Tampere University

Aisha Ahmed

Bringing up RISC-V Interrupts using Rust Programming Language

Faculty of Information Technology and Communication Sciences (ITC)
Master's of Science Thesis
Examiners: Prof. Timo Hämäläinen
Esko Pekkarinen, M.Sc.
April 2023

Abstract

Aisha Ahmed: Bringing up RISC-V Interrupts using Rust Programming Language
Master's of Science Thesis
Tampere University
Master's Degree Programme in Information Technology
April 2023

RISC-V ISA is a popular open and royalty-free ISA that enables the development of low-cost Systems-on-Chip (SoCs). Ballast is a novel multicore RISC-V-based SoC developed by SoC Hub in collaboration with different companies for Internet of Things (IoT) applications. It includes multiple subsystems that feature three RISC-V cores.

Interrupts allow processor power saving and fast interaction with external peripherals. Having proper software support for interrupts enables easy and accurate configuration. Additionally, safety is a concern, and having a language that guarantees safe operation, such as Rust, will minimize catastrophic bugs.

This thesis outlines the process of implementing interrupt software support in the Ballast SoC using the Rust programming language, as well as test techniques for ensuring proper interrupts operation. Software support was implemented as part of Hardware Abstraction Layer (HAL), and two use cases were developed to test the implemented software and validate interrupts on the Ballast SoC development board.

Interrupts were successfully brought up in Ballast SoC and were proven to function correctly using the test methodologies introduced in this thesis. Moreover, the developed software support can easily be utilized on other SoCs that use similar cores (such as the new SoC Hub chip Headsail) by including it as a dependency crate using Cargo tool.

Keywords: SoC, MPSoC, SoC-Hub, RISC-V, Rust, Interrupt.

The originality of this thesis has been checked using the Turnitin Originality Check service.

Preface

I would like to express sincere thanks to everyone who assisted me with my thesis. I am grateful to Prof. Timo Hämäläinen for giving me the opportunity to join SoC Hub and for guiding me through this thesis. I sincerely thankful to Esko Pekkarinen for his guidance and patience in reviewing this thesis several times. I am eternally thankful to Henri Lunnikivi for his endless support throughout this thesis, without which this thesis would not have been possible. My thanks extend to Tom Szymkowiak and Jarkko Passi for their consistent help with interrupts and Ballast hardware.

I thank my family back home for helping me achieve my goals, especially my father, without whom I would not even be here. Finally, I want to thank my husband for staying up late with me and supporting me all through this master.

Tampere, 25th April 2023,
Aisha Ahmed.

Contents

List of Figures

List of Tables

List of Abbreviations

MPSoC ..... Multiprocessor Systems-on-Chip

SoC ..... Systems-on-Chip.

IoT ..... Internet of Things.

HAL ..... Hardware Abstraction Layer.

ISA ..... Instruction Set Architecture.

CPU ..... Central Processing Unit.

CSRs ..... Control and Status Registers.

OS ..... Operating System.

CLINT ..... Core Local Interruptor

PLIC ..... Platform Local Interrupt Controllers.

PC ..... Program Counter

SysCtrl ..... System Control.

MPC ..... Medium-Performance Computing.

HPC ..... High-Performance Computing.

DSP ..... Digital Signal Processor.

AI ..... Artificial Intelligence.

C2C ..... Chip to Chip.

AXI ..... Advanced eXtensible Interface.

IC ..... Interconnect.

TLP ..... Top Level Peripheral.

LSP ..... Language Server Protocol.

VSC ..... Visual Studio Code.

SVD ..... System View Description.

PAC ..... Peripheral Access Crate.

ISR ..... Interrupt Service Routine.

OpenOCD ... Open On-chip Debugger.

GDB ..... GNU Debugger.

1 Introduction

Interrupts are asynchronous signals that intervent processor's current tasks to respond to an external event. They are a critical part of embedded systems as they save processor time when interacting with other peripherals. They also reduce power consumption as they allow processors to go to sleep and only operate when needed. Additionally, they play an essential part in real-time systems and meeting system deadlines. Having proper software support for interrupts will ensure proper interrupt configuration and use.

One of the popular hardware architectures in embedded systems is RISC-V ISA. RISC-V is an open standard for instruction set architecture (ISA). It is an open and royalty-free ISA that enables the development of low-cost System-on-Chip (SoC). RISC-V is also modular and flexible, with a base + extension combination that allows for customization based on needs. Because of its lack of royalty and modularity, the popularity of RISC-V has increased in recent years in both academia and industry [1].

C and C++ are the most common programming languages in embedded systems development. However, these languages have been known for being unsafe, especially when dealing with memory operations. Problems such as use-after-free and null pointer dereferencing are common and can cause high-cost, life-threatening bugs [2]. On the other hand, Rust programming language is known for its safety. It is a relatively new language, invented by Graydon Hoare, a Mozilla employee, in 2006. Rust focuses on safety and productivity. Safety is achieved by having an affine type system, ownership system rules, and a borrow checker. Additionally, the Rust compiler, rustc, statically checks for ownership system violations and generates errors whenever it detects one. All of these elements govern memory interactions and make it difficult for developers to make mistakes that lead to memory violations. Finally, since Rust is community-driven, it has a wide range of useful tools that help in developing software in general and for embedded systems [3].

The goal of this thesis is to bring up interrupts in the novel multiprocessor platform Ballast, developed by System-on-Chip (SoC) Hub. SoC Hub is a cooperation between industrial partners and Tampere University to establish a SoC design ecosystem in Finland [4]. Ballast is the first SoC designed by the SoC hub, and it consists of several subsystems with several RISC-V cores. Additionally, another goal of this thesis is to explore the use of Rust as a development programming lan-

guage to test its potential in producing safe embedded systems software. However, bringing up interrupts is a complex process. The core registers and peripherals must be correctly accessed for configuration. Moreover, a vector table must be correctly created to enable the use of interrupt vectored mode, and interrupt handlers must be registered to their designated interrupt source ID number. Finally, the implemented support must be easy to debug to verify interrupts operation.

This thesis documents the implementation of software support for three of Ballast's subsystems: Sysctrl, MPC, and HPC, using Rust as a programming language. The software support provides the means to configure the subsystem's interrupts and interrupt controllers while ensuring safety. The implemented software was tested on Granitti board –Ballast development board- using both printed output and software debuggers.

This thesis is divided into seven chapters. Chapter 2 introduces the RISC-V ISA and its interrupt handling process. It also provides background information about interrupt controllers available for RISC-V, and introduces some available SDKs for configuring interrupts in RISC-V. Chapter 3 presents Ballast and its development board, Granitti, and describes the interrupt system on this platform. In Chapter 4, Rust is introduced with brief explanations of its ownership model and the tools it provides for software development. Chapter 5 outlines the steps used to implement interrupts in Ballast and presents two use cases of how interrupt support has been implemented in Rust and C. Chapter 6 shows the results of the implementation process and discusses the findings. Finally, Chapter 7 summarizes the content of this thesis and gives recommendations for future work.

2 RISC-V and Interrupts

RISC-V is a open standard instruction set architecture (ISA) developed by Prof. Krste Asanović and graduate students Yunsup Lee and Andrew Waterman in University of California, Berkeley in USA as a part of Par Lab (Parallel Computing Laboratory) with Prof. David Patterson as a director[1].

RISC-V, the only open and royalty free ISA for now, allows development of System-on-Chip (SoC) at low costs. However, this is not the only good reason for using RISC-V ISA. RISC-V is designed to be modular and customizable by having an extensible base ISA [5]. This base contains a minimal and basic set of instructions that are needed by every RISC-V chip denoted as RV32 (for 32 bits systems), RV64(for 64 bits systems) and RV128(for 128 bits systems). On the other side, extensions are defined to allow for example integer and multiply/divide operations. Each base or extension ISA is distinguished by a letter prefixed to RV32/RV64/RV128, for example RV32IMF where the "I" stands for the Integer ISA, "M" for Multiplcation/Division extension and "F" for floating points operations. Finally, to provide more flexibility, RISC-V allow for non-standard extension for highly specialized systems [6].The flexibility and modularity that RISC-V provides along with low costs development, increased RISC-V popularity among both academia and industry. It is estimated that number of produced RISC-V AI SoCs will be 25 billion by 2027 [7].

This chapter will discuss RISC-V and RISC-V interrupts while providing background information for the thesis.

2.1 Interrupts

An interrupt is an event that occurs when the CPU (Central Processing Unit) instruction execution's sequence is changed due to external sources such as peripherals or internally when an exception is raised.

For each interrupt type there is an interrupt handler designated to it, in which the CPU executes set of instructions. This handler is written by a programmer to perform certain tasks. After the CPU exits the handler, it restores its initial state and continues what was interrupted.

Interrupts are used when communication with an interrupt source is only needed occasionally. Hence, instead of infinitely waiting for information, the CPU is only

interrupted when the information is ready. Interrupts are extremely useful that nearly every processor and microprocessor is using them; they allow to efficiently use the CPU time and reduce power consumption.

In RISC-V cores there are two types of events that cause the CPU to transfer control to a handler:

1. Interrupts: these are external asynchronous events.
2. Exceptions: internally synchronous event that is caused by erroneous state in the CPU.

Additionally, there are three types of interrupts that occur locally to core and can be controlled through Control and Status Registers (CSRs) [8]:

1. Timer interrupts.
2. Software interrupts.
3. External interrupts.

2.2 Privilege Levels

According to the RISC-V standard, privileged levels are those where control and access to system resources are limited based on the level. For instance, a user software with low privilege level will have limited control over the system's memory. Any operation outside privilege level boundaries will cause an exception to be triggered.

There are three privilege levels in RISC-V specification that are encoded in CSRs, table 2.1 shows how privilege levels are encoded:

1. Machine level: the highest privilege level with full access to all resources and the only level required by the specification. Software running at this privilege level should be secure and trusted.
2. Supervisor level: intended for operating systems and low-level firmware.
3. User level: intended for other high level user applications.

Additionally, there is a fourth level defined by an extension to privilege RISC-V specification called Hypervisor level or mode. It is an extended supervisor level with capabilities for virtualization I.e., to run an OS on top of Operating System (OS) [8].

Table 2.1 *Privilege level enconding in RISC-V specification [8].*

Level	Encodeing	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved	
3	11	Machine	M

2.3 Interrupts Controllers in RISC-V

Interrupts control in RISC-V cores can be classified into two categories: Local control and global control.

The core or core controller performs local control and provides low latency interrupt handling while operating at core clock frequency. However, they are only limited to timer and software interrupts.

CLINT (Core Local Interruptor) is SiFive famous implementation. It was implemented before the specification was released and was widely used as a golden model for local interrupt control. CLINT is a fixed priority local interrupt controller that only allows preemption from higher levels of privileges. This typically means only machine privilege level can interrupt lower levels (such as Supervisor and User privilege levels). It has two modes of operations: direct and vectored.

In direct mode the software is responsible for directing each interrupt or exception to its corresponding handler whereas in vectored mode, vector table contains memory addresses to respective handlers in predefined offset that depend on interrupt ID. Generally, the formula is: vector table address+(4*interrupt ID), for instance, the timer interrupts ID is 7 so its handler addresses is vector table address+ 0x1C [9].

In contrast, global controllers or Platform Local interrupt Controllers (PLICs) control the interrupts on platform level, i.e., across multiple cores and/or peripherals. They are external components (to the core) operating with a different clock and route all global interrupts to the core through a single interface line with a specific interrupt ID. PLICs provide programmable priority interrupt and a threshold register to mask all interrupt below a certain programmable level.

PLIC has a set of five memory mapped registers: interrupt priority, interrupt pending, interrupt enable, threshold and claim/complete register. For each interrupt source, priority is set through interrupt priority register with higher number

having higher priority. Additionally, each core can control which interrupt to handle through interrupt enable register, and it can also set a threshold upon which interrupts with certain priority are served.

For the core to handle an interrupt, interrupt source must be enabled and have a priority set. Then upon interrupt arrival to PLIC, the interrupt is claimed by reading claim/complete register which returns interrupt ID of the highest priority interrupt in pending register. Finally, after exiting the interrupt handler, the interrupt ID is written to claim/complete to notify PLIC that the interrupt has been served [10].

2.4 Interrupt Configuration Registers

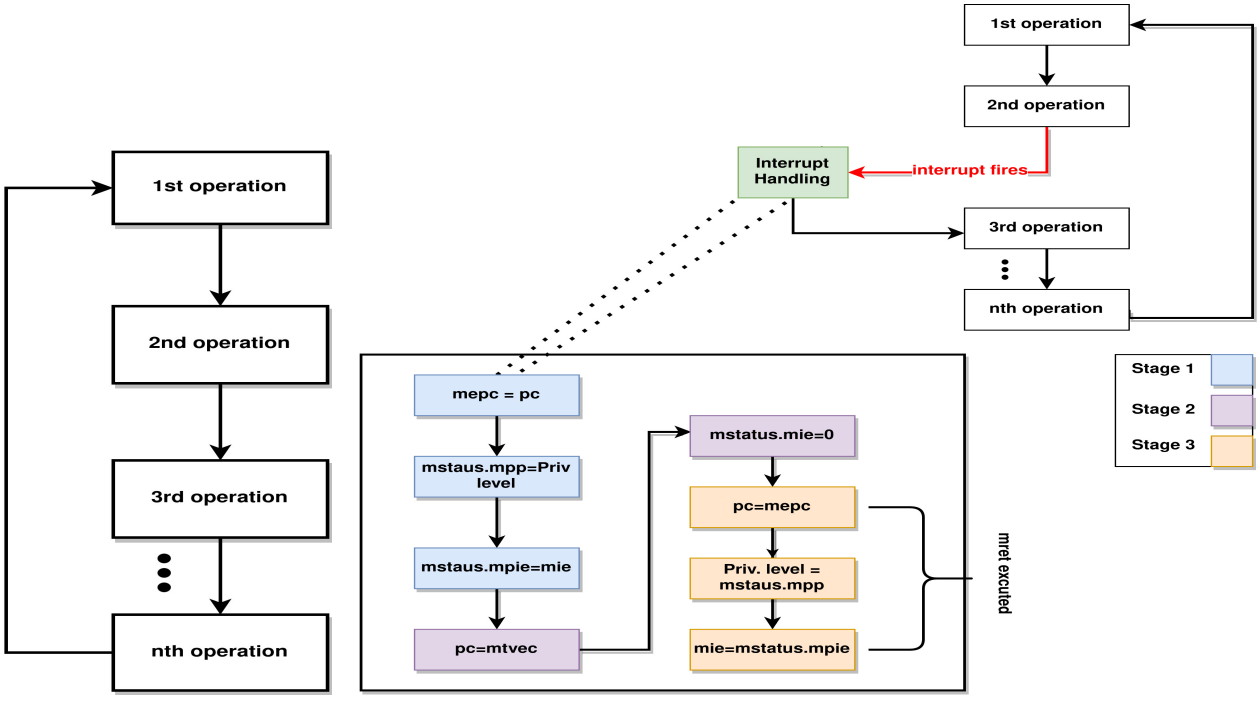
Control and Status Registers (CSRs) of RISC-V are multiple registers for configuring interrupts. CSRs are in the core and can only be accessed by the core itself, thus configuring core local interrupts. They can be read and written to through csrr (CSR read) and csrw (CSR write) instructions, respectively. There are equivalents for these registers for each privilege level, here addressed only the Machine privilege level CSRs.

- *mstatus* : Machine mode status register used for tracking the CPU current operating state and enabling global interrupt through MIE bit field.
- *mcause* : Machine mode cause register used for recording the cause of the interrupt/exception.
- *mie* : Machine mode interrupt enable used for enabling specific interrupts such as enabling software interrupt by setting MSIE bit.
- *mip* : Machine mode interrupt pending register, holds information about pending interrupts that wait to be served.
- *mtvec* : Machine mode trap vector register contains the base address of interrupt vector table and interrupt mode (direct or vectored).
- *mepc* : Machine mode program counter register, used to hold the address of the interrupted instruction i.e., instruction that was running before an interrupt/exception happens [8].

2.5 Standard Basic Interrupt Handling Scheme

Generally, the program executes instructions sequentially as shown in figure 2.1(a). When Interrupt occur, the process of handling the interrupt in RISC-V core go through the following four stages illustrated in figure 2.1(b) [9]:

1. Interrupt/exception is triggered: this is the moment that the core receives an interrupt or raises an exception.
2. Context saving: in this stage, the core prepares itself to switch control by saving:
   (a) Program counter(pc) to mepc.
   (b) Privilege level to mpp bit in mstatus.
   (c) mie to mpie bit in mstatus.
3. Context switching: the control is switched by setting pc using the interrupt handler address in mtvec. After that disable global interrupts by clearing mie bit in mstatus.
4. Return control: this is the final stage where the control of the program is restored to its original state by mret which does the following:
   (a) Load pc from mepc.
   (b) Restore the previous privilege level.
   (c) Restore mie from mstaus.



(a) Program Flow without Interrupts.

(b) Program Flow with Interrupts.

Figure 2.1 *Program Flow with and without Interrupts*

2.6 Available SDKs for RISC-V Interrupts

Pulp SDK provides fundamental libraries, interrupt support, and tools for PULP chips used in Sysctrl and MPC subsystems. It was mainly developed using C programming language and built using Makefiles and scripts. GCC is used as a compiler, and function attributes are utilized to mark functions as interrupt handlers, such as *void __attribute__((interrupt)) __rt_timer_handler()*. However, since this SDK uses C, it is exposed to unsafe features of the language, such as wrong memory access, and puts the responsibility on the developer to achieve safe operation. Additionally, the build process has many steps and needs to separately download and build the RISC-V toolchain, which complicates the build process [11].

Freedom metal is part of SiFive Freedom SDK developed by SiFive company. It enables bare metal development on RISC-V IP designed by SiFive and provides APIs for controlling the CPU. It is implemented using C and defines certain function signatures for interrupt handlers, such as *typedef void (*metal_interrupt_handler_t)(int, void*)*. It also provides inline functions to enable and register interrupt handlers. Similarly to Pulp SDK, Freedom Metal trades safety with C [12].

cortex_m_rt crate is a Rust-implemented crate that provides safe interfaces to populate the vector table and dispatch interrupts properly. It also provides the [interrupt] attribute to define functions as interrupt handlers. However, this crate is implemented for cortex-m microcontroller only [13].

None of the aforementioned SDKs were suitable for bringing up interrupts in Ballast SoC using Rust; therefore, a custom implementation was needed. To develop software that supports interrupts in Ballast, the Ballast interrupt system must be studied.

3 Interrupt System in Ballast MPSoc

This chapter introduces the Ballast MPSoC architecture and discusses the topology of interrupt systems generally and in Ballast.

3.1 Ballast MPSoc Architecture

The Ballast platform consists of multiple subsystems with three different RISC-V cores. A System Control subsystem (SysCtrl) with an IBEXRV32EC RISC-V core, a Medium Performance Computing (MPC) holding a 32-bit RISCY core, and finally a High-Performance Computing (HPC) subsystem with two 64-bit CVA6 RISC-V cores. SysCtrl subsystem is intended to perform limited tasks such as booting the whole chip. HPC and MPC are intended for more complex applications, with HPC being the most capable [14],[15]. Additionally, other subsystems include a custom Digital Signal Processor (DSP) subsystem, an Artificial Intelligence (AI) subsystem that serves as Deep-learning accelerator, an Ethernet subsystem,Chip to Chip subsystem (C2C) which allows off-chip communications through Advanced eXtensible Interface (AXI) interface, an Interconnect subsystem(IC) for on-chip communications, and Top Level Peripheral (TLP) subsystem that is responsible of provide access to global peripherals and routing interrupts across different subsystems.

Figure 3.1 shows Ballast MPSoc hosted on a development board Granitti. which is a board that was custom designed by SoC Hub to enable working with and debugging different systems in Ballast.

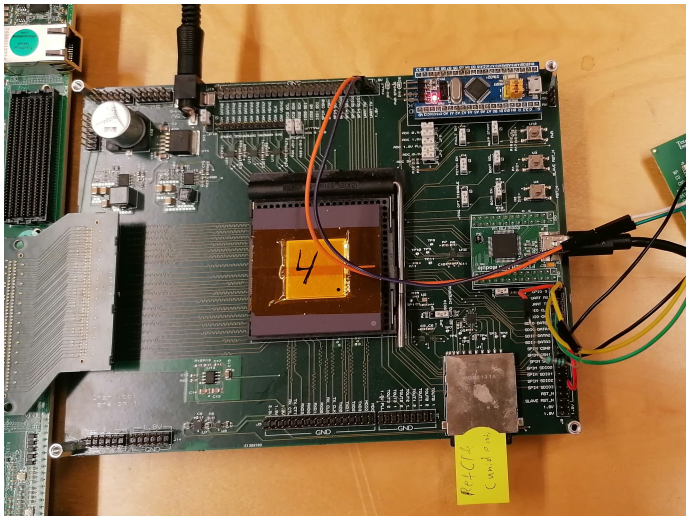


Figure 3.1 *Granitti board*

3.2 Interrupt System Topology

Interrupt system topology is vital for correct and efficient interrupt functionality. By knowing how interrupt sources are connected to the cores and how these cores interrupt each other, interrupts can be configured accurately. Usually, this is done by having memory mapped registers to configure cores interrupt sources and interrupt controllers and provide a mechanism of communication between hardware and software layer. Different topologies exist when connecting interrupt sources to the core or between cores. One example is having only CLINT to control all interrupts.

Another design for interrupt system shown in Figure is having CLINT to handle both software and timer interrupts and PLIC to arbitrate external interrupt. The advantage of this topology is it allows several external interrupt sources to be connected to the core through PLIC. In Figure 3.2 PLIC is connected to each core via external interrupt line and each core has an independent CLINT to control both software and timer interrupts.

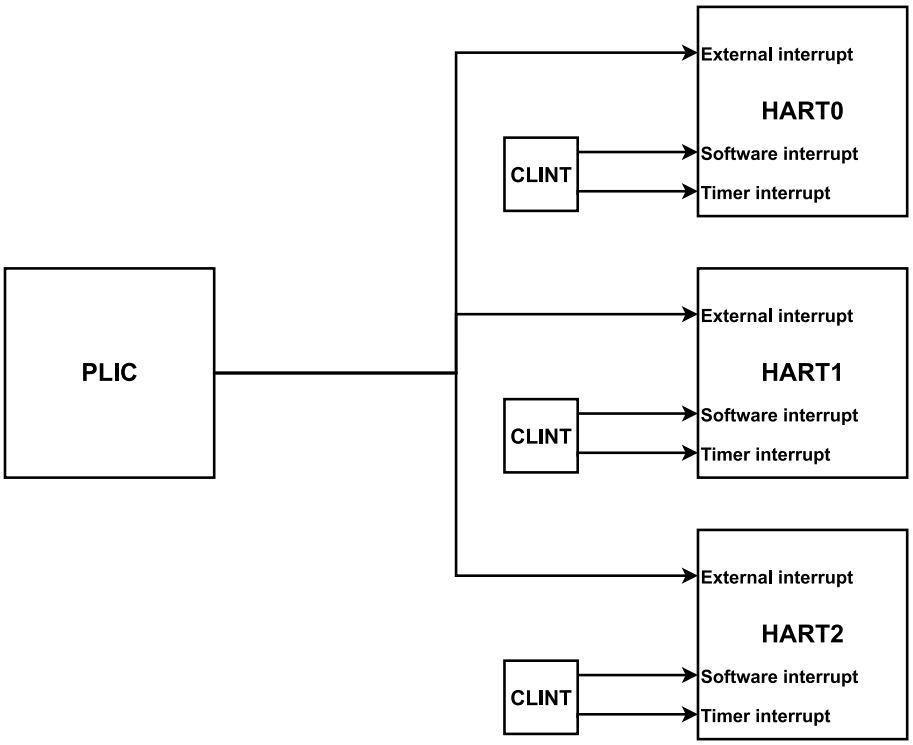


Figure 3.2 *PLIC with multiple CLINTs topology*

3.3 Ballast Interrupt Topology

SysCtrl and MPC local interrupts are handled using CLINT and external interrupts through a custom unit called Event/Interrupt unit. On the other hand, HPC utilizes CLINT and PLIC for both cores through AXI interconnect.. Figure 3.3 illustrates the topology of interrupt system in SysCtrl, MPC and HPC. In MPC and SysCtrl, cores interact with CLINT through memory-mapped registers using memory interconnect. Subsequently, HPC cores configure PLIC and CLINT using memory-mapped registers through common AXI interconnect

The Event/Interrupt Controller contains memory mapped registers and supports up to 32 lines vectorized interrupts/events and contains a FIFO for peripherals and software events. The communication between the core and Event/Interrupt Controller is totally asynchronous and depends on handshaking signals. When an interrupt is triggered the controller sends the interrupt ID to the core, if the core accept the interrupt, it acknowledges the controller using the same interrupt ID. To configure the interrupts in this controller, interrupts are enabled by setting bits in MASK register that corresponds to the interrupt line IDs. In addition to the Event/Interrupt unit, the platform contains SoC Event generator unit. This unit is responsible for routing the events from peripherals connected to an uDMA to Event/Interrupt unit. It also routes the interrupts between the peripherals, for example, if an interrupt was triggered from external timer peripheral it will trigger I2C peripheral [14],[15].

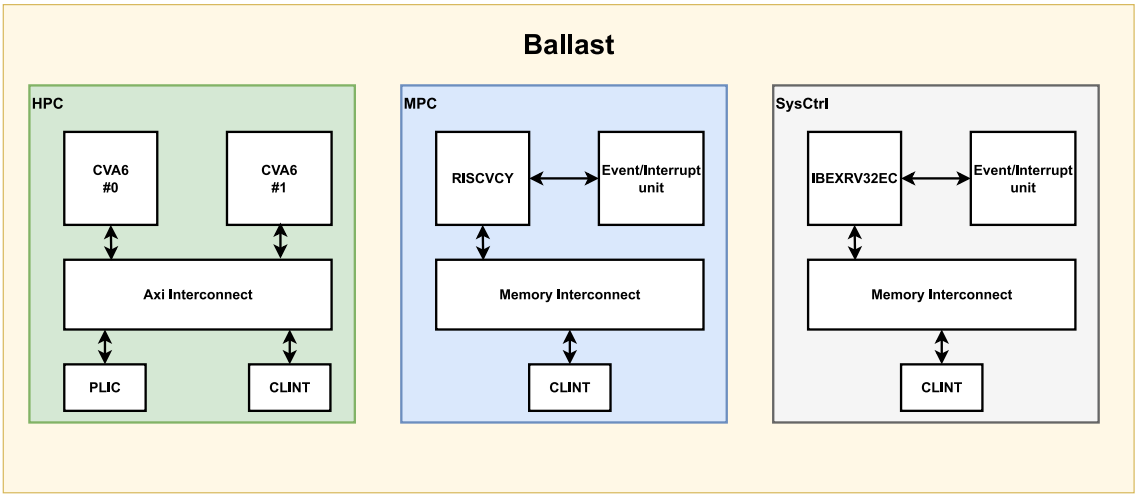


Figure 3.3 *Interrupt System Topology in Ballast*

In summary, the interrupt system in Ballast SoC included a combination of CLINT and Event/Interrupt unit for both SysCtrl and MPC and CLINT and PLIC

for HPC. These controllers can be accessed and configured using memory-mapped registers and a suitable programming language.

4 Rust Programming Language

Rust is a programming language invented by Mozilla employee Graydon Hoare in 2006 and adopted by Mozilla in 2009. It is widely adopted in companies such as Facebook, Google, and Dropbox. Rust has increased popularity in recent years and became the most loveable programming language among developers in 2022 [16].

Rust is designed with an emphasis on memory safety, performance, and productivity, especially for low-level programming systems like embedded systems software. It does not have any garbage collector like other programming languages such as C. Instead, it has an ownership model and automatic referencing that statically guarantee safe operation [3]. Another feature of Rust is its design for concurrency and parallelism; therefore, it can be used in a multi core system without having race conditions. In C for example it is very hard to avoid such cases.

Finally, although Rust is mainly maintained by Mozilla, Rust has a wealth of libraries due to its supportive and active community. This also helps in improving the language by encouraging developers to contribute to Rust GitHub mainstream.

4.1 Ownership Model and Safety

Ownership model is Rust unique feature that ensure memory safety. It is a set of rules checked by the compiler that Rust program must follow when using memory. This ownership model has 3 main rules it poses on the developer to ensure memory safety:

1. Every value has an owner, which is a variable.
2. Only one owner is allowed at a time.
3. When the owner goes out of scoop, the value will be released.

Every variable has a scope. That is, determining when the variable is valid and has been allocated memory and when it is invalid and has been freed from memory. For example, Listing 4.1 shows a variable x having a value of 10 and surrounded by brackets. Inside the brackets, x is valid; once the program goes beyond the closing bracket, x is freed and no longer exists. Attempting to use x again will cause a compiler error. Similarly, when dealing with variables that dynamically grow or change in runtime, memory is deallocated automatically when the variable leaves the scope. With this system, Rust manages memory without the need for a garbage collector like in C.

```
//x is invalid
{
    let x = 10;  // x is valid
}
// x is invalid
```

Program 4.1 *Rust program showing where variable scope is valid*

In the previous example, the data was owned by x. That means only x can change this data, and any other variable or function that wants this data must own the data by moving or taking a reference of x. Listing 4.2 shows the value is x moved to y, and any attempt to use x again will raise a compiler error.

```
Let x =10;
Let y= x;

Println!(x); // compiler error will be generated
```

Program 4.2 *Rust program showing variable data moving*

This leads to referencing and borrowing concepts in Rust. A reference to a variable is basically a pointer to the data held by that variable, with the guarantee that it points to a valid address. Borrowing in Rust happens by creating a reference of the variable to be used in other sections of the program. For example, Listing 4.3 shows a variable x with a value of 10 being borrowed by variable y. Now y points to the value 10.

```
let x = 10;
Let y = &x;
```

Program 4.3 *Rust program showing reference borrowing*

Similarly, when dealing with functions, a function can either own variables or take a reference of variables. This is defined in the function declaration: if a function takes a reference as input (indicated by input:&type in the function the function declaration), the referenced data will not be destroyed after leaving the function scope, instead the ownership is returned to original owner of the data. On the contrary, if a function takes ownership of data (indicated by input:type), the data is destroyed once it leaves the scope. Listing 4.4 illustrates this: the first string containing ”hello” is then passed to the foo1 function, which takes a reference of the string. After foo leaves, scope s can be used and printed. However, when s is passed to foo2 function, which takes ownership of s’s data, s cannot be used again as it moved the ownership of its data to foo2 and the data got freed after leaving

the scope.

```
Fn main(){
let s = String::from("hello");
foo1(&s);
Println!(s);
foo2(s);
Println!(s); // cause a compiler error
}

Foo1(string: & String){
//does something
}

Foo2(string: String){
//does something
}
```

Program 4.4 *Rust program showing ownership using functions*

That said, if the data type is copyable, that is, if it implements the copy trait, the previous example would work. For example, if s is replaced with an i32 integer, the data will not be moved into foo2, instead it will get copied, and using the integer after foo2 will be possible. In Rust, all integer types such as i32 and u32, Boolean types, floating-point types, and char types can be copied. References can be either immutable or mutable. Immutable references allow the borrower to only read the data and do not allow making changes to the data. In contrast, mutable allows both reading and changing data and is indicated by &mut. Mutable references are always unique to prevent multiple sources writing to the same memory location and ensure data integrity. On the other hand, immutable references can be multiple. Listing 4.5 gives an example of immutable and mutable references. Function foo1 takes an immutable reference of s string, and foo2 takes a mutable one. Any attempts from foo1 to change referenced data will cause a compiler error.

```
fn main() {
    let mut s = String::from("hello");
    foo1(&mut s);
}

fn foo1(string:  &String) {
   Println!(s); //this is allowed
    some_string.push_str(", world"); // this will cause error
}
```

```
fn foo2(string:  &mut String) {
   Println!(s); //this is allowed
    some_string.push_str(", world"); // this is also allowed
}
```

Program 4.5 *Rust program showing immutable and mutable references*

Rust uses an affine type system to manage data ownership and a borrow checker to enforce ownership rules. Affine types are types used once at most. Data are always uniquely owned, and ownership can only be borrowed or transferred. This will prevent multiple sources from changing the same data (i.e., race conditions). This makes the data highly trustworthy and safe to use. Additionally, defining scopes for variables so that when they go out of scope they are automatically destroyed eliminates use-after-free bugs, which are the cause of a large number of problems such as sabotaging other programs data. Furthermore, Rust allows only one mutable reference and multiple immutable ones. This ecosystem created by Rust secures memory, prevents memory violation and race conditions, and ensures that developers will not unintentionally create memory bugs by generating compile-time errors when these rules are breached.

That said, Rust has a hidden language called unsafe Rust. It is Rust without enforcement of memory safety guarantees. Even though this might seem concerning, it allows more control, especially when working with low-level systems such as embedded systems. It exists because compiler might reject operation that seems to be unsafe when statically analysis the code. It needs extra information to allow such operations, hence developers use the unsafe keyword to mark parts of code as ”guaranteed to be safe by the developer”. This keyword enables operations such as dereferencing a raw pointer, modifying mutable static variables, and using unsafe methods that, for example, write data on hardware registers.

It should be mentioned that borrow checker is not disabled when using unsafe Rust. Unsafe Rust only allows certain operations, such as those previously mentioned, to be executed. Also, if any runtime errors happened when using unsafe Rust, they will easy to trace back and the developer needs to check unsafe blocks in their code making debugging less tedious [3].

4.2 Rust Tools

Rust has many tools that ease software development. These tools can be categorized as build and management tools such as cargo, linter and code formatting

tools such as clippy, rustfmt and rust-analyzer and special tools such as svd2rust which is used in embedded development to create PAC.

4.2.1 Cargo

Cargo is an integrated tool for building applications and for package and dependency management in Rust. It automatically invokes the compiler when users build applications by using the cargo build command and gives the user the ability to customize builds in the configuration file cargo.toml by adding features that are passed to cargo build command. Figure 4.1 illustrates how dependencies are declared in cargo.toml; each package has a version and source defined. Figure 4.2 shows a snippet of cargo.toml file. It shows how the features ballast-sysctrl, ballast-mpc, and ballast-hpc are added to a project. To build for Sysctrl, for example, *cargo build –target=ballast-sysctrl* will be used.

Also, it fetches packages and builds them using information in cargo.toml and auto-generates cargo.lock, which locks the current build dependencies to allow repeatable builds. Finally, cargo allows Rust projects to be managed easily and effectively, which in turn helps increase the productivity of the working team[17].

```
[dependencies]
soc-hub-common = {path = "../soc-hub-common"}
metapac = {workspace = true}
embedded-hal = {workspace = true}
nb = {workspace = true}
ufmt = {workspace = true, optional = true}
ufmt-write = {workspace = true}
paste = {workspace = true}
doc_item = "0.3.0"
bitmask-enum = {workspace = true}
void = {version = "1.0.2", default-features = false}
```

Figure 4.1 *Dependencies management in cargo.toml*

4.2.2 Clippy, Rust-analyzer and Rustfmt

Clippy is a lint tool that is designed to catch commonly made mistakes and give suggestions for fixing them. It is automatically downloaded and used with cargo through the cargo clippy command and has several categories and lint levels, such as correctness and style with deny and warn levels, respectively. This means whenever there is a wrong or useless code, Clippy will deny compiling it. Similarly,

```
[features]
# Use this feature on SysCtrl
ballast-sysctrl = ["ballast", "metapac/ballast-sysctrl"]
# Use this feature on MPC
ballast-mpc = ["ballast", "metapac/ballast-mpc"]
# Use this feature on HPC
ballast-hpc = ["ballast", "metapac/ballast-hpc"]
```

Figure 4.2 *Features in cargo.toml*

whenever there is a style violation, it will cause a warning. This tool helps create cleaner code in terms of mistakes and style [18].

On the other hand, rust-analyzer is an open-source language server protocol (LSP) for Rust that performs semantic analysis of code while it changes. It enables features like auto-completion, go to definitions, type documentation over hover, inlay hints, and code actions. Inlay hints include types of local variables and types of chained expressions. Additionally, code actions are small fixes suggested by rust-analyzer, like adding braces to the match arm expression, and are triggered by pressing the light pulp icon in the editor. Rust-analyzer can be used as a plug-in with code editors such as Visual Studio Code (VSC)[19]. Finally, rustfmt is a formatting tool that is used to automatically format code written in rust to follow community code style. It is installed when Rust is installed [20].

4.2.3 Svd2rust

Svd2rust is a tool used to convert System View Description (SVD) files into crates with type-safe APIs to interact with hardware peripherals. It needs to be separately installed by using cargo install svd2rust. It supports multiple targets, among them RISCV. It generated a large lib.rs file that contains all APIs for all peripherals defined in the SVD file. To separate these APIs into different files, rustfmt is applied to lib.rs [21].

4.3 Embedded Rust Concepts

In Rust, different abstraction levels are available when programming software for embedded systems, and thus several types of crates (Rust libraries) are available. Figure 4.3 visualizes these levels and shows the relationship between these crates.These levels abstract the interaction with hardware and are mainly divided into four categories: micro-architecture crate, peripheral access crate (PAC), hardware abstraction layer (HAL) crates, and board crate.

The microarchitecture crate contains all routines common to the processor core and processor peripherals, such as setting timer interrupts. An example of such a crate is the RISCV cores which is the lowest crate in this hierarchy. A step upward is the PAC crate. It wraps interactions with memory-mapped peripherals defined for a particular core type, such as using an external timer in the Ballast syscrtl subsystem which uses pulpissimo RISC-V cores.

A HAL crate contains user-friendly APIs for particular processors. It uses both micro-architecture crate and PAC to create its APIs and focuses on portability. The process of creating this HAL can generally be divided into two stages:

1. SVD file to PAC: in this stage, the correct SVD (XML-style file containing all registers and their interfaces) file is generated from SVD tool or provided by the manufacturer. Then this file is converted via rust2svd tool into a PAC in a single source file. Optional formatting (using rustfmt) is done afterward to group each peripheral in a single file named after the specific peripheral for example gpio.rs and timer.rs.

2. PAC to HAL: In this stage, the PAC is used to develop types, methods, and functions to access and use peripherals easily.

Finally, there is a board crate, which is a crate for a specific board that might have several processors and external devices. It is responsible for providing a more abstract interface that, for example, predefines and configures external peripherals such as sensors and actuators [22].

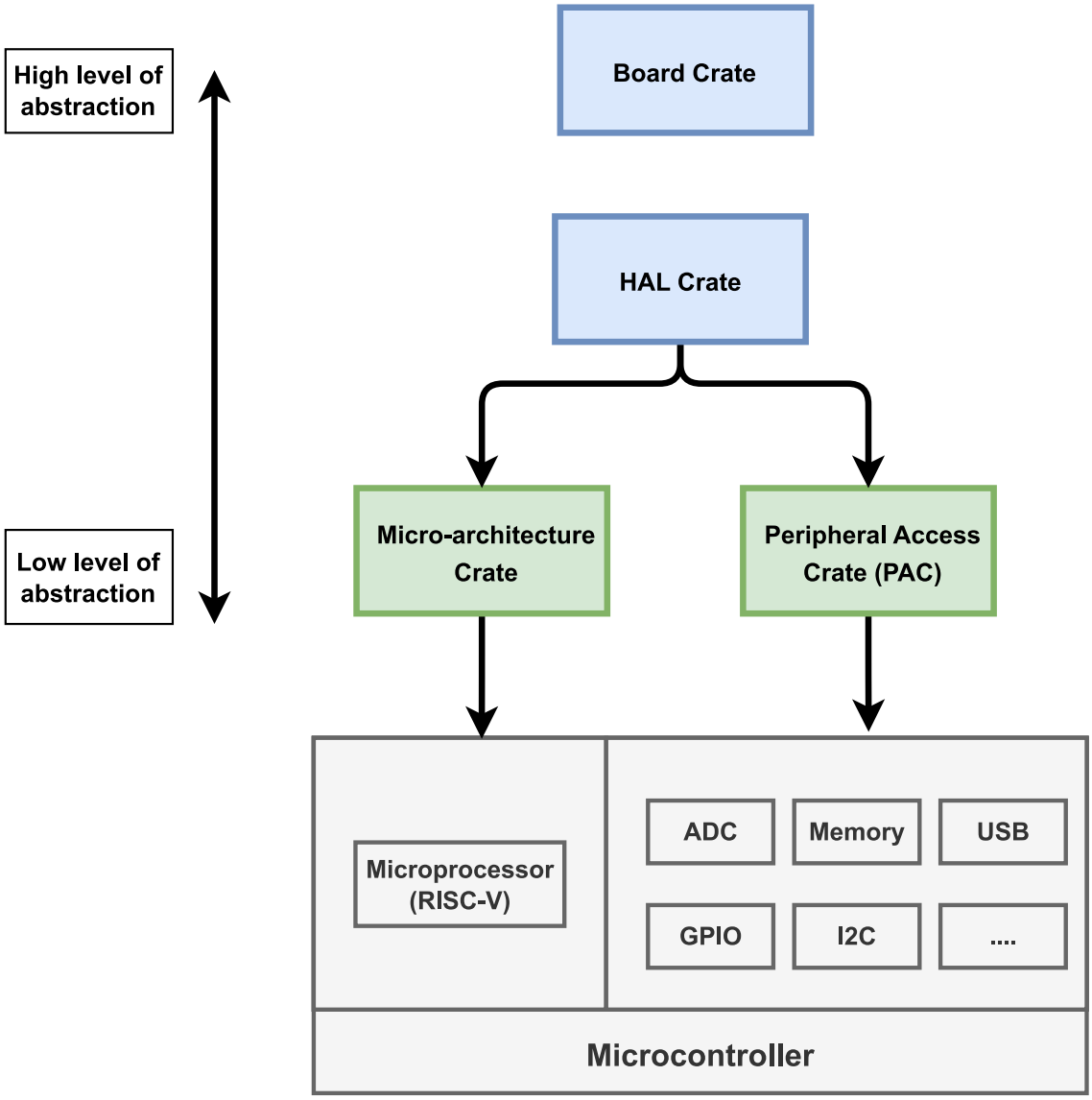


Figure 4.3 *Hierarchy of crated in Rust [22]*

5 Implementation of Interrupts

This chapter discusses the implementation of software used to bring up interrupts in Ballast MPSoC using Rust and describes the evaluation tests performed to verify the implemented software and interrupts.

5.1 Software Support Implementation

To have working interrupts on a RISC-V-based platform, three elements must be developed: access to RISC-V CSRs, a vector table, and a memory map to configure interrupt controllers.

Access to RISC-V CSRs is gained through the use of the open-source *riscv* crate, which provides APIs to set, reset, and configure each CSR register. This crate additionally provides ways to manipulate interrupts by providing functions to enable, disable, and free interrupts, to use assembly instructions such as wfi (wait for interrupts) and nop (no operation), and a delay functionality through the machine mode cycle counter. A fork from this crate was obtained and customized to remove generic implementation for mip and mie registers and replace them with matching implementation for SysCtrl and MPC cores.

SysCtrl and MPC subsystems only support interrupts in a vectored mode while, HPC uses direct mode; therefore, in SysCtrl and MPC a vector table was implemented to route interrupts to their specific handlers. This is done manually by using assembly language to jump to each interrupt handler according to the received interrupt ID. A macro function was used to create the vector table using jal (Jump And Link) instruction as follows:

jal x0, interrupt_handler_offset_1
jal x0, interrupt_handler_offset_2
...
jal x0, interrupt_handler_offset_n

where interrupt handler offsets are calculated as *mtvec.base+ interrupt ID *4*.

Additionally, another wrapper macro function was created to support calling user-defined handlers when interrupts occur. The function takes the user's ISR (Interrupt Service Routine) function address as an input and performs three operations: perform context saving and backup core registers into the stack, then jump to the user's ISR function, and finally restore context. Figure 5.1 gives a visual illustration

of this process. When an interrupt is received, the PC (Program Counter) register will be loaded with the address in the vector table corresponding to the interrupt ID. After that, it will jump to the handler function and perform context saving, then jump to the user's defined handler function and finally perform context restore after returning from the handler.

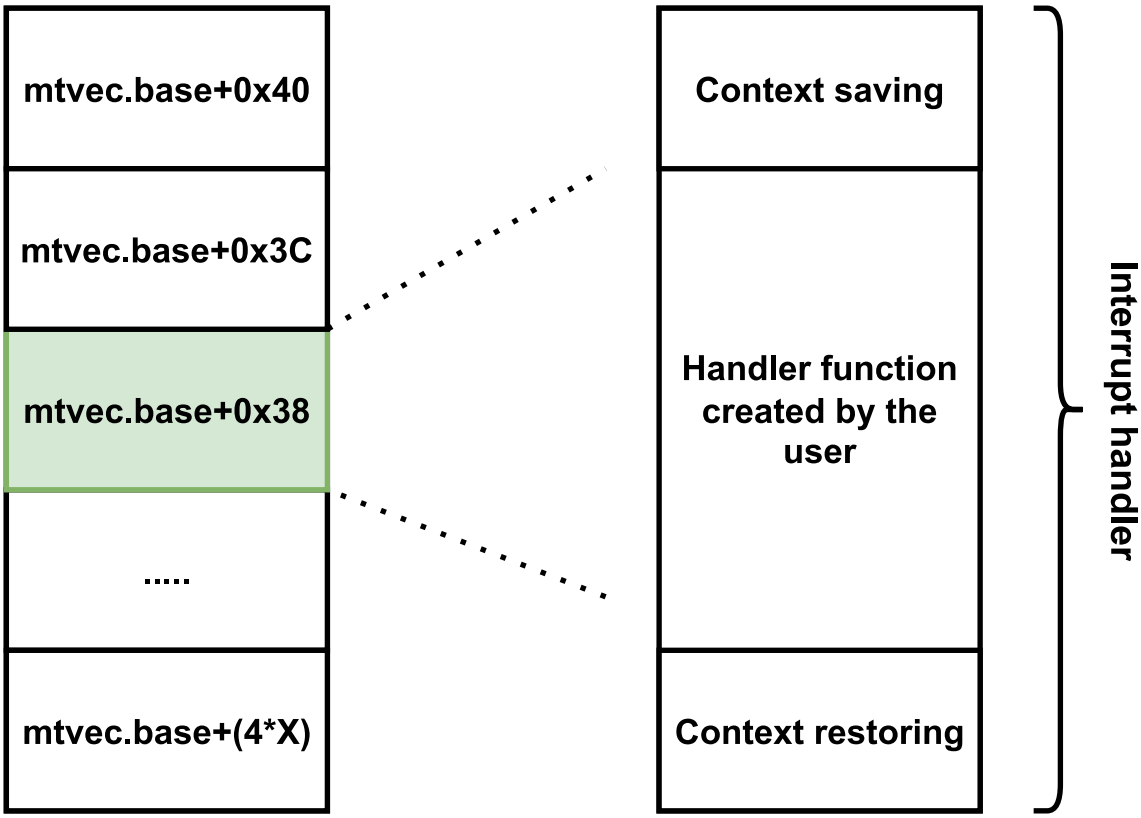


Figure 5.1 *Interrupt Handler and Vector Table*

Finally, to configure interrupt controllers on the platform, a memory-mapped interface is needed. This is produced by the Kactus2 tool using the IP-XACT memory map [23]. The generated SVD file validity is checked using the SVDConv script. Then,this SVD file is used by svd2rust tool to create a PAC interface.

Figure 5.2 summarize the previously stated steps to bring up interrupts. Getting access to CSRs and creating a PAC are essential for every interaction with the cores to configure interrupts. Then, depending on the interrupt mode, additional steps are taken. For vectored mode, it is important to create a vector table and means to register user's ISRs. Finally, both modes configure interrupt controllers using PAC directly or by creating HAL crates.

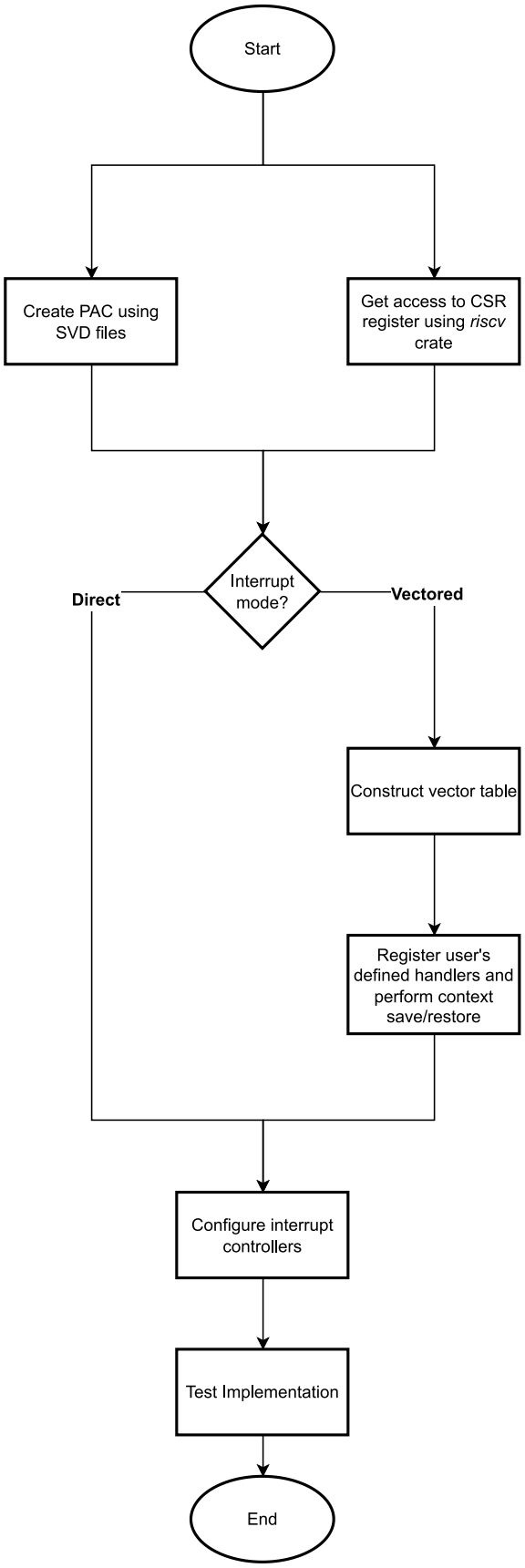


Figure 5.2 *Steps Taken to Bring up Interrupts*

Although it is not necessary to have a HAL crate to bring up interrupts, having interrupt support crates integrated inside a HAL will help in reusing software as well as managing development more efficiently. Therefore, for Ballast, soc-hub-hal was implemented. It is a container for pulpissimo-hal, a HAL crate for sysctrl and MPC subsystems, and hpc-hal. It contains several crates to abstract different MPSoCs (using the same cores as Ballast), external submodules such as RISC-V run time crate *riscv-rt*, and udma-hal.

Fundamentally, soc-hub-hal has the following structure:

1. Sub-systems PAC, which contains all register interfaces for a specific sub-system.
2. Granitti-hal, which is the HAL crate for ballast-based board Graniitti.
3. Pulpissimo-hal and hpc-hal, which are HAL crates for SysCtrl, MPC, and HPC subsystems, respectively.
4. Examples to show and test the operation of subsystem HAL crates

The following Figure 5.5 shows the architecture of soc-hub-hal as well as common functions between different subsystems.

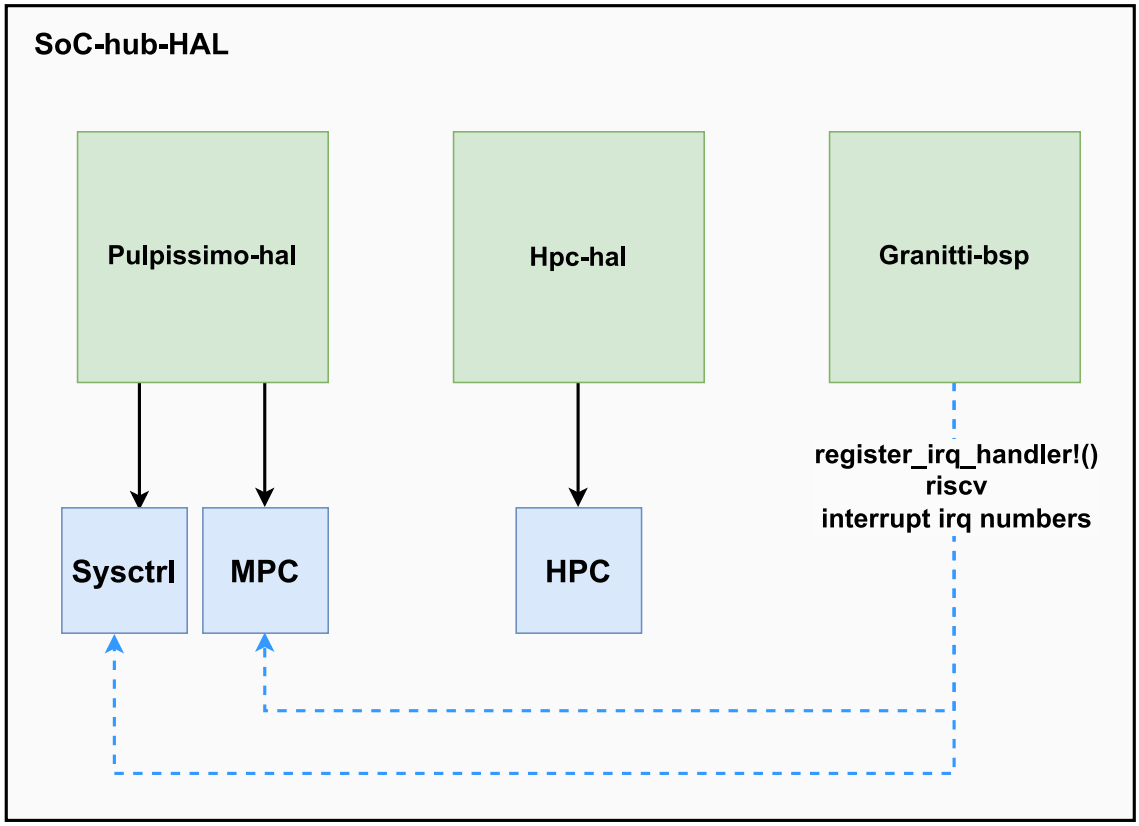


Figure 5.3 *Subsytems HAL Crates and Common Functions between the Crates*

The implementation process of interrupt support using a HAL is divided into three phases: creating a PAC for each subsystem core, creating a HAL crate and adding interrupt functionality to it, and creating examples that use the HAL crate. Kactus2 was used to create subsystem SVD files, which in turn were used to create PACs through the svd2rust tool.

In the second phase of HAL implementation, the HAL crates for each subsystem (pulpissimo-hal and hpc-hal) were created by using *cargo create lib* command. Following that is the addition of source files for each interrupt controller: *event_interrupt_unit.rs* in pulpissimo-hal (for SysCtrl and MPC), and *clint.rs* and *plic.rs* for HPC in hpc-hal. These sources are added to the src/ folder and exposed to other crates by adding them to lib.rs using *pub mod name_of_module*. To further support interrupts, the runtime crate riscv-rt and riscv crate were forked from the upstream repositories to be customized to use them in pulpissimo cores used in Ballast. The customization for riscv-rt included configuring interrupt mode to be vectored, as this is the only mode supported by pulpissimo cores, and moving interrupt vector to the start of the program. For the riscv crate, interrupt CSR registers mie, mip, and mtvec were moved, and their content was changed to match pulpissimo cores. Finally, to support interrupts in Ballast, helper macro functions and interrupt line numbers for each subsystem were added to lib.rs. *create_irq_trampoline!()* was used to create a handler that performs context saving and jumps to the interrupt handler function. On the other hand, *register_irq_handler!()* creates a vector table using the *create_irq_trampoline!()* handler.

After all elements are ready, the final stage is creating example projects to use the created crates to check implemented software functionality. This is usually done as a separate project. In these examples, usually a peripheral is initialized and configured properly to use interrupts, and then the interrupt is tested by a print function through UART.

5.2 Software Compilation

The support for interrupts in Ballast consisted of *event_interrupt_unit.rs*, *clint.rs*, and *plic.rs*. These Rust source files were encapsulated under pulpissimo-hal and hpc-hal, respectively. Additionally, commonly used functions in different subsystem cores were added to the board support library, i.e., in lib.rs under granitti-bsp.

The implemented software can be compiled individually. However, the binaries will not be useful as it is a library crate. To test the implemented software, an example must be developed. This example will include the library crate, which

is built using cargo tool with the correct target. For example, target riscv32imc-unknown-none-elf for SysCtrl and MPC and riscv64gc-unknown-none-elf for HPC. These targets are added to the rust toolchain using *rustup target add target-name*.

The build process of an example project can be divided into two stages: configuring cargo.toml and producing binaries by passing build parameters to cargo. Cargo.toml is configured by adding the necessary crates along with their version and source directory (or a GitHub link), and PAC directories for each core are added. PACs are used according to the feature passed to cargo; for example, the SysCtrl PAC will be used if *–feature=ballast-sysctrl* was passed as a parameter to cargo build command. The second stage is adding the compiler to rust toolchain using rustup add command and then passing this compiler to *cargo build* command using the *–target* parameter. Figure 5.4 summarizes the previous steps for compilation.

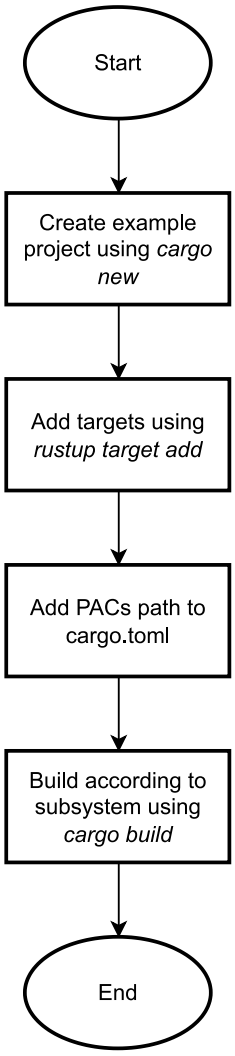


Figure 5.4 *Summary of Compilation Process*

5.3 Functionality

For the Event Interrupt unit, the crate user is given a set of methods to help mask and unmask an interrupt, that is, to either pass an interrupt from a specific source or not. Additionally, the user can read the current mask to check which interrupt sources are allowed to interrupt the core. Furthermore, the user can set, clear, and read pending interrupts; set and clear acknowledgment signals; and read unit's FIFO data. The crate also includes support for critical sections i.e. allowing only one thread in the interrupt execution environment. This is implemented in the *irq_free()* function, which, when called, disables global interrupts.

HPC is a dual-core subsystem, hence, *clint.rs* supports interrupt control for both cores. It supports raising software interrupts, signaling the core upon completion of the interrupt handler. In addition, it gives access to core and core compare timers to trigger timer interrupts. For PLIC configuration, the crate supports enabling interrupt sources, setting a priority for each source, and setting a threshold where interrupt priorities at and below the threshold value will be ignored. It also provides *complete()* and *next()* methods to signal finishing handling a pending interrupt and taking the next interrupt in the queue, respectively.

5.4 Evaluation Methods

Testing the implemented interrupt support was performed by running example codes in Graniitti. The tests examined the correct functionality of the implemented software, its memory footprint, and the safety aspects of the code. These tests also worked as verification tests for interrupts. The tests are described as follow:

1. **Correctness Tests:** In these tests, the correctness of interrupts operations was checked by using a global variable as a flag that gets set inside the interrupt handler. When the flag is set, text is printed using the UART console. Another way to test interrupts is by placing a breakpoint inside the handler and observing if the CPU reaches it, or by stepping over in the executable until reaching the interrupt handler when the interrupt is triggered.

2. **Memory Usage:** Memory usage was tested by compiling source code using interrupt support software and then evaluating the size of the executable. However, the size of the executable will mostly be effected by the number of used libraries of an SDK or set of used crates in a HAL rather than the size of the implemented software support for interrupts.

3. **Memory Safety:** Safety is tested by performing a code review of the sources

using interrupt support software and testing the compiler's ability to catch safety-related bugs when safety semantics are removed.

Figure 5.5 shows the setup used to verify interrupts. The setup included a JTAG/UART board that connects the Graniiti board to a host PC using the on-chip debugger Open On-chip Debugger (OpenOCD) server. Additionally, GNU Debugger (GDB) was connected over the openOCD server to flash executables and debug the software.

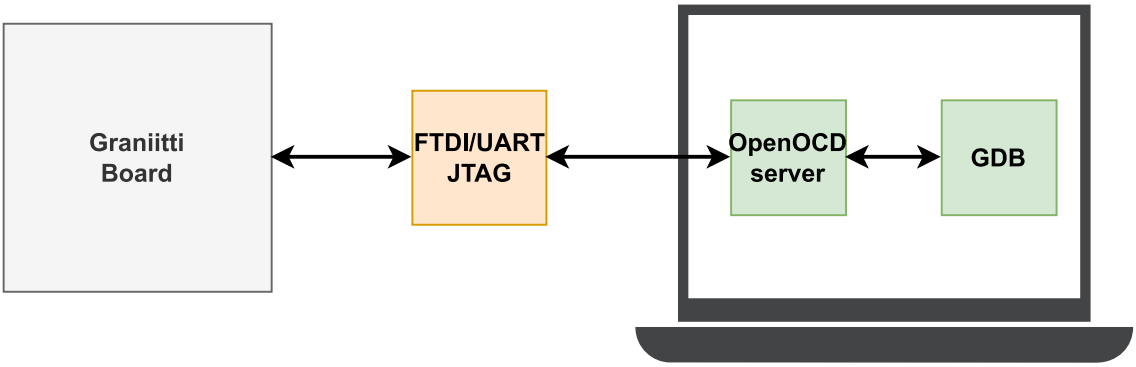


Figure 5.5 *Hardware Setup Used in Verifying Interrupts*

To use this setup, different elements must be available,the openOCD configuration file and the executable to be tested. To start the test process, the FTDI/UART board is connected to both the Graniiti board and the host PC. After that, the openOCD server is started by using *openocd –f openocd-config-file.cfg*. Then, GDB started using *riscv32imc-unknown-none-gdb* for Sysctrl and MPC and *riscv64gc-unknown-none-gdb* for HPC. Finally, to be able to see UART print, screen was used to connect to a serial port using 115200 baud rate.

5.5 Use Case 1: APB Timer Local Interrupt on Sysctrl

In pulpissimo cores, the APB timer is a timer with two upward counters, memory-mapped registers for configuration, and the ability to interrupt the core via the Event/Interrupt unit. In this use case, one of the basic APB timers (timer low) is configured to interrupt the core regularly every one second to demonstrate interrupts configuration in the Ballast SystCtrl subsystem using the implemented software support in Rust and the available C implementation using pulp-sdk. Additionally, the functionality of the timer is verified using the Graniitti board.

APB timer low is configured in continuous mode and produces interrupts on match, i.e., when the timer counter value equals the value in the timer compare register. This is done in three phases: initialize the Ballast platform, which includes

configuring clocks and resetting the timer's registers; configure interrupts, which includes configuring the event/interrupt unit and registering the interrupt handler; and finally, set the timer's compare register with a suitable value that corresponds to one second and start the timer.

5.5.1 Rust Implementation

Code snippet 5.1 shows how the APB timer low is configured using the Rust programming language. First, the clocks and UART for serial communication are initialized. Then, to configure timer interrupts, both the event/interrupt unit and timer peripherals must be brought into the current thread. In Rust, this is done by using PAC to create a handle for the device and core peripherals, bring it to the current execution context, and extract Sysctrl, the APB timer, and the Event/Interrupt unit from the device handle (p) as shown in lines 32,43 and 58 respectively. Next, the timer's start, compare, and configuration registers are cleared, and the timer counter is reset as shown in lines 47-50.

After that the interrupt handler function *handle_interrupt()* is registered using the macro function *register_irq_handler!()* in line 56. This function sets a flag to signal that a timer interrupt was triggered and the correct handler was found; it also clears timer interrupts from the event/interrupt unit after a timer interrupt is received to enable receiving a new interrupt. Timer interrupts are enabled in the event/interrupt unit and the core by setting corresponding bits in the mask and mie registers, as shown in lines 69 and 71 respectively.

Next, the timer is configured to generate interrupts on match, and the compare match is set to 0x8000, which corresponds to an approximation of one second, and counter is started. Finally, the core waits for the timer's interrupts using the *asm* instruction *wfi*.

```
1 #![no_std]
2 #![no_main]
3 #![feature(naked_functions)]
4 #![feature(fn_align)]
5
6 use core::{arch::asm, ptr};
7 use graniitti_bsp::{
8     pac, prelude::*, register_irq_handler, riscv, sprintln, udma::Serial,
9     SocControl, SysCtrlIrq,
10 };
11 use rt::entry;
12
13 static mut FLAG: bool = false;
14
```

```
15 /// Align to 4-byte boundary to allow entry from a RISC-V jump instruction
16 #[repr(align(4))]
17 fn handle_interrupt() {
18     // Write the flag (or do any other handling)
19     unsafe { ptr::write_volatile(&mut FLAG, true) };
20 
21     // Clear interrupt bit in interrupt controller to signal it's handled
22     let eu = unsafe { &(*pac::SYSCTRL::ptr()).event_interrupt_unit };
23     eu.int_clear
24         .modify(|r, w| unsafe { w.bits(r.bits() | 0b1) });
25 }
26 
27 #[entry]
28 fn main() -> ! {
29     // Obtain exclusive access to peripherals
30     let p = pac::Peripherals::take().unwrap();
31 
32     // Initialize Ballast system clocks
33     let mut sysctrl = p.SYSCTRL.split();
34     let soc_control = sysctrl.soc_control;
35     let clocks = SocControl::sysctrl(soc_control, 30.mhz());
36 
37     // Initialize SysCtrl's uDMA UART as a serial port
38     let udma = sysctrl.udma.split();
39     let serial = Serial::init(udma.uart, 115_200.bps(), &clocks);
40 
41     // Capture serial port as `sprintln` implementation
42     sprintln::capture(serial);
43 
44     let timer = &mut sysctrl.timer;
45 
46     // The PULP APB timer can be used as two separate 32-bit timers.
47     //  We reset the timer represented by the lower bits to prepare it for use.
48     timer.start_lo.reset();
49     timer.reset_lo.modify(|_, w| w.reset_lo().set_bit());
50     timer.cmp_lo.reset();
51     timer.cfg_lo.reset();
52 
53     // Register interrupt handler function for the lower timer interrupt
54     //
55     // This function creates a small interrupt trampoline to call our handler
56     // function, and writes an appropriate jump instruction in the vector table.
57     register_irq_handler!(SysCtrlIrq::TimerLoEvt, handle_interrupt);
58 
59     let eu = &sysctrl.event_interrupt_unit;
60 
61     // Clear the timer's IRQ bit in event/interrupt unit to avoid immediate
```

```
62      // triggering
63      eu.int_clear
64          .modify(|r, w| unsafe { w.bits(r.bits() | 0b1) });
65
66      // Unmask timer interrupt at core
67      unsafe { riscv::register::mie::set_timer_lo_evt() };
68
69      // Mask the interrupt at interrupt controller
70      eu.mask_set.modify(|r, w| unsafe { w.bits(r.bits() | 0b1) });
71
72      // Global interrupt enable
73      unsafe { riscv::register::mstatus::set_mie() };
74
75      // Configure timer (lower half)
76      timer.cfg_lo.modify(|_, w| {
77          w
78              // Interrupt on match
79              .irqen()
80              .set_bit()
81              // Use reference clock
82              .ccfg()
83              .set_bit()
84              // Reset after compare
85              .mode()
86              .set_bit()
87      });
88      // Set timer to trigger after 0x2000 cycles
89      timer.cmp_lo.write(|w| unsafe { w.bits(0x8000) });
90
91      // Start timer
92      timer.start_lo.modify(|_, w| w.start_lo().set_bit());
93
94      // Allow core to sleep while we wait for the interrupt
95      unsafe {
96          asm!("wfi");
97      }
98
99      // Clean up
100     unsafe { riscv::interrupt::disable() };
101     eu.mask_clear
102         .modify(|r, w| unsafe { w.bits(r.bits() | 0b1) });
103     unsafe { riscv::register::mie::clear_timer_lo_evt() };
104
105     loop {
106         continue;
107     }
```

```
108 }
```

Program 5.1 *Rust Code for Timer Interrupt Configuration*

5.5.2 Pulp SDK C Implementation

To be able to compare Rust and C interrupt support, the pulp-sdk C implementation is covered in this section. The same timer is used with similar configurations. The process is similar to Rust implementation: first, pulp.h is included to include pulp-sdk functions such as *rt_irq_set_handler()* and *rt_irq_mask_set()*, Appendix A and B shows *pulp.h* and *rt_irq.h* which includes the implementation of these functions. Then, clock and timer registers are cleared by calling *init_apb_timer()*.
To configure interrupts, an interrupt handler function is registered by linking it to an interrupt irq number through *rt_irq_set_handler()* function, as shown in line 37. The handler function has a GCC function attribute that enables the compiler to identify this function as an interrupt handler and generate suitable entry and exit sequences for it; the function is declared in line 10. Furthermore, interrupts are enabled by setting the corresponding bit in the event/interrupt unit mask register using *rt_irq_mask_set()*. After that, the timer is configured, the compare value is loaded directly into the timer's registers and timer is started as shown in lines 45, 48 and 51. Finally, the timer is started, and the core waits for interrupts using *wfi* instruction.

```
1  #include "pulp.h"
2 #include "sysctrl_defs.h"
3 #include <rt/rt_api.h>
4 #include <stdbool.h>
5 #include <stdint.h>
6 #include <stdio.h>
7
8 volatile int flag = 0;
9
10 void timer_lo_handler(void) __attribute((interrupt));
11 void timer_lo_handler(void)
12 {
13     flag = 1;
14 }
15
16 uint32_t init_apb_timer(void)
17 {
18     // clear peripheral clock divider
19     SYSCTRL_PERIPH_CLK_DIV |= 0x400;
20
```

```
21      SYSCTRL_APB_TIMER_START_LO = 0x0;
22      SYSCTRL_APB_TIMER_RESET_LO |= 0x01;
23      SYSCTRL_APB_TIMER_CMP_LO = 0x0;
24      SYSCTRL_APB_TIMER_CFG_LO = 0x0;
25  }
26
27  int main()
28  {
29
30
31      // clear/initialise registers
32      init_apb_timer();
33
34      printf("[test Timer Interrupts]\n");
35
36      // Assign irq handlers
37      rt_irq_set_handler(ARCHI_FC_EVT_TIMER0_LO, timer_lo_handler);
38
39      // Enable irq mask for above interrupts
40      rt_irq_mask_set(1 << ARCHI_FC_EVT_TIMER0_LO);
41
42      // timer configuration
43      // enable_interrupt_on_match | use FLL (just for simulation)
44      // | reset_val_after_compare
45      SYSCTRL_APB_TIMER_CFG_LO |= (1 << 2) | (0 << 7) | (1 << 4);
46
47      // Set cmp value
48      SYSCTRL_APB_TIMER_CMP_LO = 0x8000;
49
50      // Start timer
51      SYSCTRL_APB_TIMER_START_LO = 1;
52
53      while (1)
54      {
55          // Puts core to sleep while we wait for the interrupt
56          asm("wfi");
57
58          // check if interrupt is fired
59          if (flag == 1)
60          {
61
62              flag = 0;
63          }
64
65      }
66
67      return 0;
```

```
68 }
```

Program 5.2 *C Code for timer interrupt configuration*

5.6 Use Case 2: External Timer Interrupt on HPC

As mentioned in Chapter 3, both HPC cores are connected to PLIC for external interrupt control. The timer in HPC subsystem is external and connected to PLIC input. Also, HPC uses only direct interrupt mode, i.e., it does not have a vector table. Finally, there was no C implementation for PLIC in HPC, as pulp-sdk only works with pulpissimo cores, i.e., SyCtrl and MPC cores only. In this use case, PLIC is configured to handle timer interrupts, and the functionality of interrupts is tested by declaring a global flag HANDLED that gets set inside the interrupt handler. Additionally, the handler also defines the interrupt source and hart number of the interrupted core by reading mcause CSR register and mhartid. The timer generates interrupts every 2 seconds. Source code in Listing 5.3 shows how PLIC is configured. After bringing device handle dp to execution environment, HPC, PLIC, and timer peripherals are extracted, and PLIC is configured as follows:

1. The priority of the interrupt is set; the priority range is from 0–7, with 7 being the highest.

2. Enable timer interrupts in PLIC.

After that, the timer is configured and started. In this use case, the timer is set to interrupt the core every 2 seconds. Then the core waits for the interrupt to occur. When a timer interrupt is received, the handler function is executed. Inside the handler, the timer interrupt is claimed by reading the claim/complete register, which returns the ID of the claimed interrupt, in this case the ID of the timer interrupt. Then HANDLED is set and claim/complete register is written back with timer ID to indicate it has been handled. Finally, the program prints a message to indicate the interrupt is handled and also prints mcause value.

```
1 use graniitti_bsp::{
2     metapac, pac::Peripherals, prelude::*, riscv, sprintln,
3     Context, Interrupt, InterruptSource,
4     Plic,
5 };
6 use graniitti_hpc_examples::{init_clocks_and_serial, print_example_name,
7     EXAMPLE_SOC_FREQ
8 };
9 use rt::entry;
10
11 static mut HANDLED: bool = false;
```

```
12 static mut ID: u32 = 0;
13 static mut MCAUSE_CODE: usize = 0;
14
15 static mut PLIC: Option<metapac::hpc::PLIC> = None;
16
17 #[export_name = "DefaultHandler"]
18 fn handle_interrupt() {
19     let hart = riscv::register::mhartid::read();
20
21     if hart == 0 {
22         let mcause = riscv::register::mcause::read();
23         let mcause = mcause.code();
24
25         sprintln!("mcause: {}, HART: {}", mcause, hart);
26         assert_eq!(mcause, 11, "[not ok]\ninterrupt source must be mext");
27
28         if let Some(plic) = unsafe { PLIC.as_mut() } {
29             // Claim interrupt by reading interrupt id from Claim register 0
30         let claim_complete_0 = (plic.as_ptr() as usize + 0x200004) as *mut u32;
31         unsafe { ID = core::ptr::read_volatile(claim_complete_0) };
32
33             // Set Magic value which will indicate the test was succesful
34             unsafe {
35                 HANDLED = true;
36                 MCAUSE_CODE = mcause;
37                 // Clean up for next run
38                 riscv::register::mstatus::clear_mie();
39             };
40
41             // Complete interrupt by writing back the received ID
42             unsafe { core::ptr::write_volatile(claim_complete_0, ID) };
43
44         }
45     }
46 }
47
48 #[entry]
49 fn main() -> ! {
50     init_clocks_and_serial();
51     print_example_name!(false);
52
53     let p = unsafe { Peripherals::steal() };
54     let hpc = p.HPC.split();
55
56     let clint = hpc.clint;
57
58     // Disable pending CLINT timer irq by setting compare register
```

```
59      // value for timer 0 and timer 1
60      clint.mtimecmp[0].write(|w| unsafe { w.mtimecmp().bits(0x100) });
61      clint.mtimecmp[1].write(|w| unsafe { w.mtimecmp().bits(0x100) });
62
63      unsafe {
64          // N.b. original source sets `mie` at idx 11, which matches with
65          // `mext` in our RISC-V crate
66          riscv::register::mie::set_mext();
67          riscv::register::mstatus::set_mie();
68      }
69
70      // Set IRQ priorities for APB timers
71      let plic = hpc.plic;
72
73      let mut plic = Plic(plic);
74      plic.set_priority(
75          0x7,
76          graniitti_bsp::InterruptSource::Timer0,
77          Interrupt::Interrupt0,
78      );
79      plic.set_priority(
80          0x0,
81          graniitti_bsp::InterruptSource::Timer0,
82          Interrupt::Interrupt1,
83      );
84      plic.set_priority(
85          0x0,
86          graniitti_bsp::InterruptSource::Timer1,
87          Interrupt::Interrupt0,
88      );
89      plic.set_priority(
90          0x0,
91          graniitti_bsp::InterruptSource::Timer1,
92          Interrupt::Interrupt1,
93      );
94
95      // Enable APB IRQ idx 1 for timer 0 at context 0
96      plic.enable(Context::Core0Machine, 0b1 << 0);
97
98      unsafe { PLIC.insert(plic.free()) };
99
100     let timer = hpc.timer;
101
102     // Set apb timer compare register to 0, disables compare
103     // interrupt this will also zero the timer register
104     timer.timer0_cmp.reset();
105
```

```
106     // Set apb timer register to high value, which will
107     // soon overflow and send IRQ 0
108     timer
109         .timer0_timer
110         // 60_000_000 == 2 seconds
111         .write(|w| unsafe { w.bits(0xFFFFFFFF - 60_000_000) });
112 
113     // Enable timer
114     timer.timer0_ctrl.modify(|_, w| w.enable().set_bit());
115 
116     loop {
117         if unsafe { HANDLED } {
118             sprintln!("IRQ code: {}", unsafe { MCAUSE_CODE });
119             sprintln!("[ok]");
120             break;
121         } else {
122             sprintln!("not yet");
123         }
124         let mut counter =
125         graniitti_bsp::riscv::delay::McycleDelay::new(EXAMPLE_SOC_FREQ);
126         counter.delay_ms(1_000);
127     }
128 
129     // Disable timer & IRQs
130     timer.timer0_ctrl.modify(|_, w| w.enable().clear_bit());
131     unsafe {
132         riscv::register::mstatus::clear_mie();
133         riscv::register::mie::clear_mext();
134     }
135     if let Some(plic) = unsafe { PLIC.take() } {
136         let mut plic = Plic(plic);
137         plic.disable(Context::Core0Machine, 0b1 << 1);
138         // Put it back into static context
139         let plic = plic.free();
140         unsafe { PLIC.insert(plic) };
141     }
142 
143     loop {
144         continue;
145     }
146 }
```

Program 5.3 *Rust Code for PLIC Configuration*

6 Results and Discussion

This chapter reports the results obtained from developing interrupt support for Ballast MPSoc using Rust and minimal comparison with the results obtained from similar C implementation.

6.1 General Results

As discussed in Chapter 3, Ballast has three subsystems that use RISC-V cores. These subsystems are SysCtrl, MPC, and HPC. Both SysCtrl and MPC have a custom event/interrupt unit. In contrast, HPC cores have a CLINT, and external interrupts are handled through a PLIC. Overall, the implemented software was successfully working; the *risv* crate was used for CSR access, and a vector table was implemented through *register_irq_handler!()*, a function used to create a vector table, register interrupt handlers in it, and perform context save and restore by creating an assembly trampoline. Figures 6.1(a) and 6.1(b) show the output of *register_irq_handler!()* of use case 1, which are interrupt vector and assembly trampoline, respectively.

Additionally, PACs for different interrupt controllers were successfully created using kactus2-generated SVD files, svd2rust tool, and formatted to Rust standard format using rustfmt tool. However, the generated SVD files were incomplete due to missing definitions in IPXACT. Hence, it needed to be patched manually, where the patch file included fixes for unallowed empty fields and redundant prefixes in register names. This process must be done for svd2rust to work correctly and give a valid and usable PAC. Also, the generated SVD files did not support interrupts. I.e., it did not provide vector table information or interrupt handlers addresses, which had to be manually implemented. Furthermore, the implemented software was included as a part of soc-hub-hal, which will ensure portability and modularity and can compile software support backage for different platform architectures using these cores, making it platform agnostic.

Finally, there were several tools that eased the development of interrupt support using Rust, these are svd2rust, rustfmt rust-analyzer, and clippy. Svd2rust made it easy to generate PACs from Kactus2 SVD files. The output of svd2rust is a single lib.rs file that contains thousands of lines of peripheral APIs; therefore, the formatting tool rustfmt was used to separate lib.rs into several files, each representing a peripheral, which made the PAC more readable and easier to use. The most used

```
Disassembly of section .vectors:

1c008000 <_vectors>:
1c008000:       1c40006f                j       1c0081c4 <_no_irq_handler>
1c008004:       1c00006f                j       1c0081c4 <_no_irq_handler>
1c008008:       1bc0006f                j       1c0081c4 <_no_irq_handler>
1c00800c:       1b80006f                j       1c0081c4 <_no_irq_handler>
1c008010:       1b40006f                j       1c0081c4 <_no_irq_handler>
1c008014:       1b00006f                j       1c0081c4 <_no_irq_handler>
1c008018:       1ac0006f                j       1c0081c4 <_no_irq_handler>
1c00801c:       1a80006f                j       1c0081c4 <_no_irq_handler>
1c008020:       1a40006f                j       1c0081c4 <_no_irq_handler>
1c008024:       1a00006f                j       1c0081c4 <_no_irq_handler>
1c008028:       19c0006f                j       1c0081c4 <_no_irq_handler>
1c00802c:       1980006f                j       1c0081c4 <_no_irq_handler>
1c008030:       1940006f                j       1c0081c4 <_no_irq_handler>
1c008034:       1900006f                j       1c0081c4 <_no_irq_handler>
1c008038:       18c0006f                j       1c0081c4 <_no_irq_handler>
1c00803c:       1880006f                j       1c0081c4 <_no_irq_handler>
1c008040:       1840006f                j       1c0081c4 <_no_irq_handler>
1c008044:       1800006f                j       1c0081c4 <_no_irq_handler>
1c008048:       17c0006f                j       1c0081c4 <_no_irq_handler>
1c00804c:       1780006f                j       1c0081c4 <_no_irq_handler>
1c008050:       1740006f                j       1c0081c4 <_no_irq_handler>
1c008054:       1700006f                j       1c0081c4 <_no_irq_handler>
1c008058:       16c0006f                j       1c0081c4 <_no_irq_handler>
1c00805c:       1680006f                j       1c0081c4 <_no_irq_handler>
1c008060:       1640006f                j       1c0081c4 <_no_irq_handler>
1c008064:       1600006f                j       1c0081c4 <_no_irq_handler>
1c008068:       15c0006f                j       1c0081c4 <_no_irq_handler>
1c00806c:       1580006f                j       1c0081c4 <_no_irq_handler>
1c008070:       1540006f                j       1c0081c4 <_no_irq_handler>
1c008074:       1500006f                j       1c0081c4 <_no_irq_handler>
1c008078:       14c0006f                j       1c0081c4 <_no_irq_handler>
```

(a) Generated vector table

```
2503 |          unsafe extern "C" fn [< _ $handler _trampoline >]() {
2504                 core::arch::asm! {
2505 1c00938c:       7139                    addi    sp,sp,-64
2506 1c00938e:       c006                    sw      ra,0(sp)
2507 1c009390:       c216                    sw      t0,4(sp)
2508 1c009392:       c41a                    sw      t1,8(sp)
2509 1c009394:       c61e                    sw      t2,12(sp)
2510 1c009396:       c82a                    sw      a0,16(sp)
2511 1c009398:       ca2e                    sw      a1,20(sp)
2512 1c00939a:       cc32                    sw      a2,24(sp)
2513 1c00939c:       ce36                    sw      a3,28(sp)
2514 1c00939e:       d03a                    sw      a4,32(sp)
2515 1c0093a0:       d23e                    sw      a5,36(sp)
2516 1c0093a2:       d442                    sw      a6,40(sp)
2517 1c0093a4:       d646                    sw      a7,44(sp)
2518 1c0093a6:       d872                    sw      t3,48(sp)
2519 1c0093a8:       da76                    sw      t4,52(sp)
2520 1c0093aa:       dc7a                    sw      t5,56(sp)
2521 1c0093ac:       de7e                    sw      t6,60(sp)
2522 1c0093ae:       fbeff0ef                jal     ra,1c008b6c <_ZN19timer_interrupt_raw12apb_timer_lo17hb910f9b09842ae05E>
2523 1c0093b2:       4082                    lw      ra,0(sp)
2524 1c0093b4:       4292                    lw      t0,4(sp)
2525 1c0093b6:       4322                    lw      t1,8(sp)
2526 1c0093b8:       43b2                    lw      t2,12(sp)
2527 1c0093ba:       4542                    lw      a0,16(sp)
2528 1c0093bc:       45d2                    lw      a1,20(sp)
2529 1c0093be:       4662                    lw      a2,24(sp)
2530 1c0093c0:       46f2                    lw      a3,28(sp)
2531 1c0093c2:       5702                    lw      a4,32(sp)
2532 1c0093c4:       5792                    lw      a5,36(sp)
2533 1c0093c6:       5822                    lw      a6,40(sp)
2534 1c0093c8:       58b2                    lw      a7,44(sp)
2535 1c0093ca:       5e42                    lw      t3,48(sp)
2536 1c0093cc:       5ed2                    lw      t4,52(sp)
2537 1c0093ce:       5f62                    lw      t5,56(sp)
2538 1c0093d0:       5ff2                    lw      t6,60(sp)
2539 1c0093d2:       6121                    addi    sp,sp,64
2540 1c0093d4:       30200073                mret
```

(b) Trampoline function created by *register_irq_handler!()*.

Figure 6.1 *Assembly output from one of the compiles examples*

tool that made development in Rust much easier is rust-analyzer. It was used with Visual Studio Code and enabled code auto-completion, error detection without compiling the code, and fix suggestions. Additionally, the most useful feature of rust-analyzer was inlay hints, which provided the types of the variables and chained expression and function arguments without the need to change files. Figure 6.2 is an example of error detected by rust-analyzer on-the-fly without compilation. This helped with faster code writing and early error detection and provided an additional guard to prevent developers from falling into unintentional mistakes that could lead to bugs.

```rust
#[entry]
fn main() -> ! {
    let _soc_control: SocControl<SOC_CONTROL> = init_clocks_and_serial();

    let counter: McycleDelay = riscv::delay::McycleDelay::new(EXAMPLE_SOC_FREQ);

    print_example_name!(false);

    for n in 0..5 {
        counter.delay_ms(500);
        sprintln!("Tick {}", n);
    }
    sprintln!("[ok]");

    loop {
        continue;
    }
}
```

Figure 6.2 *Error generated by rust-analyzer*

Cargo is the build and package manager of Rust. It was used to customize builds for each subsystem by using the *feature* keyword; this allowed only compiling the related parts. Additionally, it was used to specify and manage different third-party crates, such as riscv-rt and riscv. And allowed to specify crates sources and versions in a single file, cargo.toml, which made it easier to manage the files and their dependencies. For example, Figure 6.3 shows cargo.toml configuration for HPC, showing author information and dependencies crates with thier specified versions.

```toml
[package]
name = "graniitti-hpc-examples"
authors = [                                                    ]
edition = "2021"
version = "0.2.0"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
soc-hub-common = { path = "../soc-hub-common" }
graniitti-bsp = { path = "../graniitti-bsp", features = ["ballast-hpc", "panic-uart"] }
rt = { package = "riscv-rt", version = "0.11" }
```

Figure 6.3 *Cargo configuration for HPC subsytem example project*

6.2 Evaluation Results

In this section, Rust and C implementations will be evaluated according to the measures mentioned in Chapter 5.

6.2.1 Compilation Process

For the Rust implemented use cases, cargo was used to build the binaries using *riscv32imc-unknown-none-elf* target for sysctrl and *riscv64gc-unknown-none-elf* for HPC . On the other hand,C implementation used pulp-sdk runtime for its interrupt

support, therefore pulp-sdk runtime had to be built and its directory had to be sourced. Both runtime build and software compilation were done using makefiles on linux machine. For building the runtime, RISC-V tool chain was downloaded and built. The following Listing 6.1 shows the Makefile used to compile one C use case. It takes pulp-sdk path from environment variable PULP_SDK_HOME and includes *pulp_rt.mk* which links pulp-sdk to use case source code.

```
PULP_APP = timer
PULP_APP_FC_SRCS = timer.c
PULP_APP_HOST_SRCS = timer.c
PULP_CFLAGS = -O3 -g
include $(PULP_SDK_HOME)/pulp_rt.mk
```

Program 6.1 *Timer example Makefile*

6.2.2 Test Setup

The hardware setup is shown in Figure 6.4. On the left side is Graniiti board connected FTDI chip and Rs 232 board for UART communication, then connected to the host PC. On the host PC, the openOCD server is started using the following command:

```
Sudo openocd -f config.cfg
```

Where *config.cfg* represents the subsystem configuration file. Finally, GDB was started using the following commands for sysctrl and HPC, respectively.

```
riscv32imc-unknown-none--gdb executable
riscv64gc-unknown-none-gdb executable
```

And then connected to openocd server using *target remote :3333* and the load command to load the executable.To further confirm the interrupt operation, a breakpoint was created in the interrupt handler using GDB command *br*, and then *stepi* was used to step inside the handler.

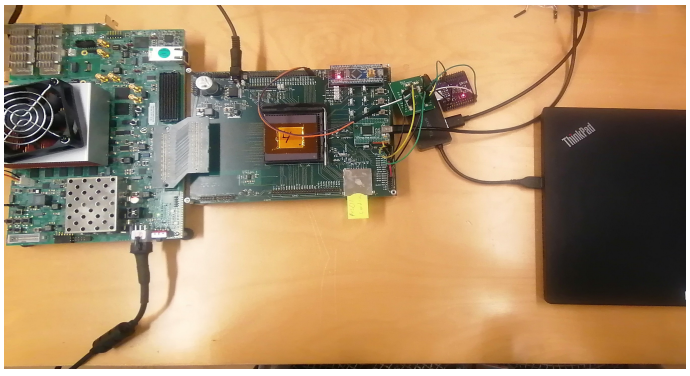
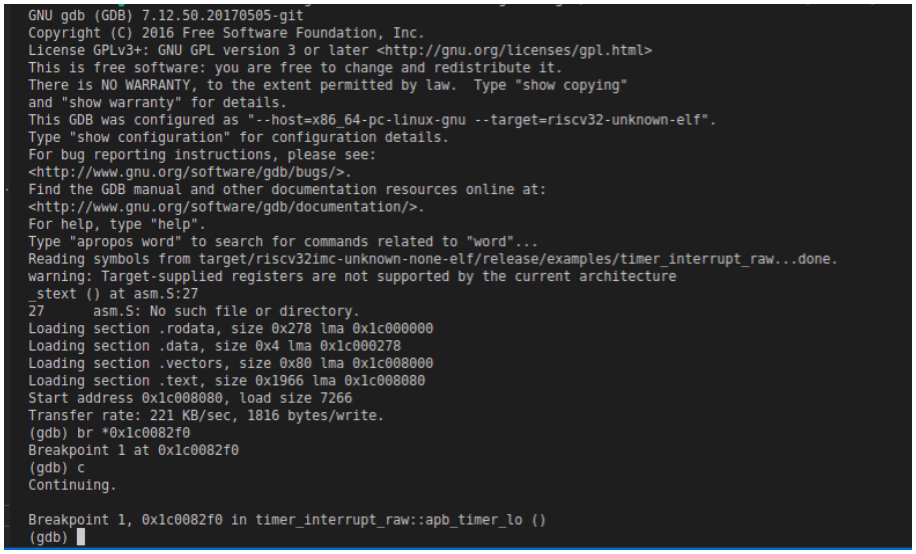


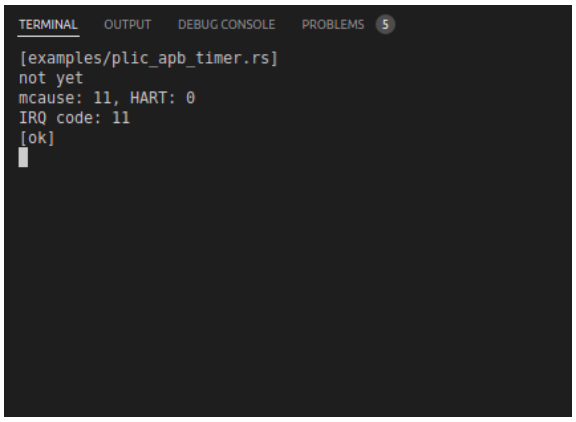
Figure 6.4 *Hardware setup using Graniitti board*

6.2.3 Correctness Tests

Both uses case successfully configured timer interrupts in their corresponding subsystems and passed the correctness tests. In each use case an interrupt was fired every second, and this was indicated by a printed output in UART ternimal ”screen” and using GDB to create a break point inside the interrupt handler. After loading the executable and starting to execute the program, the CPU jumped to the registered interrupt handler. The following Figure 6.5(a) and 6.5(b) shows GDB session for use case 1 and printed output for use case 2 respectively.



(a) GDB Session



(b) Printed Output

Figure 6.5 *Debug methods used on the use cases*

6.2.4 Memory Usage

Table 6.1 shows binaries sizes obtained from compilation of use cases in both Rust and C implementations. There is not much difference between Rust and C implementation in SysCtrl use case. This indicates that a Rust implementation can replace a C implementation without sacrificing much memory. On the other hand, HPC use case uses minimal amount of memory.

Table 6.1 *Executable sizes of both use-cases in comparison to C implementation*

Use-case	**Rust executable**	**C extecutable**
SysCtrl Use-case	3.0 MB	4.0 MB
HPC Use-case	980 KB	-

6.2.5 Memory Safety

Code memory safety is evaluated by reviewing the source code of interrupt support crates and the source codes of the use cases. Also, by checking implementation design choices regarding possible memory violations like accessing an unintended memory address and compiler-generated errors when there is a safety violation.

In Rust implementation, safety is achieved by following the rules of borrow checker and PAC usage for accessing peripherals. All Rust implemented software support for interrupts (clint.rs, plic.rs, and event_interrupt_unit.rs) follows borrow checkers rules and has the same structure of having a single instance of each peripheral by creating a type structure to leverage the use of borrow checker. For example, for an event/interrupt unit, a structure is declared as follows:

```
struct EventInterruptUnit<EVENT_INTERRUPT_UNIT>(EVENT_INTERRUPT_UNIT);
```

and in plic.rs, a structure is declared as follows:

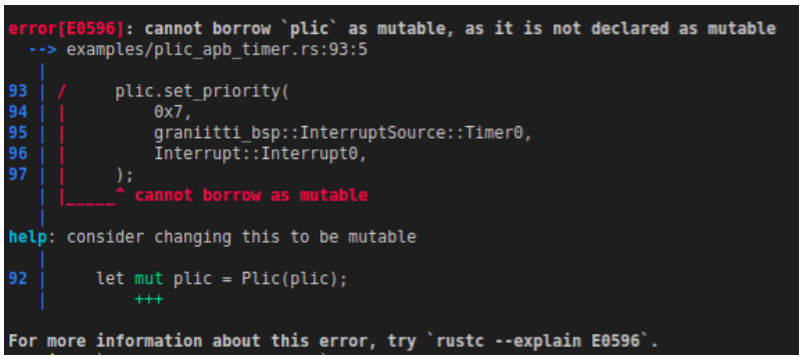
```
pub struct Plic(pub PLIC);
```

To use these peripherals, these structures must be brought to the execution environment and used to configure and interact with the peripheral through their associated method, Listing 6.2 shows code snippet for use case 2 where *Plic* from PAC is converted into crate *Plic* structure. Then priority is set through the *set_priority()* method and enabled through the *enable()* method. Each of these methods borrows a mutable reference to the currently created structure Plic by having *&mut self* as a parameter. This gives the method the ability to make changes in PLIC. After changes are made, the reference is released. Having such an ownership system prevents changes in PLIC from multiple sources simultaneously, and violation of the

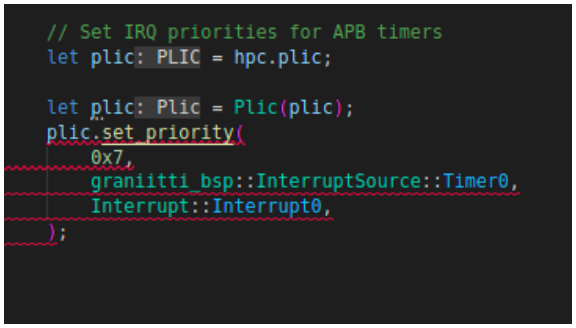
borrowing rules will raise a compiler error. Figure 6.6 shows how a compiler error is generated when attempting to pass an immutable structure, i.e., by removing the *mut* keyword from structure instantiation and calling the *set_periorty()* method and rust-analyzer error generated without compilation.

```
plic.set_priority(interrupt ID,Timer0, interrupt_on_cmp_match);
plic.enable(Context::Core0Machine, 0b1 << 0);
```

Program 6.2 *Code snippet from HPC use case*



(a) Compiler error after build



(b) rust-analyzer error without compilation

Figure 6.6 *Error generated when passing immutable reference*

That said, unsafe Rust is also used when writing CSR, for example when setting *mie* in the use case 1. Even though unsafe Rust does not follow borrow checker's rules, it makes it easier to identify bugs when they happen.

PAC is used to create a handle for the device and core peripherals and bring it to the current execution context and extract sysctrl, APB timer, and event/interrupt unit from the device handle (p) in use case 1 and HPC, APB timer, and PLIC in use case 2. In addition to providing singularity i.e., only one thread is allowed to use peripherals at a time, the correct memory addresses are guaranteed to be correctly accessed because PAC is automatically generated using svd2rust. In contrast, C uses manually written header files, which are prone to human error.

In conclusion, the process of implementing software support was successful. It included the development of three main elements: PAC, vector table, context saving trampoline and finally interrupt controller access and configuration. Two use cases were used to demonstrate the use of the software and evaluate interrupts. Both use cases passed the correctness test, and both have fairly low memory footprints.

7 Conclusion

This thesis investigated the implementation of software support crates for interrupt support on the Ballast MPSoC. The objective of this thesis was to explore implementation steps using Rust programming language, provide example use cases for using and verifying interrupts, and also present a minimal comparison of Rust and C-based implementations.

In this thesis, a study of RISC-V architecture and RISC-V interrupt system was conducted, and various interrupt controllers were presented. The Ballast SoC was chosen to bring up interrupts and was studied thoroughly to identify different interrupt system topologies. Rust was introduced and chosen for development as it is known for having high standards for memory safety with its ownership model and affine types. The design steps and the tools used in development were discussed, and test methods were used to verify correct configuration and operation. Use cases were demonstrated. Lastly, the results of the implemented software were presented, as well as the results of the use cases. Overall, bringing up the interrupt using Rust programming language was confirmed to be successful, and the interrupt worked perfectly.

Various difficulties were encountered and resolved when implementing software support. Access to interrupt controllers memory-mapped registers was obtained by automatically creating PACs using Kactus2 SVD files and the svd2rust tool. Moreover, the creation of interrupt vector tables for SysCtrl and MPC was done manually using assembly language through *register_irq_handler!()* macro function. This function also registered interrupt handlers to their correct addresses in the vector table and performed context save and restore before and after entering the interrupt handler function, respectively. To debug software and verify interrupts, two use cases were developed. GDB and a printing function were used to trace the sequence of program execution and to confirm entering the correct interrupt handler.

Several topics were left open for research in this thesis. First, adding support for interrupts in Kactus2 will allow svd2rust tool to generate a PAC that includes vector table information with interrupt handler addresses, eliminating the need to develop them manually. Additionally, adding support for the interrupt calling convention in Rust compiler will allow automatic context saving, restoring, and invocation of interrupt handlers. Finally, as the development in Ballast is heading towards booting Linux, testing the implemented software using Linux and Real-Time Linux would

be an interesting way to observe the behavior of implemented software under an Operating System (OS).

References

[1] *History of RISC-V*. URL: https://riscv.org/about/history/.

[2] Xi Wang et al. "Undefined Behavior: What Happened to My Code?" In: *Proceedings of the Asia-Pacific Workshop on Systems*. APSYS '12. Seoul, Republic of Korea: Association for Computing Machinery, 2012. ISBN: 9781450316699. DOI: 10.1145/2349896.2349905. URL: https://doi.org/10.1145/2349896.2349905.

[3] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.

[4] *Soc Hub*. 2023. URL: https://sochub.fi/.

[5] Krste Asanović and David A Patterson. "Instruction sets should be free: The case for risc-v". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

[6] Andrew Waterman et al. *The risc-v instruction set manual. volume 1: User-level isa, version 2.0*. Tech. rep. California Univ Berkeley Dept of Electrical Engineering and Computer Sciences, 2014.

[7] RISC-V Community News. *Semico Research's new report predicts there will be 25 billion RISC-V-based AI socs by 2027: Rich Wawrzyniak, Semico Research Corporation*. 2022. URL: https://riscv.org/blog/2022/02/semico-researchs-new-report-predicts-there-will-be-25-billion-risc-v-based-ai-socs-by-2027/.

[8] Andrew Waterman et al. *The risc-v instruction set manual volume 2: Privileged architecture version 1.7*. Tech. rep. University of California at Berkeley Berkeley United States, 2015.

[9] *SiFive Interrupt Cookbook Version 1.2*. URL: https://starfivetech.com/uploads/sifive-interrupt-cookbook-v1p2.pdf.

[10] *RISC-V Platform-Level Interrupt Controller Specification*. URL: https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc.

[11] *PULP SDK*. URL: https://github.com/pulp-platform/pulp-sdk/tree/v1.

[12] *Freedom Metal*. URL: https://github.com/sifive/freedom-metal.

[13] *Crate cortex_m_rt*. URL: https://docs.rs/cortex-m-rt/latest/cortex_m_rt/.

[14] *PULP Platform: Implementation*. URL: https://pulp-platform.org/implementation.html.

[15] *CVA6 RISC-V CPU*. URL: https://github.com/openhwgroup/cva6.

[16] *Stack Overflow Annual Survey*. URL: https://survey.stackoverflow.co/2022/.

[17] *The Cargo Book*. URL: https://doc.rust-lang.org/cargo/.

[18] *Clippy Lint Tool*. URL: https://github.com/rust-lang/rust-clippy.

[19] *Rust-analyzer Plugin*. URL: https://github.com/rust-lang/rust-analyzer.

[20] *Rustfmt Formatting Tool*. URL: https://github.com/rust-lang/rustfmt.

[21] *Svd2rust Tool*. URL: https://github.com/rust-embedded/svd2rust.

[22] *The Embedded Rust Book*. URL: https://docs.rust-embedded.org/book/.

[23] *Kactus2 Tool*. URL: https://github.com/kactus2/kactus2dev.

APPENDIX A. Pulp.h File

```
/*
 * Copyright (C) 2018 ETH Zurich and University of Bologna
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

/*
 * Copyright (C) 2018 GreenWaves Technologies
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

#ifndef __PULP_H__
#define __PULP_H__

#include "rt.h"
#include "bench/bench.h"
#include "rt/rt_compat.h"

#endif
```

Program 7.1 *Pulp.h Header File [11]*

APPENDIX B. Rt_irq.h File

```
/*
 * Copyright (C) 2018 ETH Zurich and University of Bologna
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

/*
 * Copyright (C) 2018 GreenWaves Technologies
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

#ifndef __RT_RT_IRQ_H__
#define __RT_RT_IRQ_H__

/// @cond IMPLEM

#include "hal/pulp.h"

void __rt_irq_init();
void rt_irq_set_handler(int irq, void (*handler)());


extern void __rt_fc_socevents_handler();

```

```
46 #ifdef __riscv__
47
48 static inline int rt_irq_disable()
49 {
50   int irq = hal_irq_disable();
51   // This memory barrier is needed to prevent the compiler to cross the irq barr
52   __asm__ __volatile__ ("" : : : "memory");
53   return irq;
54 }
55
56 static inline void rt_irq_restore(int irq)
57 {
58   // This memory barrier is needed to prevent the compiler to cross the irq barr
59   __asm__ __volatile__ ("" : : : "memory");
60   hal_irq_restore(irq);
61 }
62
63 static inline int disable_irq(void)
64 {
65   return rt_irq_disable();
66 }
67
68 static inline void restore_irq(int irq_enable)
69 {
70   rt_irq_restore(irq_enable);
71 }
72
73
74 static inline void rt_irq_enable()
75 {
76   __asm__ __volatile__ ("" : : : "memory");
77   hal_irq_enable();
78 }
79
80 static inline void rt_irq_mask_set(unsigned int mask)
81 {
82 #if defined(ARCHI_CORE_RISCV_ITC)
83   // If not irq ID 26, remap 0-15 to 16-31
84   if (mask != 0x4000000) {
85                 hal_spr_read_then_set_from_reg(0x304, (mask<<16));
86         } else {
87                 hal_spr_read_then_set_from_reg(0x304, mask);
88         }
89 #endif
90 #if defined(ITC_VERSION) && defined(EU_VERSION)
91   if (hal_is_fc()) hal_itc_enable_set(mask);
92   else eu_irq_maskSet(mask);
```

```
93 #elif defined(ITC_VERSION)
94   hal_itc_enable_set(mask);
95 #elif defined(EU_VERSION)
96   eu_irq_maskSet(mask);
97   // This is needed on architectures where the FC is using an
98   // event unit as we use an elw instead of a wfi with interrupts disabled.
99   // The fact that the event is active will make the core goes out of elw
100  // and the interrupt handler will be called as soon as interrupts are enabled.
101
102  eu_evt_maskSet(mask);
103 #endif
104 }
105
106 static inline void rt_irq_mask_clr(unsigned int mask)
107 {
108 #if defined(ARCHI_CORE_RISCV_ITC)
109   // If not irq ID 26, remap 0-15 to 16-31
110   if (mask != 0x4000000) {
111                 hal_spr_read_then_clr_from_reg(0x304, (mask<<16));
112         } else {
113                 hal_spr_read_then_clr_from_reg(0x304, mask);
114         }
115 #endif
116 #if defined(ITC_VERSION) && defined(EU_VERSION)
117   if (hal_is_fc()) hal_itc_enable_clr(mask);
118   else eu_irq_maskClr(mask);
119 #elif defined(ITC_VERSION)
120   hal_itc_enable_clr(mask);
121 #elif defined(EU_VERSION)
122   eu_irq_maskClr(mask);
123   if (hal_is_fc()) eu_evt_maskClr(mask);
124 #endif
125 }
126
127 static inline void rt_irq_clr(unsigned int mask)
128 {
129 #if defined(ITC_VERSION)
130   hal_itc_status_clr(mask);
131 #elif defined(EU_VERSION) && EU_VERSION >= 3
132   eu_evt_clr(mask);
133 #endif
134 }
135
136 #else
137
138 int rt_irq_disable();
139
```

```
140 void rt_irq_restore(int irq);
141
142 void rt_irq_enable();
143
144 void rt_irq_mask_set(unsigned int mask);
145
146 void rt_irq_mask_clr(unsigned int mask);
147
148 #endif
149
150 #if 0
151 static inline int rt_irq_disable()
152 {
153   int core_id = hal_core_id();
154   int state = pulp_irq_mask_low_read(core_id);
155   pulp_irq_mask_low_set(core_id, 0);
156   // As we are deactivating the interrupts in a peripheral,
157   // we have to put some nops to make sure interrupts are really inactive
158   // when we execute the next code.
159   __asm__ volatile ("l.nop" :  :  : "memory");
160   __asm__ volatile ("l.nop" :  :  : "memory");
161   __asm__ volatile ("l.nop" :  :  : "memory");
162   __asm__ volatile ("l.nop" :  :  : "memory");
163   __asm__ volatile ("l.nop" :  :  : "memory");
164   return state;
165 }
166
167 static inline void rt_irq_restore(int state)
168 {
169   pulp_irq_mask_low_set(hal_core_id(), state);
170 }
171
172 static inline void rt_irq_enable()
173 {
174 }
175 #endif
176
177 /// @endcond
178
179 #endif
```

Program 7.2 *Rt_irq.h Header File*