Embedded ECU dual CPU Emulator

VENKAT KIRAN KUNDLA
ABHIJIT SHIVAKUMAR INAMDAR

MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Embedded ECU dual CPU Emulator

Venkat Kiran Kundla (ve8820ku-s)

Abhijit Shivakumar Inamdar (ab6722in-s)

Master's thesis in Embedded Electronics Engineering

2023

Lund University

Technical Supervisor: Nilsson Christofer

Academic Supervisor: Joachim Rodrigues

Examiner: Pietro Andreani

Master's Thesis 2023

Department of Electrical and Electronic Engineering

Lund University

SE-221 00 Lund

Telephone: +46 (0)46 222 0000

‘

Abstract

This thesis project implements a virtual environment that emulates an embedded Electronic Control Unit (ECU) with a dual CPU in Quick EMUlator (QEMU). Hardware rigs are an expensive and time-consuming bottleneck in the development process. To obtain an edge over rig hardware, we need to execute software verification, which allows us to improve performance without compromising hardware. Emulation technology reduces the need for physical hardware and makes embedded software testing easier in the early stages of embedded software development. QEMU, an open-source emulator for several processors, is used to emulate one of two ARM CPUs on Linux Yocto and another CPU running on the AUTOSAR. Finally, the target Telematic Gateway Software (TGW3-SW) should run with the dual CPU emulated ecosystem.

Popular Science Summary

Electronic Control Units (ECUs) are essential components in modern vehicles, responsible for controlling and managing various systems such as the engine, transmission, and brakes. In the past, ECUs were mechanically implemented, but in the 1980s there was a shift toward electronic control. The first attempt to create an entirely electronic ECU was made by Intel and Ford, who named it the Electronic Engine Control (EEC) [1].

As technology has advanced, digital systems have become the norm for ECUs due to their improved performance and ease of manipulation. However, the increasing complexity of ECU software programs has made it more challenging to efficiently test and verify their functionality. This is where the concept of emulation comes in. Emulation refers to the ability to replicate the behavior of a program on another computer, using virtual machines to run any operating system or program on the current platform [3].

This thesis project aims to address this challenge by implementing a virtual environment that emulates an ECU with a dual CPU using the open-source Quick EMUlator (QEMU) software. By emulating one of two ARM CPUs on Linux Yocto and another CPU running on the AUTOSAR, the project aims to create a more efficient and effective way to test and verify AUTOSAR-based ECU software applications that are still in development. The ultimate goal is to run the target Telematic Gateway Software (TGW3-SW) with the dual CPU emulated ecosystem.

Acknowledgements

Firstly, we would like to thank Nilsson Christofer of Volvo Trucks AB for trusting us with this fascinating project work and providing us with all the resources we needed to succeed. Our heartfelt thanks go to Tengblad Rasmus, our technical supervisor at Volvo Trucks AB, without whose knowledge and day-to-day assistance this project would not have been a success. Special thanks also go to Prathipa Poojari for his prior work and support with technical challenges throughout our thesis. Our days at Volvo would have been far less joyful without the help of all of the company's employees, who welcomed us as their own and transformed the offices into a dream workplace.

We would also like to express our extended gratitude to Joachim Rodrigues, our academic supervisor, for his valuable comments on our project documentation. Furthermore, we would like to express our heartfelt appreciation to our examiner, Pietro Andreani, for providing critical comments throughout the project and motivating us to produce work of high quality.

Lastly, we would like to thank our families and friends for their patience, love, and uninterrupted support throughout our studies.

Nomenclature

AUTOSAR	AUTomotive Open System ARchitecture
ECU	Electronic Control Unit
QEMU	Quick EMUlator
TGW	Telematic Gateway
CPU	Central Processing Unit
SW	Software
ACPU	Application CPU
API	Application Programming Interface
BSW	Basic Software
CAN	Control Area Network
CDD	Complex Device Drivers
DDR	Double Data Rate
EEC	Electronic Engine Control
GCC	GNU Compiler Collection
HSEM	Hardware Semaphore
HW	Hardware
ICC	Inter CPU communication
IRQ	Interrupt Request
MCAL	Microcontroller Abstraction Layer
MCU	Microcontroller Unit
MMIO	Memory-Mapped Input/Output
MMU	Memory Management Unit
MPU	Memory Protection Unit
MSW	Main Software
OCP	On-board Connectivity Platform

OS	Operating System
QOM	QEMU Object Model
RAM	Random Access Memory
SDK	Software Development Kit
SOC	System-On-Chip
SRAM	Static Random Access Memory
VCPU	Vehicle CPU
VM	Virtual Machine

Contents

List of Figures

Chapter 1

Introduction

This chapter contains a synopsis of the thesis project's content. It starts with a brief history of Electronic Control Unit (ECU) development, and an outline of the thesis follows the project description.

1.1 Background

Historically, all control mechanisms in a typical vehicle were mechanically implemented. During the 1980s, there was a significant shift toward electronic control, with Intel and Ford making the first combined attempt to create the first entirely electronic control unit, which they named Electronic Engine Control (EEC) (now known as ECU). This ECU was built around a customized version of the Intel 8061 CPU family. It's noteworthy how 8061 and its descendants served as the foundation for nearly all Ford ECUs until 2000 [1]. Because analog circuits are not clock-speed dependent, they were used in early ECUs. Before the final move to purely digital circuitry ECU systems around 1987, there was a temporary conversion to hybrid ECU systems, which included analog and digital logic.

The analog-to-digital transition occurred because it corresponded with the moment when digital devices were fast enough to process data and respond in real-time[2]. As ECU technology advanced, digital systems demonstrated improved performance and ease of manipulation. Nonetheless, the revolution in automotive electronics has resulted in a massive increase in ECU software programs developed to execute critical vehicle operations[3].

In terms of growth, software verification would give us an advantage over rig hardware because virtual rigs are easy to implement. Virtual rigs can be quickly scaled and maintained compared to HW rigs, which are more challenging to scale and maintain. Early integration on real hardware before implementing the target is an advantage of SW verification. Emulators and emulation will test the design with the actual input data in a simulated environment. Emulation refers to the ability to replicate the behavior of a program on another computer. Using virtual machines, the emulator will enable us to run any operating system or program on the current platform.

For the thesis, QEMU was chosen since Volvo had already done some important foreground work on this project, which made it possible to get started on the thesis. In the following chapters, we'll take a look at how the chosen method fits into the overall development and testing process.

1.2 Problem Description

By creating a friendly environment for AUTOSAR application developers, the process of integrating ECU modules was accelerated. As a result, a more frequent and precise testing environment was required to demonstrate the properties of the target hardware platform. Given that all ECU applications are hardware-dependent, the ideal situation would be to equip each designer with a real board so that they could download and test the application while it was still in development.

The project aims to simplify the testing and verify AUTOSAR based ECU software applications that are still in development. Because the desired hardware is frequently a limited resource by nature, and sometimes it is even being developed at the same time, a lack of proper testing creates a risky situation. This will be addressed by the development of a verification platform, which will help decrease the risks and costs of producing new ECU software by invoking the target hardware board for testing only at the end of the process. This platform is a Virtual Machine (VM) that runs AUTOSAR OS on top of a Linux distribution[3].

1.3 Virtualization

Virtualization creates an abstraction layer over the computer hardware using software that permits the hardware elements of the computer components, like processors, memory, storage, and more, to be divided into multiple virtual computers, mainly referred to as virtual machines (VMs). A VM runs its required operating system and stands as an independent computer, although it utilizes just a portion of the particular underlying hardware. A hypervisor, a software layer, will help interact with the VMs. It acts as a communication interface between a VM and the underlying physical hardware and ensures access to the correct physical resources a VM has to execute. It even ensures that the present VMs don't interfere by affecting each other's memory space or compute cycles. There are two kinds of hypervisors:

Type 1: Hypervisors interact directly with the underlying physical resources, replacing the typical operating system.

Type 2: This type runs as an application on an existing OS. This type is mainly used on endpoint devices to run another OS. They must use the host OS to access and coordinate with the underlying hardware resources; they carry a performance overhead [4].

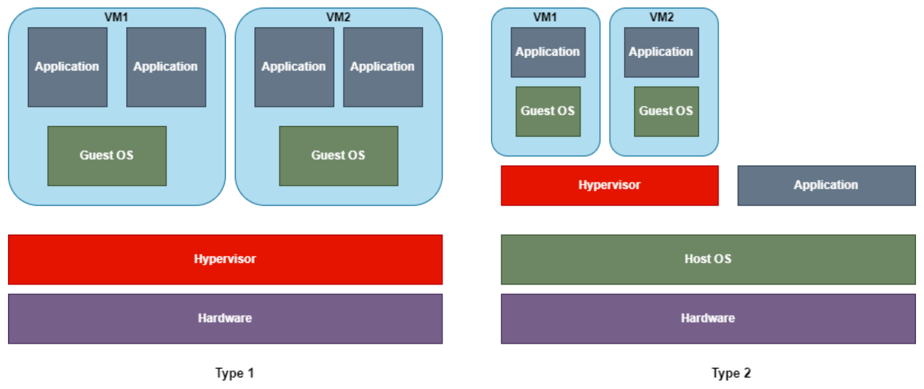


Figure 1.1: Hypervisor Types

This thesis project will try and emulate the vehicle CPU (VCPU) using the type 2 hypervisor, while the application CPU (ACPU) is already emulated within the virtual machine.

1.4 Project Goal

The project aims to emulate a dual CPU for an embedded ECU to speed up the SW verification. It should be able to run and be tested in a virtual environment. Build a platform that allows AUTOSAR and Linux to cohabit. The result would be a Linux C-based platform running an AUTOSAR application in a Linux OS environment.

1.5 Outline

This thesis work is categorized into 6 chapters.

Chapter 1: Introduction, provides an insight into the problems, how this study is relevant to the problem, and an outline about the target system

Chapter 2: Theory, explains the theory behind Operating System, Qemu, and AUTOSAR.

Chapter 3: Architecture, discusses the architecture of TGW3 and the memory map of Cortex-M3.

Chapter 4: Implementation, discusses the environment setup, workflow of QEMU and VCPU on QEMU.

Chapter 5: Results and Discussion, discusses the implementation results and experiences of testing the embedded platform to boot up in the emulated environment

Chapter 6: Conclusion, presents the final analysis and the future plans regarding the study

Chapter 2

Theory

This chapter gives insight into various topics relevant to this Master's thesis. Its goal is to lay the theoretical groundwork for the project's work and thoroughly investigate relevant basics in order to provide a complete view of the topics covered in this report.

We began by providing a general explanation of operating systems, illustrating various categories relevant to our project, such as embedded OS. Next, we discuss QEMU followed by the AUTOSAR OS functionality.

2.1 Operating Systems

A computer's operating system is a set of software components designed to manage the machine's hardware. Historically, the most well-known operating systems have been Microsoft Windows, UNIX-based (Uniplexed Information and Computing Service) distributions (e.g., Linux, Ubuntu, and Macintosh OS X). Because these three alternatives have served as the foundation for the vast majority of personal computers and mobile devices ever used, consumers engage with at least one of them daily.

Figure 2.1 demonstrates the architecture of the general concept behind an operating system. The illustrated architecture includes an application layer with service programs that control alarms and tasks, shared libraries, and a kernel. The kernel is an operating system layer that handles resource allocation (such as hardware access) for a computer system. The kernel is in charge of preventing various user and system applications from accessing restricted memory locations [5].

The hardware sits beneath the software, and its purpose is to execute the software instructions in the programs, libraries, and operating system. The major components are the CPU, which executes the instructions. It does, however, have immediate access to a limited number of extremely fast memory regions known as registers. We require main memory, which has a large amount of much slower data storage, to store vast amounts of data while a program is running, which holds the machine instructions for the programs currently running and the data they are consuming.

The CPU and main memory are linked to the system's peripheral devices, including I/O devices like keyboards and displays, network interfaces, hard disks, etc. Main memory is typically a hundred times slower than the CPU, and peripheral devices are thousands of times slower.

The hardware is interconnected by a network of busses that transport data between hardware components, machine instructions, and other control signals [6].

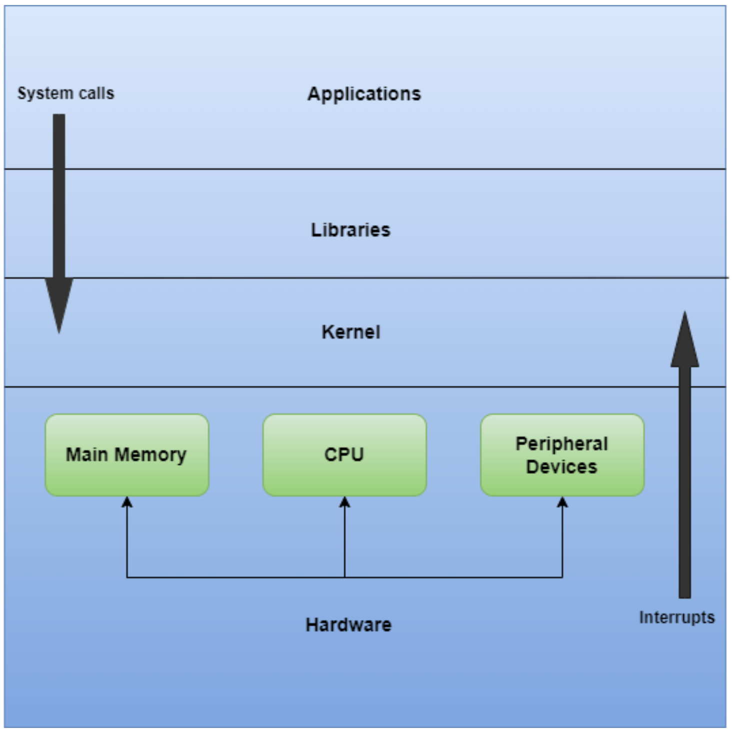


Figure 2.1: OS software architecture layout

2.2 QEMU

Quick EMUlator (QEMU) is a hypervisor that performs hardware virtualization [7]. QEMU is an open-source project started by Fabrice Bellard. The most common architectures for the QEMU emulation are ARM, x86, PowerPC, and Sparc. More specifically, it is a Type 2 hypervisor, as stated above.

QEMU supports two different modes of operation, namely, user-mode emulation and full-system emulation. QEMU supports user-mode emulation to run a Linux process compiled for one target CPU on another CPU. User-mode emulation is just a subset of full-system emulation at the CPU level. There is no MMU simulation because QEMU supposes that the host OS handles the memory mappings. QEMU includes a generic Linux system call converter to address endianness issues and 32/64-bit conversions. QEMU accurately emulates the target signals due to its exception support. Each target thread runs on one host thread [8].

Full-system emulation, preferred during this thesis, emulates the entire computer system, including peripherals. It can boot different guest operating systems; it supports emulating different instruction sets, including x86, MIPS, 32-bit ARMv7, ARMv8, and more. Specifically, this thesis focuses on the 32-bit ARMv7 architecture as Cortex-M3 was one of the CPUs.

Emulating the guest instructions alone does not provide a full emulation. During runtime, guest applications use the required peripheral registers on the embedded device, and each register is mapped to a specific memory region. When memory read and write requests are received, QEMU can intercept them and forward them to the callbacks. It is possible to produce an IRQ or modify some device registers in response to a request from the program. This might be useful for providing feedback to the application[9].

There are many official and unofficial virtual machine implementations for QEMU, but the lack of documentation is still an obstacle for new developers. The official QEMU getting started guide for developers states the following: ”QEMU does not have a high–level design description document—only the source code tells the full story” [10]. The best way to understand how a virtual machine works are by browsing through the source code and studying patches submitted by other developers.

2.3 AUTOSAR

Automotive Open System Architecture(AUTOSAR) is an open automotive software architecture; standardized jointly by automobile tool developers, manufacturers, and suppliers. The AUTOSAR-standard enables a component-based software design model to design a vehicular system. The design model uses application software components that link through an abstract component, named the virtual function bus.

The standard layered layout of the AUTOSAR is divided into the following layers:

- Application Layer.

- Runtime Environment (RTE).

- Basic Software (BSW).

- Microcontroller Unit (MCU).

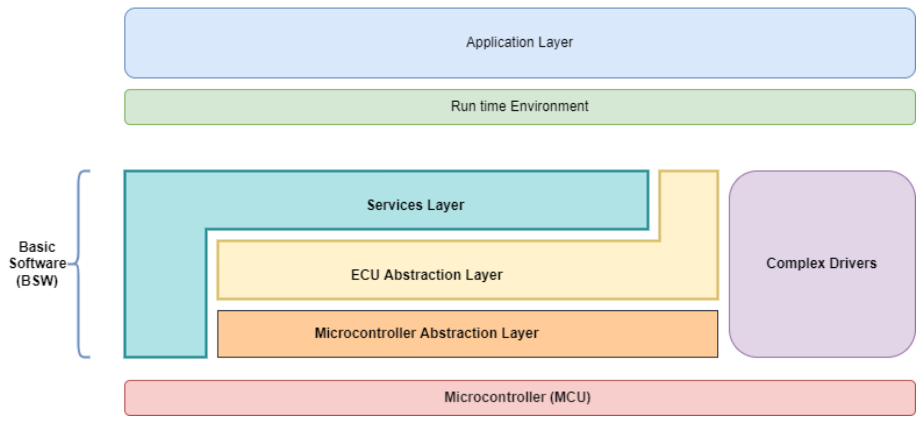


Figure 2.2: AUTOSAR software architecture

As shown in the Figure 2.2, the sub-layers are described in the following segment, consisting of discrete components:

1. Application Layer: We place the created software components for the specific functionality in this layer. In the same layer are the sensor/actuators software components according to AUTOSAR standard [11].

2. Runtime Environment: This layer serves as a conduit for communication between the application software and the outside world. [11]. Above the RTE, the software architecture style changes from "layered" to "component style." The RTE allows the software components to communicate with one another and with services [12] to call it middle-ware.

3. Basic Software: This layer is sub-divided into different layers, namely, a Service Layer, an ECU Abstraction Layer, a Complex Device Drivers (CDD) Layer, and a Microcontroller Abstraction Layer (MCAL). • The service layer is the highest layer of the BSW which applies for its relevance for the application software: while the ECU Abstraction layer covers access to I/O signals [12].

• The ECU Abstraction layer interfaces the drivers of the next layer, which is the Microcontroller Abstraction Layer. It also contains drivers for external devices. It offers API for access to the peripherals and devices regardless of their location and connection to the microcontroller [12].

• The Microcontroller Abstraction layer is the lowest software layer of the BSW. It contains internal drivers, software modules with direct access to the microcontroller and the internal peripherals [12].

4. Microcontroller Unit: It is the hardware layer of the ECU and communicates with the AUTOSAR BSW through the MCAL and CDD.

Chapter 3

Architecture

3.1 TGW3 Architecture

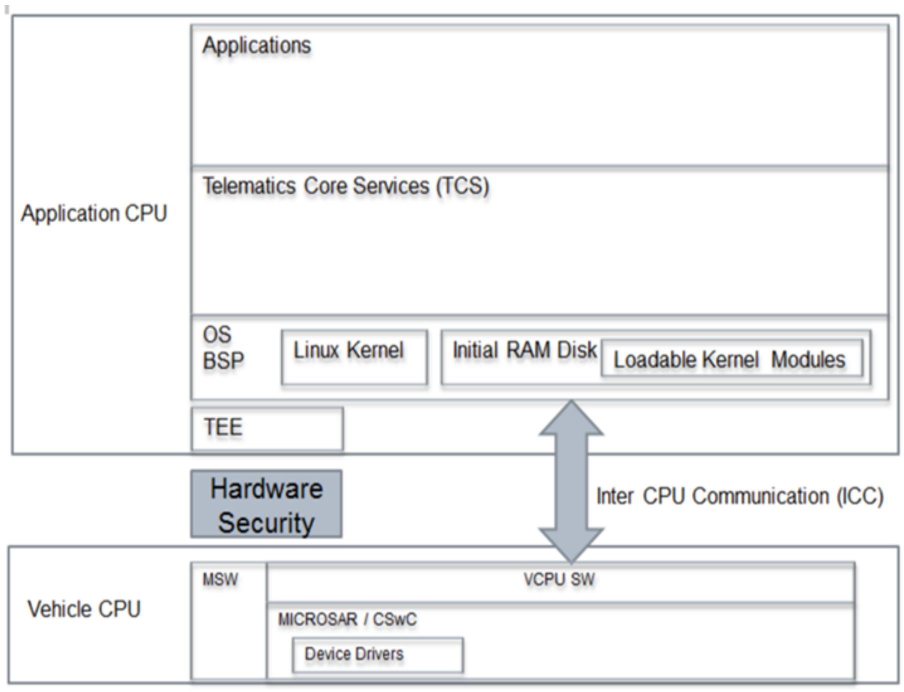


Figure 3.1: TGW3 Architecture (Source: Volvo)

As shown in the above Figure 3.1, the TGW3 is a multi-CPU solution, and the architecture is divided into two modules. The first module is an Application CPU, and the other one is Vehicle CPU. Both CPUs are connected via Inter CPU communication. The Vehicle CPU is emulated on AUTOSAR OS via QEMU, and the Application CPU is emulated on Linux OS via QEMU. The Vehicle CPU and the Application CPU act as one Logical ECU; for example, a reset of the VCPU will cause a reset of the ACPU. The main aim is to emulate both the CPUs in an OCP environment.

The main software (MSW) in the vehicle CPU is the core of the ECU. This repository contains the whole binary image of the VCPU SW, which needs to be emulated on QEMU. The VCPU SW is based on the MICROSAR, which Vector Informatik developed and cloned into the MSW repository. The core software modules of the MICROSAR packages are responsible for ensuring that the ECU performs its essential functions. They contain the AUTOSAR standard services implementations required for functional software to work correctly.

3.2 Memory Map of Cortex-M3

The memory of the Cortex-M3 is divided into sections, as shown in Figure 3.2

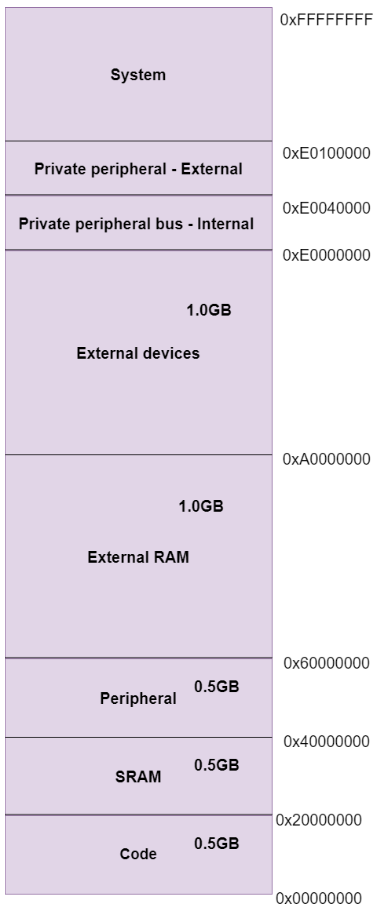


Figure 3.2: Memory Map of Cortex-M3 (Source: ARM Documentation)

The memory on the embedded platform is divided into sections, namely CODE, SRAM, and the PERIPHERALS sections. The CODE section will be loaded by a binary file produced by the compilers. It contains the program sections that will be executed during device operation. Also, any data saved into non-volatile memory is held in the same file because that binary file is the exact mirror of the static data stored in the code area. SRAM is the operating systems program memory, which in QEMU is just a byte array. It is safe to assume that the peripheral region is some message passing interface. For that reason, each' read' and' write' operations are processed by the emulator. The device memory is mapped to these input and output actions.

As shown in Figure 3.2, the CODE region begins at 0x00000000, the SRAM area begins at 0x20000000, and the PERIPHERAL region begins at 0x40000000. It does not imply that entire R/W access to the peripheral region is permissible. Attempting to access an incorrect register will result in a hardware exception. In QEMU, developers can configure the valid address range and register R/W call-backs for each peripheral.

If the device is unimplemented, an exception will be thrown on the virtual CPU since it is invalid. The relevant parameters, such as the register offset and the write value, are passed to the reading and write call-backs. The read function returns a value instantly available in the virtual CPU.

Depending on the peripheral, a write operation may initiate a job or alter the active configuration. Interrupts, for example, are deactivated and enabled using a set of registers specific to each peripheral. We may learn about the interrupt setup by reading the value from the interrupt registers. In addition, the guest system can use write-only device registers to initiate and stop tasks.

The Interrupt Vector table, stack pointer, and reset handler must be present in the start-up code, often known as boot code. Exception handlers are present in the start-up code. In the Cortex-M3, memory alignment is done in words (32 bits), and vector table interrupts are organized appropriately, starting at 0x0[13][14]. The Thumb instruction set was utilized, generally comprising 16 bit long (half word-aligned) instructions and a few 32-bit long ones [15]. The developer can pick between ARM and Thumb for the instruction set format, although only one design can be active at a time. The supportive scripts were taken from the CMSIS git repository [16].

The linker program is the (.ld) script that links the application code with the start-up code. The processor must leap to the application code from start-up, and the processor must jump to the application code, which gets connected by the linker script. The linker script converts the start-up and application code to object files and links them together.

The linker script defines the System-On-Chip (SOC) memory layout, including memory locations such as RAM and Flash with allotted memory space. The vector table is stored in flash memory at the location 0x0. With the GNU bin-utilities command, readelf -s/-S/-an filename.elf [15], the memory structure and code location may be validated. The memory allocation and vector table placement in the memory layout of the created elf are displayed by readelf -s/-S/-an filename.elf command.

Chapter 4

Implementation

This chapter provides a detailed explanation of how the project team worked, detailing and summarizing the overall work structure. This begins with a description of the On-Board Connectivity Platform (OCP), which serves as the project's primary working platform. The tools that were utilized to accomplish the desired results will be discussed in the section that follows. The final segment, which is organized in accordance with the iterations, discusses what we did during the entire project.

4.1 Environment Setup

The OCP SW Development Environment is introduced in the following manner:

The Ubuntu Linux operating system serves as the foundation for the development of OCP applications. With the help of the following command, the OCP Application SDK was installed on a VMware (virtual machine):

ocp-setup-client

The OCP Application SDK is used to develop the actual libraries and applications for OCP, which are then distributed to users. The TGW3 hardware will eventually be equipped with the libraries, headers, and tools contained within this package. By using this tool, OCP developers can cross-compile a binary or library for the TGW3 (ARM) target hardware from their development (virtual) machine. It also includes a QEMU emulator, which relieves OCP developers from the limits of hardware and allows them to debug applications that have been cross-compiled for target hardware on target hardware that has been emulated. Debugging an OCP application in the development environment requires two virtualization layers. This will, of course, result in certain limitations in terms of speed. Still, it will make it easier to verify many functional requirements that do not require special hardware or severe time limits in their verification.

In order to build OCP software, we use the Yocto SDK, C++14, and several Linux-based tools, including CMake, Git, and gcc. Eclipse and CMake are used in the development of OCP software.

4.2 QEMU Build

The qemu-4.2.0 source files were cloned from the Github project into the vcpu_-on_qemu repository. The vcpu_on_qemu repo contains the following:

```
└── vcpu_on_qemu
        ├── gcc-arm/
        ├── proj/
        ├── qemu-2.10/
        ├── qemu-4.2.0/
        └── readme.txt
```

The gcc-arm directory holds the tools for compiling and linking the arm binary. The gcc-arm-none-eabi-6-2017-q2-update toolchain was downloaded from the arm developer website[12].

Proj: This folder holds test projects that can be built and run on QEMU using the above toolchain.

Before adding the custom machine file to the QEMU, we need to build it from the source to our required target. The commands for building the QEMU is as follows:

```
# Install pixman

$ sudo apt-get install libpixman-1-dev

$ cd qemu-4.2.0

$ mkdir build

$ cd build

$ ../configure --disable-werror --enable-debug

--target-list="arm-softmmu"

$ make
```

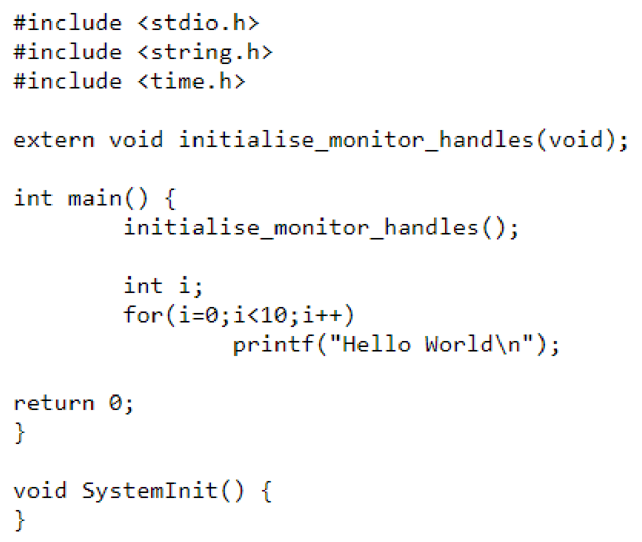
4.3 ARM cortex-m3 binary in QEMU

The steps to run a ARM cortex-m3 binary in QEMU is as follows:

The proj/simple folder has two simple applications built and linked with GCC tools: a hello_world application and an application that tests that the systick is working.

```
$ cd proj/simple
```

The hello_world.c is given in Figure 4.1



```
#include <stdio.h>
#include <string.h>
#include <time.h>

extern void initialise_monitor_handles(void);

int main() {
        initialise_monitor_handles();

        int i;
        for(i=0;i<10;i++)
                printf("Hello World\n");

return 0;
}

void SystemInit() {
}
```

Figure 4.1: hello_world.c

The following command is used to compile the application into binary or a executable is:

```
$ ../../gcc-arm/gcc-arm-none-eabi-6-2017-q2-update/bin/
arm-none-eabi-gcc hello_world.c startup_ARMCM3.S
-mthumb -mcpu=cortex-m3
-D_start=main -Os -flto -ffunction-sections -fdata-sections
--specs=nano.specs --specs=rdimon.specs -Wl,--gc-sections
-Wl,-Map=main.map -T gcc.ld -o main.axf
```

Notes:

linker command $--specs = nano.specs$, links against the reduced size nano-libc

linker command $--specs = rdimon.specs$ links against semihosting libraries, which enables some of the host i/o facilities like stdio for printf.

the linker command file gcc.ld, should locate FLASH at 0x0 and RAM at 0x20000000, which corresponds to the memory map of the cortex-m3 core.

4.3.1 Adding a Custom Machine to QEMU

The QEMU code-base is well-structured into specific folders. The main file vl.c, where the initial QEMU application is located, is in the folder /qemu-4.2.0. Processor architectures such as Alpha, ARM, i386, MIPS are emulated and are present in the /qemu-4.2.0/hw directory. As discussed in this project, the processor is the ARM Cortex-M3, and hence code for the custom machine should be placed in the /hw/arm folder. The same directory consists of emulated hardware source codes. This folder also has a makefile.objs file containing the list of source code file names that must be converted to architecture-specific object files using obj-y. The 'y' symbolizes using the architecture CONFIG_-ARM_v7M, the default entry.

The supported machines in QEMU are:

```
akita                Sharp SL-C1000 (Akita) PDA (PXA270)
ast2500-evb          Aspeed AST2500 EVB (ARM1176)
ast2600-evb          Aspeed AST2600 EVB (Cortex A7)
borzoi               Sharp SL-C3100 (Borzoi) PDA (PXA270)
canon-a1100          Canon PowerShot A1100 IS
cheetah              Palm Tungsten|E aka. Cheetah PDA (OMAP310)
collie               Sharp SL-5500 (Collie) PDA (SA-1110)
connex               Gumstix Connex (PXA255)
cubieboard           cubietech cubieboard (Cortex-A9)
emcraft-sf2          SmartFusion2 SOM kit from Emcraft (M2S010)
highbank             Calxeda Highbank (ECX-1000)
imx25-pdk            ARM i.MX25 PDK board (ARM926)
integratorcp         ARM Integrator/CP (ARM926EJ-S)
kzm                  ARM KZM Emulation Baseboard (ARM1136)
lm3s6965evb          Stellaris LM3S6965EVB
lm3s811evb           Stellaris LM3S811EVB
mainstone            Mainstone II (PXA27x)
mcimx6ul-evk         Freescale i.MX6UL Evaluation Kit (Cortex A7)
mcimx7d-sabre        Freescale i.MX7 DUAL SABRE (Cortex A7)
microbit             BBC micro:bit
midway               Calxeda Midway (ECX-2000)
mps2-an385           ARM MPS2 with AN385 FPGA image for Cortex-M3
mps2-an505           ARM MPS2 with AN505 FPGA image for Cortex-M33
mps2-an511           ARM MPS2 with AN511 DesignStart FPGA image for Cortex-M3
mps2-an521           ARM MPS2 with AN521 FPGA image for dual Cortex-M33
musca-a              ARM Musca-A board (dual Cortex-M33)
musca-b1             ARM Musca-B1 board (dual Cortex-M33)
musicpal             Marvell 88w8618 / MusicPal (ARM926EJ-S)
n800                 Nokia N800 tablet aka. RX-34 (OMAP2420)
n810                 Nokia N810 tablet aka. RX-44 (OMAP2420)
netduino2            Netduino 2 Machine
none                 empty machine
nuri                 Samsung NURI board (Exynos4210)
palmetto-bmc         OpenPOWER Palmetto BMC (ARM926EJ-S)
raspi2               Raspberry Pi 2
realview-eb          ARM RealView Emulation Baseboard (ARM926EJ-S)
realview-eb-mpcore   ARM RealView Emulation Baseboard (ARM11MPCore)
realview-pb-a8       ARM RealView Platform Baseboard for Cortex-A8
realview-pbx-a9      ARM RealView Platform Baseboard Explore for Cortex-A9
romulus-bmc          OpenPOWER Romulus BMC (ARM1176)
sabrelite            Freescale i.MX6 Quad SABRE Lite Board (Cortex A9)
smdkc210             Samsung SMDKC210 board (Exynos4210)
spitz                Sharp SL-C3000 (Spitz) PDA (PXA270)
```

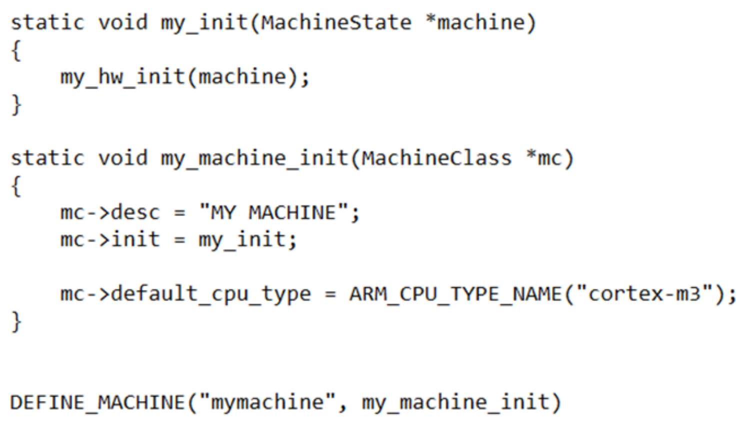
```
swift-bmc            OpenPOWER Swift BMC (ARM1176)
sx1                  Siemens SX1 (OMAP310) V2
sx1-v1               Siemens SX1 (OMAP310) V1
terrier              Sharp SL-C3200 (Terrier) PDA (PXA270)
tosa                 Sharp SL-6000 (Tosa) PDA (PXA255)
verdex               Gumstix Verdex (PXA270)
versatileab          ARM Versatile/AB (ARM926EJ-S)
versatilepb          ARM Versatile/PB (ARM926EJ-S)
vexpress-a15         ARM Versatile Express for Cortex-A15
vexpress-a9          ARM Versatile Express for Cortex-A9
virt-2.10            QEMU 2.10 ARM Virtual Machine
virt-2.11            QEMU 2.11 ARM Virtual Machine
virt-2.12            QEMU 2.12 ARM Virtual Machine
virt-2.6             QEMU 2.6 ARM Virtual Machine
virt-2.7             QEMU 2.7 ARM Virtual Machine
virt-2.8             QEMU 2.8 ARM Virtual Machine
virt-2.9             QEMU 2.9 ARM Virtual Machine
virt-3.0             QEMU 3.0 ARM Virtual Machine
virt-3.1             QEMU 3.1 ARM Virtual Machine
virt-4.0             QEMU 4.0 ARM Virtual Machine
virt-4.1             QEMU 4.1 ARM Virtual Machine
virt                 QEMU 4.2 ARM Virtual Machine (alias of virt-4.2)
virt-4.2             QEMU 4.2 ARM Virtual Machine
witherspoon-bmc      OpenPOWER Witherspoon BMC (ARM1176)
xilinx-zynq-a9       Xilinx Zynq Platform Baseboard for Cortex-A9
z2                   Zipit Z2 (PXA27x)
```

The implementation started by creating a dummy machine without a guest OS by considering the start-up and linker scripts available on the CMSIS GitHub page [16]. In the directory qemu-4.2.0/hw/arm, a simple machine was created by 'mymachine.c' with the RAM and Flash size of 1MB each by keeping the TI Stellaris model as a reference. To define a new machine in QEMU, we need to create QOM TypeInfo, associated with the MachineClass and MachineState functions.

The TypeInfo :

DEFINE MACHINE ("mymachine", my machine init)

Which generate the following code:



```
static void my_init(MachineState *machine)
{
    my_hw_init(machine);
}

static void my_machine_init(MachineClass *mc)
{
    mc->desc = "MY MACHINE";
    mc->init = my_init;

    mc->default_cpu_type = ARM_CPU_TYPE_NAME("cortex-m3");
}


DEFINE_MACHINE("mymachine", my_machine_init)
```

Figure 4.2: Initialize the Machine

This will register a new machine class type for our mymachine. It is a basic level of definition. In MachineClass, can define specific properties of the machine, such as the CPU model and, most significantly, the instance initialization method my init. Once the class is specified, an object instance will be created. The main features of our board implementation will be here.

Standard QEMU APIs

The abstraction methods used by QEMU APIs enable a very straightforward approach to creating a custom machine. Within QEMU, several categories of APIs are utilized for various reasons. A few QEMU APIs required for building a custom machine are listed below.

• MemoryRegion: The memory API model aids the QEMU machine's memory and I/O buses and controllers. Ordinary RAM, memory-mapped I/O (MMIO), and memory controllers are attempted to be modeled [17].

Types of regions represented by MemoryRegions:

RAM: A RAM region is essentially a portion of the host's memory available to the guest. Typically, memory_region_init_ram() is used to initialize them[17].

MMIO: is a set of host callbacks that implement a range of guest memory; each read or write triggers a callback on the host. In general, use memory_region_init_io() to initialize them, supplying a MemoryRegionOps structure to describe the callbacks[17].

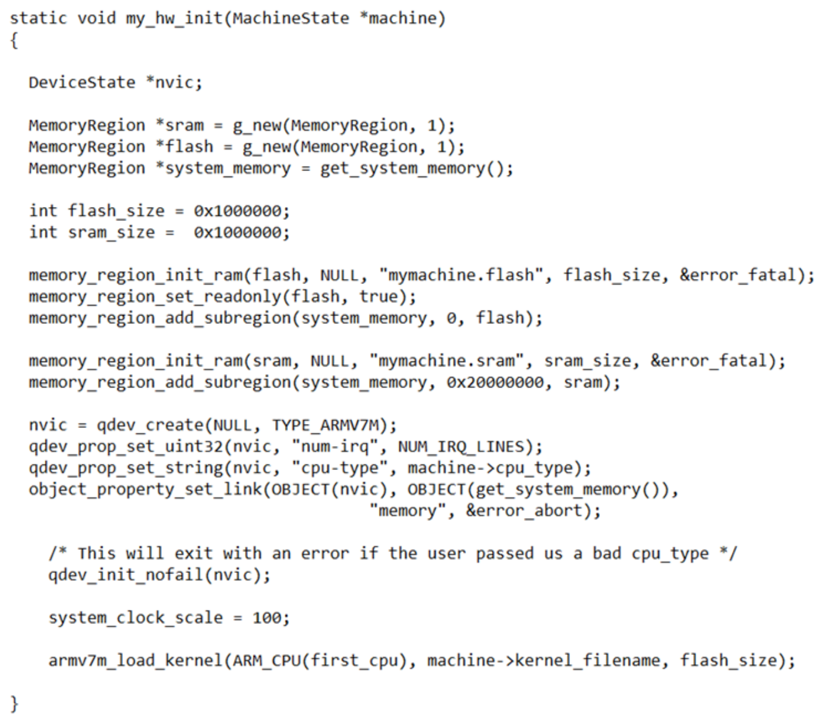
Alias: A subregion of another region is referred to as an alias. A region can be divided into discontiguous areas using aliases. Aliases can point to any location, including other aliases, but they cannot, directly or indirectly, lead back to themselves. memory_region_init_alias() is used to set them up[17].

• qdev: qdev keeps track of the device tree, organized into a hierarchy of buses and devices. As stated in the original commit, the assumption here is that it should be feasible to build a machine without knowing about other individual devices [18]. For board init processes to provide each device with a set of arguments, the board must first determine which device it deals with. This file contains an abstract API for configuring and initializing devices. Machines will inherit from a specific bus (e.g., I2C).

• QOM stands for QEMU Object Model [19][20], a framework for registering user-created kinds. User-defined types can be created and instantiated with QOM.

Some API groups fully utilize QEMU's capabilities and reduce the developer's workload while programming sophisticated hardware.

The following Figure 4.3 will give the reader how the memory regions instantiated.



```
static void my_hw_init(MachineState *machine)
{

  DeviceState *nvic;

  MemoryRegion *sram = g_new(MemoryRegion, 1);
  MemoryRegion *flash = g_new(MemoryRegion, 1);
  MemoryRegion *system_memory = get_system_memory();

  int flash_size = 0x1000000;
  int sram_size =  0x1000000;

  memory_region_init_ram(flash, NULL, "mymachine.flash", flash_size, &error_fatal);
  memory_region_set_readonly(flash, true);
  memory_region_add_subregion(system_memory, 0, flash);

  memory_region_init_ram(sram, NULL, "mymachine.sram", sram_size, &error_fatal);
  memory_region_add_subregion(system_memory, 0x20000000, sram);

  nvic = qdev_create(NULL, TYPE_ARMV7M);
  qdev_prop_set_uint32(nvic, "num-irq", NUM_IRQ_LINES);
  qdev_prop_set_string(nvic, "cpu-type", machine->cpu_type);
  object_property_set_link(OBJECT(nvic), OBJECT(get_system_memory()),
                                   "memory", &error_abort);

    /* This will exit with an error if the user passed us a bad cpu_type */
    qdev_init_nofail(nvic);

    system_clock_scale = 100;

    armv7m_load_kernel(ARM_CPU(first_cpu), machine->kernel_filename, flash_size);

}
```

Figure 4.3: Definition of MemoryRegions in Machine

The command-line to invoke the custom machine created in Linux is as follows:

gcc-arm/gcc-arm-none-eabi-6-2017-q2-update/bin/arm-none-eabi-gcc -g hello -world.c Startup ARMCM3.S -mthumb -mcpu=cortex-m3 -D start=main -Os -flto -ffunction-sections -fdata-sections –specs=nano.specs –specs=rdimon.specs -Wl,–gc-sections -Wl,-Map=main.map -T mymachine.ld -o main.axf

The above command is used for compiling.

qemu-4.2.0/build/arm-softmmu/qemu-system-arm -M mymachine -cpu cortex-m3 -kernel main.axf -monitor none -serial stdio -semihosting -nographic

From the above command, one can learn that the configuration was in a directory called 'build.' the -M flag gives the user to specify the machine which was added to emulate. The -cpu flag will let one select the CPU type and -kernel to pass the required binary file, and the rest of the flags are to re-direct the console output of the QEMU onto the Linux terminal. Using the above commands, QEMU executes the cortex-m3 binary main.axf.

4.4 VCPU on QEMU

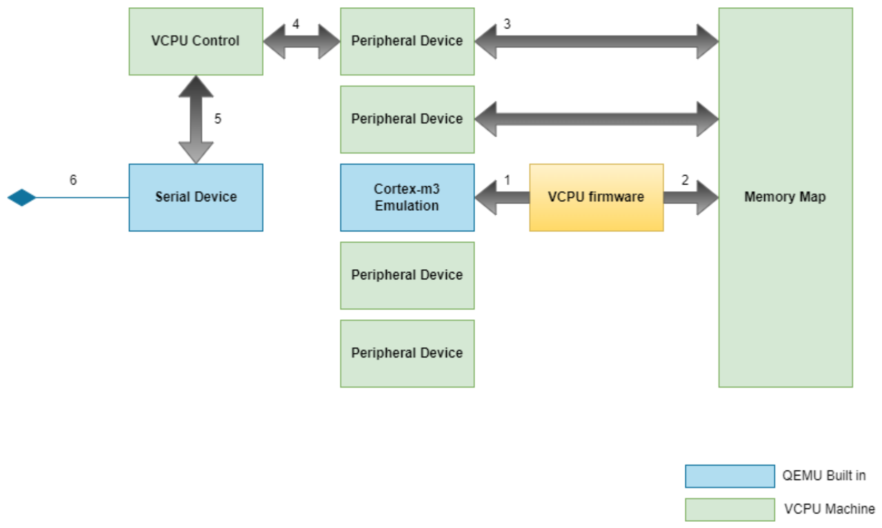


Figure 4.4: VCPU on QEMU

Figure 4.4 shows the mechanism of how the emulation works. The functionality of each module is as follows:

• The qemu cortex-m3 emulation initially loads and executes the VCPU firmware in QEMU.

• The VCPU's firmware interacts with peripheral devices through memory-mapped registers.

• The VCPU machine intercepts R/W-operations performed by the firmware and emulates the expected behavior. In the case of a communication device, e.g., a write operation to a CAN TX register, the written data can be fetched and forwarded to the VCPU control device.

• VCPU control device, in turn, passes data to the serial device via the VCPU control device.

• The serial device makes the data available to the host machine, for example, through a socket.

Steps to run VCPU firmware under QEMU:

• Linking:

The scatter file (vcpu.ld) used for linking the VCPU binary places the vector table at address 0x10000000 in RW memory and the program code in a flash starting at 0x70000000. However, the Qemu cortex-m3 emulation expects the vector table to be located at address 0x0, so a new scatter file (vcpu_qemu.ld) was created to be used when building for qemu. In this scatter file, the flash memory is located at 0x0, starting with the vector table, followed by program code.

• VCPU custom QEMU machine:

For the VCPU, a custom machine has been built and placed in qemu-4.2.0/hw/arm/tgw3/vcpu.c. It can be started by specifying "-M vcpu" on the command line when starting QEMU. The VCPU machine sets up two memory regions: FLASH, located at 0x0, and RAM, at 0x10000000. In addition to the memory setup, the VCPU machine emulates several peripheral devices. These emulations are not fully functional models of the actual hardware; they provide as much functionality as is needed for the VCPU firmware to run.

• QEMU is started with: qemu-4.2.0/build/arm-softmmu/qemu-system-arm -M vcpu -cpu cortex-m3 -kernel vcpu.elf -monitor none -serial stdio -semihosting -nographic

To check the functionalities added placed in the correct addressed spaces, we make use of the command:

qemu-system-arm -M vcpu -s -S -monitor stdio

Where -s is a shorthand to connect with the VNC server running on 127.0.0.1:5900, and -S is to freeze the CPU until a command to execute. Once the QEMU monitor pops, one can use different options to explore the machine; two of such are

-info mtree: shows the VM guest memory hierarchy.

-info qtree: shows the device tree.

the output of these commands is seen below:

```
     QEMU 4.2.0 monitor - type 'help' for more information
(qemu) VNC server running on 127.0.0.1:5900
info mtree
address-space: memory
  0000000000000000-ffffffffffffff (prio -1, i/o): system
    0000000010000000-000000001003ffff (prio 0, ram): tgw.sram
    0000000040020000-000000004002007b (prio 0, i/o): stasrcm3
    00000000481c0000-00000000481c017b (prio 0, i/o): staadc
    0000000049000000-00000000490002ff (prio 0, i/o): staccc
    0000000050020000-0000000050020fff (prio 0, i/o): stauart
    0000000050150000-0000000050150fff (prio 0, i/o): stauart
    0000000050160000-0000000050160fff (prio 0, i/o): stai2c
    0000000050170000-0000000050170fff (prio 0, i/o): ssp
    0000000050180000-00000000501827ff (prio 0, i/o): stacanram
    0000000050183000-00000000501830ff (prio 0, i/o): stacan
    0000000050183400-00000000501834ff (prio 0, i/o): stacan
    0000000050200000-000000005020011b (prio 0, i/o): stasqi
    0000000050600000-00000000506004ff (prio 0, i/o): staddr_cntl
    0000000050900000-00000000509002ff (prio 0, i/o): staddr_pub
    0000000068000000-000000006800014b (prio 0, i/o): stassca7
    0000000070000100-00000000700800ff (prio 0, ram): tgw.flash
    00000000bd000000-00000000bdffffff (prio 0, ram): tgw.ddr

address-space: I/O
  0000000000000000-000000000000ffff (prio 0, i/o): io

address-space: cpu-memory-0
  0000000000000000-ffffffffffffffff (prio 0, i/o): armv7m-container
    0000000000000000-ffffffffffffffff (prio -1, i/o): system
      0000000010000000-000000001003ffff (prio 0, ram): tgw.sram
      0000000040020000-000000004002007b (prio 0, i/o): stasrcm3
      00000000481c0000-00000000481c017b (prio 0, i/o): staadc
      0000000049000000-00000000490002ff (prio 0, i/o): staccc
      0000000050020000-0000000050020fff (prio 0, i/o): stauart
      0000000050150000-0000000050150fff (prio 0, i/o): stauart
      0000000050160000-0000000050160fff (prio 0, i/o): stai2c
      0000000050170000-0000000050170fff (prio 0, i/o): ssp
      0000000050180000-00000000501827ff (prio 0, i/o): stacanram
      0000000050183000-00000000501830ff (prio 0, i/o): stacan
      0000000050183400-00000000501834ff (prio 0, i/o): stacan
      0000000050200000-000000005020011b (prio 0, i/o): stasqi
      0000000050600000-00000000506004ff (prio 0, i/o): staddr_cntl
      0000000050900000-00000000509002ff (prio 0, i/o): staddr_pub
      0000000068000000-000000006800014b (prio 0, i/o): stassca7
      0000000070000100-00000000700800ff (prio 0, ram): tgw.flash
      00000000bd000000-00000000bdffffff (prio 0, ram): tgw.ddr
    0000000022000000-0000000023ffffff (prio 0, i/o): bitband
    0000000042000000-0000000043ffffff (prio 0, i/o): bitband
    00000000e000e000-00000000e000efff (prio 0, i/o): nvic
      00000000e000e000-00000000e000efff (prio 0, i/o): nvic_sysregs
      00000000e000e010-00000000e000e0ef (prio 1, i/o): nvic_systick

address-space: bitband-source
  0000000000000000-ffffffffffffffff (prio -1, i/o): system
    0000000010000000-000000001003ffff (prio 0, ram): tgw.sram
    0000000040020000-000000004002007b (prio 0, i/o): stasrcm3
    00000000481c0000-00000000481c017b (prio 0, i/o): staadc
    0000000049000000-00000000490002ff (prio 0, i/o): staccc
    0000000050020000-0000000050020fff (prio 0, i/o): stauart
    0000000050150000-0000000050150fff (prio 0, i/o): stauart
    0000000050160000-0000000050160fff (prio 0, i/o): stai2c
    0000000050170000-0000000050170fff (prio 0, i/o): ssp
    0000000050180000-00000000501827ff (prio 0, i/o): stacanram
    0000000050183000-00000000501830ff (prio 0, i/o): stacan
    0000000050183400-00000000501834ff (prio 0, i/o): stacan
    0000000050200000-000000005020011b (prio 0, i/o): stasqi
    0000000050600000-00000000506004ff (prio 0, i/o): staddr_cntl
    0000000050900000-00000000509002ff (prio 0, i/o): staddr_pub
    0000000068000000-000000006800014b (prio 0, i/o): stassca7
    0000000070000100-00000000700800ff (prio 0, ram): tgw.flash
    00000000bd000000-00000000bdffffff (prio 0, ram): tgw.ddr

address-space: bitband-source
  0000000000000000-ffffffffffffffff (prio -1, i/o): system
    0000000010000000-000000001003ffff (prio 0, ram): tgw.sram
    0000000040020000-000000004002007b (prio 0, i/o): stasrcm3
    00000000481c0000-00000000481c017b (prio 0, i/o): staadc
    0000000049000000-00000000490002ff (prio 0, i/o): staccc
    0000000050020000-0000000050020fff (prio 0, i/o): stauart
    0000000050150000-0000000050150fff (prio 0, i/o): stauart
    0000000050160000-0000000050160fff (prio 0, i/o): stai2c
    0000000050170000-0000000050170fff (prio 0, i/o): ssp
    0000000050180000-00000000501827ff (prio 0, i/o): stacanram
    0000000050183000-00000000501830ff (prio 0, i/o): stacan
    0000000050183400-00000000501834ff (prio 0, i/o): stacan
    0000000050200000-000000005020011b (prio 0, i/o): stasqi
    0000000050600000-00000000506004ff (prio 0, i/o): staddr_cntl
    0000000050900000-00000000509002ff (prio 0, i/o): staddr_pub
    0000000068000000-000000006800014b (prio 0, i/o): stassca7
    0000000070000100-00000000700800ff (prio 0, ram): tgw.flash
    00000000bd000000-00000000bdffffff (prio 0, ram): tgw.ddr
-------------------------------------------------------------------------------------------
(qemu) info qtree
bus: main-system-bus
  type System
  dev: vcpu-control, id ""
    chardev_out = "serial0"
    chardev_in = ""
  dev: vcpu-sta-can-ram, id ""
    mmio 0000000050180000/0000000000002800
  dev: vcpu-sta-sqi, id ""
    mmio 0000000050200000/000000000000011c
  dev: vcpu-sta-can, id ""
    mmio 0000000050183400/0000000000000100
  dev: vcpu-sta-can, id ""
    mmio 0000000050183000/0000000000000100
  dev: vcpu-sta-ccc, id ""
    mmio 0000000049000000/0000000000000300
  dev: vcpu-sta-ddr-cntl, id ""
    mmio 0000000050600000/0000000000000500
  dev: vcpu-sta-ddr-pub, id ""
    mmio 0000000050900000/0000000000000300
  dev: vcpu-sta-i2c, id ""
    mmio 0000000050160000/0000000000001000
    bus: stai2c
      type i2c-bus
  dev: vcpu-sta-adc, id ""
    mmio 00000000481c0000/000000000000017c
  dev: vcpu-sta-uart, id ""
    mmio 0000000050020000/0000000000001000
  dev: vcpu-sta-uart, id ""
    mmio 0000000050150000/0000000000001000
  dev: ssp, id ""
    gpio-out "sysbus-irq" 1
    mmio 0000000050170000/0000000000001000
    bus: ssi
      type SSI
  dev: vcpu-sta-ssca7, id ""
    mmio 0000000068000000/000000000000014c
  dev: vcpu-sta-srcm3, id ""
    mmio 0000000040020000/000000000000007c
  dev: armv7m, id ""
    gpio-in "NMI" 1
    gpio-out "SYSRESETREQ" 1
    gpio-in "" 64
    cpu-type = "cortex-m3-arm-cpu"
    memory = "/machine/unattached/system[0]"
    idau = ""
    init-svtor = 0 (0x0)
    enable-bitband = true
    start-powered-off = false
    vfp = true
    dsp = true
  dev: ARM,bitband-memory, id ""
    base = 1073741824 (0x40000000)
    source-memory = "/machine/unattached/system[0]"
    mmio ffffffffffffffff/0000000002000000
  dev: ARM,bitband-memory, id ""
    base = 536870912 (0x20000000)
    source-memory = "/machine/unattached/system[0]"
    mmio ffffffffffffffff/0000000002000000
  dev: armv7m_nvic, id ""
    gpio-in "systick-trigger" 2
    gpio-out "sysbus-irq" 1
    num-irq = 80 (0x50)
    mmio ffffffffffffffff/0000000000001000
  dev: armv7m_systick, id ""
    gpio-out "sysbus-irq" 1
    mmio ffffffffffffffff/00000000000000e0
```

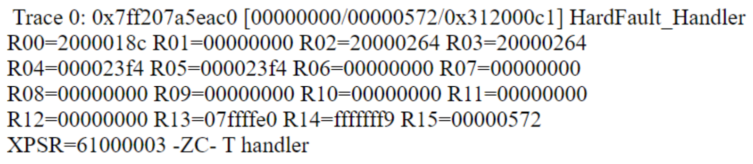
Chapter 5

Results and Discussion

This chapter discusses the implementation results and experiences of testing the embedded platform to boot up in the emulated environment. It also discusses the theory and problems behind the boot-up issues.

5.1 ARM Cortex-m3 binary analysis

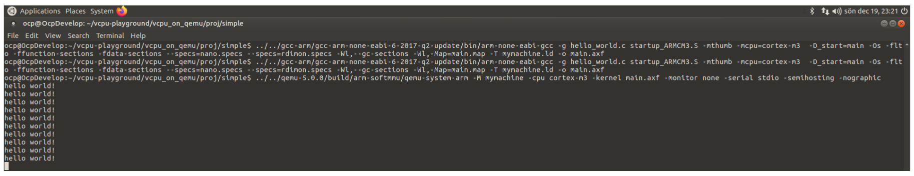
While executing a Hello World application, the QEMU encountered a HardFault error, as seen in the following Figure 5.1. The QEMU developers suggested updating the qemu version to resolve this, but this had no effect. After conducting an investigation, we discovered a solution to the problem at hand. The problem was in the ARM semihosting until the initialise_monitor_handles () method was called, at which point the problem was resolved. The research conducted for this problem revealed a basic understanding of semihosting and its applications [21].



```
 Trace 0: 0x7ff207a5eac0 [00000000/00000572/0x312000c1] HardFault_Handler
R00=2000018c R01=00000000 R02=20000264 R03=20000264
R04=000023f4 R05=000023f4 R06=00000000 R07=00000000
R08=00000000 R09=00000000 R10=00000000 R11=00000000
R12=00000000 R13=07ffffe0 R14=fffffff9 R15=00000572
XPSR=61000003 -ZC- T handler
```

Figure 5.1: HardFault handler

The screenshot in the Figure 5.2 shows an ARM cortex-m3 machine in QEMU, running a simple Hello World program. As a result, qemu-5.0.0 can now emulate a custom machine. With this knowledge, we imitated the VCPU machine in the next section.

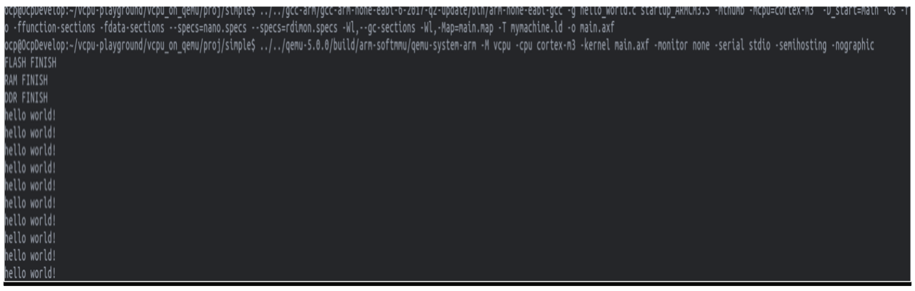

Applications Places System
sön dec 19, 23:21
ocp@OcpDevelop: ~/vcpu-playground/vcpu_on_qemu/proj/simple
File Edit View Search Terminal Help

```
ocp@OcpDevelop:~/vcpu-playground/vcpu_on_qemu/proj/simple$ ../../gcc-arm/gcc-arm-none-eabi-6-2017-q2-update/bin/arm-none-eabi-gcc -g hello_world.c startup_ARMCM3.S -mthumb -mcpu=cortex-m3  -D_start=main -Os -flt
o -ffunction-sections -fdata-sections --specs=nano.specs --specs=rdimon.specs -Wl,--gc-sections -Wl,-Map=main.map -T mymachine.ld -o main.axf
ocp@OcpDevelop:~/vcpu-playground/vcpu_on_qemu/proj/simple$ ../../gcc-arm/gcc-arm-none-eabi-6-2017-q2-update/bin/arm-none-eabi-gcc -g hello_world.c startup_ARMCM3.S -mthumb -mcpu=cortex-m3  -D_start=main -Os -flt
o -ffunction-sections -fdata-sections --specs=nano.specs --specs=rdimon.specs -Wl,--gc-sections -Wl,-Map=main.map -T mymachine.ld -o main.axf
ocp@OcpDevelop:~/vcpu-playground/vcpu_on_qemu/proj/simple$ ../../qemu-5.0.0/build/arm-softmmu/qemu-system-arm -M mymachine -cpu cortex-m3 -kernel main.axf -monitor none -serial stdio -semihosting -nographic
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
```

Figure 5.2: Hello World execution

5.2 VCPU on QEMU Anaylsis

The simple ”hello world” application was executed on the VCPU machine in a similar manner to that of mymachine, as shown in the Figure 5.3. The following step was to run the VCPU binary file (vcpu.elf) in the QEMU virtual environment. Before doing so, the vcpu.elf file was generated in the msw repository.



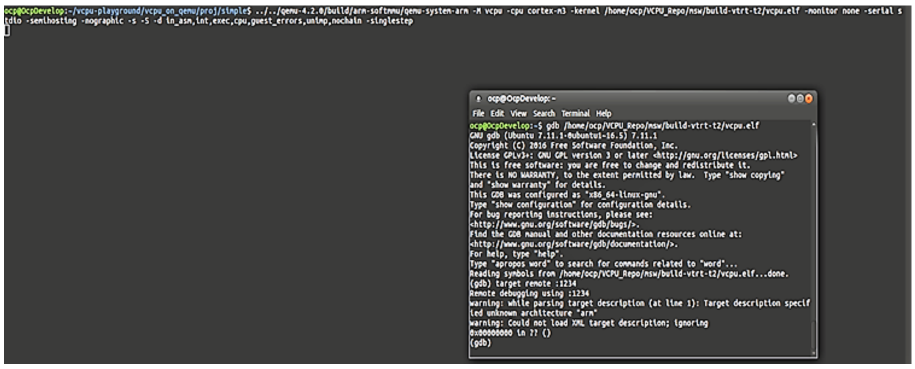
```
ocp@OcpDevelop:~/vcpu-playground/vcpu_on_qemu/proj/simple$ ../../gcc-arm/gcc-arm-none-eabi-6-2017-q2-update/bin/arm-none-eabi-gcc -g hello_world.c startup_ARMCM3.S -mthumb -mcpu=cortex-m3  -D_start=main -Os -f
o -ffunction-sections -fdata-sections --specs=nano.specs --specs=rdimon.specs -Wl,--gc-sections -Wl,-Map=main.map -T mymachine.ld -o main.axf
ocp@OcpDevelop:~/vcpu-playground/vcpu_on_qemu/proj/simple$ ../../qemu-5.0.0/build/arm-softmmu/qemu-system-arm -M vcpu -cpu cortex-m3 -kernel main.axf -monitor none -serial stdio -semihosting -nographic
FLASH FINISH
RAM FINISH
DDR FINISH
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
```

Figure 5.3: Hello World execution on VCPU machine

An unexpected hard fault exception occurred during the execution of the vcpu.elf, as shown in the Figure 5.4. To solve this issue, we used the GDB tool for debugging binary. The general setup for using gdb is shown in Figure 5.5. The option -S will freeze the CPU from execution, and the -s is shorthand to connect to a remote server ”target remote localhost:1234” [22].

```
qemu: fatal: Lockup: can't escalate 3 to HardFault (current priority -1)

R00=00000000 R01=00000000 R02=00000000 R03=00000000
R04=00000000 R05=00000000 R06=00000000 R07=00000000
R08=00000000 R09=00000000 R10=00000000 R11=00000000
R12=00000000 R13=ffffffe0 R14=fffffff9 R15=00000000
XPSR=40000003 -Z-- A handler
FPSCR: 00000000
```

Figure 5.4: HardFault exception

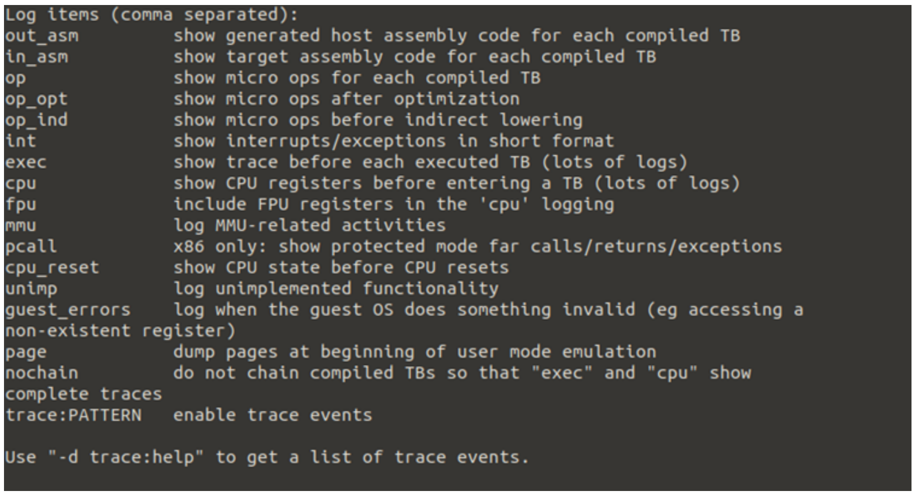


```
ocp@OcpDevelop:~/vcpu-playground/vcpu_on_qemu/proj/simple$ ../../qemu-4.2.0/build/arm-softmmu/qemu-system-arm -M vcpu -cpu cortex-m3 -kernel /home/ocp/VCPU_Repo/msw/build-vtrt-t2/vcpu.elf -monitor none -serial s
tdio -semihosting -nographic -s -S -d in_asm,int,exec,cpu,guest_errors,unimp,nochain -singlestep
```

```
ocp@OcpDevelop:~$ gdb /home/ocp/VCPU_Repo/msw/build-vtrt-t2/vcpu.elf
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /home/ocp/VCPU_Repo/msw/build-vtrt-t2/vcpu.elf...done.
(gdb) target remote :1234
Remote debugging using :1234
warning: while parsing target description (at line 1): Target description specif
ied unknown architecture "arm"
warning: Could not load XML target description; ignoring
0x00000000 in ?? ()
(gdb)
```

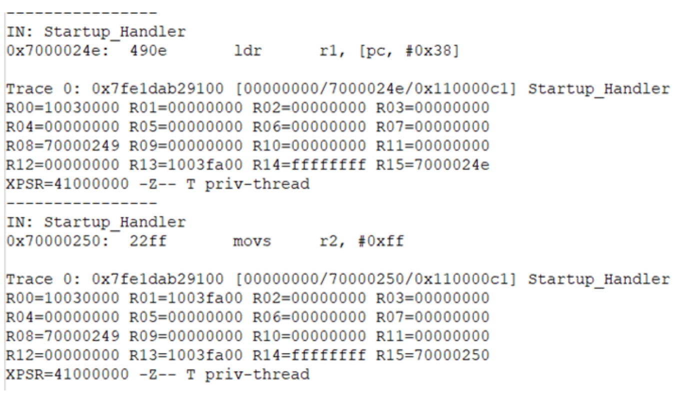
Figure 5.5: GDB setup for Debugging singlestep

In debugging, using -singlestep will help execute and trace each instruction, and the -d option has many options to choose from according to the requirements of the trace to debug, as shown in Figure 5.6. Using the -singlestep, it was determined that the issue was with our vector table not being placed at the correct address for booting the machine. By default, Qemu cortex-m3 emulation expects the vector table to be located at the address 0x0. However, when the linker script was examined, it was discovered that the vectors were inserted at the position 0x10000000. Then the vector table was relocated from 0x0 to 0x1000000 to fix this issue, and the hard fault issue was no longer present. Part of the boot is shown in Figure 5.7, where the R00 address runs until it matches the R01 address. The procedure is the same for all three blocks, the Flash, SRAM, and DDR memory, which execute the symbols within the content. The machine boots up in THUMB mode.



```
Log items (comma separated):
out_asm         show generated host assembly code for each compiled TB
in_asm          show target assembly code for each compiled TB
op              show micro ops for each compiled TB
op_opt          show micro ops after optimization
op_ind          show micro ops before indirect lowering
int             show interrupts/exceptions in short format
exec            show trace before each executed TB (lots of logs)
cpu             show CPU registers before entering a TB (lots of logs)
fpu             include FPU registers in the 'cpu' logging
mmu             log MMU-related activities
pcall           x86 only: show protected mode far calls/returns/exceptions
cpu_reset       show CPU state before CPU resets
unimp           log unimplemented functionality
guest_errors    log when the guest OS does something invalid (eg accessing a
non-existent register)
page            dump pages at beginning of user mode emulation
nochain         do not chain compiled TBs so that "exec" and "cpu" show
complete traces
trace:PATTERN   enable trace events

Use "-d trace:help" to get a list of trace events.
```

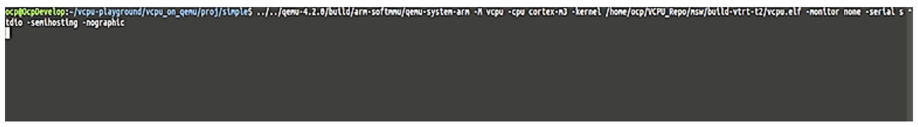
Figure 5.6: -d help options



```
----------------
IN: Startup_Handler
0x7000024e:  490e       ldr      r1, [pc, #0x38]

Trace 0: 0x7fe1dab29100 [00000000/7000024e/0x110000c1] Startup_Handler
R00=10030000 R01=00000000 R02=00000000 R03=00000000
R04=00000000 R05=00000000 R06=00000000 R07=00000000
R08=70000249 R09=00000000 R10=00000000 R11=00000000
R12=00000000 R13=1003fa00 R14=ffffffff R15=7000024e
XPSR=41000000 -Z-- T priv-thread
----------------
IN: Startup_Handler
0x70000250:  22ff       movs     r2, #0xff

Trace 0: 0x7fe1dab29100 [00000000/70000250/0x110000c1] Startup_Handler
R00=10030000 R01=1003fa00 R02=00000000 R03=00000000
R04=00000000 R05=00000000 R06=00000000 R07=00000000
R08=70000249 R09=00000000 R10=00000000 R11=00000000
R12=00000000 R13=1003fa00 R14=ffffffff R15=70000250
XPSR=41000000 -Z-- T priv-thread
```

Figure 5.7: A part of Machine Boot

After successfully placing the vector table, we tried to execute the VCPU binary file (vcpu.elf) again; however, this time, the output was a blank screen, as seen in the Figure 5.8. The reasons for the lock up will be explained in detail in the boot issues section.



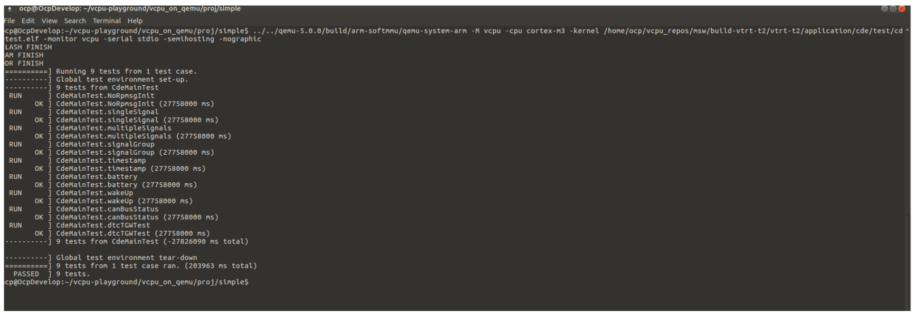
```
ocp@OcpDevelop:~/vcpu-playground/vcpu_on_qemu/proj/simple$ ../../qemu-4.2.0/build/arm-softmmu/qemu-system-arm -M vcpu -cpu cortex-m3 -kernel /home/ocp/VCPU_Repo/msw/build-vtrt-t2/vcpu.elf -monitor none -serial s
tdio -semihosting -nographic
```

Figure 5.8: Freeze of vcpu machine with the given binary image

5.3 Theory behind Boot up Issues

Theory 1: Issue in the VCPU Machine

The first possibility for the lock up might be an issue with the machine. To verify that our machine is working. We tried to implement some of the VCPU test files available in the MSW repository, such as the CDE test file, and run them on the VCPU machine in QEMU. The Figure 5.9 shows the successful implementation of the CDE test file on the VCPU machine. This indicated that our custom VCPU machine was running correctly. One drawback in our custom VCPU machine was that some of the peripherals were not implemented; this might have caused the lock up. This was a time-consuming scenario, as several protocols required independent testing rather than being implemented directly on the machine.

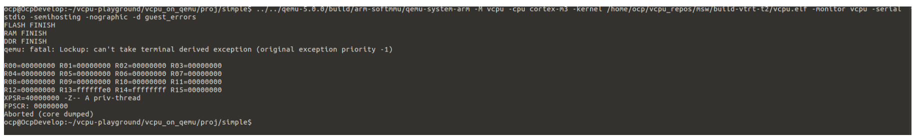


```
ocp@OcpDevelop: ~/vcpu-playground/vcpu_on_qemu/proj/simple
File  Edit  View  Search  Terminal  Help
cp@OcpDevelop:~/vcpu-playground/vcpu_on_qemu/proj/simple$ ../../qemu-5.0.0/build/arm-softmmu/qemu-system-arm -M vcpu -cpu cortex-m3 -kernel /home/ocp/vcpu_repos/msw/build-vtrt-t2/vtrt-t2/application/cde/test/cd
test.elf -monitor vcpu -serial stdio -semihosting -nographic
LASH FINISH
AM FINISH
DR FINISH
==========] Running 9 tests from 1 test case.
----------] Global test environment set-up.
----------] 9 tests from CdeMainTest
 RUN      ] CdeMainTest.NoRpmsgInit
       OK ] CdeMainTest.NoRpmsgInit (27758000 ms)
 RUN      ] CdeMainTest.singleSignal
       OK ] CdeMainTest.singleSignal (27758000 ms)
 RUN      ] CdeMainTest.multipleSignals
       OK ] CdeMainTest.multipleSignals (27758000 ms)
 RUN      ] CdeMainTest.signalGroup
       OK ] CdeMainTest.signalGroup (27758000 ms)
 RUN      ] CdeMainTest.timestamp
       OK ] CdeMainTest.timestamp (27758000 ms)
 RUN      ] CdeMainTest.battery
       OK ] CdeMainTest.battery (27758000 ms)
 RUN      ] CdeMainTest.wakeUp
       OK ] CdeMainTest.wakeUp (27758000 ms)
 RUN      ] CdeMainTest.canBusStatus
       OK ] CdeMainTest.canBusStatus (27758000 ms)
 RUN      ] CdeMainTest.dtcTGWTest
       OK ] CdeMainTest.dtcTGWTest (27758000 ms)
----------] 9 tests from CdeMainTest (-27826090 ms total)

----------] Global test environment tear-down
==========] 9 tests from 1 test case ran. (203963 ms total)
  PASSED  ] 9 tests.
cp@OcpDevelop:~/vcpu-playground/vcpu_on_qemu/proj/simple$
```

Figure 5.9: CDE test result

Theory 2: Issue in the vector table placement

The linker script of vcpu.elf expects the vector table to be located at address 0x10000000 in the QEMU. In general, at reset, the Arm processor will boot from the reset vector location at the address 0x0. We created a simple scatter file to verify this issue where the vector table and OS_Code were placed at the reset vector address and tested on the VCPU machine. The test file was successfully implemented, as seen in Figure 5.9. However, when we relocated the vector table and OS_Code at 0x10000000 in the scatter file. The Qemu throws the lock-up error, as seen in the Figure 5.10. This means that when the controller is reset, it begins reading data from the base address 0x00000000 [23][24].



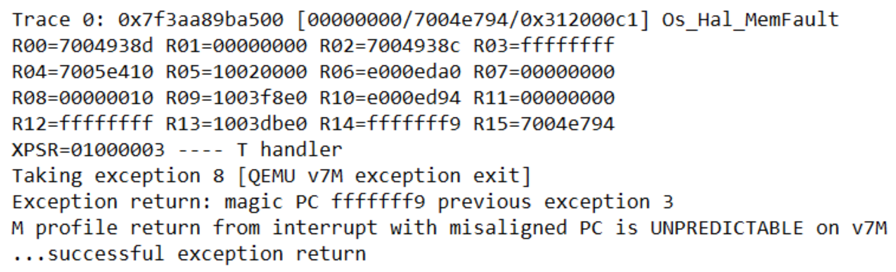
```
ocp@OcpDevelop:~/vcpu-playground/vcpu_on_qemu/proj/simple$ ../../qemu-5.0.0/build/arm-softmmu/qemu-system-arm -M vcpu -cpu cortex-m3 -kernel /home/ocp/vcpu_repos/msw/build-vtrt-t2/vcpu.elf -monitor vcpu -serial
stdio -semihosting -nographic -d guest_errors
FLASH FINISH
RAM FINISH
DDR FINISH
qemu: fatal: Lockup: can't take terminal derived exception (original exception priority -1)

R00=00000000 R01=00000000 R02=00000000 R03=00000000
R04=00000000 R05=00000000 R06=00000000 R07=00000000
R08=00000000 R09=00000000 R10=00000000 R11=00000000
R12=00000000 R13=ffffffe0 R14=ffffffff R15=00000000
XPSR=40000000 -Z-- A priv-thread
FPSCR: 00000000
Aborted (core dumped)
ocp@OcpDevelop:~/vcpu-playground/vcpu_on_qemu/proj/simple$
```

Figure 5.10: VCPU Lock up

In this case, the PC is zero, and the Thumb bit is not set. It signifies that our guest code did something that prompted the CPU to attempt to take an exception, but because our ELF file did not include an exception vector table, the vector table entry for this exception was zero. That means that the CPU will attempt to execute from address 0 with the Thumb bit cleared, resulting in an instantaneous UsageFault exception, which will typically result in the exception-within-an-exception Lock-up case shown in Figure 5.10 [26].

Theory 3: Issue in running complete binary image

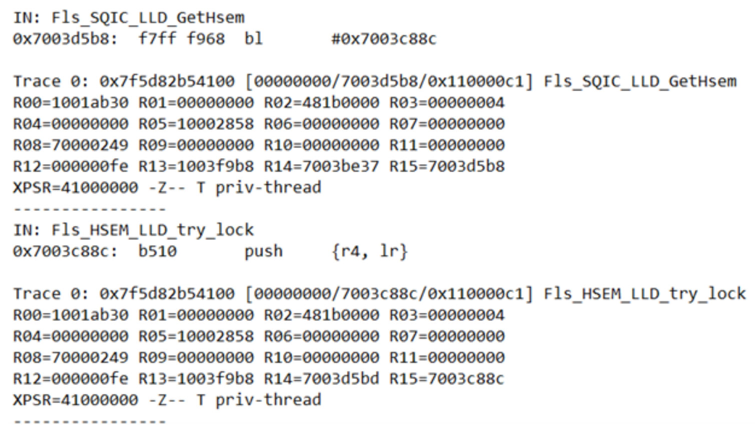
While running the complete binary image in QEMU [26], after the initial boot, as seen in Figure 5.7, the machine goes into an infinite Fls_init() function loop. Examining this situation, commenting on the function, and trying to reboot the machine was one way to check whether the machine could boot without that particular function. In return, we got an Os_Hal_MemFault, as seen in the Figure 5.11. The reason for this error might be that the PC value on the stack that we are trying to return is incorrect. Furthermore, Memory Protection Unit (MPU) was wrongly programmed. This process allowed us to look into the HSEM module of our machine.



```
Trace 0: 0x7f3aa89ba500 [00000000/7004e794/0x312000c1] Os_Hal_MemFault
R00=7004938d R01=00000000 R02=7004938c R03=ffffffff
R04=7005e410 R05=10020000 R06=e000eda0 R07=00000000
R08=00000010 R09=1003f8e0 R10=e000ed94 R11=00000000
R12=ffffffff R13=1003dbe0 R14=fffffff9 R15=7004e794
XPSR=01000003 ---- T handler
Taking exception 8 [QEMU v7M exception exit]
Exception return: magic PC fffffff9 previous exception 3
M profile return from interrupt with misaligned PC is UNPREDICTABLE on v7M
...successful exception return
```

Figure 5.11: Memory Fault

Hardware Semaphore (HSEM) is our embedded platform's main feature, which manages the access permissions and synchronization of the resources shared between multiple processes running on the same CPU or different CPUs [25]. With this module not implemented in the machine, it goes into an infinite loop within the function, fetching the correct addresses to exit and executing the process that it cannot get hold off. A part of this execution is represented in Figure 5.12; this issue may have caused the lock-up.



```
IN: Fls_SQIC_LLD_GetHsem
0x7003d5b8:  f7ff f968  bl       #0x7003c88c

Trace 0: 0x7f5d82b54100 [00000000/7003d5b8/0x110000c1] Fls_SQIC_LLD_GetHsem
R00=1001ab30 R01=00000000 R02=481b0000 R03=00000004
R04=00000000 R05=10002858 R06=00000000 R07=00000000
R08=70000249 R09=00000000 R10=00000000 R11=00000000
R12=000000fe R13=1003f9b8 R14=7003be37 R15=7003d5b8
XPSR=41000000 -Z-- T priv-thread
----------------
IN: Fls_HSEM_LLD_try_lock
0x7003c88c:  b510       push     {r4, lr}

Trace 0: 0x7f5d82b54100 [00000000/7003c88c/0x110000c1] Fls_HSEM_LLD_try_lock
R00=1001ab30 R01=00000000 R02=481b0000 R03=00000004
R04=00000000 R05=10002858 R06=00000000 R07=00000000
R08=70000249 R09=00000000 R10=00000000 R11=00000000
R12=000000fe R13=1003f9b8 R14=7003d5bd R15=7003c88c
XPSR=41000000 -Z-- T priv-thread
----------------
```

Figure 5.12: Infinite Loop for HSEM lock

Chapter 6

Conclusion

The purpose of this chapter is to present the reader with our perspectives on challenges we have encountered and to assess whether we were successful in achieving our project objectives in the form of the final product of our project. The chapter will conclude with our opinions on future development, including what ways we believe this project could be taken in the future.

By virtualizing systems, emulation broadens the range of options for debugging and testing embedded devices, allowing developers to concentrate on software development. In many ways, the QEMU has assisted embedded system developers in increasing their productivity by supporting a variety of architectures.

QEMU includes a remote GDB interface that enables users to directly monitor and influence the execution of programs running on the virtual CPU. The debugger also provides access to the RAM and CPU registers, eliminating a separate microcontroller debugger. Even after their continuous integration and releases on new and upgraded QEMU, the platform has remained untapped since it is difficult to comprehend for new developers and requires an expert to have hands-on experience.

Despite all the research and analysis, we could not pinpoint a specific solution to the infinite loop fault that was occurring in our QEMU. It has been challenging to comprehend the entire workflow of the Main software (MSW). Limited resources on QEMU have made our task more difficult to complete in the limited time available for this project. After all of our efforts, the thesis aim remained unattained.

The work and research completed to support the thesis can be utilized as a foundation for future study and extensions—discovered limitations and possibilities for the emulated platform. The main use of an emulator is to help in the development and the transition between platforms. Even though the results were mixed outcomes, future works use these issues and implement them with more suitable tools. The aspects that would have helped in results are:

1. By the time we started the thesis, the QEMU's latest version was 6.0.0, as we were using a pretty old QEMU-4.2.0. The forum is active, but it was of little

use because the API and build had undergone significant modifications. To use the latest version, it needs the support of the least Ubuntu 18.04 LTS or higher as running two VMs made the PC slow down and even crash sometimes.

2. The method tried in the thesis was to emulate both the CPUs separately and combine them later, which was one of the issues. As both the CPUs are dependent on being a complete embedded platform. For example, Xilinx has a "ZynqMP ZCU102" board in QEMU, with 4xA53s and 2xR5Fs. The example can be used as a model for other people's future efforts. This might potentially be one of the reasons why our machine is stuck in an infinite loop, as addressed in Chapter 5.

Bibliography

[1] T. Cloud, Technical Notes on The EEC-IV MCU, Nov. 1997.

[2] “Engine control unit,” Hyundai Wiki, Feb. 2016. [Online]. Available: http://hyundai.wikia.com/wiki/Engine_Control_Unit

[3] Koltsidas Athanasios and Oscar Peterson. Virtual AUTOSAR Environment on Linux Evaluation Study on Performance Gains from Running ECU Applications on POSIX Threads Master’s Thesis in Embedded Electronic System Design.

[4] Education, I., 2022. virtualization-a-complete-guide. [online] Ibm.com. Available at: https://www.ibm.com/cloud/learn/virtualization-a-complete-guide.

[5] A. Holt and C.-Y. Huang, Embedded operating systems. Springer, 2014.

[6] Minnie.tuhs.org. 2022. Introduction to Systems Architecture. [online] Available at: https://minnie.tuhs.org/CompArch/Lectures/week01.html.

[7] Fabrice Bellard. Qemu home page. URL: http://wiki.qemu.org/Main_-Page.

[8] “QEMU, a Fast And Portable Dynamic Translator_Shawn-CSDN.” QEMU, a Fast And Portable Dynamic Translator_Shawn-CSDN, Blog.csdn.net, 2 March. 2001, https://blog.csdn.net/yearn520/article/details/6600034.

[9] Gupea.ub.gu.se. 2022. [online] Available at: https://gupea.ub.gu.se/bitstream /2077/66920/1/gupea_2077_66920_1.pdf.

[10] “Getting Started Developers.” https://wiki.qemu.org/index.php/ Documentation/GettingStartedDevelopersGetting_to_know_the_code, 2019.

[11] Autosar: Layered software architecture. [Online]. Available: http:// www.autosar.org/fileadmin/files/releases/4-2/software-architecture/ general/ auxiliary/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf.

[12] (2014) AUTOSAR layered software architecture. [Online]. Available: http: //www.autosar.org/fileadmin/files/releases/4-2/software-architecture/

[13] Arm Developer Available at: https://developer.arm.com/documentation /ddi0337/h.

[14] ARM. Elf for the arm architecture. Technical report, ARM, 2015. http:// infocenter.arm. com/help/topic/com.arm.doc.ihi0044f /IHI0044F_aaelf.pdf

[15] “Readelf(1) - Linux Manual Page.” Readelf(1) - Linux Manual Page, Man7.org, https://man7.org/linux/man-pages/man1/readelf.1.html.

[16] ARM-software. “CMSIS_5/Device/ARM/ARMCM3/Source/GCC At Develop · ARM-software/CMSIS_5.” GitHub, Github.com, https://github.com/ARM-software/CMSIS_5/tree/develop/Device/ARM/ARMCM3/Source/GCC.

[17] “The Memory API mdash; QEMU 6.2.50 Documentation.” The Memory API mdash; QEMU 6.2.50 Documentation, Qemu.readthedocs.io, https://qemu.readthedocs.io/en/latest/devel/memory.html.

[18] Eduardo Habkost. habkost.net. Technical report, [Qemu-devel] documentation on qemu, 2016. https://habkost.net/posts/2016/11/incomplete-list-of-qemu-apis.html.

[19] Anthony Liguori. qom: add the base object class(v2). https://github.com/ qemu/qemu/ commit/2f28d2ff9dce3c404b36e90e64541a4d48daf0ca.

[20] Anthony Liguori Gerd Hoffmann. Qemuopts: framework for storing and parsing options. https: //github.com/ qemu/qemu/ commit/e27c88fe9eb26648e4 fb282cb3761c41f06ff18a

[21] Messaoudi, Amine El. “Introduction To ARM Semihosting — Interrupt.” Interrupt, Interrupt.memfault.com, 16 February. 2021, https:// interrupt.memfault .com /blog /arm-semihosting.

[22] “GDB Usage mdash; QEMU Documentation.” GDB Usage and mdash; QEMU Documentation, Qemu-project.gitlab.io, https://qemu-project.gitlab.io /qemu/system/gdb.html.

[23] Ijsr.net. 2022. [online] Available at: https://www.ijsr.net /archive /v8i10/ ART20202270.pdf.

[24] And ARM Germany GmbH, ARM Ltd. “Compiler Getting Started Guide: Vector Table for M-profile Architectures.” Compiler Getting Started Guide: Vector Table for M-profile Architectures, Www.keil.com, https://www.keil.com /support/man/docs/armclang_intro /armclang_intro_ccg1505913593658.htm.

[25] St.com. 2022. [online] Available at: https://www.st.com/content/ccc/ resource/training/technical/product_training/ group0/73/c4/ce/99/f6/3f/43/5e/ STM32WB-System-Hardware-Semaphore-HSEM/ files/ STM32WB- System - Hardware-Semaphore-HSEM.pdf/ jcr:content/ translations/ en. STM32WB-System- Hardware-Semaphore-HSEM.pdf.

[26] “Qemu: Fatal: Lockup: Can’t Escalate 3 To HardFault (current Priority -1).” Qemu: Fatal: Lockup: Can’t Escalate 3 To HardFault (current Priority -1), Www.mail-archive.com, 16 September. 2021, https://www.mail-archive.com/qemu-discuss@nongnu.org/msg06931.html.

Printed by Tryckeriet i E-huset, Lund 2023

Series of Master’s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2023-912
http://www.eit.lth.se