Virtual Controllers

Dalibor Lovric

Christian Olsson

LUND
UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6022
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden by Tryckeriet i E-huset
Lund 2016

Abstract

Small ARM Cortex CPU based system boards, called controllers, are used in building automation for regulation of heating, ventilation, and air conditioning. A controlling project can incorporate several thousands of these controllers. The controllers communicate with a SCADA system over the TCP/IP protocol. For the purpose of testing the Supervisory Control And Data Acquisition (SCADA) system when communicating with several hundred controllers simultaneously, a software implementation of a controller that can run in multiple instances, is needed. In this thesis, three different kinds of virtual controllers are proposed and evaluated for their performance. The performance data is based on controller's response time and is acquired in a benchmark tool that is simulating SCADA. The implementation work consisted of designing and implementing a benchmark tool and three controller solutions: emulated, ported and simulated. The three solutions differ significantly in the number of instances that can be run simultaneously on the same machine. The conclusion is that the simulated solution is the most suitable since it can run in 6000 instances contra the ported with 200 instances. The emulated solution was eventually deemed as impractical to accomplish in the scope of this thesis.

Keywords: Virtual controllers, RTOS, Building automation, Response time, Porting, SCADA

Acknowledgements

We would like to thank our supervisors Anton Cervin, at LTH and Harald Stribén at Regin AB for their guidance during this project. Further, words of gratitude to Simon Jönson and Urban Fosseus for their support and providing us with their technical knowledge and skills.

Contents

Acronyms

HVAC Heating Ventilation and Air Condition. 2

KVM Kernel-based Virtual Machine. 7, 10–13

lwIP Lightweight IP. 17, 28, 29, 40

PCAP Packet CAPture. 17, 40

QEMU Quick EMUlator. 7, 8, 10–15

RTOS Real Time Operating System. 6, 7

SCADA Supervisory Control And Data Acquisition. i, 1, 2, 4, 18

VM Virtual Machine. 9

VMM Virtual Machine Monitor. 9

Chapter 1
Introduction

In these times of digital revolution, smart homes and buildings are extensively using products for building automation. That is, heating, ventilation, and cooling are automated and centrally controlled, providing more comfortable living and working environment. The main benefits of building automation in large scale facilities such as hotels, offices, and industrial plants are improved life cycle of utilities and reduced energy consumption, and thus operational cost [1]. Furthermore, benefits on reduced energy consumption reduce the negative impact on the climate and decreases environmental pollution.

AB Regin is a company that develops controllers and software used for controlling, monitoring, and testing various systems in building automation. The controllers are embedded systems running a real time operating system and applications for controlling sensors and actuators in building projects. A building project can in some cases incorporate several thousands of these controllers, and can use several hundreds of communication routes when connected with a governing SCADA system [2]. When interacting with such large projects, the system's simultaneous data flow increases the workload in the SCADA system. Regin has identified the need for future projects of sizes that has not been realized yet, and the behaviour of the SCADA system in interaction with such project sizes has to be tested.

This thesis examines the possibilities of developing virtual controllers, software implemented imitations of physical controllers, for testing Regin's SCADA system. The idea is to propose and develop virtual controllers developed using three different techniques, and to evaluate their performance in a simulated test. The examined techniques are emulation, porting, and simulation which are described in detail later in this chapter.

1.1 Background

Building automation is a term that refers to systems of electrical devices that are centrally and automatically controlled. Such systems can consist of multiple subsystems which in

turn consist of multiple distributed networks of electronic devices. This system is used to control a building's Heating Ventilation and Air Condition (HVAC). Modern systems control even lighting, security, fire alarms, and generally anything that is electrical. All these can be provided on a scheduled basis depending on occupancy or the seasons of the year. This reduces the energy consumption, operational and maintenance costs in comparison to a non-automated building.

Regin's software consists of a real time operating system, control applications, an application interpreter, design tools and monitoring applications. The controllers can have functionality for logging data, status reporting, and malfunction alarms. Regin has developed their own communication protocol for the communication, EXOline, that can be used for communication over serial, local, and wide area network. The controllers can communicate with each other and also with a governing system, EXOscada, which is a type of SCADA system [3].

A station is a term describing a system consisting of a station master and any number between zero and up to 254 slave controllers, which are interconnected in some network topology. A station master is a single controller with a master role within its station. A building project is a system consisting of stations. It can incorporate several hundreds of such stations and also several hundreds of communication routes. A communication route is a communication channel between a station master and EXOscada.

When EXOscada is connected to a building project it uses communication routes to communicate with station masters. For a project consisting of hundreds of stations, EXOscada uses hundreds of communication routes for fetching data. For instance, accumulated logs, alarms, and other data can be retrieved. With increased number of communication routes the data flow increases resulting in a higher workload on the CPU and thus longer data queues. The concurrent data flow from the routes into EXOscada implies greater demands on system correctness.

With customer demands for ever larger systems, some that could consist of thousands of controllers and hundreds of communication routes, there is a need to be able to test the EXOscada system in a practical way. Past experiences have shown that bugs can appear in EXOscada in such very large systems. Performing tests when connected to real building project systems is not viable nor acceptable because of the impact the unknown behaviour could cause to the systems. But systems consisting of virtual controllers can offer an opportunity for behaviour testing. It can eliminate the need for an environment with physical controllers and implicitly eliminate the cost for buying and setting up these controllers. An additional benefit of virtual controllers is the possibility of implementing and testing new features. In this way, research and development of physical controllers can be benefited. Customers have also expressed interest in having virtual controllers for testing their project applications without having to use the physical products.

Controllers

The controllers are small CPU based systems with STM32F20x CPU family manufactured by STMicroelectronics [4]. The particular model used in controllers is STM32F207ZE with high-performance ARM Cortex-M3 32-bit RISC core with frequency up to 120 MHz. The built-in memory is high-speed embedded flash memory up to 1 Mbyte, 128 Kbytes of system SRAM, 4 Kbytes of backup SRAM [5].

It provides an extensive range of inputs and outputs and peripherals [6]. It features standard and advanced communication interfaces such as 5 volt input and output ports, UARTs (Universal Synchronous/Asynchronous Receiver Transmitter), CAN (Controller Area Network), USB and Ethernet interfaces for communication purposes. The inputs and outputs can be either analog, digital, or, in some cases, both.

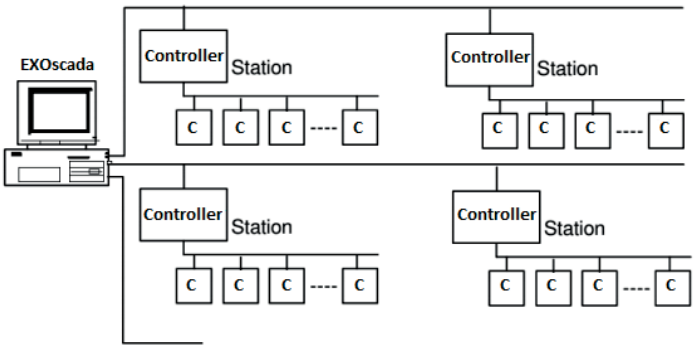

Figure 1.1: EXOscada in connection to a project [7].

A station can consist of solely one station master controller or one station master and up to 254 slave controllers, see Figure 1.1 for an example of controller organisation in a building project. Each station master provides a communication route. The controllers use EXOline protocol to communicate with each other within its own station and with EXOscada over local and wide area network communication lines. Each controller in a station has a unique EXOline address in a project.

A building project is designed using a design tool, EXOdesigner, into applications for controlling sensors and actuators, malfunction alarms, and logging and reporting functions. The applications are compiled in EXOdesigner into code called EXOL, and is an intermediate code format. The EXOL application code is loaded into controllers from EXOdesigner via the EXOline protocol. In the controller, the EXOL code is interpreted, scheduled, and executed by the EXOcol scheduler. Controllers also have a persistent memory structure for storing variables. Some examples of what is stored in this structure is the EXOline address, the current time, and all variables related to EXOL applications. All logging data is also stored here. The operating system that runs on the controllers is a real time operating system called FreeRTOS [8]. The EXOcol interpreter is scheduled by FreeRTOS as a task and executed by FreeRTOS on the controllers. The collection name for all the controller software is EXOreal and consists of FreeRTOS, the EXOcol scheduler, EXOL tasks, and others, such as for instance communication managers, see Figure 1.2 for the structural layer of the EXOreal software.

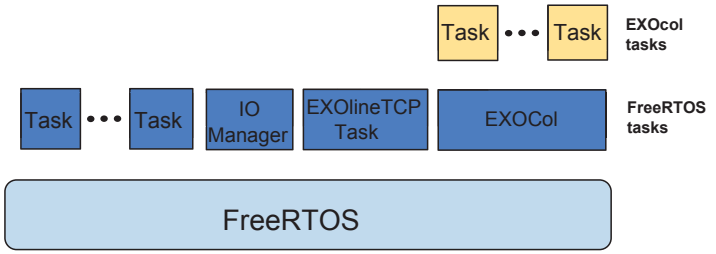

Figure 1.2: EXOreal software

EXOline

EXOline is half duplex protocol. It is of the type Master / Slave and is independent of physical media. It can be incorporated over the protocols Modbus, BACnet and TCP/IP. An EXOline query looks like this:

```
<SoM><EXOline address><OPC><DATA n><CS><EoM>
```

It starts with a start-of-message token (SoM) and ends with an end-of-message (EoM) token. It has a command type (OPC) and has a one byte check-sum (CS). DATA is dependant on the specific command. For example if an integer value in the variable storage is requested, DATA is the variables location. Multiple such commands can be combined into a so called MulCmd. The recipient is specified using the EXOline address. The recipient replies with:

```
<SoA><DATA n><CS><EoM>
```

where SoA is start-of-answer.

EXOscada

The term SCADA refers to Supervisory Control and Data Acquisition and is a supervisory tool. EXOscada is Regin's own developed supervisory software. It is used for monitoring and setting set points for control processes in a user friendly web based graphical interface. Pulling alarm and logging data from controllers on demand or on regular basis is one of most used functionalities. This one generates the most communication data. The data is stored in databases and can be used for viewing current status and for report generation.

All controllers in a project are organized by their communication route meaning the controllers in one station share the same thread in EXOscada. EXOscada is aware of building projects and thus knows all of the stations and their controllers.

Any controller in a station, regardless of being a master or a slave, is accessed through the same communication route and by the controller's unique EXOline address. EXOscada supports multiple communication lines and TCP/IP connections. More specifically, it can communicate with several or all stations in a project simultaneously. EXOscada can establish a connection in two different ways. EXOscada can connect directly to a station in the

traditional case where a station master has a static IP that has been assigned in the project design phase. When a station master has a dynamically assigned IP, it must be the one to contact EXOscada. When a station establishes this connection with the EXOscada server for the first time it does a so called BackConnect. In the BackConnect phase, the station master takes the initiative by contacting the EXOscada server, exchanging some feature commands and finally presenting its id. After that, the station switches to the normal ForwardConnect phase and starts listening for messages from the server. The EXOscada's role in a communication route with a station is the EXOline master role and the station's role is slave.

1.2 Problem definition

With increased number of communication routes in a building project the data flow into EXOscada increases. When the connections to a project, over all routes, is established the amount of communication data leads to higher CPU load and longer data storage queues. An example of a particular problem that was encountered in EXOscada while communicating with a project of the described size was a race condition bug. The amount of communication routes was around 250 and the communication speed with EXOline over TCP/IP was unlimited which caused the rendering of slave (station) answers to overload the CPU as the data flowed in. This discovery emphasizes the need to test the EXOscada's behaviour when connected to a project exceeding several hundred communication routes.

For testing purposes a virtual controller that can to some extent emulate the behaviour of physical controllers is needed. The most basic and necessary functionality is that controllers have to be able to establish connection to the EXOscada server and make use of the EXOline protocol. In this thesis three proposed techniques for controller virtualization that are assumed to be feasible are investigated. The main goal of this thesis is to clarify which solution is the best one for testing purposes. This is based on communication responsiveness and scalability aspects of the solutions when communicating with test program that simulates EXOscada.

1.3 Method

The work method in this thesis is of an exploratory nature. This section provides a general description of the methods used in this thesis work. In order to reach the main goal, a prototype virtual controller for each proposed solution was developed.

Each prototype is based on one of the following techniques:

Emulated solution
Use ARM CPU and platform emulation with QEMU [9] for running the entire virtual hardware stack in multiple instances.

Ported solution
Porting the controller's software for the x86 architecture.

Simulated solution

Develop a software prototype to only simulate the parts of controller's behaviour that EXOscada is interested in.

First some clarifying of terms related to virtualization, simulation and emulation with regards to implementation aspects of the software. Virtualization in this work is the technique of developing software that replicates the behaviour of a physical controller. Emulation uses unmodified software that is actually run on the physical controller. This provides the opportunity to use all the features of a controller and to test even the hardware related code. Simulation, on the other hand, uses a simplified stack of controller software and puts focus on testing specific aspects or features of the software. When considering the scalability aspects of virtual controllers, the simulation focuses on simplifying the operating conditions of the real controller environment for the sake of scalability, whereas the emulation focuses on the behaviour accuracy rather then scalability.

Each solution is to some degree based on existing EXOreal code. The non-essential parts were stripped away while retaining the main functionality. The first controller solution is based on unmodified EXOreal code and the complete controller is emulated in a virtual machine. In the second solution, all hardware dependant code is removed and is replaced by a compatibility layer for the Intel x86 architecture. The third controller solution is based on reused code for EXOline functionality, and reading and writing a flash file holding variables. New code for handling TCP communication is added. The motivation for the choice of these solutions was based on the solution's natural delimitation ant the solutions were appreciated as doable in the time frame for the thesis.

The controller prototype of each solution in connection with EXOscada over TCP/IP was tested. The test data consisting of response time for each controller had to be collected. And for this purpose a test environment was developed. The responsiveness of the controllers were tested in this environment with varying numbers of controllers. The test environment was run on a separate machine in order to not affect the test. More detailed method approach is described in Chapter 3.

1.4 Related work

In this section, the related work performed in the areas of software porting, simulation, and emulation are presented. In our search for academical work and research, we found published work describing the virtualization techniques and the evaluation of these. Although virtualization has been popular and used for decades the amount of academic work on emulating an ARM based device running a Real Time Operating System (RTOS) with its applications seems to be quite sparse. Further, application migration to a different architecture is the area of software porting. There are approaches for different architectures suggesting different methods for porting. These approaches differentiate the porting into processes with contrasted host migration paradigms. Additionally application migration involves challenges, solutions, and benefits for each target architecture.

Magnus Granberg Opsahl, 2013 [10] has performed evaluation of the most prominent open source virtualization technologies and did architecture comparisons in the terms of resource usage. The main purpose was to evaluate the performance based on CPU, memory, and IO benchmark and to present a conclusion for what operations the compared

technologies are best suited for when emulating a RTOS system for ARM devices. In their CPU and memory intensive benchmark results, Quick EMUlator (QEMU)/Kernel-based Virtual Machine (KVM) stood out in terms of performance and that supported the idea of QEMU as plausible tool for full emulation of the ARM controller in many instances.

Mehdi Aichouch, Jean-Christophe Prevotet and Fabienne Nouvel, 2013 [11] evaluated RTOS running in virtualized environment. They chose to measure in detail the internal fine-grained overheads and latencies instead of global results from an application perspective. They claim that this approach is fundamental in analysis of a real time system in real time critical perspectives while sharing the same hardware with another operating system and executing applications with different priorities. In our work we are not concerned with the issues of real time criticality but more of the performance aspects since our goal is to run our solution of the virtual controller in several hundreds of instances. It is valuable for our work to get appointed on the factors that have cause on the performance degradation for QEMU and KVM.

William Weinberg, 2007 [12] has written a guide for migrating applications from legacy RTOS to Linux. The report describes several approaches: virtualization, porting to native application, and creating a compatibility layer[1]. The guideline for each approach is presented and also several decisions for architectural options that need to be considered, are described. Some important considerations are whether to implement RTOS tasks using processes or threads and what synchronization methods are appropriate for that choice, and what types of inter/intra-process communication channels are available. These aspects are considered useful in this thesis for both the process of migrating the EXOreal to x86 platform and also for simulating the required basic functionality of a physical controller.

1.5 Scope

It is a fact that embedded real time operating systems have stringent timing and performance constraints. Virtual controllers developed in this thesis work will not control any actual hardware or regulation processes. Since the virtual controllers are to be executed on top of a general purpose operating system the real time requirements can not be met nor is it considered an issue as it is outside the scope for this thesis work.

In our work we are making use of few different Operating System (OS) for desktop computers of x86 platform. So when a particular solution of a virtual controller is developed and proved working in one OS, it is considered as sufficient and no work will be done on adjusting the controller software for any other operating system of the same platform.

The basic functionality that virtual controllers need to have is the ability to make a network connection to an EXOscada server over TCP/IP and make use of an essential subset of the EXOline protocol. The virtual controller solution that is ported to x86 might actually provide more functionality over the basic ones since they are already implemented in the original code and remain unchanged. The QEMU emulated controller is also assumed to provide additional functionality since the controller code is executed unmodified. For the simulated solution the extent of the functionality implemented will focus entirely on EXOline over TCP/IP only. This is considered to be sufficient for acquiring the response data when testing the solutions. In a case where there is time left this solution will include

[1] Described in more detail in Section 2.2

support for EXOcol tasks, as long as the implementation complexity is acceptable. Including the support for the EXOcol tasks to the simulated controller solution would actually make the comparison more equal since all solutions would execute the same EXOcol tasks.

QEMU supports emulation of quite a few ARM based device models and two of the ones listed in the specification [13] are STM's device models. The virtual controller solution emulated in QEMU is assumed to almost work out of box. It is needed to confirm that QEMU can emulate the right networking interface and that the controller can actually establish a network connection and make use of the EXOline protocol. In the case where a QEMU emulated device model does not satisfactorily emulate the network interface so the virtual controller lacks the networking ability, no additional time will be spent on implementing this or any extra virtual hardware functionality. This is mainly based on the complexity of the physical controller's hardware and to keep focus on exploring the solution and not extending QEMU's device model support.

1.6 Individual contributions

With regard to the practical work, Christian wrote the ported solution and the benchmark program and Dalibor explored the possibilities of using emulation. The simulated solution was written together. Testing the solutions in the benchmark program was done together.

The report was mostly written together but everything related to emulation was written by Dalibor and Christian wrote the majority of the porting content.

1.7 Outline

This section briefly summarizes the layout of the remaining part of the report. Chapter 2 describes the technologies and tools used in the work of developing and testing virtual controllers. Further, the underlying theory, required to understand these technologies and techniques is also included here. The approach chosen in order to achieve the goals set in this work is outlined in Chapter 3. The results achieved in development and testing work are presented in Chapter 4. In Chapter 5 the discussion regarding controller prototypes and test results is presented. The conclusion based on benchmark data and future work proposals are presented Chapter 6.

Chapter 2
Theory and Tools

In this chapter, the theory relevant for the proposed virtual controller solutions is presented.

2.1 Emulation

The idea of virtualization originates from the concept of virtual memory and time sharing. Virtualization is a software technique that allows simultaneous existence of one computer in another [10]. Some reasons for virtualization are, among many, isolation and performance. In the isolation case, it is beneficial to separate machines to run on their own systems and thus not affecting or conflicting with each other or other machines in a case of errors.

In the case of performance, providing an application with exclusive access to system's resources benefits the application with better performance in contrast to using resources shared with other applications [14]. It is possible to consolidate multiple physical machines if they are not fully utilized, possibly with different operating systems, and run them all on a single machine using virtualization. A Virtual Machine (VM) is software that is executed on an hardware abstraction layer and it is usually referred to as a guest. A Virtual Machine Monitor (VMM) is software that provides the hardware abstraction layer to VM and is referred as host, see Figure 2.1 for virtualization structure. Hypervisor is an other term for VMM. Based on the implementation VMMs can be classified to:

- Type 1, runs natively in direct communication with the hardware, resource allocation and scheduling is performed by kernel.
- Type 2, runs inside host operating system (OS), all the resource allocation and scheduling is managed by the host OS.
- Hybrid Virtual Machine (HVM), privileged instructions are interpreted in software and hardware access is manged by special driver for the guest.

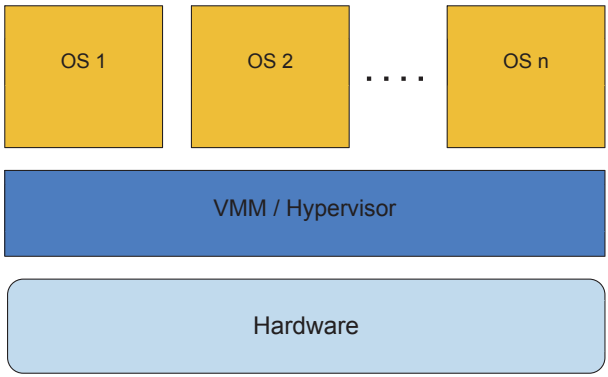

Figure 2.1: Virtualization basic structure. Hardware at the bottom and the abstraction layer providing resources to the operating systems which are unaware of the abstraction.

Hardware virtualization was the first one to be developed. A lot of work has been done on hardware virtualization over the last few decades and several different techniques have been used. The common techniques for the x86 architecture are hardware-assisted, full virtualization, and paravirtualization [10].

In hardware-assisted virtualization, the distinction between the host and guest running mode on the processor is facilitated by the hardware. The support for this type of virtualization is added to x86 through hardware extensions. Intel and AMD provide the hardware extensions for their x86 architecture processors.

On the other hand, full virtualization, as the name suggests, makes it possible for an operating system and its kernel to run on the original hardware using a binary translation technique. Binary translation is a virtualization technique where the guest is allowed to execute its translated instructions directly on the hardware except for the privileged instructions which are caught and emulated by the hypervisor. This way, the guest does not need to be modified in order to be run.

In the paravirtualization technique the virtualized guest needs some modifications in order to be able to execute on the host, see Figure 2.2. The guest operating system is modified by adding hyper-calls which are used for calls to the VMM and thus the guest OS have to be aware of being virtualized. The guest is provided a software interface that supports calls to the hardware. Intel and AMD provide the interface support through hardware extensions.

Other advantage that virtualization offers, besides the isolation of virtual machines are the advantages of using virtualization tools in software development. The virtualization tools like KVM and QEMU provide developers with flexible environment for testing and debugging purposes. Since the environment is more easily created and reconfigured than the real hardware application testing and debugging is possible in the early development

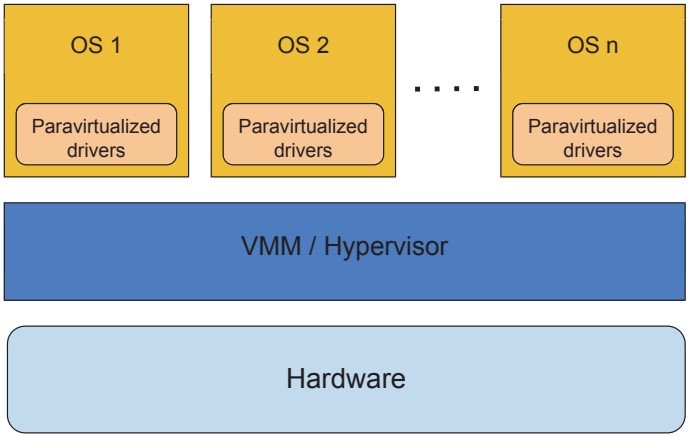

Figure 2.2: The paravirtualization layers with the modified drivers in operating systems.

stages and it is widespread among Linux developers. Additional advantage is that it can eliminate the necessity of real hardware as for example QEMU does. This feature is of particular interest in this project.

There are both proprietary and open source virtualization tools. QEMU is an open source tool and it will be explored and used in this project. Another such tool is kernel-based virtual machine KVM and since it can be used in conjunction with QEMU it will be considered for use in this project. It actually originates form the QEMU project and uses the hardware virtualization extensions on the X86 architecture through the Linux kernel device drivers.

2.1.1 KVM

Kernel-based virtual machine is a open source virtualization solution for Linux for the x86 platform. Essentially it is a kernel module integrated into the Linux kernel. It uses hardware assisted virtualization through hardware extensions. Intel and AMD provide the hardware extensions with the Intel VT-x [15] and AMD-V [16] technology and KVM makes use of these technologies.

Usually a hypervisor is most importantly composed of a scheduler, memory manager, I/O-stack manager and drivers for the hypervisor target architecture. KVM is a kernel module implemented in such way that the Linux kernel has the hypervisor role and KVM is solely handling the guest emulation. Since the Linux kernel already has the scheduler among other kernel modules, the scheduling responsibility is left to the Linux scheduler. The KVM module handles all the communication between the guest and the hardware, see Figure 2.3. The KVM kernel module is exposed to the guest via `dev/KVM` inter-

face. A guest is usually initialized with a user space tool/program for instance QEMU that has KVM support. KVM is responsible for the management of the guest to host context switches. This entails managing processor registers, MMU registers and the hardware associated registers for the emulated PCI hardware.

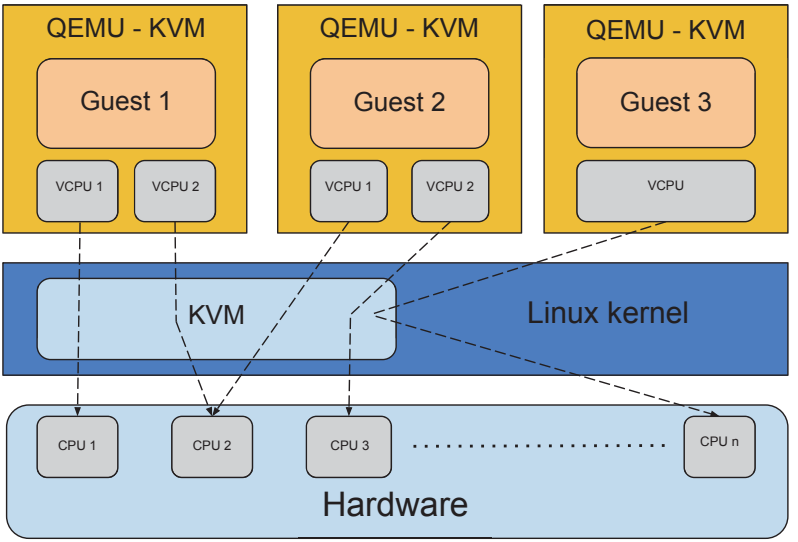

Figure 2.3: The overview of the KVM layer structure.

A guest processor is emulated by executing a thread in the user space and is scheduled with the other Linux processes by the hypervisor, i.e. the Linux kernel. A guest's physical memory is mapped to the host's virtual and is handled by the user-space tool by keeping a shadow page tables. Usually the memory translation is emulated in the software but addition of Intel's and AMD's new technologies for virtualization support, the handling of the memory mapping has moved to the hardware. This allows the guest to manage its own page tables. I/O accesses and the storage is also handled by the user-space tool.

2.1.2 QEMU

QEMU is abbreviation for Quick Emulator and is open source processor emulator that can run unmodified operating systems in a virtual machine [17]. See Figure 2.4 for QEMU illustration. QEMU can be installed on several operating systems such as Linux, Mac OS X and Windows and a few others. The host and target processor architectures can be different or the same.

QEMU consists of several subsystems, CPU emulator, emulated devices, generic devices, machine description for instantiation of emulated devices, debugger and user interface. It can emulate several hardware architectures such as ARM, x86, PowerPC. It can

also emulate hardware devices such as network cards, VGA displays and other peripherals [18].

QEMU uses a portable dynamic translator for the guest CPU instruction translation to the host CPU instructions. Usually the dynamic translation of the guest CPU instructions to the host CPU instruction set is performed as just-in-time compilation at run-time. Blocks of instructions are fetched and translated into Translation Blocks (TB) in host binary. These translation blocks are stored in the cache (Translation Cache) for future use and therefore the instruction decoding is only performed once. This translation process is simplified in QEMU where chunks of offline pre-generated machine code are concatenated instead [18].

The hardware like network, storage, IO interface, PCI that are exposed to the guest are emulated and handled by QEMU except for when KVM is invoked in the conjunction with QEMU. Then QEMU is started with the `-enable-KVM` parameter and the CPU emulation in QEMU is switched off. The physical memory of the guest is emulated by mapping it into its own process address space. QEMU allocates the memory for the guest at start. On the host QEMU is executed in the user space and is scheduled by the operating system.

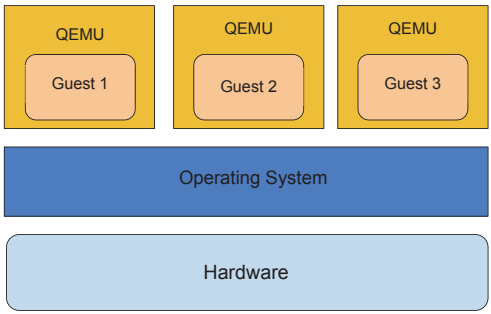

Figure 2.4: The QEMU architecture layer overview.

The guest CPU instructions are executed on the host CPU through the hypervisor without the instruction translation in the case where the guest and host CPU architectures are identical.

Portable dynamic translation

The translation process is started by dividing all the guest CPU instructions into smaller and simpler units called micro operations. These micro operations are then manually coded into C functions and compiled by GCC to a host object file. The micro operation set is much smaller then the guest CPU instruction set. QEMU then uses the `dyngen` tool at run-time to disassemble the guest instructions to micro operations. The dyngen tool maps these micro instructions to the pre-compiled ones in the object file and concatenates

them for the execution on the host. The binary translation can achieve almost the same execution speed as the host machine [18].

QEMU Usage

In a UNIX environment the QEMU is started from command line. Depending on what architecture is to be emulated the command differs but for ARM it is `QEMU-system-arm` followed by command parameters for memory size, disc, flash file, network configuration, and many more. Since the list of command parameters can get long QEMU provides a possibility to read the the parameters from file with `-readconfig` and to save them to file with `-writeconfig` parameters.

QEMU has a list of supported embedded devices that it can emulate. The command `QEMU-system-arm` with parameter `-machine help` reveals the list. The command `-QEMU-system-arm -M device` is used to list the CPU models that are supported for a specific device followed by the parameter `-cpu model`. A example of command for starting emulation is:

```
QEMU-system-arm -M MACHINE -cpu cortex-m3
-singlestep -kernel Controller.elf
```

where MACHINE is the argument that specifies the board to emulate.

QEMU can emulate several models of network cards and these can be connected to an arbitrary number of Virtual Local Area Networks (VLANs). See Figure 2.5 for an example of QEMU's Virtual Local Area Network configuration. The standard way of connecting QEMU to a real network is using host's virtual network device called tap that can be configured as a real Ethernet card. On Linux the device `/dev/net/tun` must be present in the host kernel.

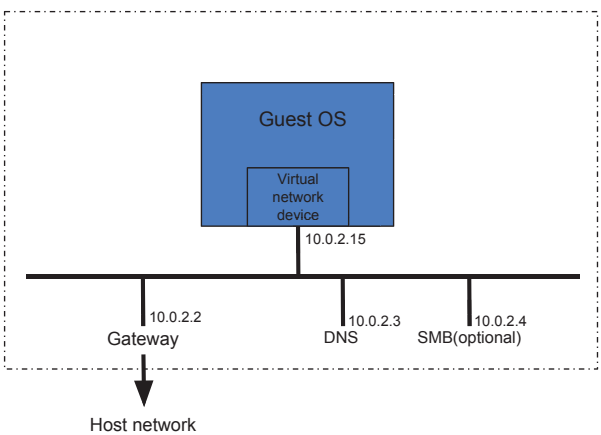

Figure 2.5: The QEMU network [19].

To connect the QEMU network card to the communication for the outside world the tap interface can be connected to a to a virtual bridge on the host. For this network configuration additional arguments had to be supplied

```
sudo QEMU-system-arm -semihosting -M MACHINE
-cpu cortex-m3 -kernel VirtualController.elf
    -net nic,vlan=0,macaddr=52:54:55:55:55:55
    -net bridge,br=virbrN -net tap,ifname=tapN
```

where bridge is the host's virtual network interface and tap is a tap interface. Mac address on the QEMU's VLAN can also be supplied along with the other network parameters.

Instrumenting the QEMU with GBD

QEMU is flexible since it provides the possibility to define new hardware and to include it for emulation. Besides using QEMU for running guest operating system different from the host, it can be used for debugging the software since the virtual machine can be stopped for state inspection, saving, or even reloading. This is very useful for simulating embedded devices while they are still in development phase. QEMU includes a built-in GDB stub that can emulate the functionality of the usual JTAG interface debugging [20]. The GDB stub controls the instruction flow to the guest machine and thus it emulates debugging from the very first guest instruction. The GDB stub routines are executed directly on the host and the communication between a GDB client and the stub is forwarded through the virtual network of the host. When QEMU is started with the '-s' command parameter, it waits for a GDB connection on the localhost network interface. The GDB is started as usual and is then provided with the connection parameters `target remote localhost:1234`. Tracing is enabled when QEMU is run in debug mode. The QEMU emulator generates complete tracer logs of the PC register for tracking the software execution flow.

2.2 Porting

With this solution we wanted to use the existing EXOreal code base that compiles to a complete operating system based on FreeRTOS and instead convert it into an application for Windows or Linux. This means we have to remove or replace any processor-specific code such as, in-line assembly, and hardware interacting code. This also includes a hardware tick interrupt which FreeRTOS needs to be able to support preemptive multitasking. In our project we tried to change the original code as little as possible so as to minimize the work required and also so that the code could easily be incorporated back into the main development branch. One important thing to note is that since the controller software will be running as an application on a non-real time operating system. Therefore it is impossible to guarantee real time performance. This is not a problem since we already restricted our scope in Section 1.5.

Porting is the process of converting and adapting software from one computing environment to a different one. A computing environment can be an operating system, a CPU architecture , or a particular library. In our case we want to move from ARM to x86 and

from FreeRTOS to Windows or Linux. There are a couple of different techniques that can be used. The software can be changed to not rely on features that are specific to the original environment. Another possible technique is to make some kind of compatibility layer that emulates the behaviour of the previous environment [12].

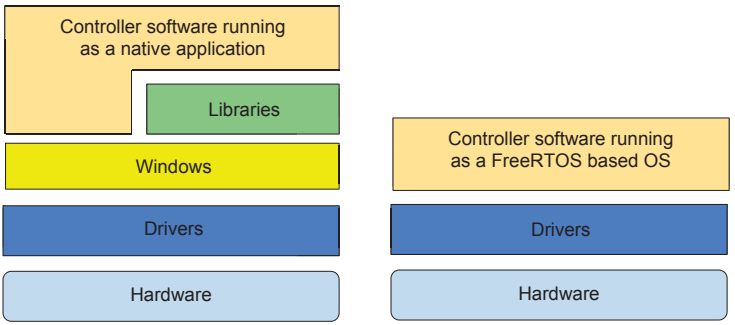

Figure 2.6: The architectural differences between the ported solution and the physical controller.

A famous example of the latter is Wine[1] which is a compatibility layer that allows for running Windows applications on Unix-like operating systems such as Linux. It is also possible to do a hybrid approach, change some parts and use a compatibility layer for some parts. This unfortunately runs the risk of introducing inconsistencies between the two approaches. The third approach mentioned in Weinberg's article [12], virtualization, is talked about in Section 2.1.

Since it was desired to change the original code as little as possible, the compatibility layer approach appear to the best suited. Basically the entire controller software stack would run as an application on Windows or Linux with a layer in-between (see Figure 2.6). To simulate more controllers, more instances of the program are simply run.

32-bit vs. 64-bit

Another important consideration is whether to make the x86 program 32-bit or 64-bit. Since common integer types and especially pointers can have different sizes on different architectures it is an advantage if at least the word size and pointer size remains the same. If the data types size changes it could mean that bugs might be introduced which could be hard to find. Another consideration is that bigger data types obviously also means higher memory usage. An initial concern was that the total memory usage might become a bottleneck. On the other hand, code compiled for 64-bit on x86 is usually slightly faster. One source reports a 5-15% increase in performance for an average program simply from recompiling existing software to take advantage of the 64-bit architecture [21]. This is in part because of the additional general purpose registers that can be taken advantage of. There are also vector instructions that can be taken advantage of for computing intensive

[1] `https://www.winehq.org/`

tasks but that does not apply in this case. The original software on the controllers is 32-bit and for these reasons it made sense to also make our port 32-bit.

2.2.1 FreeRTOS

The FreeRTOS is a open source real time operating system developed in partnership with world's leading chip companies [8]. It is a very small operating system that is designed to be easy to port to new architectures. There is official support for ports to over 30 architectures.

The kernel largely consists of three C-files: `tasks.c, queue.c, and list.c`. Tasks.c contains functions for task creation and management. It handles, amongst others, task creation and the scheduling for FreeRTOS. Queue.c implements message queues that can be used to pass messages between tasks and is also used for synchronization. List.c implements a doubly linked list that is used to implement the queue. Those three files contain no platform dependant code and are thus portable. The parts that need to be ported are the hardware initialization code and other boot related code, drivers, and code that handles context switching and interrupts on an assembler level [22].

FreeRTOS is also very configurable and different parts can be turned on or off as desired in the FreeRTOSConfig.h file. You can turn on or off things like mutexes, timers, and co-routines.

The system tick on FreeRTOS is normally a hardware generated interrupt that happens with some configured frequency. On the controllers the tick period is 1 ms. On each tick FreeRTOS will do a context switch and the highest priority task will be run [23]. This is true as long as interrupts are not disabled, which you would do in a critical section for example.

2.2.2 lwIP and PCAP

Lightweight IP (lwIP) is the networking stack that is used on the controllers. Since the program obviously needs to be able to use Ethernet there was a need to port lwIP as well. This includes both the parts that form an interface with FreeRTOS and the parts that interface with hardware. Fortunately there exists a port of lwIP for FreeRTOS that utilizes Packet CAPture (PCAP).

PCAP is a library that can be used to capture packets and is used by many tools to analyze network traffic and is for example used by the very popular packet analyzer Wireshark. It captures link-layer packets directly from the network adapter [24]. Contrary to what the name suggests, PCAP also has the capability to send raw link-layer packets to the network adapter. This makes PCAP a perfect choice for imitating a network adapter. A program can pretend to be a whole computer (or multiple computers) with its own hardware and Internet address. For Windows there is a fork of PCAP called WinPCAP which has diverged slightly from the original but the two are still mostly compatible.

2.3 Simulation

Simulation is software technology consisting of designing a model of a real or theoretical system, execution of that model on a computer and the analysis of the execution outcome [25]. The main benefit of simulation is that it can provide valuable knowledge about systems or products before any actual time and money investments are done [26]. Besides knowledge about the system's behavior, the simulation can provide detailed insights into the design, processes, and architectures. Simulation is useful in critical areas since it can be used to discover issues with systems before they become problems. It can also be a great support tool in training.

The usual simulation practice is to simulate hardware in some simulation tool where input parameters are provided and output data acquired for the analysis purpose. The simulation in this work differs from the usual one in such way that no such simulation tool is used and the software that simulate controller and EXOscada functionality communicate over the physical Ethernet line. The simulated solution's role in the simulation is to communicate with the SCADA system, meaning the network functionality of the controller is the only thing that has to be simulated. On the other side of the communication channel is the benchmark program that simulates only basic functionality of the EXOscada. It awaits back connection from simulated controller then sends requests and receives replies over the establishes forward connection and collects the response time.

2.3.1 Threads and processes

An important consideration for this solution was whether to use threads or processes to achieve concurrency. Threads have the advantage of being easily portable between Linux and Windows. Pthreads (POSIX threads) is available for both. Threads are also faster to start up but start-up time is not very relevant for this thesis. Processes have the advantage of being more robust [12] and giving each process their own memory which means you won't have to think about synchronization and thread safety. Processes are also very easy to use on Linux with a simple fork call. Processes on Linux utilize copy-on-write (COW) which means that it's still very fast to make a new process while still allowing each process to write to their own memory if they so choose. Communication is arguably more difficult between processes since you cannot simply communicate via shared memory. In this project however, it is not necessary to communicate between different virtual controllers (except, perhaps, using the normal communication protocols over TCP/IP). In the end, we implemented this solution using both processes and threads.

Chapter 3

Approach

All the work was divided and structured into three phases. The first phase was the development of prototypes based on their proposed solutions. The factor that decided when the work on a controller was completed was when the controller could be discovered over the network and supported the EXOline protocol. For this discovery, the network sniffing package tool Wireshark was used. Designing and implementing a tool for measuring the responsiveness of controllers was the second phase. The last phase was the testing itself.

3.1 QEMU emulation

The available source code of QEMU was downloaded from the QEMU homepage and from the git repositories for three similar QEMU fork projects. Each project code had to be configured and compiled separately on a x86 machine running Linux. The idea was to examine and emulate QEMU boards similar to the physical controller board. The procedure was straight forward using the trial and error method. The QEMU project was given the priority and was tested first, because this project involves lot larger group of experts than the fork versions. For each QEMU version that failed, the examination continued with version that had implemented more support for STM32 board device.

The virtual machine was started via the command line with arguments for emulating a STM device with ARM Cortex-M3 CPU. The arguments consisting of the executable file of pre-compiled EXOreal code, parameters for debugging and parameters for emulated interfaces such as a network interface were provided to the virtual machine. A debugging was done the whole test time for each QEMU version in order to confirm the correctness of the controller's code execution.

3.2 Software porting

All the software on the controllers in total add up to about 175 000 lines of C-code and 660 lines of assembly code (not counting in-lined assembly). The absolute vast majority of the C code is well written in a platform independent way. Almost all integer type declarations use fixed bit-length integers[1], as an example, which is good for portability. The code specifically for the drivers is around 14 000 lines of code.

The original controller software was compiled with a collection of make-files, a port of Eclipse specifically designed for ARM C/C++ development called Atollic, and GCC for cross-compiling the source code for the specific ARM architecture. Keeping with the theme of wanting to keep things simple, the only significant change that was made to the build environment was to change the compiler to GCC for 32-bit Intel x86 instead.

The approach taken in process of porting the software follows the model outlined in Weinberg's paper [12]. The process can be divided into eleven steps:

1. Set up a Linux-based cross development environment including cross development tools.
2. Copy RTOS application source tree to development environment.
3. Modify build scripts and IDE configurations to link emulation libraries (if any).
4. Modify/alias pathnames and/or modify source files to reference substitute header files (original RTOS header files can introduce conflicts with native Linux headers).
5. Add `#includes` for Linux header files to your application sources themselves (usually stdio.h, stdlib.h, string.h, unistd.h, and errno.h) or via emulation headers (if any).
6. Attempt to make/build and examine results.
7. FIRST resolve symbolic issues for implemented APIs (e.g., simple naming and type-safe linkage issues).
8. Address unimplemented APIs and data structures. (See below.)
9. Repeat steps 5-8 as needed (a.k.a. "whack-a-mole").
10. Tune performance, as needed.
11. Selectively recode and re-architect to leverage native Linux constructs.

The steps above are taken from Weinberg's paper but modified to not refer to any particular products.

[1] Like `uint8_t`, for instance

3.3 Simulated prototype

When developing the simulated virtual controller solution, the technique of software porting was mainly used. Parts of RTOS applications were moved to Linux applications in order to simulate the RTOS applications to some extent. Out of the eleven steps for software porting proposed by Weinberg, seven were useful and directly applicable, namely the steps five to eleven.

Parts of code that involved the desired network functionality, which are referred to as a module, were identified so that modularity and code reuse is exploited as well for this software solution. The controller solution is built up of modules in our persistence to keep the reused modules as minimally modified as possible. One reason is that extending the functionality in the future would remain simple. All it would take is including additional modules from the EXOreal code base. The already included module would still coexist with the newly added ones. Another reason is the time saving factor by reducing the effort since there be no need to implement all the functionality from the beginning. The EXOreal software modules that are reused in the controller are the modules that constitutes the EXOline functionality, the module for EXOreal data types, and the module for reading and writing a flash file holding the variables[2]. An additional module was developed to handle TCP socket connections, and receiving and sending TCP messages.

After a backward connection has been established, the controller uses the socket module to receive request messages and interprets the message and extracts commands using the EXOline module. It then reads and writes variable values from the memory structure and constructs the answer with the EXOline module. Finally, it sends the data back with the socket module.

After baseline functionality with EXOline was achieved, support for EXOcol tasks was added via the inclusion of the EXOcol module from EXOreal. This was done in order to make the comparison between solutions “fairer” since the other solutions support these tasks. Another reason was because Regin wanted support for alarms and logging which are implemented as EXOcol tasks in EXOreal.

3.4 Testing

What we want to do in this thesis is find out, for each different solution, how many virtual controllers we can have up-and-running without losing communication with them. Initial experiments showed that the response time of simple ICMP echo request (ping) remained relatively constant as the number of virtual controllers where added until the number of controllers hit a critical point. At this point the response time started increasing very rapidly until all the requests completely timed out. At this point the computer obviously could not process requests faster than it received them.

What we want to do in our tests is to measure the responsiveness of the virtual controllers in a simple benchmark. In order to accomplish this, we have designed a benchmark program that is solution agnostic. The program was designed so that it acts as EXOscada server when communicating with stations. The virtual controllers (or even real controllers)

[2]The module handling flash files is taken from the porting solution

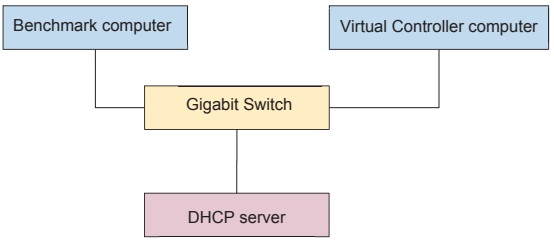

Figure 3.1: Test environment setup

will establish a connection to the benchmark tool. When all the initial connection establishment is complete, the test suite waits for an operator to start the benchmark. The benchmark will issue simple requests, selected by cycling through a small set of four requests, to the connected controllers. The requests were selected to be representative of real world requests, and were in fact captured from an actual communication session between EXOscada and a controller. They all consist of reading variables from the variable storage. Each request consists of one or more commands. Each command is roughly the same length which means that the length of the request is a rough estimate for its complexity.

There are many possible sources of errors that need to be taken into account. Care needs to be taken so that the performance of the computer running the test is not what is measured. Neither can the network's performance be allowed to be a bottleneck. Transient errors such as sudden spikes in activity on either the client side or on the server that is running the test tool can introduce some noise in the sampling process. Spikes in network activity is avoided by running the test in an isolated network. The benchmark program and the virtual controllers were run on separate machined in order to reduce the affect on performance. The test environment can be seen in 3.1.

Based on this reasoning, initially two kinds of benchmarks were designed. The first kind, that is called periodic benchmark, sends a request every second to all controllers and measures the response time. Because the benchmark always waits until it gets a response, the critical point that was observed when testing with ping will not appear. Instead, it was decided to set a limit for responsiveness of one second so that the test would not get "easier" since the request rate would fall below one per second. This benchmark will never saturate the network because the data flow per controller is just the average message length which is 40 B/s.

The second kind, that we call intensive benchmark, has no explicit delay between requests, and will send a new request as soon as it has received the answer for the previous request. This benchmark examines how much data the virtual controllers are able to receive and produce and also how the responsiveness is affected when the controllers are under heavy load.

The response times was measured for each request that was sent to each controller. When the test was done, the results was stored in separate file for each controller. The groups started with only one virtual controller and were gradually increased in size. The limiting factor for the size of groups was the one that occurs first of the following: an operating system limit that could not be circumvented, or when the maximum average

response time in a group of virtual controllers reached one second. Mean values were computed for every controller using a simple script. Then, using the same script the median value were computed for each test group and presented in a box-plot. The data flow for the intensive benchmark was presented as a simple graph.

Chapter 4
Results

This chapter describes the results obtained in development work of the virtual controller prototypes and the response data from testing of the successful controller prototypes.

4.1 Emulated solution

One of the proposed solutions was to investigate possibility of executing the controller's compiled software on emulated controller hardware in QEMU. The emulated controller solution did not work satisfactorily and was considered as failed. The QEMU implemented board devices were unable to execute the controller's code correctly and it was anticipated to be the reason for the failure. The results obtained working on this solution with different QEMU versions are presented further down in this section.

4.1.1 The QEMU project

The minimum requirement, for this controller solution, is that the Cortex-M3 family CPU can be emulated with at least flash memory and an Ethernet interface. The QEMU project supports a collection of board devices from various manufacturers. Using the help command provides a list of supported board devices `qemu-system-arm -M help`. Closer examination using the command

```
qemu-system-arm -M machine-name -cpu help
```

reveals a device list of supported CPU models. The device board Stellaris LM3S6965EVB from Texas Instruments' Stellaris family has support for ARM Cortex-M3. Examination and comparison of the Stellaris board's and the real controller's data sheets revealed that the LM3S6965EVB board provided almost identical features as the real controller. The CPU operation speed differed slightly; it was only 50 MHz which is half of the real controller's.

Initially, the only required hardware to be emulated was the CPU so that it could be verified that the EXOreal code executes without errors. The execution is started with the line:

```
qemu-system-arm -M lm3s6965evb -cpu cortex-m3
-kernel VirtualController.elf
```

The execution of the program code crashed when it was executed for the first time. The crash was caused by a segmentation fault. Debugging followed, and for that purpose the `-s -S -singlestep` arguments in the command line were supplied.

```
qemu-system-arm -M lm3s6965evb -cpu cortex-m3
-s -S -singlestep -kernel VirtualController.elf
```

This way, the QEMU stops the code execution in the virtual machine and awaits for a connection of a GDB debugging session to the GDB-stub in QEMU. Once a GDB session is started with `arm-none-eabi-gdb VirtualController.elf`, it connects to QEMU with `target remote localhost:1234` and then the debugging is done as usual.

On the physical controller the code is stored in flash memory. First section in the code is interrupt handler vector (ISR vector) and is stored at a predetermined address starting at 0x0600 0000. This address is specified in the linking file. The execution starts from the ISR vector in function called Reset_Handler. When QEMU starts to execute the code, it starts at the address 0x0 and the ISR vector should be placed here. This address mismatch caused the segmentation fault and the execution to crash. Adjusting the addresses fixed the crash problem and the code did run without errors. Debugging of the code followed, and it was revealed that the execution could not reach the main function where a break point had been placed. It started in `Reset_Handler`, did some hardware initialisation and then got stuck in the interrupt vector table. It proved that the code did not execute properly on the emulated board. This solution was considered as failed.

4.1.2 Other QEMU project forks

Beside the Qemu project there are other projects that originate in QEMU, but have implemented support for additional boards, namely the support for the STM32 microcontroller. The source code of these projects is available on GitHub. The assumption was made that the emulation of the STM boards from these projects would make the code execution more accurate. The emulation results for these development boards are presented further down in the report.

Beckus

Beckus' project is a copy of QEMU that has been modified in a manner to include an implementation of the STM32 microcontroller [27]. It also includes implementations of an Olimex STM32-P103 and STM32-MAPLE, a couple of Arduino like development boards. Both boards have been emulated and the controller code executed on them. It ran without apparent errors but debugging revealed that the code executed incorrectly. The execution

started in `Reset_Handler` as it should, and it was trapped in a break point in the `main` function. After that, it invoked the functions `systemHardwareInit` , `xTaskCreate` , `vTaskStartScheduler`, and then got stuck in the function `ADC_GetFlagStatus`. This board was considered as insufficient in correctness for the Virtual Controller emulation.

martijnthe

Another fork of QEMU can be found in martijnthe's GitHub repository [28]. This repository includes the implementation of both Beckus' STM32F1xx and additionally support for STM32F2xx has been added. The execution in this fork trapped in `main` break point and was followed by the functions `systemHardwareInit`, `xTaskCreate`, and `vTaskStartScheduler` which is similar to Beckus. The difference was that it got stuck in `static __INLINE void __DSB` which is STM firmware core function. This QEMU fork was considered as insufficient as well.

Wallacollo

This QEMU fork is a version of GNU ARM Eclipse's QEMU fork repository [29]. This repository supports some additional CPU/board definitions such as Kinetis chips. But most importantly it supports STM's STM32F3-Discovery and STM32F4-Discovery boards that can execute the EXOreal code. The execution on STM32F3-Discovery emulated board did not trigger the break point in the `main` function. It started in `Reset_Handler` but got stuck in the function `g_pfnVectors`. When executed on STM32F4-Discovery emulated board, it followed the same pattern as the martijnthe fork. In the end, it got stuck in `static __INLINE void __DSB`. So finally, this QEMU fork could be added to the others as insufficient in execution correctness.

4.1.3 Emulation STM32F1xx board

The EXOreal code is possible to compile to few different target boards. One such target is STM32F107 board. The EXOreal code compiled for this target did execute correctly on the Beckus QEMU implantation of this board. Debugging and stepping through the code showed that the controller executed logically correct and corresponded to the intentions in the code. The biggest drawback was that the board did not supply any network interface and when compiled for this target did not include the code for TCP support as it is specified in code definitions. This solution was insufficient for the communication purposes.

4.2 Ported solution

Porting the EXOreal code to x86 followed the steps outlined in Section 3.2 fairly closely. The development environment on Linux in the beginning consisted of GCC, the original EXOreal make-files, and VIM as the code-editor. Later on, when the work moved on to porting the software to Windows, Atollic was used as the editor. The source code was

fetched from the internal Regin subversion repository and the work was ready to start. In the beginning, no particular libraries were needed to be included in the build script.

The first thing after that was to simply get every non-driver C-file to compile. The work first started on a version for Linux. As EXOreal had always been compiled on Windows before there were some difficulties. Windows file-system is usually not case-sensitive which meant that a lot of header-files became "misspelled" on Linux, i.e. the capitalization was wrong. Another problem was that the Make-files were written for use with Windows-style paths, and with Windows specific shell commands. With some quick butchering of the Make-files and after renaming a lot of inclusions, the C-files finally compiled into object files.

Next step was to get the object files to link into an executable program. It was realized that trying to compile the drivers was not only very difficult, but also quite pointless so those were the first to go. Removing all the FreeRTOS and lwIP source code as well, resulted in over 200 undefined references[1] when trying to link the software. It was suggested, that a closer look should be taken at a demo application that had successfully ported FreeRTOS+lwIP (using WinPCAP to simulate an Ethernet adapter) to Windows. The demo application was quite outdated and some work was required to make it interface correctly with the existing code. The code also had optimizations for the demo application like filtering away all IP traffic that was not intended for a specific, source code defined, IP-address. Since our application needed to be able to change its IP-address and be able to use the DHCP protocol, that optimization had to be removed. Further the lwIP and FreeRTOS code was imported from that project but there were many incompatibilities because that project used FreeRTOS 7.0 while EXOreal used version 8.2. After a while it was realized that it was easier to use the original FreeRTOS 8.2 code and update the FreeRTOS platform dependant code from the demo application to 8.2. The missing functions were implemented as stub functions and finally managed to link the solution into an executable.

Step three consisted of replacing some of the stub functions so that the virtual controller could actually do something. And therefore, the work included implementation of a virtual flash memory and methods that could read and write a "flash-file" that had been taken from a real controller's flash memory. Some preliminary work for future virtual input and output capabilities were completed. The application actually worked surprisingly well almost from the start. It still crashed a lot, though, and it was much slower than a real controller. A lot of time was spent polishing this solution to improve stability and performance.

In the end, the ported solution turned out to be a very faithful imitation of a real controller. There are a couple of functions that have not been implemented such as the ability to change the time, and all the digital and analog inputs and outputs are always zero. The tick frequency is also around 5% slower because it is currently implemented using a thread that continuously sleeps for (around) a millisecond, sets an event, sleeps, and repeats everything again. The application very rarely crashes, whether under load or not.

4.2.1 Porting difficulties

One of the first non-trivial bug was a pretty interesting one. It turned out that a segmentation fault was generated when the program was trying to read a variable from the controller's variable storage. It was very confusing because it was known that the variable was

[1] missing functions and variables

there and the variable could be accessed manually in the debugger. The bug only appeared when the optimization flag to GCC was not `-Os`[2]. A pointer to a stack variable suddenly became `null` between function calls. It turned out to be caused by an optimization compiler directive for a particular function. When this compiler directive was removed, the bug magically disappeared. A couple of looks at the generated assembly code did not result in finding a reason for the bug.

Early testing with the ported application revealed that most packets were not recognized at all by the EXOline client on the virtual controller. On closer inspection, it was discovered that the packets were discarded because the checksum appeared to be incorrect. It turns out that the checksum calculation is often offloaded to hardware on modern network adapters [30]. Disabling checksum validation in lwIP solved this problem.

Another annoying problem that was discovered was a deadlock that made debugging difficult. It turned out that the debug messages themselves were the culprit. When multiple threads wanted to write to the console, the application got stuck. Changing the way debug messages worked solved this problem.

4.3 Simulated solution

This controller solution is built up of modules from the EXOreal code that contribute to the needed network functionality. The EXOreal software modules that are reused in the controller are EXOline and EXOcol. The EXOline module constitutes the network functionality and the EXOcol module contributes with data types and the functionality for reading from and writing to a memory structure holding the variables. An additional module, called the main module, was developed to handle TCP socket connections, and for receiving and sending TCP messages. Figure 4.1 shows the module overview.

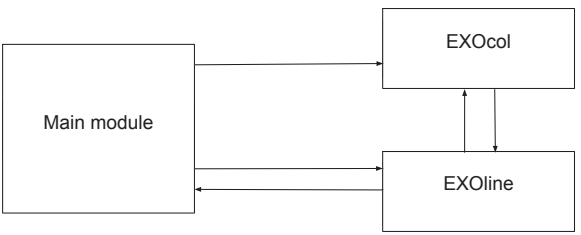

Figure 4.1: The design overview.

The original code in the EXOcol and EXOline modules had to be modified. The code lines that referred to non existing modules had to be excluded. Large chunks got

[2]Optimizations based on code size

surrounded with the `#ifndef MULTICONTROLLER` and `#endif` compiler directives, and single lines were commented out.

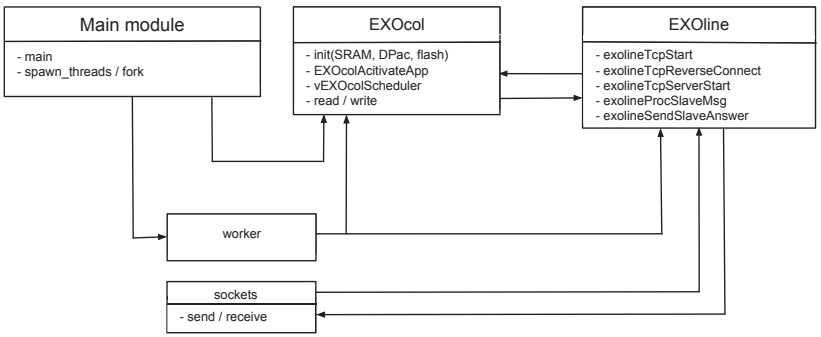

Figure 4.2: High level design of the simulated controller.

The interaction between the modules is outlined in Figure 4.2 and an explanation follows further down this section. The entry point of the controller is the main method in the the main module. Initiation functions from the EXOcol module are also invoked from here. When compiled for Linux, a process for each controller is created in main. The worker starts the EXOcol scheduler for each process in a separate thread, and invokes the `exolineTcpStart` function in the EXOline module, which uses sockets for establishing a TCP connection with the EXOscada server. The number of virtual controllers and the EXOscada server address is defined in the config header file. The TCP port to the server is a predetermined number, used as standard port, and is hard coded in the source code. EXOline implementation in this solution uses sockets to send and receive all TCP messages. An EXOline message is processed in `exolineProcSlaveMsg` and forwarded to read/write functions in the EXOcol module. Read/write functions access the memory structure for variables, and also return data to the `exolineSendSlaveAnswer` function in the EXOline module, which prepares EXOline messages and sends them back via sockets.

In this solution, the controller has to initiate the communication with the server since all the virtual controllers will have the same IP address. The controller starts with the BackConnect procedure, invoking the `exolineTcpReverseConnect` function. See Figure 4.3 for the reverse connection scenario. The controller sends requests for capabilities and set-feature-reverse initiative to the EXOscada server, and in turn receives the answer for each. When the `Ok!` answer is received, the controller gets request for its capabilities and station ID which it replies on. With the ID reply the BackConnect phase is finished and the controller switches to the ForwardConnect phase. In this phase the controller is waiting for requests from the EXOscada server.

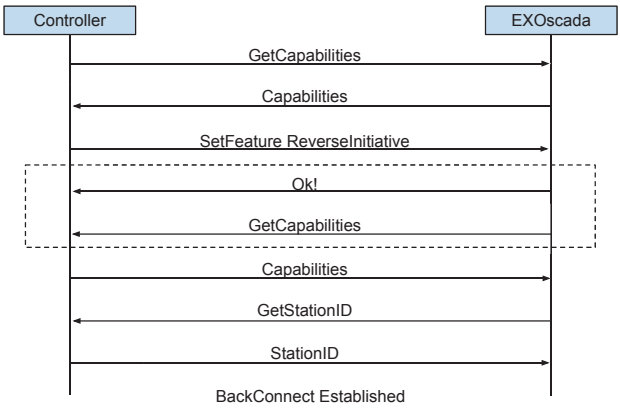

Figure 4.3: BackConnect scenario. The messages in the dashed rectangle are usually sent together.

4.4 Benchmark tool

The benchmark tool is developed for two types of testing, periodic and intensive. The periodic benchmark sends a request every second to all controllers and measures the response time while the intensive sends requests without delays. The tool behaves as the EXOscada server; after start, the tool waits for the controllers to establish the BackConnect-connection and to switch to ForwardConnect phase. Then, the test can be started by entering a start command in the tool. When the test is finished, the tool writes the results to files and waits for the stop command to exit the test. Initially, the tool was designed to start the test for all controllers at the same time. The analysis of the very first testing results revealed a very high variance for the simulated solution when there was a high number of virtual controllers, 6000 controllers. This led us to examine the individual response times for each controller. The result can be seen in Figure 4.7 which shows response times for each controller in the tests that are close to the maximum number of controllers. This extreme stratification was unexpected. The response results from the equivalent test for the ported solution is more uniformly distributed, as it can bee seen in Figure 4.4. The comparison of the two test results motivated the modification of the benchmark program so that it waits a short, random time for each connected controller before starting the benchmark. The result of the modification was an extreme improvement in median response time and the stratification was no longer present, as can be seen in Figure 4.8.

4.5 Test results

The goal in this thesis was to determine the best virtual controller solution based on the same response data and scalability factor. The solutions differed, not only in the implementation structure, but also in the number of controllers that could be run simultaneously

Solution \ Message type	1 (length 11)	2 (length 17)	3 (length 46)	4 (length 86)
Ported (203 controllers)	426.427 ms	410.070 ms	404.776 ms	424.495 ms
Simulated (6000 controllers)	20.7682 ms	16.4916 ms	17.4357 ms	16.8450 ms

Table 4.1: The average response time for each message type for each successful solution.

on same computer.

The reason for choosing different message types in the first place was of course to see if it affected the results in a significant way. In Table 4.1 it can bee observed that there is at most a very small difference. A one-way ANOVA (ANalysis Of VAriance) reveals that there is no significant difference for the ported solution ($p = 0.63$), but a very significant difference for the simulated solution($p < 10^{-16}$). This result seems to indicate that the time spent processing the requests contributes an insignificant portion to the response time in the ported solution. On the other hand, processing the requests in the simulated solution apparently takes a significant amount of time. What the shortest command does, is to fetch the IP-address as a string from the variable storage. The way strings are stored is slightly more convoluted than the way integers and floating point values are because there is an additional layer of indirection. The string also has to copied to a intermediate buffer. It is believed that this explains why that particular command is slower. The ANOVA tables can be found in Chapter A.1 in the appendix.

First some clarification of the boxplots used in Figures 4.6, 4.9, 4.10 and 4.12. The line connects the median values of the response time for the tested controller groups. The box bottoms represent the first quartiles (Q_1) and the box tops represent third quartiles (Q_3). The whiskers, upper and lower, represent the largest and smallest non-outliers, respectively. The lower whisker is calculated by `median - 1,5 * IQR` and the upper whisker is calculated by `median + 1,5 * IQR`, where IQR is the interquartile range calculated by Q_3 `-` Q_1. Outliers are represented with small circles and the straight horizontal line represents the response time limit of 1 second.

The graphs, Figure 4.9 and Figure 4.10, in the following sections show the response time performance of the two successful solutions from the periodic benchmark. In each test group, consisting of a certain number of virtual controllers, the mean response time for each controller was calculated. What is shown in the graph are boxplots of the median response times for each such mean response time with a line between each median. In order to not clutter the graph unnecessarily, boxplots for some test groups are omitted.

Figures 4.4, 4.7 and 4.8 are scatter plots from the periodic benchmark of the median response times for each, for every individual controller when running 185 ported and 6000 simulated controllers. Results from the intensive benchmark tool representing the response time for ported and simulated controllers are presented in Figure 4.6 and Figure 4.12. The data flow for ported and simulated solution captured by the intensive benchmark tool is presented in Figure 4.11 and Figure 4.5.

4.5.1 Emulated solution

While it was not feasible to run the benchmark program with the emulated solution, it it was found as highly interesting to do an estimation of the kind of performance one could expect. In order to do this estimation, a CPU benchmark, CoreMark[3], was run inside QEMU. The QEMU board ARM Versatile/PB was used for the test in Linux in QEMU. This provided an "emulated CPU speed" of a single core when compared with the performance of a real controller running the same benchmark. From talking to Regin's experts, it was learned that the load on the controllers can be anywhere between 10 and 90%, typically. This gives an estimation of the lower and upper bound on the number of controllers one would be able to run inside QEMU. The number of controllers, n, becomes bounded by the formula:

$$\frac{CQ}{R \cdot 0.90} \leq n \leq \frac{CQ}{R \cdot 0.10}$$

where $C = 6$, is the number of cores on our machine's CPU, Q is the CPU benchmark score in QEMU and $R = 398$ [31] is the score of a real STM32F2xx board. The benchmark was run twice with the results 2363 and 2386, resulting in an average $Q = 2375$. This gives a bound on n of

$$39 \leq n \leq 358$$

4.5.2 Ported solution

The ported solution was run on machine with the Windows 10 operating system. This controller solution could run in approximately 203 simultaneous instances. In this case, it was the operating system on the side of the running virtual controllers that set the limit. For some reason, it was not possible to run more than this number. This solution of virtual controller did not quite reach the limit for a mean response time of 1 second, except for few outliers, for both periodic and intensive benchmark, as can be observed in Figure 4.9 and Figure 4.6. The spread of the response times increases with the number of controllers. The widest distribution range is approximately 500 milliseconds, excluding the outliers, otherwise it is approximately 800 milliseconds. The number of outliers, in this solution, increases with the number of controllers as well. Further, the intensive benchmark tool captured the data flow sent to and received from controllers while performing the test. It is clear form the Figure 4.5 that the amount of data increases with increased numbers of controllers, hitting a peak at around 100 controllers and then to starting to decrease. It is obvious when compared with the equivalent results from the simulated solution (Figure 4.11) that the limiting factor for the data flow is not because the network got overloaded.

4.5.3 Simulated solution

The simulated solution could be run in approximately 6200 instances on Linux machine while communicating with the benchmark tool on another machine running Linux. In this case, it was Linux that set the limit for the number of controllers. Simply, it hit the limit for how many processes it could create for virtual controllers on one side and the number of processes for the connecting controllers on the other side of the connection.

[3] `http://www.eembc.org/coremark/about.php`

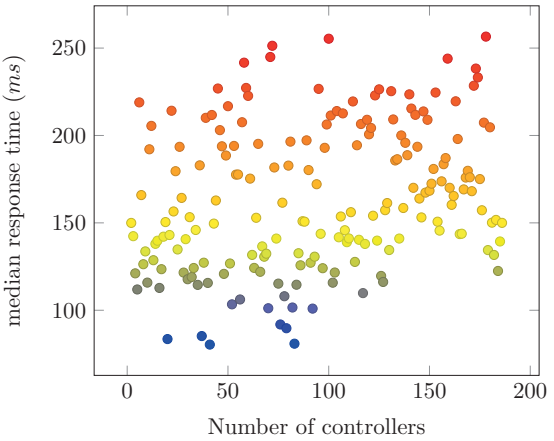

Figure 4.4: A scatter plot of the median response times for each individual controller in the ported solution for 185 virtual controllers.

Overall, the median response time for all controller groups is in the span of few decimal milliseconds up to under hundred milliseconds when running in several thousands instances simultaneously. So, this solution of virtual controller was far from reaching the limit for mean response time of 1 second, as it can be observed in Figure 4.10 and Figure 4.12. It is obvious that with increasing number of controllers, the distribution of response time increases as well but does not exceed 100 milliseconds even with the outliers included. The growth of response time when looking at the graph appears to be linear[4], but further analysis would be needed to confirm this hypothesis.

The response times for the simulated solution obtained in the intensive benchmark, can be seen in Figure 4.12. It can be observed that response times are similar to the periodic benchmark. The graph shows a truly beautiful curve that is almost definitely linear. The analysis of the data flow was also done, as can be seen in Figure 4.11. It looks like it hits a relatively constant maximum between around 1000 and 4000 controllers.

[4]note that the y-axis is logarithmic

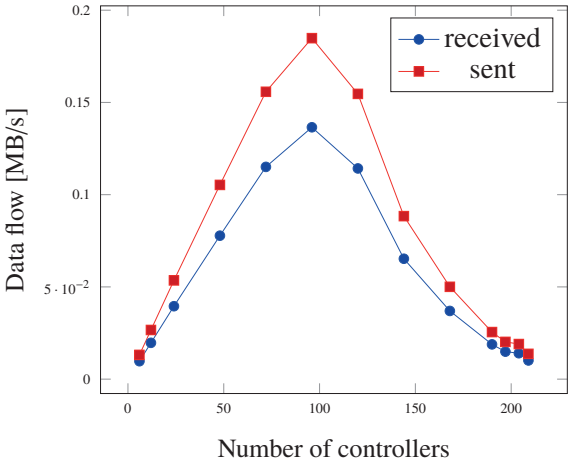

Figure 4.5: The total data flow for the ported solution as a function of the number of virtual controllers in the intensive benchmark.

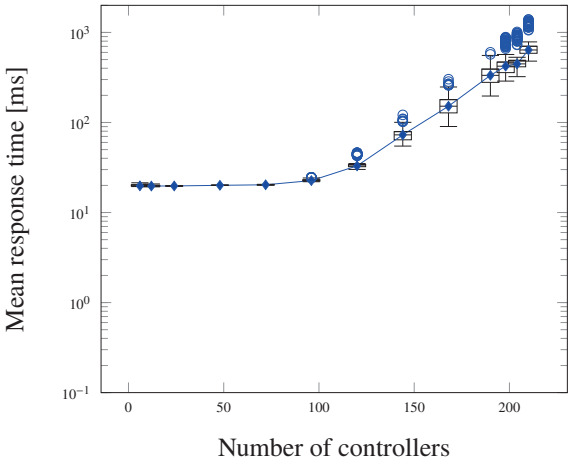

Figure 4.6: A box-plot with the mean response times for the ported solution for selected values of controller instances with the intensive benchmark.

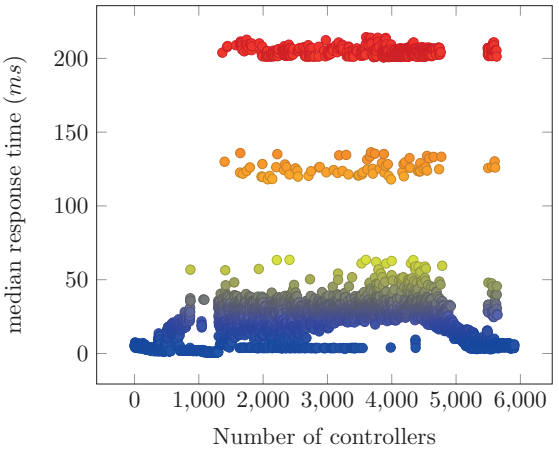

Figure 4.7: A scatter plot of the median response times for each individual controller in the simulated solution for 6000 virtual controllers.

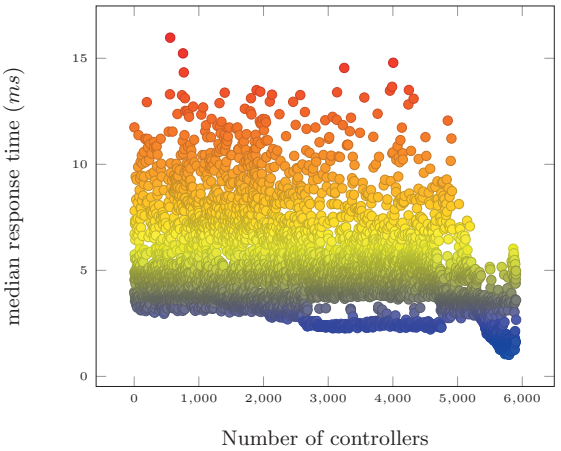

Figure 4.8: A scatter plot of the median response times for each individual controller in the simulated solution with the benchmark program modification described in Section 5.3, for 6000 virtual controllers.

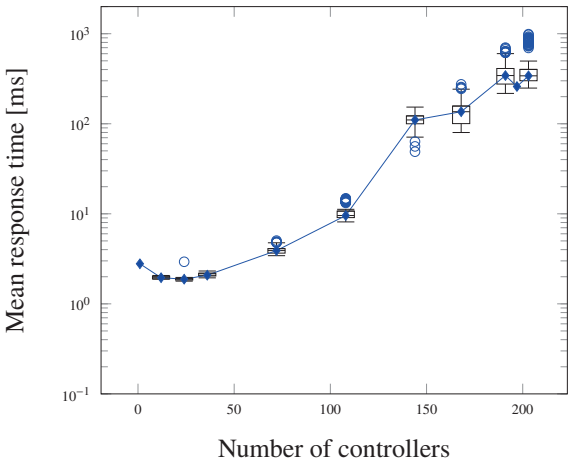

Figure 4.9: A box-plot with the mean response times for the ported solution for selected values of controller instances in the periodic benchmark.

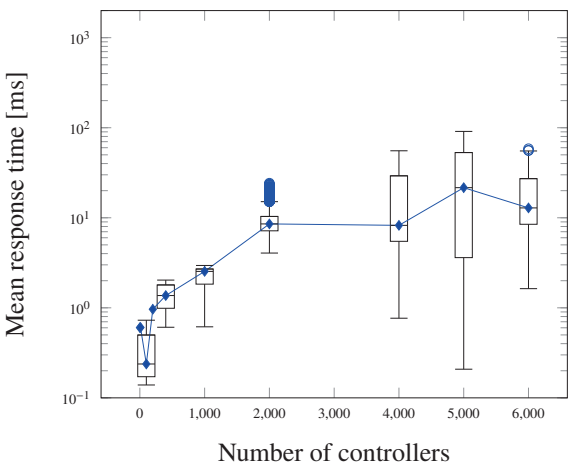

Figure 4.10: A box-plot with the mean response times for the simulated solution for selected values of controller instances in the periodic benchmark.

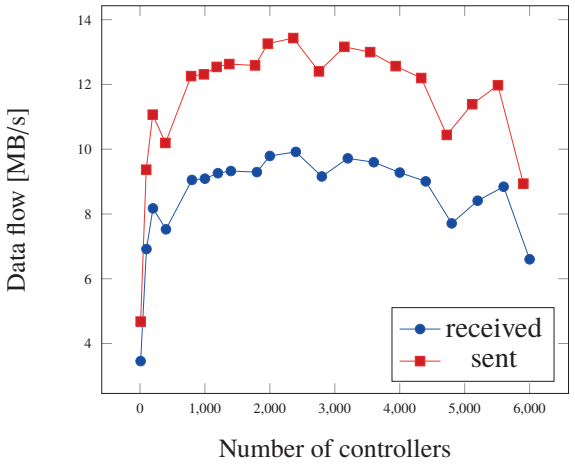

Figure 4.11: The total data flow for the simulated solution as a function of the number of virtual controllers in the intensive benchmark.

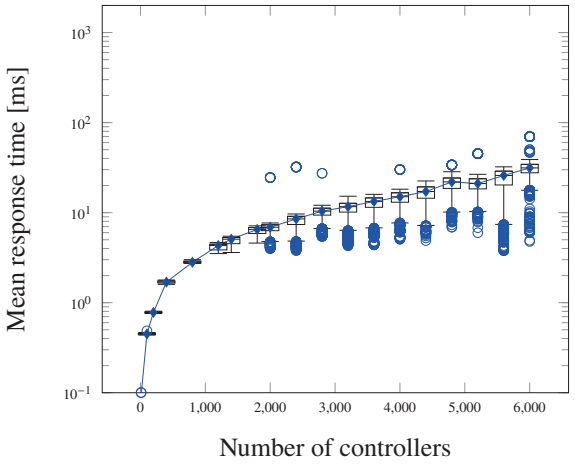

Figure 4.12: A box-plot with the mean response times for the simulated solution for selected values of controller instances in the intensive benchmark.

Chapter 5
Discussion

In this chapter a discussion is given on failed controller solution and the differences of the succeeded ones. It has been anticipated from the start that the simulated solution would have the best performance since it is the solution that does the least. It was unexpected, that there would be such a big difference compared to the ported solution. It is surprising considering that both solutions include an implementation for the relatively performance heavy EXOcol tasks and EXOline functionality. Further down this section, we discuss in detail the reasons for this performance disparity.

5.1 Emulated solution

For practical reasons, and because of the ease of use of an emulator, the emulated solution was considered the easiest one of the proposed. The EXOreal code was anticipated to execute in the emulator without the need for any modifications. Even a few modifications would be acceptable since it would not have much effect on the complexity of this solution. That was one of the reasons to state the scope of this solution to not implement or modify the emulator's device support. The other reason was that implementing the full functionality of the physical controller's hardware for QEMU, would actually move the focus from the goal set in this project to a completely different one, and thus exceed the time frame assigned for it. So, when debugging through the code, while emulating in all the presented QEMU versions, revealed that the effort of solving problems exceeds the intended, and the solution was considered as insufficient.

The emulation in some QEMU forks resulted in the execution getting stuck in the interrupt table, and it did not even reach the break point of the main function. One possible explanation could be that the emulated board triggered an interrupt that was not defined in the interrupt table of the EXOreal code. On other QEMU forks that had better support for STM32, the execution on the emulated device started from ResetHandler, as it is supposed to. It then proceeded through the main function, hardware initialization, task creation and

task scheduler starting functions, to finally get stuck in a hardware CPU core function. This could also point to differences between the supported hardware in the QEMU implementation and the real hardware.

One might think that compiling the EXOreal code for the target without TCP support was not so meaningful in achieving the goal in this work. More precisely, when the EXOreal code was compiled for STM32F107 target board in the Beckus QEMU version, which lacks a network hardware interface, and the TCP code was stripped out by the compiler. The reason for doing that, is that it could be useful for determining if the insufficient network implementation in QEMU was the source of the problems.

The result of developing this controller solution is unsatisfactory and because of the inability to perform the test, it lacks response time data. This leaves few questions unanswered. One is, what performance would the solution have in test. The other is, how would it preform in comparison with other solutions, what would be the maximum number of controllers instances that could be run simultaneously.

Performance estimation

We did an estimate in Section 4.5.1 that the number of controllers we could run in QEMU, n, was between 39 and 358. Since the EXOcol tasks that run on the virtual controllers during the benchmark does not do very much, the higher number is far likelier than the lower one. It should be noted that this is just a very rough estimate. The system board emulated is not the same as the one that real controllers use. The estimate assumes there are no additional costs with running multiple instances of QEMU (i.e. a linear increase), CoreMark is run on top of the relatively performance demanding operating system Linux, the numbers for the STM board is taken from STM who have presumably been very careful when choosing the compiler and optimizations, and CoreMark is not perfectly representative of EXOreal.

Still, it is possible that this solution could have been faster than the ported one which is very interesting. The biggest reason to believe that it would not, though, is the estimation's validity depends on the assumption that the increase in performance requirements is linear when more instances of QEMU are run. Processes are fairly resource demanding and the increasing need for process scheduling and task switching are not negligible.

5.2 Ported solution

Using the PCAP port of lwIP obviously has a performance cost compared to a native TCP/IP stack. However, disconnecting the EXOreal code from the lwIP code would be a significant undertaking, and using normal sockets would limit the controller functionality significantly. There would, for instance, be no way to use DHCP without simulating a network adapter in some way. WinPCAP has relatively good performance and it can easily saturate an Ethernet link without maxing out the processor [24]. Therefore, we felt that using PCAP in this solution was an acceptable trade-off between simplicity, performance, and functionality.

In Figure 4.4, the responses here are more uniformly distributed and this is what we expected, in contrast to the results for the simulated solution in Figure 4.7. This was the

case even before the benchmark program was modified as described in Section 4.4. This is almost definitely because the number of processes were much lower.

While the ported solution is undeniably slower than the simulated solution, and may be unsuitable for the purpose of overloading EXOscada, it is still very useful in other circumstances. A common request from developers for Regin's controllers is that they would like to have a "simulated" controller, running as an application so that they can test their control applications. It is also useful for debugging and trying out new features instead of using physical controllers. It is also useful if more realistic behaviour is required for testing EXOscada. If we have a complete project already designed, with routing between controllers and other features that the simulated solution does not have, then this solution is still relevant. If more virtual controllers are required, more or more powerful computers can be used. With ten consumer grade computers or one powerful server, 2000 virtual controllers can be run.

5.3 Simulated solution

The extreme stratification revealed in the very first test results was not at all what we expected, see Figure 4.7. We determined that the most probable cause was that the network got overloaded when the test started and 6000 TCP packets got sent, more or less, at the same time. We modified the benchmark program to wait a short, random time for each connected controller before starting the benchmark. The result was an extreme improvement in median response time and the previous stratification was not present in the new results, which can be observed in Figure 4.8. But, another phenomenon can be seen here, where the last thousand or so virtual controllers have markedly lower median response times. We believe that this is a result of some scheduling inequality. If one wanted to explore further, it would be interesting to try different schedulers and see if it can significantly affect the outcome.

The results from the intensive benchmark was surprisingly impressive. Even with the very small TCP-packets, the data flow still exceeded 100 Mb/s as can be seen in Figure 4.11. In Figure 4.12, we can see that the responsiveness does not appear to be impacted much by the more intensive request rate when compared with periodic benchmark results in Figure 4.10. This seems to indicate that something other than the network is the bottleneck for the simulated solution. But on the other hand, in the data flow graph in Figure 4.11 we can see that the data flow becomes more or less constant after around a thousand controllers which seems to indicate that the network can indeed be a bottleneck. It is also interesting that for some reason the variance, in intensive benchmark, seemed to decrease a huge amount when compared to the periodic benchmark. We have so far not been able to determine the reason for this.

5.4 Benchmark tool

During testing, the benchmark tool did not get overloaded to the same degree as EXOscada could be when getting log data. This was not the intention, either, but rather collecting the response data for controller evaluation. Another reason was that communicating larger

amount of data could presumably congest the network. If that were the case, the response data would not reflect the performance of the controllers, but rather the responsiveness of the network. It is reasonable to presume that distribution of virtual controllers on multiple nodes (machines) would circumvent the network congestion problem on the controller side of the network. For those reasons the EXOline requests were selected to not ask the controllers for large response data, but only variable values and simple strings. The assumption was made that the message types did not have any significant affect on the response times, but as it can be see in Section 4.5, it did have some effect.

The periodic benchmark is assumed to not overload the network since there is an explicit delay of 1 second between sending new request. As described in Section 3.4 the data flow is almost negligible. On the other hand, the intensive benchmark could cause congestion on the network to some degree. But, we still believe, based on the comparison of the response data of the two benchmarks, that the limitation lies on the virtual controller's side of the communication channel and not the network.

Chapter 6

Conclusion

In this thesis, the performance of three different solutions for virtual controllers have been examined. In order to accomplish this, a simulated virtual controller was implemented specifically for this thesis. The existing controller software was also ported to be able run as a native application. Unfortunately the emulation solution did not result in state to work well enough to be tested in the benchmark program. But an estimation was done of the number of controllers it could have achieved based on Coremark performance estimation data. Of the tested solutions, the simulated solution has undoubtedly the best performance. It can easily run 6000 virtual controllers while still being highly responsive. The ported solution, on the other hand, can only just run 200 virtual controllers. However, while the simulated solution is clearly faster, the ported solution is a more faithful imitation of the original controllers which is useful in other circumstances. One such circumstance is for developing new features for actual controllers. Another one is for customers to be able to initially test their software in a virtual environment. It is also possible to use a mixture of the two successful solutions if different characteristics are desired for some subsets of the controllers. For example, a hundred virtual controllers could be run on the ported solution and a thousand could be run on the simulated solution.

The performed performance estimation shows that it is possible that the emulated solution could have been better than the ported solution. It is a shame that this estimation could not be verified in a test. If successful, the emulated solution would clearly have been the most faithful imitation of a virtual controller in terms of capabilities.

For the purposes of trying to overload EXOscada with a high number of simultaneous network connections, at the present time, the simulated solution is clearly the best.

6.1 Future work

Emulated solution

An obvious improvement of the emulated solution is possible by forking QEMU and implementing either Beckus' or marijnthe's QEMU fork project. The debugging indicated that these two could execute hardware independent code and got stuck in hardware specific functions. Work would be required on examining and implementing the needed support for the actual hardware of the STM32F2xx board. It is suitable as a future development project of the emulated controller solution.

An other possible improvement is to modify the EXOreal code; to port it to a new ARM board that is officially supported by QEMU. The EXOreal code has been ported to new boards before, and the expertise required should be available at Regin.

Ported solution

Some improvements can be made on all IO ports functionality so that it would, to some degree, correspond to the functionality of physical controller. For instance a graphical board with switches and sliders for value visualization and manipulation. This would improve the suitability of this type of virtual controller for the contribution in development work of future functionality for the physical controller.

Improving the performance of the ported solution is definitely a valuable avenue of improvement. Not much time has been spent on optimizing the performance so there is definitely room for improvement.

Simulated solution

One possible improvement for the simulated controller is to develop a graphical user interface (GUI). The purpose of the GUI, in first place, is not to improve any functionality of the controller, but rather to make handling easier. Mainly, it is needed for configuration management of the controllers, like specifying IP address and TCP port of the EXOscada server, assigning EXOline id's to controllers, and the number of controllers to instantiate.

Test environment

While testing, the benchmark tool collected the response data for controller responsiveness evaluation. In order to reduce the risk of network congestion, the test environment could be enhanced. The benchmark could be distributed between multiple nodes, each with a dedicated gigabit (or better) connection, for example.

Bibliography

[1] Adam Hirsch Shanti Pless Paul Torcellini Matthew Leach, Chad Lobato. Strategies for 50% energy savings in large office buildings. Technical report, 2010.

[2] SCADA. Supervisory control and data acquisition. https://en.wikipedia.org/wiki/SCADA, last accessed, 2016.

[3] Regin. Exoscada. http://www.regincontrols.com/Root/Documentations/56_43981/EXOscada_prsh_sv.pdf, last accessed, 2016-10-11.

[4] STM. http://www.st.com/content/st_com/en.html, last accessed, 2016-10-11.

[5] STM32F207ZE. http://www.st.com/content/st_com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32f2-series/stm32f2x7/stm32f207ze.html, last accessed, 2016-10-11.

[6] STM32F207ZE Datasheet. http://www.st.com/content/ccc/resource/technical/document/datasheet/bc/21/42/43/b0/f3/4d/d3/CD00237391.pdf/files/CD00237391.pdf/jcr:content/translations/en.CD00237391.pdf, last accessed, 2016-10-11.

[7] Regin Exomatic A. Exo system, scada driver design guide, 2004.

[8] FreeRTOS. http://www.freertos.org/, last accessed, 2016.

[9] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[10] Jan Magnus Granberg Opsahl. Open-source virtualization functionality and performance of qemu/kvm, xen, libvirt and virtualbox, Spring 2013.

[11] Mehdi Aichouch, Jean-Christophe Prevotet, and Fabienne Nouvel. Evaluation of an RTOS on top of a hosted virtual machine system. In *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pages 290–297, Cagliari, Italy, October 2013.

[12] William Weinberg. Moving legacy applications to linux: Rtos migration revisited. Technical report, 2008.

[13] Qemu registered trademark of Fabrice Bellard. Arm-system-emulator. `https://qemu.weilnetz.de/qemu-doc.html#ARM-System-emulator`, last accessed, 2016-06-10.

[14] Robert Rose. Survey of system virtualization techniques. Technical report, 2004.

[15] Intel. `http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html`, last accessed, 2016-10-11.

[16] AMD. `http://searchservervirtualization.techtarget.com/definition/AMD-V`, last accessed, 2016-10-11.

[17] Qemu registered trademark of Fabrice Bellard. Qemu, main page. `http://wiki.qemu.org/Main_Page`, last accessed, 2016-06-10.

[18] Fabrice Bellard. Qemu, a fast and portable dynamic translator. `http://static.usenix.org/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf`, last accessed, 2016-06-10.

[19] `http://wiki.qemu.org/Documentation/Networking`, last accessed, 2016-10-11.

[20] Bojan Mihajlović, Željko Žilić, and Warren J. Gross. Dynamically instrumenting the qemu emulator for linux process trace generation with the gdb debugger. *ACM Trans. Embed. Comput. Syst.*, 13(5s):167:1–167:18, December 2014.

[21] Andrey Karpov. Optimization of 64-bit programs. `http://www.viva64.com/en/a/0030/`, 2008. last accessed, 2016-10-11.

[22] Heradon Douglas. *Thin hypervisor-based security architectures for embedded platforms*. PhD thesis, Royal Institute of Technology, 2010.

[23] Christopher Svec. Chapter 3: Freertos. In *The Architecture of Open Source Applications (Volume 2)*. `http://www.aosabook.org/en/freertos.html`, 2012.

[24] Fulvio Risso and Loris Degioanni. An architecture for high performance network analysis. In *Computers and Communications, 2001. Proceedings. Sixth IEEE Symposium on*, pages 686–693. IEEE, 2001.

[25] What is simulation. https://www.cise.ufl.edu/~fishwick/introsim/node1.html, last accessed, 2016-10-11.

[26] Alan M. Christie. Simulation: An enabling technology in software engineering. http://www.sei.cmu.edu/library/assets/simulation-enabling-technology-sw-engineering.pdf, last accessed, 2016-10-11.

[27] Beckus. https://github.com/beckus/qemu_stm32, last accessed, 2016-10-11.

[28] Martijnthe. https://github.com/martijnthe/qemu_stm32, last accessed, 2016-10-11.

[29] Wallacoloo. https://github.com/Wallacoloo/qemu-kinetis, last accessed, 2016-10-11.

[30] Wireshark. Offloading. https://wiki.wireshark.org/CaptureSetup/Offloading, 2013. last accessed, 2016-10-11.

[31] STMicroelectronics. Stm32 32-bit arm cortex mcus. http://www.st.com/content/st_com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus.html?querycriteria=productId=SC1169. last accessed, 2016-10-11.

Appendices

Appendix A
Other tables and graphs

A.1 ANOVA

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Message type	3	68120.50	22706.83	116.86	0.0000
Residuals	23612	4587882.85	194.30		

Table A.1: ANOVA table for message types for the simulated solution

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Message type	3	69274.63	23091.54	0.57	0.6349
Residuals	808	32731924.28	40509.81		

Table A.2: ANOVA table for message types for the ported solution

A.2 Graphs

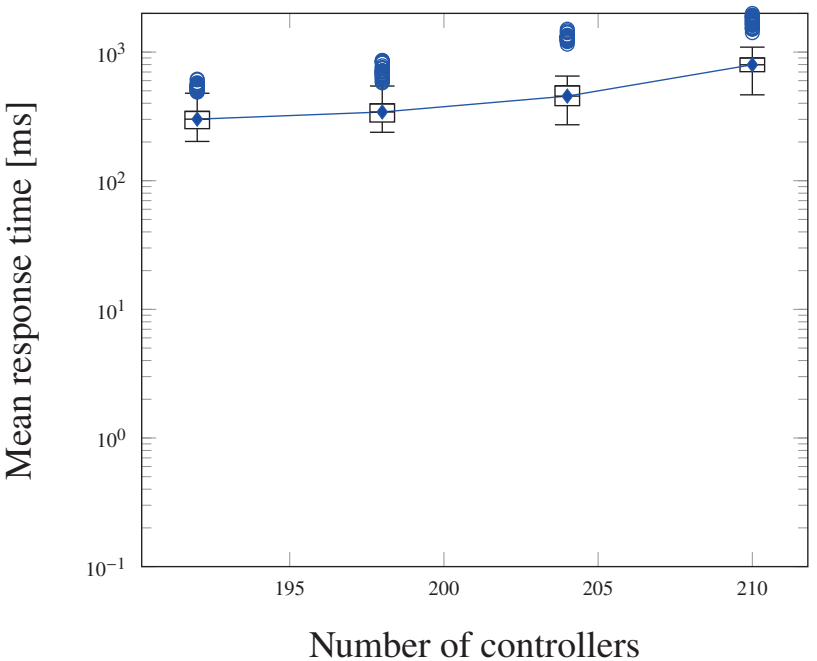

Figure A.1: A plot with the mean response times for the ported solution compiled with `-O2` for different numbers of controllers with box-plots for selected values of controller instances.

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER´S THESIS

Date of issue
November 2016

Document Number
ISRN LUTFD2/TFRT--6022--SE

Author(s)
Dalibor Lovric
Christian Olsson

Supervisor
Harald Stribén, Regin
Anton Cervin, Dept. of Automatic Control, Lund University, Sweden
Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner)

Sponsoring organization

Title and subtitle
Virtual Controllers

Abstract

Small ARM Cortex CPU based system boards, called controllers, are used in building automation for regulation of heating, ventilation, and air conditioning. A controlling project can incorporate several thousands of these controllers. The controllers communicate with a SCADA system over the TCP/IP protocol. For the purpose of testing the Supervisory Control And Data Acquisition (SCADA) system when communicating with several hundred controllers simultaneously, a software implementation of a controller that can run in multiple instances, is needed. In this thesis, three different kinds of virtual controllers are proposed and evaluated for their performance. The performance data is based on controller's response time and is acquired in a benchmark tool that is simulating SCADA. The implementation work consisted of designing and implementing a benchmark tool and three controller solutions: emulated, ported and simulated. The three solutions differ significantly in the number of instances that can be run simultaneously on the same machine. The conclusion is that the simulated solution is the most suitable since it can run in 6000 instances contra the ported with 200 instances. The emulated solution was eventually deemed as impractical to accomplish in the scope of this thesis.

Keywords
Virtual controllers, RTOS, Building automation, Response time, Porting, SCADA

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-52

Recipient's notes

Security classification

http://www.control.lth.se/publications/