University of Minho
Engineering School

Nelson Pinheiro Duarte Naia

Real-Time Linux and Hardware Accelerated Systems on QEMU

Master's Thesis
Master's Degree in Industrial Electronics Engineering and Computers

Carried out under guidance of
PhD Adriano José da Conceição Tavares

October, 2015

Declaração

Nome:
Nelson Pinheiro Duarte Naia

Endereço Eletrónico: npnaia@gmail.com

Telefone/Telemóvel: 91 057 96 72

Número do Bilhete de Identidade: 13783902

Título da Dissertação: Real-Time Linux and Hardware Accelerated Systems on QEMU

Orientador: Professor Doutor Adriano José da Conceição Tavares

Designação do Mestrado:
Mestrado Integrado em Engenharia Eletrónica Industrial e Computadores

Universidade do Minho, ____/_____/________
Assinatura:____

Acknowledgments

”I would like to thank my supervisor, PhD Adriano Tavares, for all the knowledge insight throughout the project. I would especially like to thank engineer Vítor Silva, for his friendship and all the help and time spent providing guidance and sharing his knowledge and brilliant vision in engineering. I would also like to thank all professors and members of the Embedded Systems Research Group, for all the support and resources made available throughout this dissertation. A special thanks should also go to Embedded Systems Research Group students Marcelo Sousa, César Monteiro and João Gonçalves, that such as myself are graduating, and accompanied me through this journey, being workspace companions, providing peer reviews and above all tight friendships. Finally I would like to thank my parents and close family, for all the moral, emotional and financial support throughout all this engineering course. Without them, I wouldn’t be able to do it at all. To everyone, my sincerest gratitude.”

Abstract

Software application acceleration, using parallelization techniques and dedicated hardware components, is often an optimization compromise in a cost-benefit relationship during the migration of software processes to hardware **I**ntellectual **P**roperty (IP) dedicated cores or accelerators. In real-time applications extra care is needed when dealing with these issues, so that the real-time requirements of the application are not compromised. An isolated validation, as far as application domains are concerned, does not guarantee integral system functionality. Using an integrated co-simulation environment, chances of early system problem detection before moving to the physical implementation phase are improved. By adopting a design flow aided by co-simulation, not only is the development process sped up, but also resource independent, since the system can be developed in its entirety in a host platform without being bound to a physical target platform.

This dissertation aims to adopt a methodology of hardware-software co-design aided by co-simulation and extend embedded system simulation techniques to hardware IP co-simulation and integral validation, improving the design process of hardware accelerated embedded systems in their various development phases. Using **Q**uick **EMU**lator (QEMU) as a tool for emulating embedded software platforms in a Linux-based environment, modifications were idealized and developed to enable QEMU to extend its embedded software platform emulating capabilities for custom hardware co-processor development purposes. Two QEMU extensions were developed, enabling easy integration of behavioral devices and co-simulation with external **R**egister-**T**ransfer **L**evel (RTL) models in QEMU's target platforms. A Verilog PLI library was also developed to allow Verilog simulators that support PLI to perform co-simulation with QEMU. To demonstrate the capabilities of following a hardware-software embedded co-design using the developed simulation environment, a demonstration application scenario was developed following a design flow that takes advantage of said simulation environment possibilities.

Resumo

A aceleração de aplicações de *software*, utilizando técnicas de paralelização e componentes de *hardware* dedicados, é frequentemente um compromisso de optimização numa relação de custo-benefício durante a migração de processos de *software* para aceleradores ou *cores hardware* IP dedicados. Em aplicações *real-time*, cuidados extra são necessários ao lidar com estas problemáticas, de forma a que os requisitos *real-time* da aplicação não sejam comprometidos. Uma validação isolada, no que respeitam os vários domínios de aplicação, não garante uma funcionalidade integral do sistema. Utilizando um ambiente de co-simulação integrado, falhas no sistema podem ser detectadas numa fase inicial do projecto, antes de ser atingida uma fase de implementação física. Ao adoptar um *design flow* auxiliado por co-simulação, não só é o processo de desenvolvimento agilizado, mas também isento de dependências a nível da plataforma *target*, uma vez que o sistema pode ser desenvolvido inteiramente na plataforma *host* sem estar dependente dos recursos físicos associados uma plataforma *target*. Esta dissertação surge no âmbito da validação de uma metodologia de *hardware-software co-design* auxiliada por co-simulação, no extender de técnicas de simulação de sistemas embebidos, com ou sem aceleração de processos em hardware RTL, e na validação integral, aperfeiçoando o processo de *design* dos mesmos ao longo das várias fases de desenvolvimento. Utilizando o QEMU como ferramenta para emulação de ambientes baseados em Linux para plataformas de CPU+FPGA, alterações foram idealizadas e desenvolvidas para permitir extender as capacidades de emulação das mesmas no QEMU, para propósitos de desenvolvimento de aceleradores em *hardware* customizados, possibilitando a integração de *devices* comportamentais e co-simulação com modelos RTL externos nas plataformas *target* do QEMU. Para demonstrar as capacidades de seguir um *co-design* de *hardware-software* embebido utilizando o ambiente de simulação desenvolvido, um cenário de aplicação demonstrador foi desenvolvido seguindo um *design flow* que toma partido das possibilidades do referido ambiente de simulação.

Contents

List of Figures

List of Tables

Acronyms List

ACC	**ACC**ess
API	**A**pplication **P**rogram **I**nterface
ARM	**A**dvanced **R**ISC **M**achine
ASIC	**A**pplication-**S**pecific **I**ntegrated **C**ircuit
ASIP	**A**pplication-**S**pecific **I**nstruction set **P**rocessor
AXI	**A**dvanced e**X**tensible **I**nterface
BSP	**B**oard **S**upport **P**ackage
CLB	**C**onfigurable **L**ogic **B**lock
CPU	**C**entral **P**roccessing **U**nit
DCR	**D**evice **C**ontrol **R**egister
DMA	**D**irect **M**emory **A**ccess
DPI	**D**irect **P**rogramming **I**nterface
DSP	**D**igital **S**ignal **P**rocessor
DTB	**D**evice **T**ree **B**lob
DTC	**D**evice **T**ree **C**ompiler
DTS	**D**evice **T**ree **S**ource
ESRG	**E**mbedded **S**ystems **R**esearch **G**roup
FDT	**F**lat **D**evice **T**ree
FMI	**F**unctional **M**ock-up **I**nterface

FMU **F**unctional **M**ock-up **U**nit

FPGA **F**ield **P**rogrammable **G**ate **A**rray

FSL **F**ast **S**implex **L**ink

GPIO **G**eneral **P**urpose **I**nput/**O**utput

GCC **G**NU **C**ompiler **C**ollection

GPL **G**eneral **P**ublic **L**icense

GUI Graphical User Interface

HDL **H**ardware **D**escription **L**anguage

HLS **H**igh **L**evel **S**ynthesis

IDE **I**ntegrated **D**evelopment **E**nvironment

I/O **I**nput/**O**utput

IP **I**ntellectual **P**roperty

IRQ **I**nterrupt **R**e**Q**uest

JTAG **J**oint **T**est **A**ction **G**roup

LAB **L**ogic **A**rray **B**lock

LCD **L**iquid-**C**rystal **D**isplay

LED **L**ight **E**mitting **D**iode

LUT **L**ook**U**p **T**ables

MMC **M**ulti**M**edia**C**ard

MPI **M**essage **P**assing **I**nterface

OEM **O**riginal **E**quipement **M**anufacturer

OS **O**perating **S**ystem

PC **P**rogram **C**ounter

PCIe **P**eripheral **C**omponent **I**nterconnect **e**xpress

PLB **P**rocessor **L**ocal **B**us

PLI **P**rogramming **L**anguage **I**nterface

POSIX **P**ortable **O**perating **S**ystem **I**nterface

QDEV **Q**EMU **DEV**ice

QEMU **Q**uick **EMU**lator

QOM **Q**EMU **O**bject **M**odel

RAM **R**andom **A**ccess **M**emory

RC **R**elease **C**andidate

RISC **R**educed **I**nstruction **S**et **C**omputer

RTL **R**egister-**T**ransfer **L**evel

RTOS **R**eal-**T**ime **O**perating **S**ystem

SD **S**ecure **D**igital

SDK **S**oftware **D**evelopment **K**it

SoC **S**ystem-**o**n-**C**hip

TCG **T**iny **C**ode **G**enerator

TCP/IP **T**ransmission **C**ontrol **P**rotocol/**I**nternet **P**rotocol

TF **T**ask/**F**unction

UART **U**niversal **A**synchronous **R**eceiver **T**ransmitter

UML **U**nified **M**odeling **L**anguage

VFS **V**irtual **F**ile **S**ystem

VPI **V**erilog **P**rocedural **I**nterface

XML e**X**tensible **M**arkup **L**anguague

Chapter 1

Introduction

This chapter has an introductory purpose, presenting a brief contextualization of this dissertation's scope, as well as defining its motivation and main objectives. The main contributions of this project to the scientific community are also presented, with a clarification of the document's structure closing the chapter.

1.1 Contextualization

We live in a world with technology growing and evolving day by day, with a high tendency for digital systems to take over everyday common tasks. With the rapid growth and intrusion of technology in human life, there is a also a growing need for better digital system solutions. Most digital systems in our every day lives are embedded systems.

An embedded system is a computational system that was designed for a specific purpose, being tailored for a particular set of tasks. Embedded systems are usually integrant part of a larger system, performing lower-level tasks, often interfacing with the physical world through control or monitoring systems, performing analogue data acquisition and driving circuitry that acts on the physical world, such as switches or motors. Embedded systems are everywhere, and can be thought of every digital system that's expected to hide a computational system inside, from MP3 players to printers to most domestic electric appliances. In fact, almost everything that is not a desktop computational system may be thought of as an embedded system.

Embedded development is usually made on available embedded development boards, that are frequently shipped with a development toolchain, comprising such tools as compilers, linkers, debuggers and even other tools that enable software development for the embedded platforms. Embedded software is most commonly developed in a host desktop platform, and later debugged and deployed in the target embedded platform. Figure 1.1 represents a traditional embedded development setup.

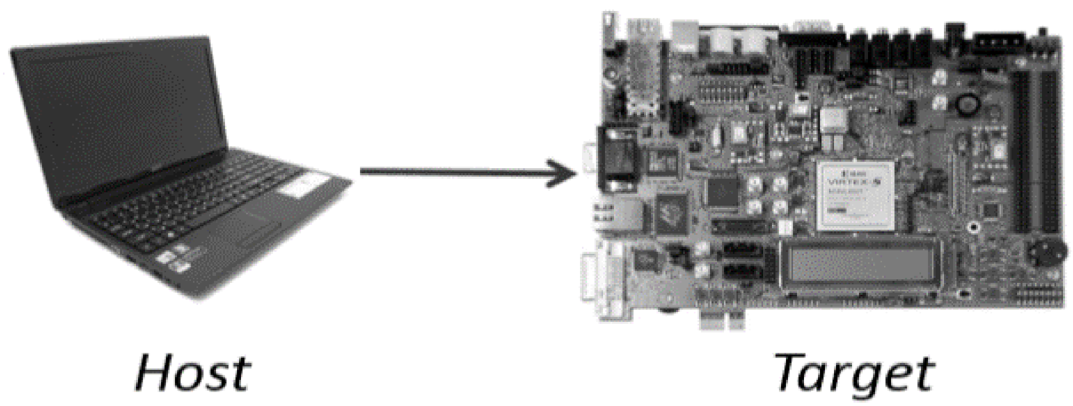


Figure 1.1: Embedded development setup

Unfortunately, many of the performance enhancing features in personal computer architecture, such as instruction pipelining and the use of caches in the memory system, which make them so fast also make it difficult to predict their performance at all times accurately. This is a problem in real-time embedded systems, which require applications with deterministic behavior.

Real-time embedded systems are systems with critical tasks that must meet deadlines, as far as execution is concerned. Some systems, such as anti-lock brakes in a car, have such real-time requirements that when not met may fail utterly and completely, and even be harmful to the user or its environment. For this reason, embedded software development is radically different from desktop software development, requiring different designs and a completely different set of skills from the developer. A modern technique to enhance real-time requirements satisfaction is the computational offload through dedicated hardware co-processors. This technique makes use of the inherent parallel nature of hardware, with custom hardware co-processor integration and the use of **S**ystem-**o**n-**C**hip (SoC)-based architectures made possible with recent advances in **F**ield **P**rogrammable **G**ate **A**rray (FPGA) technology.

Software application acceleration, using parallelization techniques and dedicated hardware components, is often an optimization compromise in a cost-benefit relationship, during the migration of software processes to hardware IP dedicated cores or accelerators. In real-time applications extra care is needed in dealing with these issues, so that the real-time requirements of the application are not compromised. Through linear programming mathematical tools, it is possible to establish a good compromise, minimizing the desired system metric, like power consumption, temporal latency or silicon area.

In order to characterize the temporal evolution on a system level, an integrated co-simulation environment where all metric costs are contemplated would be ideal. Such an environment would not only be useful by providing measurements of the system metrics, so the linear mathematical tools can aid in design optimizations, but also in system modeling. Figure 1.2 presents a diagram of a co-simulation environment overview.

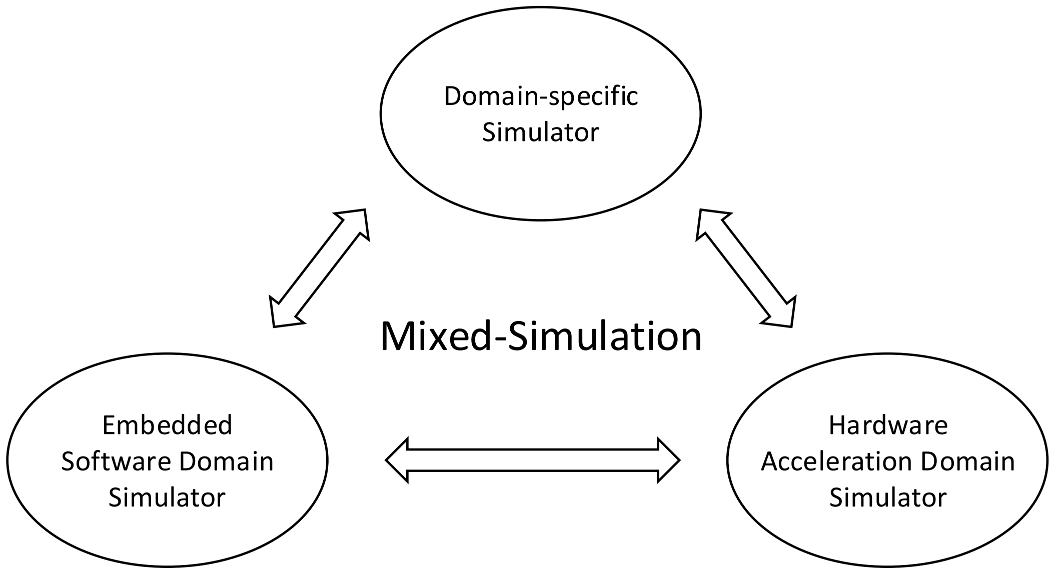

Domain-specific Simulator
Mixed-Simulation
Embedded Software Domain Simulator
Hardware Acceleration Domain Simulator

Figure 1.2: Co-simulation overview

Usually, hardware-software co-design is a long process of iterative nature. If the system is modeled and simulated on separate application domains and later implemented and debugged physically, the development time can grow exponentially. An isolated validation, as far as application domains are concerned, does not guarantee integral system functionality, and with an integrated co-simulation environment, system problems can be detected early before moving to the physical implementation phase.

By adopting a design flow aided by co-simulation, the the development process is sped up, and also resource independent, since the system can be developed in its entirety in a host platform without being bound to a physical target platform. This kind of development has some obvious benefits, as the system can be modeled first, profiled and analyzed later according to certain metrics, and only afterwards is a target platform selected. Also, design teams with limited available prototypes are not slowed down by limited platform resources.

1.2 Motivation and Objectives

Power electronic devices often need fast-responding and deterministic controllers, requiring systems with hard to meet real-time constraints. A modern technique to mitigate latencies and non-determinism intrinsic to operating systems in the context of real-time embedded systems is to use hardware acceleration for computational offloading. Hardware accelerated embedded systems development is usually a long iterative process with conventional design flows being slow and with difficult system integration on the various application domains. Local caches are commonly used to stimulate and validate each application domain in controllers of this nature, which implies lengthy development cycles in most cases.

Currently, there is no tool available on the market that allows to simulate domain crossing interactions, and support a design flow based on co-simulation. As such, one of the main objectives of this project is to develop a simulation environment that supports domain crossing validation. With the concept of co-simulation being tackled in the **E**mbedded **S**ystems **R**esearch **G**roup (ESRG) of *Universidade do Minho*, an idealization on how a simulation framework for hardware accelerated embedded systems could greatly improve future project development processes began to surface. This dissertation is a product of that vision, aiming to be the first pragmatic step towards such a framework.

1.3 Contributions

This focus of this dissertation is placed on embedded platform development with the help QEMU. QEMU, as an open-source project, is subject to any kind of modifications and changes by anyone, with development being made by a large community, including such contributions as Xilinx itself, a major company of FPGA chips and SoC-based solutions. As such, this document tries to emphasize what advances could be made in this regard, and tries to introduce new concepts that hopefully improve development of hardware accelerated embedded applications in the long run.

As such, a scientific article is also planned for publication to complement this dissertation and reach the scientific community.

1.4 Dissertation Structure

This document presents the development of simulation tools extensions and libraries that allow QEMU to be used as a full-system emulator in a real-time hardware accelerated application context. Its content was divided in six chapters that are going to be briefly presented.

The second chapter introduces the reader in the most common tools and methodologies of hardware accelerated application development for embedded systems. It presents what technologies and tools are available for this kind of development, as well as the commonly followed design flows, focusing on what choices can be made to adopt a simulation based environment, backing up eventual development choices made in this project. Special attention is given to QEMU, exposing its features and possibilities, namely in embedded platform emulation. Verilog simulation interface standards are also approached, specifically Verilog PLI. Finally, co-simulation extension standards are mentioned, bringing up their relevance in the context of this dissertation.

The third chapter presents development work that was done in terms of extending simulation environments capabilities, presenting the developed extensions. Developed content presentation is functionality oriented, providing an insight on how development was made, as well as on how to make use of these mechanisms. Examples are given whenever possible, being mentioned as work is presented to help the reader in understanding how to use the provided interfaces as a developer.

The fourth chapter presents a case of study, providing a practical example on how the extended simulation environment's features work, highlighting what advantages does this kind of development bring in embedded hardware accelerated application development. The chosen case of study was in the power electronics domain, being an instantaneous power monitor based on the pq theory.

The document's closing chapter discusses results and conclusions concerning developed work. Future work is also mentioned, clarifying what can be further done in terms of giving this work continuity and further improve it to provide a full hardware accelerated embedded system validation framework.

Chapter 2

State of the Art

In this chapter, background is provided regarding the technologies and methodologies within the scope of this dissertation. Embedded systems are briefly contextualized, with a strong focus on topics like embedded systems with **O**perating **S**ystems (OSs) and real-time embedded constraints. Hardware acceleration and its common design flows and simulation approaches to embedded design are also discussed, with deep details of two important domains in the developed work, the QEMU and the Verilog Programming Language Interface. Finally, the Functional Mock-up Interface is an important standard for co-simulation and model interchange widely adopted in the industry's most commonly used simulators, and as an important reference for this project's system design, it is also covered.

2.1 Embedded Systems

Currently, embedded systems can be found everywhere in our every day lives, ranging from consumer electronics and electrical appliances to office automation, industrial automation, military defense systems, transportation systems, aerospace systems, medical systems and so forth. Out of more than six billion microprocessors produced in 2002, more than 98% were microprocessors for embedded purposes, rather than personal computer purposes (Turley, 2002). These numbers may very well have even increased with the smartphone and tablet business boom of recent years, and subsequent rise of popularity of the ARM architecture, resulting in ever greater demand in the embedded market.

2.1.1 Definition

An embedded system is an applied computer system, as distinguished from other types of computer systems such as personal computers or supercomputers. However, the definition of ”embedded system” is fluid and difficult to pin down, as it constantly evolves with advances in technology and dramatic decreases in the cost of implementing various hardware and software components. In recent years, the field has outgrown many of its traditional descriptions (Noergaard, 2013). Some of its traditional descriptions may include:

- Embedded systems are computational systems with limited hardware resources, such as memory or processing performance;
- Embedded systems are simpler computational systems, with less software abstraction, with no operative system, or a limited one;
- Embedded systems are computational systems where software and hardware are designed designed for a specific task, often running applications with limited or no user interface at all.
- Embedded systems are computational systems that are embedded as part of a complete device often including hardware and mechanical parts;
- Embedded systems are systems which hide computational systems inside, but are not thought of as computers;

While most of these rather outdated definitions are not necessarily false, some of them do not cover all of the modern embedded systems. For instance, a modern smartphone, generally classified as an embedded system, does have a complex OS, fairly acceptable hardware resources and is not designed for a single specific task. Summarily, a good way of defining an embedded system is considering as an embedded system any system that includes a computer, but is not designed to be a general purpose desktop personal computer.

Generally, modern day embedded systems can be thought of as low-end, middle-end and high-end embedded systems regarding their complexity, with processing units ranging from simple 4-bit processors to complex multi-core 32 and 64 bit architectures, and software ranging from bare-metal applications to complex OSs running multiple applications. Some examples of embedded systems are:

- Low-end:

Digital thermostats, automatic vending machines, electrical toothbrushes, traffic lights, generic remote controls, washing machines.

- Middle-end:

  MP3 players, DVD players, TVs, cellphones, routers, printers, digital cameras.

- High-end:

  Smartphones, smart TVs, tablets, cars, trains, airplanes.

2.1.2 Embedded Processors

The main component of an embedded system is the computational processing unit. Without it, it wouldn't even qualify as an embedded system, since an embedded system must be a computational system. This does not mean, however that this processing unit must me a programmable processor. There are several types of processors, and unlike desktop computers, a processor with lots of instructions and capabilities does not mean that it is best suited for an embedded purpose. Figure 2.1 presents the different kinds of processors, in relation to their flexibility and performance and consumption efficiency.

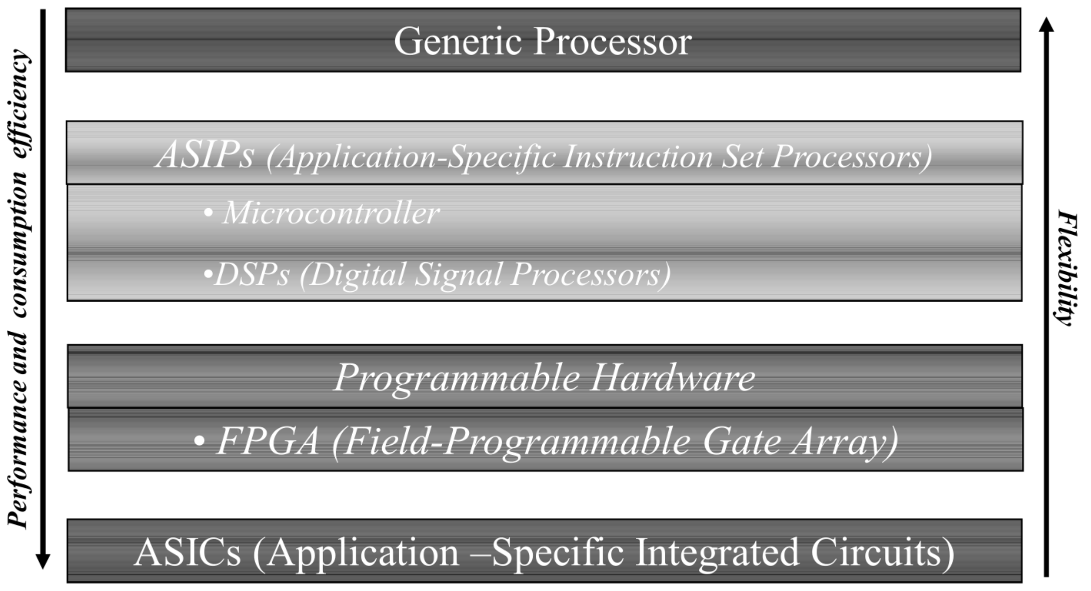

Performance and consumption efficiency
Generic Processor
ASIPs (Application-Specific Instruction Set Processors)
• Microcontroller
•DSPs (Digital Signal Processors)
Programmable Hardware
• FPGA (Field-Programmable Gate Array)
ASICs (Application –Specific Integrated Circuits)
Flexibility

Figure 2.1: Processor types

At the top, general purpose processors, also known as microprocessors, are the most flexible type of all processing units. These are the **C**entral **P**roccessing **U**nit (CPU) used in personal computers, and are the least efficient of them all, as far as energy consumption-computation performance ratio is concerned. The flexibility they provide instruction-wise is greatly suited for personal computers, but unnecessary in most embedded scenarios. For this reason, and due to their high power consumption they are not a very well suited to embedded systems, being often overlooked and quite rightly so.

At the other end of the spectrum are **A**pplication-**S**pecific **I**ntegrated **C**ircuit (ASIC). These processors are the least flexible of all, and are hardwired to carry out a specific task, not being programmable in any way. With very high time-to-market and development costs, it is only used if maximum efficiency and performance is necessary and a large number of system copies can be sold, relying on an economy of scale.

FPGA provide a low-cost solution to ASIC, albeit a more power hungry one. These processors are more flexible than ASIC, as their internal hardware connections may be reprogrammed. As a low-cost alternative, these are frequently used to prototype and debug ASIC designs during the development process, as the manufacturing costs of creating a new ASIC mask design are what make the technology expensive.

Lastly, **A**pplication-**S**pecific **I**nstruction set **P**rocessor (ASIP) are flexible processors which are capable of running software instructions, but are not as generic as microprocessors, being a compromise between single-purpose processors such as FPGA or ASIC and microprocessors. These processors are particularly optimized for a particular class of applications and often have hardwired components additional to the actual processing unit, such as memory blocks or other peripherals. For these reasons, they are often the ideal choice for embedded systems, and comprise the most frequent choice in embedded system design. Some examples of ASIP are:

- Microcontrollers;

  Microcontrollers are chips that integrate microprocessors, usually 8 to 16 bits or 32 bits, and memory blocks in the same package. They are generally aimed towards low-end control-oriented embedded systems, being packed with little memory and sometimes even additional peripherals, to provide a complete single chip control solution.

- **D**igital **S**ignal **P**rocessors (DSPs);

  Digital **S**ignal **P**rocessors (DSP);

  DSP are a particular set of ASIP which are optimized for data stream-oriented applications, often including dedicated instructions and peripherals that are suited for digital signal processing, such as floating-point units.

- SoC;

  An SoC integrates all components of a computer system in a single chip, and is best suited for middle-end/high-end embedded purposes, being one of the most popular choices in embedded system design nowadays. SoC may be comprised of one or more microprocessors, usually 32 or 64 bits, DSPs, microcontrollers, memory blocks, peripherals such as timers and external interfaces, such as USB or Ethernet, also found in some microcontrollers.

Figure 2.2 presents a block diagram of a possible SoC chip.

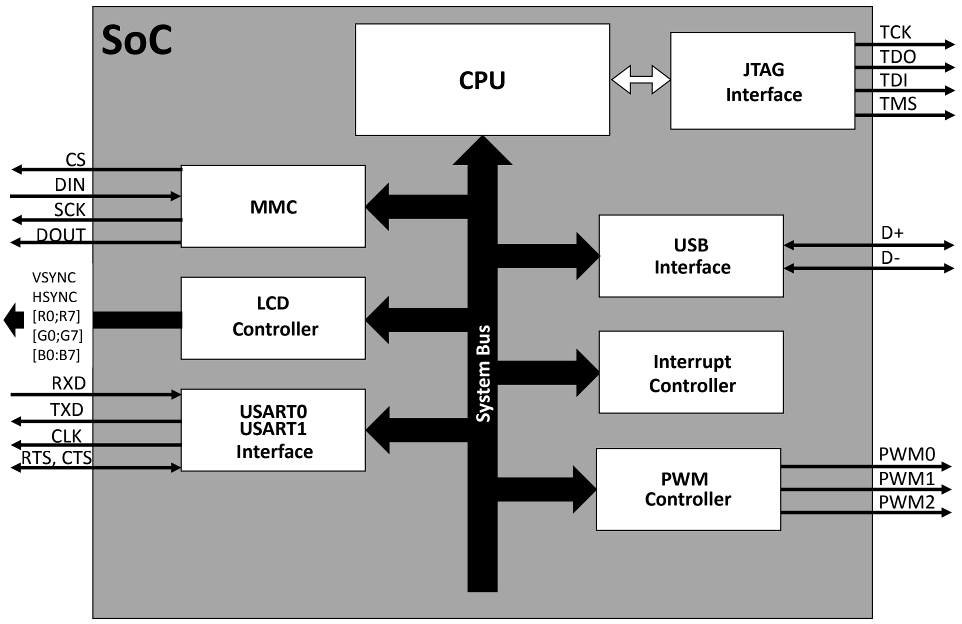


Figure 2.2: System on chip block diagram (Oliveira, 2013)

2.1.3 Embedded Development

Embedded system physical hardware and respective boards are often designed and developed along with software, being tailored and designed specifically for the target scenario. However, it is not practical to wait until the hardware prototype is finished to develop software, and not a very good idea to mix software with board development, as it may be hard to trace system faults and identify them as a software bug or a board malfunction.

To develop and debug software more easily, development platforms are used as a more practical way of prototyping embedded systems. Embedded development platforms are boards that are designed often by the processing chip's own manufacturers, containing a general set of components and interfaces that complement the chip's functionality, such as memories, Ethernet connectors, **M**ulti**M**edia**C**ard (MMC) card connectors and **G**eneral **P**urpose **I**nput/**O**utput (GPIO) expansion connectors, which combined will produce a platform that can be used to easily prototype embedded systems. Figure 2.3 presents a block diagram of a possible development board designed around an SoC.

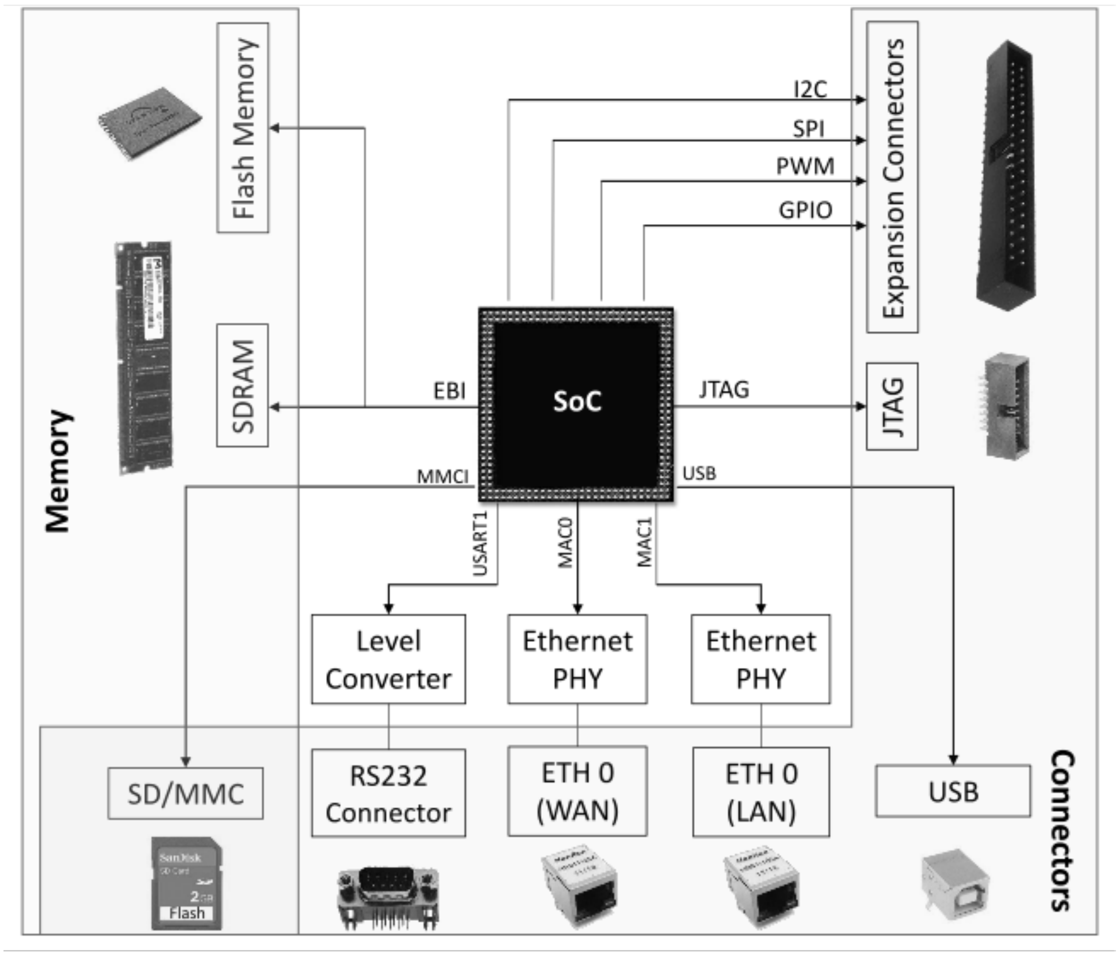


Figure 2.3: Embedded development platform block diagram (Oliveira, 2013)

Software development for these platforms is usually done on a host desktop system, where software is coded and later transferred to the target embedded platform. To be able to do all of this, a toolchain is usually provided for the target architecture. A toolchain comprises programming tools needed to effectively use the target platform, such as compilers, linkers, debuggers, loaders and other utilities. The act of compiling software for an architecture that is different from the one where compilation is being done is called cross-compiling, with embedded toolchains generally comprising cross-compilers and other such utilities.

After a prototype is developed and ready for production, the system may be integrated in a final board or boards which are tailored for the application's purpose, and possibly leaving out elements present in the development platform that are not used. Consider, for instance a router. If the development board included an LCD, it should be left out of the final product board, as it is not used by the router.

2.1.4 Real-Time Embedded Systems

Real-time embedded systems are a special kind of systems which are designed to perform certain critical tasks and satisfy specific time constraints, being referred to as deadlines. This means that these constraints drive the design, and if system responsiveness is limited, the whole system could be compromised.

Sometimes, real-time requirements are mistaken with high performance, which is not true. For instance, consider two chess software application scenarios are considered, one for a tournament with timed moves and one for personal recreational purposes with indefinite time to compute a move. In both applications, high performance is desired. However, in the tournament application, if a decision is not made in the given time and the deadline is not met, the computer loses the game. Thus, the correctness of a real-time embedded system depends not only on the results produced, but also when they are produced.

Real-time embedded systems may be classified in three categories: hard real-time, firm real-time and soft real-time systems.

Hard real-time systems, are systems in which missing a deadline results in total system failure. Examples of hard-real time systems are a nuclear reactor control system, or anti-lock brakes on a vehicle. In both these cases, if the system does not meet its required deadlines, total system failure occurs with very harmful consequences. Other examples of hard real-time systems include pacemakers and avionic applications.

Firm and soft real-time systems, are systems that do not fail if a deadline is not met, but their services degrade considerably with missing deadlines. Further distinction is made between firm and soft real-time systems, regarding a result's validity after the given deadline. If a system's computed result is completely invalidated by not being produced in its given time window, it is considered firm real-time. If the a system's computed result is still valid, although with reduced value after failing to meet a deadline, it is considered soft real-time. An example of a firm real-time system could be a weather forecast system. If a deadline is missed, no system failure will occur and the system may continue making predictions. However, if the system makes a late prediction, like predicting a storm that has already happened, it is totally invalidated and has absolutely no use. An example of a soft real-time system can be a nuclear reactor user interface. Although the nuclear reactor control system itself is a hard real-time system, providing an interactive man-machine interface is not as critical, and delayed results, although degrading the system's value, are still valid.

A real-time embedded system should be dependable, and although not exclusive to real-time embedded systems, some of these characteristics are particularly desirable when designing such a system:

- Reliability;

  Continuity of service delivery while in use, that is, the probability of the system working properly since it started working after start-up. If a system is reliable, it won't need to be checked often, that is, if it it's down and needs a reset.

- Maintainability;

  Probability of the system working properly a few moments after an error occurrence, that is the impact a system error has on the system, and how quickly it can be properly running again.

- Availability;

  Readiness for use when needed, that is, probability of the system working at a given time instant.

- Safety;

  Does not cause catastrophic effects on the environment or user as a consequence of failure.

- Security.

  Provides communication confidentiality and authentication.

Real-time embedded system development is an especially difficult area, requiring varied skill sets from engineers, from knowledge on compiler technology and debugging techniques to hardware devices and knowledge on programming techniques that enhance system responsiveness, all the while keeping in mind timings and not compromising its real-time requirements. Real-time system design often deals with concurrency, with multiple tasks being executed with different priority levels.

2.2 Operating Systems

Operating systems are software that manage machine resources, and provide several abstractions and commonly shared application services that create execution environments that aim to facilitate application development. As embedded technology grew to meet ever growing system demands, the adoption of OS for system development became very common, being an integral part for several embedded systems.

2.2.1 Concurrency

Responsiveness and concurrency are often important aspects of an embedded system. This may prove to be a paradox, since responsiveness means that the system should respond quickly to external events, and concurrency means that several tasks must be carried out simultaneously. Microprocessors are capable of executing only one instruction at a time, and if an external event triggers a processing task, it should mean that the processor would be unavailable for any other external events. However, there are techniques to overcome this limitation. Virtual parallelization is a concurrent programming technique that shares multiple task execution code to achieve an illusion of concurrent execution. Figure 2.4 presents an example diagram of two tasks being executed concurrently.

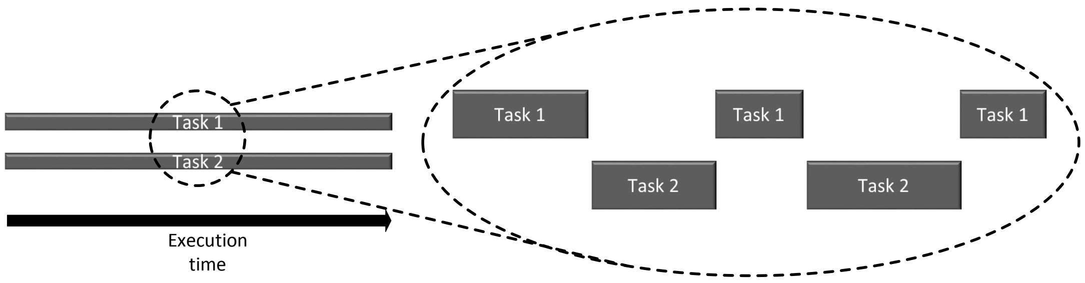


Figure 2.4: Virtual parallelization of two tasks

Two tasks are executed by the processor once at a time, with CPU execution jumping from one task to the other, with each task being executed in little bits. By not being confined to executing a task all the way to the end exclusively, the system is not held until the task finishes and its responsiveness is not compromised. Task switches can be implemented using timer interrupts, with scheduling code being ran in the interrupt to decide which task code should the interrupt return to. If the task switches are frequent enough, it may seem as though the system is running two tasks in parallel. This is the basis of the multi-tasking paradigm used in most computational systems nowadays, and it is one of the most important aspects of an operating system.

At its absolute most basic form, an OS can be a software program that allows for task code implementation, running a scheduler that decides which task should be in execution. There is a plethora of scheduling algorithms and techniques, with the two main groups being cooperative scheduling and preemptive scheduling.

In cooperative scheduling, it is up to the tasks to make system calls that give up execution to other tasks. In this scheduling model, there is no need for timer interrupts and OS intervention, as it is up to the tasks to yield execution and cooperate among them to maintain a balanced system execution. The downside of this model is that a badly implemented task may compromise the system and hold up the processor. Although now rarely used in large OS it was once a widely popular scheduling scheme.

In contrast, preemptive scheduling is a scheduling scheme that implements OS intervention, usually through a timer interrupt that runs scheduling algorithms and decides which task should run next. This strategy provides some abstraction to the user, as tasks can be implemented without having in mind system management, also ensuring better guaranties that a task will not hold up the processor. Nowadays most large OS have scheduling algorithms and techniques that fall under the preemptive category, from Windows and Mac-OS to Linux.

2.2.2 Kernel and User Spaces

An OS is a software program that allows for concurrent task execution, through scheduling algorithms that implement virtual parallelization. However, most OS are much more complex, managing software and hardware resources and providing a set of services that are common to most applications. In modern OS, the kernel is the core of the OS and has privileged access to the hardware resources, managing memory access, scheduling application execution in the CPU, and other hardware related responsibilities. Most modern large OS implement at least two execution spaces in them: kernel space and user space. Figure 2.5 presents a diagram of the common software layers of an OS.

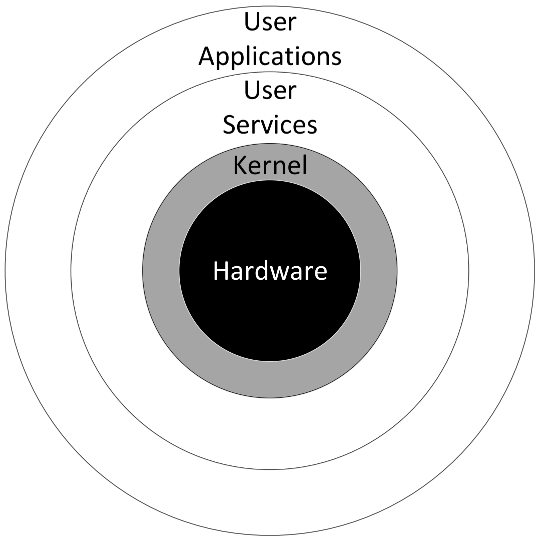


Figure 2.5: Operating system layers diagram

Kernel space runs code that accesses hardware like memory and I/O, providing an interface through system calls to upper layers. User space runs OS services that do not require direct hardware access, like network services and user interface services like **G**raphical **U**ser **I**nterfaces and command line shells. User applications are also ran in user space, often being thought of as another layer, given that these applications frequently run on top of user space OS services.

This architecture of kernel and user space is not set in stone though, with existing several approaches and architectures that run some services in user space, and others that run the same services in kernel space. The two main type of kernels are microkernels and monolithic kernels.

Microkernels contain the minimum amount of services running in kernel space, with most services being ran in user space, whereas monolithic kernels run most of their services in kernel space. Regardless of being implemented in user space or kernel space, most OS provide a set of basic services, using a set of concepts that make possible the modern computing systems that we see today.

- Processes and multi-tasking

  Perhaps the most important feature of an OS as was previously covered, is the implementation of schedulers and multi-tasking, which is essential to system responsiveness and concurrent application execution. Besides of what was previously covered, most modern OS also implement the concept of processes and threads. A process is a program in execution, often with its own memory space. Threads are also a common concept, with a process being composed of one or more threads. Figure 2.6 presents a diagram of a common architectural approach of processes and threads in an OS, with two processes being represented, one single-threaded and one multi-threaded.

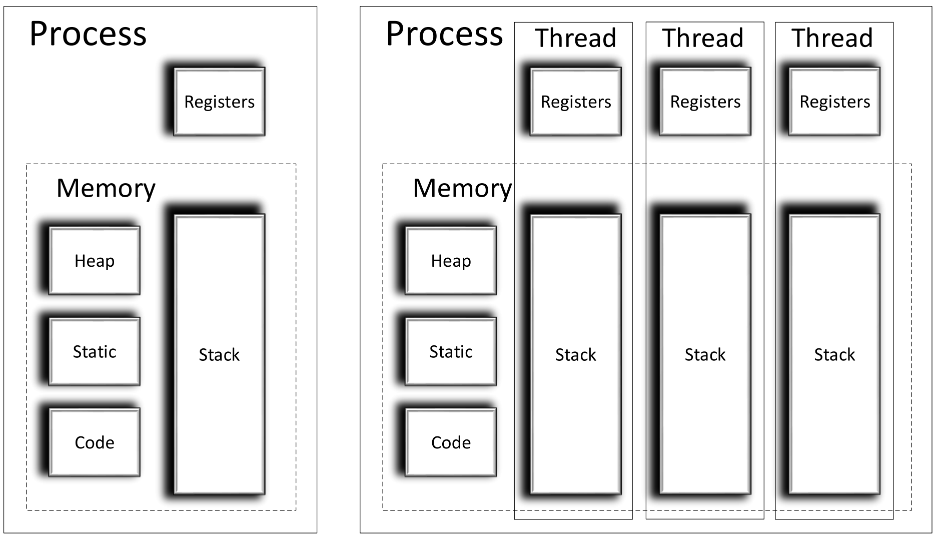


Figure 2.6: Processes and threads block diagram

  Processes are programs in execution, with separate memory spaces and and execution contexts. Execution context is illustrated with a combination of registers and stack, representing the CPU state for each task. OS processes often implement a hierarchy within the multi-tasking model, implementing internal sub-tasks, called threads. Threads are tasks within processes which share the same memory belonging to their respective process, but have their own execution context, enabling for concurrent execution and virtual parallelism within a single process.

- Virtual Memory and Memory Management

  Virtual memory is a technique that greatly improves system performance, usually tightly coupled with the concept of processes, although it is also used in single address space OS. With this technique a process assumes that the whole system's memory is free, liberating processes from the responsibility of managing memory addresses. Processes' memory addresses are interpreted by the OS as virtual addresses and then translated to actual physical addresses. Figure 2.7 presents a diagram of two processes with their respective memory addresses being mapped into physical memory.

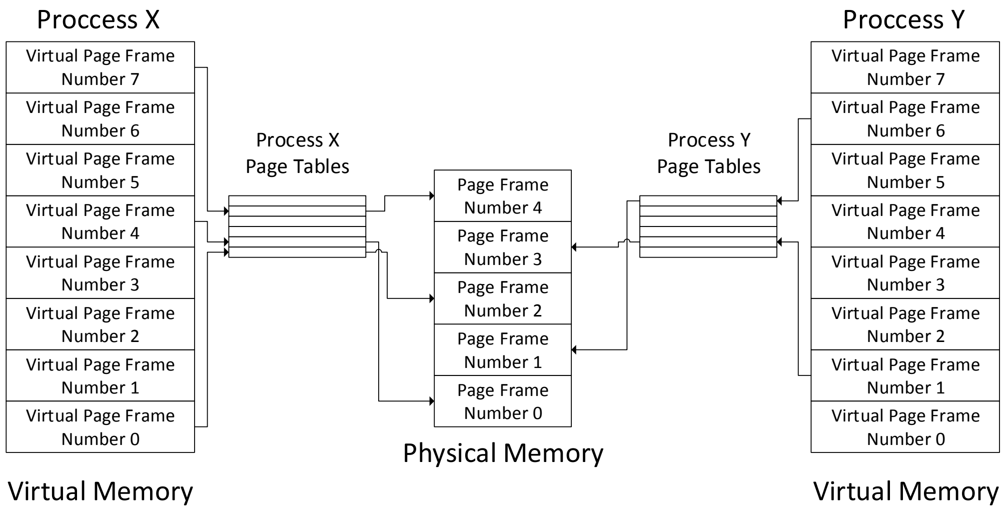

Proccess X
Virtual Page Frame Number 7
Virtual Page Frame Number 6
Virtual Page Frame Number 5
Virtual Page Frame Number 4
Virtual Page Frame Number 3
Virtual Page Frame Number 2
Virtual Page Frame Number 1
Virtual Page Frame Number 0
Virtual Memory
Process X Page Tables
Page Frame Number 4
Page Frame Number 3
Page Frame Number 2
Page Frame Number 1
Page Frame Number 0
Physical Memory
Process Y Page Tables
Proccess Y
Virtual Page Frame Number 7
Virtual Page Frame Number 6
Virtual Page Frame Number 5
Virtual Page Frame Number 4
Virtual Page Frame Number 3
Virtual Page Frame Number 2
Virtual Page Frame Number 1
Virtual Page Frame Number 0
Virtual Memory

Figure 2.7: Virtual memory diagram

  Usually, memory is divided into unitary blocks commonly referred to as page frames. In OS that employ separate memory address spaces for each process, such as the one illustrated, page tables must be associated with processes. This is done so, because page tables must supply different virtual memory mappings for different processes, as they may use identical virtual addresses.

  Virtual memory may be augmented with secondary storage such as hard disks, MMC and **S**ecure **D**igital (SD) cards, with the two latter being the most common choices in embedded systems. Memory page frames can be swapped between the system's main memory and secondary storage, being moved in and out of the main memory according to the needs of the process that is in execution.

In this case, the page tables must keep additional information for each page, often in the form of bits like the present bit and the dirty bit, that represent their states in the secondary storage memory space. The present bit indicates what pages are currently present in physical memory or secondary storage devices, allowing the OS to know how to treat these different pages, like whether to load a page frame from secondary storage devices and page another page frame in physical memory out. The dirty bit allows for performance optimization, and is used to signal if a page frame that's loaded onto the main memory has been changed since loaded from the secondary storage device. If not, when the page frame is paged out, there is no need to update the page frame on the secondary storage device.

- File system

  A file system is an abstraction that is provided to manipulate data in storage devices. It provides a way of organizing data, usually in a hierarchy of directories or folders arranged in a directory tree. Each group of data is called a file, and follows a given structure and logic rules so data can be recognized and organized. Often, file systems are complemented with a **V**irtual **F**ile **S**ystem (VFS), that allows applications to access concrete file systems in a uniform way.

- Device Drivers

  Device drivers are software entities that contain implementation of software that is related to the specificness of the underlying hardware. System calls are usually provided, so applications can invoke device drivers to interact with devices, releasing the user applications from specific hardware implementation responsibilities. Some kernel architectures, usually monolithic kernel ones, implement device drivers in kernel space to prevent user applications from crashing the whole system with ease.

2.2.3 Real-time Operating Systems

Although OS have greatly improved computing systems as a whole, not everything is an upside about them. In fact, in the embedded systems field, namely real-time systems, common OS overhead and uncertainty is not acceptable. A general purpose OS has lots of overhead sources, from boundary crossing memory copies to task switching. Boundary crossing memory copies occur when user space and kernel space transfer data, usually in device drivers. Frequent buffer copies are a heavy toll on the system, providing a great deal of unnecessary overhead for some embedded systems. Also, task switching introduces latency, as one task's context must be saved, and another restored.

Finally, most general purpose scheduling algorithms are not greatly suited for real-time scenarios as task's deadlines are not taken into account, and uncertainty on when a context switch is going to occur,and thus when a task gets processor time and executes is a dooming factor.

For all of the above reasons, real-time embedded systems often deploy **R**eal-**T**ime **O**perating **S**ystems (RTOS). **R**eal-**T**ime **O**perating **S**ystem (RTOS) are operating systems that are specifically designed for real-time application purposes, implemented to schedule real-time tasks with deadlines, usually under a system of priorities, with high priority tasks guaranteed to run under a certain amount of time. Often RTOS do not implement many common general purpose OS services, with common microkernel architectures, for instance implementing device drivers in user-space to avoid boundary crossing overhead. Examples of real time operating systems include FreeRTOS, MontaVista Linux and RTEMS.

2.2.4 Linux

Linux is an open-source OS, initially developed by Linus Torvalds as a hobby. Its kernel, which is monolithic, was first released on the 5th of October, 1991. Since its first inception, Linux has evolved rapidly, being widely adapted to most modern systems and becoming one of the most successful OS, on par with famous commercial OS such as Windows and MacOS.

The Linux kernel is licensed under **G**eneral **P**ublic **L**icense (GPL) which is a license that grants free access to the original software code, as well as the possibility to change it and redistribute at will. Several OS make use of the Linux kernel, being referred to as Linux distributions. Although there are not that many desktop Linux users compared to other popular OSs, it is the preferred OS of choice in supercomputers, servers and most embedded devices, such as MP3 players, DVD players and HD TVs.

Although Linux was initially developed for Intel x86 compatible machines, no other OS has been more ported to other architectures (Silva, 2011). With a large selection of device drivers for several common hardware devices, it is obviously a hugely popular platform for embedded development, as rapid deployment of an OS with wide hardware support for a multitude of architectures and devices reduces time-to-market significantly. Some of the following characteristics may explain Linux's popularity, namely in the embedded system field:

- Available on a big selection of architectures;
- Supports a large selection of hardware devices;
- Product distribution free of royalties;
- Support from the world's major suppliers, from hardware platforms to semiconductors and software applications. Some of these include: IBM; Texas Instruments; Atmel; Samsung (Silva, 2011);
- Open source and widely documented, allowing for easy portability to new architectures and expansion to new hardware devices.

Linux, being originally developed for desktop systems, was not designed with real-time requirements in mind. It is based on time sharing policies, which aim to improve user experience by sharing CPU execution as equally as possible. However, this is contradictory with real-time requirements, as a real-time process must have absolute priority on the CPU to meet the required deadlines (Silva, 2011).

To help solve some of these limitations, there is a Linux kernel patch for hard real-time systems, formally known as, RT Preempt, that changes several kernel source files to implement real-time policies on the kernel. However, this patch is insufficient for some scenarios, and other techniques should be used to meet hard to meet deadlines, like user space memory mapping to inhibit boundary crossing memory copying latencies, using **D**irect **M**emory **A**ccess (DMA) peripherals or accelerate applications through programmable hardware. Appendix A.1.2 details the steps to apply the RT Preempt patch and compile the kernel.

2.2.5 Device Tree

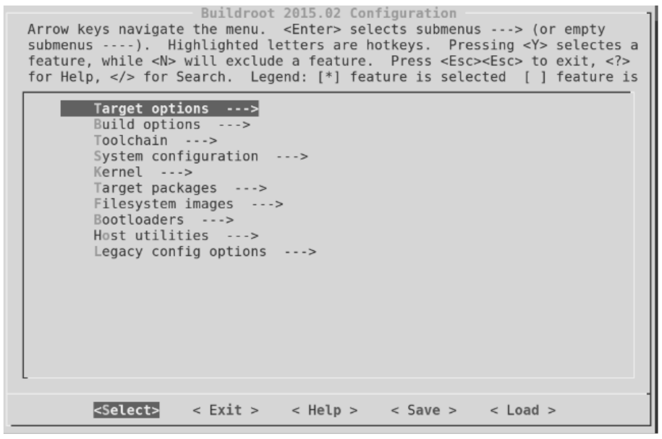
Traditionally in the Linux kernel, all boards were described through source files, implementing static descriptions of all the hardware. If a new board should be supported, that meant adding source files to implement support and recompile the kernel, even if differences between the new board and an already supported board were minimal. From Linux 2.6 onwards, a solution to this problem was implemented, which separates hardware descriptions features from the kernel implementation through the dynamic use of a data structure that describes the hardware.

Inspired on Open Firmware, the **F**lat **D**evice **T**ree (FDT) is a description of the machine hardware configuration, describing most board aspects such as the number and type of CPUs, base addresses, size of **R**andom **A**ccess **M**emory (RAM), buses, bridges, peripheral device connections, interrupt controllers and **I**nterrupt **R**e**Q**uest (IRQ) line connections (Petazzoni). This information is described in **D**evice **T**ree **S**ources (DTS) which are human-readable files, later compiled into their binary forms, **D**evice **T**ree **B**lobs (DTB), through the **D**evice **T**ree **C**ompiler (DTC). Appendix A.2 contains an example of a .dts file.

Linux currently supports the device tree in the ARM, x86, MicroBlaze, PowerPC, and SPARC architectures, with mandatory use becoming the rule for new ARM SoCs. The device tree is useful in these cases, specially if the SoCs contain programmable logic, allowing for quick kernel expansion for custom hardware devices. Although included in the Linux kernel sources, the main goal is to migrate the device tree out of the kernel, since the description is platform independent and aims to be an adopted standard for several OSs, with the hope of solving perpetual **B**oard **S**upport **P**ackage (BSP) development, which is a problem amongst many embedded platforms. **D**evice **T**ree **B**lob (DTB) may be linked into the Linux kernel image, as well as dynamically parsed at kernel runtime during its initialization process.

2.2.6 Buildroot

Buildroot is a an open-source free tool developed by Free Electrons, consisting of several makefiles and patches which aim to help automate the process of cross-building an embedded Linux system (Free Electrons). This tool automatically downloads, extracts and builds packages, enabling the generation a complete toolchain for the target platform, target root filesystem as well as respective Linux kernel image and necessary bootloader. It is easily configurable through a set of graphical interfaces being triggered as makefile rules, which allow target package selection, as well kernel fine-tuning, overall enabling an easy configuration of the provided services. Figure 2.8 presents menuconfig, one of the possible graphical interfaces for configuration.



```
                     Buildroot 2015.02 Configuration
 Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
 submenus ----).  Highlighted letters are hotkeys.  Pressing <Y> selectes a
 feature, while <N> will exclude a feature.  Press <Esc><Esc> to exit, <?>
 for Help, </> for Search.  Legend: [*] feature is selected  [ ] feature is

            Target options  --->
            Build options  --->
            Toolchain  --->
            System configuration  --->
            Kernel  --->
            Target packages  --->
            Filesystem images  --->
            Bootloaders  --->
            Host utilities  --->
            Legacy config options  --->

         <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 2.8: Buildroot make menuconfig prompt

Being actively maintained and frequently updated as well as thoroughly documented, it is a very popular tool with some notorious adopters such as Google, Atmel and Analog Devices. Recent hobbyist trends in development boards such as Raspberry Pi or BeagleBone Black have prompted Buildroot popularity also, due to the offered simplicity in getting Linux to run on these boards. Appendix A.1 contains several information on how to use Buildroot, namely installing it, compiling Linux with real-time kernel support, as well as cross-compiling an application.

2.3 Hardware Acceleration

Some real-time controllers require tight deadlines, proving to be very difficult scenarios where the software resources are not enough to meet the overall system constraints. As mentioned in subsection 2.2.3, the use of an OS introduces uncertainty and overhead to the system, sometimes impossible to mitigate to an acceptable degree even enforcing real-time scheduling policies, and zero-copy boundary crossing techniques. In these scenarios, it is a modern practice to incorporate hardware acceleration, making use of the hardware true parallel nature and offload critical application kernels to the custom hardware co-processors, thus freeing up CPU resources.

2.3.1 Hardware Description Languages

Hardware **D**escripton **L**anguages (HDL) are programming languages that are used to describe electronic circuits, more specifically digital electronic circuits. These languages allow for circuits to be accurately described, and can be used to simulate these circuits or even synthesize them for physical deployment. Several abstraction levels can be used in hardware implementation:

- System Level;
- Register Transfer Level;
- Gate Level;
- Transistor Level.

The abstraction level defines how much implementation detail is contained in the design, being very useful for big projects, in which a high abstraction level can be used to start the overall design and validation process, with subsequent development iterations consisting in implementation detail refinement. Due to the circuit's concurrent nature, **H**ardware **D**escription **L**anguage (HDL) programming paradigms are radically different from control-flow programming paradigms used on languages such as C/C++.

HDL designs are systems that continuously act on a set of inputs and outputs being composed by a set of modules, each with its own set of inputs and outputs which can be thought of blackbox subsystems being interconnected to form the overall device. Figure 2.9 presents a diagram of an HDL design example, to illustrate its hierarchical organization.

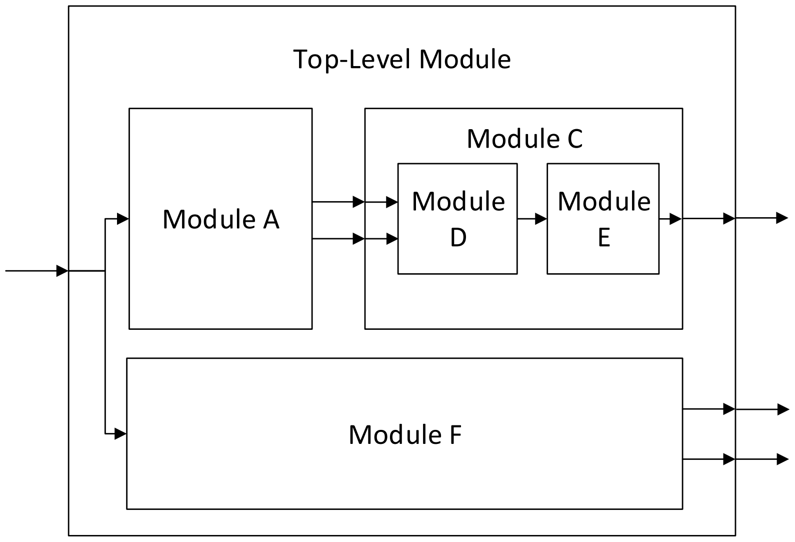


Figure 2.9: HDL design diagram

Synthesis tools compile the HDL code, and generate the physical implementation of the design based on parsed code. However, before the design is properly deployed, it should be validated through the use of HDL simulation tools. As shown previously the designed device requires input stimuli and produce outputs accordingly, so to properly simulate HDL designs, the concept of testbench is generally used. Under this concept, non-synthesizable constructs are separated from synthesizable construct, mimicking a traditional electronic testbench with signal generation to provide stimuli for the devices under test, subsequently monitor their behavior. Figure 2.10 presents a block diagram of an HDL testbench.

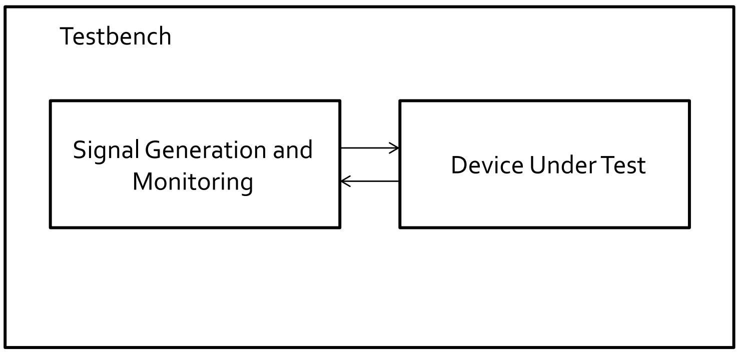


Figure 2.10: HDL testbench diagram

There are currently two main HDLs in use: VHDL and Verilog. VHDL's syntax is similar to the ADA programming language, being a rich and strongly typed language, deterministic and more verbose than Verilog. As a result, designs written in VHDL are generally extensive, and thus considered self-documenting. VHDL emphasizes unambiguous semantics making it well suited for implementing designs under system level abstractions. Verilog's syntax is very C-like, which is a common language used by engineers in the field thus providing a good entry level learn-curve. It is weakly typed and doesn't emphasize semantics as rigidly as VHDL, which may cause design problems that are only later found when being used by inexperienced developers. On the upside, it is best suited for lower level abstractions and structural implementations for the same reasons. VHDL is more commonly used in academic contexts and the scientific community, while corporations usually employ Verilog. There are also regional preferences, with Europe showing a predominant use of VHDL and Verilog being more adopted in America. The two languages are not mutually exclusive, with most modern synthesis and simulation tools allowing for designs that mix VHDL and Verilog.

2.3.2 FPGA

FPGA, briefly mentioned in subsection 2.1.2, are integrated circuits which contain reprogrammable hardware. The two main FPGA suppliers, Xilinx and Altera, share a market duopoly, with Altera being recently acquired by Intel. The FPGA technology's uniqueness derives from the fact that unlike other processor technologies, the hardware fabric may be reprogrammed, enabling them to implement other devices that are usually implemented in chips using ASIC technology, like microprocessors or any other chip that contains its implementation hardwired in silicon. The two basic elements that are present on an FPGA chip are flip-flops and **L**ook**U**p **T**ables (LUT). These elements are organized in an array with interconnections amongst them that may be activated or not, thus implementing several behaviors and consequently more complex circuits. Figure 2.11 presents a diagram the internal architecture of an FPGA.

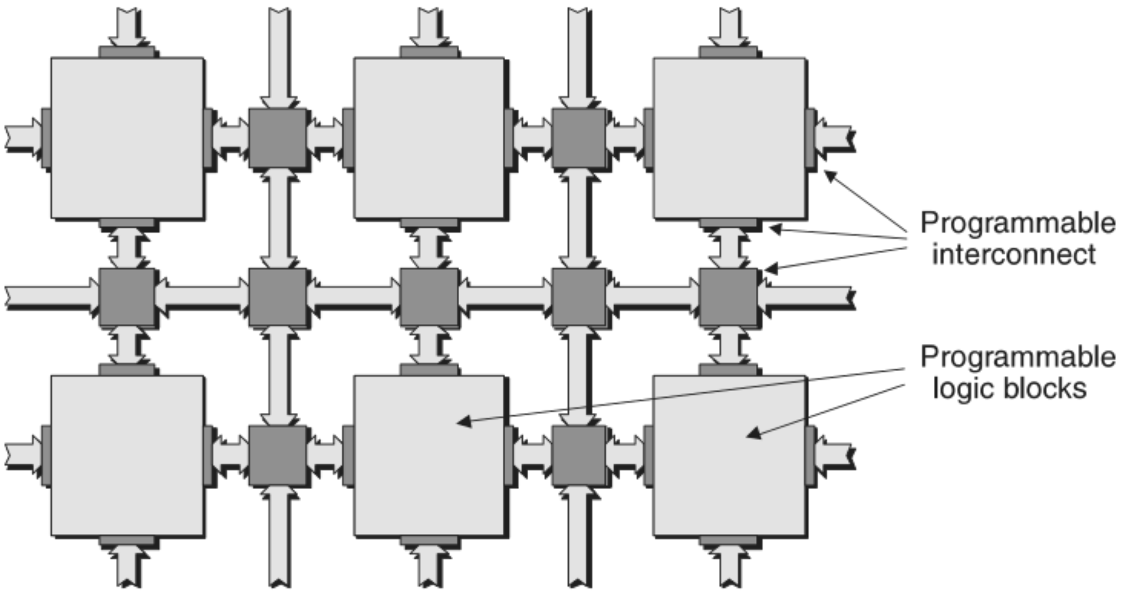


Figure 2.11: FPGA internal architecture (Maxfield, 2009)

Look**U**p **T**abless (LUTs) implement behavior of digital gates, by containing a table of gate outputs that are to be used according to gate inputs. By combining digital gates implemented in LUT and flip-flops, any kind of combinatorial or sequential circuit may be implemented. Figure 2.12 presents a diagram a simplified view of a 2-input LUT implementing an AND gate.

Most modern FPGA introduce larger elements in the interconnected array, with vendors often adopting their own nomenclature for these elements. Xilinx calls them **C**onfigurable **L**ogic **B**locks (CLB), while Altera calls them **L**ogic **A**rray **B**locks, for instance. These larger blocks contain LUT, flip-flops, multiplexers in a single block, allowing for faster designs with better resource utilization. However implementing all circuits using only LUT-based logic blocks may be rather inefficient and costly when compared their silicon hardwired counterparts, being slower, occupying large silicon areas and consequently dissipating more power. For this reason modern FPGA chips include hardwired elements that implement commonly used functions such as multipliers, memories and generic DSP blocks. FPGA commonly store their configuration in volatile memories, needing to be programmed when turned on. The configuration file of an FPGA chip is called bitstream. Most modern FPGA development boards include non-volatile memory

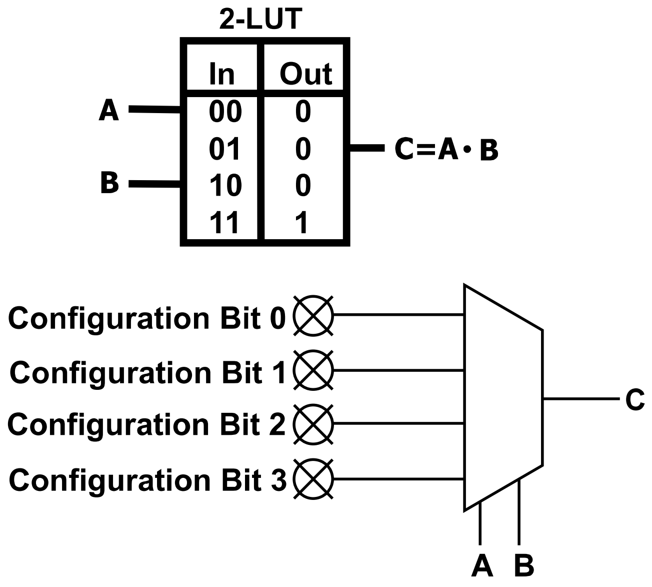


Figure 2.12: FPGA LUT (Huffmire et al., 2010)

chips to store bitstreams, and circuitry that may program the FPGA chip upon power-on.

2.3.3 FPGA SoC

Traditionally, FPGAs were used to prototype chips that would be implemented using ASIC fabric. ASICs have extremely high manufacturing costs for the chip's mask, which makes a mistake very costly, but implementing product designs in FPGA chips was not an option, due to significant power dissipation and latency when compared to hardwired silicon chip designs. These circumstances led to a clear flow where FPGA would be used as prototyping platforms for designs to be eventually implemented in ASIC chips. However, recent advances in FPGA technology have been making it more and more an available option for field deployment, specially in cases where there a large number of ASIC chips are not going to be manufactured, thus not covering the chip's mask costs. For instance, a lot of modern HD TV set models feature FPGA chips. This is due to the fact that several models are not being designed in way that favors a common processing architecture amongst them, and creating ASIC chips for every processing architecture is not profitable due to insufficient produced chips to cover these costs. There are also other advantages to this approach, like allowing hardware architecture updates and reconfigurations on manufactured products, thus adding flexibility to designs and eventual hardware bug fixes. Typically, when FPGA are employed this way, softcore processors are used, with possible architectural changes according to the desired purpose. Softcore processors, are general purpose microprocessors being implemented in FPGA fabric, often being implemented along with other peripherals constituting SoC architectures. Another field where modern SoC architectures are being highly employed is real-time embedded systems, where the FPGA fabric is being used for hardware acceleration purposes, offloading critical algorithms to dedicated hardware co-processors that execute in parallel with the CPU. In these cases, flexibility on the CPU architecture itself is not very desirable, with the main focus being on custom hardware for critical algorithm processing. Also softcore processors, like any design implemented in programmable logic, are more efficient when implemented hardwired in silicon. Thus, modern FPGA chips often constitute SoC architectures where whole CPUs are hardwired in silicon along with other commonly used SoC peripherals, and only a portion of the silicon in the chip is dedicated for FPGA fabric, to be used as programmable hardware for custom hardware peripherals and co-processors that may be connected to the chip's hardwired resources. Figure 2.13 presents the architecture on the latest Xilinx SoC architecture, the Zynq® UltraScale+$^{\text{TM}}$ MPSoC. This heterogeneous processing platform contains 4 ARM 64-bit cores, 2 ARM 32-bit real-time cores, an ARM GPU core and other elements such as DDR controllers, DMAs and many other peripherals hardwired in silicon, as well as a programmable logic silicon area all integrated into a single chip.

2.4 Hardware-Software Co-Design

Formally, the design of hardware accelerated embedded software systems follows a pattern. First and foremost, an application in a software language that models the behavior of the system is developed, without metrical concerns. This way, the algorithms that are going to be used in the system are developed and debugged. The application initially is deployed on the host platform, given that cross-compilation for the target platform is not relevant, due to the main concern being the general system behavior modeling. The next phase is the identification of the hot-processing kernels within the application, and their subsequent software parallelization with the help of multi-thread programming models. At this phase, the profiling tools are then used to characterize the application so that critical kernels can be selected as candidates to computational offloading to FPGA fabric. The next phase consists in the use of HDL to develop RTL models for the

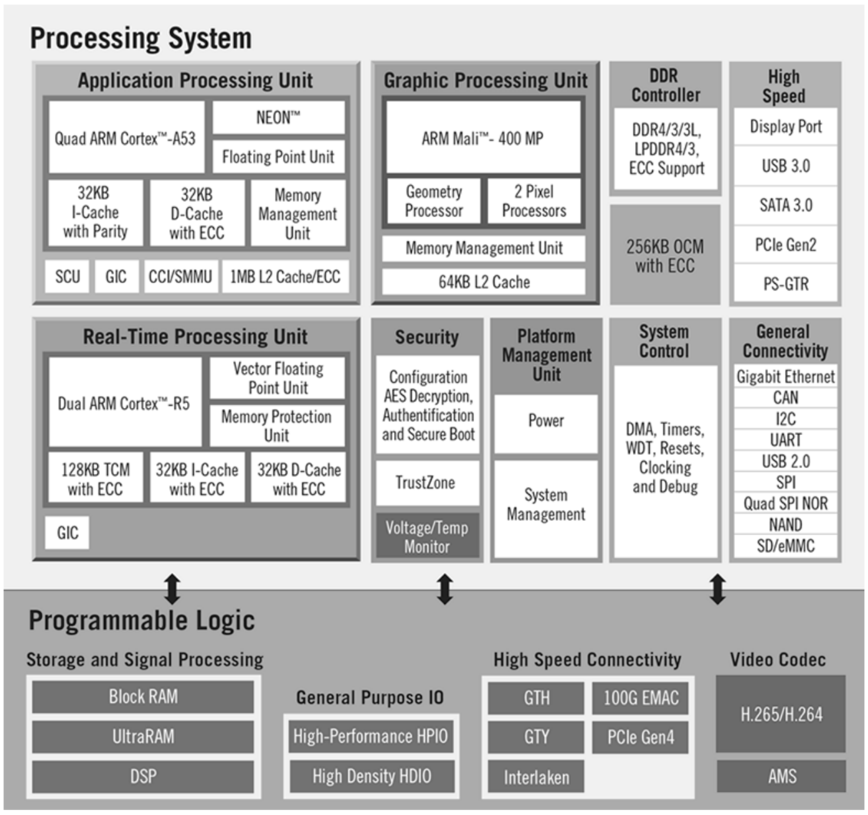


Figure 2.13: Zynq® UltraScale+^{TM} MPSoC architecture diagram (Xilinx)

identified processing kernels and the overall system then recompiled to the target platform and validated, with the design being completed at implementation phase, ensuring the system desired metrics are satisfied. This is an iterative process, so if the desired metrics are not met, the development may return to the hardware IP development phase, or even the profiling phase.

2.4.1 System Modeling

To model the system, an application is typically developed in more flexible and functional software programming languages, like C/C++, which are programming languages of choice for embedded software. In this way the code is as generic and portable as possible, as it is to be compiled either to host or target platforms, the later through cross-compilation. The application is first tested at the host platform abstracting away the target platform implementation details.

2.4.2 Software Parallelization

Software parallelization is usually achieved be means of multi-thread programming models such as the **P**ortable **O**perating **S**ystem **I**nterface (POSIX) API, widely used in Unix-based systems. After the tasks are identified, the multiple task algorithms are assigned to different threads. Since threads tend to share data, synchronization mechanisms are mandatory to avoid race conditions, and are usually provided by the multi-threading API. This technique is not a true nature parallelization, as only one thread is executing at one CPU core in a certain time, hence being a virtual parallelization.

2.4.3 Profiling

Profiling a software application, is the act of using several profiling tools to characterize the application. In this phase, the application by now developed in the host is profiled. Normally, the differences in the host and target platform are not relevant enough to justify a profiling in the target platform, so profiling the host application is acceptable enough. However it is highly recommended that the OS in the host platform runs a real-time scheduler, so that the application being profiled can enforce a deterministic behaviour. This is a specially important aspect in profiling, since profiling tools must be used several times to gather data, being that statistics of the gathered data are the deliverables for this phase.

2.4.4 Hardware Design

Hardware IP are developed with the aid of HDL languages, namely Verilog or VHDL. The development is done on a register transfer level, and is then validated through debug on RTL simulation tools. Example of these tools are Isim from Xilinx and Modelsim from MentorGraphics.

Synthesis tools will afterward synthesize the IP and routing tools map the foorplan, so that the design may be implement in an ASIC or FPGA. These tools are normally integrated in a toolchain, and made available in an integrated development environment, like Xilinx ISE or Altera Quartus II

Currently, microcontrollers and custom IP devices are developed in similar design flow, and such design is then validated and debugged in FPGA-based platforms.

At completion the overall design is then shifted to ASIC development. The development process may be a long and complex one, and very time consuming even with current tools available, specially in this approach, as validation is performed at the target platform. Besides, HDL coding paradigms are inherently different than software programming paradigms. Due to these reasons, there are tools that attempt to translate software programming coded algorithms to HDL code. In such tools one can follow the Vivado **H**igh **L**evel **S**ynthesis (HLS) design flow.

2.4.5 Validation

In this phase the developed system must be validated integrally. Following a co-simulation methodology, the development process can be accelerated in a substantial way, anticipating design decisions before committing to the hardware platform, and enabling the test and debugging of the entire system. In this design methodology, the target platform deployment is achieved when full-system integration and validation is concluded.

2.4.6 Metrics Confirmation/Implementation

Ideally this phase must consist of the implementation of the system in the target platform, with the meeting of the designed metrics and system functionality being confirmed afterward. Through tools like Chipscope, that create a silicon wrapper in the developed system, it is possible to analyze the temporal signals of the physical system, in order to confirm that the temporal requirements are really met.

The concept of integrated co-simulation environment may be expanded so that a multitude of metrics are validated after qualitative validation, so that a quantitative validation is done, expanding the its utility.

2.5 Hardware Accelerated Embedded System Simulation

The use of simulation is frequent in several engineering fields, allowing engineers to test designs and simulate systems, proving to be an almost indispensable approach in the development process. In hardware accelerated embedded systems, development is usually done on multiple application domains, with system complexity often standing in the way of accurate simulations. Contemporary electronic systems are built usually as tightly coupled: user designed specialized hardware, the embedded system used to control that hardware and to process and retransmit the acquired data, and software running on that embedded computer. Testing of all listed parts separately is difficult, or even impossible. Without possibility to perform the joint simulation of both - hardware and software components, the development becomes expensive and time consuming. In typical development cycle we have to design the hardware prototype, basing on some assumptions regarding possible software solution. The work on software part may be started when the hardware specifications are ready, but thorough testing of the interactions between the hardware and software parts must be delayed, until the hardware prototype is ready. If the results of tests show the need for significant changes in hardware design, it is necessary to prepare next prototype and subject it to tests. Sometimes the above step must be repeated a few times, depending on the design complexity, and on skills of the development team (Zabołotny, 2012).

In this context, focus will be made on two different kinds of simulations: instruction-accurate simulation (often referred to as software simulation) and cycle-accurate simulation (often referred to as hardware RTL simulation). Software simulators like QEMU can emulate the target platform providing instruction-accurate simulation of the software running on it (Zabołotny, 2012). The machine's hardware is emulated functionally, and the instruction set for the target machine is emulated, allowing for full software stack simulation for the target platform. With this kind of simulation, user application software may be validated, as well as OS components themselves. Hardware RTL simulators like Isim or ModelSim simulate hardware specified by HDL sources, and offer cycle-accurate simulations that accurately simulate hardware behavior. This kind of simulation is generally used to validate hardware components themselves.

2.5.1 Full System RTL Simulation

One approach to simulate the whole system is to use RTL simulation to simulate the whole machine running software on it. Figure 2.14 presents a diagram of full-system RTL simulation on a development host.

This approach is adequate for system that are centered around simple embedded

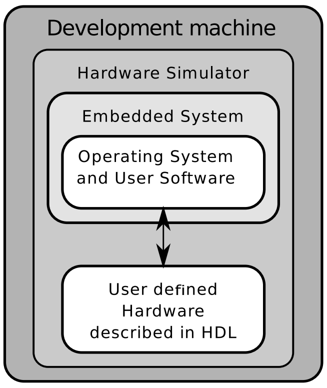

Development machine
Hardware Simulator
Embedded System
Operating System and User Software
User defined Hardware described in HDL

Figure 2.14: Full system RTL simulation diagram (Zabołotny, 2012)

processors, but as RTL simulation analyzes the state of all logic gates and registers in every integration step, it is highly innefective to simulate embedded systems that contain embedded processors that implement software layers, specially if software is complex like OS environments. An example of the inadequacy of this simulation for software is that a simulation of an OpenRisc CPU running a simple C program, takes 40 seconds of simulation time for a simple welcoming message display Balducci (2009), so simulation of any serious OS running on such platform would be too slow to be useful (Zabołotny, 2012). The solution to this problem is to avoid software simulation in RTL simulators altogether, running software parts externally of RTL simulation.

2.5.2 RTL Simulation with Host Software

In this approach, hardware designs that need validation are simulated in RTL simulation, while software is ran directly on the host development machine. Figure 2.15 presents a diagram of an RTL simulation being provided with stimuli from software being executed on the host development machine.

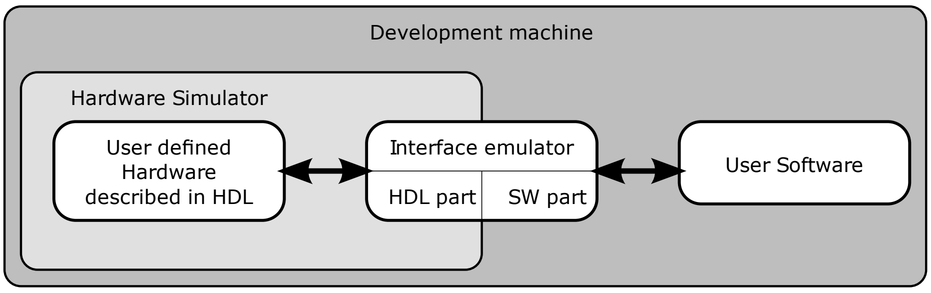


Figure 2.15: RTL simulation with host software diagram(Zabołotny, 2012)

The embedded software application must be compiled for the host development machine, replacing device driver system calls with **A**pplication **P**rogram **I**nterface (API) calls for an interface to the RTL simulation that must be implemented to emulate system bus transactions. Similar work must also be done on the HDL design, implementing an interface that emulates software accesses to the design. This approach is significantly better than full system RTL simulation, as native performances are achieved for software with RTL simulation only simulating the hardware that's relevant to be simulated this way. Also, the embedded application behavior may be debugged and validated, being an effective validation method for simple software. However, more complex software often needs further validation, as OS components are not validated, like device drivers, performance of data transfers, and so forth. To emulate these elements of an embedded system, software simulation must be combined with RTL simulation

2.5.3 RTL-Software Co-Simulation

Using this approach, an instruction-accurate simulator is used in conjunction with a cycle-accurate simulator. Figure 2.16 presents a diagram of an RTL-software co-simulation.

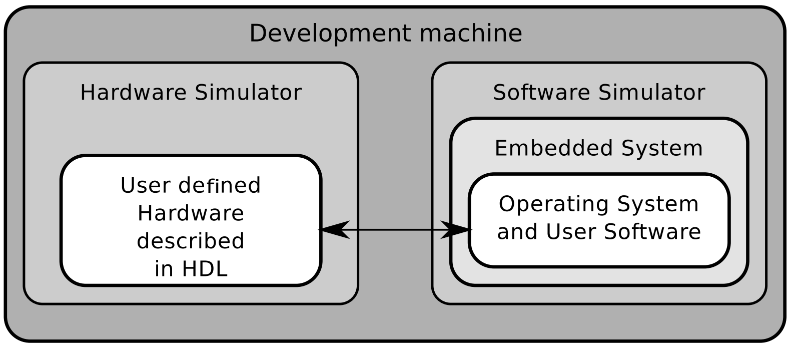

Development machine
Hardware Simulator
User defined Hardware described in HDL
Software Simulator
Embedded System
Operating System and User Software

Figure 2.16: RTL-Software co-simulation diagram(Zabołotny, 2012)

This approach is very useful, as the entire software stack is properly simulated for the target platform with good performance, and HDL models that are being developed are properly simulated in an RTL simulation. Provided that the simulators are open-source, or provide interfaces to extend the simulator through the usage of dynamic libraries, interfaces may be implemented that allow the simulation tools to exchange information, such as hardware access information to emulate system bus transactions or simulation time to synchronize simulations. This simulation approach obviously favors validation of OS elements, allowing also for custom hardware designs to be validated along with their respective device drivers. As such, this approach is tackled in this dissertation, with development being done to support its usage. The downside to this approach is that due to different simulation granularity in both simulations, synchronization may be difficult. If approaches are used which involve synchronization with other simulation tools that simulate other domains, such as physical systems or analog electronics, synchronization may be even impossible, as the RTL simulator will be always delayed in relation to the software simulator. Absolute simulation times are not as relevant when using only RTL and software and simulation, as the modeling can be done with only the relative transaction times of the RTL simulator being taken into account by the software simulator. However, if RTL simulation contains hardware designs that react to other simulator's stimuli, it will almost always be behind the software simulator's simulation time.

2.5.4 Full System Software Simulation

In this approach, hardware is only emulated functionally by the software emulator, with the full system being emulated in the software simulator. Figure 2.17 presents a diagram of full system software simulation.

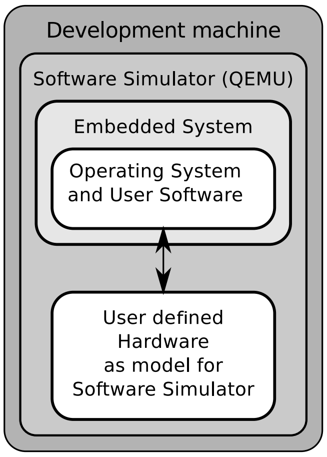

Development machine
Software Simulator (QEMU)
Embedded System
Operating System and User Software
User defined Hardware as model for Software Simulator

Figure 2.17: Full system software simulation diagram(Zabołotny, 2012)

In this approach, hardware models are implemented functionally and integrated into software simulation. This can only be done for open-source simulators by implementing models using the simulator's internal framework and adding them to the list of supported models, or for simulators that allow to load external models as plugins. It may seem undesirable to maintain separate hardware model descriptions, an HDL for hardware synthesis and a functional one to be integrated in software simulation, but it is in fact a very useful feature for design space exploration early in the project. By using this approach, device driver development and validation may start before there is any commitment to an HDL implementation, allowing not only for a flexibility that suits design space exploration very well, but also concurrent development given that software design teams may start device driver development concurrently with hardware design teams. For these reasons, this approach is also tackled in this dissertation with development being done to facilitate its usage.

2.6 QEMU

QEMU (which stands for **Q**uick **EMU**lator) is a powerful open source machine emulator and virtualizer. It dynamically translates the guest binary code to host binary code, supporting not only emulation of different guests but also virtualization of the same guests running code natively on the host machine. Given the scope of this dissertation, the focus will be on QEMU being used as an emulator, with its virtualization capabilities being ignored for the most part. QEMU is capable of full-system emulation, emulating functionally system components like interrupt controllers, memories and other peripheral devices. It is a very useful tool in the context of embedded system development, enabling development and debug of a target platform system without a physical target machine. The entire software stack may be developed entirely in QEMU, and then deployed to the target machine once validated and debugged. Other operating features are: Linux Userspace Emulation, that allows for quick validation of simple applications, cross-compiled for architectures other than the host development machine; and Virtualizion, through KVM and XEN hosting, commonly used in functional hardware emulation, where the guest code is executed on KVM and Xen virtualizers.

When QEMU was first developed by Fabrice Bellard it was a major breakthrough due to its dynamic binary translation algorithms. It was widely adopted as an emulator by companies for development purposes, although each company maintained their internal private versions of QEMU, implementing modifications as suited and implementing support for their machines and platforms. Nokia for instance, used it to develop all of their cellphone products, incorporating graphical interfaces that mimicked their screens. Nowadays, QEMU continues to be very popular, for instance being integrated as part of the Android **S**oftware **D**evelopment **K**it (SDK) and as part of Xilinx's PetaLinux solution as an emulator. Popular virtualization products also draw heavy inspiration on QEMU, such as VirtualBox which uses some of QEMU's virtual hardware devices and bases its dynamic recompiler on QEMU, or Xen-HVM which runs a QEMU device model daemon as a backend to provide **I**nput/**O**utput (I/O) virtualization.

Due to its open source nature, tremendous possibilities are opened up extension-wise. For these reasons, one of the main focus of this dissertation is QEMU as an embedded system validation tool, with development efforts to extend QEMU's features to perform simulations in hardware accelerated embedded systems using full-system software simulation and RTL-software co-simulation approaches.

QEMU is poorly documented program as far as development goes, making source code analysis mandatory. Documentation is sparse and almost non-existent, and source code comments are frequently outdated. The main communication vehicle between QEMU developers is QEMU's development mailing list, which makes searching for old e-mails and reading the conversations the best solution to get a quick insight on the internals of QEMU (QEMU Development Documentation). Given the low amount of available QEMU literature and information, it is relevant to provide an insight into its internal architecture, with special focus on functional hardware emulation, in order to understand the work developed in this dissertation.

2.6.1 Dynamic Binary Translation

QEMU translates target binary code to guest code dynamically as code is discovered, resulting in very fast emulations when compared to other emulation techniques. Instead of emulating each target instruction independently, QEMU translates larger chunks of code, emulating them in a functional fashion. Guest code blocks are translated dynamically into a single sequence of host instructions that constitute translated blocks and are then cached into a translation cache that is used to speed up similar code blocks found. A block length is established when translating guest code, up to the nearest jump or an instruction that changes the static CPU state in a way that cannot be deduced at translation time. If the change of control flow, for instance a conditional jump, results in a virtual **P**rogram

Counter (PC) address of an already translated block that is stored in cache, that code gets executed directly without the need of guest code translation overhead. For this reason, cached blocks are indexed using their guest virtual addresses, so they can be found easily using the virtual PC value, and are purged every time they fill up (Liaw, 2014). Figure 2.18 presents a diagram of the translation and caching process.

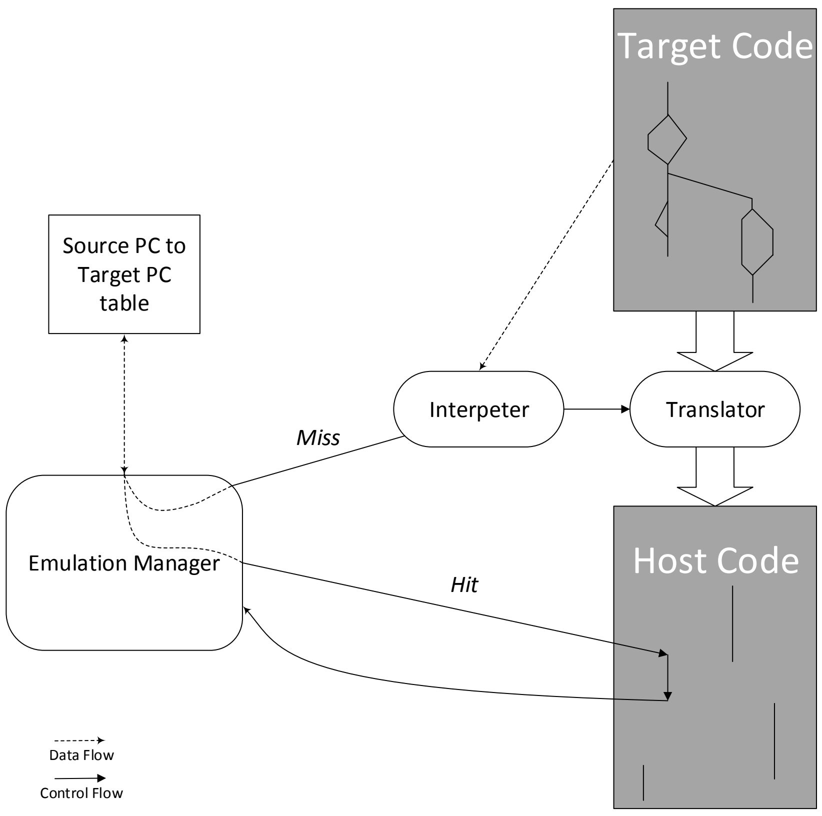


Figure 2.18: Dynamic translation and caching diagram

This simulation technique allows for very fast emulations when compared to techniques that employ simulation interpreters that translate and execute one instruction at a time. Further optimizations to this technique are used that allow block chaining. When executing translated blocks, a prologue and epilogue are used to switch contexts between the main QEMU loop and the translated execution environment. When a translated block returns to the main loop and the next block is known and already translated, QEMU can patch the original block to jump directly into the next block instead of jumping to the epilogue (Marazakis). When this is done on several consecutive blocks, the blocks will form chains and loops, allowing for QEMU to emulate tight loops without running any extra code in between. Figure 2.19 presents a diagram of QEMU's block chaining of translated blocks in cache.

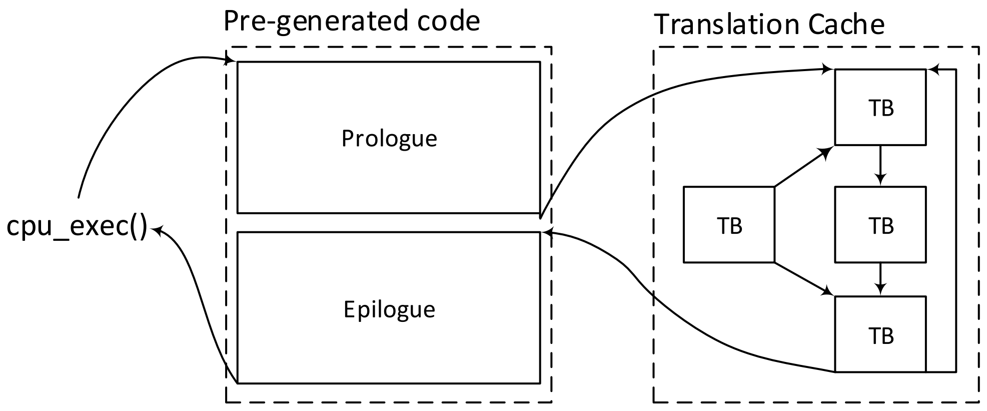


Figure 2.19: Block chaining diagram

Up to version 0.9.1, binary translation was carried out by an entity called DynGen. Target code blocks were converted to C code by QEMU's DynGen and subsequently compiled to the host architecture using **G**NU **C**ompiler **C**ollection (GCC). The issue with this procedure was that QEMU was highly dependent on GCC, which brought some issues as new GCC versions started to come out. To remove the tight coupling of the translator to GCC a new approach was tackled: the **T**iny **C**ode **G**enerator (TCG) Wen-Ren (2011). The TCG aims to remove the shortcomings of relying on a particular version of GCC or any compiler for that matter, instead incorporating the TCG compiler into other tasks performed by QEMU at run time. The whole translation task thus consists of two parts: blocks of target code being rewritten in TCG operations - a machine-independent intermediate representation, and subsequently this notation compiled for the host architecture. TCG requires not only frontends and backends to be written when supporting new architectures, but also that the target instruction translation be rewritten to take advantage of TCG ops. Beyond QEMU version 0.10.0, TCG is integrant part of the stable releases.

2.6.2 QEMU Monitor

When QEMU is running, it provides a monitor console for interaction with simulation. Through various commands, the monitor allows you to inspect the running guest OS, change removable media and USB devices, take screenshots and audio grabs, and control various aspects of the virtual machine. The monitor is accessed from within QEMU by holding down the 'Ctrl' and 'A' keys, and pressing 'C'. Once in the monitor, the same procedure switches back to the guest OS. Typing help or '?' in the monitor brings up a list of all commands. Alternatively the monitor can be redirected to using the *'-monitor <dev>'* command line option. Passing the *'-monitor stdio'* parameter, at boot initialization, will redirect the monitor to the standard output. QEMU monitor provides a useful command, 'info', which lists several parameters useful to inspect the hardware environment that was instantiated, along with its properties, such as device addresses or interrupt lines. This information proved to be very useful throughout the development of the functionalities ans extensions proposed by this dissertation. The following is an example of the that command along with its respective output for the QEMU ARM Versatile Express for Cortex -A9 machine, trimmed to the last three listed devices:

```
$(qemu) info qtree
bus: main-system-bus
 type System
 dev: arm.cortex-a9-global-timer, id ""
   gpio-out "sysbus-irq" 1
   num-cpu = 1 (0x1)
   mmio ffffffffffffffff/0000000000000020
 dev: arm_gic, id ""
   gpio-in "" 96
   num-cpu = 1 (0x1)
   num-irq = 96 (0x60)
   revision = 1 (0x1)
   mmio ffffffffffffffff/0000000000001000
 dev: a9-scu, id ""
   num-cpu = 1 (0x1)
   mmio ffffffffffffffff/0000000000000100
```

2.6.3 QEMU Platforms

In QEMU, platforms are instantiated statically. That is, there must be source code for each of the supported platforms with hardware construction and mapping API invocation. The following command is an example of the options used to print a list of supported platforms (that is left column for option argument and right column with its corresponding description) for the **A**dvanced **R**ISC **M**achine (ARM) architecture QEMU binary:

```
$ qemu-system-arm -machine ?
Supported machines are:
akita            Akita PDA (PXA270)
borzoi           Borzoi PDA (PXA270)
canon-a1100        Canon PowerShot A1100 IS
cheetah          Palm Tungsten|E aka. Cheetah PDA (
   OMAP310)
collie           Collie PDA (SA-1110)
connex           Gumstix Connex (PXA255)
cubieboard        cubietech cubieboard
highbank          Calxeda Highbank (ECX-1000)
integratorcp       ARM Integrator/CP (ARM926EJ-S)
kzm             ARM KZM Emulation Baseboard (ARM1136)
lm3s6965evb        Stellaris LM3S6965EVB
lm3s811evb        Stellaris LM3S811EVB
mainstone         Mainstone II (PXA27x)
midway           Calxeda Midway (ECX-2000)
musicpal          Marvell 88w8618 / MusicPal (ARM926EJ-S)
n800            Nokia N800 tablet aka. RX-34 (OMAP2420)
n810            Nokia N810 tablet aka. RX-44 (OMAP2420)
none            empty machine
nuri            Samsung NURI board (Exynos4210)
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 for
   Cortex-A9
smdkc210          Samsung SMDKC210 board (Exynos4210)
spitz            Spitz PDA (PXA270)
sx1             Siemens SX1 (OMAP310) V2
sx1-v1           Siemens SX1 (OMAP310) V1
```

```
terrier          Terrier PDA (PXA270)
tosa            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            ARM Virtual Machine
xilinx-zynq-a9      Xilinx Zynq Platform Baseboard for
   Cortex-A9
z2              Zipit Z2 (PXA27x)
```

Appendix B, section B.1 contains similar QEMU generated outputs for all QEMU architectures. It is not uncommon that developers add their own machines' support, provided that the platform's CPU is supported in QEMU's binary translation, as it is relatively simple to use the construction and mapping API to instantiate and map the hardware respectively. Section B.2 shows an example of a machine source code file.

One cannot ignore that given the necessary hardware models' support, machine instantiation procedure can be improved by dynamically parsing a platform description, and then instantiating and mapping the hardware accordingly. An example of this dynamic procedure can be found in Petalogix QEMU support, regarding Xilinx's Microblaze soft-core architecture with the FDT generated machine. Instead of running static machine initialization code, QEMU parses a target device tree and instantiates hardware accordingly, thus enabling new support to target boards to be generated on-the-fly by means of DTB file. However, there may be some issues to be solved, as it's not pushed to upstream QEMU yet.

Due to the open-source nature of QEMU, its behavior is inconsistent throughout its multiple binaries (among different architectures), so some tweaking on the arguments may be needed so it can run without breaking. As of version 2.4, there were over 149 options, with 14 being deprecated, and 2 being used internally (Armbruster, 2015).

Buildroot currently provides a feature to aid a quick QEMU bring-up, by supporting default configuration for QEMU machines, and respective QEMU boot commands in a text file. Appendix A.1.4 details this feature.

2.6.4 QEMU Device Model

QEMU emulates the machine's hardware functionally, allowing full-system simulation. Figure 2.20 presents a simplified system stack where an embedded application, running Linux on an embedded platform is being emulated by QEMU. In the same Figure, system devices are also represented as QEMU devices attached to System Bus.

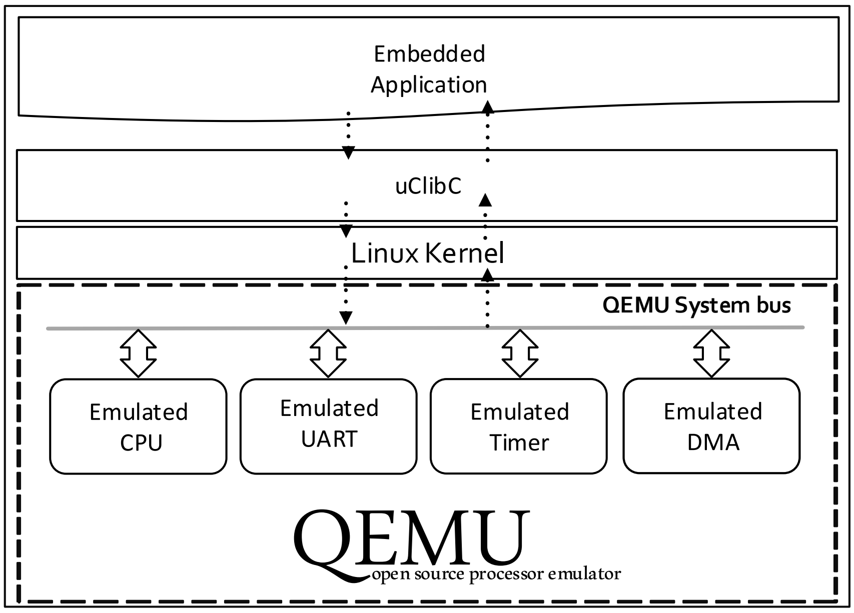


Figure 2.20: QEMU emulation overview Diagram

QEMU translates and runs the whole software stack, including the Linux kernel, the C-library and the embedded application that's running on top of the operating system. Each time hardware is accessed, QEMU will call a set of functions and routines that will emulate hardware inherent behavior. Device modeling code is called as response to the kernel's device drivers access through the system bus.

System bus transaction latencies and bus-specific behaviors are not modeled. Instead, QEMU keeps a list of the devices associated with a bus and their respective addresses. Whenever an operation associated with a system bus write or read is detected, functions that emulate the respective device's read or write transactions will be called.

Currently, QEMU's device model follows an object oriented paradigm. Each device is an object, with each kind of device being a class. For instance a machine may have multiple instances of the same **U**niversal **A**synchronous **R**eceiver **T**ransmitter (UART) device model mapped into different addresses, which is achieved by instantiating multiple objects of a class that implements that UART model. Device models may be instantiated statically, that is, their instantiations are contained in machine initialization code, or instantiated at runtime, by using command line arguments when launching QEMU.

To instantiate a given device model, it is necessary that the correspondent class for that device be present in QEMU's source files at compile time. This proves to be a major drawback for QEMU integration on custom hardware design flows, as it is necessary to be aware of the QEMU internals and also, a new recompilation iteration is needed every time a device model is added or changed.

A SysBusDevice is a class in the QEMU source code that presents a device mapped to the machine system bus. Specific devices like UART or timers, when mapped to the system bus, are implemented as a child class of SysBusDevice. Figure 2.21 presents the SysBusDevice class in QEMU. For the sake of an understandable diagram, some of the more low level implementation features are omitted. Also, due to incoherent semantics (for instance DeviceClass and SysBusClass are used for static data structures, DeviceState and SysBusDevice are used for object data structures; sysbus_ suffix for SysBusDevice member fuctions, qdev_ suffix for DeviceState member functions) alternate semantics were used for some classes or their respective members.

The MemoryRegion class is used to provide an API to register the device in a specific memory region. MemoryRegion has several constructors, with the two most important ones being represented in the **U**nified **M**odeling **L**anguage (UML), the first for a memory block, and the second for an I/O-mapped device. Each device is responsible for transaction behavior implementation, and as such, each SysBusDevice is associated with a function that implements read transactions, and a function that implements write transactions. These functions are registered as callbacks in MemoryRegionOps structure present in a MemoryRegion. Alternatively devices can be accessed as port-mapped I/O. As memory-mapped I/O is the tendency nowadays, and suits well the requirements of this dissertation, the port-mapping technique was discarded.

Concerning the IRQ API, input IRQs are used for devices that handle interrupt lines as inputs, for instance interrupt controllers or CPUs, whereas output IRQs

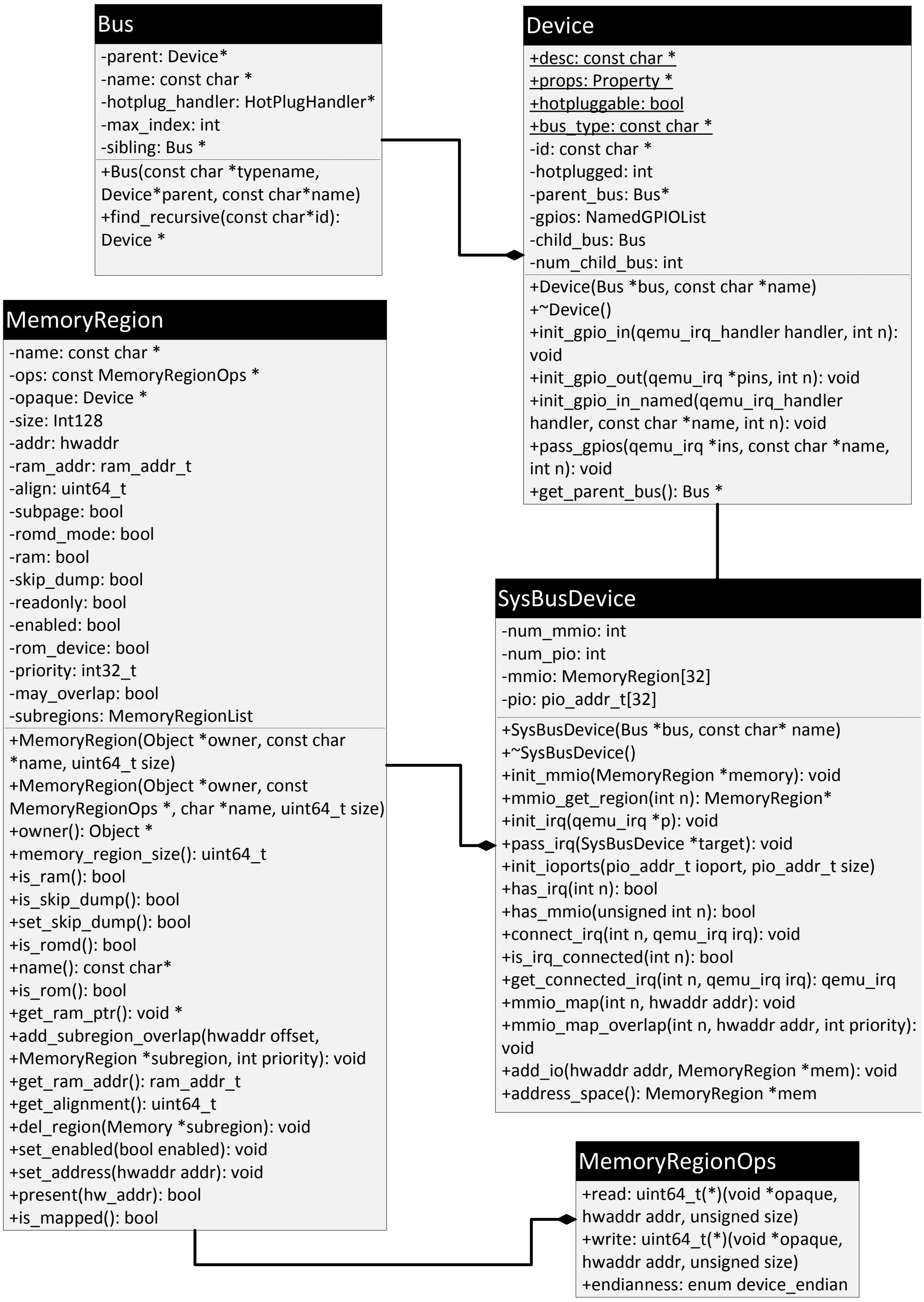

Bus
-parent: Device*
-name: const char *
-hotplug_handler: HotPlugHandler*
-max_index: int
-sibling: Bus *
+Bus(const char *typename, Device*parent, const char*name)
+find_recursive(const char*id): Device *
Device
+desc: const char *
+props: Property *
+hotpluggable: bool
+bus_type: const char *
-id: const char *
-hotplugged: int
-parent_bus: Bus*
-gpios: NamedGPIOList
-child_bus: Bus
-num_child_bus: int
+Device(Bus *bus, const char *name)
+~Device()
+init_gpio_in(qemu_irq_handler handler, int n): void
+init_gpio_out(qemu_irq *pins, int n): void
+init_gpio_in_named(qemu_irq_handler handler, const char *name, int n): void
+pass_gpios(qemu_irq *ins, const char *name, int n): void
+get_parent_bus(): Bus *
MemoryRegion
-name: const char *
-ops: const MemoryRegionOps *
-opaque: Device *
-size: Int128
-addr: hwaddr
-ram_addr: ram_addr_t
-align: uint64_t
-subpage: bool
-romd_mode: bool
-ram: bool
-skip_dump: bool
-readonly: bool
-enabled: bool
-rom_device: bool
-priority: int32_t
-may_overlap: bool
-subregions: MemoryRegionList
+MemoryRegion(Object *owner, const char *name, uint64_t size)
+MemoryRegion(Object *owner, const MemoryRegionOps *, char *name, uint64_t size)
+owner(): Object *
+memory_region_size(): uint64_t
+is_ram(): bool
+is_skip_dump(): bool
+set_skip_dump(): bool
+is_romd(): bool
+name(): const char*
+is_rom(): bool
+get_ram_ptr(): void *
+add_subregion_overlap(hwaddr offset,
+MemoryRegion *subregion, int priority): void
+get_ram_addr(): ram_addr_t
+get_alignment(): uint64_t
+del_region(Memory *subregion): void
+set_enabled(bool enabled): void
+set_address(hwaddr addr): void
+present(hw_addr): bool
+is_mapped(): bool
SysBusDevice
-num_mmio: int
-num_pio: int
-mmio: MemoryRegion[32]
-pio: pio_addr_t[32]
+SysBusDevice(Bus *bus, const char* name)
+~SysBusDevice()
+init_mmio(MemoryRegion *memory): void
+mmio_get_region(int n): MemoryRegion*
+init_irq(qemu_irq *p): void
+pass_irq(SysBusDevice *target): void
+init_ioports(pio_addr_t ioport, pio_addr_t size)
+has_irq(int n): bool
+has_mmio(unsigned int n): bool
+connect_irq(int n, qemu_irq irq): void
+is_irq_connected(int n): bool
+get_connected_irq(int n, qemu_irq irq): qemu_irq
+mmio_map(int n, hwaddr addr): void
+mmio_map_overlap(int n, hwaddr addr, int priority): void
+add_io(hwaddr addr, MemoryRegion *mem): void
+address_space(): MemoryRegion *mem
MemoryRegionOps
+read: uint64_t(*)(void *opaque, hwaddr addr, unsigned size)
+write: uint64_t(*)(void *opaque, hwaddr addr, unsigned size)
+endianness: enum device_endian

Figure 2.21: QEMU SysBusDevice UML

are used to model devices that generate interrupt lines. IRQ objects are contained in a list of input and output IRQ objects in the Device class. Typically, at machine initialization, an IRQ object array is used to connect device IRQ objects to CPU objects, connecting output IRQ objects to input IRQ objects respectively.

A device can also have a list printable attributes, that are commonly used for QEMU Monitor debugging purposes. A set of macros to manipulate these properties is provided as an API in QEMU.

Appendix B.3 provides an LED device example, to resume the API usage in a simplistic model of a memory-mapped device.

2.6.5 Time in QEMU

QEMU timers provide a way of calling a routine after a predetermined time interval. It is an important feature, useful in hardware behavior modeling, as some devices must be time-aware. Three types of clock sources for QEMU timers are available: Virtual, Host and Real-time. Most devices use the Virtual Clock as source for their timers while modeling time-aware behavior, since the Virtual Clock only runs when the machine is being emulated. The Host Clock, runs even when the virtual machine is stopped and provides a resolution of 1000 Hz. Similarly, Realtime Clock, runs when the virtual machine is stopped, but is sensitive to time changes to the host's system clock.

Despite the fact that Virtual clock only ticks during the machine execution, the amount of time elapsed is not accurate, even when the same code block is being emulated, as result of virtual machine non-deterministic implementation.

Machine behavior is implemented following the usual multi-thread programing models that use contention mechanisms to avoid concurrency, thus undermining the host machine determinism and consequently Virtual clock count. This also raises similar problems on device implementation, for instance, if a device transaction is expected to be be quick, but the device behavior code was translated to long and non-deterministic implementation, the result will be a longer execution time due to emulation block and Virtual clock still counting.

To implement deterministic machine emulation, a patch was submitted that implements an 'icount' option, which changes how Virtual Clock increments its value. With each virtual CPU instruction, Virtual Timer increments based on a scale factor that presents the machine's theoretical frequency. With this feature, machine emulation code may be non-deterministic, as the way behavior is implemented does not directly affect simulation time steps. This provides greater freedom while modeling hardware devices, as the implementation will not affect the Virtual Clock directly. The following command is an example of its usage:

```
$ qemu-system-[arch] [flags] -icount [N]
```

The 'N' parameter is used for configuring the instruction counter. It will configure how many clock cycles does a single instruction take, with clock cycles per instruction = 2^N.

2.7 Verilog Programming Language Interface

Most FPGA design flows are heavily conditioned by the FPGA vendors' **I**ntegrated **D**evelopment **E**nvironments (IDE) and imposed workflows. Although it is possible to use open-source tools to a certain extent, it is relevant to adopt standard HDL interfacing mechanisms when implementing HDL simulation interfaces, as not only to be independent of simulation tools, but also be able to use commercial simulation tools that are supported by the the main vendors IDE.

Given that Verilog is the HDL of choice in this dissertation, focus will be placed on standard Verilog simulation interfaces. One of the major strengths of the Verilog language is the **P**rogramming **L**anguage **I**nterface (PLI), which allows users and Verilog application developers to infinitely extend the capabilities of the Verilog language and the Verilog simulator. In fact, the overwhelming success of the Verilog language can be partly attributed to the existence of its PLI (Sutherland, 2002). Given that the **V**erilog **P**rocedural **I**nterface (VPI) set of routines (often referred to as PLI 2.0) cover what was provided by **T**ask/**F**unction (TF) and **ACC**ess (ACC) routines(PLI 1.0), VPI routines render TF and ACC routines obsolete. As such, all PLI mentions in this document will refer to the VPI set of routines.

Due to Verilog being a weakly typed language and not having many constructs that support validation, system verification is not so easy to do, unlike for instance VHDL. To suppress these limitations that are intrinsic to the language, SystemVerilog was developed. SystemVerilog is an extension to the Verilog language that favors system level abstraction, introducing new data types and overall features that improve system level design, and system validation.

The learning curve related to the Verilog PLI is somewhat steep, as it includes understanding the PLI API and workflow, and knowing how to compile and link a PLI application to a specific simulator. To enable HDL designers C-function invocation from HDL validation code, SystemVerilog introduces the **D**irect **P**rogramming **I**nterface (DPI). DPI allows for C code invocation without complex system task definitions, with different kinds of routines for each system task, and other PLI associated implementation. The downsides to the DPI are that it doesn't provide a way to interact with simulation data structures, or any other complex interactions like synchronization with time or value changes. For these reasons, it doesn't cover all the features provided by PLI to replace it, but is nevertheless is a quick alternative for validation code integration. Consequently, PLI was adopted in this dissertation to develop simulation extensions that support co-simulation approaches with QEMU making use of the possibilities of PLI.

2.7.1 PLI Overview

A PLI application is a user-defined C language application which can be executed by a Verilog simulator (Sutherland, 2002). The PLI application can interact with the simulation in a myriad of ways, from reading and modifying simulation logic values, to performing certain actions in a specific simulation moment, such as beginning or end of simulation.

One way to start interaction with the simulation is using user-defined system tasks/functions. In the Verilog language, a system task or a system function is a command which is executed by a Verilog simulator. The name of a system task or a system function begins with a dollar sign ($) (Sutherland, 2002). They are invoked as Verilog statements, with the main difference between a system task and a system function being that a system function provides a return value, whereas a system task does not.

The following is an example of a system task that prints a decimal value in the simulator's command line output at each positive edge of a clock, not unlike a printf interface:

```
always @(posedge clock)
        $display("value = %d", value);
```

The following is an example of a system function that generates a random value at each positive edge of a clock:

```
always @(posedge clock)
        vector <= $random();
```

There are 3 kinds of system tasks/functions:

- Standard built-in system tasks/system functions;

  These are defined as part of the IEEE standard, such as $display, $random, and $finish. All IEEE compliant Verilog simulators will have these standard system tasks and system functions (Sutherland, 2002).

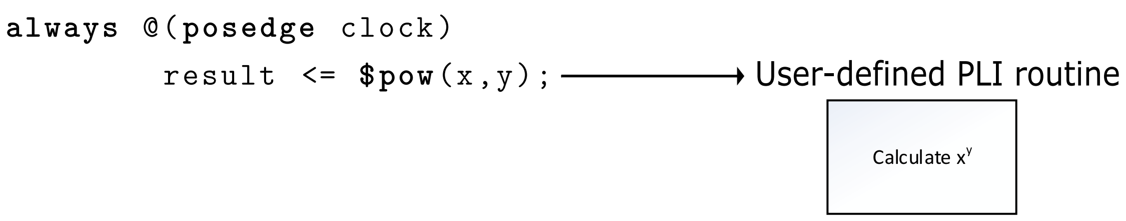
- Simulator specific built-in system tasks/functions;

  These are proprietary, which are defined as part of a simulator and may not exist in other simulators (Sutherland, 2002).

- User-defined system tasks/functions.

  These are created through PLI. Using PLI, the name and functionality of the system tasks/functions are specified (Sutherland, 2002).

User-defined system tasks/functions are associated with a PLI application. When a Verilog simulator encounters a system task/function name, it will execute the PLI application code associated with the name (Sutherland, 2002).



```
always @(posedge clock)
        result <= $pow(x,y);
```

Figure 2.22: User-defined system function

Another other way is to register callback routines, which are to be ran at specific moments during simulation, such as when a simulation value changes.

2.7.2 PLI Routines

The VPI portion of PLI standard defines several types of PLI routines. The type of the routine determines when the simulator will execute the routine. Some types of routines are runtime routines, which are called during simulation, and some types are elaboration or linking time routines, which are called prior to simulation. The types of PLI routines are:

- Calltf routines;
- Compiletf routines;
- Sizetf routines;
- Simulation callback routines.

Calltf, compiletf and sizetf routines are directly associated with system tasks/-functions, with calltf routines being called at runtime and compiletf and sizetf being called at compile time. Simulation callback routines are routines which are registered for a kind of event, and are called each time that the event occurs.

2.7.3 Calltf Routines

Calltf is a routine that is called at runtime, every time that a system task/function is called during simulation. Its purpose is to implement system task/function behavior only, with verification responsibilities belonging to compiletf routines.

Figure 2.23 presents a calltf routine of a system function that calculates the power of 'x' by 'y':

Appendix C.1.1 contains C code for the power calltf routine.

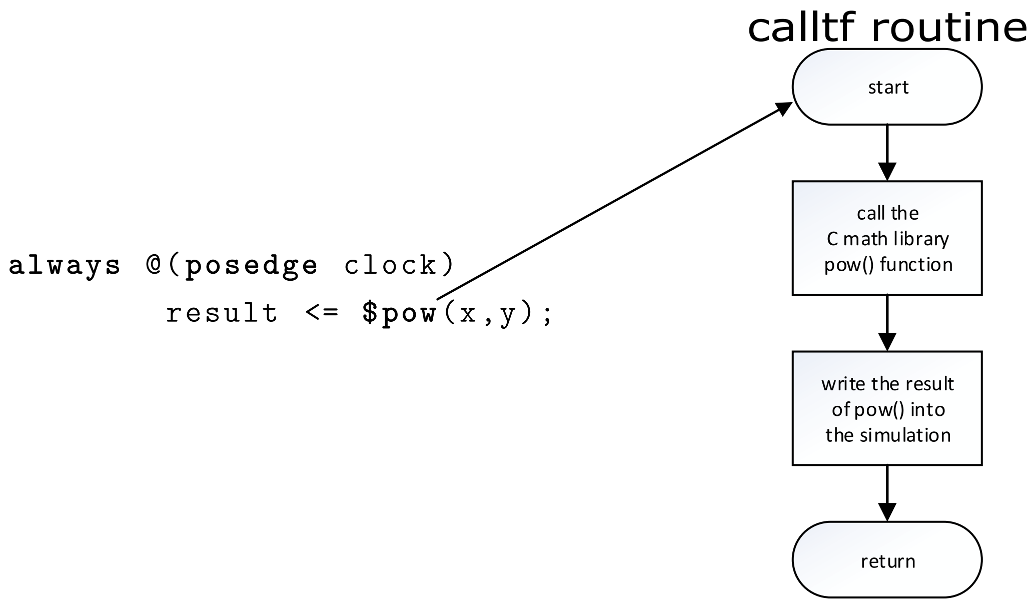

calltf routine
start
call the C math library pow() function
write the result of pow() into the simulation
return
always @(posedge clock)
result <= $pow(x,y);

Figure 2.23: Power calltf routine

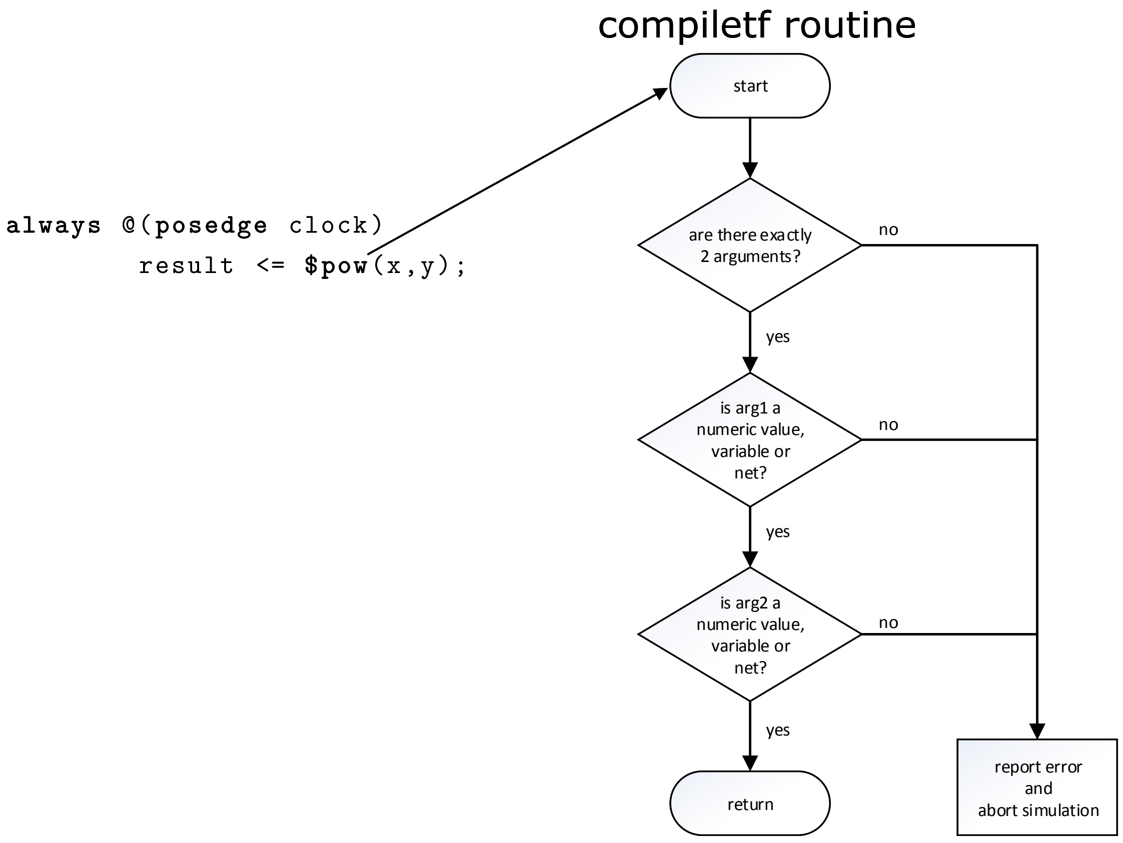
2.7.4 Compiletf Routines

Compiletf is a routine that is called at compile time, to check if a system task/-function is being used correctly(that is, if it was called with the correct number of arguments, argument types...). It is called only once for each system task/function call, by the simulator's compiler when the simulator loads and prepares its simulation data structure, being ran before simulation time 0. For instance, if one system task/function is called three times in the Verilog source code, compiletf will be called three times. That is, it will be called only once for each source code invocation.

Given that at runtime calltf may be called very frequently, performing validation on this routine may severely affect simulation performance. Compiletf routine's main purpose, as it should be clear by now, is to offload computation from the calltf routine, thus improving simulation performance.

Figure 2.24 presents a compiletf routine of a system function that calculates the power of 'x' by 'y':

Appendix C.1.2 contains C code for the power compiletf routine.


```
always @(posedge clock)
        result <= $pow(x,y);
```

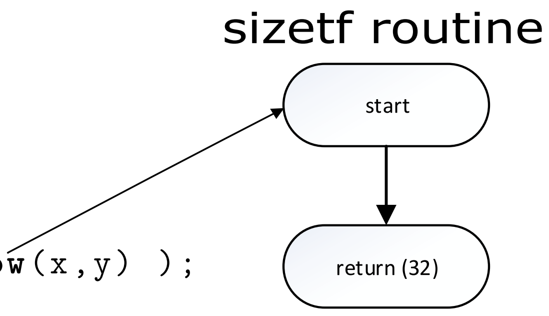
compiletf routine
start
are there exactly 2 arguments?
no
yes
is arg1 a numeric value, variable or net?
no
yes
is arg2 a numeric value, variable or net?
no
yes
return
report error and abort simulation

Figure 2.24: Power compiletf routine

2.7.5 Sizetf Routines

Sizetf routines are also called at compile time, before simulation time 0. These routines are used to determine the return size, in bits, of a system function. Unlike compiletf, even if the system function is called multiple times in Verilog source code, it is only called once. The return value of the first call is used to determine the return size of all instances of the system function.

Figure 2.25 presents a sizetf routine of a system function that calculates the power of 'x' by 'y':



```
always @(posedge clock)
        result <= (in + $pow(x,y) );
```

Figure 2.25: Power sizetf routine

Appendix C.1.3 contains C code for the power sizetf routine.

2.7.6 Simulation Callback Routines

The VPI provides means for PLI applications to be called for specific simulation events. The VPI portion of the PLI standard refers to these types of routines as simulation callback routines (Sutherland, 2002). Some examples of simulation callbacks are:

- Start of simulation(just before simulation time 0);
- End of Simulation;
- Change of value of a given net or register;
- Execution of a Verilog procedural statement.

A common usage of simulation callback routines is to perform tasks at the very beginning and the very end of a simulation (Sutherland, 2002).

Figure 2.26 presents a routine for start of simulation callback, that prints a start of simulation message on the simulator's console. The routine is called before simulation time 0.

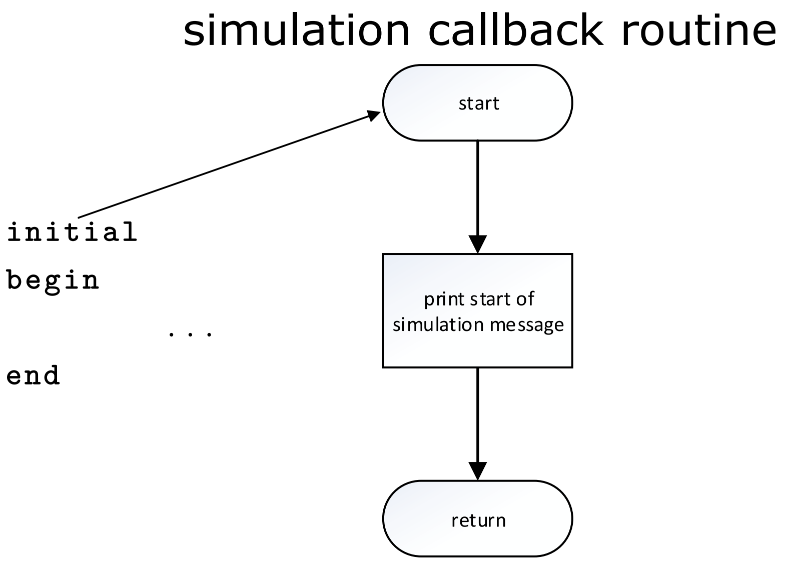

simulation callback routine
initial
begin
...
end
start
print start of simulation message
return

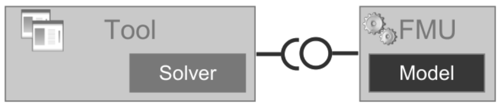
Figure 2.26: Simulation callback routine

Appendix C.2.1 contains C code for this routine.

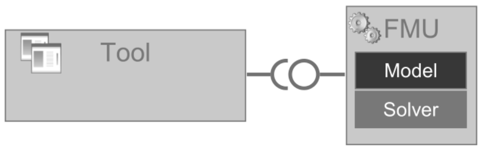
2.8 Functional Mock-up Interface

Functional **M**ock-up **I**nterface (FMI) is a tool independent standard to support both model exchange and co-simulation of dynamic models using a combination of .xml files and C-code (either compiled in DLL/shared objects or in source code). The first version, FMI 1.0, was published in 2010. The FMI development was initiated by Daimler AG with the goal to improve the exchange of simulation models between suppliers and **O**riginal **E**quipment **Ma**nufacturers (OEM). As of today, development of the standard continues through the participation of 16 companies and research institutes. FMI 1.0 is supported by over 45 tools and is used by automotive and non-automotive organizations throughout Europe, Asia and North America (Otter et al., 2013).

The FMI defines a component which implements simulation interfaces, called **F**unctional **M**ock-up **U**nit (FMU). An FMU contains an e**X**tensible **M**arkup **L**anguague (XML) file with description of the interface, and source code or dynamic library which implements the interface. Source code is used if target platform independence is desired, however the latter solution is used whenever the suppliers want to hide the source code to secure model or tool know-how. The FMI standard provides two main interfaces: Model Exchange, and Co-simulation. Figures 2.27a and 2.27b represent the two interface scenarios.



(a) FMI model exchange diagram (Otter et al., 2013)



(b) FMI co-simulation diagram (Otter et al., 2013)

Figure 2.27: FMI simulation standards

2.8.1 FMI for Model Exchange

The FMI for Model Exchange interface defines an interface to the model of a dynamic system described by differential, algebraic and discrete-time equations and to provide an interface to evaluate these equations as needed in different simulation environments, as well as in embedded control systems, with explicit or implicit integrators and fixed or variable step-size. The interface is designed to allow the description of large models (Otter et al., 2013).

In this interface, an FMU's source code or compiled code only contains implementation specific to the model, with its respective interpretation and solving implementation responsibilities belonging to the simulation tool. Model Exchange FMUs may be connected together to form larger model subsystems. Figure 2.28 presents Model Exchange FMUs connected together, with outputs and inputs being connected between them to form a larger model subsystem. Figure 2.29 presents the model signals of an FMU.

The arrows represent inputs provided to the FMU, and outputs provided by the FMU. Exposed variables could be used for plotting purposes, and internal model state monitoring, whereas inputs and outputs are used for model integration in the overall simulation. The following excerpt represents an example of using the FMI provided API to interact with an FMU.

```
// Set input arguments
fmiSetTime(m, time);
```

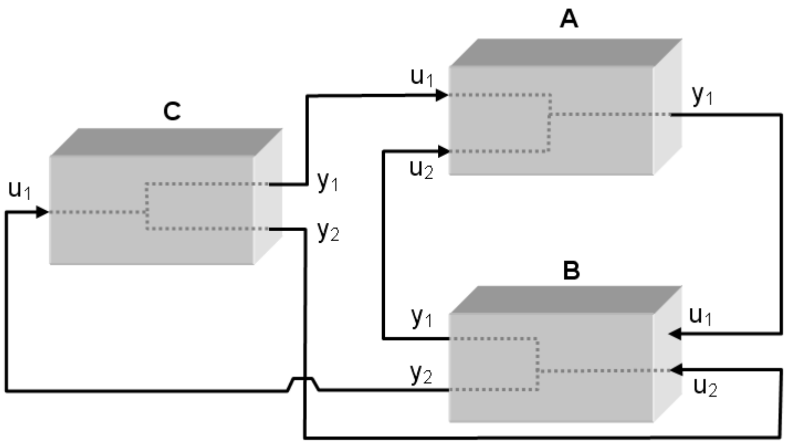


Figure 2.28: Model exchange FMU interconnection (Otter et al., 2013)

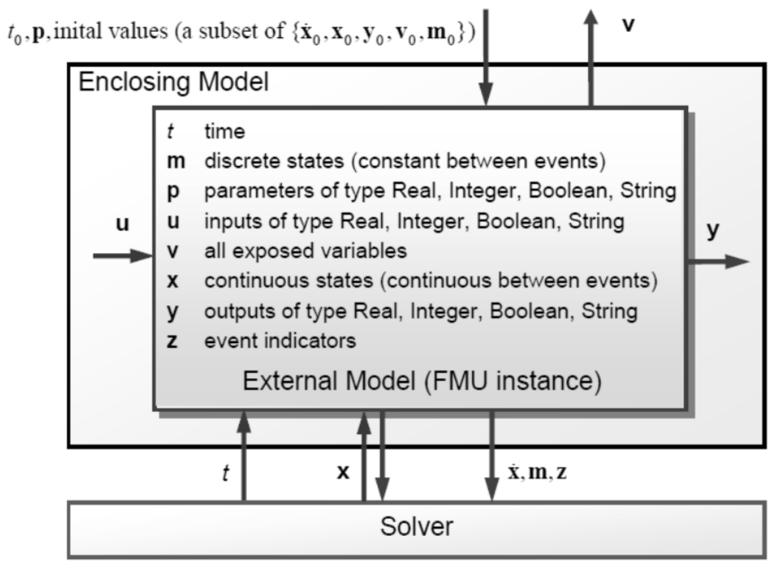
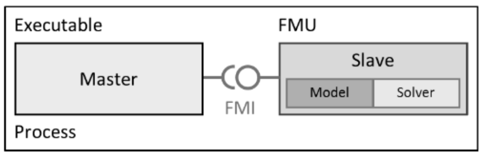

t_0, **p**, inital values (a subset of $\{\dot{\mathbf{x}}_0, \mathbf{x}_0, \mathbf{y}_0, \mathbf{v}_0, \mathbf{m}_0\}$)
v
Enclosing Model
t time
m discrete states (constant between events)
p parameters of type Real, Integer, Boolean, String
u inputs of type Real, Integer, Boolean, String
v all exposed variables
x continuous states (continuous between events)
y outputs of type Real, Integer, Boolean, String
z event indicators
External Model (FMU instance)
u
y
t
x
$\dot{\mathbf{x}}, \mathbf{m}, \mathbf{z}$
Solver

Figure 2.29: Model exchange FMU interface signals (Otter et al., 2013)

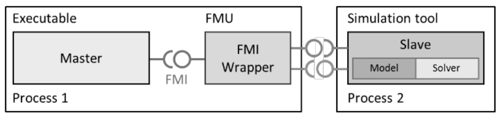
```
fmiSetReal(m, id_u1, u1, nu1);
fmiSetContinuousStates(m, x, nx);
// Get results
fmiGetContinuousStates(m, derx, nx);
fmiGetEventIndicators (m, z, nz);
```

2.8.2 FMI for Co-simulation

The FMI for Co-Simulation interface is designed both for the coupling of simulation tools (simulator coupling, tool coupling), and coupling with subsystem models, which have been exported by their simulators together with their solvers as runnable code. Figures 2.30a and 2.30b represent both scenarios, co-simulation stand alone and co-simulation with tool coupling respectively.



(a) Stand alone co-simulation (Otter et al., 2013)



(b) Tool coupling co-simulation (Otter et al., 2013)

Figure 2.30: FMI co-simulation

In co-simulation stand alone, an FMU contains not only a model, but also solver code exported by another simulation tool to solve the model during simulation. Figure 2.31 represents a co-simulation slave FMU, which contains both model and solver.

In the tool coupling case, the FMU implementation wraps the FMI function calls to API calls provided by the simulation tool, which contains both model and solver. In its most general form, a tool coupling based co-simulation is implemented on distributed hardware with subsystems being handled by different computers, possibly even with different OSs (cluster computer, computer farm, computers at different locations). The data exchange and communication between the subsystems is typically done using one of the network communication technologies (for example **M**essage **P**assing **I**nterface (MPI), **T**ransmission **C**ontrol **P**rotocol/**I**nternet **P**rotocol (TCP/IP)). The definition of this communication layer is not part of the FMI standard. However distributed co-simulation scenarios can be implemented using FMI. Figure 2.32 presents this particular scenario.

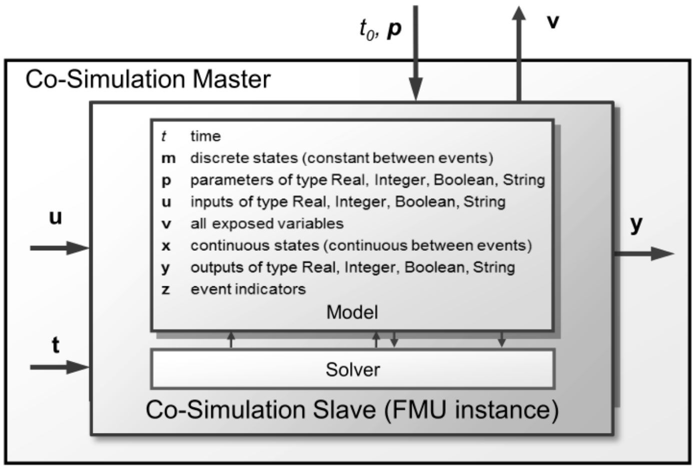

t_0, **p**
v
Co-Simulation Master
t time
m discrete states (constant between events)
p parameters of type Real, Integer, Boolean, String
u inputs of type Real, Integer, Boolean, String
v all exposed variables
x continuous states (continuous between events)
y outputs of type Real, Integer, Boolean, String
z event indicators
Model
u
y
t
Solver
Co-Simulation Slave (FMU instance)

Figure 2.31: Co-Simulation FMU with slave interface signals (Otter et al., 2013)

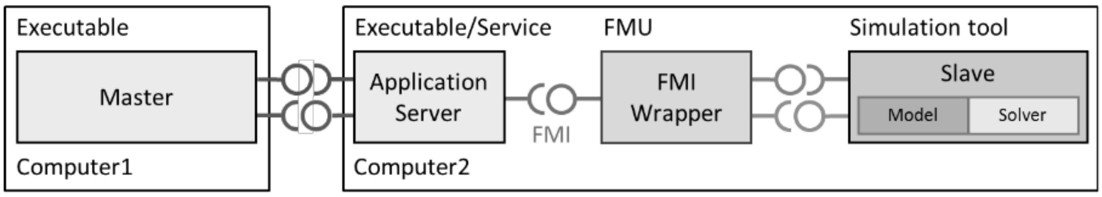


Figure 2.32: FMI distributed co-simulation (Otter et al., 2013)

The master simulation tool has to implement the communication layer. Additional parameters for establishing the network communication (for example identification of the remote computer, port numbers, user account) are to be set via user interface by the master. This data is not transferred via the FMI API. Master algorithms control the data exchange between subsystems and the synchronization of all simulation solvers (slaves). Both, rather simple master algorithms, as well as more sophisticated ones are supported. Note, that the master algorithm itself is not part of the FMI standard (Otter et al., 2013).

2.8.3 FMI in the Industry

The FMI is a fast growing standard, with a wide appeal to the industry. Below are listed some of the principles and property ideas that guided the low-level design of the FMI to make it appealing for industry-wide adaptation. The listed characteristics are sorted, starting from high-level properties to low-level implementation issues.

- Expressivity: The FMI provides the necessary features that Modelica®, Simulink® and SIMPACK® models can be transformed to an FMU (Otter et al., 2013).

- Stability: FMI is expected to be supported by many simulation tools world wide. Implementing such support is a major investment for tool vendors. Stability and backwards compatibility of the FMI has therefore high priority. To support this, the FMI defines 'capability flags' that will be used by future versions of the FMI to extend and improve the FMI in a backwards compatible way, whenever feasible (Otter et al., 2013).

- Implementation: FMUs can be written manually or can be generated automatically from a modeling environment. Existing manually coded models can be transformed manually to a model according to the FMI standard (Otter et al., 2013).

- Processor independence: It is possible to distribute an FMU without knowing the target processor. This allows to run an FMU on a PC, a Hardware-in-the-Loop simulation platform or as part of the controller software of an ECU, e. g. as part of an AUTOSAR SWC. Keeping the FMU independent of the target processor increases the usability of the FMU and is even required by the AUTOSAR software component model. Implementation: using a textual FMU (distribute the C source of the FMU) (Otter et al., 2013).

- Simulator independence: It is possible to compile, link and distribute an FMU without knowing the target simulator. Reason: The standard would be much less attractive otherwise, unnecessarily restricting the later use of an FMU at compile time and forcing users to maintain simulator specific variants of an FMU. Implementation: using a binary FMU. When generating a binary FMU, e. g. a Windows dynamic link library (.dll) or a Linux shared object library (.so), the target operating system and eventually the target processor must be known. However, no run-time libraries, source files or header files of the target simulator are needed to generate the binary FMU. As a result, the binary FMU can be executed by any simulator running on the target platform (provided the necessary licenses are available, if required from the model or from the used run-time libraries) (Otter et al., 2013).7

- Small run-time overhead: Communication between an FMU and a target simulator through the FMI does not introduce significant run time overhead. This is achieved by a new caching technique (to avoid computing the same variables several times) and by exchanging vectors instead of scalar quantities (Otter et al., 2013).

- Small footprint: A compiled FMU (the executable) is small. Reason: An FMU may run on an ECU (Electronic Control Unit, for example a micro processor), and ECUs have strong memory limitations. This is achieved by storing signal attributes (names, units, etc.) and all other static information not needed for model evaluation in a separate text file (= Model Description File) that is not needed on the micro processor where the executable might run (Otter et al., 2013).

- Hide data structure: The FMI for Model Exchange does not prescribe a data structure (a C struct) to represent a model. Reason: the FMI standard shall not unnecessarily restrict or prescribe a certain implementation of FMUs or simulators (whoever holds the model data), to ease implementation by different tool vendors (Otter et al., 2013).

- Support many and nested FMUs: A simulator may run many FMUs in a single simulation run and/or multiple instances of one FMU. The inputs and outputs of these FMUs can be connected with direct feed through. Moreover, an FMU may contain nested FMUs (Otter et al., 2013).

- Numerical Robustness: The FMI standard allows that problems which are numerically critical (for example time and state events, multiple sample rates, stiff problems) can be treated in a robust way (Otter et al., 2013).

- Hide cache: A typical FMU will cache computed results for later reuse. To simplify usage and to reduce error possibilities by a simulator, the caching mechanism is hidden from the usage of the FMU. Reason: First, the FMI should not force an FMU to implement a certain caching policy. Second, this helps to keep the FMI simple. Implementation: The FMI provides explicit methods (called by the FMU environment) for setting properties that invalidate cached data. An FMU that chooses to implement a cache may maintain a set of 'dirty' flags, hidden from the simulator. A get method, e. g. to a state, will then either trigger a computation, or return cached data, depending on the value of these flags (Otter et al., 2013).

- Support numerical solvers: A typical target simulator will use numerical solvers. These solvers require vectors for states, derivatives and zero-crossing functions. The FMU directly fills the values of such vectors provided by the solvers. Reason: minimize execution time. The exposure of these vectors conflicts somewhat with the 'hide data structure' requirement, but the efficiency gain justifies this (Otter et al., 2013).

- Explicit signature: The intended operations, argument types and return values are made explicit in the signature. For example an operator (such as 'compute_derivatives') is not passed as an int argument but a special function is called for this. The 'const' prefix is used for any pointer that should not be changed, including 'const char*' instead of 'char*'. Reason: the correct use of the FMI can be checked at compile time and allows calling of the C code in a C++ environment (which is much stricter on 'const' as C is). This will help to develop FMUs that use the FMI in the intended way (Otter et al., 2013).

- Few functions: The FMI consists of a few, 'orthogonal' functions, avoiding redundant functions that could be defined in terms of others. Reason: This leads to a compact, easy to use, and hence attractive API with a compact documentation (Otter et al., 2013).

- Error handling: All FMI methods use a common set of methods to communicate errors (Otter et al., 2013).

- Allocator must free: All memory (and other resources) allocated by the FMU are freed (released) by the FMU. Likewise, resources allocated by the simulator are released by the simulator. Reason: this helps to prevent memory leaks and runtime errors due to incompatible runtime environments for different components (Otter et al., 2013).

- Immutable strings: All strings passed as arguments or returned are read-only and must not be modified by the receiver. Reason: This eases the reuse of strings (Otter et al., 2013).

- Named list elements: All lists defined in the fmiModelDescription.xsd XML schema file have a String attribute name to a list element. This attribute must be unique with respect to all other name attributes of the same list (Otter et al., 2013).

- Use C: The FMI is encoded using C, not C++. Reason: Avoid problems with compiler and linker dependent behavior. Run FMU on embedded target (Otter et al., 2013).

The following properties are desirable properties to further improve the FMI standard in future versions:

- The FMI for Model Exchange is for ordinary differential equations in state space form (ODE). It is not for a general differential-algebraic equation system. However, algebraic equation systems inside the FMU are supported (for example the FMU can report to the environment to re-run the current step with a smaller step size since a solution could not be found for an algebraic equation system) (Otter et al., 2013).
- Special features as might be useful for multi-body system programs, like SIMPACK, are not included (Otter et al., 2013).
- The interface is for simulation and for embedded systems. Properties that might be additionally needed for trajectory optimization, for example derivatives of the model with respect to parameters during continuous integration, are not included (Otter et al., 2013).
- No explicit definition of the variable hierarchy in the XML file (Otter et al., 2013).
- The number of states and number of event indicators are fixed for an FMU and cannot be changed (Otter et al., 2013).

Several tools already support the FMI standard. Appendix D contains a table of all the tools that currently support the FMI standard.

Chapter 3

System Design

This chapter describes simulation extensions and libraries that assist hardware accelerated embedded system design and were developed in the scope of this dissertation. The main goal is to improve system design, debug, assist to computational offload and validate the final design, still being able to support simulation approaches that aim to reduce time and thus speed-up overall design flow. A hardware accelerated embedded Linux case study was used as stimulus for simulation extension solutions based on QEMU. Figure 3.1 presents an overview diagram of the case study.

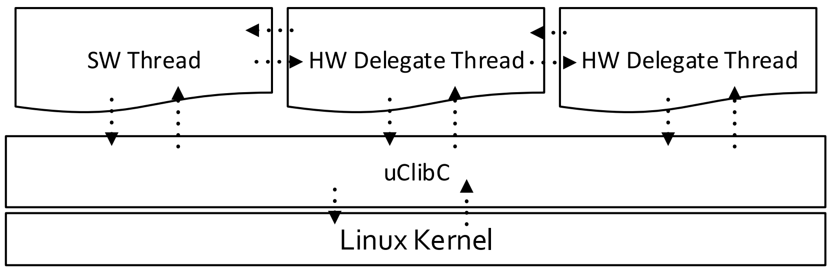


Figure 3.1: Case of study overview

In this scenario, an embedded Linux application is composed by a collection of threads, consisting of software threads and hardware delegate threads. Software threads, tasks that contain data processing, are synchronized with hardware delegate threads, which are threads that delegate data processing responsibilities to hardware accelerators instead of implementing them. These threads act as software representations of hardware accelerators, and provide an interface to custom hardware IP.

To support a hardware software co-design, two kinds of simulation approaches were defined in section 2.5 and adopted, with development being done to expand QEMU's functionality to support these approaches and integrate them into targeted the design flow. Figure 3.2 presents a diagram of the validation phases to the proposed design flow.

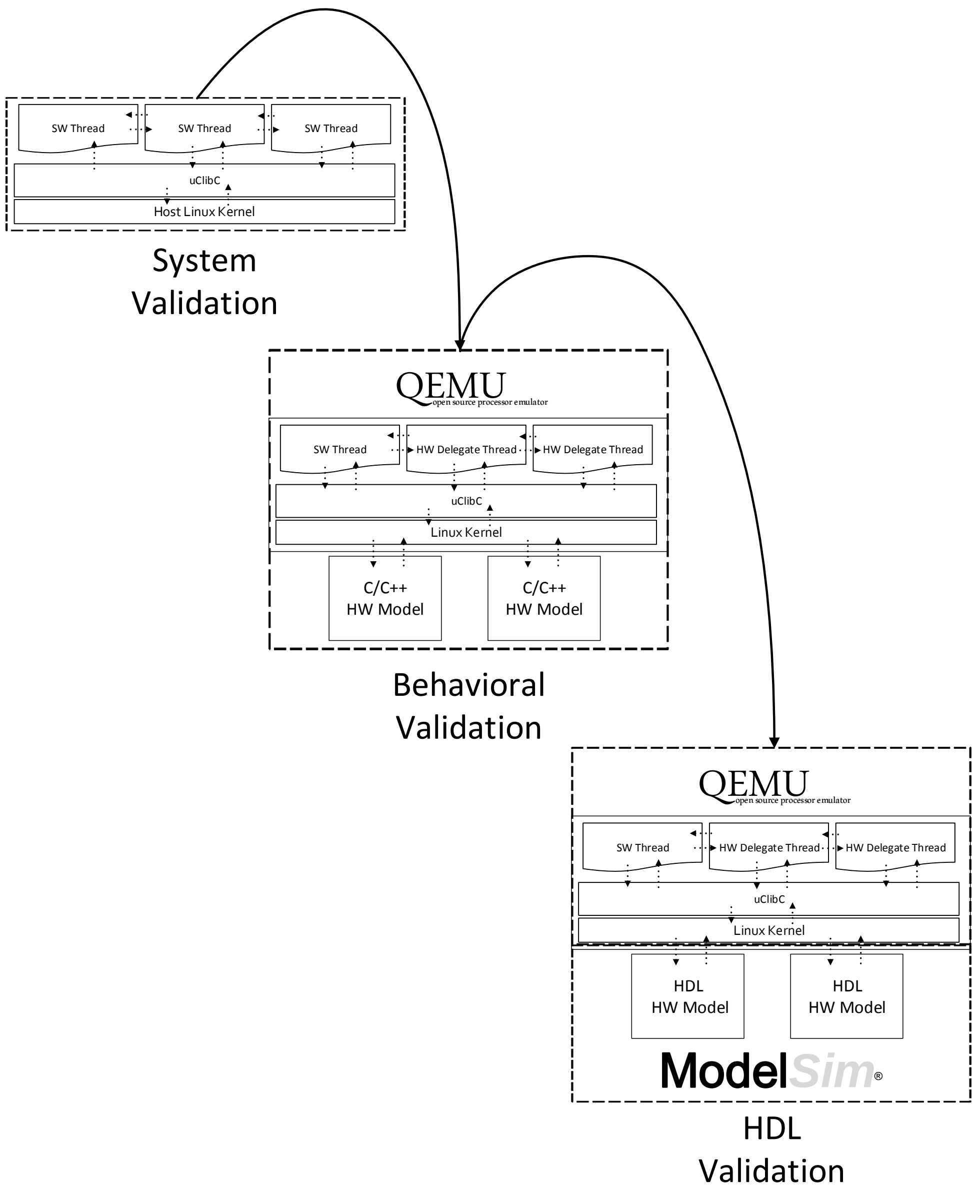


Figure 3.2: Hardware-software validation phases

The first stage of validation occurs during system modeling, and consists in validation of the software application to the desired system. During this phase, the application is running directly on the host OS. The second stage of validation occurs during the hardware design phase, when possible solutions are being explored as far as hardware acceleration is concerned. The system model software application is modified, with delegate threads replacing software threads that are to be migrated to hardware. Validation of the target system is done using QEMU, with behavioral C/C++ models for the migrated hardware IP being integrated into the emulated machine. After a hardware accelerated system design has been chosen and validated, the behavioral C/C++ models are implemented using HDL, with subsequent replacement on the emulated machine. This is achieved by coupling QEMU with ModelSim, with the latter simulating the RTL designs of migrated IP.

As such, two QEMU extensions were developed, as well as a VPI dynamic library for Verilog simulators, to support this validation approach. Although modeled with this system design in mind, QEMU extensions were developed in a modular and layered way, allowing for uses other than the ones proposed in the previously mentioned scenarios.

This chapter is further divided into three sections, describing these extensions with focus being placed on how to use them, and providing examples whenever suitable. Firstly, QEMU plugin extension is an extension that allows QEMU to load behavioral hardware models as dynamic libraries, which is useful for device driver development during a design space exploration phase, early in the project. Secondly, QEMU external model extension is an extension that allows QEMU to use hardware device models modeled in another simulation tool, which is useful for system-wide validation with device models that require validation in other application domains, not being confined to the hardware acceleration domain. Lastly, QEMU Co-simulation PLI library is a PLI dynamic library that allows Verilog simulators that support VPI to simulate hardware IPs integrated into QEMU's emulated machines, making use of the QEMU external model extension.

3.1 QEMU Plugin Extension

As mentioned in section 2.6, to model hardware devices in QEMU it is necessary to understand its source code and API in order to write device hardware code, as well as to integrate it in the sources and recompile the whole program. This may prove to be a hindering for developers wanting to use QEMU to speed-up device driver development, given that QEMU is admittedly a non-documented program from the developers' point of view. To tackle the whole program is heavy duty for a developer whose main purpose is not expanding QEMU itself, as there is a big bulk of source code to understand. Also, frequent recompilation of the whole program is not practical at all, not to mention the possibility of breaking the program unexpectedly.

As such, this extension tries to solve some of these issues, firstly by abstracting some of the intricacies of QEMU's internal API, thus making it easier to extend QEMU with a hardware device *ex nihilo* via a simplified API, and secondly by reducing compilation effort by allowing the use of plugins containing device models. This is specially useful when the project is in an early stage and hardware IPs are being designed, enabling the designer to try out behavioral hardware models and their respective device drivers.

3.1.1 Extension Overview

Figure 3.3 presents an overview of a hardware accelerated embedded Linux case of study running on a QEMU emulation that makes use of the extended plugin capabilities.

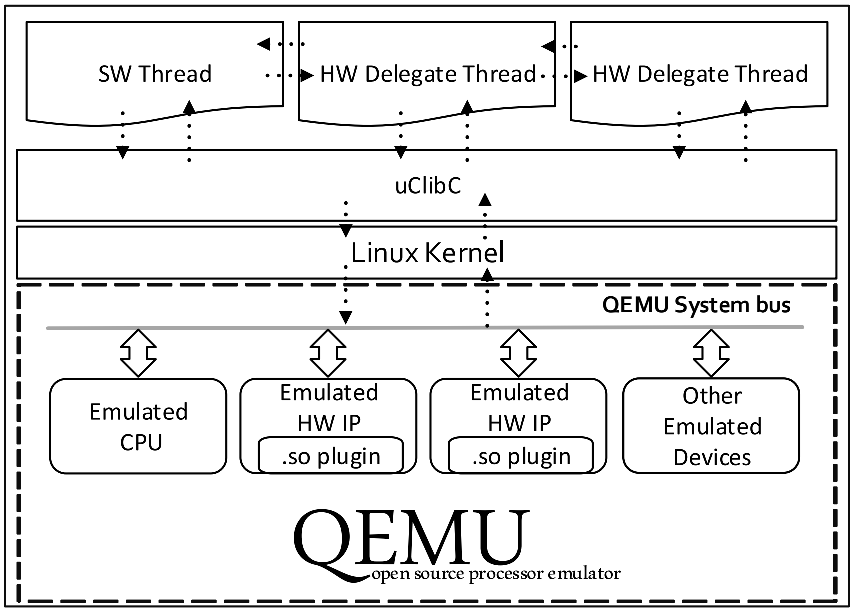


Figure 3.3: Plugin extension overview

A hardware accelerated embedded application is usually composed by a collection of threads. Hardware delegate threads are threads that interact with other threads on a software level, representing critical processing that was migrated to hardware. Their terminology is derived from the fact that these threads delegate processing algorithms to hardware accelerators instead of implementing them, through device drivers system calls.

Hardware accelerators are accessed whenever device drivers perform hardware transactions, and are modeled in the emulated machine through device models contained in .so files. For each shared object that contains a plugin device model, there is a corresponding plugin interface device object that is instantiated in QEMU.

A plugin interface device is a system bus device that was developed in the context of this extension and included in QEMU's static device models. Its purpose is loading information and calling device modeling functions from a shared object, acting as an interface to a model contained in the former. Appendix E.1 contains a QEMU Monitor device property output for a plugin interface device that corresponds to a plugin LED model.

Figure 3.4 presents a class diagram of a plugin interface device.

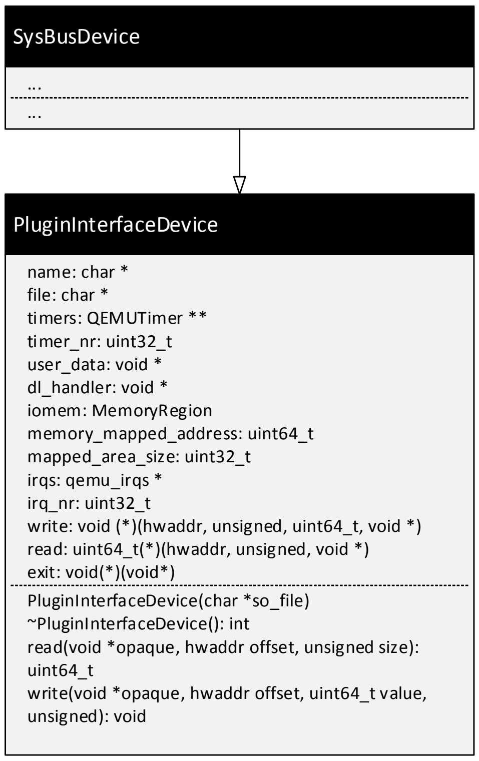

SysBusDevice
...
...
PluginInterfaceDevice
name: char *
file: char *
timers: QEMUTimer **
timer_nr: uint32_t
user_data: void *
dl_handler: void *
iomem: MemoryRegion
memory_mapped_address: uint64_t
mapped_area_size: uint32_t
irqs: qemu_irqs *
irq_nr: uint32_t
write: void (*)(hwaddr, unsigned, uint64_t, void *)
read: uint64_t(*)(hwaddr, unsigned, void *)
exit: void(*)(void*)
PluginInterfaceDevice(char *so_file)
~PluginInterfaceDevice(): int
read(void *opaque, hwaddr offset, unsigned size): uint64_t
write(void *opaque, hwaddr offset, uint64_t value, unsigned): void

Figure 3.4: Plugin interface device UML

Figure 3.5 presents a sequence diagram of a plugin device load process, and subsequent hardware transaction in QEMU.

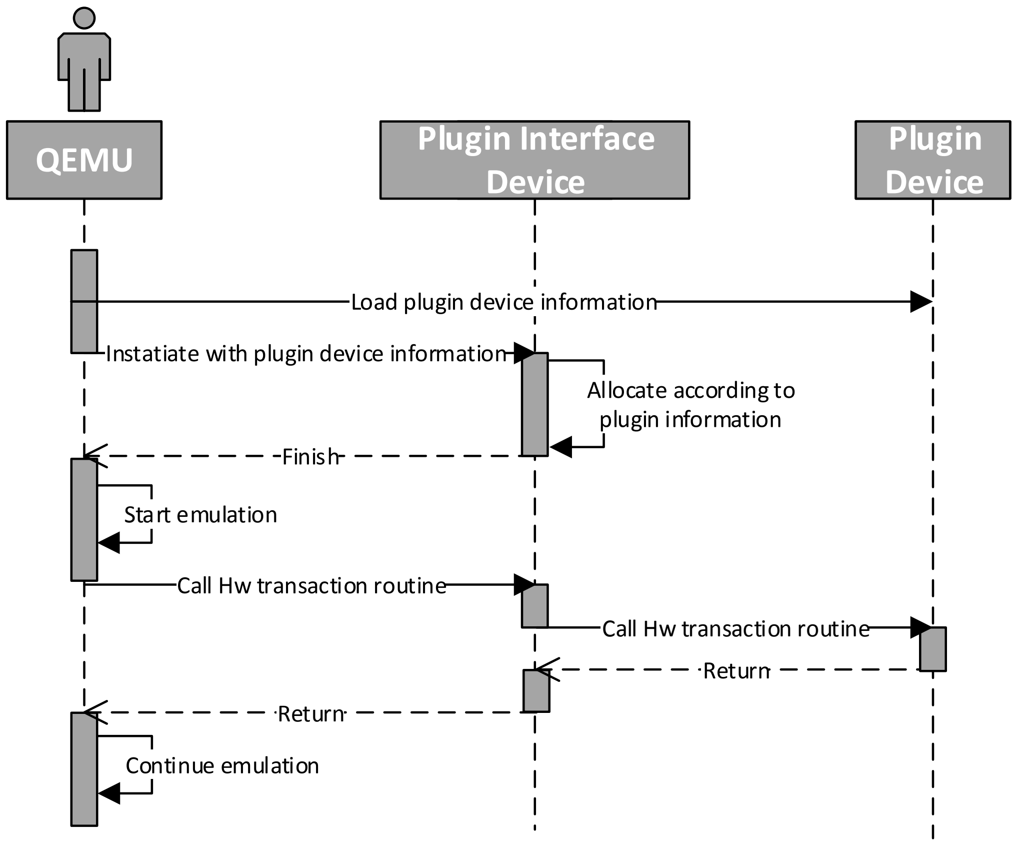

QEMU
Plugin Interface Device
Plugin Device
Load plugin device information
Instatiate with plugin device information
Allocate according to plugin information
Finish
Start emulation
Call Hw transaction routine
Call Hw transaction routine
Return
Return
Continue emulation

Figure 3.5: Plugin device loading sequence diagram

Plugin interface devices are instantiated per existing plugin device. They are instantiated according to information loaded from the plugin device, such as base address and hardware transaction behavior. The plugin interface device is mapped in the emulated machine address space upon instantiation using memory region information that is loaded from the corresponding plugin device. Hardware transaction functions are also registered upon plugin interface device instantiation, and are executed whenever a plugin interface hardware transaction function gets called. This is done so that hardware modeling functions implemented in plugin devices are visible within QEMU's device modeling environment.

QEMU plugin extension uses a specific directory for plugin devices, scanning it and loading all found shared objects. An environment variable is set with the directory that contains plugin devices that are to be instantiated, loading plugins after machine initialization, but before any emulation starts. For each .so file found, hardware-related information is loaded and a plugin interface device is instantiated with its respective plugin information. Figure 3.6 presents a flowchart of how the process of loading plugin devices is done.

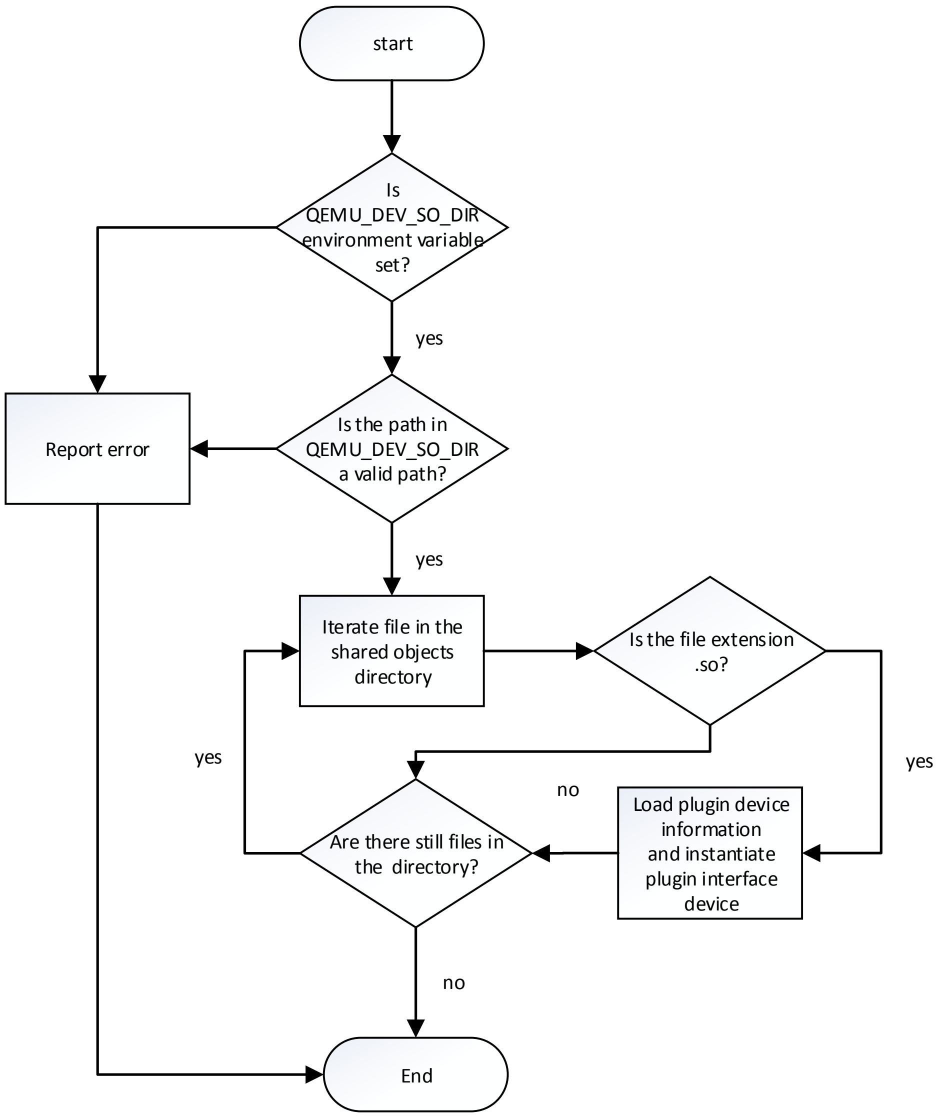

start
Is QEMU_DEV_SO_DIR environment variable set?
yes
Report error
Is the path in QEMU_DEV_SO_DIR a valid path?
yes
Iterate file in the shared objects directory
Is the file extension .so?
yes
no
yes
Load plugin device information and instantiate plugin interface device
Are there still files in the directory?
no
End

Figure 3.6: Plugin extension initialization flowchart

3.1.2 Device API

To develop a plugin device, an API is provided. This API comprises plugin device registering mechanisms, as well as functions and variables to model hardware behavior. A structure to be filled with device related information is used to represent a plugin device. When the shared object is loaded, a plugin interface device will be instantiated with that information. The following information is loaded from the plugin device:

- Name;

  This is a string with the device name. Its main purpose is debugging through QEMU monitor property printing.

- Base address;

  This is the device's memory-mapped address. The plugin interface device will be mapped in this address, and whenever QEMU catches a hardware access within its memory region, that is, within the base address and its corresponding block size, the plugin interface device hardware transaction function will be called.

- Block size;

  This is the size of the memory-mapped block in bytes. It will be used in conjunction with the base address to determine the device's memory-mapped area.

- Interrupt requests;

  This is an array with the requested interrupt numbers to be registered by the plugin interface device.

- Number of interrupt requests;

  This is the size of the interrupt requests array.

- Timer requests;

  To model time-aware hardware behavior, a timer API is provided to schedule functions. This is an array with timer requests to be allocated by the plugin interface device.

- Number of timers;

  This is the size of the timer requests array.

- Write transaction function;

  This is a function to be called by the plugin interface device at hardware write transactions.

- Read transaction function;

  This is a function to be called by the plugin interface device at hardware read transactions.

- Device initialization function;

  This is a function to be called by the plugin interface device when it's instantiated. It is useful to perform memory allocation by the plugin device.

- Device exit function;

  This is a function to be called by the plugin interface device when it's instantiated. It is useful to free memory allocated by the plugin device.

All this information is filled in a structure provided by the API. Appendix E.2 contains an example of this structure.

To model certain hardware behavior, a set of functions and variables are part of the API. Since QEMU symbols are not available at runtime to shared objects, and to prevent them to have to be linked to several QEMU object files and thus be bound to a specific QEMU version, function pointers are used. These function pointers are declared in the API header file, and initialized at QEMU runtime by the plugin extension when loading shared objects, allowing these pointers to be used by plugin devices as an API. A macro identifying the device module should be used, to avoid API variables and pointers re-declaration on other code modules of the plugin device .so file. Table 3.1 presents an overview of the provided API.

To compile a plugin device, a Makefile is provided. Install and uninstall rules are used to copy or remove the plugin from the directory that contains plugins to be loaded by QEMU. Appendix E.3 contains an example Makefile for a plugin device.

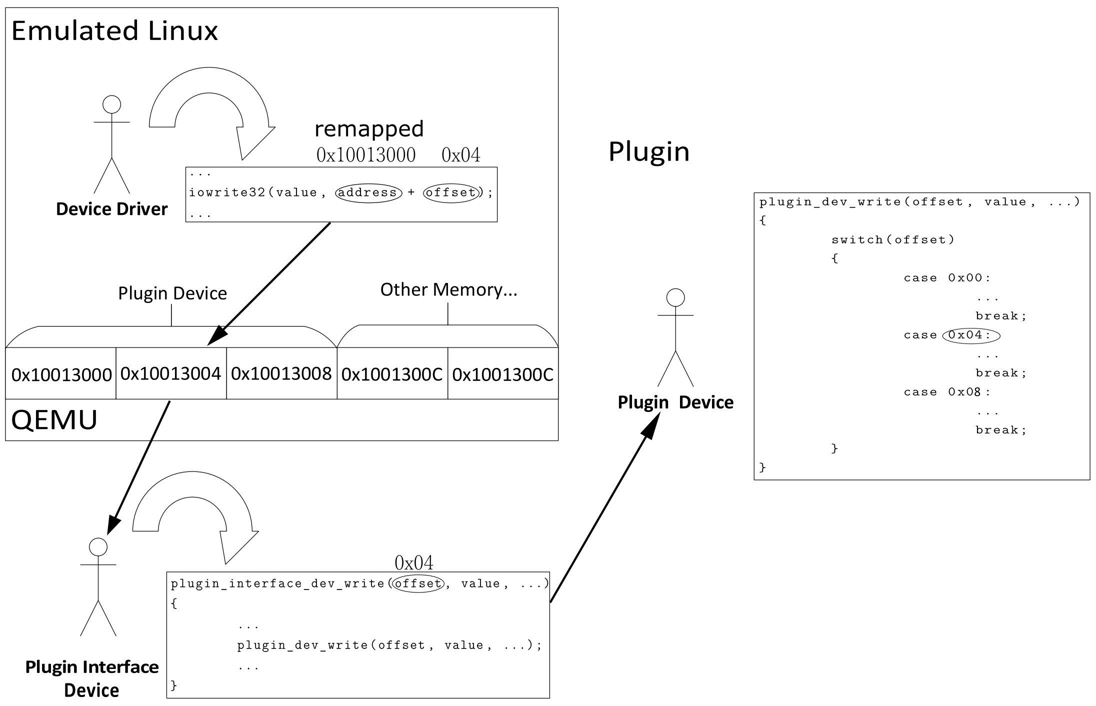
Table 3.1: Plugin device API

Name	Description	API category
interrupts	Provided array according to requests	Interrupts
irq_raise	Trigger a requested interrupt	Interrupts
irq_lower	Clear a requested interrupt	Interrupts
timers	Provided array according to requests	Timing
start_timer	Schedule a requested timer	Timing
stop_timer	Cancel a running timer	Timing
get_time	Get a clock's value	Timing
write_to_bus	Trigger master write transaction	Bus Master
read_from_bus	Trigger a master read transaction	Bus Master

3.1.3 I/O Mapping

Device modeling in QEMU functionally emulates transactions, with each hardware device object being responsible for the respective transaction behavior implementation. As such, whenever there is a hardware access on the emulated application, the write or read transaction behavior functions for the corresponding hardware device object get executed. Port-mapped operations are not supported in the developed extension as an option, given that the majority of modern architectures use memory-mapped I/O. Figure 3.7 presents how a write transaction is handled for a plugin device.

Whenever there is a hardware read or write, QEMU identifies the device object via registered memory region, and calls its respective transaction function, passing the memory offset of the hardware transaction as an argument. It is up to the transaction function to implement behavior for the different address offsets, not unlike an ioctl operation maps commands in a Linux device driver.



```
...
iowrite32(value, address + offset);
...
```

```
plugin_interface_dev_write(offset, value, ...)
{
        ...
        plugin_dev_write(offset, value, ...);
        ...
}
```

```
plugin_dev_write(offset, value, ...)
{
        switch(offset)
        {
                case 0x00:
                        ...
                        break;
                case 0x04:
                        ...
                        break;
                case 0x08:
                        ...
                        break;
        }
}
```

Figure 3.7: Plugin transactions task graph

When a plugin device model is loaded into QEMU, the respective plugin interface device is mapped according to plugin supplied information, namely base address and block size. To handle plugin transactions, the plugin interface device transaction functions simply call the respective plugin transaction functions that were registered at plugin information loading, and forward the received arguments abstracting some arguments to simplify the API.

To model register storing behavior, dynamic memory used with the initialization and exiting functions is recommended, although plugin device register mapping and memory management implementation details are up to the developer. Appendix E.4 contains a LED plugin device example, making use of the initialize and exit functions to implement dynamic memory, and read and write functions to implement transaction behavior.

3.1.4 Interrupt API

By using the provided interrupt API, the plugin device is able to model interrupt behavior by triggering and clearing interrupts. Figure 3.8 presents a sequence diagram of a plugin device making use of the interrupt API.

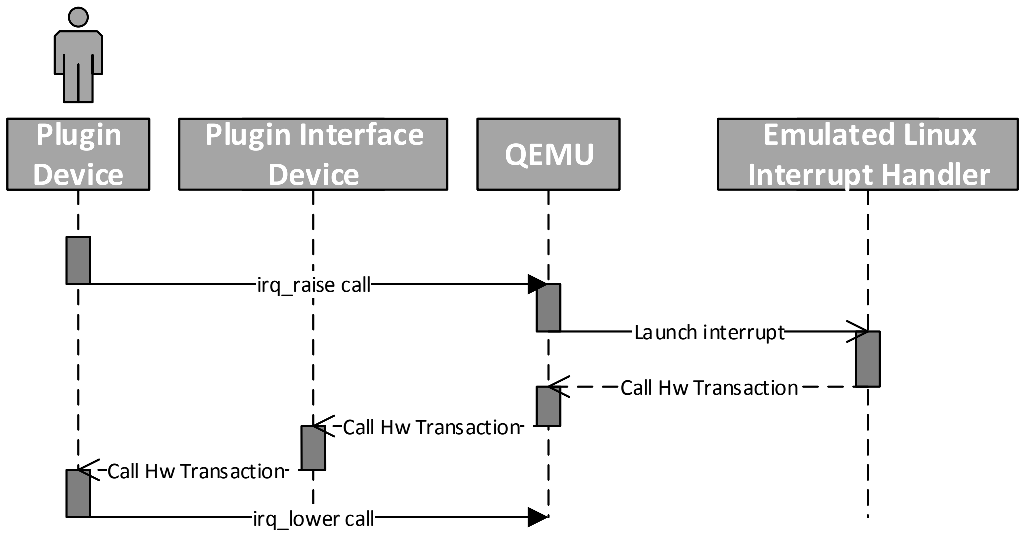
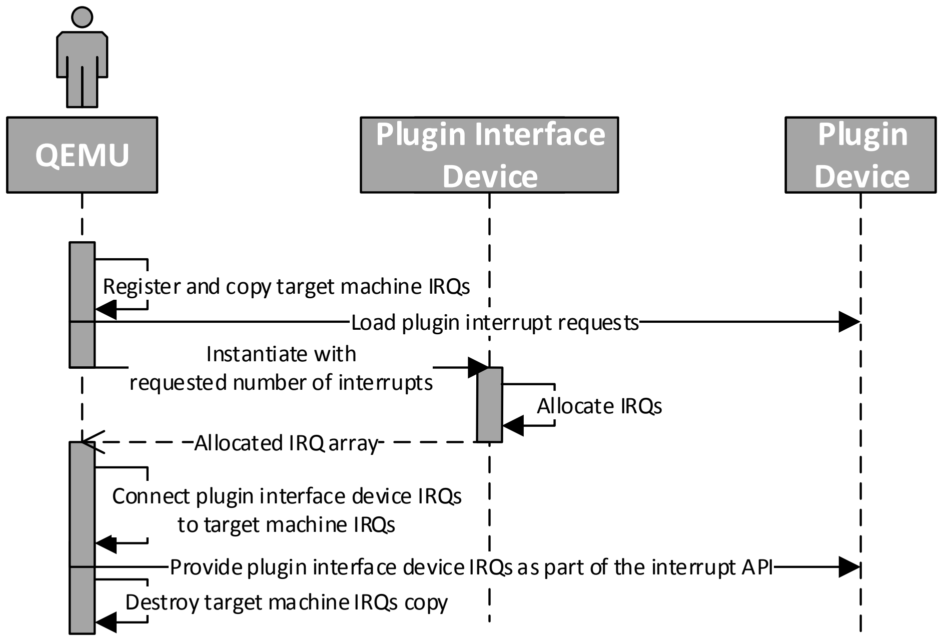


Figure 3.8: Interrupt API sequence diagram

When using the API to trigger an interrupt, the plugin device will indirectly use the internal QEMU IRQ API. The interrupt should remain active until handled, meaning that the plugin device must map a hardware read or write transaction to issue an irq_lower call. This is necessary to enable the emulated software interrupt handler to clear the interrupt upon handling it. To use the interrupt API functions, an IRQ array is provided to be used as an argument, according to what was requested. Interrupts are requested when registering plugin device information, as was previously mentioned. Figure 3.9 presents a sequence diagram for an IRQ allocation by the plugin extension:

As mentioned in section 2.6.4, at target machine initialization there is an array of IRQ objects used to connect IRQs between CPU and peripheral devices. The plugin extension registers this array and copies it internally, to be able to connect allocated IRQ objects to it, as it is destroyed after machine initialization. IRQs objects are allocated by plugin interface devices, and connected to the copied target machine's IRQ array according to requests. Requests are made by registering an array of interrupt numbers on a plugin device, which are to be used when matching IRQ objects to their corresponding machine IRQ array indexes. This obviously means that the plugin device developer must have knowledge of the QEMU machine being used, as to know the the correct IRQ numbers when implementing requests.



QEMU
Plugin Interface Device
Plugin Device
Register and copy target machine IRQs
Load plugin interrupt requests
Instantiate with requested number of interrupts
Allocate IRQs
Allocated IRQ array
Connect plugin interface device IRQs to target machine IRQs
Provide plugin interface device IRQs as part of the interrupt API
Destroy target machine IRQs copy

Figure 3.9: Plugin extension IRQ allocation sequence diagram

After IRQs are allocated and connected, an array is provided to the plugin device with the allocated interrupts, allowing the plugin device to model interrupt behavior by using it with the provided interrupt API. Once all plugin devices have been loaded, the machine IRQ copy is destroyed.

If IRQs are not supported in the target machine, the interrupt API function pointers will be initialized with a dummy function, having no effect whatsoever when called. A warning for this effect will be issued upon plugin loading so that the user is aware of this, but emulation is not stopped since it may not be an impairment for certain device models. Table 3.2 presents the provided interrupt API.

Table 3.2: Interrupt API

API	**Description**
qemu_irq interrupts[]	Allocated interrupt array
void irq_raise(qemu_irq desired_irq)	Triggers interrupt
void irq_lower(qemu_irq desired_irq)	Clears interrupt

The provided interrupt array size is the same as the number of requested interrupts. Each index will correlate with the requested interrupts order. For instance if interrupt requests are 8 and 10, by this order, to use interrupt 8 as an argument on the API index 0 should be used, and to use interrupt 10 index 1 should be used. Appendix E.5 contains a LED plugin device example, making use of the interrupt API by raising an interrupt every time it's turned on.

To add IRQ support for plugin devices on a given QEMU machine, it is necessary change the machine's source file and register the machine IRQ array, so that the plugin extension may copy it internally. Appendix E.6 contains an example of how to add IRQ support for the plugin extension on an unsupported machine.

3.1.5 Timing API

By using the timing API, a plugin device is able to model time-aware hardware behavior by accessing simulation time or register functions to be executed after a pre-determined time interval has elapsed. Figure 3.10 provides a sequence diagram of a plugin device making use of the timing API.

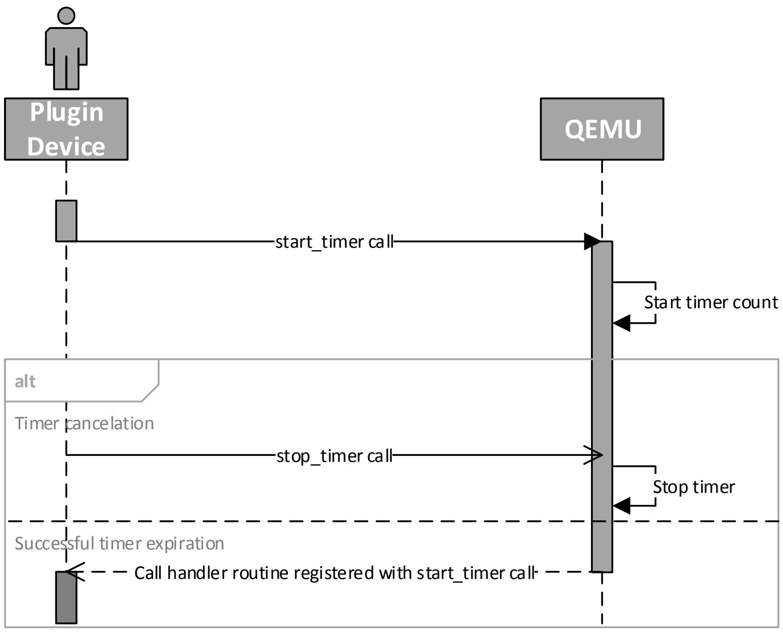


Figure 3.10: Timing API sequence diagram

QEMU provides an internal API to register functions to be called after a specific simulation time has passed. The plugin timing API indirectly uses QEMU's own API, providing QEMU timers that schedule timer handling functions to plugins. Timers should be requested using the API provided structure that contains plugin device information. To register timer requests, an array of timer requests is used, with the following information:

- Handler function;

  This is the function to be called when the timer reaches its expiration date.

- Clock source;

  This is the clock to be used as clock source for the timer. Available clock sources are the same as QEMU clock sources, mentioned in 2.6.5.

- Time scale.

  This is the timing scale for the timer. Available scales are nanoseconds, microseconds and milliseconds.

Timers are allocated by plugin interface devices according to the requests, and provided to the plugin by the extension. Figure 3.11 presents a sequence diagram for timer allocation by the plugin extension:

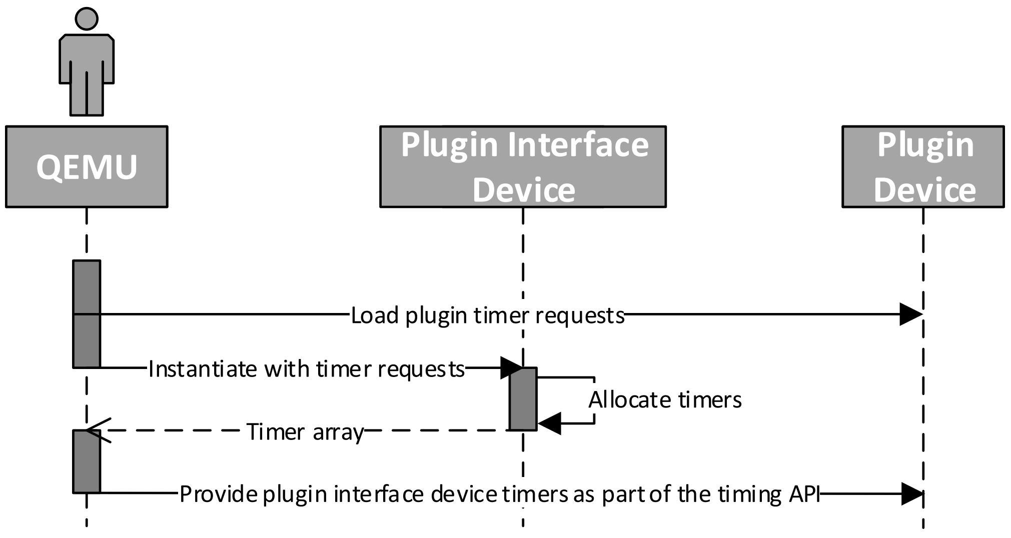

QEMU
Plugin Interface Device
Plugin Device
Load plugin timer requests
Instantiate with timer requests
Allocate timers
Timer array
Provide plugin interface device timers as part of the timing API

Figure 3.11: Plugin extension timer allocation sequence diagram

Timer requests are loaded along with the plugin device information, and passed to its respective plugin interface device upon instantiation. After being allocated by the plugin interface device, they are provided to the plugin device as part of the timing API. Table 3.3 presents the timing API.

Table 3.3: Timing API

API	Description
QEMUtimer timers[]	Allocated timer array
void start_timer(QEMUtimer desired_timer, int64_t expiration_time)	Starts running a requested timer
void stop_timer(QEMUtimer desired_timer)	Cancels a running timer
int64_t get_time(QEMUClocktype clock, int scale)	Returns a clock's value

To start running a registered timer, 'start_timer' is used. When the timer reaches expiration time, its associated handler is called. QEMU timers are one-shot, meaning that timers must be started again in the handler if continuous behavior is desired. They may also be canceled before timeout is reached and their respective handlers called. There is also a function to check the current time of a clock, using the same macros for clock types and time scales used on timer registering. Appendix E.7 contains a blinking LED plugin device example, making use of the provided timing API to model blinking behavior.

3.1.6 Bus Master API

Sometimes, when modeling hardware behavior, there are devices (such as DMA or **J**oint **T**est **A**ction **G**roup (JTAG) peripherals) that need to model master system bus access behavior. To provide means of modeling such behavior, a bus master API is provided. Figure 3.12 provides a sequence diagram of a plugin device making use of the bus master API to access another QEMU device.

To make use of the API, the plugin device simply calls the master transaction functions, and QEMU issues a system bus transaction not unlike when an instruction that accesses the system bus is detected. Table 3.4 presents the bus master API.

Table 3.4: Bus master API

API	Description
void write_to_bus(hwaddr address, char *buffer, int buffer_size)	Master write
void read_from_bus(hwaddr address, char *buffer, int buffer_size)	Master read

Appendix E.8 contains a master plugin device example, that acts on any of the previously presented plugin examples.

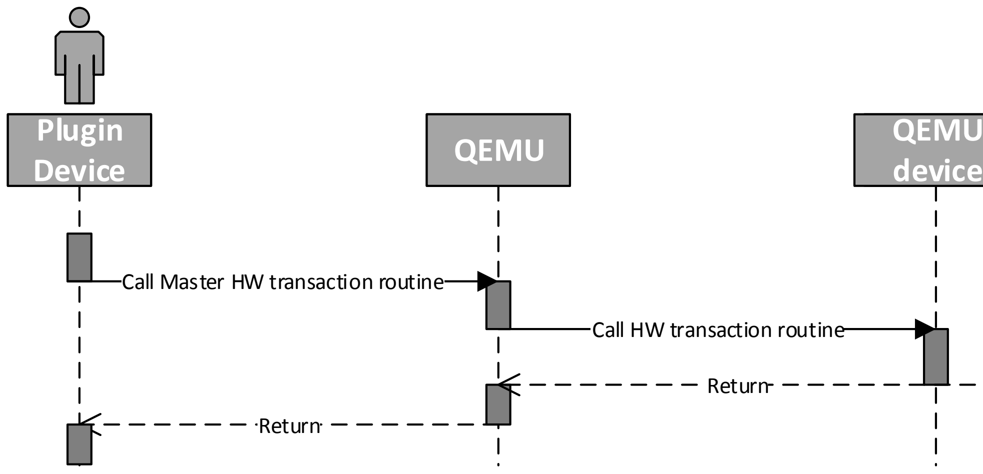

Plugin Device
QEMU
QEMU device
Call Master HW transaction routine
Call HW transaction routine
Return
Return

Figure 3.12: Bus master API sequence diagram

3.2 QEMU External Model Extension

In embedded systems with development in several application domains, one of the main challenges is full-system integration. When following an isolated validation design flow, most of the domain-crossing system faults are not detected in simulation, being later detected when the system is physically implemented. This design methodology is highly ineffective, with a high risk of lengthy development cycles, exponentially growing with the system's complexity. The more domains and interactions the system has, the higher chance of system integration failures to occur upon system deployment and respective validation.

As such, this extension intends to allow QEMU to be used in a co-simulation context, enabling hardware devices to be modeled externally in other simulation tools. This is useful in several application domains, namely in hardware acceleration. During later stages of hardware accelerator development, hardware models and respective drivers are already developed but not integrally validated, making co-simulation a desirable validation technique.

3.2.1 Extension Overview

Figure 3.13 presents an overview of a hardware accelerated embedded Linux case of study running on a QEMU emulation that makes use of the extended co-simulation capabilities.

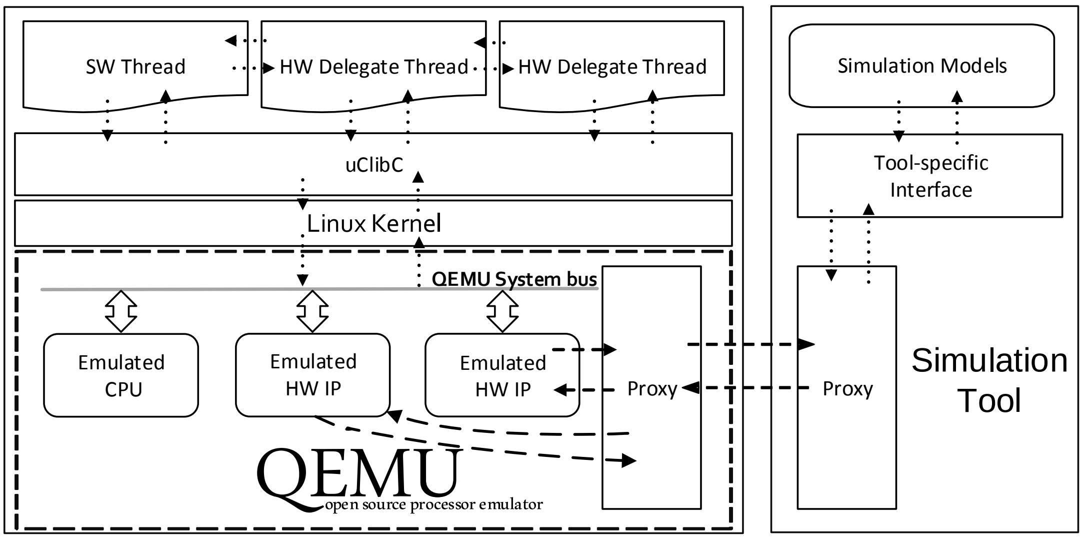


Figure 3.13: External model extension overview

A hardware accelerated embedded application is usually composed by a collection of threads, including hardware delegate threads. Hardware delegate threads delegate processing algorithms to hardware accelerators by interacting with them through device driver system calls. This extension enables hardware accelerators to be modeled externally of QEMU, by incorporating models contained in other simulation tools in QEMU emulation. For every external model that is to be integrated into QEMU there is a corresponding proxy interface device that exchanges information between QEMU and the respective external simulation tool via a proxy entity.

A proxy interface device is a type of system bus device that was developed in the context of this extension and included in QEMU's static device models. Its purpose is loading information from an external model and represent it in QEMU simulation. When a hardware transaction is issued, the proxy interface device forwards transaction information to the respective external model through the proxy channel. The external tool must also have a proxy to establish interactions with its simulation models and QEMU, usually through tool-specific interface frameworks or libraries. Figure 3.14 presents a class diagram of a proxy interface device.

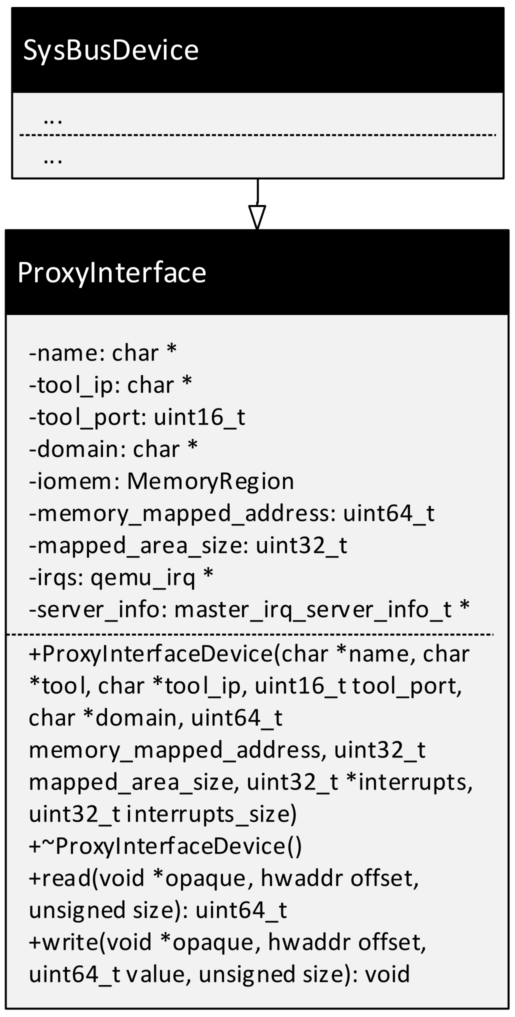
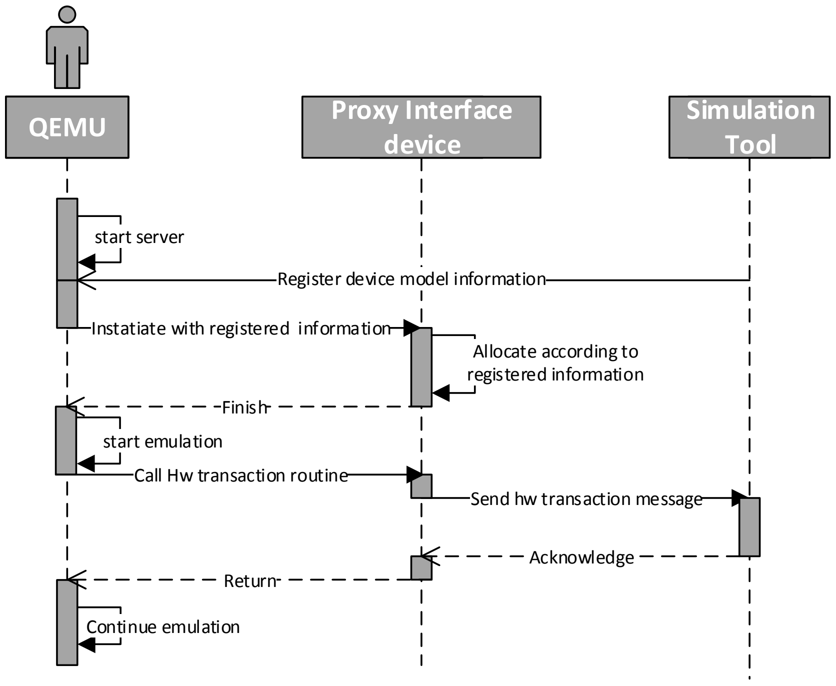

SysBusDevice
...
...
ProxyInterface
-name: char *
-tool_ip: char *
-tool_port: uint16_t
-domain: char *
-iomem: MemoryRegion
-memory_mapped_address: uint64_t
-mapped_area_size: uint32_t
-irqs: qemu_irq *
-server_info: master_irq_server_info_t *
+ProxyInterfaceDevice(char *name, char *tool, char *tool_ip, uint16_t tool_port, char *domain, uint64_t memory_mapped_address, uint32_t mapped_area_size, uint32_t *interrupts, uint32_t interrupts_size)
+~ProxyInterfaceDevice()
+read(void *opaque, hwaddr offset, unsigned size): uint64_t
+write(void *opaque, hwaddr offset, uint64_t value, unsigned size): void

Figure 3.14: Proxy interface device UML

Appendix F.1 contains a QEMU Monitor device property output of a proxy interface device that represents a LED device modeled in ModelSim. Figure 3.15 presents a sequence diagram of an external model loading process, and subsequent hardware transaction.



QEMU
Proxy Interface device
Simulation Tool
start server
Register device model information
Instatiate with registered information
Allocate according to registered information
Finish
start emulation
Call Hw transaction routine
Send hw transaction message
Acknowledge
Return
Continue emulation

Figure 3.15: External model loading and use sequence diagram

A server is used at QEMU start-up to synchronize with external tools. External tools will establish a connection with this server, and register the models that are to be integrated into QEMU's emulation. This registration triggers a proxy entity instantiation for each tool and respective proxy interface devices instantiations for each of the registered device models. As previously mentioned, transactions on proxy interface devices are forwarded to their respective external simulation models. To make QEMU use these features, an '-ext-tools' argument must be provided on the QEMU launch command. The following is a template of a QEMU launch command with external tools co-simulation active:

```
$ qemu-system-[arch] [flags] -ext-tools [number of tools
   ],[server port(optional)]
```

Multiple external tools are supported in the extension, with the number of simulation tools being passed as a parameter. This parameter informs QEMU of how many external tools will try to connect to the start-up synchronization server, with the server port being an optional parameter. By default the server port is 10 000.

3.2.2 External Tool Synchronization

Figure 3.16 presents an overview diagram of two simulation tools, 'A' and 'B', connected to QEMU during the start-up synchronization process.

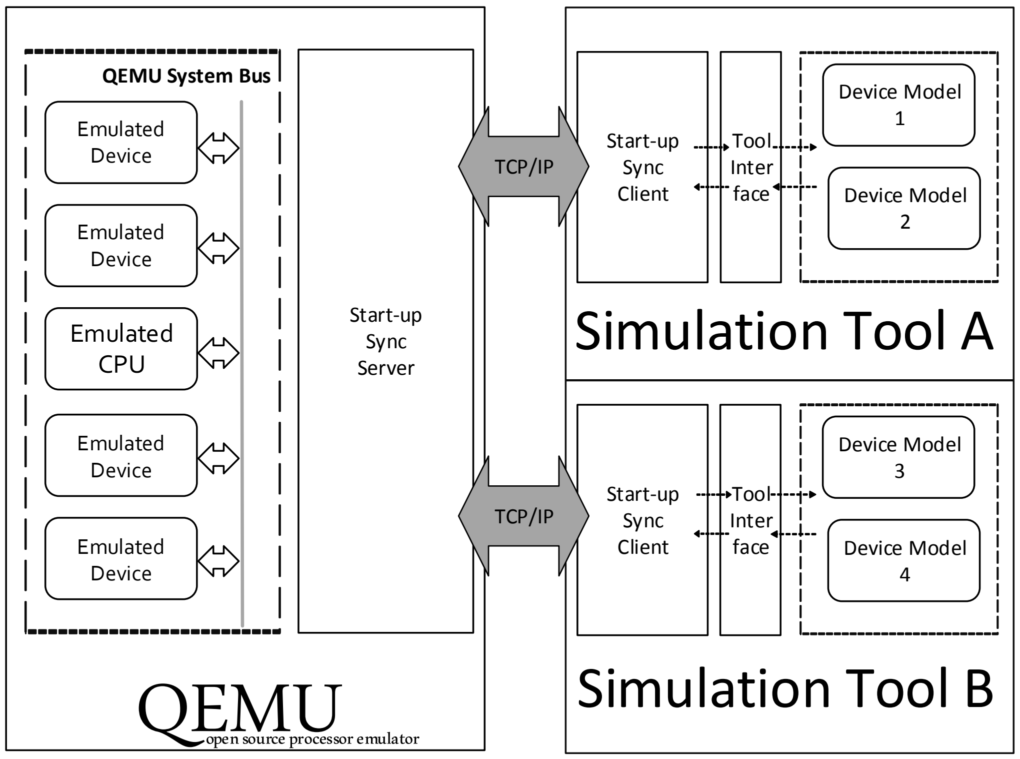


Figure 3.16: External model extension start-up synchronization overview

At start-up, a TCP/IP server is started to synchronize with external tools. This server is temporary, used only to initialize proxy interface devices and proxies themselves, blocking QEMU until the requested number of external tools do connect. After all tools have finished the registration process, the connection is closed and the start-up synchronization server is terminated.

Simulation tools use a client TCP/IP connection to connect with the start-up synchronization server launched by QEMU, instantiating their proxies during this process and closing the synchronization connection after registering their device models. Figure 3.17 presents an overview diagram of the same scenario presented in figure 3.16 after the registering process is done, and emulation is started.

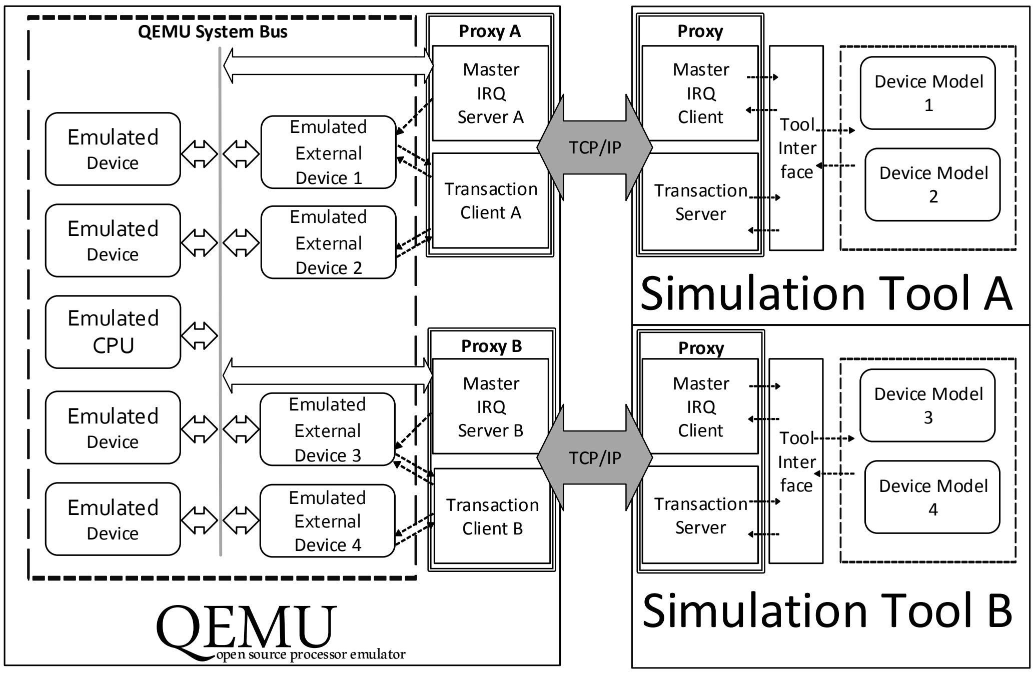


Figure 3.17: External model extension runtime overview

The proxy entities are composed of client and server pair connections per tool in QEMU, with a corresponding pair in each simulation tool. In simulation tools, transaction servers receive proxy device transaction messages, triggering hardware transactions in their native device models. Since there is only one transaction being ran at a time, only one client connection is used per tool in QEMU, with the communication channel being shared between proxy interface devices with no risk of race conditions. A server is also started on the QEMU proxy for each simulation tool, to receive messages that trigger system bus master accesses or interrupts. Obviously, there is also a client channel in each simulation tool to send bus master access or interrupt messages.

All proxy servers allocate their ports dynamically, with this information being exchanged during the start-up synchronization process. Figure 3.18 presents a flowchart of the registration process at the external model extension start-up. For each tool that connects to the start-up synchronization server, a thread is launched to deal with the tool's registration process. Figure 3.19 presents a flowchart of the registration process when a thread is launched upon tool connection.

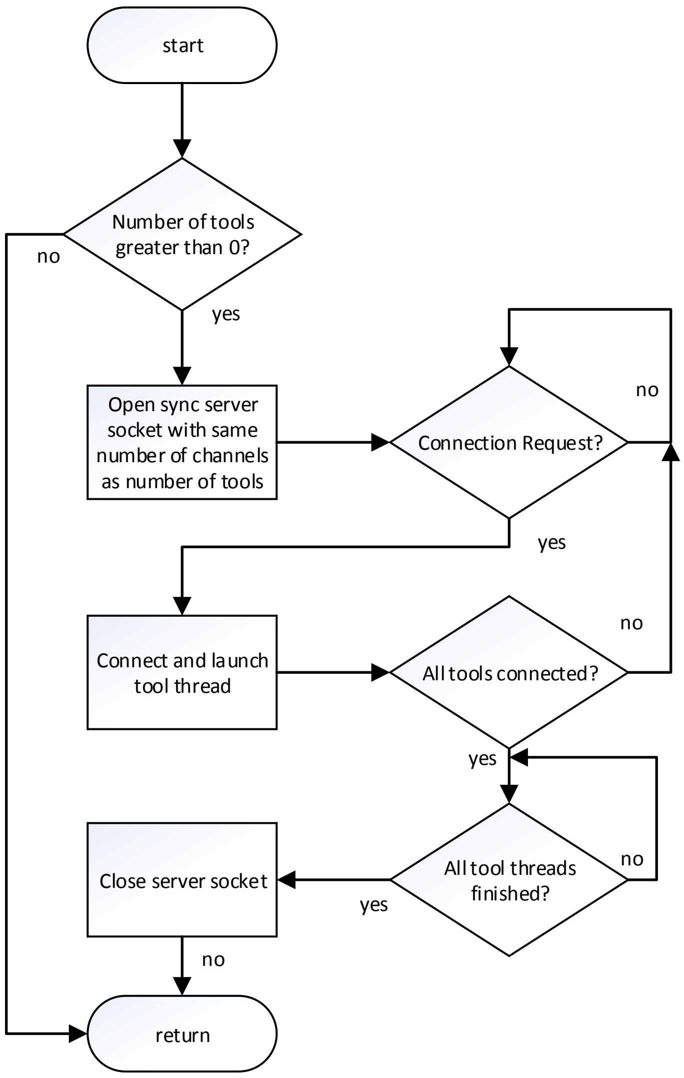

start
Number of tools greater than 0?
no
yes
Open sync server socket with same number of channels as number of tools
Connection Request?
no
yes
Connect and launch tool thread
All tools connected?
no
yes
All tool threads finished?
no
yes
Close server socket
no
return

Figure 3.18: Start-up sync server flowchart

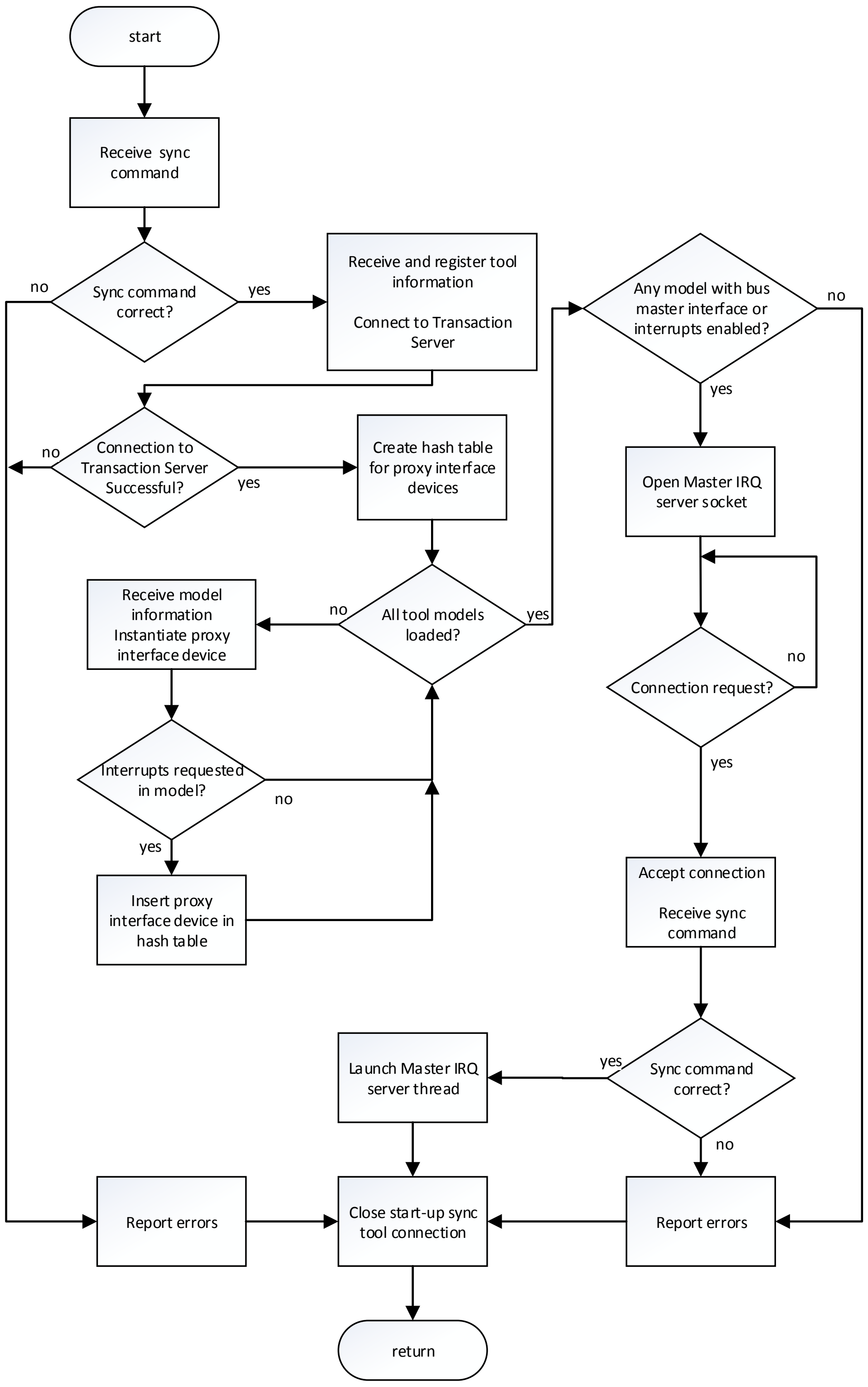

start
Receive sync command
Sync command correct?
no
yes
Receive and register tool information
Connect to Transaction Server
Connection to Transaction Server Successful?
no
yes
Create hash table for proxy interface devices
All tool models loaded?
no
yes
Receive model information
Instantiate proxy interface device
Interrupts requested in model?
no
yes
Insert proxy interface device in hash table
Any model with bus master interface or interrupts enabled?
no
yes
Open Master IRQ server socket
Connection request?
no
yes
Accept connection
Receive sync command
Sync command correct?
yes
no
Launch Master IRQ server thread
Report errors
Close start-up sync tool connection
Report errors
return

Figure 3.19: Start-up sync server flowchart

Upon tool registration, connection to its transaction server is attempted by QEMU. The server port is attributed dynamically, and is received along with tool information. A proxy interface device hash table is created for proxy interface devices of external models with interrupt requests. This is done so because the Master IRQ server needs to access a proxy interface device to raise or lower their IRQ objects.

The next step is registering all tool models, and instantiating proxy interface devices for each model. If interrupts for an external model are requested, IRQs are allocated upon proxy interface device instantiation, and the proxy interface device is inserted in the hash table.

The last step of the tool registration process is The Master IRQ server launching, which is only done if needed. This server's port attribution is also dynamic. After all tool registration is done, the start-up sync connection with the tool is closed, and the thread is terminated.

Each tool provides information upon tool, used to register models and instantiate proxy interface devices, consisting in:

- Name;

  This is a string the simulation tool's name, such as ModelSim, PSIM or other simulators. Its main purpose is debugging through QEMU monitor property printing.

- Domain

  This is string with the simulation tool application domain, for instance, hardware acceleration. Its main purpose is debugging through QEMU monitor property printing.

- Number of models;

  This is the number of models that the tool wishes to register for co-simulation usage. It will be used in the communication protocol, indicating the number of expected models.

- Transaction server port;

  This is the server port that was attributed to the tool's transaction server, and will be used to create a client connection to the latter.

- Bus master interface.

This is a flag that is used to enable or disable bus master capabilities from the extension. If only slave models are used, the tool should set this flag to 0.

After tool information is registered, models are registered individually to instantiate proxy interface devices. Individual model specific information comprises:

- Name;

  This is a string with the device name. Its main purpose is debugging, through QEMU monitor property printing.

- Base address

  This is the device's memory-mapped address. The proxy interface device will be mapped in this address, and whenever QEMU catches a hardware access within its memory region, that is, within the base address and its corresponding block size, the proxy interface device's hardware transaction function will be called.

- Block size;

  This is the size of the memory-mapped block in bytes. It will be used in conjunction with the base address to determine the device's memory-mapped area.

- Interrupt requests;

  This is an array with the requested interrupt numbers to be registered by the proxy interface device.

- Number of interrupt requests.

  This is the size of the interrupt requests array.

Appendix F.5 contains a sequence diagram that details the registration communication protocol.

3.2.3 I/O mapping

Similarly to plugin devices, port-mapped I/O operations are not supported in external models, with memory mapping being the only available I/O mapping mechanism. Proxy interface devices are accessed whenever a hardware transaction for an external model is performed, and handle external model transactions by sending TCP/IP messages to the external simulation tools. External tools map their hardware as desired, with no imposition from the extension whatsoever. Transactions are carried out by the external tool upon message reception on the tool's transaction server. The following information is comprised in a transaction message:

- Base address;

  This field is used to identify the device. Offset and base address were separated to help to identify the message destination node.

- Offset;

  This is the offset that added with the base address determines the transaction's target address.

- Size;

  This is a field used to identify the size of the transaction in bytes. Useful to implement single byte transactions for machines that support such instructions.

- Value;

  This field is sent in write transactions, and received in read transactions.

- Time;

  This field is used to inform the external simulation tool of QEMU's current virtual time, and to return to QEMU how much time the transaction took. Time scale is implemented in nanoseconds.

Appendix F.5 contains a sequence diagram that details transaction communication protocol.

3.2.4 Timing

The external model extension supports cycle-accurate device emulation. This is done so, because unlike behavioral modeling validation, co-simulation is best used for system integration debugging with structural models that are simulated in cycle-accurate fashion, such as when developing in a hardware acceleration domain. To correctly use the extension's device time modeling, QEMU emulation should be ran deterministically, with the instruction counter as virtual clock.

QEMU's instruction counter virtual clock was modified, to enable time warping at each transaction. By default, the virtual clock is ran as long as the virtual machine is running, running also when device behavior code is being ran. With the instruction counter enabled, the virtual clock only increments with each new instruction, being frozen when in a device modeling context. This is ideal for a co-simulation scenario, allowing for external model's simulation time to be taken into account by warping the instruction counter virtual clock at each transaction.

Figure 3.20 presents a sequence diagram for time warping in a hardware transaction.

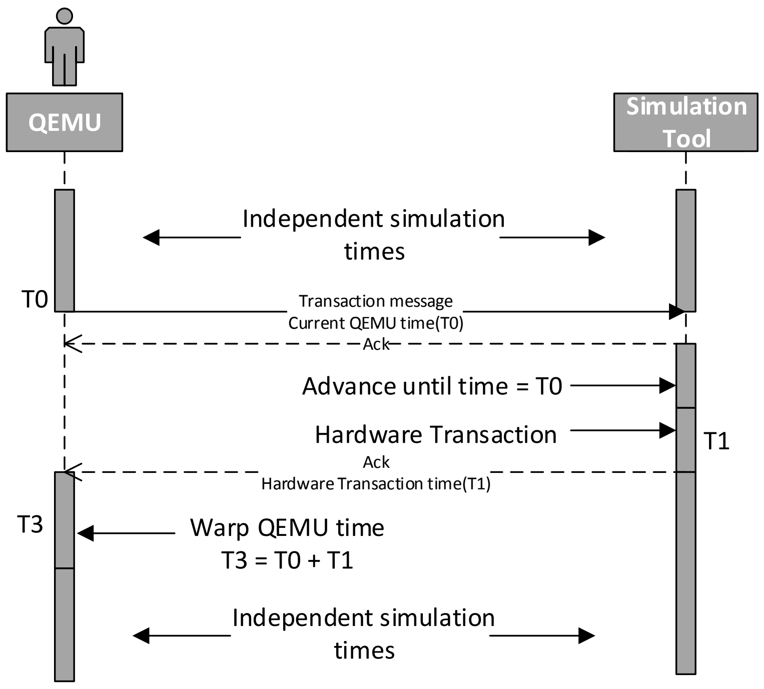


Figure 3.20: External model transaction timing sequence diagram

Simulation timesteps are kept independently between external tools and QEMU. When a transaction message is sent, current QEMU time is included in the transaction message sent to the external simulation tool. The tool will then evaluate its own simulation time, and if it is behind QEMU, the simulation will advance until its simulation time matches QEMU's. This implementation was chosen, given that most domain-specific simulators have much longer integration timesteps than QEMU, which is known for very fast emulations due to its dynamic binary translation techniques.

After the transaction is completed, transaction time is sent along with the acknowledgment message. QEMU will then warp the time to match the time spent in the simulation tool, and both tools will return to independent simulation times. If an external tool does not wish to implement this behavior, it may simply ignore QEMU's simulation time, and send 0 as warp time on the acknowledgment message. This is useful for domains that do not implement cycle-accurate device models.

3.2.5 Interrupts and Bus Master Transactions

To enable interrupts and bus master interactions between QEMU and external tools, there is a TCP/IP server for each tool to receive bus master transaction messages, and interrupt raise or lower messages. This server is only started if the tool requires bus master accesses or if any of the external models uses interrupts.

The following information is comprised in a bus master message:

- Memory address;

  The hardware memory-mapped address on which the read or write transactions should be performed.

- Value;

  This is a field used in bus master messages on write transactions, and used on server acknowledge messages on read transactions.

- Size.

  The number of bytes of the transaction. Useful for machines with single-byte memory access instructions to differentiate hardware accesses.

The following information is comprised in an interrupt message:

- Base address

  This is the base address where the external model that requests the interrupt is mapped. It is a way of identifying the external model that requests the interrupt, so the corresponding proxy interface device may be accessed and the respective IRQ object used.

- Interrupt number

  This is the index of the requested interrupts array. Similar to the plugin interrupt API, if requested interrupts were 8 and 10, in this order, to trigger interrupt 8 an interrupt message with index 0 should be sent, and to trigger interrupt 10 an interrupt message with index 1 should be sent.

When an interrupt message is received, the Master IRQ server will look for the corresponding proxy interface device in its hash table, and will raise or lower the IRQ in the proxy interface device IRQ array object according to the received index.

Figure 3.21 presents a flowchart for a Master IRQ server.

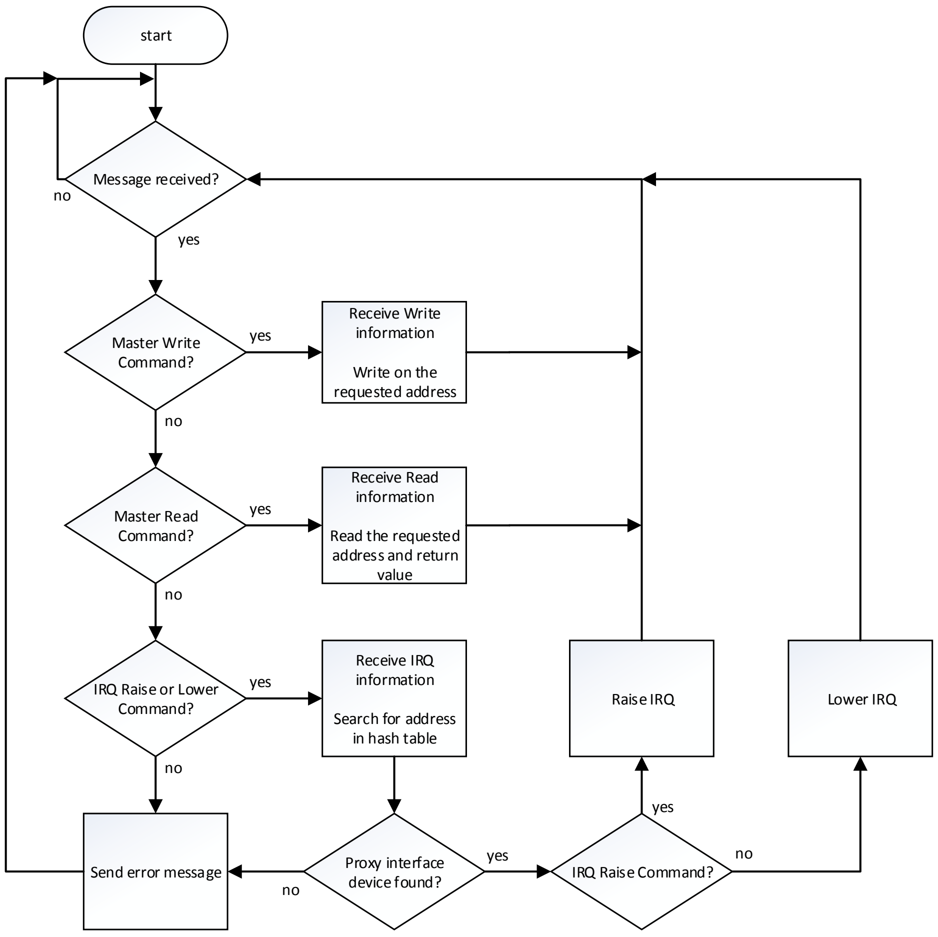


Figure 3.21: Master IRQ server thread

3.2.6 QEMU External Tool Library

By analyzing the external model overview diagram, it is clear that despite varying tools and their respective specific interfaces, the network proxy services that are going to be implemented are the same. Figure 3.22 presents an overview of an external tool that supports QEMU co-simulation.

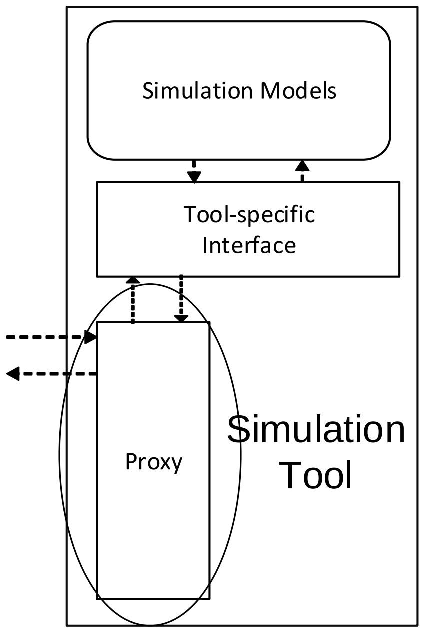


Figure 3.22: External tool Interface

For this reason, a library that implements the proxy entity was developed, abstracting those details for the tool-specific interface layer. This library implements the network protocols, implementing tool and respective models registering, as well as instantiation and management of network communication channels. Table 3.5 presents the API of this library.

Function qemu_connect_tool is used to connect to QEMU's start-up synchronization server, register the tool and its models, and start the transaction server and Master IRQ client connections that comprise the proxy. Tool information, which includes an array of model information. must be provided to this function. Optionally, start-up synchronization server IP address or port may be provided, with default values being localhost and 10 000 respectively.

Whenever the transaction server receives a transaction message, functions registered using qemu_register_slave_write/read will be called, delegating behavior

Table 3.5: QEMU external tool library API

API	Description
qemu_connect_tool	Establish a connection to QEMU, register tool and models and start network services
qemu_close_connection	Close connection and terminate all services
qemu_register_slave_write	Register function to be called when a write message is received
qemu_register_slave_read	Register function to be called when a read message is received
qemu_master_write	Send a bus master write message to QEMU
qemu_master_read	Send a bus master read message to QEMU
qemu_raise_interrupt	Send a raise interrupt message to QEMU
qemu_raise_interrupt	Send a lower interrupt message to QEMU

implementation responsibilities to the tool-specific interface.

Master and interrupt message sending functions share the Master IRQ client channel, and as such, client channel access is mutually exclusive. Any of the four functions related to interrupt or bus master messages may cause the caller to sleep, which should be taken into account when implementing the simulation interface. Also, if a master read or write function is called for a local device model, those functions will call the slave registered functions instead of sending transaction messages to QEMU.

3.3 QEMU Co-simulation PLI Library

Some embedded system scenarios present difficult to meet real-time requirements, with tight task deadlines, impossible to meet without relying on hardware accelerated applications.

Typically, hardware co-processors and peripherals are developed using hardware description languages, and validated by HDL simulators. HDL designs are validated with stimuli being simulated by testbenches that use non-synthesizable constructs of the development language. However, this development approach does not account for software domain interactions, as the IP that is being developed must be wrapped in a system bus interface wrapper so it can be integrated in the target machine.

To validate integration and debug system interactions, validating both device drivers and hardware acceleration domain components, a PLI library was developed, in the context of this dissertation, that implements an interface with QEMU using the QEMU external tool library. This library allows for HDL simulators that support PLI to simulate hardware IPs integrated in QEMU machines.

3.3.1 Library Overview

Figure 3.23 presents an overview of a hardware accelerated embedded Linux case of study running on a QEMU emulation that interfaces hardware IPs simulated in ModelSim.

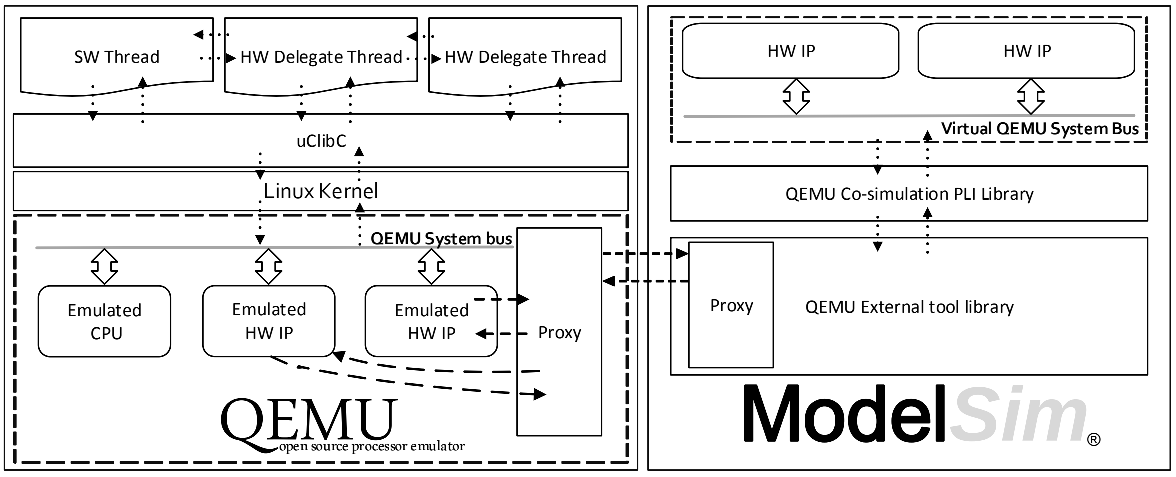


Figure 3.23: QEMU co-simulation PLI library overview

A hardware accelerated embedded application is usually composed by a collection of threads, including hardware delegate threads. Hardware delegate threads delegate processing algorithms to hardware accelerators by interacting with them through device driver system calls.

The developed library enables hardware accelerators modeled in ModelSim to be integrated into QEMU simulation, through a PLI API that supports a virtual QEMU system bus model in Verilog. Slave wrappers are used to ensure a connection from HDL models to the virtual bus, implementing logic that interfaces the virtual bus. A set of PLI system tasks and callbacks extend Verilog non-synthesizable constructs capabilities to support virtual system bus interactions with QEMU. QEMU interactions are implemented using the external tool library, benefiting from its provided abstractions.

Figure 3.24 presents a sequence diagram of a hardware IP in ModelSim being registered in QEMU and subsequently accessed in a transaction.

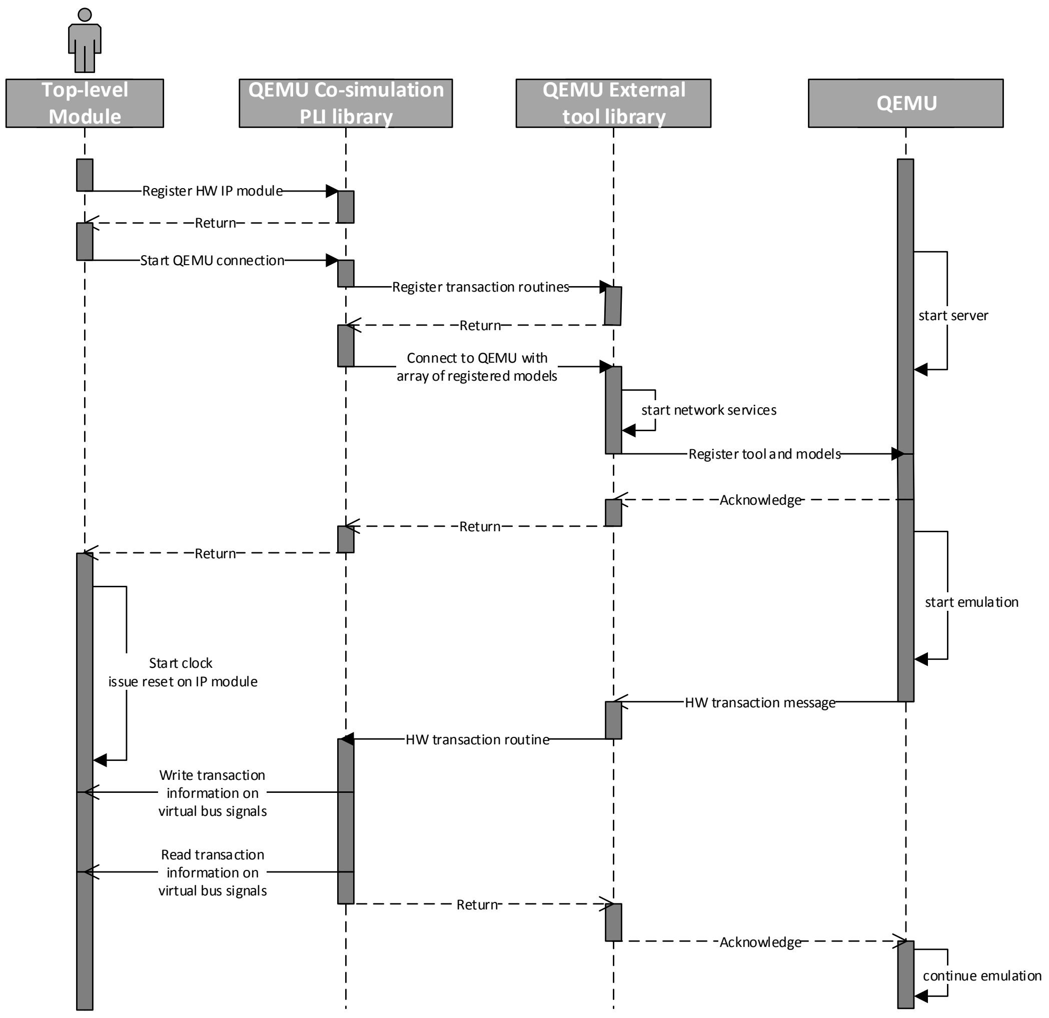

Top-level Module
QEMU Co-simulation PLI library
QEMU External tool library
QEMU
Register HW IP module
Return
Start QEMU connection
Register transaction routines
start server
Return
Connect to QEMU with array of registered models
start network services
Register tool and models
Acknowledge
Return
Return
start emulation
Start clock issue reset on IP module
HW transaction message
HW transaction routine
Write transaction information on virtual bus signals
Read transaction information on virtual bus signals
Return
Acknowledge
continue emulation

Figure 3.24: PLI library initialization and transaction sequence diagram

The top-level module must first register the HW-IP models that are going to be integrated into QEMU through system tasks provided by the PLI library, starting the connection after all models are registered. The PLI library will consequently register the slave transaction functions in the external tool library, with the latter starting connection with QEMU. Tool information is provided, along with information from all the models that were registered with the IP model registration system task.

When a hardware transaction message gets sent in QEMU emulation context, the QEMU external tool library calls the corresponding PLI library registered slave transaction function, which accesses Verilog virtual system bus signals, and manipulates them to emulate the transaction. The top-level module of the Verilog simulation contains those virtual bus signals, such as address or value of the transaction.

3.3.2 Virtual QEMU System Bus Model

To integrate QEMU system bus interactions in HDL simulation, a virtual QEMU system bus model was designed. This model is comprised of a top-level module, where the bus signals are instantiated and later manipulated by the PLI library, and IP modules that contain virtual QEMU system bus slave HW-IP models. A minimal bus interface was designed, with shared inputs and tristate buses for data output and transaction acknowledgment.

To interface the bus, a slave IP wrapper module that connects to the the bus was developed, with generative constructs being used to allow for easy wrapper adaptation when mapping a custom HW-IP. Parametrization was also used whenever possible to provide flexibility for different scenarios, such as systems with architectures that employ different word sizes. Verilog parameter names are going to be capitalized whenever mentioned.

Figure 3.25 presents an overview of the virtual bus model.

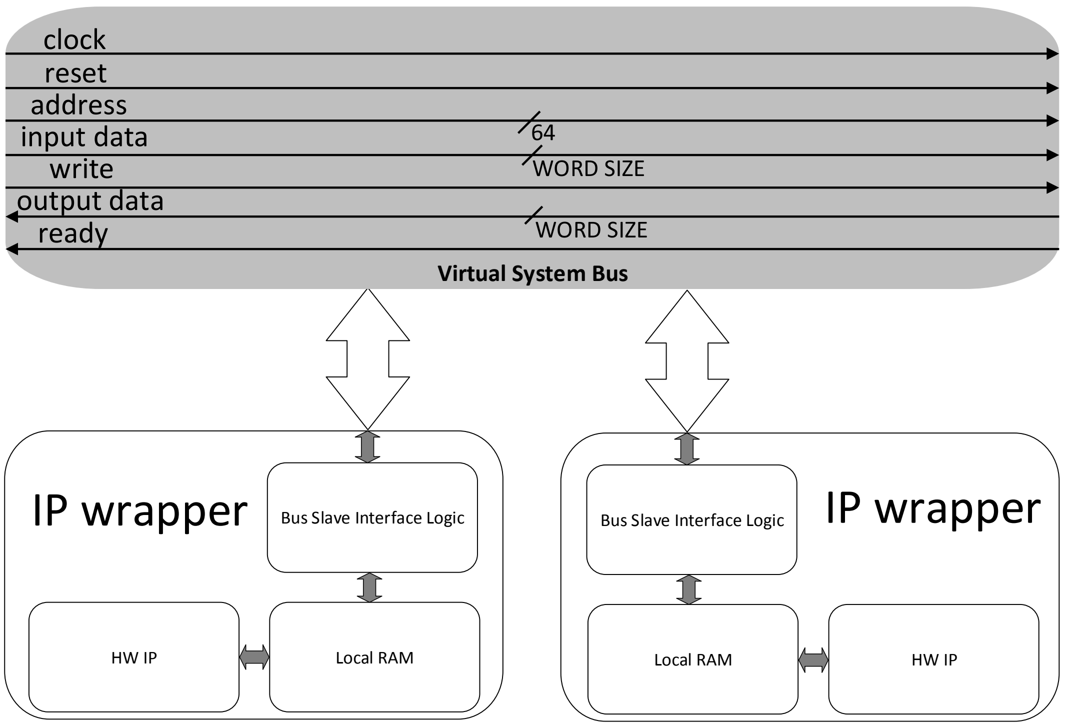


Figure 3.25: Virtual QEMU system bus overview diagram

The IP wrapper contains local RAM registers that are connected to the hardware IP's input and output ports. Consequently, transactions are performed on the local RAM registers, via bus slave interface logic. Figure 3.26 presents the bus slave interface logic of the IP wrapper.

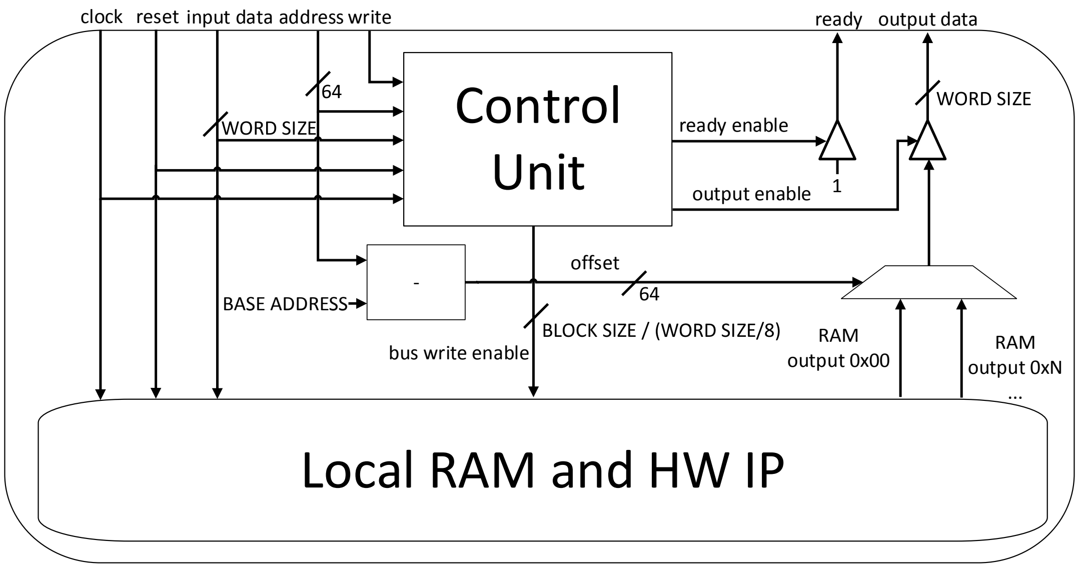


Figure 3.26: Virtual QEMU system bus slave logic diagram

The control unit interprets the bus signals, and drives the local RAM and hardware IP's datapaths. Whenever the address signal contains an address within the mapped region, the control unit will activate the bus write enable or output enable signals according to the write signal.

There is a bus write enable signal for each of the local RAM's registers, with BLOCK SIZE being the size of the RAM in bytes, and WORD the word length of each address in bits. When this signal is driven high, the corresponding local RAM register gets written with the value in input data, which is the case for write transactions. If the transaction happens to be a read transaction, the output enable signal is driven high, which activates data writing in the tri-state bus for output data.

The offset in relation to the IP's base address drives a multiplexer that selects the RAM output that is going to be written in the output.

Finally, the ready signal is triggered when the hardware transaction is completed. Figure 3.27 presents a state chart of the control unit's state machine.

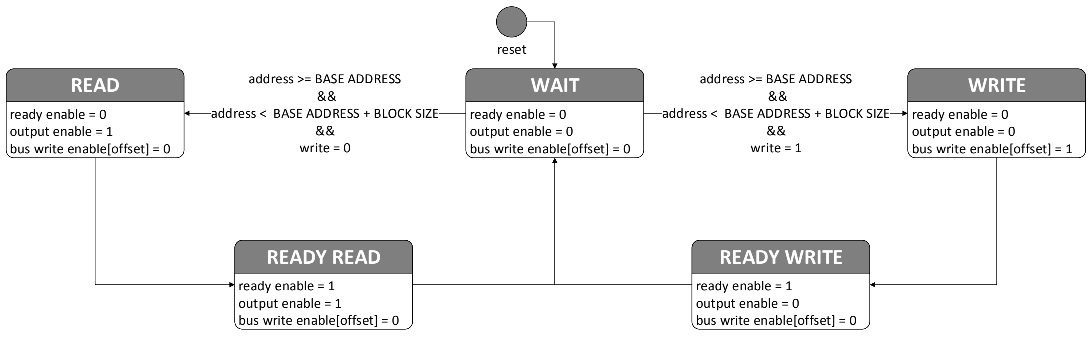


Figure 3.27: Slave control unit state chart

Since the address is 0 for a non-active bus, detection of an address within the mapped region is sufficient for transaction triggering.

All memory registers have independent inputs, allowing for IP ports to be mapped as register inputs, outputs, or both. Figure 3.28 presents a diagram of an example hardware IP with one input mapped to local RAM in address 0x00 , and an output mapped to local RAM in address 0x01.

Every register has inputs either from the data received from the bus,or from the IP, and a correspondent write enable. When there is a write transaction with incoming data, the bus write enable signal gets active for the target register, acting both as

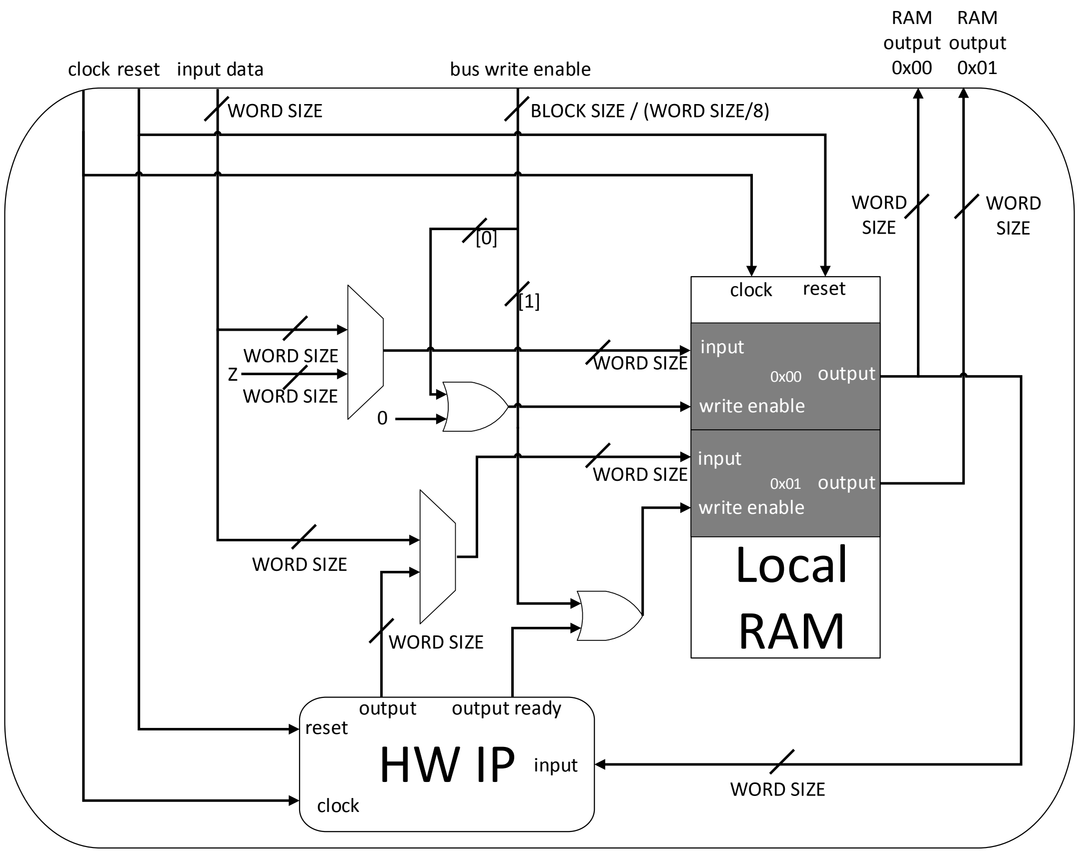


Figure 3.28: Slave local RAM and HW IP diagram

data input selector and write enable for the register. When there is no write data transaction in course, every local RAM's register input is connected to the IP's input signal, with a write enable allowing for an IP signal to write in the RAM.

IP output signals and respective write enables are generated according to the memory block size using generative Verilog constructs. If a register is to be used only for IP input mapping, such as the one in address 0x00, the corresponding IP output signal write enables should be inactive. Contrarily, the register in address 0x1 is mapped to an IP output, and as such, its IP output and respective write enable are connected to IP ports.

To better illustrate how to instantiate hardware IPs and map them using the provided wrapper, Appendix F contains a Verilog module with the wrapper and IP instantiation of the example mentioned above in section F.3. Section F.2 contains a full diagram representation of the virtual bus model, fusing all of the diagrams shown above.

3.3.3 PLI API

To allow for co-simulation between QEMU and Verilog simulation, the PLI library provides a set of system tasks. These system tasks must be called in the virtual bus model context, that is, called in an IP wrapper module or top-level bus module. Table 3.6 lists all the provided system tasks, along with the module where they're supposed to be used.

The context of where these system calls are called is important for two reasons. Firstly the top-level module must contain the virtual bus signals, which are going to be accessed by the PLI library to perform transactions. Secondly, IP wrappers must contain a set of parameters that will be used to register the model in QEMU.

Virtual bus signals are accessed through their names, therefore, the top-level module must contain the following signals that comprise the virtual bus:

- clk

  This is the system's clock. This signal will be registered in a PLI callback, and every time there's a value change in the clock, the PLI library will handle any pending transactions, accessing the other bus signals for that purpose.

- addr

Table 3.6: QEMU co-simulation PLI library API

API	**Description**	**Caller Module**
$qemu_register_model	Register a wrapper Verilog module that contains an IP to be used as external model	Top-level
$qemu_connect	Connect to QEMU and start co-simulation	Top-level
$qemu_write_bus	Perform a write transaction on QEMU's system bus	IP Wrapper
$qemu_read_bus	Perform a read transaction on QEMU's system bus	IP Wrapper
$qemu_raise_interrupt	Raise an interrupt in QEMU	IP Wrapper
$qemu_lower_interrupt	Lower an interrupt in QEMU	IP Wrapper

This is the address bus, with a maximum width 64 bits. When handling transactions, the target address is written by the PLI library on this bus.

- din

  This is the input data bus, and is only used in write transactions. Its width is parametrized, to match machine word bit length.

- write

  This signal simply indicates that the transaction in course is a write or read transaction. Value of 1 for write, 0 for read.

- dout

  This is a tristate bus so it can be shared between IPs, and is pulled down by default. It is used by IPs to output their values in a read transaction.

- ready

  This is an acknowledgement tristate signal, pulled down by default. It is used by IPs to acknowledge a finished transaction

These signals will be registered by the PLI library when $qemu_connect is called, and further manipulated when transactions are performed.

Concerning the IP wrapper, the following parameters must be set, to fully represent a model so that a call $qemu_register_model may register it:

- MEMORY_MAPPED_ADDRESS;

  This is the base address of the IP, with a maximum length of 64 bits.

- MAPPED_AREA_SIZE;

  This is the block size of the local RAM in bytes, with a maximum width of 32 bits.

- MASTER;

  This is a flag that is used to signal if the model accesses the bus as master. It is a single bit parameter, 0 for false, 1 for true.

- NAME;

  This is a string that is going to be used in QEMU as a device property.

- INTERRUPTS_SIZE;

  This is the number of interrupts that will be used by the model, and is used to request IRQ allocation inQEMU.

- INTERRUPT_N;

  This is the interrupt number that is used by the requested corresponding IRQ object in QEMU. There must be an interrupt parameter for each of the interrupts that are going to be requested, with N being the index of the interrupt. (Ex: For an INTERRUPT_SIZE of 3, there must be INTERRUPT_0, INTERRUPT_1 and INTERRUPT_2 parameters).

- WORD.

  This is the size of the target architecture's word in bits, and is going to be used as transaction word. It is configurable at instantiation with a default size of 32 bit.

To start the co-simulation, all modules that contain IPs that are going to be used as external models should be first registered through $qemu_register_model system task. Afterwards, IP's should be initialized by issuing a reset, and lastly, $qemu_connect should be called to establish connection with QEMU. This call may be optionally provided with QEMU's IP address or port.

All of the system tasks' compiletf and calltf routine flowcharts may be found in

Appendix F.6.

3.3.4 Transaction Handling

QEMU co-simulation PLI library uses the external tool library as well as the top-level bus signals to implement transactions. The slave functions that PLI implements register transaction messages received from QEMU, and synchronize transaction information by issuing requests to a simulation callback. This simulation callback is called at each positive edge of the simulation clock, and is able to change internal simulation values. If there are any pending transaction requests, the virtual bus' signals will be manipulated in the callback, effectively carrying out the transactions.

A condition variable synchronizes the slave function and the simulation callback, being associated with a state that presents the virtual bus state. Figure presents flowcharts of the slave function and simulation callback behavior in each positive clock transition regarding transactions, respectively.

The simulation time that the transaction took is retrieved along with the transaction information, and a broadcast is sent to wake-up all slave functions that should be awaiting data. This is necessary, because despite multiple slave transactions from QEMU not being possible, there could be multiple slave functions waiting on the condition variable given that the QEMU external tool library implements bus master accesses on local models by calling the slave registered function.

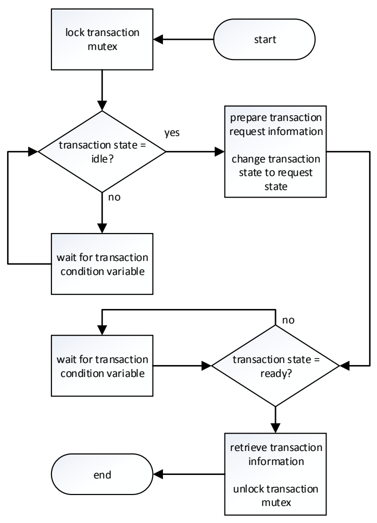
3.3.5 Bus Master API

Bus master operations in QEMU's system bus are supported, through a system task API. Table 3.7 presents the available bus master API.

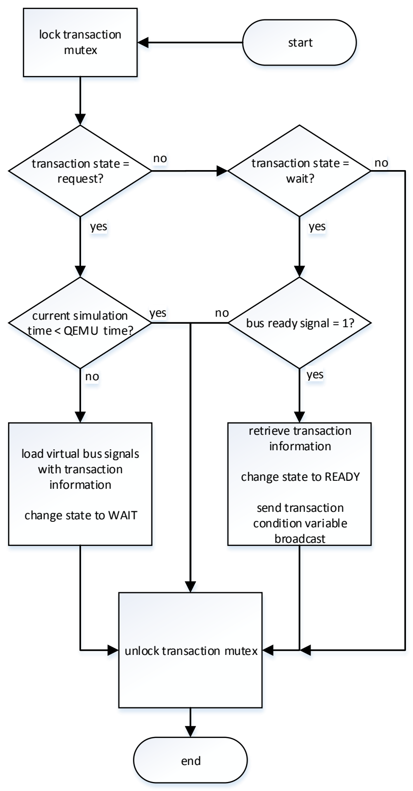
Table 3.7: Bus master API

API	Description
$qemu_write_bus(address, offset, value, WORD)	Master write
$qemu_read_bus(address, offset, return value reg/wire, WORD)	Master read

These system tasks implement calls to their analogous counterparts of the external tool library. Since the client channel for the interrupt and master operation


lock transaction mutex
start
transaction state = idle?
yes
no
prepare transaction request information
change transaction state to request state
wait for transaction condition variable
no
wait for transaction condition variable
transaction state = ready?
retrieve transaction information
unlock transaction mutex
end

(a) Slave transaction function flowchart

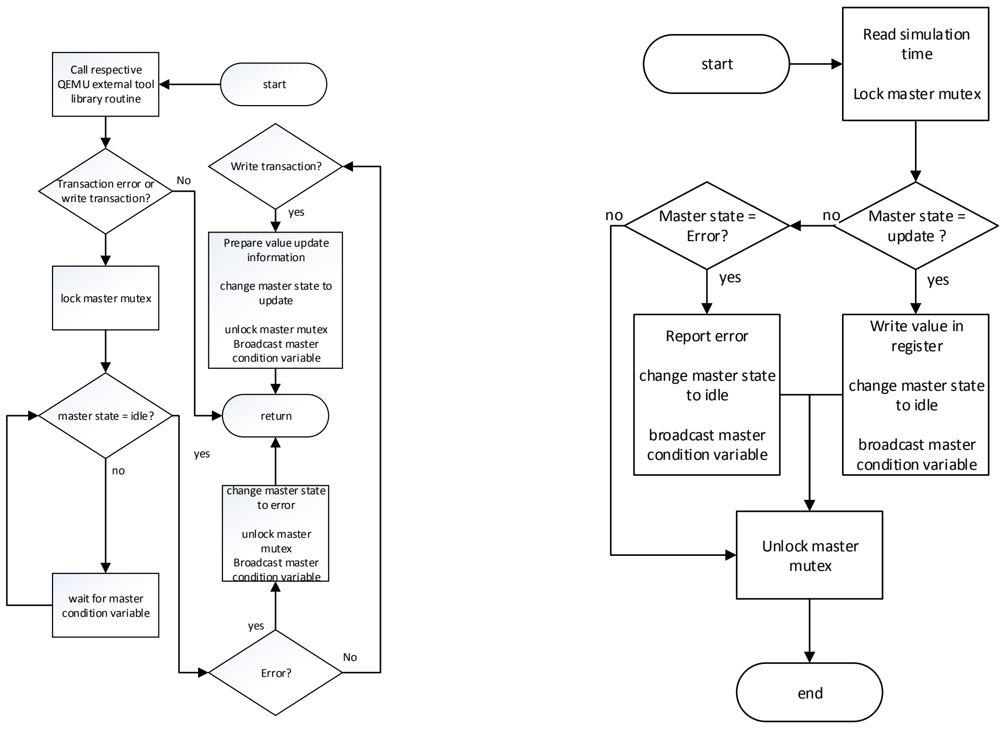

lock transaction mutex
start
transaction state = request?
no
transaction state = wait?
no
yes
yes
current simulation time < QEMU time?
yes
no
bus ready signal = 1?
no
yes
load virtual bus signals with transaction information
change state to WAIT
retrieve transaction information
change state to READY
send transaction condition variable broadcast
unlock transaction mutex
end

(b) Clock change simulation callback transaction handling flowchart

Figure 3.29: PLI library transaction flowcharts

requests is shared and atomically protected, the system tasks launch a thread that will call the co-simulation library functions, enabling the simulation to continue running if the thread is put to sleep. Another possible scenario where the thread is put to sleep is when the target device for the transaction is a local model, which causes slave registered routine execution, and further suspension until the transaction is handled by the clock edge simulation callback. Appendix F.6 contains flowcharts for the system tasks' calltf and compiletf routines.

Read bus master accesses must return a value to be written in a target register or wire. However, since a thread that maybe put to sleep is launched, the target register or wire is passed as an argument to the system task so a value change can be scheduled on the return value wire or register when the read transaction is complete. This change is also made in the clock change simulation callback. The request mechanism is not unlike a slave transaction, with a condition variable being associated with a state variable to request changes on a register/wire simulation object. Figure 3.30a presents a flowchart of the master thread requesting a value change upon a concluded read transaction, and figure 3.30b presents a flowchart of the clock change simulation callback answering requests process.


start
Call respective QEMU external tool library routine
Transaction error or write transaction?
No
lock master mutex
master state = idle?
no
yes
wait for master condition variable
Write transaction?
yes
Prepare value update information
change master state to update
unlock master mutex
Broadcast master condition variable
return
change master state to error
unlock master mutex
Broadcast master condition variable
yes
Error?
No
start
Read simulation time
Lock master mutex
Master state = update ?
no
Master state = Error?
no
yes
Report error
change master state to idle
broadcast master condition variable
yes
Write value in register
change master state to idle
broadcast master condition variable
Unlock master mutex
end

(a) Master thread flowchart

(b) Clock change simulation callback master update flowchart

Figure 3.30: PLI library bus master flowcharts

3.3.6 Interrupt API

Triggering interrupts in QEMU's machine is supported, through a system task API. Table 3.8 presents the available interrupt API.

Table 3.8: Interrupt API

API	Description
$qemu_raise_interrupt(interrupt index)	Trigger an interrupt
$qemu_lower_interrupt(interrupt index)	Clear an interrupt

Like bus master operations, these system tasks call their external tool library counterparts, launching a thread for some of the same reasons. The system tasks' argument should be the interrupt index that was registered in the parameters (0 for INTERRUPT_0, 1 for INTERRUPT_1, etc...).

To better use this API generative constructs are used to generate signals that call these system tasks on the provided slave IP wrapper. For each requested interrupt, there is a signal to enable and a signal to disable interrupts. Positive edges on the interrupt enable signal will cause the $qemu_raise_interrupt system task to be called, and negative edges on the interrupt disables signal will cause the $qemu_lower_interrupt system task to be called. This is done so that interrupt activation and clearing are not intrinsically connected, allowing them to be mapped to different signals or registers. An example of its use is provided in appendix F in section F.4, where an IP output signal is mapped to an interrupt enable signal, and the corresponding interrupt disable signal is mapped to local RAM, allowing software to clear the interrupt flag upon interrupt handling. Compiletf and calltf routines for the interrupt API system tasks are available in appendix in section F.6.

Chapter 4

Case of Study

This chapter describes the case of study that was chosen to provide stimuli for system design and demonstrate the developed work in a practical context. The case of study is integrated in the power electronics field, consisting in an application that acquires data from voltage and current sensors and monitors the instantaneous active and reactive powers, as defined by the pq theory(Akagi et al., 2007).

This system will be modeled using a multi-threading paradigm, profiled in the host and subsequently hardware accelerated using a design flow aided by the developed simulation extensions, as proposed in chapter 3. As such HW IPs selected for acceleration will be validated firstly using QEMU plugins and lastly as HDL models.

4.1 Instantaneous Active and Reactive Powers Monitoring Application

To provide a practical case of study, an instantaneous active and reactive power monitoring system was selected. Active and reactive power monitoring of an electrical installation is used to determine what part of the power that is being consumed actively contributes to energy consumption in the load. Reactive power is an undesired component of power that does not contribute to energy consumption, but is nevertheless provided by the electrical grid, and therefore taxed if significant. The pq theory(Akagi et al., 2007) defines instantaneous active and reactive powers through the use of the Clarke transform, which the monitoring application will use to monitor instantaneous active and reactive powers. Appendix G contains the mathematical fundamentals of the pq theory. Figure 4.1 represents an overview diagram of the monitoring system.

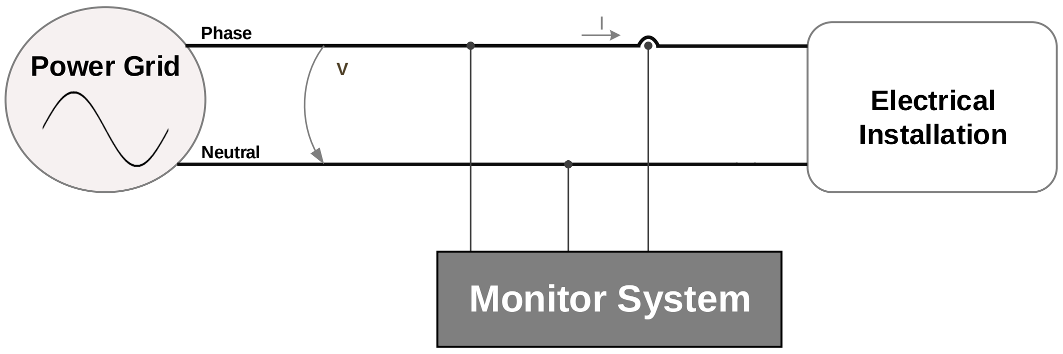


Figure 4.1: Monitoring system overview diagram

To monitor power in the installation, two channels are necessary, to acquire current and voltage signals. These signals will be then processed and used to calculated active and reactive powers for each instant, monitoring power consumption by the electrical installation. Given that the pq theory is defined for a 3-phase electrical system and the system being modeled is a single phase system, a 3-phase balanced voltage and current system must be virtualized using 120 degree angular displacement on the grid's original phase, generating phases B and C from the same samples. Figure 4.2 represents a virtual 3-phase voltage system generated from a single phase voltage system.

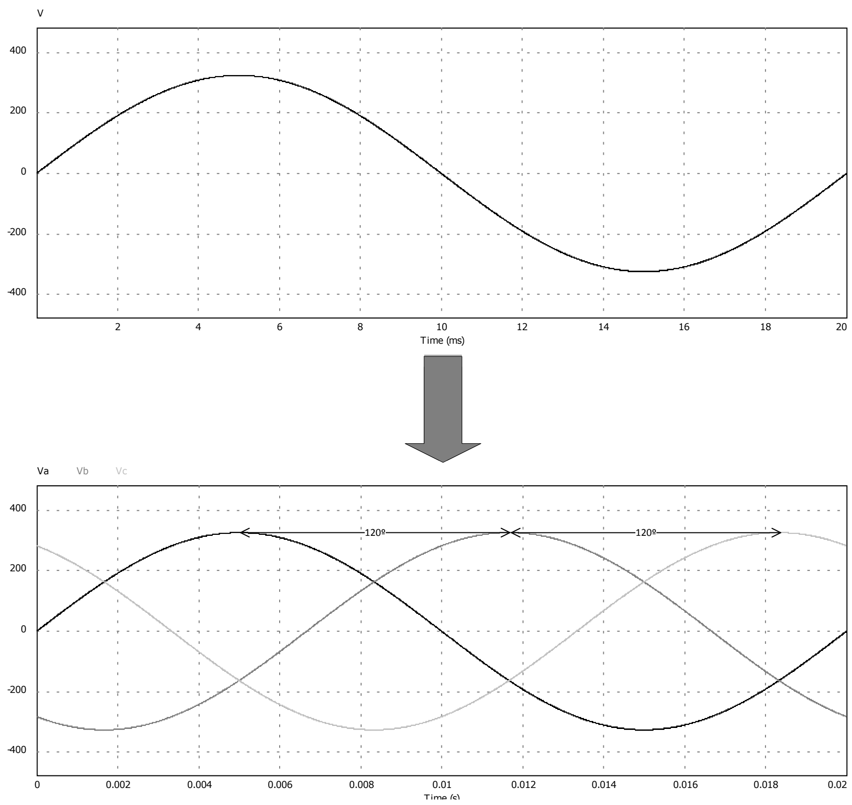


Figure 4.2: 3-phase virtualization

Phase B is 120^o degrees late, or 240^o early, meaning samples being used as phase B should be offset two thirds in the sample buffer. Conversely, phase C is 120 early, or 240^o late, meaning samples being used as phase C should be offset one third in the sample buffer. The control blocks for the monitoring application are represented in figure 4.3.

Current and voltage inputs are virtualized in the virtual 3-phase mirror block. The Clarke Transform is applied afterwards, with voltages and currents being changed into an α-β-*0* reference system, allowing for instantaneous powers to be calculated according to the pq theory. This block however contains an extra operation which is the division of both powers by 3. Due to the system's 3-phase virtualization, powers are supposedly consumed by three identical loads that are mirrored after the actual load, resulting in power values that are an exact triple of the actual powers.

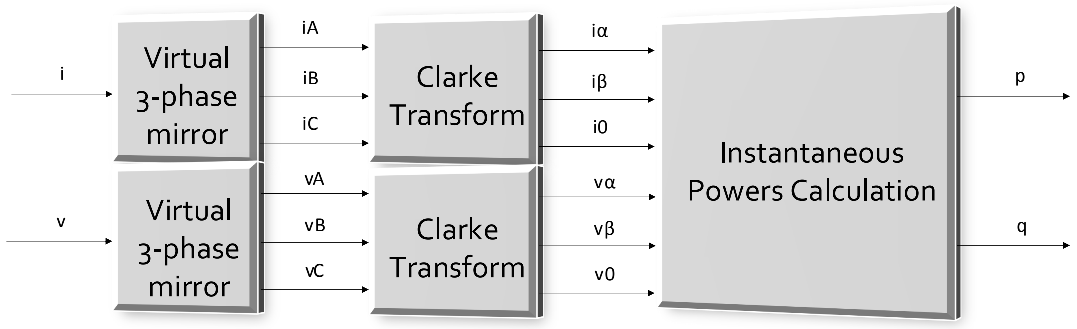


Figure 4.3: Single-phase pq theory instantaneous powers blocks

4.2 System Modeling

To model the system, a C++ application was developed. Tasks identified in the system's blocks were separated into different threads. Similar blocks were grouped in the same thread, such as current and voltage acquisiton, or clarke transforms for current and voltage values. Queues were used to store data and allow threads to process data independently, breaking the sequentiality between them. Figure 4.4 presents a task graph of the modeled system.

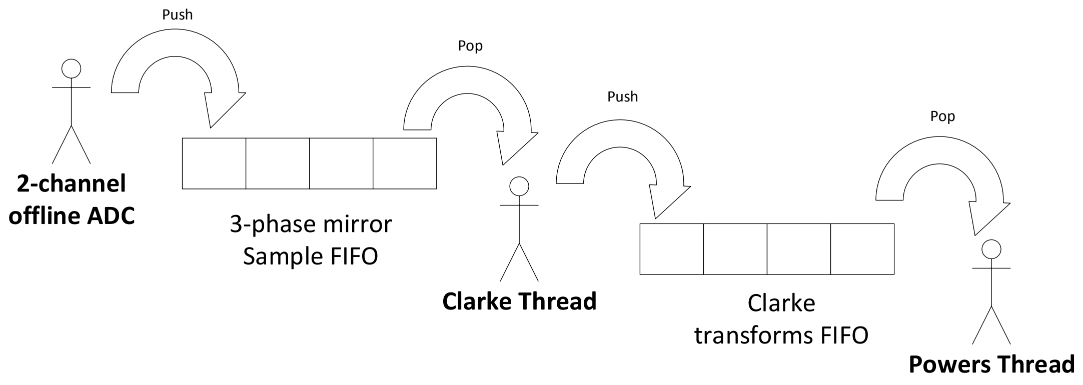


Figure 4.4: Monitoring application task graph

To implement tasks in software, POSIX threads were used, as well as POSIX synchronization mechanisms, namely mutexes and condition variables. Mesa semantics are used to implement producer-consumer synchronization. Figure 4.5 presents a class diagram of the developed application.

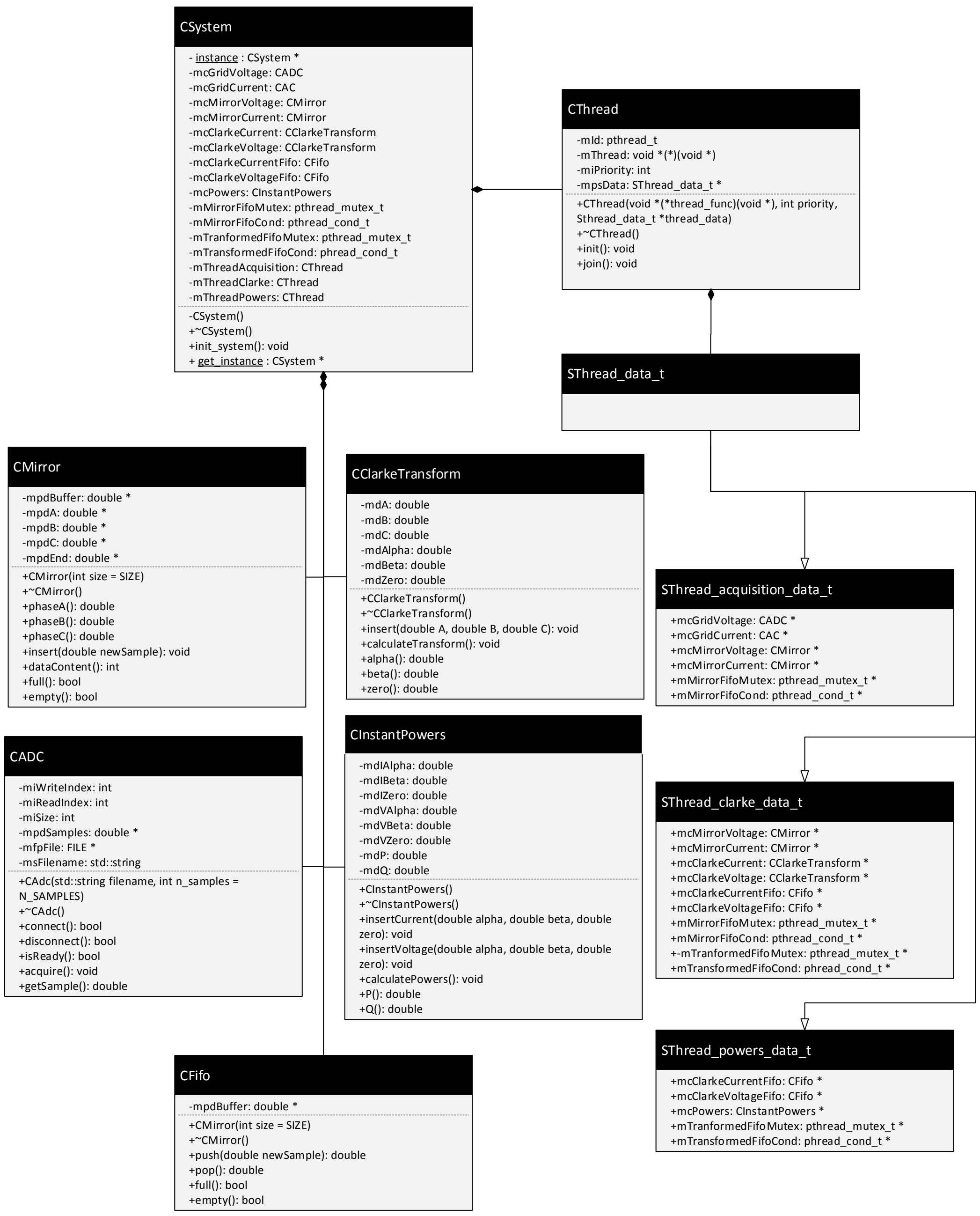

CSystem
- instance : CSystem *
-mcGridVoltage: CADC
-mcGridCurrent: CAC
-mcMirrorVoltage: CMirror
-mcMirrorCurrent: CMirror
-mcClarkeCurrent: CClarkeTransform
-mcClarkeVoltage: CClarkeTransform
-mcClarkeCurrentFifo: CFifo
-mcClarkeVoltageFifo: CFifo
-mcPowers: CInstantPowers
-mMirrorFifoMutex: pthread_mutex_t
-mMirrorFifoCond: pthread_cond_t
-mTranformedFifoMutex: pthread_mutex_t
-mTransformedFifoCond: phread_cond_t
-mThreadAcquisition: CThread
-mThreadClarke: CThread
-mThreadPowers: CThread
-CSystem()
+~CSystem()
+init_system(): void
+ get_instance : CSystem *
CThread
-mId: pthread_t
-mThread: void *(*)(void *)
-miPriority: int
-mpsData: SThread_data_t *
+CThread(void *(*thread_func)(void *), int priority, Sthread_data_t *thread_data)
+~CThread()
+init(): void
+join(): void
SThread_data_t
CMirror
-mpdBuffer: double *
-mpdA: double *
-mpdB: double *
-mpdC: double *
-mpdEnd: double *
+CMirror(int size = SIZE)
+~CMirror()
+phaseA(): double
+phaseB(): double
+phaseC(): double
+insert(double newSample): void
+dataContent(): int
+full(): bool
+empty(): bool
CClarkeTransform
-mdA: double
-mdB: double
-mdC: double
-mdAlpha: double
-mdBeta: double
-mdZero: double
+CClarkeTransform()
+~CClarkeTransform()
+insert(double A, double B, double C): void
+calculateTransform(): void
+alpha(): double
+beta(): double
+zero(): double
SThread_acquisition_data_t
+mcGridVoltage: CADC *
+mcGridCurrent: CAC *
+mcMirrorVoltage: CMirror *
+mcMirrorCurrent: CMirror *
+mMirrorFifoMutex: pthread_mutex_t *
+mMirrorFifoCond: pthread_cond_t *
CADC
-miWriteIndex: int
-miReadIndex: int
-miSize: int
-mpdSamples: double *
-mfpFile: FILE *
-msFilename: std::string
+CAdc(std::string filename, int n_samples = N_SAMPLES)
+~CAdc()
+connect(): bool
+disconnect(): bool
+isReady(): bool
+acquire(): void
+getSample(): double
CInstantPowers
-mdIAlpha: double
-mdIBeta: double
-mdIZero: double
-mdVAlpha: double
-mdVBeta: double
-mdVZero: double
-mdP: double
-mdQ: double
+CInstantPowers()
+~CInstantPowers()
+insertCurrent(double alpha, double beta, double zero): void
+insertVoltage(double alpha, double beta, double zero): void
+calculatePowers(): void
+P(): double
+Q(): double
SThread_clarke_data_t
+mcMirrorVoltage: CMirror *
+mcMirrorCurrent: CMirror *
+mcClarkeCurrent: CClarkeTransform *
+mcClarkeVoltage: CClarkeTransform *
+mcClarkeCurrentFifo: CFifo *
+mcClarkeVoltageFifo: CFifo *
+mMirrorFifoMutex: pthread_mutex_t *
+mMirrorFifoCond: pthread_cond_t *
+-mTranformedFifoMutex: pthread_mutex_t *
+mTransformedFifoCond: phread_cond_t *
SThread_powers_data_t
+mcClarkeCurrentFifo: CFifo *
+mcClarkeVoltageFifo: CFifo *
+mcPowers: CInstantPowers *
+mTranformedFifoMutex: pthread_mutex_t *
+mTransformedFifoCond: phread_cond_t *
CFifo
-mpdBuffer: double *
+CMirror(int size = SIZE)
+~CMirror()
+push(double newSample): double
+pop(): double
+full(): bool
+empty(): bool

Figure 4.5: Monitoring application UML

4.3 Results

A simple resistive load of 4 Ω was used to validate the monitoring application. The simulated electrical system is identical to the Portuguese electrical system, 325 V peak value with a frequency of 50 Hz. Figure 4.6 presents the schematic of the PSIM circuit used to generate the grid samples used in the monitoring application.

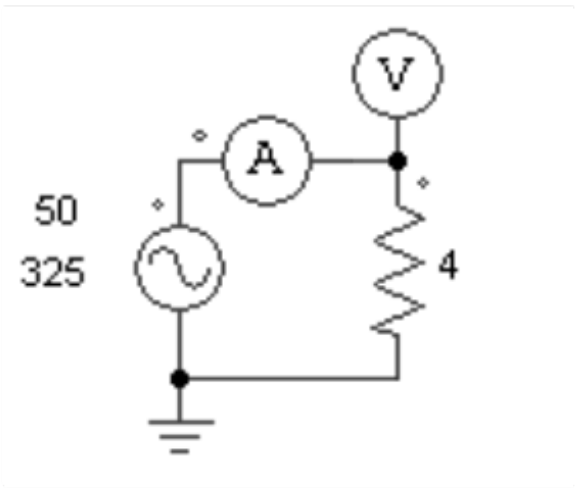


Figure 4.6: PSIM single-phase grid with 4 Ohm resistive load

4.3.1 Profiling

The software-only application was profiled using a non-intrusive profiling tool, Oprofile. Figure 4.7 presents the profiling statistics of the application threads for five executions of 10 minutes. Profiling identified the Clarke transform thread as the most critical thread in terms of processing. As such, it was selected as candidate for hardware migration.

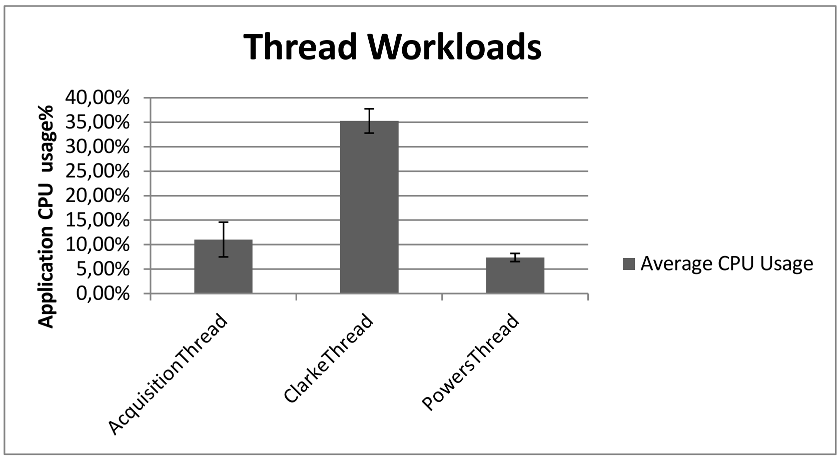


Figure 4.7: Monitoring application profile

4.3.2 Software-only and Hardware-Software

Results of the instantaneous powers, that is, outputs for the monitoring application when stimulated with the previously shown grid sampling scenario, were collected for the software-only implementation of the application as well as the hardware-software implementation with the Clarke transform thread migrated to hardware. Results for the hardware accelerated incarnation of the application were collected via QEMU-ModelSim co-simulation using the corresponding developed extensions.

Figures 4.8 and 4.9 present the results for both software-only and hardware-software applications respectively.

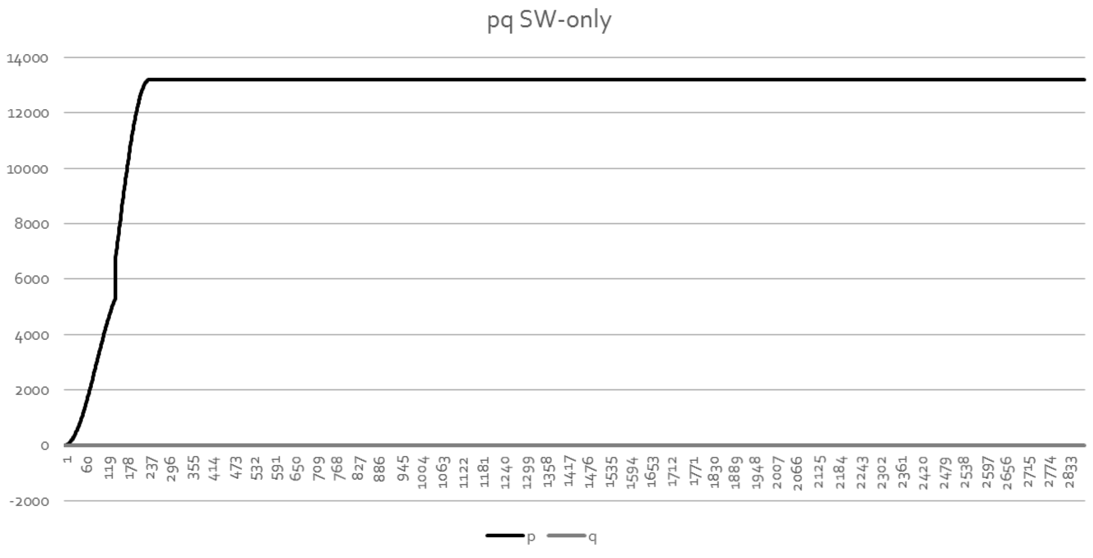


Figure 4.8: p and q values outputs of the software-only application

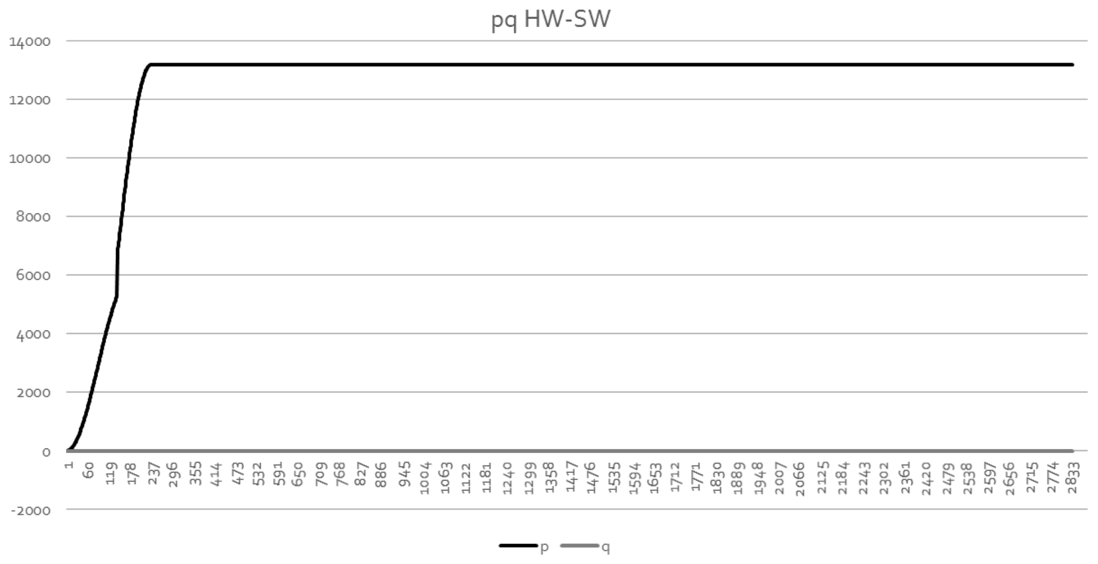


Figure 4.9: p and q values outputs of the hardware-software application

Roughly four grid cycles are represented, with a cycle consisting of 720 samples. During the first third of the first cycle, the values are not valid, as the mirror buffer is not filled enough to correctly virtualize the 3-phase system.

The q value is obviously 0, given that there is no energy storing elements in the load. With a resistive load of 4 Ω, p should be 13225 as $p = \frac{(325/\sqrt{2})^2}{4}$. The results collected for both applications plot similar looking graphics, with the hardware-software application producing the same outputs as the software-only application, thus validating the functionality of the developed extensions.

Chapter 5

Conclusion

This chapter concludes the dissertation, tying up what was previously shown by reviewing the developed work, and bringing up what interesting work could be done in the future.

This project required a varied set of skills, given the ambitiousness and broad spectrum of the project, encompassing overlapping necessary knowledge areas. Knowledge areas included embedded system design and platform bringup, Linux device driver development, HDL dedicated co-processor design, dynamic library integration and network sub-systems, and HDL simulation interface frameworks. QEMU, being an undocumented and rather large open-source program, clouded the project with a lot of uncertainty at its beginning stages, not being clear if the initial vision and planning was realistic or compatible with the author's unfamiliarity with QEMU itself.

For the reasons mentioned above and due to the project's obtained results, it is concluded that the objectives of this project were accomplished.

5.1 Developed Work

Concerning developed work, the simulation extensions allow for initial software application trials using rapid deployment of behavioral hardware IPs as plugins. Applications and respective device drivers may be developed and debugged with experimentation on possible acceleration scenarios without committing to their HDL implementations. After, or during this phase, HDL implementation can be carried out with further validation and integration using co-simulation between software and hardware acceleration domains. However the HDL simulation model that was used doesn't model any system bus interactions accurately, as the bus that connects all peripherals is a virtual bus that doesn't emulate any concrete system bus behavior. Real system buses will most surely be slower, taking more clock cycles in hardware transactions.

5.2 Future Work

Concerning future work, this project opens up interesting possibilities in carrying out further improvements, resulting in better design flows that aim to help minimize development cycles.

Firstly, commonly used SoC-based platforms could be added to QEMU, adding base design machines for boards such as Digilent ZYBO or ZedBoard with minimal hardware device instantiation, leaving most address space free for plugin devices or external models. Dynamic device address and interrupt mapping support could be also added for plugins and external models, enabling the developer to implement plugins and external models without concerning itself with target architectures. This would mean of course, using hotplugging mechanisms in device drivers for these devices.

Secondly, FDT generated machines could be explored, improving their utility by including plugin devices or external models. It would be interesting if FDT machines, while parsing the device tree and instantiating devices would look for plugins or external models instead of ignoring unrecognized devices, which is the current behavior. With this feature, it would be possible to use the same DTBs generated for platform deployment by platform generation tools, such as Vivado Design Suite or Xilinx Platform Studio, in QEMU simulation, with devices being loaded from plugins or external models.

Thirdly, other simulation domains could be integrated in this co-simulation model, allowing for data exchange between domain-specific simulators such as PSIM or SPICE simulators and QEMU devices or HDL models. Full system validation would be a very appealing feature while developing power electronics scenarios, as power stage and analog hardware must be also debugged and validated, and as it is this process is usually not done integrating embedded software and hardware acceleration domains. Also virtual the bus model could be modified to implement cycle-accurate emulation for system buses commonly used in target architectures, such as **A**dvanced e**X**tensible **I**nterface (AXI), **P**rocessor **L**ocal **B**us (PLB) or **P**eripheral **C**omponent **I**nterconnect **e**xpress (PCIe). Before platform deployment, hardware IPs are usually wrapped in a slave wrapper for the specific architecture's system bus. As it is, the currently supported design flow partially helps system wide validation, but mistakes made when mapping the hardware IP in the target system bus may go by unnoticed.

Fourthly, QEMU co-simulation PLI could be ported to DPI, or at least adapted according to DPI's possibilities. Xilinx's standard Verilog simulators do not support PLI, and integrating QEMU connection in Vivado Simulator or Isim would be a nice feature.

Finally, the existing extensions' implementations could be enhanced, adding support for port-mapped I/O on plugins and external models, allowing to simulate port-mapped accelerators with low latency dedicated interfaces such as **D**evice **C**ontrol **R**egister (DCR) or **F**ast **S**implex **L**ink (FSL) buses. Also, adapting these extensions to follow FMI standards would allow for a more portable interface, linking up with other simulators of various domains, since there are many simulation tools in other domains that support this standard.

Bibliography

H. Akagi, E. Hirokazu, and M. Aredes, *Instantaneous power theory and applications to power conditioning.* Wiley-IEEE Press, 2007.

M. Armbruster, "QEMU interface introspection: From hacks to solutions," 2015. [Online]. Available: https://drive.google.com/file/d/0BzyAwvVlQckeWW5DRldRU2tKYlU/view

F. Balducci, "OpenRisc Verilog simulation of serial port communication," 2009. [Online]. Available: https://balau82.wordpress.com/2009/12/17/openrisc-verilog-simulation-of-serial-port-communication/

T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, a. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J. V. Peetz, and S. Wolf, "The Functional Mockup Interface for Tool independent Exchange of Simulation Models," *8th International Modelica Conference 2011*, pp. 173–184, 2009. [Online]. Available: http://www.ep.liu.se/ecp_article/index.en.aspx?issue=76;article=17

Free Electrons, "Buildroot, Making Embedded Linux Easy." [Online]. Available: http://buildroot.uclibc.org

T. Huffmire, C. Irvine, T. D. Nguyen, T. Levin, R. Kastner, and T. Sherwood, *Handbook of FPGA Design Security.* Springer, 2010.

J. Liaw, "QEMU Binary Translations," 2014. [Online]. Available: http://pt.slideshare.net/RampantJeff/qemu-binary-translation

M. Marazakis, "QEMU: Architecture and Internals Lecture for the Embedded Systems Course." [Online]. Available: http://www.csd.uoc.gr/~hy428/reading/qemu-internals-slides-may6-2014.pdf

C. Maxfield, *FPGAs: World Class Designs.* Newnes, 2009, vol. 1.

T. Noergaard, *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*, 2nd ed. Newnes, 2013.

P. Oliveira, “Sistema de Aquisição de Sinais em Tempo Real Baseado em Linux,” Master Thesis, Universidade do Minho, 2013.

M. Otter, T. Blochwitz, and M. Arnold, *Functional Mockup Interface for Model Exchange and Co-Simulation, Version 2.0*, Modelica Association Project "FMI", 2013.

T. Petazzoni, “Device Tree for Dummies.” [Online]. Available: https://events.linuxfoundation.org/sites/events/files/slides/petazzoni-device-tree-dummies.pdf

Documentation/GettingStartedDevelopers. [Online]. Available: http://wiki.qemu.org/Documentation/GettingStartedDevelopers

V. Silva, “Sistema de Aquisição de Dados Tempo Real baseado em Linux,” Master’s thesis, Universidade do Minho, 2011.

S. Sutherland, *The Verilog PLI Handbook: User’s Guide and Comprehensive Reference on the Verilog Programming Language Interface*, 2nd ed. Kluwer Academic Publishers, 2002, no. 1.

——, “The Verilog PLI Is Dead (maybe) Long Live The SystemVerilog DPI !” in *Synopsis User Group San Jose*, no. March, 2004, p. 18.

J. Turley, *The Essential Guide to Semiconductors*, 1st ed. Prentice Hall, 2002.

C. Wen-Ren, “QEMU Detailed Study,” 2011. [Online]. Available: https://lists.gnu.org/archive/html/qemu-devel/2011-04/pdfhC5rVdz7U8.pdf

Xilinx, “Zynq UltraScale+ MPSoC.” [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

W. M. Zabołotny, “Development of embedded PC and FPGA based systems with virtual hardware,” in *Proc. SPIE 8454, Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments*, 2012.

Appendix A

Linux Support Material

A.1 Real-time Linux Patching, Compilation and QEMU Boot with Buildroot

In this section, detailed steps will be shown on how to compile a real-time Linux kernel using Buildroot, presenting the available real-time solutions. It will also be shown how to generate a kernel image and root filesystem with an example cross-compiled C++ hello world application, as well as how to use QEMU to boot the kernel and execute the application.

A.1.1 Buildroot Installation

The latest stable buildroot version can be downloaded from:

http://buildroot.uclibc.org/downloads/

This location also maintains also stable versions of Buildroot, as well as **R**elease **C**andidate (RC) versions, although their use is not encouraged. As of this document's elaboration, the latest stable release is 2015.11 and it will be the one used throughout this appendix.

To download Buildroot:

```
$ wget http://buildroot.uclibc.org/downloads/buildroot
   -2015.11.tar.gz
```

Afterwards, extract it to a desired directory:

```
$ tar -xvf buildroot-2015.11.tar.gz -C "target folder"
```

The result of the extraction is a folder called "buildroot-2015.11" with the set of makefiles and scripts that comprise Buildroot. However, only the essential core is there, with all toolchain, Linux sources and other services related to target system generation being downloaded as needed. As such, all output produced is maintained in "buildroot-2015.11/output". However, Buildroot supports and encourages keeping directories for output generation outside of Buildroot's main folder, as several output directories may be maintained for separate architectures and target systems. An example of the directory organization may as follows:

```
working directory
├Buildroot 2015.11
└Buildroot outputs
  ├Zedboard output
  ├Atmel SAMA5D3 Xplained output
  └BeagleBone Black output
```

The following command generates an output on an external directory:

```
Buildroot 2015.11$ make O="external output directory"
```

Afterwards, make commands may be ran directly from the output directory.

A.1.2 Real-time Linux Compilation

Buildroot provides a set of default configurations for a wide number of supported boards. To list the supported defconfigs run:

```
$ make list-defconfigs
```

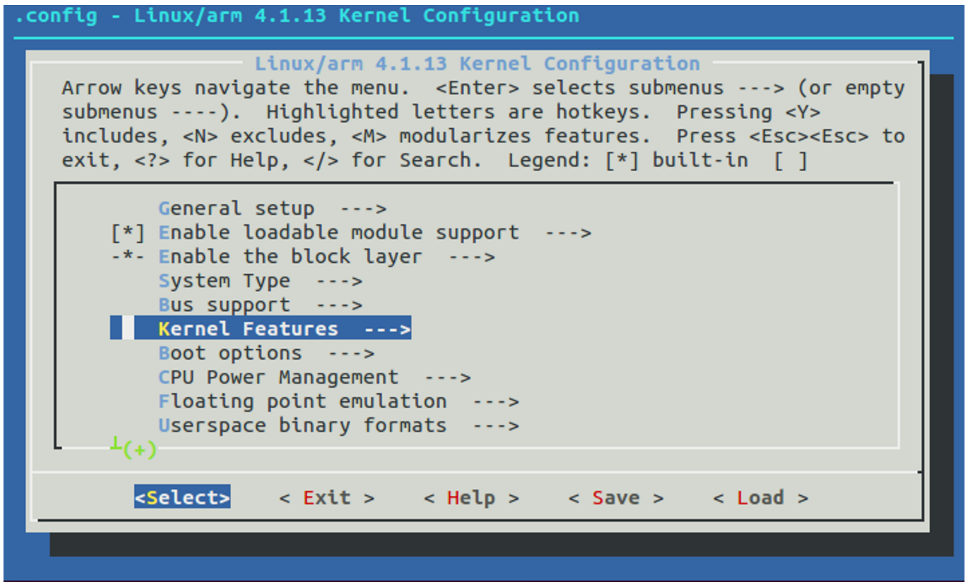
For this example, the following defconfig will be used:

```
$ make qemu_arm_vexpress_defconfig
```

To access kernel-specific configuration, a graphical menu can be prompted using:

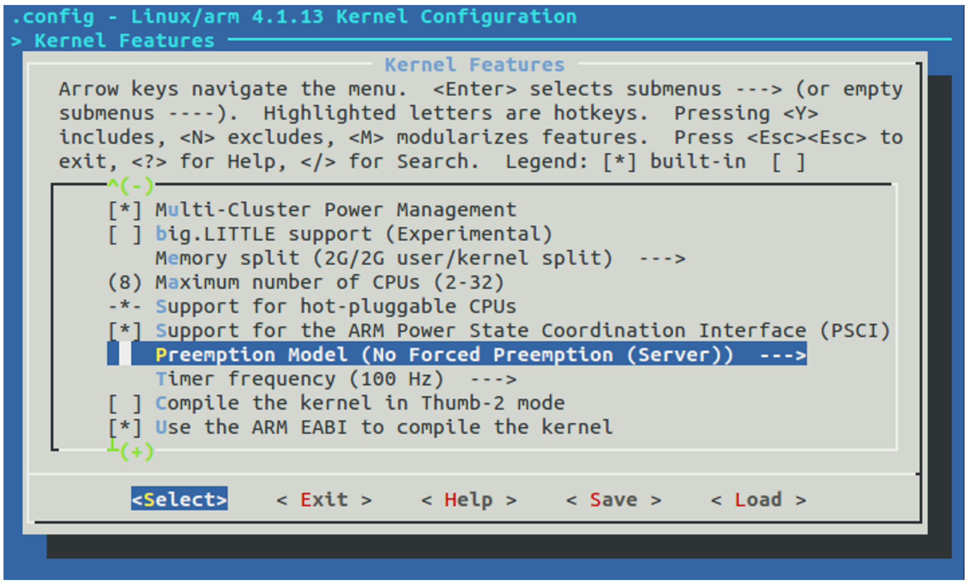
```
$ make linux-menuconfig
```

By accessing the kernel features, it is possible to observe that no real-time preemption is available



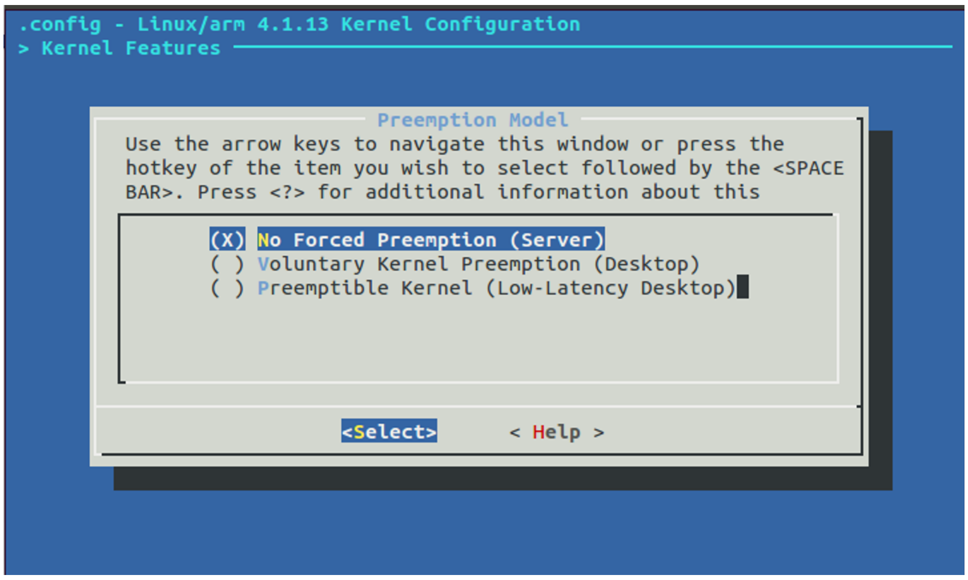
```
.config - Linux/arm 4.1.13 Kernel Configuration
                    Linux/arm 4.1.13 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]

          General setup  --->
      [*] Enable loadable module support  --->
      -*- Enable the block layer  --->
          System Type  --->
          Bus support  --->
          Kernel Features  --->
          Boot options  --->
          CPU Power Management  --->
          Floating point emulation  --->
          Userspace binary formats  --->
      (+)

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure A.1: Linux menuconfig kernel configuration



```
.config - Linux/arm 4.1.13 Kernel Configuration
> Kernel Features
                         Kernel Features
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]
      ^(-)
      [*] Multi-Cluster Power Management
      [ ] big.LITTLE support (Experimental)
          Memory split (2G/2G user/kernel split)  --->
      (8) Maximum number of CPUs (2-32)
      -*- Support for hot-pluggable CPUs
      [*] Support for the ARM Power State Coordination Interface (PSCI)
          Preemption Model (No Forced Preemption (Server))  --->
          Timer frequency (100 Hz)  --->
      [ ] Compile the kernel in Thumb-2 mode
      [*] Use the ARM EABI to compile the kernel
      (+)

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure A.2: Linux menuconfig kernel features



```
.config - Linux/arm 4.1.13 Kernel Configuration
> Kernel Features
                         Preemption Model
   Use the arrow keys to navigate this window or press the
   hotkey of the item you wish to select followed by the <SPACE
   BAR>. Press <?> for additional information about this

           (X) No Forced Preemption (Server)
           ( ) Voluntary Kernel Preemption (Desktop)
           ( ) Preemptible Kernel (Low-Latency Desktop)

                     <Select>      < Help >
```

Figure A.3: Linux menuconfig preemption

To enable real-time preemption, a patch is maintained in kernel.org, formally known as the RT-Preempt Patch. The latest stable version of the RT-Preempt patch can be found in:

https://www.kernel.org/pub/linux/kernel/projects/rt

Buildroot provides support to patch the kernel. To launch Buildroot configuration menu:

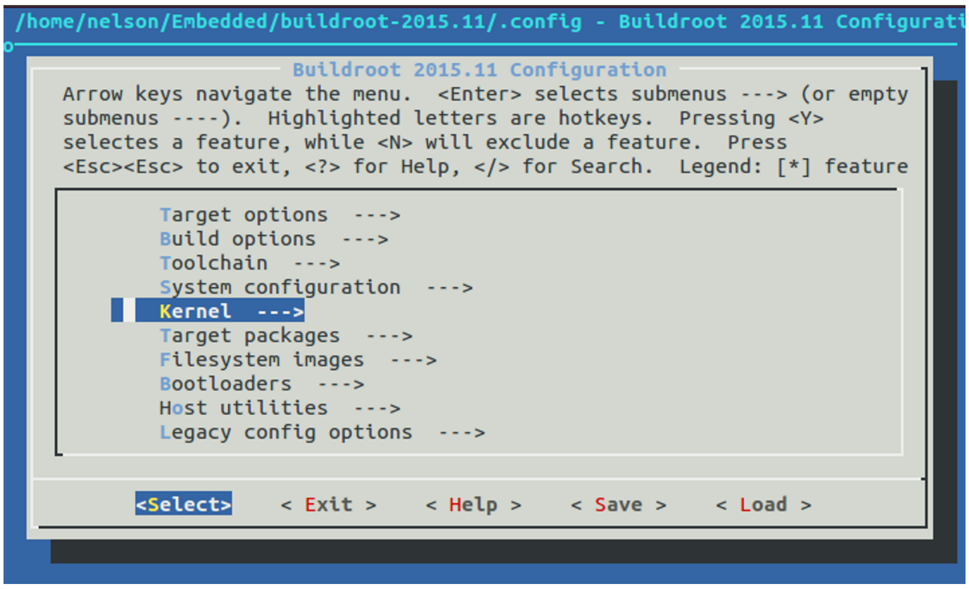
```
$ make menuconfig
```

Fill the Custom Kernel patch field with the the .gz patch file URL, matching it with the Kernel version.

Afterwards, if the kernel configuration menu is accessed again, real-time preemption will be available.

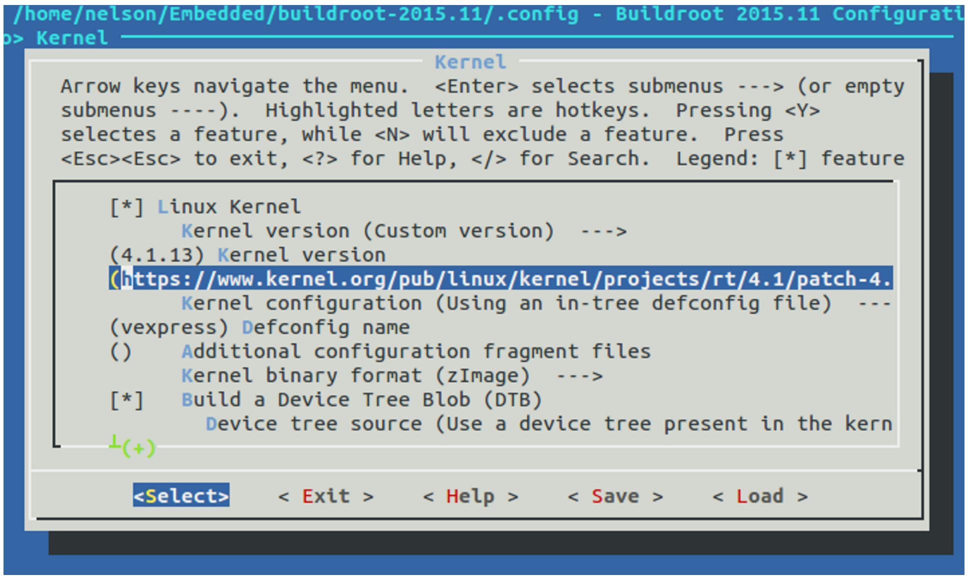
As an alternative, Buildroot supports Xenomai and RTAI extensions to the kernel. To enable them :

```
$ make menuconfig
```



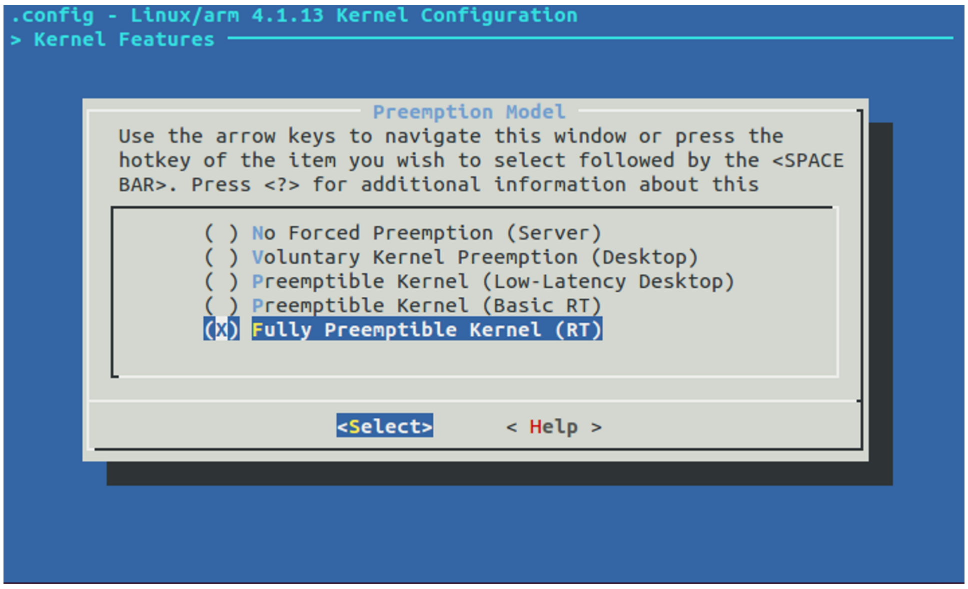
```
/home/nelson/Embedded/buildroot-2015.11/.config - Buildroot 2015.11 Configurati
o
                    Buildroot 2015.11 Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  selectes a feature, while <N> will exclude a feature.  Press
  <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature

          Target options  --->
          Build options  --->
          Toolchain  --->
          System configuration  --->
          Kernel  --->
          Target packages  --->
          Filesystem images  --->
          Bootloaders  --->
          Host utilities  --->
          Legacy config options  --->

       <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure A.4: Menuconfig Kernel



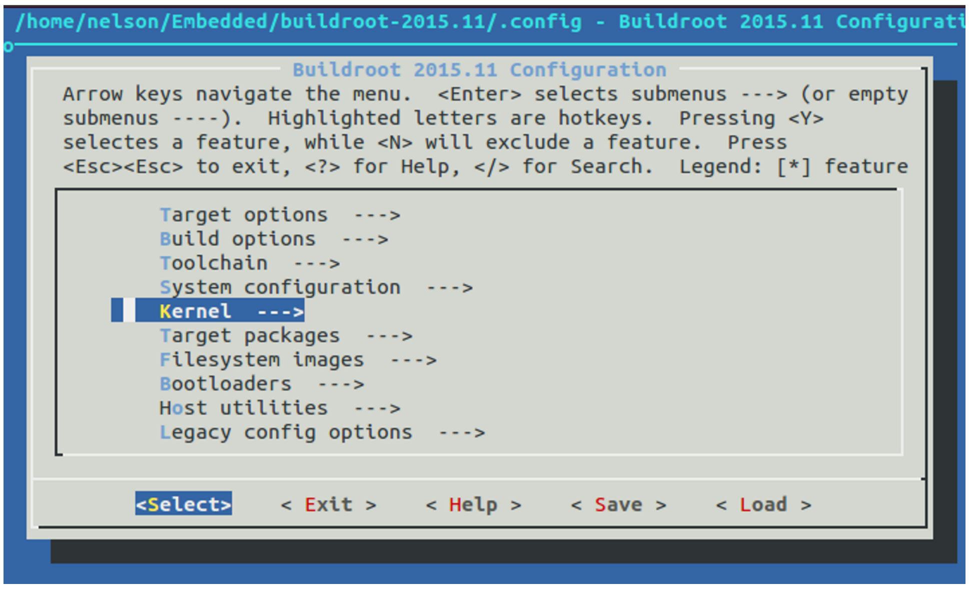
```
/home/nelson/Embedded/buildroot-2015.11/.config - Buildroot 2015.11 Configurati
> Kernel
                                 Kernel
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  selectes a feature, while <N> will exclude a feature.  Press
  <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature

      [*] Linux Kernel
            Kernel version (Custom version)  --->
      (4.1.13) Kernel version
      (https://www.kernel.org/pub/linux/kernel/projects/rt/4.1/patch-4.
            Kernel configuration (Using an in-tree defconfig file)  ---
      (vexpress) Defconfig name
      ()    Additional configuration fragment files
            Kernel binary format (zImage)  --->
      [*]   Build a Device Tree Blob (DTB)
              Device tree source (Use a device tree present in the kern
      ┴(+)

       <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure A.5: Menuconfig kernel patch



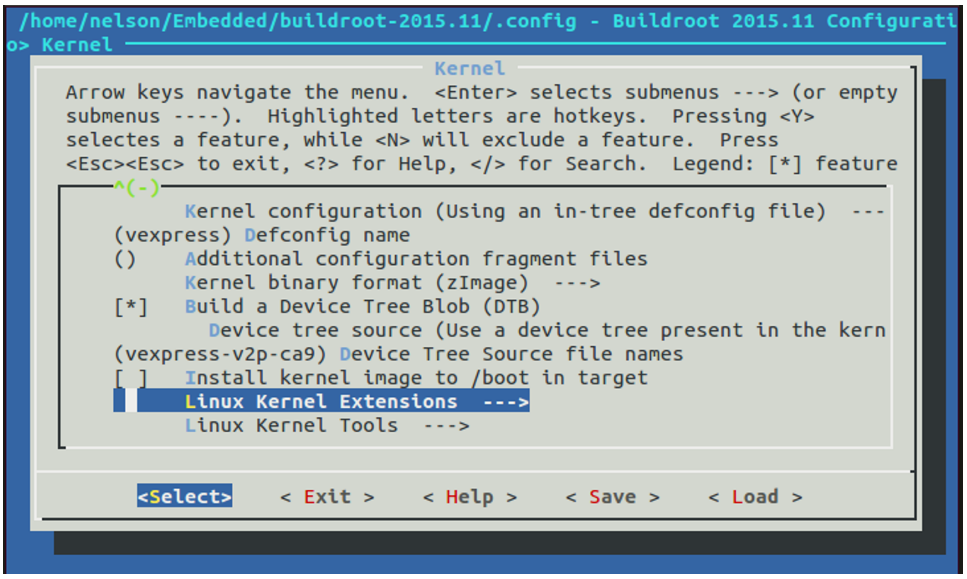
```
.config - Linux/arm 4.1.13 Kernel Configuration
> Kernel Features

                    Preemption Model
 Use the arrow keys to navigate this window or press the
 hotkey of the item you wish to select followed by the <SPACE
 BAR>. Press <?> for additional information about this

        ( ) No Forced Preemption (Server)
        ( ) Voluntary Kernel Preemption (Desktop)
        ( ) Preemptible Kernel (Low-Latency Desktop)
        ( ) Preemptible Kernel (Basic RT)
        (X) Fully Preemptible Kernel (RT)

                <Select>      < Help >
```

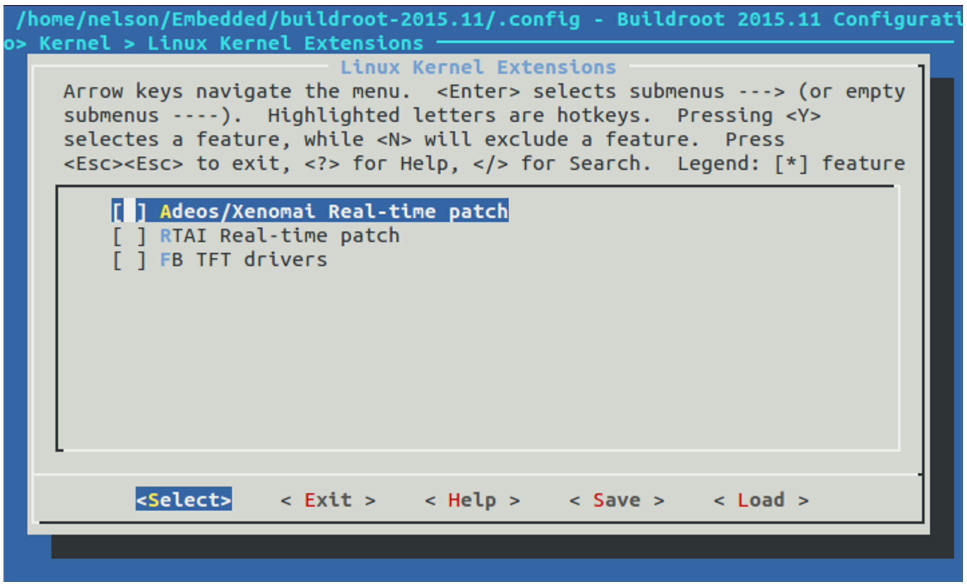
Figure A.6: Linux menuconfig real-time preemption



```
/home/nelson/Embedded/buildroot-2015.11/.config - Buildroot 2015.11 Configurati

                    Buildroot 2015.11 Configuration
 Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
 submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
 selectes a feature, while <N> will exclude a feature.  Press
 <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature

         Target options  --->
         Build options  --->
         Toolchain  --->
         System configuration  --->
         Kernel  --->
         Target packages  --->
         Filesystem images  --->
         Bootloaders  --->
         Host utilities  --->
         Legacy config options  --->

      <Select>    < Exit >    < Help >    < Save >    < Load >
```

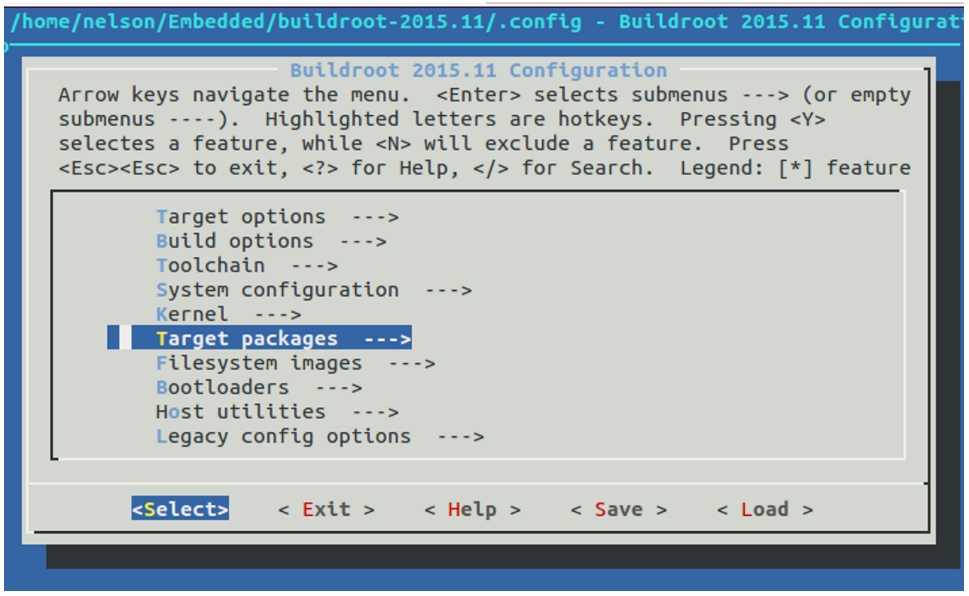
Figure A.7: Menuconfig kernel


```
/home/nelson/Embedded/buildroot-2015.11/.config - Buildroot 2015.11 Configurati
o> Kernel ──────────────────────────────────────────────────────────────────────
                              Kernel
   Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
   submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
   selectes a feature, while <N> will exclude a feature.  Press
   <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature
       ^(-)
           Kernel configuration (Using an in-tree defconfig file)  ---
       (vexpress) Defconfig name
       ()     Additional configuration fragment files
              Kernel binary format (zImage)  --->
       [*]    Build a Device Tree Blob (DTB)
                Device tree source (Use a device tree present in the kern
       (vexpress-v2p-ca9) Device Tree Source file names
       [ ]    Install kernel image to /boot in target
              Linux Kernel Extensions  --->
              Linux Kernel Tools  --->

         <Select>    < Exit >    < Help >    < Save >    < Load >
```
Figure A.8: Menuconfig kernel extensions


```
/home/nelson/Embedded/buildroot-2015.11/.config - Buildroot 2015.11 Configurati
o> Kernel > Linux Kernel Extensions ────────────────────────────────────────────
                       Linux Kernel Extensions
   Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
   submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
   selectes a feature, while <N> will exclude a feature.  Press
   <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature

       [ ] Adeos/Xenomai Real-time patch
       [ ] RTAI Real-time patch
       [ ] FB TFT drivers

         <Select>    < Exit >    < Help >    < Save >    < Load >
```
Figure A.9: Menuconfig Xenomai/RTAI

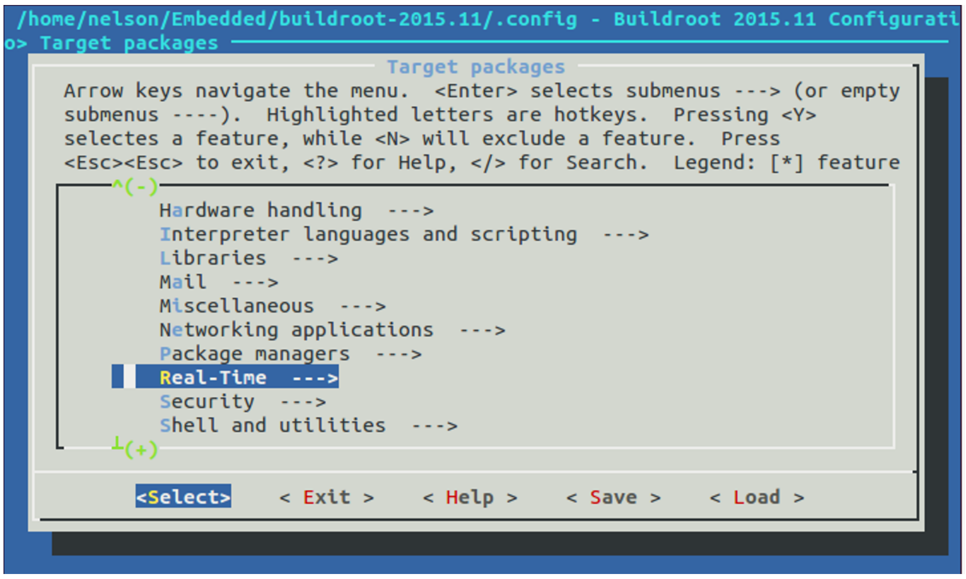
Specific to Xenomai:

```
$ make menuconfig
```



```
/home/nelson/Embedded/buildroot-2015.11/.config - Buildroot 2015.11 Configurat
                      Buildroot 2015.11 Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  selectes a feature, while <N> will exclude a feature.  Press
  <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature

          Target options  --->
          Build options  --->
          Toolchain  --->
          System configuration  --->
          Kernel  --->
          Target packages  --->
          Filesystem images  --->
          Bootloaders  --->
          Host utilities  --->
          Legacy config options  --->

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure A.10: Menuconfig target packages

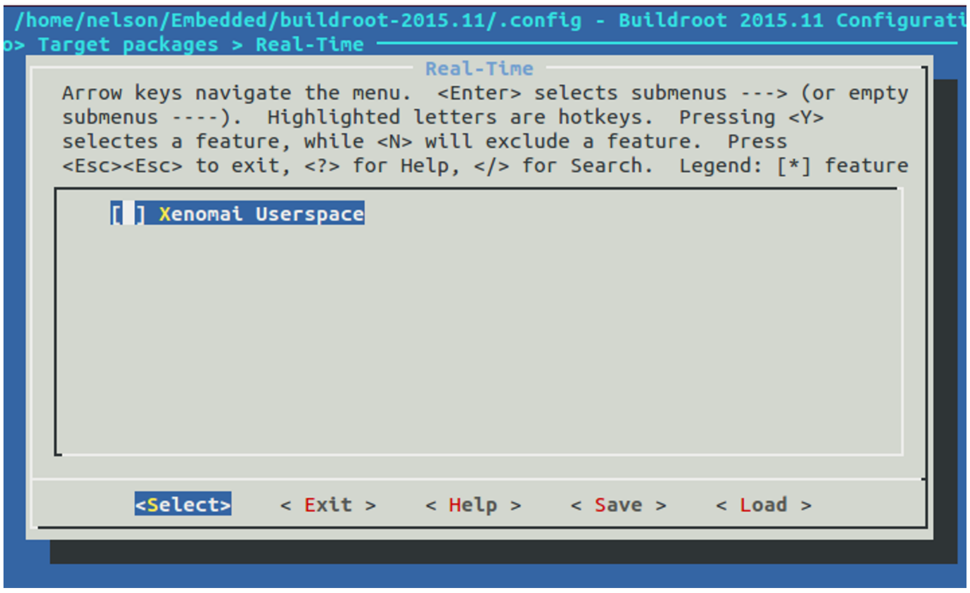


```
/home/nelson/Embedded/buildroot-2015.11/.config - Buildroot 2015.11 Configurati
o> Target packages
                      Target packages
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  selectes a feature, while <N> will exclude a feature.  Press
  <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature
      ^(-)
          Hardware handling  --->
          Interpreter languages and scripting  --->
          Libraries  --->
          Mail  --->
          Miscellaneous  --->
          Networking applications  --->
          Package managers  --->
          Real-Time  --->
          Security  --->
          Shell and utilities  --->
      ┴(+)

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure A.11: Menuconfig real-time

Finally, to compile the kernel simple run:

```
$ make
```



```
/home/nelson/Embedded/buildroot-2015.11/.config - Buildroot 2015.11 Configurati
o> Target packages > Real-Time
                              Real-Time
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  selectes a feature, while <N> will exclude a feature.  Press
  <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature

      [ ] Xenomai Userspace

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure A.12: Menuconfig Xenomai Userspace

A.1.3 Cross-compiling a Hello World Application with Buildroot

Buildroot will download and install cross-compilation tools in a corresponding output directory. The cross-compiling toolchain can be found in:

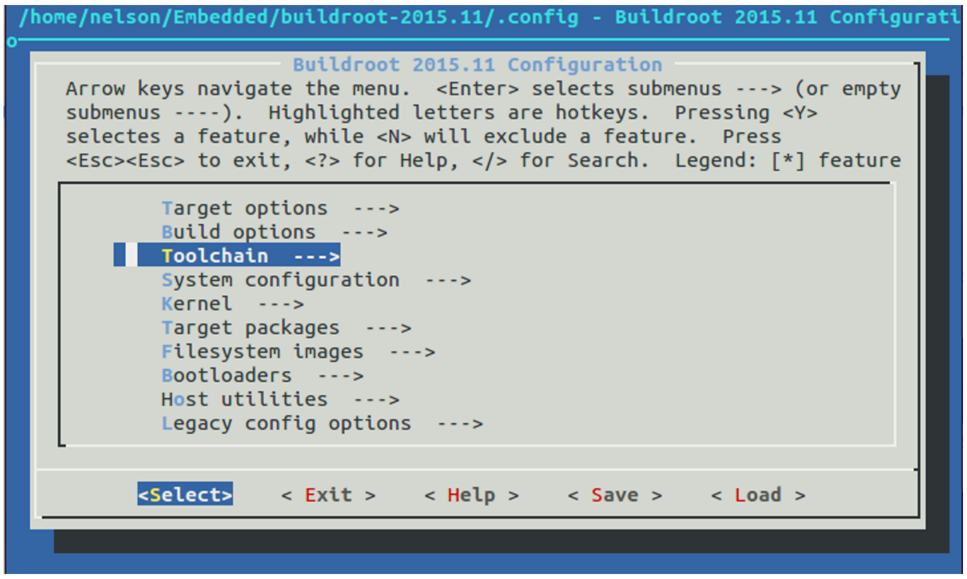
```
(buildroot output directory)/host/usr/bin
```

For example, for an ARM architecture with uClibc, an hello world C++ application can be compiled with

```
$ (buildroot output directory)/host/usr/bin/arm-
   buildroot-linux-uclibcgnueabi-g++ hello world.c -o
   hello world
```

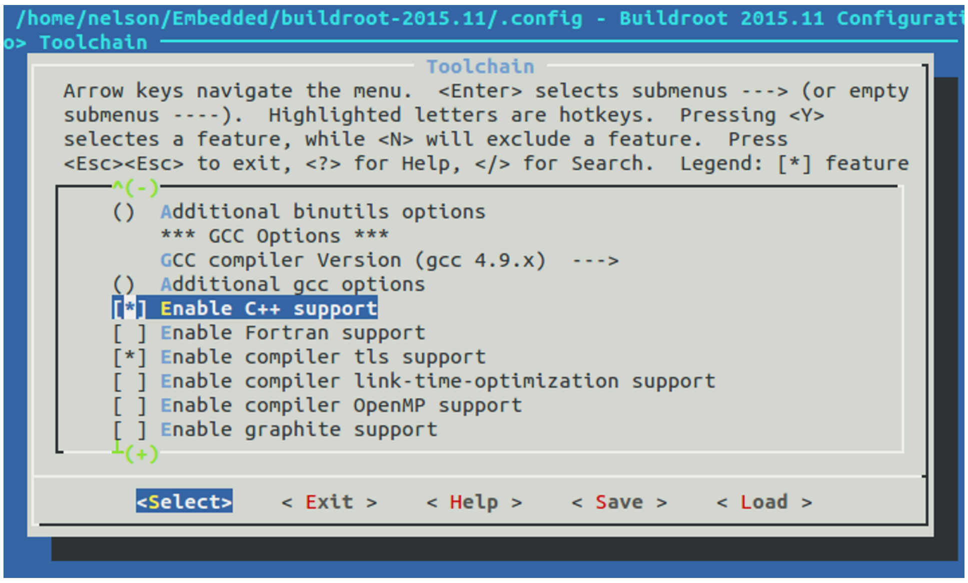
To enable C++ support, and thus install cross-compile g++:

```
$ make menuconfig
```



```
 /home/nelson/Embedded/buildroot-2015.11/.config - Buildroot 2015.11 Configurati
o──────────────────────────────────────────────────────────────────────────────
                       Buildroot 2015.11 Configuration
    Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
    submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
    selectes a feature, while <N> will exclude a feature.  Press
    <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature

            Target options  --->
            Build options  --->
            Toolchain  --->
            System configuration  --->
            Kernel  --->
            Target packages  --->
            Filesystem images  --->
            Bootloaders  --->
            Host utilities  --->
            Legacy config options  --->

         <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure A.13: Menuconfig toolchain



```
 /home/nelson/Embedded/buildroot-2015.11/.config - Buildroot 2015.11 Configurat
o> Toolchain ──────────────────────────────────────────────────────────────────
                                 Toolchain
    Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
    submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
    selectes a feature, while <N> will exclude a feature.  Press
    <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature
         ^(-)
         ()  Additional binutils options
             *** GCC Options ***
             GCC compiler Version (gcc 4.9.x)  --->
         ()  Additional gcc options
         [*] Enable C++ support
         [ ] Enable Fortran support
         [*] Enable compiler tls support
         [ ] Enable compiler link-time-optimization support
         [ ] Enable compiler OpenMP support
         [ ] Enable graphite support
         └(+)

         <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure A.14: Menuconfig enable C++ support

A.1.4 Booting a Cross-compiled Application with Linux in QEMU

Buildroot provides a set of defconfigs for QEMU machines. The following are supported as of Buildroot 2015.11

```
qemu_aarch64_virt_defconfig
qemu_arm_nuri_defconfig
qemu_arm_versatile_defconfig
qemu_arm_vexpress_defconfig
qemu_microblazebe_mmu_defconfig
qemu_microblazeel_mmu_defconfig
qemu_mips64_malta_defconfig
qemu_mips64el_malta_defconfig
qemu_mips_malta_defconfig
qemu_mipsel_malta_defconfig
qemu_ppc64_pseries_defconfig
qemu_ppc_g3beige_defconfig
qemu_ppc_mpc8544ds_defconfig
qemu_ppc_virtex_ml507_defconfig
qemu_sh4_r2d_defconfig
qemu_sh4eb_r2d_defconfig
qemu_sparc64_sun4u_defconfig
qemu_sparc_ss10_defconfig
qemu_x86_64_defconfig
qemu_x86_defconfig
```

For each QEMU board, there is a corresponding readme.txt containing a QEMU command that can be used to boot the generated image in:

```
Buildroot2015.11/board/qemu/(machine)
```

For the arm-vexpress machine:

```
$ cat Buildroot2015.11/board/qemu/arm-vexpress/readme.
   txt
Run the emulation with:

  qemu-system-arm -M vexpress-a9 -m 256 -kernel output/
     images/zImage -dtb output/images/vexpress-v2p-ca9.
     dtb -drive file=output/images/rootfs.ext2,if=sd -
     append "console=ttyAMA0,115200 root=/dev/mmcblk0" -
     serial stdio -net nic,model=lan9118 -net user

The login prompt will appear in the terminal that
   started Qemu. The
graphical window is the framebuffer.

Tested with QEMU 2.3.0
```

By modifying the command and redirecting the host 2222 port to port 22 of the target machine, it is possible to send the application from host to emulated target if the emulated target platform supports network. To redirect TCP/IP ports:

```
$ (QEMU command) -redir tcp:2222::22
```

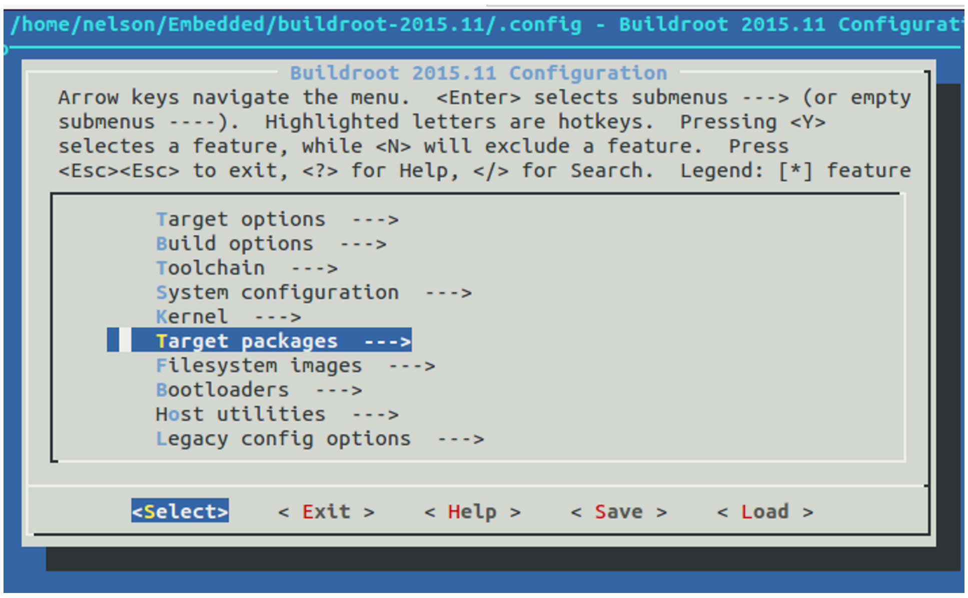
After emulation is running in QEMU, the hello world application can be sent using scp from the host:

```
$ scp -P 2222 'hello world' (username)@localhost:(target
    directory)
```

To add support to ssh and enable dhcp in the generated kernel image:

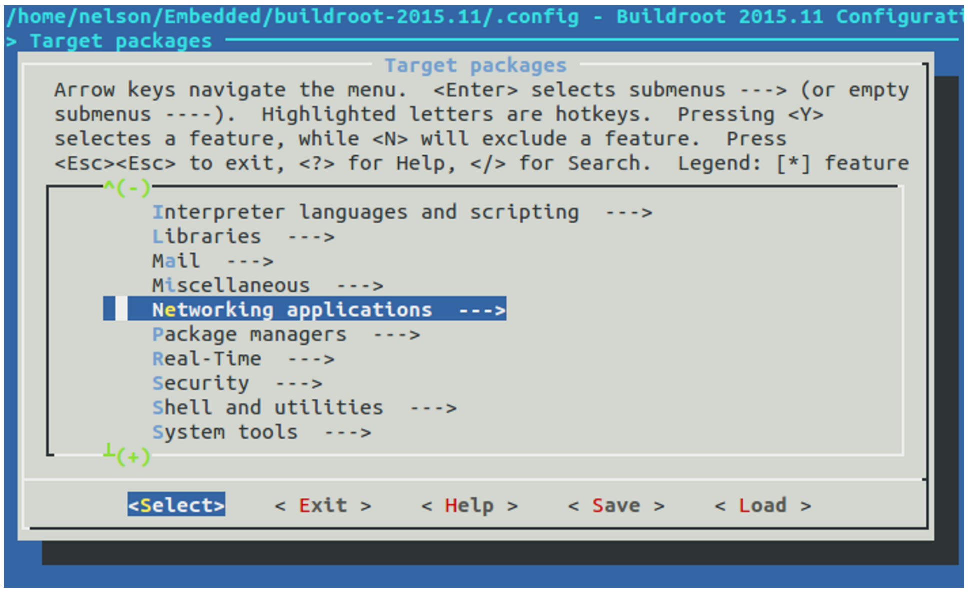
```
$ make menuconfig
```

Enable DHCP daemon, dropbear and openSSH:



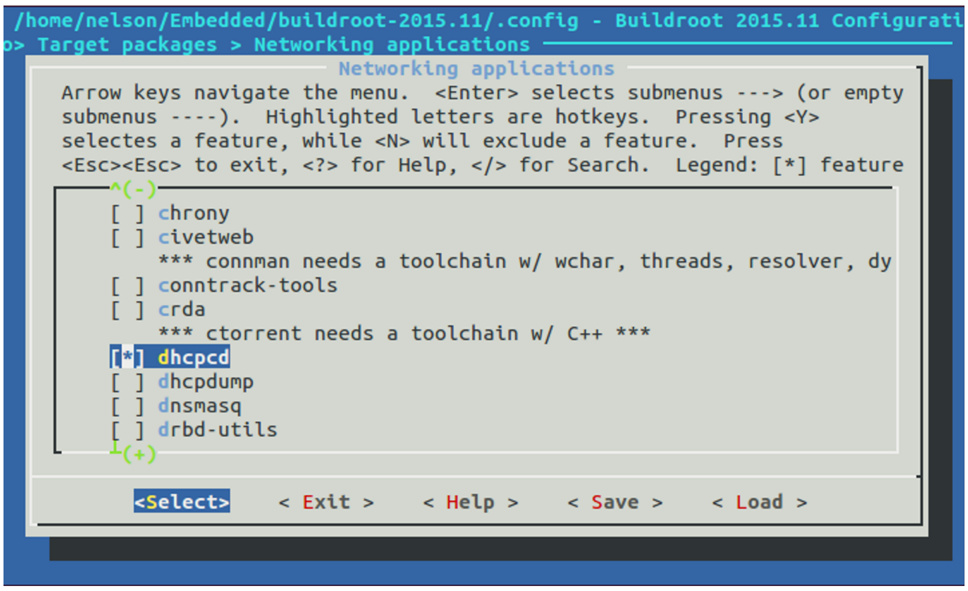
```
/home/nelson/Embedded/buildroot-2015.11/.config - Buildroot 2015.11 Configurat
                 Buildroot 2015.11 Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  selectes a feature, while <N> will exclude a feature.  Press
  <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature

          Target options  --->
          Build options  --->
          Toolchain  --->
          System configuration  --->
          Kernel  --->
          Target packages  --->
          Filesystem images  --->
          Bootloaders  --->
          Host utilities  --->
          Legacy config options  --->

      <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure A.15: Menuconfig target packages



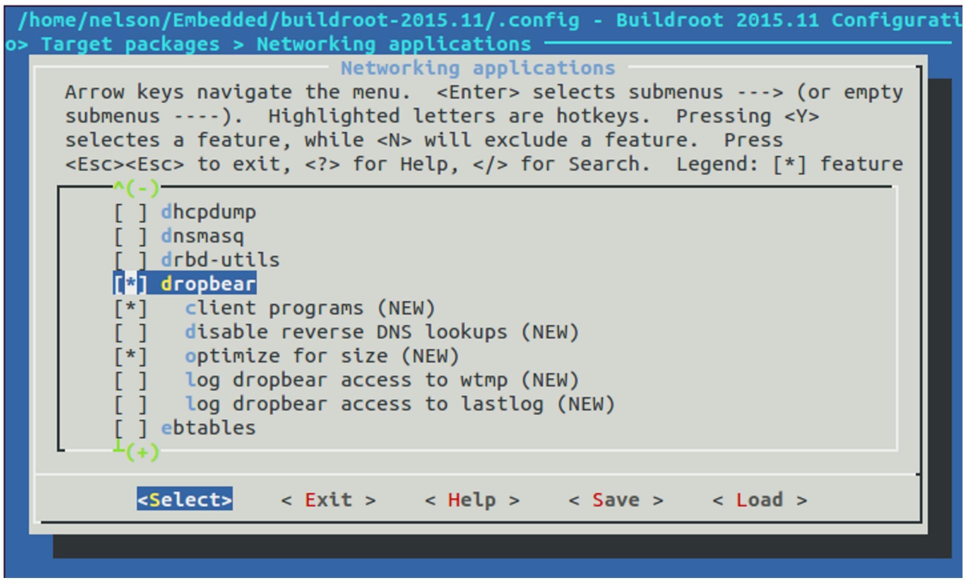
```
/home/nelson/Embedded/buildroot-2015.11/.config - Buildroot 2015.11 Configurat
> Target packages
                         Target packages
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  selectes a feature, while <N> will exclude a feature.  Press
  <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature
      ^(-)
          Interpreter languages and scripting  --->
          Libraries  --->
          Mail  --->
          Miscellaneous  --->
          Networking applications  --->
          Package managers  --->
          Real-Time  --->
          Security  --->
          Shell and utilities  --->
          System tools  --->
      ┴(+)

      <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure A.16: Menuconfig networking applications



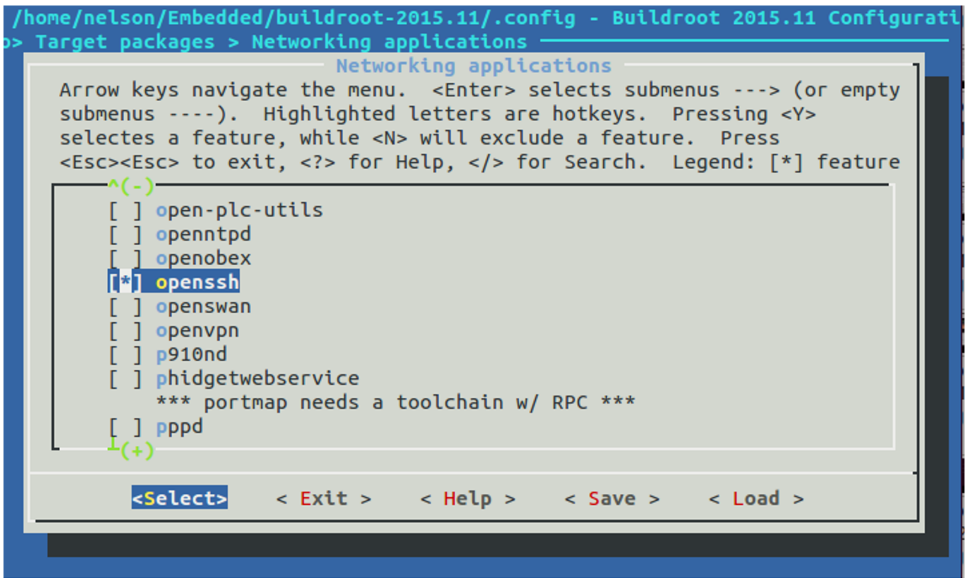
```
/home/nelson/Embedded/buildroot-2015.11/.config - Buildroot 2015.11 Configurati
o> Target packages > Networking applications ─────────────────────────────────
   ┌────────────────────── Networking applications ──────────────────────────┐
   │  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty │
   │  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>        │
   │  selectes a feature, while <N> will exclude a feature.  Press           │
   │  <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature │
   │ ┌────^(-)──────────────────────────────────────────────────────────────┐│
   │ │      [ ] chrony                                                      ││
   │ │      [ ] civetweb                                                    ││
   │ │          *** connman needs a toolchain w/ wchar, threads, resolver, dy││
   │ │      [ ] conntrack-tools                                             ││
   │ │      [ ] crda                                                        ││
   │ │          *** ctorrent needs a toolchain w/ C++ ***                   ││
   │ │      [*] dhcpcd                                                      ││
   │ │      [ ] dhcpdump                                                    ││
   │ │      [ ] dnsmasq                                                     ││
   │ │      [ ] drbd-utils                                                  ││
   │ └────└(+)──────────────────────────────────────────────────────────────┘│
   ├─────────────────────────────────────────────────────────────────────────┤
   │        <Select>    < Exit >    < Help >    < Save >    < Load >         │
   └─────────────────────────────────────────────────────────────────────────┘
```

Figure A.17: Menuconfig DHCP daemon



```
/home/nelson/Embedded/buildroot-2015.11/.config - Buildroot 2015.11 Configurati
o> Target packages > Networking applications ─────────────────────────────────
   ┌────────────────────── Networking applications ──────────────────────────┐
   │  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty │
   │  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>        │
   │  selectes a feature, while <N> will exclude a feature.  Press           │
   │  <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature │
   │ ┌────^(-)──────────────────────────────────────────────────────────────┐│
   │ │      [ ] dhcpdump                                                    ││
   │ │      [ ] dnsmasq                                                     ││
   │ │      [ ] drbd-utils                                                  ││
   │ │      [*] dropbear                                                    ││
   │ │      [*]   client programs (NEW)                                     ││
   │ │      [ ]   disable reverse DNS lookups (NEW)                         ││
   │ │      [*]   optimize for size (NEW)                                   ││
   │ │      [ ]   log dropbear access to wtmp (NEW)                         ││
   │ │      [ ]   log dropbear access to lastlog (NEW)                      ││
   │ │      [ ] ebtables                                                    ││
   │ └────└(+)──────────────────────────────────────────────────────────────┘│
   ├─────────────────────────────────────────────────────────────────────────┤
   │        <Select>    < Exit >    < Help >    < Save >    < Load >         │
   └─────────────────────────────────────────────────────────────────────────┘
```

Figure A.18: Menuconfig dropbear

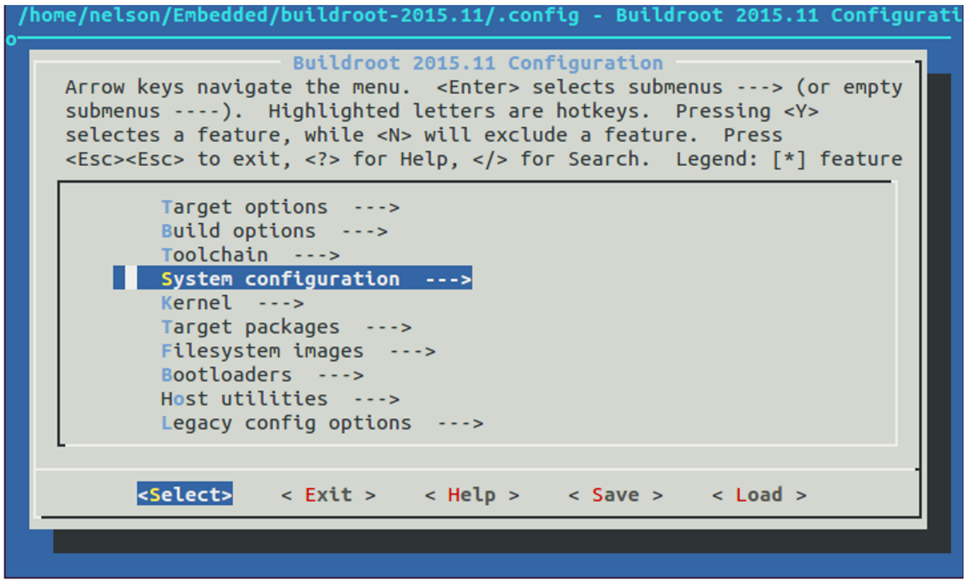


```
/home/nelson/Embedded/buildroot-2015.11/.config - Buildroot 2015.11 Configurati
> Target packages > Networking applications
                         Networking applications
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  selectes a feature, while <N> will exclude a feature.  Press
  <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature
       ^(-)
       [ ] open-plc-utils
       [ ] openntpd
       [ ] openobex
       [*] openssh
       [ ] openswan
       [ ] openvpn
       [ ] p910nd
       [ ] phidgetwebservice
           *** portmap needs a toolchain w/ RPC ***
       [ ] pppd
        (+)

         <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure A.19: Menuconfig openSSH

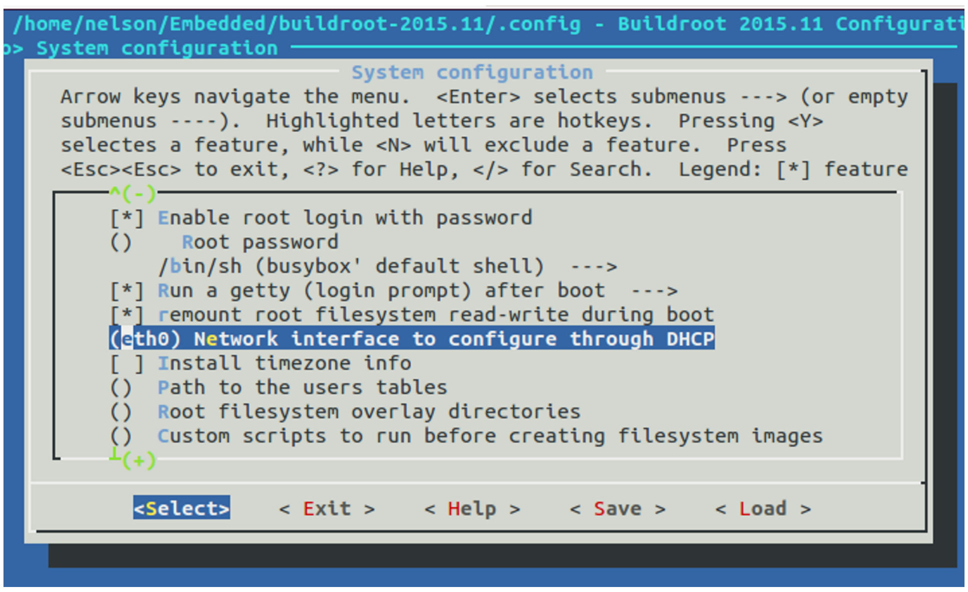
Make sure eth0 is assigned to be configured through DHCP:

```
$ make menuconfig
```



```
/home/nelson/Embedded/buildroot-2015.11/.config - Buildroot 2015.11 Configurati

                       Buildroot 2015.11 Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  selectes a feature, while <N> will exclude a feature.  Press
  <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature

           Target options  --->
           Build options  --->
           Toolchain  --->
           System configuration  --->
           Kernel  --->
           Target packages  --->
           Filesystem images  --->
           Bootloaders  --->
           Host utilities  --->
           Legacy config options  --->

         <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure A.20: Menuconfig system configuration



```
/home/nelson/Embedded/buildroot-2015.11/.config - Buildroot 2015.11 Configurati
> System configuration ─────────────────────────────────────────────────────
                          System configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  selectes a feature, while <N> will exclude a feature.  Press
  <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature
      ^(-)
      [*] Enable root login with password
      ()    Root password
          /bin/sh (busybox' default shell)  --->
      [*] Run a getty (login prompt) after boot  --->
      [*] remount root filesystem read-write during boot
      (eth0) Network interface to configure through DHCP
      [ ] Install timezone info
      ()  Path to the users tables
      ()  Root filesystem overlay directories
      ()  Custom scripts to run before creating filesystem images
      └(+)

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure A.21: Menuconfig DHCP for eth0

Alternatively, if there is no network support on the machine, the cross-compiled application can be copied to a directory to be included in the generated root filesystem before running make:

```
$ cp -f  'hello world' (Buildroot output)/target/(root
   filesystem directory)
```

A.2 Device Tree Versatile Express A-9 .dts File Example

```
/*
 * ARM Ltd. Versatile Express
 *
 * CoreTile Express A9x4
 * Cortex-A9 MPCore (V2P-CA9)
 *
 * HBI-0191B
 */
```

```
/dts-v1/;

/ {
        model = "V2P-CA9";
        arm,hbi = <0x191>;
        arm,vexpress,site = <0xf>;
        compatible = "arm,vexpress,v2p-ca9", "arm,
           vexpress";
        interrupt-parent = <&gic>;
        #address-cells = <1>;
        #size-cells = <1>;

        chosen { };

        aliases {
                serial0 = &v2m_serial0;
                serial1 = &v2m_serial1;
                serial2 = &v2m_serial2;
                serial3 = &v2m_serial3;
                i2c0 = &v2m_i2c_dvi;
                i2c1 = &v2m_i2c_pcie;
        };

        cpus {
                #address-cells = <1>;
                #size-cells = <0>;

                cpu@0 {
                        device_type = "cpu";
                        compatible = "arm,cortex-a9";
                        reg = <0>;
                        next-level-cache = <&L2>;
                };

                cpu@1 {
                        device_type = "cpu";
                        compatible = "arm,cortex-a9";
                        reg = <1>;
                        next-level-cache = <&L2>;
                };

                cpu@2 {
                        device_type = "cpu";
                        compatible = "arm,cortex-a9";
                        reg = <2>;
                        next-level-cache = <&L2>;
                };

                cpu@3 {
                        device_type = "cpu";
                        compatible = "arm,cortex-a9";
                        reg = <3>;
                        next-level-cache = <&L2>;
                };
        };

        memory@60000000 {
                device_type = "memory";
                reg = <0x60000000 0x40000000>;
        };

        clcd@10020000 {
                compatible = "arm,pl111", "arm,primecell
                   ";
                reg = <0x10020000 0x1000>;
                interrupts = <0 44 4>;
                clocks = <&oscclk1>, <&oscclk2>;
                clock-names = "clcdclk", "apb_pclk";
        };

        memory-controller@100e0000 {
                compatible = "arm,pl341", "arm,primecell
                   ";
                reg = <0x100e0000 0x1000>;
                clocks = <&oscclk2>;
                clock-names = "apb_pclk";
        };

        memory-controller@100e1000 {
                compatible = "arm,pl354", "arm,primecell
                   ";
                reg = <0x100e1000 0x1000>;
                interrupts = <0 45 4>,
                             <0 46 4>;
                clocks = <&oscclk2>;
                clock-names = "apb_pclk";
        };

        timer@100e4000 {
                compatible = "arm,sp804", "arm,primecell
                   ";
                reg = <0x100e4000 0x1000>;
                interrupts = <0 48 4>,
                             <0 49 4>;
                clocks = <&oscclk2>, <&oscclk2>;
                clock-names = "timclk", "apb_pclk";
                status = "disabled";
        };

        watchdog@100e5000 {
                compatible = "arm,sp805", "arm,primecell
                   ";
                reg = <0x100e5000 0x1000>;
                interrupts = <0 51 4>;
                clocks = <&oscclk2>, <&oscclk2>;
                clock-names = "wdogclk", "apb_pclk";
        };

        scu@1e000000 {
                compatible = "arm,cortex-a9-scu";
                reg = <0x1e000000 0x58>;
```

```
};

timer@1e000600 {
        compatible = "arm,cortex-a9-twd-timer";
        reg = <0x1e000600 0x20>;
        interrupts = <1 13 0xf04>;
};

watchdog@1e000620 {
        compatible = "arm,cortex-a9-twd-wdt";
        reg = <0x1e000620 0x20>;
        interrupts = <1 14 0xf04>;
};

gic: interrupt-controller@1e001000 {
        compatible = "arm,cortex-a9-gic";
        #interrupt-cells = <3>;
        #address-cells = <0>;
        interrupt-controller;
        reg = <0x1e001000 0x1000>,
              <0x1e000100 0x100>;
};

L2: cache-controller@1e00a000 {
        compatible = "arm,pl310-cache";
        reg = <0x1e00a000 0x1000>;
        interrupts = <0 43 4>;
        cache-level = <2>;
        arm,data-latency = <1 1 1>;
        arm,tag-latency = <1 1 1>;
};

pmu {
        compatible = "arm,cortex-a9-pmu";
        interrupts = <0 60 4>,
                     <0 61 4>,
                     <0 62 4>,
                     <0 63 4>;
};

dcc {
        compatible = "arm,vexpress,config-bus";
        arm,vexpress,config-bridge = <&
           v2m_sysreg>;

        osc@0 {
                     /* ACLK clock to the AXI master
                        port on the test chip */
                     compatible = "arm,vexpress-osc";
                     arm,vexpress-sysreg,func = <1
                        0>;
                     freq-range = <30000000
                        50000000>;
                     #clock-cells = <0>;
                     clock-output-names = "extsaxiclk
                        ";
        };

        oscclk1: osc@1 {
                     /* Reference clock for the CLCD
                        */
                     compatible = "arm,vexpress-osc";
                     arm,vexpress-sysreg,func = <1
                        1>;
                     freq-range = <10000000
                        80000000>;
                     #clock-cells = <0>;
                     clock-output-names = "clcdclk";
        };

        smbclk: oscclk2: osc@2 {
                     /* Reference clock for the test
                        chip internal PLLs */
                     compatible = "arm,vexpress-osc";
                     arm,vexpress-sysreg,func = <1
                        2>;
                     freq-range = <33000000
                        100000000>;
                     #clock-cells = <0>;
                     clock-output-names = "tcrefclk";
        };

        volt@0 {
                     /* Test Chip internal logic
                        voltage */
                     compatible = "arm,vexpress-volt"
                        ;
                     arm,vexpress-sysreg,func = <2
                        0>;
                     regulator-name = "VD10";
                     regulator-always-on;
                     label = "VD10";
        };

        volt@1 {
                     /* PL310, L2 cache, RAM cell
                        supply (not PL310 logic) */
                     compatible = "arm,vexpress-volt"
                        ;
                     arm,vexpress-sysreg,func = <2
                        1>;
                     regulator-name = "VD10_S2";
                     regulator-always-on;
                     label = "VD10_S2";
        };

        volt@2 {
                     /* Cortex-A9 system supply,
                        Cores, MPEs, SCU and PL310
                        logic */
                compatible = "arm,vexpress-volt"
                   ;
                arm,vexpress-sysreg,func = <2
                   2>;
                regulator-name = "VD10_S3";
                regulator-always-on;
                label = "VD10_S3";
        };

        volt@3 {
                /* DDR2 SDRAM and Test Chip DDR2
                     I/O supply */
                compatible = "arm,vexpress-volt"
                   ;
                arm,vexpress-sysreg,func = <2
                   3>;
                regulator-name = "VCC1V8";
                regulator-always-on;
                label = "VCC1V8";
        };

        volt@4 {
                /* DDR2 SDRAM VTT termination
                   voltage */
                compatible = "arm,vexpress-volt"
                   ;
                arm,vexpress-sysreg,func = <2
                   4>;
                regulator-name = "DDR2VTT";
                regulator-always-on;
                label = "DDR2VTT";
        };

        volt@5 {
                /* Local board supply for
                   miscellaneous logic external
                   to the Test Chip */
                arm,vexpress-sysreg,func = <2
                   5>;
                compatible = "arm,vexpress-volt"
                   ;
                regulator-name = "VCC3V3";
                regulator-always-on;
                label = "VCC3V3";
        };

        amp@0 {
                /* PL310, L2 cache, RAM cell
                   supply (not PL310 logic) */
                compatible = "arm,vexpress-amp";
                arm,vexpress-sysreg,func = <3
                   0>;
                label = "VD10_S2";
        };

        amp@1 {
                /* Cortex-A9 system supply,
                   Cores, MPEs, SCU and PL310
                   logic */
                compatible = "arm,vexpress-amp";
                arm,vexpress-sysreg,func = <3
                   1>;
                label = "VD10_S3";
        };

        power@0 {
                /* PL310, L2 cache, RAM cell
                   supply (not PL310 logic) */
                compatible = "arm,vexpress-power
                   ";
                arm,vexpress-sysreg,func = <12
                   0>;
                label = "PVD10_S2";
        };
```

```
        power@1 {
                /* Cortex-A9 system supply,
                   Cores, MPEs, SCU and PL310
                   logic */
                compatible = "arm,vexpress-power
                   ";
                arm,vexpress-sysreg,func = <12
                   1>;
                label = "PVD10_S3";
        };
};

smb {
        compatible = "simple-bus";

        #address-cells = <2>;
        #size-cells = <1>;
        ranges = <0 0 0x40000000 0x04000000>,
                 <1 0 0x44000000 0x04000000>,
                 <2 0 0x48000000 0x04000000>,
                 <3 0 0x4c000000 0x04000000>,
                 <7 0 0x10000000 0x00020000>;

        #interrupt-cells = <1>;
        interrupt-map-mask = <0 0 63>;
        interrupt-map = <0 0  0 &gic 0  0 4>,
                        <0 0  1 &gic 0  1 4>,
                        <0 0  2 &gic 0  2 4>,
                        <0 0  3 &gic 0  3 4>,
                        <0 0  4 &gic 0  4 4>,
                        <0 0  5 &gic 0  5 4>,
                        <0 0  6 &gic 0  6 4>,
                        <0 0  7 &gic 0  7 4>,
                        <0 0  8 &gic 0  8 4>,
                        <0 0  9 &gic 0  9 4>,
                        <0 0 10 &gic 0 10 4>,
        };
};

                                           <0 0 11 &gic 0 11 4>,
                                           <0 0 12 &gic 0 12 4>,
                                           <0 0 13 &gic 0 13 4>,
                                           <0 0 14 &gic 0 14 4>,
                                           <0 0 15 &gic 0 15 4>,
                                           <0 0 16 &gic 0 16 4>,
                                           <0 0 17 &gic 0 17 4>,
                                           <0 0 18 &gic 0 18 4>,
                                           <0 0 19 &gic 0 19 4>,
                                           <0 0 20 &gic 0 20 4>,
                                           <0 0 21 &gic 0 21 4>,
                                           <0 0 22 &gic 0 22 4>,
                                           <0 0 23 &gic 0 23 4>,
                                           <0 0 24 &gic 0 24 4>,
                                           <0 0 25 &gic 0 25 4>,
                                           <0 0 26 &gic 0 26 4>,
                                           <0 0 27 &gic 0 27 4>,
                                           <0 0 28 &gic 0 28 4>,
                                           <0 0 29 &gic 0 29 4>,
                                           <0 0 30 &gic 0 30 4>,
                                           <0 0 31 &gic 0 31 4>,
                                           <0 0 32 &gic 0 32 4>,
                                           <0 0 33 &gic 0 33 4>,
                                           <0 0 34 &gic 0 34 4>,
                                           <0 0 35 &gic 0 35 4>,
                                           <0 0 36 &gic 0 36 4>,
                                           <0 0 37 &gic 0 37 4>,
                                           <0 0 38 &gic 0 38 4>,
                                           <0 0 39 &gic 0 39 4>,
                                           <0 0 40 &gic 0 40 4>,
                                           <0 0 41 &gic 0 41 4>,
                                           <0 0 42 &gic 0 42 4>;

                /include/ "vexpress-v2m.dtsi"
        };
};
```

Appendix B

QEMU Support Material

B.1 QEMU Machines

B.1.1 Aarch64 Architecture

```
$ qemu-system-aarch64 -machine ?
Supported machines are:
akita                 Akita PDA (PXA270)
borzoi                Borzoi PDA (PXA270)
canon-a1100           Canon PowerShot A1100 IS
cheetah               Palm Tungsten|E aka. Cheetah PDA (
   OMAP310)
collie                Collie PDA (SA-1110)
connex                Gumstix Connex (PXA255)
cubieboard            cubietech cubieboard
highbank              Calxeda Highbank (ECX-1000)
integratorcp          ARM Integrator/CP (ARM926EJ-S)
kzm                   ARM KZM Emulation Baseboard (
   ARM1136)
lm3s6965evb           Stellaris LM3S6965EVB
lm3s811evb            Stellaris LM3S811EVB
mainstone             Mainstone II (PXA27x)
midway                Calxeda Midway (ECX-2000)
musicpal              Marvell 88w8618 / MusicPal (
   ARM926EJ-S)
```

```
n800                 Nokia N800 tablet aka. RX-34 (
   OMAP2420)
n810                 Nokia N810 tablet aka. RX-44 (
   OMAP2420)
none                 empty machine
nuri                 Samsung NURI board (Exynos4210)
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
smdkc210             Samsung SMDKC210 board (Exynos4210)
spitz                Spitz PDA (PXA270)
sx1                  Siemens SX1 (OMAP310) V2
sx1-v1               Siemens SX1 (OMAP310) V1
terrier              Terrier PDA (PXA270)
tosa                 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                 ARM Virtual Machine
xilinx-zynq-a9       Xilinx Zynq Platform Baseboard for
   Cortex-A9
z2                   Zipit Z2 (PXA27x)
```

B.1.2 Alpha Architecture

```
$ qemu-system-alpha -machine ?
Supported machines are:
clipper              Alpha DP264/CLIPPER (default)
none                 empty machine
```

B.1.3 ARM Architecture

```
  $ qemu-system-arm -machine ?
Supported machines are:
akita                Akita PDA (PXA270)
borzoi               Borzoi PDA (PXA270)
canon-a1100          Canon PowerShot A1100 IS
cheetah              Palm Tungsten|E aka. Cheetah PDA (
   OMAP310)
collie               Collie PDA (SA-1110)
connex               Gumstix Connex (PXA255)
cubieboard           cubietech cubieboard
highbank             Calxeda Highbank (ECX-1000)
integratorcp         ARM Integrator/CP (ARM926EJ-S)
kzm                  ARM KZM Emulation Baseboard (
   ARM1136)
lm3s6965evb          Stellaris LM3S6965EVB
lm3s811evb           Stellaris LM3S811EVB
mainstone            Mainstone II (PXA27x)
midway               Calxeda Midway (ECX-2000)
musicpal             Marvell 88w8618 / MusicPal (
   ARM926EJ-S)
n800                 Nokia N800 tablet aka. RX-34 (
   OMAP2420)
n810                 Nokia N810 tablet aka. RX-44 (
   OMAP2420)
none                 empty machine
nuri                 Samsung NURI board (Exynos4210)
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
smdkc210             Samsung SMDKC210 board (Exynos4210)
```

```
spitz                   Spitz PDA (PXA270)
sx1                     Siemens SX1 (OMAP310) V2
sx1-v1                  Siemens SX1 (OMAP310) V1
terrier                 Terrier PDA (PXA270)
tosa                    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                    ARM Virtual Machine
xilinx-zynq-a9          Xilinx Zynq Platform Baseboard for
   Cortex-A9
z2                      Zipit Z2 (PXA27x)
```

B.1.4 CRIS Architecture

```
$ qemu-system-cris -machine ?
Supported machines are:
axis-dev88              AXIS devboard 88 (default)
none                    empty machine
```

B.1.5 i386 Architecture

```
$ qemu-system-i386 -machine ?
Supported machines are:
pc                      Standard PC (i440FX + PIIX, 1996) (
   alias of pc-i440fx-2.2)
pc-i440fx-2.2           Standard PC (i440FX + PIIX, 1996) (
   default)
pc-i440fx-2.1           Standard PC (i440FX + PIIX, 1996)
pc-i440fx-2.0           Standard PC (i440FX + PIIX, 1996)
pc-i440fx-1.7           Standard PC (i440FX + PIIX, 1996)
pc-i440fx-1.6           Standard PC (i440FX + PIIX, 1996)
pc-i440fx-1.5           Standard PC (i440FX + PIIX, 1996)
pc-i440fx-1.4           Standard PC (i440FX + PIIX, 1996)
pc-1.3                  Standard PC (i440FX + PIIX, 1996)
```

```
pc-1.2               Standard PC (i440FX + PIIX, 1996)
pc-1.1               Standard PC (i440FX + PIIX, 1996)
pc-1.0               Standard PC (i440FX + PIIX, 1996)
pc-0.15              Standard PC (i440FX + PIIX, 1996)
pc-0.14              Standard PC (i440FX + PIIX, 1996)
pc-0.13              Standard PC (i440FX + PIIX, 1996)
pc-0.12              Standard PC (i440FX + PIIX, 1996)
pc-0.11              Standard PC (i440FX + PIIX, 1996)
pc-0.10              Standard PC (i440FX + PIIX, 1996)
q35                  Standard PC (Q35 + ICH9, 2009) (
   alias of pc-q35-2.2)
pc-q35-2.2           Standard PC (Q35 + ICH9, 2009)
pc-q35-2.1           Standard PC (Q35 + ICH9, 2009)
pc-q35-2.0           Standard PC (Q35 + ICH9, 2009)
pc-q35-1.7           Standard PC (Q35 + ICH9, 2009)
pc-q35-1.6           Standard PC (Q35 + ICH9, 2009)
pc-q35-1.5           Standard PC (Q35 + ICH9, 2009)
pc-q35-1.4           Standard PC (Q35 + ICH9, 2009)
isapc                ISA-only PC
none                 empty machine
```

B.1.6 LatticeMicro32 Architecture

```
$ qemu-system-lm32 -machine ?
Supported machines are:
lm32-evr             LatticeMico32 EVR32 eval system (
   default)
lm32-uclinux         lm32 platform for uClinux and u-
   boot by Theobroma Systems
milkymist            Milkymist One
none                 empty machine
```

B.1.7 Motorolla 68000 Architecture

```
$ qemu-system-m68k -machine ?
Supported machines are:
an5206               Arnewsh 5206
dummy                Dummy board
```

```
mcf5208evb           MCF5206EVB (default)
none                 empty machine
```

B.1.8 Microblaze Softcore Architecture

```
$ qemu-system-microblaze -machine ?
Supported machines are:
none                 empty machine
petalogix-ml605      PetaLogix linux refdesign for
   xilinx ml605 little endian
petalogix-s3adsp1800 PetaLogix linux refdesign for
   xilinx Spartan 3ADSP1800 (default)
```

B.1.9 Microblaze Softcore Little Endian Architecture

```
$ qemu-system-microblazeel -machine ?
Supported machines are:
none                 empty machine
petalogix-ml605      PetaLogix linux refdesign for
   xilinx ml605 little endian
petalogix-s3adsp1800 PetaLogix linux refdesign for
   xilinx Spartan 3ADSP1800 (default)
```

B.1.10 MIPS Architecture

```
$ qemu-system-mips -machine ?
Supported machines are:
magnum               MIPS Magnum
malta                MIPS Malta Core LV (default)
mips                 mips r4k platform
mipssim              MIPS MIPSsim platform
none                 empty machine
pica61               Acer Pica 61
```

B.1.11 MIPS Little Endian Architecture

```
$ qemu-system-mipsel -machine ?
Supported machines are:
```

```
magnum                 MIPS Magnum
malta                  MIPS Malta Core LV (default)
mips                   mips r4k platform
mipssim                MIPS MIPSsim platform
none                   empty machine
pica61                 Acer Pica 61
```

B.1.12 MIPS64 Architecture

```
$ qemu-system-mips64 -machine ?
Supported machines are:
magnum                 MIPS Magnum
malta                  MIPS Malta Core LV (default)
mips                   mips r4k platform
mipssim                MIPS MIPSsim platform
none                   empty machine
pica61                 Acer Pica 61
```

B.1.13 MIPS64 Little Endian Architecture

```
$ qemu-system-mips64el -machine ?
Supported machines are:
fulong2e               Fulong 2e mini pc
magnum                 MIPS Magnum
malta                  MIPS Malta Core LV (default)
mips                   mips r4k platform
mipssim                MIPS MIPSsim platform
none                   empty machine
pica61                 Acer Pica 61
```

B.1.14 Moxie Softcore Architecture

```
$ qemu-system-moxie -machine ?
Supported machines are:
moxiesim               Moxie simulator platform (default)
none                   empty machine
```

B.1.15 OpenRisc32 Architecture

```
$ qemu-system-or32 -machine ?
Supported machines are:
none                 empty machine
or32-sim             or32 simulation (default)
```

B.1.16 PowerPC Architecture

```
$ qemu-system-ppc -machine ?
Supported machines are:
bamboo               bamboo
g3beige              Heathrow based PowerMAC (default)
mac99                Mac99 based PowerMAC
mpc8544ds            mpc8544ds
none                 empty machine
ppce500              generic paravirt e500 platform
prep                 PowerPC PREP platform
ref405ep             ref405ep
taihu                taihu
virtex-ml507         Xilinx Virtex ML507 reference
   design
```

B.1.17 PowerPC64 Architecture

```
$ qemu-system-ppc64 -machine ?
Supported machines are:
bamboo               bamboo
g3beige              Heathrow based PowerMAC
mac99                Mac99 based PowerMAC
mpc8544ds            mpc8544ds
none                 empty machine
ppce500              generic paravirt e500 platform
prep                 PowerPC PREP platform
ref405ep             ref405ep
pseries-2.1          pSeries Logical Partition (PAPR
   compliant) v2.1
```

```
pseries              pSeries Logical Partition (PAPR
   compliant) v2.2 (alias of pseries-2.2)
pseries-2.2          pSeries Logical Partition (PAPR
   compliant) v2.2 (default)
taihu                taihu
virtex-ml507         Xilinx Virtex ML507 reference
   design
```

B.1.18 PowerPC Embedded Architecture

```
$ qemu-system-ppcemb -machine ?
Supported machines are:
bamboo               bamboo
none                 empty machine
ref405ep             ref405ep
taihu                taihu
virtex-ml507         Xilinx Virtex ML507 reference
   design
```

B.1.19 ESA/390 Architecture

```
$ qemu-system-s390x -machine ?
Supported machines are:
none                 empty machine
s390-ccw             VirtIO-ccw based S390 machine (
   alias of s390-ccw-virtio)
s390-ccw-virtio      VirtIO-ccw based S390 machine
s390                 VirtIO based S390 machine (alias of
    s390-virtio)
s390-virtio          VirtIO based S390 machine (default)
```

B.1.20 SuperH4 Architecture

```
$ qemu-system-sh4 -machine ?
Supported machines are:
none                 empty machine
r2d                  r2d-plus board
shix                 shix card (default)
```

B.1.21 SuperH4 Big Endian Architecture

```
$ qemu-system-sh4eb -machine ?
Supported machines are:
none                 empty machine
r2d                  r2d-plus board
shix                 shix card (default)
```

B.1.22 SPARC Architecture

```
$ qemu-system-sparc -machine ?
Supported machines are:
LX                   Sun4m platform, SPARCstation LX
SPARCClassic         Sun4m platform, SPARCClassic
SPARCbook            Sun4m platform, SPARCbook
SS-10                Sun4m platform, SPARCstation 10
SS-20                Sun4m platform, SPARCstation 20
SS-4                 Sun4m platform, SPARCstation 4
SS-5                 Sun4m platform, SPARCstation 5 (
   default)
SS-600MP             Sun4m platform, SPARCserver 600MP
Voyager              Sun4m platform, SPARCstation
   Voyager
leon3_generic        Leon-3 generic
none                 empty machine
```

B.1.23 SPARC64 Architecture

```
$ qemu-system-sparc64 -machine ?
Supported machines are:
Niagara              Sun4v platform, Niagara
none                 empty machine
sun4u                Sun4u platform (default)
sun4v                Sun4v platform
```

B.1.24 Tricore Architecture

```
$ qemu-system-tricore -machine ?
```

```
Supported machines are:
none                 empty machine
tricore_testboard    a minimal TriCore board
```

B.1.25 Unicore32 Architecture

```
$ qemu-system-unicore32 -machine ?
Supported machines are:
none                 empty machine
puv3                 PKUnity Version-3 based on
   UniCore32 (default)
```

B.1.26 x86_64 Architecture

```
$ qemu-system-x86_64 -machine ?
Supported machines are:
pc                   Standard PC (i440FX + PIIX, 1996) (
   alias of pc-i440fx-2.2)
pc-i440fx-2.2        Standard PC (i440FX + PIIX, 1996) (
   default)
pc-i440fx-2.1        Standard PC (i440FX + PIIX, 1996)
pc-i440fx-2.0        Standard PC (i440FX + PIIX, 1996)
pc-i440fx-1.7        Standard PC (i440FX + PIIX, 1996)
pc-i440fx-1.6        Standard PC (i440FX + PIIX, 1996)
pc-i440fx-1.5        Standard PC (i440FX + PIIX, 1996)
pc-i440fx-1.4        Standard PC (i440FX + PIIX, 1996)
pc-1.3               Standard PC (i440FX + PIIX, 1996)
pc-1.2               Standard PC (i440FX + PIIX, 1996)
pc-1.1               Standard PC (i440FX + PIIX, 1996)
pc-1.0               Standard PC (i440FX + PIIX, 1996)
pc-0.15              Standard PC (i440FX + PIIX, 1996)
pc-0.14              Standard PC (i440FX + PIIX, 1996)
pc-0.13              Standard PC (i440FX + PIIX, 1996)
pc-0.12              Standard PC (i440FX + PIIX, 1996)
pc-0.11              Standard PC (i440FX + PIIX, 1996)
pc-0.10              Standard PC (i440FX + PIIX, 1996)
q35                  Standard PC (Q35 + ICH9, 2009) (
   alias of pc-q35-2.2)
```

```
pc-q35-2.2            Standard PC (Q35 + ICH9, 2009)
pc-q35-2.1            Standard PC (Q35 + ICH9, 2009)
pc-q35-2.0            Standard PC (Q35 + ICH9, 2009)
pc-q35-1.7            Standard PC (Q35 + ICH9, 2009)
pc-q35-1.6            Standard PC (Q35 + ICH9, 2009)
pc-q35-1.5            Standard PC (Q35 + ICH9, 2009)
pc-q35-1.4            Standard PC (Q35 + ICH9, 2009)
isapc                 ISA-only PC
none                  empty machine
```

B.1.27 Xtensa Architecture

```
$ qemu-system-xtensa -machine ?
Supported machines are:
kc705                 kc705 EVB (dc232b)
lx200                 lx200 EVB (dc232b)
lx60                  lx60 EVB (dc232b)
ml605                 ml605 EVB (dc232b)
none                  empty machine
sim                   sim machine (dc232b) (default)
```

B.1.28 Xtensa Big Endian Architecture

```
$ qemu-system-xtensaeb -machine ?
Supported machines are:
kc705                 kc705 EVB (fsf)
lx200                 lx200 EVB (fsf)
lx60                  lx60 EVB (fsf)
ml605                 ml605 EVB (fsf)
none                  empty machine
sim                   sim machine (fsf) (default)
```

B.2 QEMU Tricore Testboard Machine Example

```
#include "hw/hw.h"
#include "hw/devices.h"
#include "net/net.h"
```

```c
#include "sysemu/sysemu.h"
#include "hw/boards.h"
#include "hw/loader.h"
#include "sysemu/block-backend.h"
#include "exec/address-spaces.h"
#include "hw/block/flash.h"
#include "elf.h"
#include "hw/tricore/tricore.h"
#include "qemu/error-report.h"


/* Board init.  */

static struct tricore_boot_info tricoretb_binfo;

static void tricore_load_kernel(CPUTriCoreState *env)
{
    uint64_t entry;
    long kernel_size;

    kernel_size = load_elf(tricoretb_binfo.
       kernel_filename, NULL,
                           NULL, (uint64_t *)&entry,
                              NULL,
                           NULL, 0,
                           ELF_MACHINE, 1);
    if (kernel_size <= 0) {
        error_report("qemu: no kernel file '%s'",
                tricoretb_binfo.kernel_filename);
        exit(1);
    }
    env->PC = entry;

}

static void tricore_testboard_init(MachineState *machine
   , int board_id)
```

```
{
    TriCoreCPU *cpu;
    CPUTriCoreState *env;

    MemoryRegion *sysmem = get_system_memory();
    MemoryRegion *ext_cram = g_new(MemoryRegion, 1);
    MemoryRegion *ext_dram = g_new(MemoryRegion, 1);
    MemoryRegion *int_cram = g_new(MemoryRegion, 1);
    MemoryRegion *int_dram = g_new(MemoryRegion, 1);
    MemoryRegion *pcp_data = g_new(MemoryRegion, 1);
    MemoryRegion *pcp_text = g_new(MemoryRegion, 1);

    if (!machine->cpu_model) {
        machine->cpu_model = "tc1796";
    }
    cpu = cpu_tricore_init(machine->cpu_model);
    if (!cpu) {
        error_report("Unable to find CPU definition");
        exit(1);
    }
    env = &cpu->env;
    memory_region_init_ram(ext_cram, NULL, "
       powerlink_ext_c.ram", 2*1024*1024, &error_abort);
    vmstate_register_ram_global(ext_cram);
    memory_region_init_ram(ext_dram, NULL, "
       powerlink_ext_d.ram", 4*1024*1024, &error_abort);
    vmstate_register_ram_global(ext_dram);
    memory_region_init_ram(int_cram, NULL, "
       powerlink_int_c.ram", 48*1024, &error_abort);
    vmstate_register_ram_global(int_cram);
    memory_region_init_ram(int_dram, NULL, "
       powerlink_int_d.ram", 48*1024, &error_abort);
    vmstate_register_ram_global(int_dram);
    memory_region_init_ram(pcp_data, NULL, "
       powerlink_pcp_data.ram", 16*1024, &error_abort);
    vmstate_register_ram_global(pcp_data);
    memory_region_init_ram(pcp_text, NULL, "
       powerlink_pcp_text.ram", 32*1024, &error_abort);
    vmstate_register_ram_global(pcp_text);

    memory_region_add_subregion(sysmem, 0x80000000,
       ext_cram);
    memory_region_add_subregion(sysmem, 0xa1000000,
       ext_dram);
    memory_region_add_subregion(sysmem, 0xd4000000,
       int_cram);
    memory_region_add_subregion(sysmem, 0xd0000000,
       int_dram);
    memory_region_add_subregion(sysmem, 0xf0050000,
       pcp_data);
    memory_region_add_subregion(sysmem, 0xf0060000,
       pcp_text);

    tricoretb_binfo.ram_size = machine->ram_size;
    tricoretb_binfo.kernel_filename = machine->
       kernel_filename;

    if (machine->kernel_filename) {
        tricore_load_kernel(env);
    }
}

static void tricoreboard_init(MachineState *machine)
{
    tricore_testboard_init(machine, 0x183);
}

static QEMUMachine ttb_machine = {
    .name = "tricore_testboard",
    .desc = "a minimal TriCore board",
    .init = tricoreboard_init,
    .is_default = 0,
};
```

```

static void tricore_testboard_machine_init(void)
{
    qemu_register_machine(&ttb_machine);
}

machine_init(tricore_testboard_machine_init);
```

B.3 QEMU Memory-Mapped I/O LED Device Example

```
 #include "hw/sysbus.h"

/*For casting and type-checking*/
#define TYPE_MMIO_LED "mmio-led"
#define MMIO_LED(obj) OBJECT_CHECK(MMIOLEDState, (obj),
   TYPE_MMIO_LED)
#define BASE_ADDRESS 0x1340455

/*Convention for object related data structures is
   ObjectState*/
typedef struct MMIOLEDState {
    SysBusDevice parent;
    MemoryRegion iomem;
    uint32_t led_on_state;
}MMIOLEDState;

/*Hardware transactions behaviour, to be registered in
   MemoryRegionOps*/
static uint64_t mmio_led_read(void *opaque, hwaddr
   offset,
unsigned size)
{
        MMIOLEDState *s = MMIO_LED(opaque);
        if(s->led_on_state)
                printf("DEBUG: LED IS ON\n");
        else
                printf("DEBUG: LED IS OFF\n");
        return s->led_on_state;
}
static void mmio_led_write(void *opaque, hwaddr offset,
uint64_t value, unsigned size)
{
        MMIOLEDState *s = MMIO_LED(opaque);
        if(value)
        {
                s->led_on_state = 0x1;
                printf("DEBUG: LED TURNED ON\n");
        }
        else
        {
                s->led_on_state = 0x0;
                printf("DEBUG: LED TURNED OFF\n");
        }
        return;
}

/*To be associated with a MemoryRegion*/
static const MemoryRegionOps mmio_led_ops = {
        .read = mmio_led_read,
        .write = mmio_led_write,
        .endianness = DEVICE_NATIVE_ENDIAN,
};

/*Memory-mapped LED device constructor*/
static int mmio_led_init(SysBusDevice *dev)
{
    MMIOLEDState *s = MMIO_LED(dev);
    /*Memory region constructor for io-mapped devices*/
    memory_region_init_io(&s->iomem, OBJECT(s), &
       nn_led_ops, s, TYPE_MMIO_LED, 4);//4 bytes as
       block size
    /*Initiate parent mmio with constructed object*/
    sysbus_init_mmio(dev, &s->iomem);
    /*Map the MemoryRegion in index 0 of the internal
       SysBusDevice MemoryRegion array to BASE_ADDRESS*/
    sysbus_mmio_map(sysbus_dev, 0, BASE_ADDRESS);
    /*LED is initally turned off*/
        s->led_on_state = 0x0;
        printf("DEBUG: initializing LED device\n");
    return 0;
}

/*Class related initiation and static construction*/
static void mmio_led_class_init(ObjectClass *obj_class,
   void *data)
{
    SysBusDeviceClass *c = SYS_BUS_DEVICE_CLASS(
       obj_class);
    /*register Memory-mapped LED device constructor*/
    c->init = mmio_led_init;
    printf("DEBUG: registering MMIO_LED type...\n");
}
/*Memory-Mapped LED device type data register*/
static const TypeInfo mmio_led_info = {
    .name          = TYPE_MMIO_LED,
    .parent        = TYPE_SYS_BUS_DEVICE,
    .instance_size = sizeof(MMIOLEDState),
    .class_init    = mmio_led_class_init,
};
static void mmio_led_register_types(void)
{
    type_register_static(&mmio_led_info);
}
type_init(mmio_led_register_types)
```

Appendix C

VPI Example Routines

C.1 Pow System Function

```
#include <stdlib.h> /* ANSI C standard library */
#include <stdio.h> /* ANSI C standard input/output
   library */
#include <stdarg.h> /* ANSI C standard arguments library
    */
#include "vpi_user.h" /* IEEE 1364 PLI VPI routine
   library */
#include <math.h> /*Mathematical operations library*/
```

C.1.1 Calltf Routine

```
PLI_INT32 PLIbook_PowCalltf(PLI_BYTE8 *user_data)
{
        s_vpi_value value_s;
        vpiHandle systf_handle, arg_itr, arg_handle;
        PLI_INT32 base, exp;
        double result;
        systf_handle = vpi_handle(vpiSysTfCall, NULL);
        arg_itr = vpi_iterate(vpiArgument, systf_handle)
           ;
        if (arg_itr == NULL)
        {
                vpi_printf("ERROR: $pow failed to obtain
                    systf arg handles\n");
                return(0);
        }
        /* read base from systf arg 1 (compiletf has
           already verified) */
        arg_handle = vpi_scan(arg_itr);
        value_s.format = vpiIntVal;
        vpi_get_value(arg_handle, &value_s);
        base = value_s.value.integer;
        /* read exponent from systf arg 2 (compiletf has
            already verified) */
        arg_handle = vpi_scan(arg_itr);
        vpi_free_object(arg_itr); /* not calling scan
           until returns null */
        vpi_get_value(arg_handle, &value_s);
        exp = value_s.value.integer;
        /* calculate result of base to power of exponent
            */
        result = pow( (double)base, (double)exp );
        /* write result to simulation as return value $
           pow */
        value_s.value.integer = (PLI_INT32)result;
        vpi_put_value(systf_handle, &value_s, NULL,
           vpiNoDelay);
        return(0);
}
```

C.1.2 Compiletf Routine

```
PLI_INT32 PLIbook_PowCompiletf(PLI_BYTE8 *user_data)
{
        vpiHandle systf_handle, arg_itr, arg_handle;
        PLI_INT32 tfarg_type;
        int err_flag = 0;
        do
```

```
{ /* group all tests, so can break out of group
    on error */
        systf_handle = vpi_handle(vpiSysTfCall,
           NULL);
        arg_itr = vpi_iterate(vpiArgument,
           systf_handle);
        if (arg_itr == NULL)
        {
                vpi_printf("ERROR: $pow requires
                    2 arguments; has none\n");
                err_flag = 1;
                break;
        }
        arg_handle = vpi_scan(arg_itr);
        tfarg_type = vpi_get(vpiType, arg_handle
           );
        if( (tfarg_type != vpiReg) && (
           tfarg_type != vpiIntegerVar) && (
           tfarg_type != vpiConstant) )
        {
                vpi_printf("ERROR: $pow arg1
                   must be number, variable or
                   net\n");
                err_flag = 1;
                break;
        }
        arg_handle = vpi_scan(arg_itr);
        if (arg_handle == NULL)
        {
                vpi_printf("ERROR: $pow requires
                    2nd argument\n");
                err_flag = 1;
                break;
        }
        tfarg_type = vpi_get(vpiType, arg_handle
           );
```

```
                if ( (tfarg_type != vpiReg) && (
                   tfarg_type != vpiIntegerVar) && (
                   tfarg_type != vpiConstant) )
                {
                        vpi_printf("ERROR: $pow arg2
                           must be number, variable or
                           net\n");
                        err_flag = 1;
                        break;
                }
                if (vpi_scan(arg_itr) != NULL)
                {
                        vpi_printf("ERROR: $pow requires
                            2 arguments; has too many\n"
                           );
                        vpi_free_object(arg_itr);
                        err_flag = 1;
                        break;
                }
        } while (0 == 1); /* end of test group; only
           executed once */
        if (err_flag)
        {
                vpi_control(vpiFinish, 1); /* abort
                   simulation */
        }
        return(0);
}
```

C.1.3 Sizetf Routine

```
PLI_INT32 PLI_PowSizetf(PLI_BYTE8 *user_data)
{
        return(32); /* $pow returns 32-bit values */
}
```

C.1.4 System Function Register

```
void PLI_pow_register()
{
        s_vpi_systf_data tf_data;
        tf_data.type = vpiSysFunc;
        tf_data.sysfunctype = vpiSizedFunc;
        tf_data.tfname = "$pow";
        tf_data.calltf = PLI_PowCalltf;
        tf_data.compiletf = PLI_PowCompiletf;
        tf_data.sizetf = PLI_PowSizetf;
        tf_data.user_data = NULL;
        vpi_register_systf(&tf_data);
}
void (*vlog_startup_routines[])() =
{
        /*** add user entries here ***/
        PLI_pow_register,
        0 /*** final entry must be 0 ***/
};
```

C.2 Start of Simulation Callback

```
#include <stdlib.h> /* ANSI C standard library */
#include "vpi_user.h" /* IEEE 1364 PLI VPI routine
   library */
```

C.2.1 Callback Routine

```
PLI_INT32 PLI_callback(s_cb_data *callback_data)
{
        vpi_printf("Message: Start of simulation");
}
```

C.2.2 Callback Register

```
void PLI_callback_register()
{
        s_cb_data cb_start_data;
        cb_start_data.reason = cbStartOfSimulation;
        cb_start_data.cb_rtn = PLI_callback;
        cb_start_data.obj = NULL;
        cb_start_data.user_data = NULL;
        cb_start_data.time = NULL;
        cb_start_data.value = NULL;
        cb_finish_handle = vpi_register_cb(&
           cb_start_data);
        vpi_free_object(cb_finish_handle);
}
void (*vlog_startup_routines[])() =
{
        /*** add user entries here ***/
        PLI_callback_register,
        0 /*** final entry must be 0 ***/
};
```

Appendix D

FMI Simulators

Table D.1: Simulators with FMI support

Tool	FMI Version	Model Exchange	Co-Simulation
20-sim	2.0	No Export No Import	Planned Slave No Master
	1.0	No Export No Import	Planned Slave Planned Master
20-sim	2.0	No Export Planned Import	No Slave Planned Master
	1.0	No Export Planned Import	No Slave Planned Master
Adams	2.0	Planned Export Planned Import	Available Slave Available Master
	1.0	No Export No Import	Available Slave Available Master
Amesim	2.0	No Export No Import	Planned Slave Available Master
	1.0	Available Export Available Import	Available Slave Available Master
ANSYS SCADE Display	2.0	No Export No Import	No Slave No Master
	1.0	Available Export No Import	Available Slave No Master
ANSYS	2.0	No Export	No Slave

Tool	FMI Version	Model Exchange	Co-Simulation
SCADE Suite		No Import	No Master
	1.0	Available Export No Import	Available Slave No Master
ANSYS Simplorer	2.0	No Export No Import	No Slave No Master
	1.0	No Export Available Import	Planned Slave No Master
ASim AUTOSAR Simulation	2.0	No Export No Import	No Slave No Master
	1.0	Available Export No Import	Available Slave No Master
@Source	2.0	No Export No Import	No Slave No Master
	1.0	Available Export No Import	Available Slave No Master
AVL Cruise	2.0	No Export No Import	No Slave No Master
	1.0	Planned Export No Import	Available Slave Available Master
Building Controls Virtual Test bed	2.0	No Export No Import	No Slave No Master
	1.0	No Export No Import	No Slave Available Master
CANoe	2.0	Planned Export No Import	Planned Slave Available Master
	1.0	Planned Export No Import	Planned Slave Available Master
CarMaker	2.0	No Export No Import	No Slave No Master
	1.0	No Export No Import	No Slave Available Master
CATIA	2.0	No Export No Import	No Slave No Master
	1.0	Available Export	Available Slave

Tool	FMI Version	Model Exchange	Co-Simulation
		Available Import	Available Master
ControlBuild	2.0	No Export No Import	Available Slave Available Master
	1.0	Available Export Available Import	Available Slave Available Master
CosilMate	2.0	No Export No Import	Available Master Available Slave
	1.0	No Export Available Import	Available Slave Available Master
Cybernetica CENIT	2.0	No Export No Import	No Slave No Master
	1.0	No Export Available Import	No Slave Planned Master
Cybernetica ModelFit	2.0	No Export No Import	No Slave No Master
	1.0	No Export Available Import	No Slave Available Master
DACCOSIM	2.0	No Export No Import	No Slave Available Master
	1.0	No Export No Import	No Slave No Master
DS - FMU Export from Simulink	2.0	Available Export No Import	Available Slave No Master
	1.0	Available Export No Import	Available Slave No Master
DS - FMU Import into Simulink	2.0	No Export Planned Import	No Slave Planned Master
	1.0	No Export Planned Import	No Slave Planned Master
DSHplus	2.0	No Export No Import	No Slave No Master
	1.0	Planned Export No Import	Available Slave No Master
dSPACE	2.0	No Export	No Slave

Tool	FMI Version	Model Exchange	Co-Simulation
		No Import	Available Master
SCALEXIO	1.0	No Export No Import	No Slave Available Master
dSPACE TargetLink	2.0	No Export No Import	Planned Slave No Master
	1.0	No Export No Import	No Slave No Master
dSPACE VEOS	2.0	Available Export Available Import	Available Slave Available Master
	1.0	Available Export Available Import	Available Slave Available Master
DYNA4	2.0	No Export Available Import	Available Master Available Master
	1.0	No Export Available Import	No Slave Available Master
Easy5	2.0	Planned Export Planned Import	Available Slave Available Master
	1.0	Planned Export Planned Import	Available Slave Available Master
EnergyPlus	2.0	No Export No Import	No Slave No Master
	1.0	No Export No Import	Available Slave Available Master
ETAS - ASCMO	2.0	No Export No Import	Available Slave No Master
	1.0	No Export No Import	Available Slave No Master
ETAS - FMI-based Integration and Simulation Platform	2.0	No Export No Import	No Slave No Master
	1.0	No Export Planned Import	No Slave Planned Master
ETAS -	2.0	No Export	No Slave

Tool	**FMI Version**	**Model Exchange**	**Co-Simulation**
FMU		No Import	No Master
Generator	1.0	Planned Export	No Slave
for ASCET		Planned Import	No Master
ETAS -	2.0	No Export	No Master
FMU		No Import	No Slave
Generator			
for	1.0	Planned Export	No Slaves
Simulink®		No Import	No Master
ETAS -	2.0	No Import	No Slave
		No Export	No Master
INCA-FLOW	1.0	No Export	No Slave
(MiL/SiL Connector)		Planned Import	Available Master
ETAS -	2.0	No Export	No Slave
ISOLAR-EVE		No Import	No Slave
(ETAS	1.0	No Export	Available Slave
Virtual ECU)		No Import	No Master
ETAS -	2.0	No Export	No Slave
LABCAR		No Import	No Master
-OPERATOR	1.0	No Export	Available Slave
		No Import	Available Master
Flowmaster	2.0	No Export	No Slave
		No Import	No Master
	1.0	Available Export	No Slave
		No Import	No Master
FMI Add-in	2.0	No Export	No Slave
		No Import	No Master
for Excel	1.0	No Export	No Slave
		No Import	Available Master
FMI add-on	2.0	No Export	No Slave
for NI		No Import	No Master
VeriStand	1.0	No Export	No Slave
		Available Import	Available Master
FMI	2.0	No Export	No Slave
Blockset		Available Import	Available Master

Tool	FMI Version	Model Exchange	Co-Simulation
for Simulink	1.0	No Export No Import	No Slave Available Master
FMI L Library	2.0	Planned Export Available Import	Planned Slave Available Master
	1.0	Planned Export Available Import	Planned Slave Available Master
FMI Target for Simulink Coder	2.0	No Export No Import	No Slave No Master
	1.0	No Export No Import	Available Slave No Master
FMI Toolbox for MATLAB/ Simulink	2.0	Planned Export Available Import	Available Slave Available Master
	1.0	Available Export Available Import	Available Slave Available Master
FMUSDK	2.0	Available Export No Import	Available Slave No Slave
	1.0	Available Export Available Import	Available Slave Available Master
GES	2.0	Planned Export Planned Import	Planned Slave Planned Master
	1.0	Available Export Available Import	Planned Slave Planned Master
GT-SUITE	2.0	Planned Import No Export	Planned Slave Planned Master
	1.0	No Export Available Import	Available Slave Available Master
Hopsan	2.0	No Export No Export	No Slave No Master
	1.0	Available Export Available Import	No Slave No Master
IBM Rational Rhapsody	2.0	No Export No Import	No Slave No Master
	1.0	Available Export Planned Import	Planned Slave Planned Master

Tool	**FMI Version**	**Model Exchange**	**Co-Simulation**
ICOS	2.0	No Export No Import	No Slave No Master
	1.0	No Export Available Import	Available Slave Available Master
IGNITE	2.0	No Import No Export	Planned Slave Planned Master
	1.0	No Import No Export	Planned Slave Available Master
JavaFMI	2.0	No Export No Import	No Slave Available Master
	1.0	No Export No Import	No Slave Available Master
JFMI	2.0	No Export No Import	No Slave No Master
	1.0	No Export No Import	Available Slave Available Master
JModelica.org	2.0	Available Export No Import	Available Slave Available Master
	1.0	Available Export Available Import	Available Slave Available Master
LMS Virtual.Lab Motion	2.0	No Export No Import	No Slave No Master
	1.0	No Export Available Import	Available Slave Available Master
MapleSim	2.0	Available Export Planned Import	Available Slave Planned Master
	1.0	Available Export No Import	Available Slave No Master
Mechanical Simulation: CarSim, TruckSim, BikeSim	2.0	No Export No Import	Planned Slave Planned Master
	1.0	No Export No Import	Planned Slave Planned Master
MESSINA	2.0	No Export	No Slave

Tool	FMI Version	Model Exchange	Co-Simulation
		No Import	No Master
	1.0	No Export	No Slave
		Available Import	Available Master
MWorks	2.0	No Export	No Slave
		No Import	No Master
	1.0	Available Export	Planned Slave
		Planned Import	Planned Master
NI LabVIEW	2.0	No Export	No Slave
		No Import	No Master
	1.0	No Export	No Slave
		Planned Import	No Master
OpenModelica	2.0	Available Export	Planned Slave
		Available Import	Planned Master
	1.0	Available Export	Planned Slave
		Available Import	Available Master
Ptolemy II	2.0	No Export	No Slave
		No Import	No Master
	1.0	No Export	No Slave
		No Import	Planned Master
PyFMI	2.0	No Export	No Slave
		Available Import	Available Master
	1.0	No Export	No Slave
		Available Import	Available Master
RecurDyn	2.0	No Export	No Slave
		No Import	No Master
	1.0	Planned Export	Planned Slave
		Planned Import	Available Master
Reference	2.0	No Export	No Slave
		No Import	No Master
FMUs	1.0	Planned Export	Planned Slave
		No Import	No Master
Scilab/	2.0	No Export	No Slave
Xcos		No Import	No Master
FMU	1.0	Planned Export	Planned Slave

Tool	**FMI Version**	**Model Exchange**	**Co-Simulation**
wrapper		Planned Import	Planned Master
Silver	2.0	No Export Available Import	Available Slave Available Master
	1.0	Available Export Available Import	Available Slave Available Master
SIMPACK	2.0	Planned Export Available Import	Available Slave Available Master
	1.0	Available Export Available Import	Available Slave Available Master
SimulationX	2.0	Planned Export Planned Import	Planned Slave Planned Master
	1.0	Available Export Available Import	Available Slave Available Master
SystemModeler	2.0	No Export No Import	No Slave No Master
	1.0	Available Export Available Import	Planned Slave Planned Master
TLK FMI Suite	2.0	No Export No Import	No Slave No Master
	1.0	No Export Available Import	No Slave Available Master
TLK TISC Suite	2.0	No Export No Import	No Slave No Master
	1.0	No Export Available Import	No Slave Available Master
TWT Co-Simulation Framework	2.0	No Export No Import	No Slave No Master
	1.0	Available Import No Export	Available Master Available Slave
TWT FMU Trust Centre	2.0	No Export No Import	No Slave No Master
	1.0	No Export No Import	Available Slave No Master
VALDYN	2.0	No Export	Planned Slave

Tool	FMI Version	Model Exchange	Co-Simulation
		No Import	No Master
	1.0	No Export	Available Slave
		No Import	No Master
WAVE-RT	2.0	No Export	Planned Slave
		No Import	No Master
	1.0	No Export	Available Slave
		No Import	No Master
XFlow	2.0	No Export	No Slave
		No Import	No Master
	1.0	No Export	Available Slave
		No Import	No Master
xMOD	2.0	No Export	No Slave
		Available Import	Available Master
	1.0	No Import	No Slave
		Available Export	Available Master

Appendix E

Plugin Extension Additional Material

E.1 QEMU Monitor LED Plugin Device Property Output

```
dev: plugin-device, id ""
  name = "plugin-led"
  dynamic library = "/opt/qemu/plugin_devices/
     plugin_led.so"
  memory mapped address = 268513280 (0x10013000)
  mapped area size = 4 (0x4)
  number of interrupts = 0 (0x0)
  number of timers = 0 (0x0)
  mmio 0000000010013000/0000000000000004
```

E.2 Plugin Device Structure

```
PluginDev device = {
        .name = "example_device",
        .memory_mapped_address = 0x340000,
        .mapped_area_size = 1,
        .write = write_func,
        .read = read_func,
        .init = init_func,
        .exit = exit_func,
        .interrupt_requests_size = 0,
        .timer_requests_size = 0
};
```

E.3 Plugin Device Makefile

```
NAME=example_device
INC=/usr/include/glib-2.0 /usr/lib/x86_64-linux-gnu/glib-2.0/include $(QEMU_SOURCE
    ) $(QEMU_SOURCE)/include

CC= gcc
CFLAGS = -c -fPIC
LDFLAGS = -shared
INC_PARAMS = $(INC:%=-I%)
SRC = $(wildcard *.c)
OBJ = $(SRC:.c=.o)


$(NAME).so: $(OBJ)
        $(CC) $(LDFLAGS) $(OBJ) -o $@

%.o: %.c
        $(CC) $(CFLAGS) $(INC_PARAMS) $< -o $@

.PHONY: clean

clean:
        rm -f *.o *~ $(NAME).so

install:
        cp -f $(NAME).so $(QEMU_PLUGINS)

uninstall:
        rm $(QEMU_PLUGINS)/$(NAME).so
```

E.4 LED Plugin Device

```
#define DEVICE_MODULE /*Every device module should
    contain this macro*/

#include "hw/plugin_dev_user.h" /*plugin device API*/
#include <stdlib.h> /*malloc, free*/

```

```
void write_func(hwaddr offset, unsigned size, uint64_t
    value, void *user_data)
{
        int *state = (int *)user_data;
        if(value == '1')
                *state = 0x1;
        else if (value == '0')
                *state = 0x0;
}
uint64_t read_func(hwaddr offset, unsigned size, void *
    user_data)
{
        int *state = (int *)user_data;
        if(*state)
                return '1';
        else
                return '0';
}
void init_func(void **user_data)
{
        *user_data = malloc(sizeof(int));
}
void exit_func(void *user_data)
{
        free(user_data);
}
PluginDev device = {
        .name = "plugin-led",
        .memory_mapped_address = 0x10013000,
        .mapped_area_size = sizeof(uint32_t),
        .write = write_func,
        .read = read_func,
        .init = init_func,
        .exit = exit_func,
        .interrupt_requests_size = 0,
        .timer_requests_size = 0
};
```

E.5 LED Plugin Device with IRQ

```c
#define DEVICE_MODULE /*Every device module should contain this macro*/

#include "hw/plugin_dev_user.h" /*plugin device API*/
#include <stdlib.h> /*malloc, free*/

void write_func(hwaddr offset, unsigned size, uint64_t value, void *user_data)
{
        switch(offset)
        {
                case 0x00:
                {
                        int *state = (int *)user_data;
                        if(value == '1')
                        {
                                *state = 0x1;
                                /*index 0 to use the first requested interrupt*/
                                irq_raise(interrupts[0]);
                        }
                        else if (value == '0')
                        {
                                *state = 0x0;
                        }
                }
                break;
                case 0x04:
                        /*index 0 to use the first requested interrupt*/
                        irq_lower(interrupts[0]);
                break;
        }
```

```
}
uint64_t read_func(hwaddr offset, unsigned size, void *
    user_data)
{
        int *state = (int *)user_data;
        switch(offset)
        {
                case 0x00:
                {
                        int *state = (int *)user_data;
                        if(*state)
                                return '1';
                        else
                                return '0';
                }
                break;
                case 0x4: //not mapped to anything
                        return 0;
                break;
        }
}
void init_func(void **user_data)
{
        *user_data = malloc(sizeof(int));
}
void exit_func(void *user_data)
{
        free(user_data);
}
/*Request interrupt on index 8 of the target machine's
    IRQ array*/
unsigned interrupt_numbers[] = {
        8
};
PluginDev device = {
        .name = "plugin-led-irq",
        .memory_mapped_address = 0x10013000,
        .mapped_area_size = sizeof(uint32_t) * 2,
        .write = write_func,
        .read = read_func,
        .init = init_func,
        .exit = exit_func,
        .interrupt_requests = interrupt_numbers,
        .interrupt_requests_size = sizeof(
          interrupt_numbers)/sizeof(unsigned),
        .timer_requests_size = 0

};
```

E.6 Adding IRQ Support in a Target QEMU Machine

If a target machine has interrupts, an IRQ array is present on its initiation code. According to QEMU machine source file convention, object functions registering should be at the bottom of the file.

Firstly, one should find the initialization function. Secondly, identify the qemu_irq array that is being used on device instantiation. Sometimes it is declared in the beginning of the machine initialization function, sometimes it's not, there's no convention.

Lastly, after all devices have been instantiated in the initialization function, call plugin_devs_set_irqs function provided in hw/plugin_dev_internals.h file, to register the qemu_irq array, just before a load kernel function is called, if it exists. If the irq array is declared statically, sizeof operand may be used to register its size. If not, irq array size must be figured out and registered explicitly.

The following is an example to add IRQ support in the Microblaze architecture's PetaLogix refdesign for Xilinx ml605 little endian machine, with the added line highlited in red:

```
...
            sysbus_connect_irq(busdev, i+1, cs_line);
        }
    }
```

```

                    plugin_devs_set_irqs(irq, sizeof(irq));
    microblaze_load_kernel(cpu, MEMORY_BASEADDR,
       ram_size,
                                machine->initrd_filename,
                                BINARY_DEVICE_TREE_FILE,
                                machine_cpu_reset);

}
...
```

E.7 Blinking LED Plugin Device

```
#define DEVICE_MODULE /*Every device module should
   contain this macro*/

#include "hw/plugin_dev_user.h" /*plugin device API*/
#include <stdlib.h> /*malloc, free*/

void write_func(hwaddr offset, unsigned size, uint64_t
   value, void *user_data)
{
        int *state = (int *)user_data;
        if(value == '1')
                start_timer(timers[0], 500); //500ms
                   blinking period
        else if (value == '0')
                stop_timer(timers[0]);
}
uint64_t read_func(hwaddr offset, unsigned size, void *
   user_data)
{
        int *state = (int *)user_data;
        if(*state)
                return '1';
        else
```

```
                return '0';
}
void init_func(void **user_data)
{
        *user_data = malloc(sizeof(int));
}
void exit_func(void *user_data)
{
        free(user_data);
}
void timer_handler(void *user_data)
{
        int *state = (int *)user_data;
        *state = !(*state);
        start_timer(timers[0], 500);//500ms blinking
           period
}
timer_request_t t_request[] = {
        { QEMU_CLOCK_VIRTUAL, timer_handler, SCALE_MS}
}
PluginDev device = {
        .name = "blinking-plugin-led",
        .memory_mapped_address = 0x10013000,
        .mapped_area_size = sizeof(uint32_t),
        .write = write_func,
        .read = read_func,
        .init = init_func,
        .exit = exit_func,
        .interrupt_requests_size = 0,
        .timers_request = t_request,
        .timers_request_size = sizeof(t_request)/sizeof(
           t_request)
};
```

E.8 Master LED Proxy Plugin Device

```
#define DEVICE_MODULE /*Every device module should
```

```
    contain this macro*/

#include "hw/plugin_dev_user.h" /*plugin device API*/
#include <stdlib.h> /*malloc, free*/

void write_func(hwaddr offset, unsigned size, uint64_t
    value, void *user_data)
{

        write_to_bus(0x10013000, (char *)value, sizeof(
           uint64_t));
}
uint64_t read_func(hwaddr offset, unsigned size, void *
    user_data)
{
        read_from_bus(0x10013000, (char *)value, sizeof(
           uint64_t));
        return 0;
}
void init_func(void **user_data)
{
}

void exit_func(void *user_data)
{
}

PluginDev device = {
        .name = "master-to-led",
        .memory_mapped_address = 0x10013018,
        .mapped_area_size = sizeof(uint32_t),
        .write = write_func,
        .read = read_func,
        .init = NULL,
        .exit = NULL,
        .interrupt_requests_size = 0,
        .timers_request_size = 0
};
```

Appendix F

Co-simulation Additional Material

F.1 QEMU Monitor LED HW IP Proxy Interface Device Property Output

```
dev: external-model, id ""
    model name = "LED HW IP"
    simulation domain = "Hardware Acceleration"
    simulation tool = "Modelsim"
    tool ip = "127.0.0.1"
    tool port = 53497 (0xd0f9)
    memory mapped address = 268513280 (0x10013000)
    mapped area size = 38 (0x26)
    number of interrupts = 0 (0x0)
    mmio 0000000010013000/0000000000000026
```

F.2 Virtual QEMU System Bus Slave Logic Example

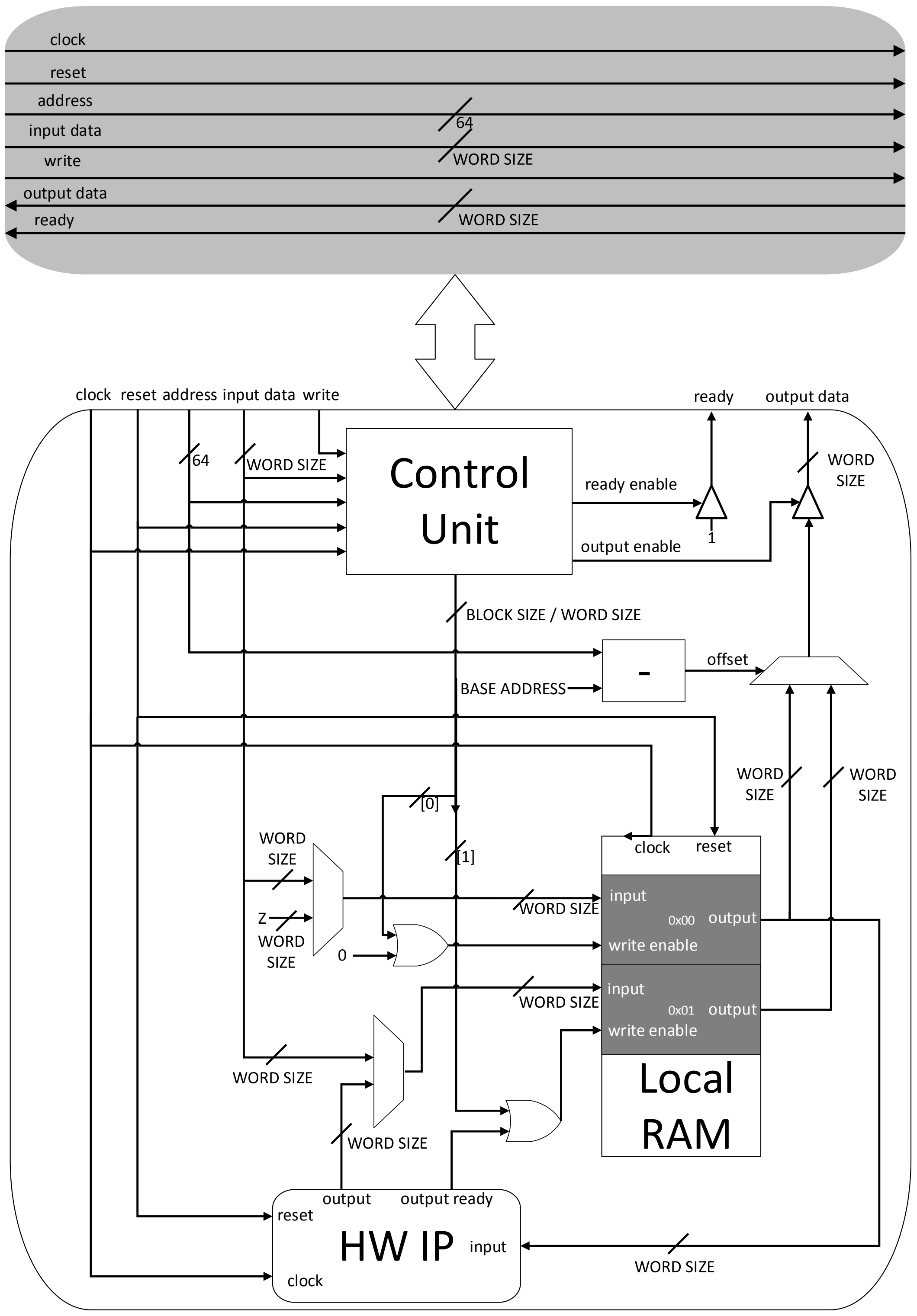


Figure F.1: Virtual QEMU system bus slave logic example diagram

F.3 Verilog QEMU Virtual System Bus Slave IP Wrapper Example

```verilog
module wrapper#(parameter WORD = 32)
(
    input clk,
    input rst,
    input [63 : 0]addr,
    input [WORD - 1 : 0]din,
    input write,
    output [WORD - 1 : 0]dout,
    output ready
);
    localparam MAPPED_AREA_SIZE = (WORD * 2);
    localparam INTERRUPTS_SIZE = 32'd0;
    localparam MASTER = 1'd0;
    parameter MEMORY_MAPPED_ADDRESS = 64'h10013000;
    parameter NAME = "Example HW IP";

    localparam MAPPED_REG_SIZE = MAPPED_AREA_SIZE/(WORD
       /8);
    wire [63 : 0] offset = (addr - MEMORY_MAPPED_ADDRESS
       ) / (WORD/8);
    wire [ WORD - 1 : 0 ]mapped_registers_input[
       MAPPED_REG_SIZE - 1 : 0];
    wire [ WORD - 1 : 0 ]mapped_registers_next_value[
       MAPPED_REG_SIZE - 1 : 0];
    wire [ WORD - 1 : 0 ]write_enable[MAPPED_REG_SIZE -
       1 : 0];
    wire bus_write_enable[MAPPED_REG_SIZE - 1 : 0];
    wire [ WORD - 1 : 0 ]ip_write_enable[MAPPED_REG_SIZE
        - 1 : 0];
    wire [INTERRUPTS_SIZE - 1 : 0]interrupt_enable;
    wire [INTERRUPTS_SIZE - 1 : 0]interrupt_disable;

    reg [ WORD - 1 : 0 ]mapped_registers[MAPPED_REG_SIZE
```

```
        - 1 : 0];

     //Write access state machine
     //states
    localparam WAIT = 3'd0;
    localparam READ = 3'd1;
    localparam WRITE = 3'd2;
    localparam READY_READ = 3'd3;
    localparam READY_WRITE = 3'd4;
    reg [2 : 0]state;
    reg [2 : 0]next_state;

     //Sequential state logic
        always@(posedge clk)
        begin
           if(rst)
               state <= WAIT;
           else
               state <= next_state;
        end

        //Next state logic
        always@(*)
        begin
            case (state)
                WAIT:
                begin
                    if((addr >= MEMORY_MAPPED_ADDRESS)
                       && (addr < (MEMORY_MAPPED_ADDRESS
                        + MAPPED_AREA_SIZE)))
                        next_state = write ? WRITE :
                           READ;
                    else
                        next_state = WAIT;
                end
                READ:
                   next_state = READY_READ;
                WRITE:
                    next_state = READY_WRITE;
                READY_READ:
                   next_state =MEM WAIT;
                READY_WRITE:
                   next_state = WAIT;
            endcase
        end

    assign dout = (state == READ || state == READY_READ)
        ? mapped_registers[offset] : 'hz;
    assign ready = (state == READY_READ || state ==
       READY_WRITE) ? 'h1 : 'hz;


    genvar i;
    genvar j;
    generate
        for (i=0; i < MAPPED_REG_SIZE; i=i+1) begin :
           MAPPED_REGS
            assign bus_write_enable[i] = ((state ==
               WRITE) && (offset == i));
            assign mapped_registers_next_value[i] = (
               bus_write_enable[i]) ? din :
               mapped_registers_input[i];

            for (j=0; j < WORD; j=j+1) begin :
               MAPPED_REGS_BITS
                assign write_enable[i][j] = (
                   bus_write_enable[i] ||
                   ip_write_enable[i][j]);

                always@(posedge clk)
                begin
                    if(rst)
                        mapped_registers[i][j] <= 'd0;
                    else if(write_enable[i][j])
```

```
                            mapped_registers[i][j] <=
                               mapped_registers_next_value[i
                               ][j];
                    end
                end
            end
        endgenerate

        generate
                for (i=0; i < INTERRUPTS_SIZE; i=i+1) begin
                   : INTERRUPTS
                    always@(posedge interrupt_enable[i])
                    begin
                    if(!rst)
                        $qemu_raise_interrupt(i);
                    end
                    always@(negedge interrupt_disable[i])
                    begin
                    if(!rst)
                        $qemu_lower_interrupt(i);
                    end
                end
        endgenerate


         //IP instantiation
        wire [WORD -1 : 0]_input;
        wire [WORD -1 : 0]_output;
        wire _output_ready
        example_IP example_instance(clock(clk), .reset(rst),
            .input(_input), .output(_output), .output_ready(
               _output_ready)
                                    );

        //Port INPUT
        assign _input = mapped_registers[0];
        assign mapped_registers_input[0] = 'hz;
```

```
    assign ip_write_enable[0] = 'd0;

    //PORT OUTPUT
    assign mapped_registers_input[1] = _output;
    assign ip_write_enable[1] = {WORD{_output_ready}};

endmodule
```

F.4 Verilog QEMU virtual System bus Slave IP wrapper with IRQ Mapping Example

```
module wrapper#(parameter WORD = 32)
(
    input clk,
    input rst,
    input [63 : 0]addr,
    input [WORD - 1 : 0]din,
    input write,
    output [WORD - 1 : 0]dout,
    output ready
);
    localparam MAPPED_AREA_SIZE = (WORD * 2);
    localparam INTERRUPTS_SIZE = 32'd1;
    localparam MASTER = 1'd0;
    parameter MEMORY_MAPPED_ADDRESS = 64'h10013000;
    parameter NAME = "Example HW IP";
                localparam INTERRUPT_0 = 32'd8;

    localparam MAPPED_REG_SIZE = MAPPED_AREA_SIZE/(WORD
       /8);
    wire [63 : 0] offset = (addr - MEMORY_MAPPED_ADDRESS
       ) / (WORD/8);
    wire [ WORD - 1 : 0 ]mapped_registers_input[
       MAPPED_REG_SIZE - 1 : 0];
    wire [ WORD - 1 : 0 ]mapped_registers_next_value[
       MAPPED_REG_SIZE - 1 : 0];
    wire [ WORD - 1 : 0 ]write_enable[MAPPED_REG_SIZE -
       1 : 0];
    wire bus_write_enable[MAPPED_REG_SIZE - 1 : 0];
    wire [ WORD - 1 : 0 ]ip_write_enable[MAPPED_REG_SIZE
        - 1 : 0];
    wire [INTERRUPTS_SIZE - 1 : 0]interrupt_enable;
    wire [INTERRUPTS_SIZE - 1 : 0]interrupt_disable;

    reg [ WORD - 1 : 0 ]mapped_registers[MAPPED_REG_SIZE
        - 1 : 0];

     //Write access state machine
     //states
    localparam WAIT = 3'd0;
    localparam READ = 3'd1;
    localparam WRITE = 3'd2;
    localparam READY_READ = 3'd3;
    localparam READY_WRITE = 3'd4;
    reg [2 : 0]state;
    reg [2 : 0]next_state;

     //Sequential state logic
        always@(posedge clk)
        begin
           if(rst)
               state <= WAIT;
           else
               state <= next_state;
        end

        //Next state logic
        always@(*)
        begin
            case (state)
                WAIT:
                begin
                    if((addr >= MEMORY_MAPPED_ADDRESS)
                      && (addr < (MEMORY_MAPPED_ADDRESS + MAPPED_AREA_SIZE))
                        next_state = write ? WRITE : READ;
                    else
                        next_state = WAIT;
                end
                READ:
                   next_state = READY_READ;
                WRITE:
                    next_state = READY_WRITE;
                READY_READ:
                   next_state =MEM WAIT;
                READY_WRITE:
                   next_state = WAIT;
            endcase
        end

    assign dout = (state == READ || state == READY_READ) ? mapped_registers[offset] : 'hz;
    assign ready = (state == READY_READ || state == READY_WRITE) ? 'h1 : 'hz;


    genvar i;
    genvar j;
    generate
        for (i=0; i < MAPPED_REG_SIZE; i=i+1) begin : MAPPED_REGS
            assign bus_write_enable[i] = ((state == WRITE) && (offset == i));
            assign mapped_registers_next_value[i] = (bus_write_enable[i]) ? din : mapped_registers_input[i];

            for (j=0; j < WORD; j=j+1) begin : MAPPED_REGS_BITS
```

```
                assign write_enable[i][j] = (
                   bus_write_enable[i] ||
                   ip_write_enable[i][j]);

                always@(posedge clk)
                begin
                    if(rst)
                        mapped_registers[i][j] <= 'd0;
                    else if(write_enable[i][j])
                        mapped_registers[i][j] <=
                           mapped_registers_next_value[i
                           ][j];
                end
            end
        end
    endgenerate

    generate
            for (i=0; i < INTERRUPTS_SIZE; i=i+1) begin
               : INTERRUPTS
                always@(posedge interrupt_enable[i])
                begin
                if(!rst)
                    $qemu_raise_interrupt(i);
                end
                always@(negedge interrupt_disable[i])
                begin
                if(!rst)
                    $qemu_lower_interrupt(i);
                end
            end
    endgenerate


     //IP instantiation
    wire [WORD -1 : 0]_input;
    wire [WORD -1 : 0]_output;
    wire _output_ready
    example_IP example_instance(clock(clk), .reset(rst),
        .input(_input), .output(_output), .output_ready(
            _output_ready)
                                    );

    //Port INPUT
    assign _input = mapped_registers[0];
    assign mapped_registers_input[0] = 'hz;
    assign ip_write_enable[0] = 'd0;

    //PORT OUTPUT
    assign mapped_registers_input[1] = _output;
    assign ip_write_enable[1] = {WORD{_output_ready}};

                  /INTERRUPT0
    //enable
    assign interrupt_enable[0] = _output_ready;
    assign interrupt_disable[0] = mapped_registers
       [0][0];
    assign mapped_registers_input[0][0] =
       interrupt_enable[0];
    assign ip_write_enable[0][0] = interrupt_enable[0];

endmodule
```

F.5 Sequence diagrams

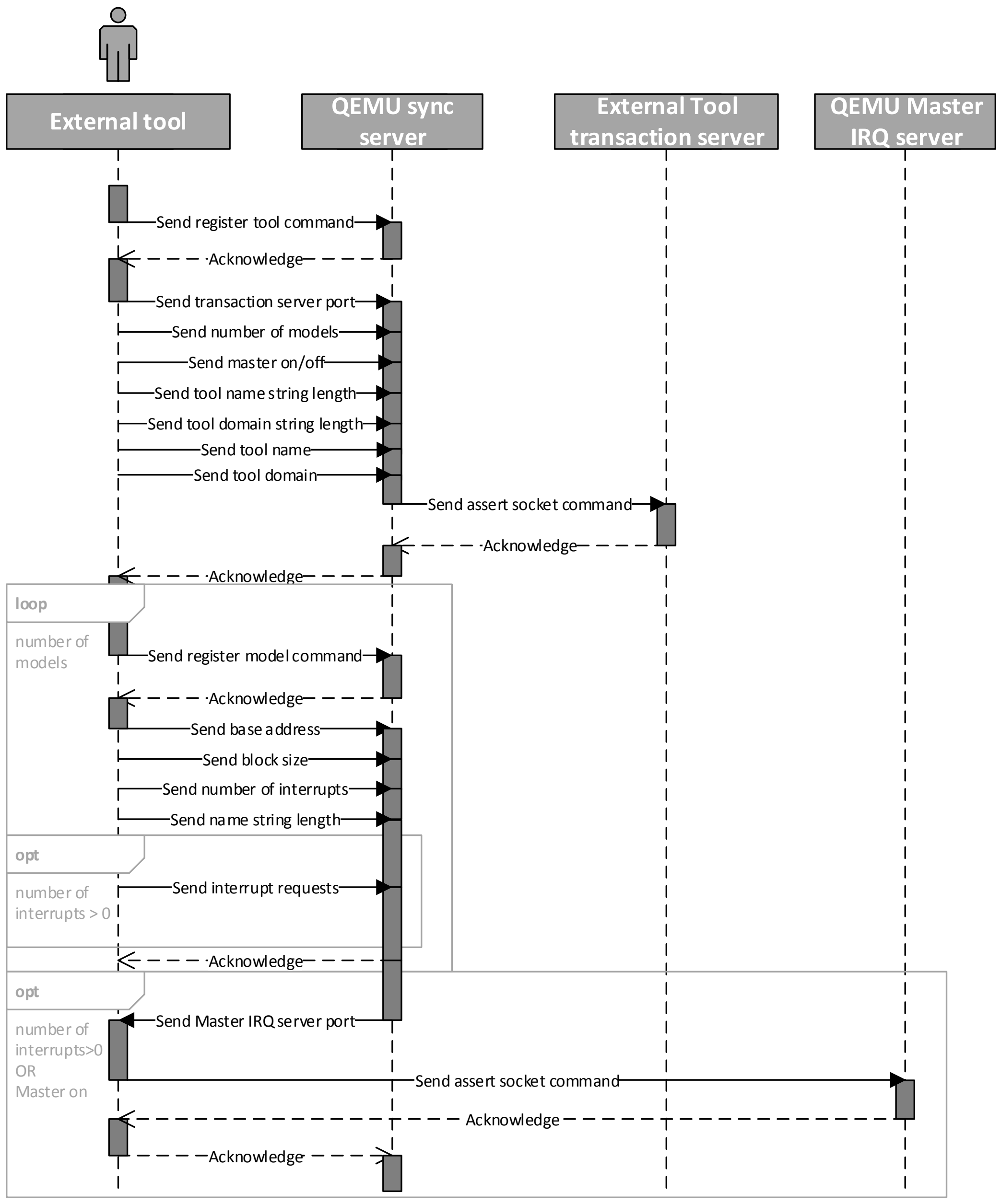
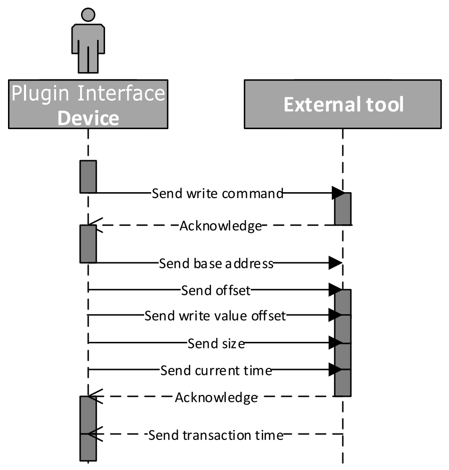
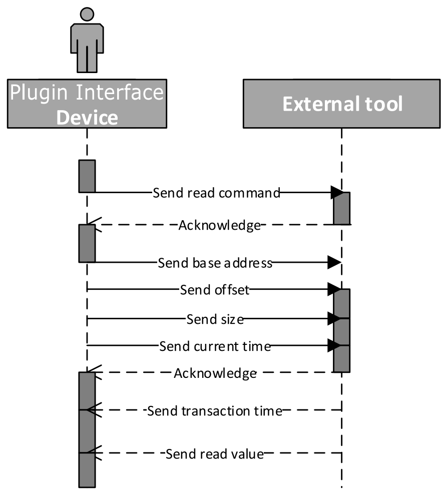


Figure F.2: Start-up server synchronization sequence diagram

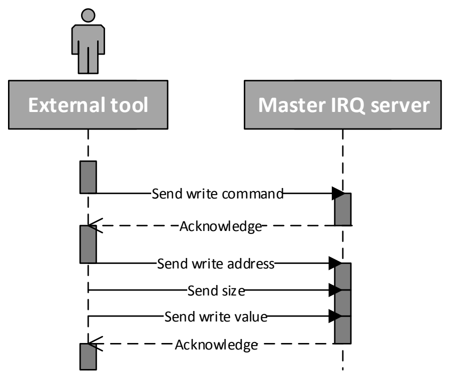


(a) Proxy interface device write sequence diagram

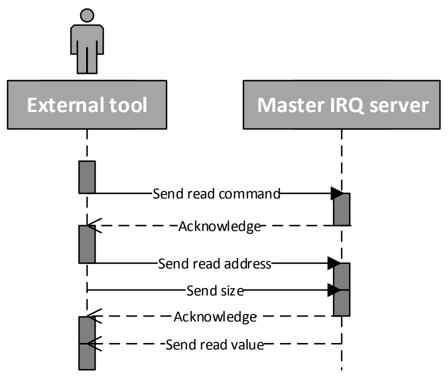


(b) Proxy interface device read sequence diagram

Figure F.3: Proxy transactions sequence diagram



(a) Master write sequence diagram



(b) Master read sequence diagram

Figure F.4: Master transactions sequence diagram

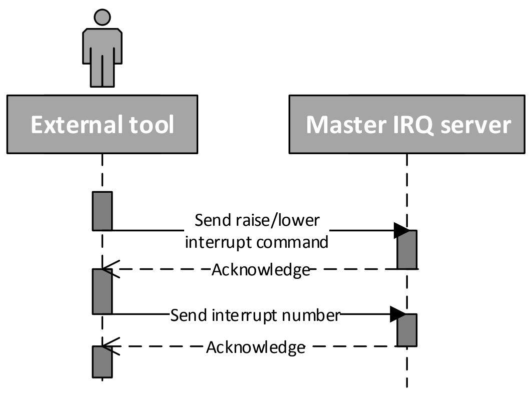


Figure F.5: Interrupt sequence diagram

F.6 QEMU Co-simulation PLI library

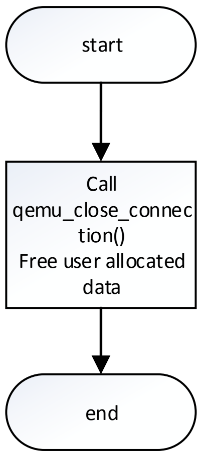


Figure F.6: End of simulation simulation callback flowchart

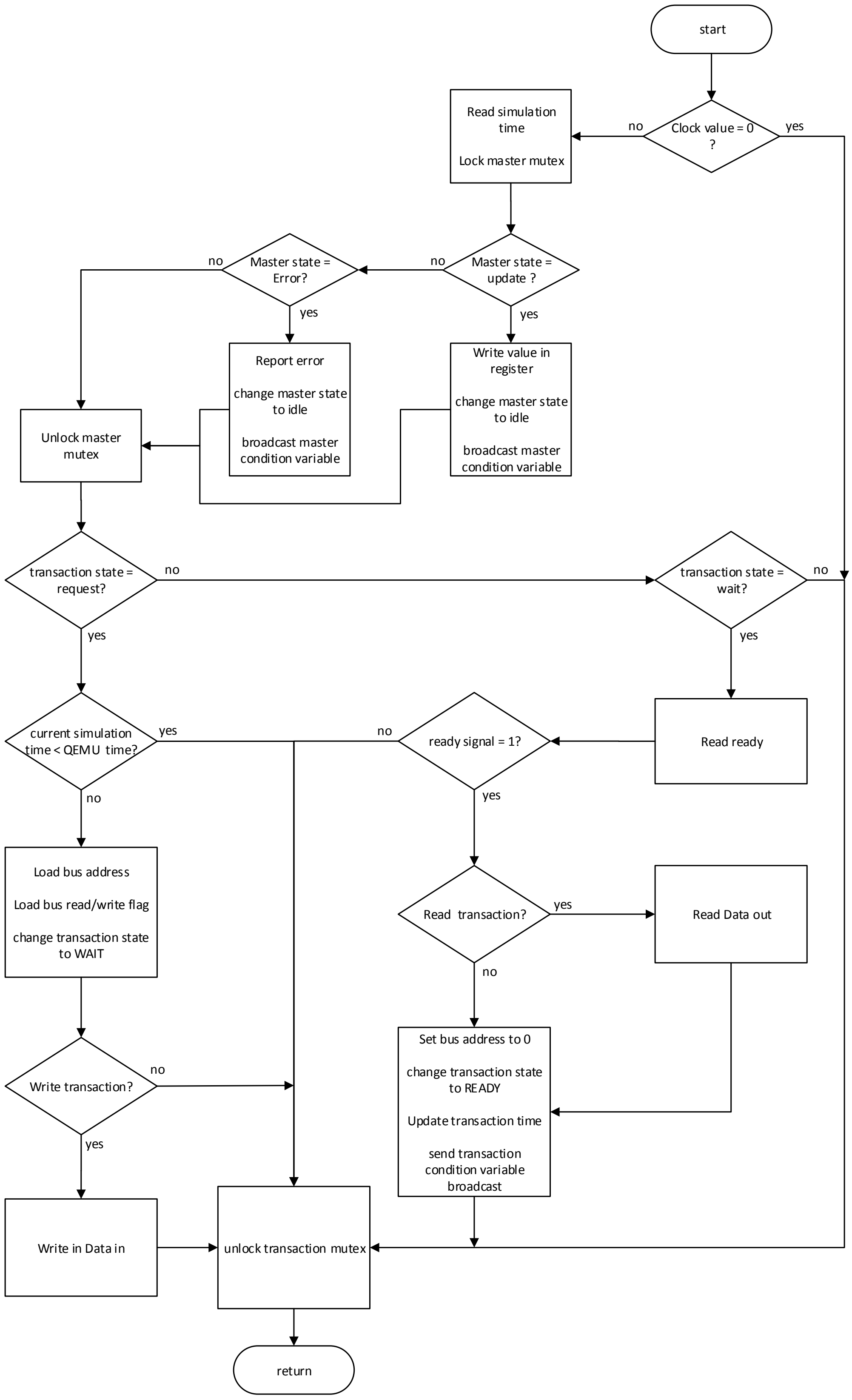
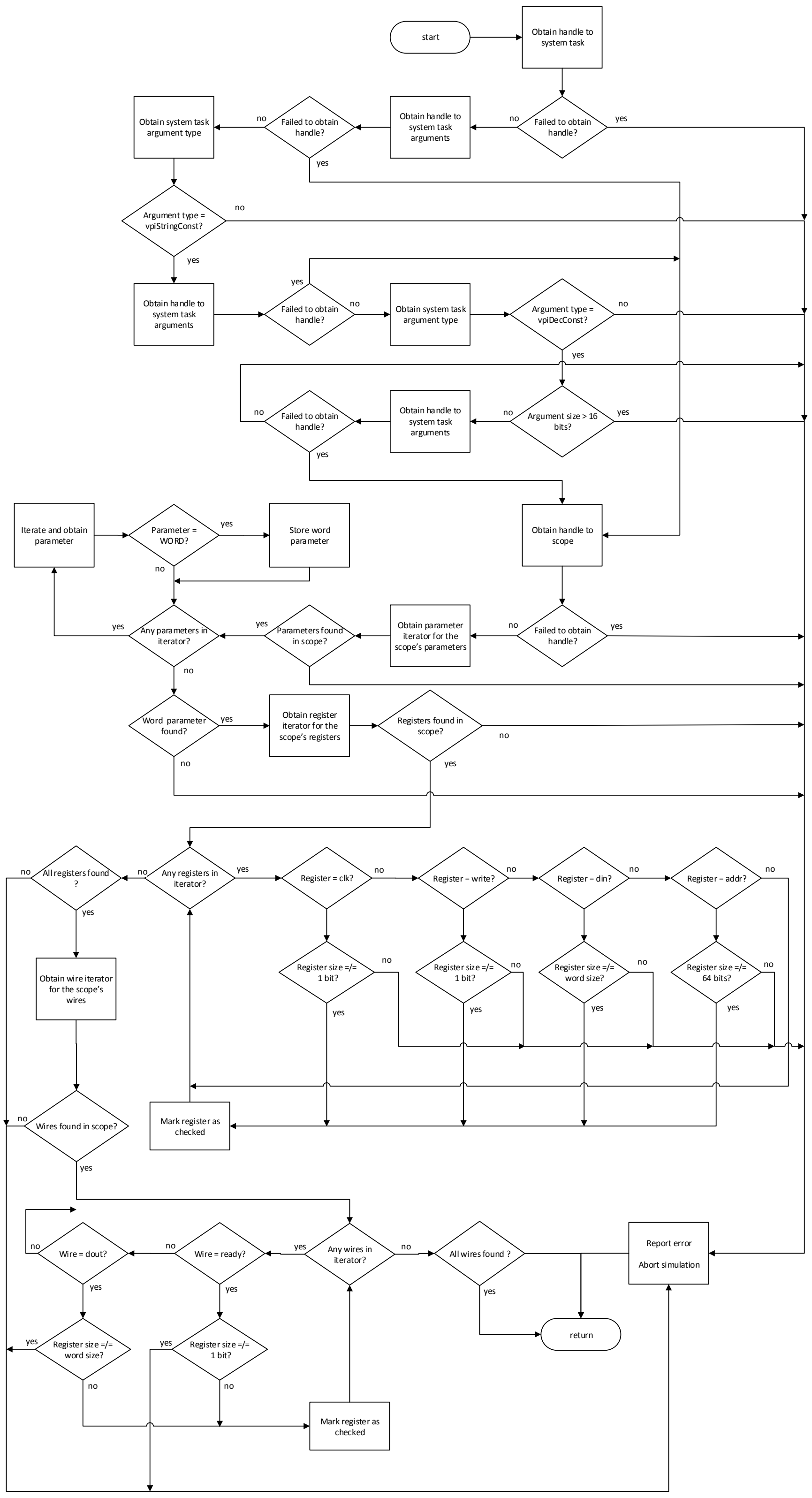
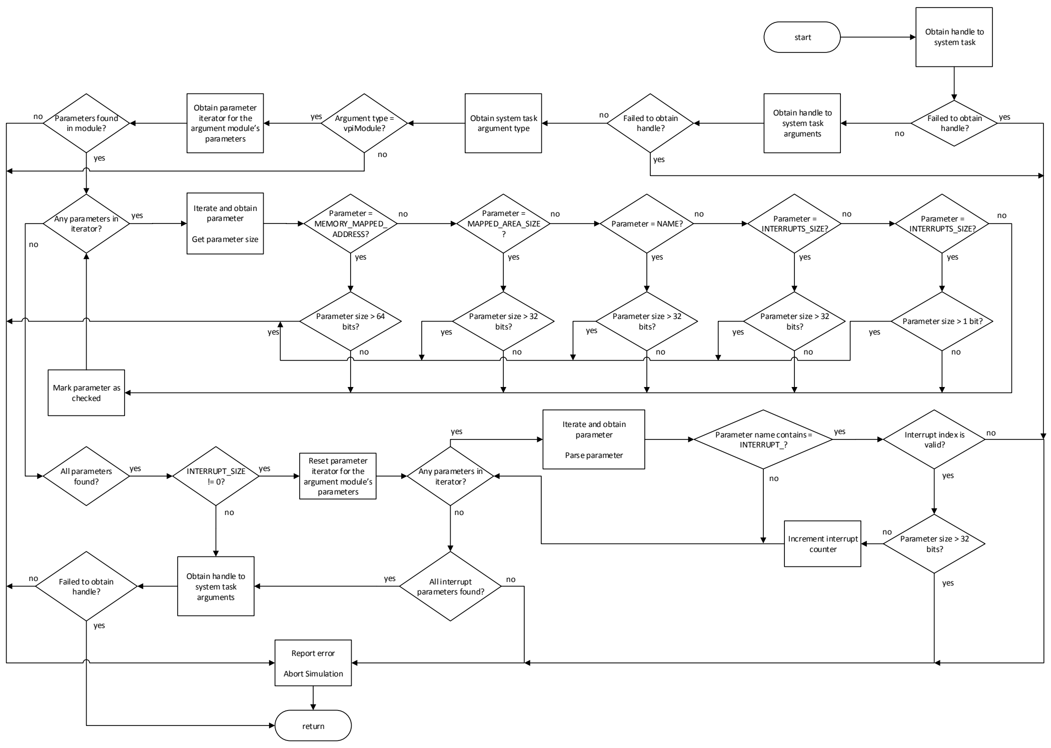


Figure F.7: Clock change simulation callback flowchart



start
Obtain handle to system task
Failed to obtain handle?
Obtain handle to system task arguments
Failed to obtain handle?
Obtain system task argument type
Argument type = vpiStringConst?
Obtain handle to system task arguments
Failed to obtain handle?
Obtain system task argument type
Argument type = vpiDecConst?
Argument size > 16 bits?
Obtain handle to system task arguments
Failed to obtain handle?
Obtain handle to scope
Failed to obtain handle?
Obtain parameter iterator for the scope's parameters
Parameters found in scope?
Any parameters in iterator?
Iterate and obtain parameter
Parameter = WORD?
Store word parameter
Word parameter found?
Obtain register iterator for the scope's registers
Registers found in scope?
Any registers in iterator?
All registers found ?
Register = clk?
Register = write?
Register = din?
Register = addr?
Register size =/= 1 bit?
Register size =/= 1 bit?
Register size =/= word size?
Register size =/= 64 bits?
Mark register as checked
Obtain wire iterator for the scope's wires
Wires found in scope?
Any wires in iterator?
Wire = ready?
Wire = dout?
Register size =/= 1 bit?
Register size =/= word size?
Mark register as checked
All wires found ?
Report error
Abort simulation
return
yes
no

Figure F.8: $qemu_connect compiletf flowchart



start
Obtain handle to system task
Failed to obtain handle?
yes
no
Obtain handle to system task arguments
Failed to obtain handle?
no
yes
Obtain system task argument type
Argument type = vpiModule?
yes
no
Obtain parameter iterator for the argument module's parameters
Parameters found in module?
no
yes
Any parameters in iterator?
yes
no
Iterate and obtain parameter
Get parameter size
Parameter = MEMORY_MAPPED_ADDRESS?
no
yes
Parameter = MAPPED_AREA_SIZE ?
no
yes
Parameter = NAME?
no
yes
Parameter = INTERRUPTS_SIZE?
no
yes
Parameter = INTERRUPTS_SIZE?
no
yes
Parameter size > 64 bits?
yes
no
Parameter size > 32 bits?
yes
no
Parameter size > 32 bits?
yes
no
Parameter size > 32 bits?
yes
no
Parameter size > 1 bit?
yes
no
Mark parameter as checked
All parameters found?
yes
INTERRUPT_SIZE != 0?
yes
no
Reset parameter iterator for the argument module's parameters
Any parameters in iterator?
yes
no
Iterate and obtain parameter
Parse parameter
Parameter name contains = INTERRUPT_?
yes
no
Interrupt index is valid?
no
yes
Parameter size > 32 bits?
no
yes
Increment interrupt counter
All interrupt parameters found?
yes
no
Obtain handle to system task arguments
Failed to obtain handle?
no
yes
Report error
Abort Simulation
return

Figure F.9: $qemu_register_model compiletf flowchart

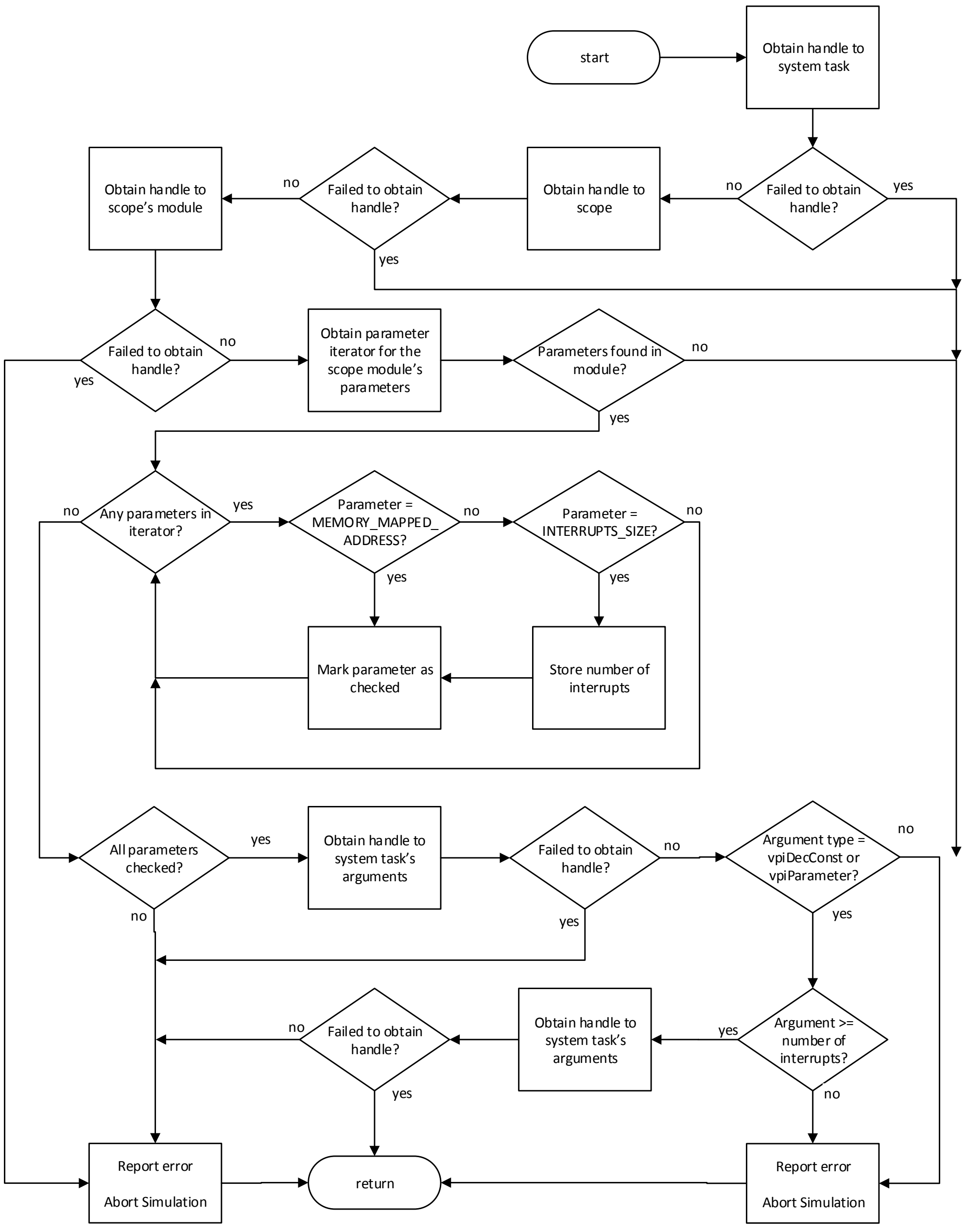

start
Obtain handle to system task
Failed to obtain handle?
yes
no
Obtain handle to scope
Failed to obtain handle?
no
yes
Obtain handle to scope's module
Failed to obtain handle?
no
yes
Obtain parameter iterator for the scope module's parameters
Parameters found in module?
no
yes
Any parameters in iterator?
no
yes
Parameter = MEMORY_MAPPED_ ADDRESS?
no
yes
Parameter = INTERRUPTS_SIZE?
no
yes
Mark parameter as checked
Store number of interrupts
All parameters checked?
yes
no
Obtain handle to system task's arguments
Failed to obtain handle?
no
yes
Argument type = vpiDecConst or vpiParameter?
no
yes
Argument >= number of interrupts?
yes
no
Obtain handle to system task's arguments
Failed to obtain handle?
no
yes
Report error
Abort Simulation
return
Report error
Abort Simulation

Figure F.10: $qemu_raise/lower_interrupt compiletf flowchart

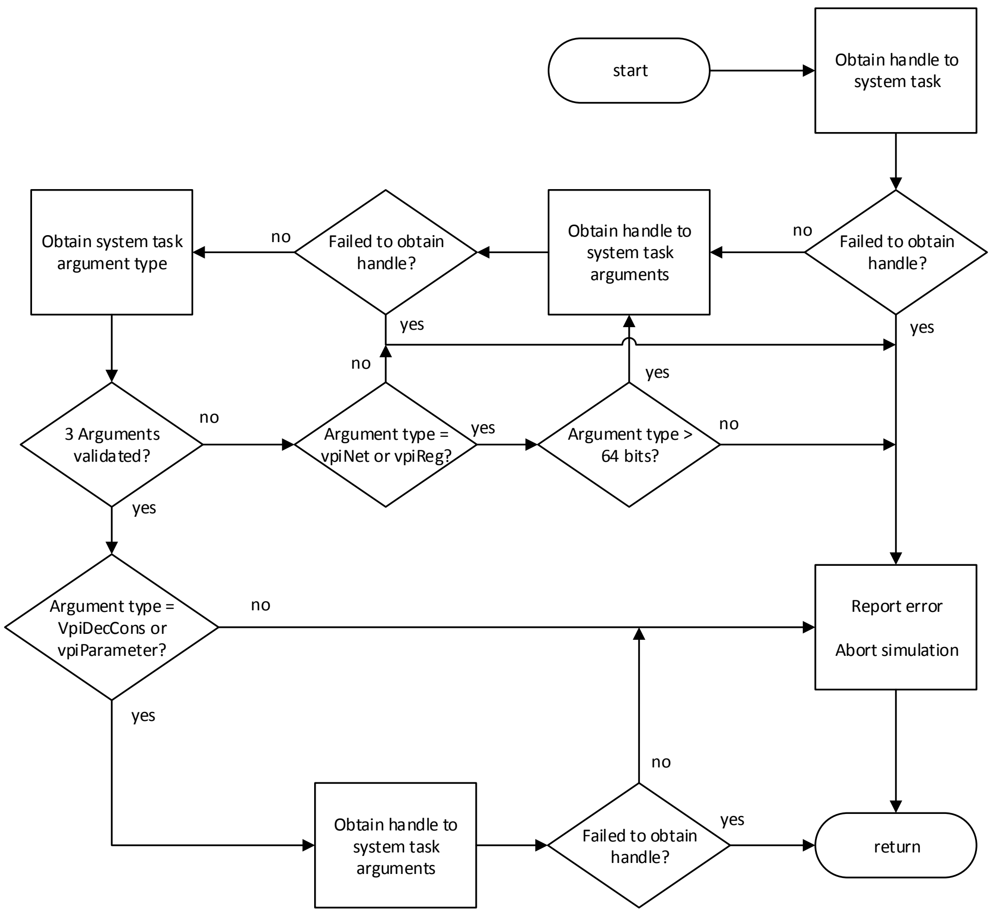

start
Obtain handle to system task
Failed to obtain handle?
no
yes
Obtain handle to system task arguments
Failed to obtain handle?
no
yes
Obtain system task argument type
3 Arguments validated?
no
yes
Argument type = vpiNet or vpiReg?
no
yes
Argument type > 64 bits?
yes
no
Argument type = VpiDecCons or vpiParameter?
no
yes
Report error
Abort simulation
Obtain handle to system task arguments
Failed to obtain handle?
no
yes
return

Figure F.11: $qemu_read_from_bus/write_to_bus compiletf flowchart

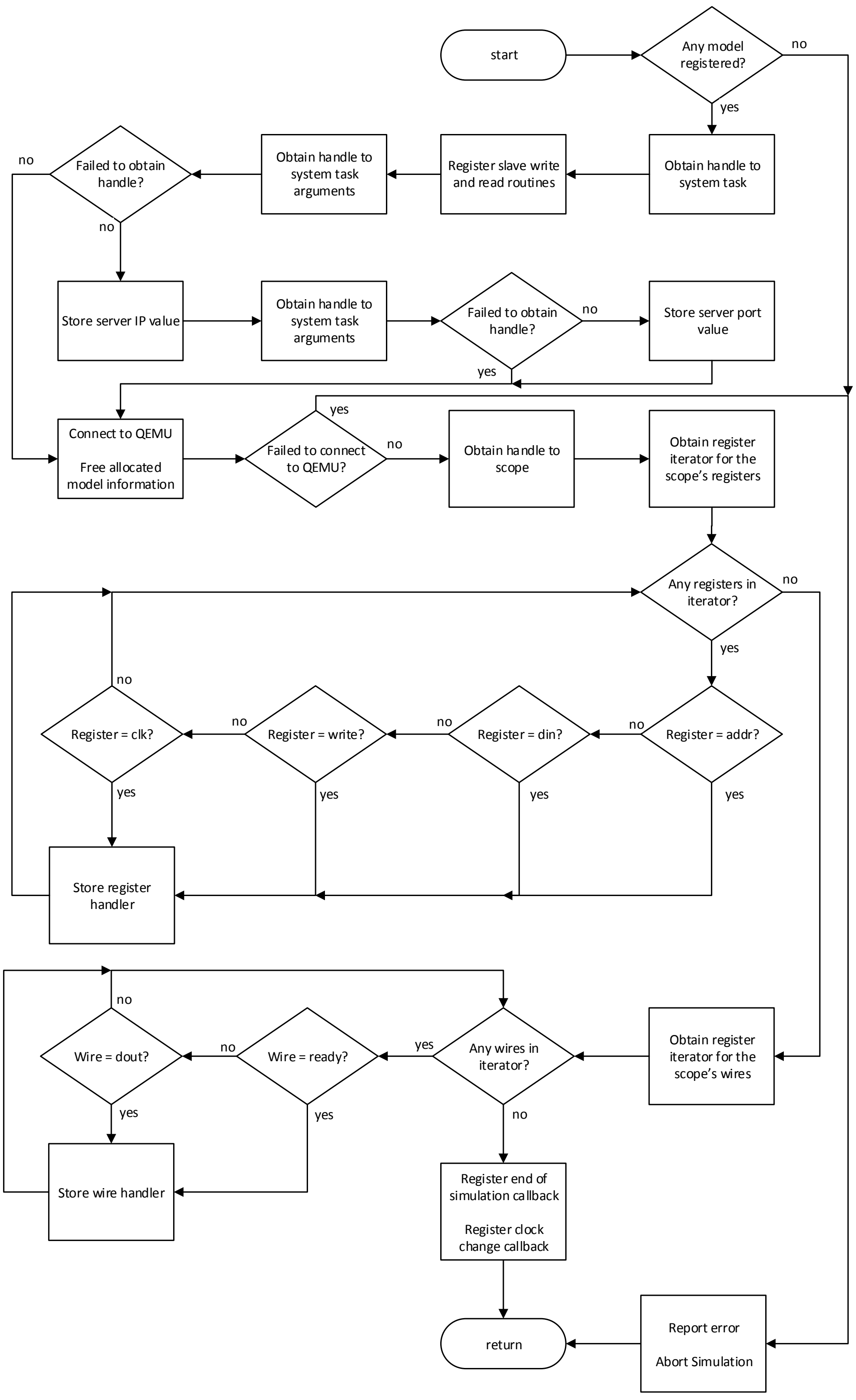


Figure F.12: $qemu_connect calltf flowchart

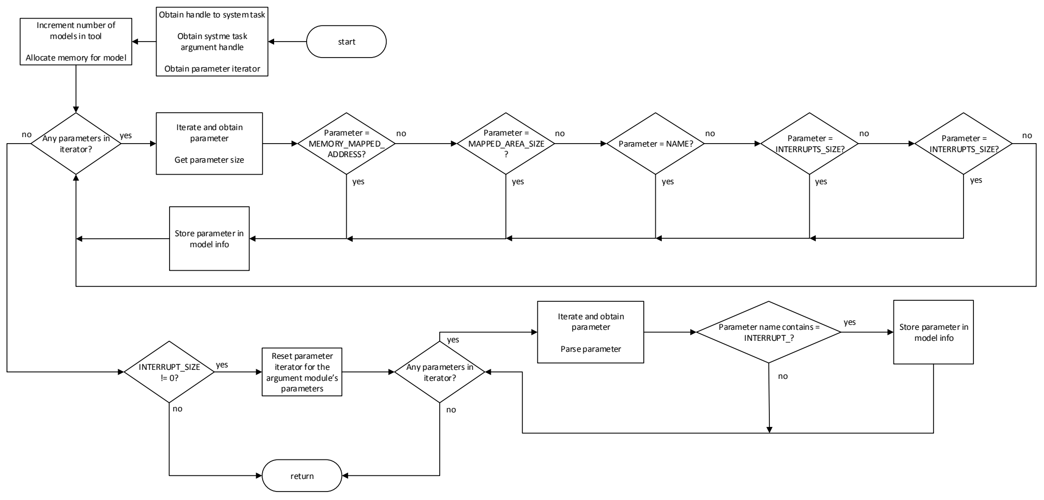

start
Obtain handle to system task
Obtain systme task argument handle
Obtain parameter iterator
Increment number of models in tool
Allocate memory for model
Any parameters in iterator?
no
yes
Iterate and obtain parameter
Get parameter size
Parameter = MEMORY_MAPPED_ ADDRESS?
no
yes
Parameter = MAPPED_AREA_SIZE ?
no
yes
Parameter = NAME?
no
yes
Parameter = INTERRUPTS_SIZE?
no
yes
Parameter = INTERRUPTS_SIZE?
no
yes
Store parameter in model info
INTERRUPT_SIZE != 0?
yes
no
Reset parameter iterator for the argument module's parameters
Any parameters in iterator?
yes
no
Iterate and obtain parameter
Parse parameter
Parameter name contains = INTERRUPT_?
yes
no
Store parameter in model info
return

Figure F.13: $qemu_register_model calltf flowchart

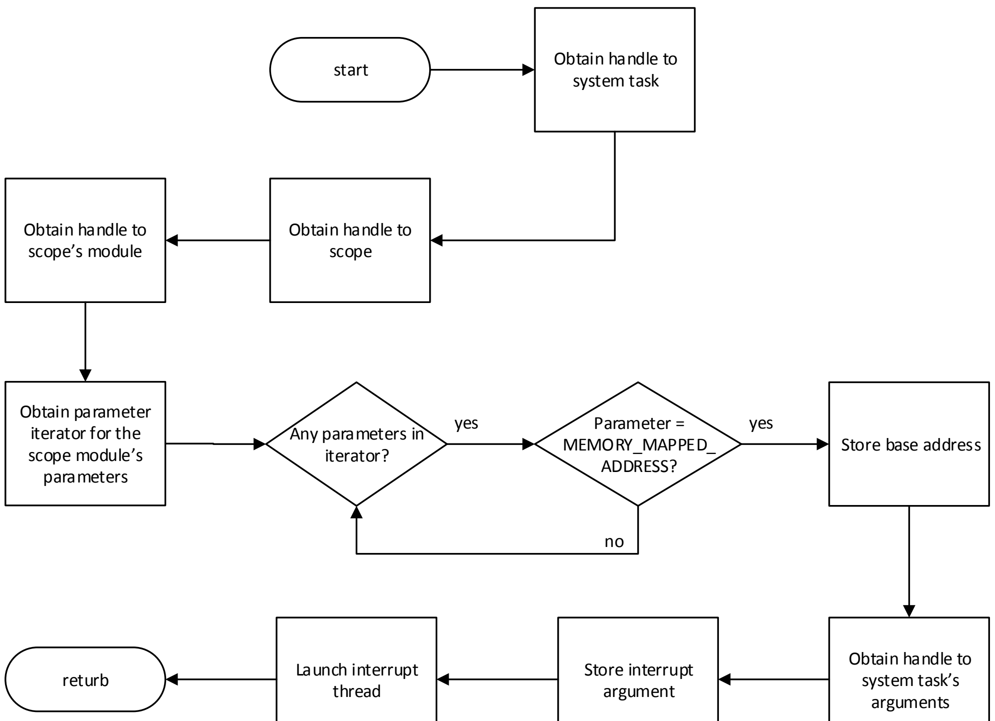

start
Obtain handle to system task
Obtain handle to scope
Obtain handle to scope's module
Obtain parameter iterator for the scope module's parameters
Any parameters in iterator?
yes
Parameter = MEMORY_MAPPED_ ADDRESS?
yes
no
Store base address
Obtain handle to system task's arguments
Store interrupt argument
Launch interrupt thread
returb

Figure F.14: $qemu_raise/lower_interrupt calltf flowchart

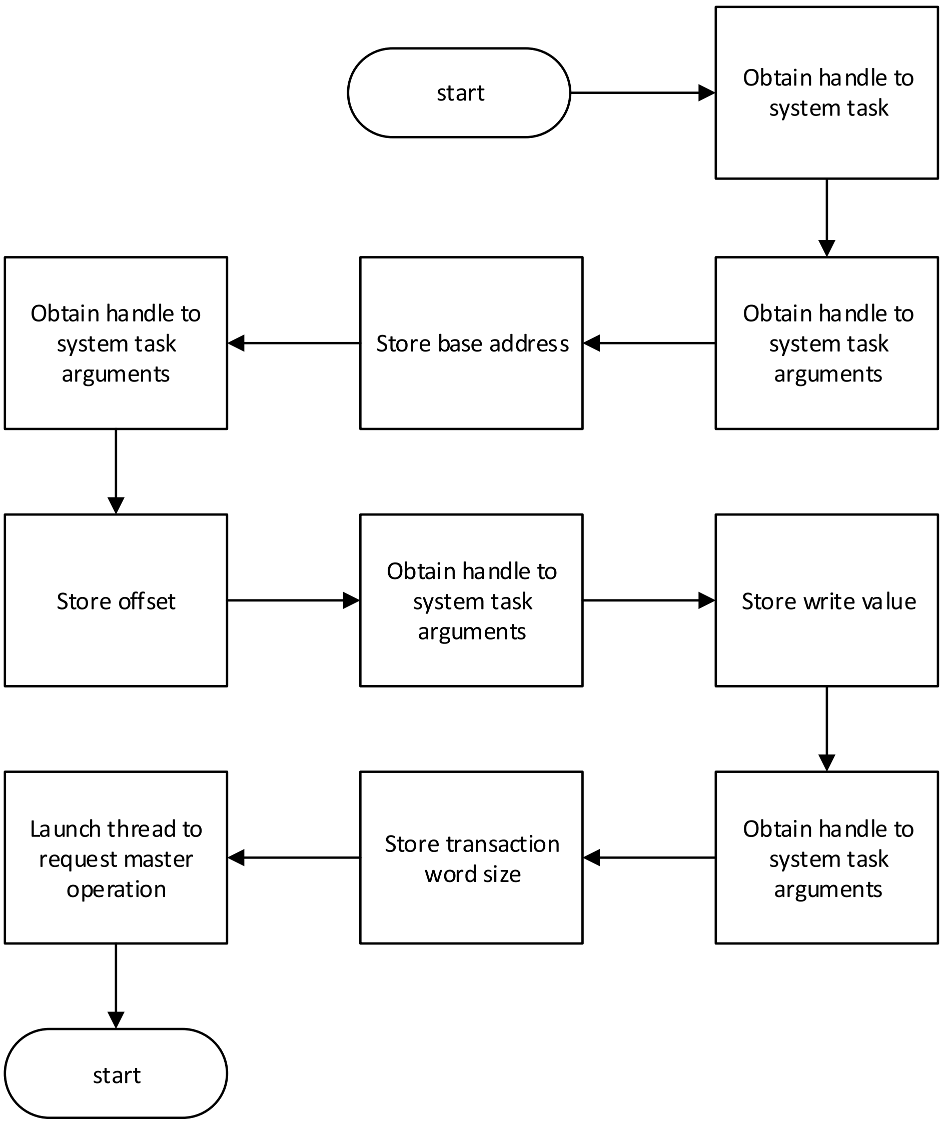

start
Obtain handle to system task
Obtain handle to system task arguments
Store base address
Obtain handle to system task arguments
Store offset
Obtain handle to system task arguments
Store write value
Obtain handle to system task arguments
Store transaction word size
Launch thread to request master operation
start

Figure F.15: $qemu_write_to_bus calltf flowchart

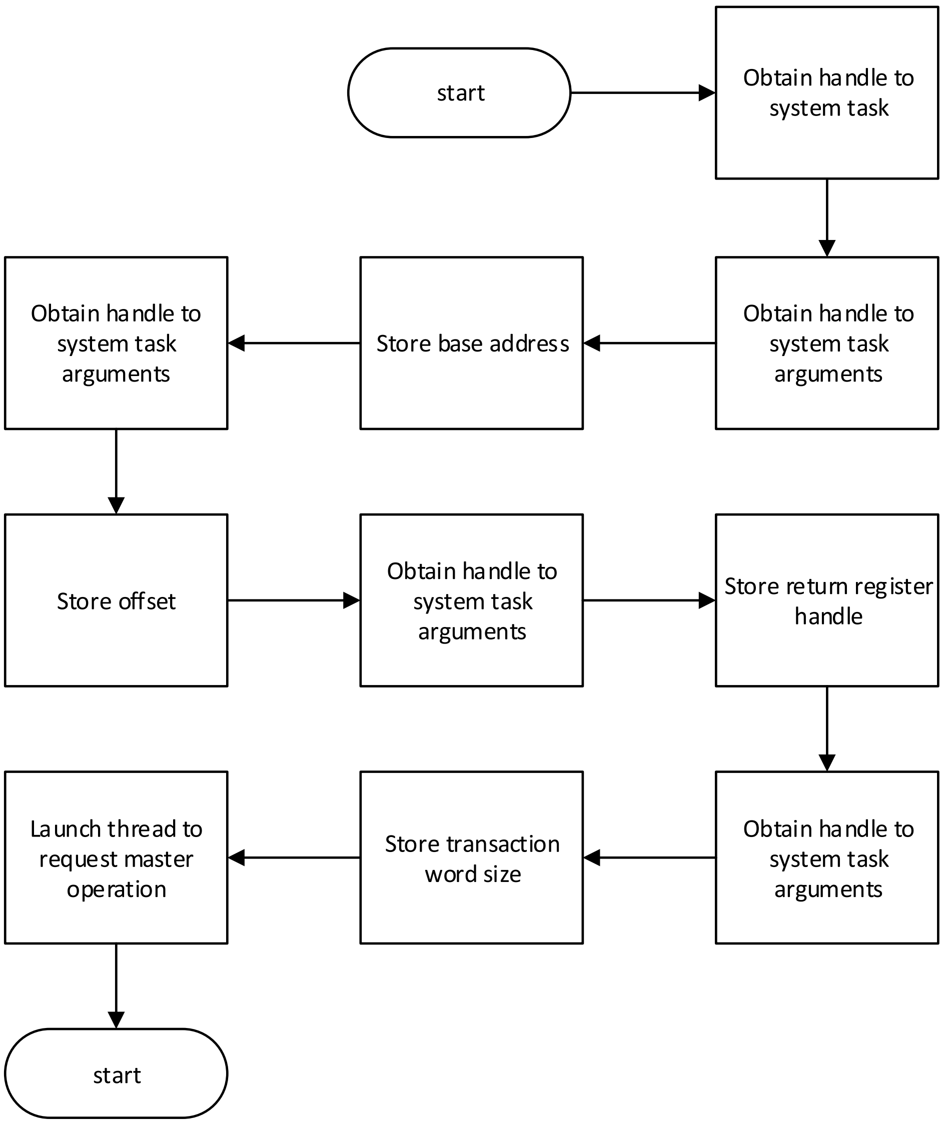

start
Obtain handle to system task
Obtain handle to system task arguments
Store base address
Obtain handle to system task arguments
Store offset
Obtain handle to system task arguments
Store return register handle
Obtain handle to system task arguments
Store transaction word size
Launch thread to request master operation
start

Figure F.16: $qemu_read_from_bus calltf flowchart

Appendix G

pq theory

The pq theory, also known as instantaneous real and imaginary power theory, was first proposed by Akagi et al. in 1993 (Akagi et al., 2007)

This theory is based on the Clarke transform, or α-β-*0* transform, which is a space vector transformation of time-domain signals (e.g. voltage, current, flux, etc) from a natural three-phase coordinate system (ABC) into a stationary two-phase reference frame (α-β-*0*). It is named after electrical engineer Edith Clarke.

The mathematical expressions used to perform the Clarke transform on currents ia, ib, ic and voltages va,vb,vc are the following:

$$\begin{bmatrix} v_0 \\ v_\alpha \\ v_\beta \end{bmatrix} = \sqrt{\frac{2}{3}} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} v_a \\ v_b \\ v_c \end{bmatrix} \tag{G.1}$$

$$\begin{bmatrix} i_0 \\ i_\alpha \\ i_\beta \end{bmatrix} = \sqrt{\frac{2}{3}} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix} \tag{G.2}$$

Once the Clarke transform is calculated, the instantaneous real power(p), imaginary power(q) and zero-sequence power(p0) may be calculated using both voltage and current transformed values using the following expression:

$$\begin{bmatrix} p_0 \\ p \\ q \end{bmatrix} = \sqrt{\frac{2}{3}} \begin{bmatrix} v_0 & 0 & 0 \\ 0 & v_\alpha & v_\beta \\ 0 & -v_\beta & v_\alpha \end{bmatrix} \begin{bmatrix} i_0 \\ i_\alpha \\ i_\beta \end{bmatrix} \tag{G.3}$$