIOT* as well as *RIOT* itself should happen in two phases.

- Develop using the *native* board, employing all the debugging tools available.
  - For distributed applications, use *DES-Virt* and *Wireshark* with several virtual instances to make sure the network code works as intended.
- Once everything is in a good shape, run it on a physical system.
  - In order to facilitate comparable results, use an open testbed.
  - In the case of network errors, try to identify the defining features and create a minimal version of the failing scenario with *DES-Virt*, starting again at step one.

There is a certain probability that the application will not work as intended on the target platform, although it runs fine on native. The evaluation in 4.4 suggests that this is likely due to errors in the hardware dependent code. With the exception of errors in the native platform itself (which are publicized in the *RIOT*'s issue tracker), it is safe to assume that an application that does not run on native will not run on any other platform.

That being said, it is also possible that the emulator is not reasonable to use.

One reason is the implementation of drivers for specific hardware. While it makes sense to define the interface and a dummy implementation using *native*, the code that actually interfaces with the hardware likely does not. Of course it is possible to implement a simulator for the hardware at hand in order to debug and test the hardware driver, but this is a tedious task. Depending on the hardware interface, it appears more promising to make the hardware itself available within native as outlined in 6.1.5.

5.1 Debugging Tools

5.1.1 Compiler Warnings

The native platform already has all *compiler warnings* enabled. These warnings show up during the compilation process and should be taken seriously and addressed. In general, a warning *does* indicate a defect in the code. Even if the particular application might not trigger the error now, chances are high that one day the issue becomes a problem due to changed invocation, changing compiler optimization levels, changes in the compiler, because it is being used on a different architecture, etc. To enforce attending to compiler warnings, the "−Wfatal−errors" option can be used to make the compiler treat warnings as errors. The native platform does currently not set this option.

5.1.2 memcheck

The *Valgrind memcheck* tool can be used by translating the application with the *all-valgrind*[1] build target and running it with the *term-valgrind* target. Valgrind traces all memory accesses and checks if an access is valid. A reading access is invalid, if the memory location has not been written to before. A writing access is invalid, if the memory location is not within the reserved stack or heap space of the application. Whenever an invalid memory access is detected, it gets reported along with a *backtrace*.

5.1.3 memcheck Options

The *term-valgrind* build target[2] exists for convenience, and sets the following options to increase the usefulness of *memcheck*:

- −−read−var−info=yes
  This causes Valgrind to increase the detail level of error messages.
- −−fullpath−after=/path/to/sources[3],
  Causes Valgrind to print the part of the path to a source file after the given path. This is useful for two reasons: First, it is possible that the same filename exists twice, and it is also possible that the same function signature exists in both files. Second, as *RIOT* is relatively large, one might not know which module the respective file is from and this information can be easily read from the path.
- −−track−origins=yes
  Makes Valgrind print explicitly where an undefined value was first created.

[1] It is necessary to rebuild the application completely (i.e. make clean all−valgrind). This is because the *native* port relies on *defines* being set and *make* does not regard these in its decision for recompiling code.
[2] This target is available on the native platform only.
[3] "/path/to/sources" should be set to point to the directory where *RIOT*'s sources are.

5.1.4 memcheck and Debugger

One notable option that is not set by default is −−db−attach=yes which causes Valgrind to offer attaching the process it analyzes to a debugger[4] whenever an error is detected. When the user decides to do so, the debugger is started and the process is being attached in the state before it actually performs the invalid memory access. After inspecting the situation and exiting the debugger, Valgrind picks up the process in the state it was left in the debugger.

5.1.5 Complementing memcheck

Compilers do have the ability to inject additional code into an application in order to do certain checks at runtime. One example is *stack smashing protection* which can be activated by passing the option "−fstack−protector−all"[5]. For the native platform, it is automatically added when the *DEVELHELP* macro is defined. In result, the memory on stack will be guarded with markers which are checked for errors after each call of a function. One effect of this is that every thread needs more stack space for these markers, another is that the execution of the checks takes CPU time. Errors that can be found with this method are complementary to the errors that Valgrinds *memcheck* tool finds because they compose a writing access to valid memory.

5.1.6 Tests

Automated systematic testing is key to writing and maintaining good software.

RIOT has support for the *EMBUnit* [68] unit test framework. When writing software modules, one should always write unit tests to cover their functionality.

For applications, scripted integration tests are recommended. For network applications, this could include an option to run tests in a virtual network as outlined in section 5.2.2.

5.2 Network Development Tools

When developing network protocols, the *DES-Virt* framework should be used to define simple topologies for testing hypothesis prior to testing on physical networks. Only after simple assumptions have been proven to work, more complex scenarios should be tested. While it might be tempting to create a realistic scenario using the virtual testbed, this is hard. The main reason for this is that wireless networks do have very complex features. There is a high probability that tests with physical networks yield more results after a certain threshold has been surpassed.

When implementing existing protocols, the *Wireshark* protocol analyzer can be used to verify the correctness of headers and to analyze traffic. A Wireshark "dissector" for the *nativenet* protocol is included with *RIOT*. It is necessary to configure Wireshark to use it

[4] The default is GDB, but a different debugger (or debugger front end) can be specified with the −−db−command=... option.

[5] Compiler options can be passed to the build system by adding them to the *CFLAGS* environment variable.

in order to be able to debug higher layer protocols. Configuration instructions can be found along with the dissector at [69].

5.2.1 Inaccuracies of the emulated network

The network emulation that *DES-Virt* provides is not perfect. There are several aspects, where the expected results might not be met:

- Due to resource sharing, delay can vary in addition to the value set by the framework.
- Loss can occur on the bridge which is used to connect the virtual instances in addition to what is configured through the framework.
- The link speed can be truncated further due to resource sharing.

In addition to these peculiarities of the network emulation software, it is also possible for *RIOT* modules to introduce surprising behavior. For example, the current transceiver implementation can loose packets if the lower layer (i.e. *nativenet* in this case) adds packets to the buffer quicker than the upper layer (i.e. *sixlowpan*) can process them. It is possible to increase the buffer size to reduce the impact of this issue.

5.2.2 *pcap*

For manual analysis, creating *pcap* [61] captures is highly useful. *Wireshark* and *tcpdump* [61] can be used to capture and save network traffic in the *pcap* format for later analysis with one of the many tools [47] which support this format. *pcap* files can be used to save network traffic for comparison with later tests as well as sharing with other members of the community. In this regard, online tools like CloudShark [**?**] can be used to share and analyze the captures online.

Test Automation

For automated testing packet captures could be replayed into a network. The *Wireshark* website lists tools [47] that could help in this regard. While this has not been done in *RIOT* so far, it is a promising option.

5.2.3 Physical Testbeds

Once all the tests are running fine in the defined conditions of the virtual testbed, it is time make sure the implementation also works in the real world. Because not everyone can run their own testbed, open testbeds have been created by the research industry. One open public testbed is the *FIT IoT Lab* [70].

Besides the possibility to run experiments in a large physical network without having to maintain it, the use of an open testbed has an additional advantage: known hardware. As the hardware in the open testbed is known to, and used by the community, it can serve as a reference platform. Having a reference platform means it is possible to narrow down potential sources of failures. If for example a network protocol implementation runs fine in

the testbed but fails to establish any communication on the local platform, it is likely that the problem is in the local physical or data link layer.

Finally, having reference testbed means it possible to reasonably compare results. In **??**, the authors show that many published simulation results are not trustworthy because they make assumptions about the network which are both, far from reality and not clearly stated. Running experiments in an open physical testbed helps alleviate both problems.

5.3 Workflow

To summarize, the workflow is outlined in whole before going into details.

First, do anything possible to make sure defects are found using static analysis.

- Enable all compiler warnings and fix all problems found.
- Use a linter like *cppcheck* to find code smells and identify misuse of standard APIs.

Then, use dynamic analysis tools to find further defects.

- Run the application using the *native* emulator for systematic testing via *unit tests* and *integration tests*.
- Use *Valgrind memcheck* and *stack smashing protection* to make sure there is no undefined behavior as a result of invalid memory accesses above the hardware layer.

In case of network applications or protocols, make sure the network code is fine in theory.

- Use *DES-Virt* to create defined network test scenarios.
- Use *Wireshark* to make sure network protocols are syntactically correct.

In case of failures, use analysis tools to find defects.

- Use *GDB* to analyze the system state in case of semantic errors. It has many powerful features that are only available on desktop hardware like virtually unlimited conditional breakpoints, record and replay, catchpoints, tracepoints and watchpoints[6].
- Use *Wireshark* analyze network communication patterns in case of network problems.
- Use *gprof* or *cachegrind* for profiling in case of resource problems on the target hardware.

Once the implementation has proven error-free in the emulator, test it on the target platform as well as in a physical testbed. After analyzing the failure and attempting to fix the defect, repeat all steps to make sure the fix did not itself introduce a new defect.

5.4 Differences Between Emulated and Embedded Systems

There are three noteworthy differences between the typical IoT device and native.

- memory size (both flash and RAM)
- processing power

[6] Refer to the *GDB* manual [71]

- features of the (virtual) medium

When looking at results, especially when it comes to performance measurements, these points should always be kept in mind. Again, while it is possible that an application can run on the native platform but not on real hardware due to limitations of the hardware, the reverse is not true. In short: A virtual instance typically has more memory, more processing power, and a "better" medium. The finer points of the differences between virtual and physical platforms are outlined in the following subsections.

5.4.1 Memory and Processing Performance Considerations

When switching from native to a physical board, stack sizes should be kept an eye on. If the board allows for it, the *ps* command can be used to quickly get an overview of the boards stack needs.

As for performance, there are two particularities. On one hand, a modern desktop computer has many times the processing power an embedded CPU has. On the other hand, it has more advanced features and they tend to be supported well. One of these features is a richer instruction set, another is a memory management unit (MMU), and then there is frequency scaling and resource sharing. Also, the native platform is typically one of many processes running on the hosts CPU.

Frequency scaling and resource sharing (i.e. multiprocessing) lead to a greatly varying numbers of instructions per second for the virtual *RIOT* instance. Memory protection means that errors in memory access "only" lead to segmentation faults for the virtual instance, while it might easily have arbitrary effects on a physical system because parts of the periphery can be accessed by writing to specific memory locations.

The richer ISAs of a modern CPUs mean that the same line of source code could translate to one line of machine code[7] for the native platform, while it has hundreds of lines on an embedded device. A typical example for this are floating point operations. Desktop computers have dedicated floating point units (FPUs) as part of their CPUs while embedded devices often do not have one.

Due to the differences in performance, the same application can behave differently, even for the same input. If, for example, a timer is set to stop the generation of prime numbers after one second, the result will be different depending on how many iterations fit into that second. While this is a somewhat artificial example, the underlying principal can be found in many applications. Although the scheduler and application logic is deterministic, different executions can result whenever external states, like sensor readings[8], are involved.

5.4.2 Emulated Network Considerations

When developing network applications, the differences between emulated and embedded systems are magnified.

[7] I.e.: one instruction along with its parameters.

[8] It is noteworthy that a timer is also a kind of sensor, i.e. one that measures time and can trigger when a certain time value has been reached.

Link Speed

While the data rates of typical IoT transceivers are relatively low, a native instance could theoretically transmit data at the speed of the host CPUs bus. When using the *DES-Virt* framework, the speed of links can be limited. Although this helps to increase realism, the speed limit is only ever between two nodes. Due to this limitation, the total maximum bandwidth per node is the sum of the bandwidth limits to all its neighbours.

Loss

Another difference of the medium is loss. Loss can also be defined on a link basis in *DES-Virt*. But here again, the limitations of the framework result in a distorted behavior compared to physical networks. The loss is applied on a link basis according to some probabilistic distribution. In this model, the loss per link probabilities are unrelated to each other which is not the case in the physical world.

Delay

The third parameter that can be modified in *DES-Virt* is delay. In this case, the deviation from a realistic model is not only that this is set independently on a per link basis, but also that the delay is fixed. For typical IoT transceivers, a higher variation in delay is common because of the shared medium. Whenever a transmission is initiated, the transceiver waits until it can not detect other radio waves before starting to send. Therefore, a more realistic model would make the delay a function of the active transmissions in its vicinity. As outlined in 6.1.1, a network simulator would be useful for development of applications when a more realistic model of the medium matters.

MTU

Last but not least, the data units in *native* are larger per default than on the typical IoT hardware. While *nativenet* over Ethernet uses its full *maximum transfer unit (MTU)* of 1500 bytes[9], a typical IoT transceiver offers much smaller payload sizes. Depending on technology this can vary drastically – 802.15.4 for example defines 127, and Bluetooth low energy (BLE) 23 bytes of payload. To make experimentation easier, the MTU of the *nativenet* transceiver can be controlled at compile time by overriding the "NATIVE_MAX_DATA_LENGTH" preprocessor macro.

Effects on Protocols

The sum of these particularities can lead to network protocols behaving very differently compared to the same protocol running on an embedded system. For example, on an unlimited medium, it is easily possible that the transceiver buffer is swamped when its neighbours send data quickly. The effects can also become very complex, like for example

[9] After subtracting the *nativenet* header, 1496 bytes remain for the payload.

growing window sizes of TCP. Routing protocols are another example where complex effect chains are to be expected.

CHAPTER 6

Conclusion

The goals set out for this thesis have largely been met. With the native port, it is possible to run regular applications written for *RIOT* as a native process under Linux, OS X and FreeBSD. This enables the use of development tools that were not available for embedded system software development, and significantly lowers the cost and usefulness for ones that do exist.

The GNU debugger GDB can be used to inspect a running *RIOT* process without the need for additional hardware. Automatic runtime verification of memory accesses has been enabled through support for the Valgrind tool memcheck which is not usable on embedded systems at all. The profilers gprof and cachegrind are usable without modifications to the program under surveillance thus eliminating the additional work required for profiling on embedded platforms. The use of tap networking has enabled the inclusion of *RIOT* in the virtual testbed framework *DES-Virt*. Using *DES-Virt* it is possible to create arbitrary network topologies with defined properties running vast numbers of *RIOT* native processes.

Applying all of these tools methodologically during development promises to lower development costs by finding some runtime errors automatically, making debugging more efficient, and allowing for well defined network testing.

One of the main challenges in the virtualization of *RIOT* was the fact that it uses a preemptive scheduling strategy. As there is no built-in support for preemptive threading in either POSIX or Linux, a means for preemptive threading has been implemented by using undocumented features of the POSIX signal handling implementations that are available in at least Linux, OS X and FreeBSD.

Although difficult to measure scientifically, the native port has already proven itself as an invaluable instrument for development and testing of *RIOT* which is reflected by its widespread adoption within the community.

6.1 Perspectives

The native platform presented in this thesis allows for basic functionality tests of hardware independent code, and for emulation of arbitrary network topologies. In order to build a

fully integrated testing framework for *RIOT*, some pieces are missing. The following sections describe these missing pieces and comment on methods for their integration where this is possible.

6.1.1 Adaption Layer for ns-3

As outlined in section 2.8, a network emulator is sufficient for testing purposes, but cannot replace a network simulator. The open source network simulator *ns-3* [72] appears to be an ideal candidate for this task. On one hand, Contiki has already been ported to it, and on the other hand, it is the most widely used network simulation tool. This would not only provide a basis for a *RIOT* port implementation wise, but would also allow for better grounded evaluations between implementations. Another interesting treat is that it would be possible to implement interfaces for different layers in *RIOT* that could then provide access to parts of the *ns-3* protocol stack. For example, one might add a 6LoWPAN interface to *RIOT* and evaluate the implementations in ns-3 and *RIOT* for performance and functionality using the same test application.

One possibility to integrate *RIOT* with ns-3 is to use the native platform as a basis and adapt it to communicate with ns-3 instead of the host OS. Such a *RIOT* instance would then still run as process in the host OS, using basically the same implementation as the native platform, but all of its system calls would be substituted with code that communicates with ns-3. For example, instead of sending a network packet on a tap interface, it would be passed to the ns-3 scheduler, and instead of getting the hosts OS's system clock, the ns-3 scheduler would be queried. In doing so, ns-3 can synchronize time, and network events across the *RIOT* instances.

The architecture would consist of two principal parts:

- A proxy node within ns-3 which communicates with the *RIOT* process, and
- proxy hardware interfaces within *RIOT* that communicate with their counterpart in the ns-3 process.

This approach has already been implemented for Contiki[1] [**?**], so it should be possible to reuse some of it.

6.1.2 Event Framework

Another missing piece for a proper testing framework is an event generator.

One candidate is *Node-RED* [73], which is an easy to use general purpose scripting framework. *Node-RED* has a web based visual editor and is centered around *flows* of messages between *nodes*. It's *nodes* produce and/or interpret messages which *flow* from one *node* to another. *Nodes* that are interesting for testing purposes include *UART* and *TCP* input/output (for system interaction), storage output (for logging), timers (for generating timed events), delay and trigger (for event based timed events), and several logic (for decisions) types.

A simple solution for interacting with a *RIOT* instance for testing purposes, is scripted use

[1] It has not been released into the public as of this writing.

of the *RIOT* shell. For *RIOT* native instances, this can be done via natives stdio redirection to TCP sockets. In order to reuse these tests on embedded hardware, some software that makes serial interfaces accessible via TCP could be used.

In order to be more useful, the implementation of additional drivers for *RIOT* native is probably a good idea. Along with the addition of virtual sensors (for things like temperature, and humidity, etc.) a method to set the values of these sensors would need to be implemented.

6.1.3 Integration into libvirt

libvirt [74] is the most versatile open source virtualization management backend. In contrast to systems like *vmware* [75], *libvirt* is agnostic to the actual virtualization software. For easier integration with existing virtualization management software, it would be interesting to add support for the management of native *RIOT* instances to *libvirt*.

6.1.4 Generalization of the Virtual Platform

Probably the most compelling treat of the native port is the ability to use Valgrind's memcheck to test the whole OS. As outlined in section 2.6.2, this is not feasible with other virtualization solutions like QEMU, and not possible on embedded hardware. Most OS' for the IoT will have hardware abstraction layers similar to *RIOT*. Splitting *RIOT*'s native port into a virtualization library so that it can be added to other systems with relatively little effort would bring this (and of course all the other features) to these systems as well. Ideally, this would be implemented either along with the ns-3 port or at least with it in mind, so that a virtualization abstraction framework for the IOT can grow out of it. Furthermore it might be interesting to check if the integration with CASE tools is possible. Nowadays, code generators for embedded devices usually come with their own libraries. For a task like that, defining common abstraction interfaces would be the most important step.

6.1.5 Virtualization of Peripheral Drivers

In order to make the native platform more useful for hardware developers, the implementation of *RIOT*'s peripheral interfaces might be very helpful. By providing access to the host's various Pulse Width Modulation (PWM)[2], Serial Peripheral Interface (SPI)[3], I^2C[4], USART, and analog-to-digital converter (ADC) interfaces, it should be possible to develop drivers for devices that attach to those interfaces. From the view of the device itself, it does not make a difference whether it is attached to a MCU or a more powerful computer. Platforms

[2] The Linux *sysfs* PWM interface [76] can be used to add portable support for PWM devices under Linux. On platforms such as the *Raspberry Pi* which do not come with PWM devices, it is possible to use GPIO devices as PWM devices via a pseudo module [77].

[3] Linux comes with a userspace API named *spidev* [78] which can be used to implement limited but portable access to SPI devices. There is a Linux kernel module which uses GPIO pins to transparently provide access to an SPI bus.

[4] Linux provides support for accessing I^2C devices through its *dev* interface [79], which might be used for very limited but portable I^2C support in *native*. As for SPI, there is also a Linux kernel module which runs I^2C over GPIO pins.

like the *Raspberry Pi* [63], *UDOO* [80], or the *BeagleBoard* [81], which are designed to make these interfaces accessible to developers, could then be used as cheap and more powerful debugging platforms for hardware.

APPENDIX A

Evaluation

A.1 Execution Time and Memory

In order to start instances and measure time, the following shell scripts were used:

```
1 time(
2     for PORT in $(seq 2000 2099); do
3         qemu-system-i386 -daemonize -m 2m -serial telnet::${
              PORT},server,nowait -display none -monitor /dev/null
               -kernel bin/qemu-i386/hello-world.hex;
4     done)
```

Listing A.1: time starting 100 *hello-world* QEMU instances

```
1 time(
2     for X in $(seq 0 99); do
3         ./bin/native/hello-world.hex -d;
4     done)
```

Listing A.2: time starting 100 *hello-world* native instances

```
1 time(
2     for PORT in $(seq 2000 2099); do
3         qemu-system-i386 -daemonize -m 2m -serial telnet::${
              PORT},server,nowait -display none -monitor /dev/null
               -kernel bin/qemu-i386/default.hex;
4     done)
```

Listing A.3: time starting 100 *default* QEMU instances

```
1 time(
2     for X in $(seq 0 99); do
3         ./bin/native/default.elf tap${X} -d -t 200${X};
4     done)
```

```
> tg 100 10 2 2
pid | name        | state    Q | pri | stack (used) location | runtime | switches
  0 | idle        | pending  Q |  31 |  8192 (1084) 0x805e640 | 999/1k |        1
  1 | main        | running  Q |  15 | 16384 (2916) 0x805a640 |   0/1k |       22
  2 | uart0       | bl rx    _ |  14 |  8192 ( 880) 0x80613c0 |   0/1k |       32
  3 | Transceiver | bl rx    _ |  12 | 16384 ( 816) 0x8076c80 |   0/1k |        7
  5 | expect      | sleeping _ |  13 | 16384 ( 436) 0x8071980 |   0/1k |        0
Sending 100 packets of length 10 to 2..done
pid | name        | state    Q | pri | stack (used) location | runtime | switches
  0 | idle        | pending  Q |  31 |  8192 (1084) 0x805e640 | 999/1k |      101
  1 | main        | running  Q |  15 | 16384 (2916) 0x805a640 |   0/1k |      222
  2 | uart0       | bl rx    _ |  14 |  8192 ( 880) 0x80613c0 |   0/1k |       32
  3 | Transceiver | bl rx    _ |  12 | 16384 (2716) 0x8076c80 |   0/1k |      207
  5 | expect      | sleeping _ |  13 | 16384 ( 436) 0x8071980 |   0/1k |        0
```

Figure A.1.: native sender

Listing A.4: time starting 100 *default* native instances (without network)

```
time(
    ../../cpu/native/tapsetup.sh create 100;
    for X in $(seq 0 99); do
        ./bin/native/default.elf tap${X} -d -t 200${X};
    done)
```

Listing A.5: time starting 100 *default* native instances (with network)

A.2 Compile and Deploy Time

```
BOARDS="native qemu-i386 msba2 msb-430 chronos arduino-due"
for BOARD in ${BOARDS}; do
    export BOARD; echo ${BOARD}
    make clean; time make all
    make clean; time make flash
done
```

```
> server 100 1 2
setting variables.. registering server..registering at transceiver.. done
pid | name        | state    Q | pri | stack (used) location | runtime | switches
  0 | idle        | pending  Q |  31 |  8192 (1084) 0x805e640 | 999/1k |        1
  1 | main        | pending  Q |  15 | 16384 (2660) 0x805a640 |   0/1k |       23
  2 | uart0       | bl rx    _ |  14 |  8192 ( 880) 0x80613c0 |   0/1k |       34
  3 | Transceiver | bl rx    _ |  12 | 16384 ( 816) 0x8076c80 |   0/1k |        7
  5 | expect      | running  Q |  13 | 16384 (2436) 0x8071980 |   0/1k |        1
expecting 100 packets
pid | name        | state    Q | pri | stack (used) location | runtime | switches
  0 | idle        | pending  Q |  31 |  8192 (1084) 0x805e640 | 999/1k |      101
  1 | main        | bl reply _ |  15 | 16384 (2660) 0x805a640 |   0/1k |       24
  2 | uart0       | bl rx    _ |  14 |  8192 ( 880) 0x80613c0 |   0/1k |       35
  3 | Transceiver | bl rx    _ |  12 | 16384 ( 864) 0x8076c80 |   0/1k |      207
  5 | expect      | running  Q |  13 | 16384 (2436) 0x8071980 |   0/1k |      101
```

Figure A.2.: native recipient

```
> tg 100 10 2 2
pid | name        | state    Q | pri | stack (used) location | runtime | switches
  0 | idle        | pending  Q |  31 |   160 ( 148) 0x4000004c | 988/1k |      31
  1 | main        | running  Q |  15 |  2560 (1000) 0x400000ec |  11/1k |      42
  2 | uart0       | bl rx    _ |  14 |   512 ( 296) 0x40000d18 |   0/1k |      90
  3 | Transceiver | bl rx    _ |  12 |   512 ( 300) 0x40002980 |   0/1k |      13
  5 | expect      | sleeping _ |  13 |  2560 ( 140) 0x40001c30 |   0/1k |       0
Sending 100 packets of length 10 to 2..done
pid | name        | state    Q | pri | stack (used) location | runtime | switches
  0 | idle        | pending  Q |  31 |   160 ( 148) 0x4000004c | 982/1k |     131
  1 | main        | running  Q |  15 |  2560 (1000) 0x400000ec |  13/1k |     242
  2 | uart0       | bl rx    _ |  14 |   512 ( 296) 0x40000d18 |   0/1k |      90
  3 | Transceiver | bl rx    _ |  12 |   512 ( 300) 0x40002980 |   3/1k |     213
  5 | expect      | sleeping _ |  13 |  2560 ( 140) 0x40001c30 |   0/1k |       0
```

Figure A.3.: msba2 sender

```
> server 100 1 2
setting variables.. registering server..registering at transceiver.. done
pid | name        | state    Q | pri | stack (used) location | runtime | switches
  0 | idle        | pending  Q |  31 |   160 ( 148) 0x4000004c | 946/1k |      32
  1 | main        | pending  Q |  15 |  2560 ( 816) 0x400000ec |  51/1k |      43
  2 | uart0       | bl rx    _ |  14 |   512 ( 296) 0x40000d18 |   0/1k |      93
  3 | Transceiver | bl rx    _ |  12 |   512 ( 300) 0x40002980 |   0/1k |      13
  5 | expect      | running  Q |  13 |  2560 ( 680) 0x40001c30 |   0/1k |       1
expecting 100 packets
pid | name        | state    Q | pri | stack (used) location | runtime | switches
  0 | idle        | pending  Q |  31 |   160 ( 148) 0x4000004c | 964/1k |     132
  1 | main        | bl reply _ |  15 |  2560 ( 816) 0x400000ec |  29/1k |      44
  2 | uart0       | bl rx    _ |  14 |   512 ( 296) 0x40000d18 |   0/1k |      94
  3 | Transceiver | bl rx    _ |  12 |   512 ( 316) 0x40002980 |   0/1k |     213
  5 | expect      | running  Q |  13 |  2560 ( 680) 0x40001c30 |   4/1k |     101
```

Figure A.4.: msba2 recipient

```
avsextrem: 6746
msba2: 5414
pttu: 5305
qemu-i386: 2887
redbee-econotag: 2577
native: 2305
stm32f4discovery: 2128
stm32f0discovery: 1809
stm32f3discovery: 1765
udoo: 1632
arduino-due: 1632
chronos: 1509
pca10000: 1190
pca10005: 1186
wsn430-v1_4: 846
msb-430h: 842
telosb: 829
z1: 826
wsn430-v1_3b: 812
mbed_lpc1768: 807
msb-430: 693
```

Figure A.5.: Amount of C source code lines for each board (board/* cpu/*).

```
#!/bin/sh

APP=$1

if [ -z "${APP}" ]; then
    echo "usage: $0 <app>"
    exit 1
fi

BOARDS=$(make -C ${APP} buildtest \
    | grep '^Building' \
    | awk '{print $3}')

#echo ${BOARDS}
for BOARD in ${BOARDS}; do
    echo -n "${BOARD}:"

    DIRS="$(make -C ${APP} BOARD=${BOARD} 2>/dev/null \
        | sed -n -e 's/^"make".*\/RIOT\/\(\(cpu\|boards\)\/[^\/]*\)$/\1/p' \
        | tr '[\n]' '[ ]')"

    #echo -e "\t${DIRS}"
    cloc ${DIRS} | grep '^C ' | awk '{print " "$5}'

    #echo ""
done
```

Figure A.6.: Script to gather the source code metrics par board, for one application.

```
> ps
pid | name                 | state    Q | pri | stack ( used) | location
==27694== Invalid read of size 4
==27694==    at 0x8049CA2: thread_measure_stack_free (core/thread.c:113)
==27694==    by 0x8054460: thread_print_all (sys/ps/ps.c:75)
==27694==    by 0x80524AA: _ps_handler (sys/shell/commands/sc_ps.c:25)
==27694==    by 0x805193B: handle_input_line (sys/shell/shell.c:197)
==27694==    by 0x8051BAA: shell_run (sys/shell/shell.c:275)
==27694==    by 0x8050122: main (examples/default/main.c:171)
==27694==  Location 0x8061078 is 0 bytes inside idle_stack[6968],
==27694==  a global variable declared at kernel_init.c:80
==27694==
  1 | idle                 | pending  Q |  15 |  8192 ( 1224) | 0x805f540
  2 | main                 | running  Q |   7 | 16384 ( 2900) | 0x805b540
  3 | uart0                | bl rx    _ |   6 |  8192 ( 1288) | 0x8076480
  4 | radio                | bl rx    _ |   5 |  8192 (  884) | 0x806c440
  5 | Transceiver          | bl rx    _ |   4 | 16384 (  868) | 0x80723c0
    | SUM                  |            |     | 57344 ( 7164)
```

Figure A.7.: Valgrind memcheck false positive in examples/default

A.3 Undecided Valgrind memcheck Report

Bibliography

[1] H. Kagermann, W. Wahlster, and J. Helbig, Eds., *Umsetzungsempfehlungen für das Zukunftsprojekt Industrie 4.0*. Berlin: Forschungsunion im Stifterverband für die Deutsche Wirtschaft e.V., Oct. 2012. [Online]. Available: http://forschungsunion.de/pdf/industrie_4_0_umsetzungsempfehlungen.pdf

[2] J. Davis, T. Edgar, J. Porter, J. Bernaden, and M. Sarli, "Smart manufacturing, manufacturing intelligence and demand-dynamic performance ," *Computers & Chemical Engineering*, vol. 47, no. 0, pp. 145 – 156, 2012, {FOCAPO} 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0098135412002219

[3] C. Bormann, M. Ersue, and A. Keranen, "Terminology for Constrained-Node Networks," RFC 7228 (Informational), Internet Engineering Task Force, May 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7228.txt

[4] F. Mattern and C. Floerkemeier, "From Active Data Management to Event-based Systems and More," K. Sachs, I. Petrov, and P. Guerrero, Eds. Berlin, Heidelberg: Springer-Verlag, 2010, ch. From the Internet of Computers to the Internet of Things, pp. 242–259. [Online]. Available: http://dl.acm.org/citation.cfm?id=1985625.1985645

[5] A. Dunkels, "Full TCP/IP for 8 Bit Architectures," in *Proceedings of the First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys 2003)*, San Francisco, May 2003. [Online]. Available: http://dunkels.com/adam/mobisys2003.pdf

[6] N. Kushalnagar, G. Montenegro, and C. Schumacher, "IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals," RFC 4919 (Informational), Internet Engineering Task Force, Aug. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4919.txt

[7] ZigBee Alliance. ZigBee for Developers. [Online]. Available: http://www.zigbee.org/zigbee-for-developers/

[8] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey ," *Computer Networks*, vol. 54, no. 15, pp. 2787 – 2805, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128610001568

[9] N. V. Schooenderwoert and R. Morsicato, "Taming the Embedded Tiger - Agile Test Techniques for Embedded Software," in *Proceedings of the Agile Development Conference*, ser. ADC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 120–126. [Online]. Available: http://agile2004.agilealliance.org/files/XR5-1.pdf

[10] J. Grenning, *Test-Driven Development for Embedded C*. Pragmatic Bookshelf, 2011.

[11] Wireshark Foundation. Wireshark. [Online]. Available: https://www.wireshark.org/

[12] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, Nov 2004, pp. 455–462.

[13] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An Operating System for Sensor Networks," in *Ambient Intelligence*, W. Weber, J. Rabaey, and E. Aarts, Eds. Berlin/Heidelberg: Springer Berlin Heidelberg, 2005, ch. 7, pp. 115–148. [Online]. Available: http://dx.doi.org/10.1007/3-540-27139-2_7

[14] Real Time Engineers Ltd. FreeRTOS. [Online]. Available: http://www.freertos.org/RTOS.html

[15] The RIOT community. RIOT. [Online]. Available: http://riot-os.org/

[16] Free Software Foundation. GNU Lesser General Public License, version 2.1. [Online]. Available: https://www.gnu.org/licenses/lgpl-2.1.html

[17] E. Baccelli, O. Hahm, M. Wählisch, M. Günes, and T. Schmidt, "RIOT: One OS to Rule Them All in the IoT," INRIA, Rapport de recherche RR-8176, Dec. 2012. [Online]. Available: http://hal.inria.fr/hal-00768685

[18] A. Zeller, *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009.

[19] The Valgrind Developers. Valgrind. [Online]. Available: http://valgrind.org/

[20] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A Call Graph Execution Profiler," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, ser. SIGPLAN '82. New York, NY, USA: ACM, 1982, pp. 120–126. [Online]. Available: http://docs.freebsd.org/44doc/psd/18.gprof/paper.pdf

[21] The DES-Testbed Team. DES-Virt – the DES-Testbed virtualization framework. [Online]. Available: https://github.com/des-testbed/desvirt

[22] Black Duck Software. RIOT Open Source Project on Open Hub. [Online]. Available: https://www.openhub.net/p/RIOT-OS/contributors/summary

[23] Creative Commons. Creative Commons Legal Code – Attribution 3.0 Unported. [Online]. Available: http://creativecommons.org/licenses/by/3.0/legalcode

[24] H. Will, K. Schleiser, and J. Schiller, "A real-time kernel for wireless sensor networks employed in rescue scenarios," in *Local Computer Networks, 2009. LCN 2009. IEEE 34th Conference on*, Oct 2009, pp. 834–841.

[25] E. Baccelli, G. Bartl, A. Danilkina, V. Ebner, F. Gendry, C. Guettier, O. Hahm, U. Kriegel, G. Hege, M. Palkow, H. Pertersen, T. Schmidt, A. Voisard, M. Wählisch, and H. Ziegler, "Area & Perimeter Surveillance in SAFEST using Sensors and the Internet of Things," in *Workshop Interdisciplinaire sur la Sécurité Globale (WISG2014)*, Troyes, France, Jan. 2014. [Online]. Available: https://hal.inria.fr/hal-00944907

[26] The RIOT community. RIOT – GitHub Pull Requests. [Online]. Available: https://github.com/RIOT-OS/RIOT/pulls

[27] ——. RIOT Mailing Lists. [Online]. Available: http://lists.riot-os.org/

[28] netsplit.de. #riot-os on IRC network freenode. [Online]. Available: http://irc.netsplit.de/channels/details.php?room=%23riot-os&net=freenode

[29] E. Baccelli. RIOT Vision. [Online]. Available: https://github.com/RIOT-OS/RIOT/wiki/RIOT-Vision

[30] The RIOT community. RIOT Development Procedures. [Online]. Available: https://github.com/RIOT-OS/RIOT/wiki/Development-procedures

[31] GitHub Inc. GitHub Help – Using pull requests. [Online]. Available: https://help.github.com/articles/using-pull-requests/

[32] K. J. Stewart and S. Gosain, "The Impact of Ideology on Effectiveness in Open Source Software Development Teams," *MIS Q.*, vol. 30, no. 2, pp. 291–314, Jun. 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=2017307.2017313

[33] ISO, "ISO/IEC 9126-1:2001, Software engineering – Product quality – Part 1: Quality model," International Organization for Standardization, Tech. Rep., 2001. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749

[34] ParaDiSe Labs. DIVINE – Model Checking for Everyone. [Online]. Available: http://divine.fi.muni.cz/index.html

[35] C. D. Udma, "Chapter 16 - Software Development Tools for Embedded Systems ," in *Software Engineering for Embedded Systems*, R. Oshana and M. Kraeling, Eds. Oxford: Newnes, 2013, pp. 511 – 562.

[36] The GDB developers. GDB: The GNU Project Debugger. [Online]. Available: http://www.gnu.org/software/gdb/

[37] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision." in *Proceedings of the USENIX'05 Annual Technical Conference*, 2005. [Online]. Available: http://www.valgrind.org/docs/memcheck2005.pdf

[38] The Valgrind Developers. Valgrind's Tool Suite. [Online]. Available: http://valgrind.org/info/tools.html

[39] A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

[40] The IEEE and The Open Group. The Open Group Base Specifications Issue 7, 2013 Edition. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/

[41] J. Dike, *User Mode Linux*. Prentice Hall, 2006.

[42] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. The Java® Virtual Machine Specification. [Online]. Available: http://web.archive.org/web/20150217194717/http://docs.oracle.com/javase/specs/jvms/se7/html/index.html

[43] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247401

[44] M. Hamdaqa and L. Tahvildari, "Cloud Computing Uncovered: A Research

Landscape," *Advances in Computers*, vol. 86, pp. 41–85, 2012. [Online]. Available: http://dx.doi.org/10.1016/B978-0-12-396535-6.00002-8

[45] D. Kotz, C. Newport, and C. Elliott, "The mistaken axioms of wireless-network research," Dartmouth Computer Science, Tech. Rep., 2003. [Online]. Available: http://www.cs.dartmouth.edu/~dfk/papers/kotz-axioms-tr.pdf

[46] M. Krasnyansky. Universal TUN/TAP device driver. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/tuntap.txt

[47] Wireshark Foundation. Wireshark Tools. [Online]. Available: http://wiki.wireshark.org/Tools

[48] Linux Foundation. netem – Network Emulation. [Online]. Available: https://web.archive.org/web/20141023070826/http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

[49] M. Güneş, B. Blywis, F. Juraschek, and P. Schmidt, "Concept and Design of the Hybrid Distributed Embedded Systems Testbed," Freie Universität Berlin, Tech. Rep. TR-B-08-10, 2008.

[50] O. Hahm, M. Günes, and K. Schleiser, "The DES-Framework - Extending a Wireless Multi-Hop Testbed by virtualization and simulation," in *10th Würzburg Workshop on IP: Joint ITG, ITC, and Euro-NF Workshop "Visions of Future Generation Networks" (EuroView2010)*, Würzburg, Germany, 08/2010 2010. [Online]. Available: http://www.euroview2010.com/data/abstracts/Session1_2_Hahm_Guenes_Multi-Hop_Testbed.pdf

[51] T. DeMarco, *Peopleware : Productive Projects and Teams.* Dorset House Pub, 1999.

[52] Contiki developers. Contiki – Native, minimal net. [Online]. Available: http://web.archive.org/web/20140826101054/https://github.com/contiki-os/contiki/wiki/Native,-minimal-net

[53] [Online]. Available: http://web.archive.org/web/20150302121400/https://github.com/contiki-os/contiki/wiki/An-Introduction-to-Cooja

[54] K. Schleiser, "Entwurf und Entwicklung eines interoperablen Echtzeit-Mikrokernels für drahtlose Sensornetze," Master's thesis, FU-Berlin, 2009.

[55] R. S. Engelschall, "Portable Multithreading," 2000.

[56] Stefan. GCC Bugzilla – Bug 25967 - Add attribute naked for x86. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=25967

[57] The IEEE and The Open Group. The Open Group Base Specifications Issue 7, 2013 Edition – clock_getres. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/functions/clock_gettime.html

[58] ——. The Open Group Base Specifications Issue 7, 2013 Edition – getitimer. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/functions/getitimer.html

[59] The RIOT community. RIOT – Transceiver API. [Online]. Available: http://riot-os.org/api/group__sys__transceiver.html

[60] A. Tanenbaum, *Computer Networks*, 4th ed. Prentice Hall Professional Technical

Reference, 2002.

[61] Tcpdump/Libpcap. TCPDUMP/LIBPCAP public repository. [Online]. Available: http://www.tcpdump.org/

[62] The RIOT community. Virtual RIOT network. [Online]. Available: https://web.archive.org/web/20150221011433/https://github.com/RIOT-OS/RIOT/wiki/Virtual-riot-network

[63] Raspberry Pi Foundation. Raspberry Pi. [Online]. Available: http://www.raspberrypi.org/

[64] [Online]. Available: https://www.kernel.org/doc/Documentation/gpio/sysfs.txt

[65] Valgrind™ Developers. (2012, August) Using and understanding the Valgrind core: Advanced Topics - The Client Request mechanism. [Online]. Available: http://web.archive.org/web/20130816054507/http://valgrind.org/docs/manual/manual-core-adv.html#manual-core-adv.clientreq

[66] U. Lamping, R. Sharpe, and E. Warnicke. (2014) Wireshark User's Guide. [Online]. Available: https://www.wireshark.org/download/docs/user-guide-a4.pdf

[67] W.-B. Pöttner and L. Wolf, "IEEE 802.15.4 packet analysis with Wireshark and off-the-shelf hardware," in *Proceedings of the Seventh International Conference on Networked Sensing Systems (INSS2010)*, Kassel, Germany, June 2010. [Online]. Available: http://www.ibr.cs.tu-bs.de/papers/poettner-inss2010-sniffer.pdf

[68] T. Punkka. embUnit – Embedded Unit Testing Framework. [Online]. Available: http://embunit.sourceforge.net/embunit/

[69] The RIOT community. RIOT wireshark dissector. [Online]. Available: https://github.com/RIOT-OS/RIOT/tree/master/dist/tools/wireshark_dissector

[70] The FIT consortium. FIT/IoT-LAB – Very large scale open wireless sensor network testbed. [Online]. Available: https://www.iot-lab.info/

[71] The GDB developers. Debugging with GDB – Documentation. [Online]. Available: https://sourceware.org/gdb/onlinedocs/gdb/index.html

[72] ns-3 developers. ns-3. [Online]. Available: http://www.nsnam.org/

[73] IBM Corp. Node-RED. [Online]. Available: http://nodered.org/

[74] libvirt development team. libvirt: The virtualization API. [Online]. Available: http://libvirt.org/

[75] VMware, Inc. VMware – Virtualization for Desktop & Server, Application, Public & Hybrid Clouds. [Online]. Available: http://www.vmware.com/

[76] Pulse Width Modulation (PWM) interface. [Online]. Available: https://www.kernel.org/doc/Documentation/pwm.txt

[77] bifferboard. PWM GPIO. [Online]. Available: https://web.archive.org/web/20141023150309/https://sites.google.com/site/bifferboard/Home/pwm-gpio

[78] SPI Dev Interface. [Online]. Available: https://www.kernel.org/doc/Documentation/spi/spidev

[79] I2C Dev Interface. [Online]. Available: https://www.kernel.org/doc/Documentation/

i2c/dev-interface

[80] SECO USA Inc. UDOO: Android Linux Arduino in a tiny single-board computer. [Online]. Available: http://www.udoo.org/

[81] BeagleBoard.org Foundation. BeagleBoard.org. [Online]. Available: http://beagleboard.org/