B.TECH. PROJECT REPORT

On

Implementing Realtime Kernel in Rust Programming Language

BY

KANISHKAR JOTHIBASU

Discipline of Computer Science and Engineering,

INDIAN INSTITUTE OF TECHNOLOGY INDORE

NOVEMBER 2019

Implementing Realtime Kernel in Rust Programming Language

PROJECT REPORT

Submitted in partial fulfillment of the requirements for the award of the degrees

of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

Submitted by:

KANISHKAR JOTHIBASU, 160001028,

Discipline of Computer Science and Engineering,

Indian Institute of Technology, Indore

Guided by:

Dr. GOURINATH BANDA,

Associate Professor,

Computer Science and Engineering,

IIT Indore

INDIAN INSTITUTE OF TECHNOLOGY INDORE
November, 2019

CANDIDATES' DECLARATION

I hereby declare that the project entitled **"Implementing Realtime Kernel in Rust Programming Language"** submitted in partial fulfillment for the award of the degree of Bachelor of Technology in 'Computer Science and Engineering' completed under the supervision of **Dr. GOURINATH BANDA, Associate Professor, Computer Science and Engineering**, IIT Indore is an authentic work.

Further, I declare that I have not submitted this work for the award of any other degree elsewhere.

KANISHKAR JOTHIBASU
160001028

CERTIFICATE by BTP Guide

It is certified that the above statement made by the student is correct to the best of my knowledge.

Dr. GOURINATH BANDA,
Associate Professor,
Discipline of Computer Science and Engineering,
IIT Indore

PREFACE

This report on "Implementing Realtime Kernel in Rust Programming Language" is prepared under the guidance of Dr. GOURINATH BANDA, Associate Professor, Computer Science and Engineering, IIT Indore.

Through this report, I have tried to provide a detailed description of our approach to implementing Real-time Kernel in Rust programming language. I also have explained the reasons to choose Rust and how it helped develop a safe kernel. The *design implementation* decisions taken in the due course of the project have been explained.

I have tried my best to explain the proposed solution. The Kernel has been implemented and is open-sourced under MIT license.

ACKNOWLEDGEMENTS

I want to thank my B.Tech Project supervisor **Dr. GOURINATH BANDA** for his guidance and constant support in structuring the project and providing valuable feedback throughout the course of this project. His overseeing the project meant there was a lot that I learnt while working on it. I thank him for his time and efforts.

I am grateful to the Institute for the opportunity to be exposed to systemic research especially Dr. GOURINATH BANDA's Lab for providing the necessary hardware utilities to complete the project.

Lastly, I offer my sincere thanks to everyone who helped me complete this project, whose name I might I have forgotten to mention.

KANISHKAR JOTHIBASU

ABSTRACT

Software failure caused $1.7 trillion in financial losses in the year 2017. A large percentage of these bugs fall into the category of Memory and Concurrency bugs. Many programming languages address this issue by handling memory and concurrency safety through a runtime platform, but this adds significant execution overhead. Hence, the languages which are focused on performance tend to provide compiled binaries; thus, they do not have any mechanism to guarantee safety.

Rust is a programming language that offers safety and reliability without sacrificing runtime performance. It achieves this with the help of a powerful type system and static code analysis to ensure safety at compile-time. Rust enables developers to write Safe and Robust software with ease. The goal of this thesis is to develop a real-time kernel in rust. Real-time kernels are an integral part of embedded systems. They guarantee low interrupt latency and highly predictable task scheduling. They typically are a part of safety-critical real-time systems, which mandate reliability, safety, and predictability.

Contents

List of source codes

Chapter 1

Introduction

1.1 Real-time Kernels

A Real-time Kernel is a Kernel intended to serve real-time applications, i.e., they respond to an input/event and produce the result within a guaranteed time interval (deadline) [1]. They enable the development of the application split as individual tasks instead of a monolith. They also provide the infrastructure required for task scheduling, inter-task synchronization, and communication.

1.2 Embedded Hardware

An embedded system can be defined as a special type of computer system that executes some specific predefined programs which are generally used within an electro-mechanical system [10]. Industrial machines, toys, automobiles, etc. are some areas where Embedded systems are used extensively. At present, most of the Embedded systems used are Microcontroller-based systems.

1.2.1 Microcontrollers

Microcontrollers can be understood as Computers on a single chip. They are inexpensive, small, and low power consuming devices [10]. It operates on the data provided to it through the serial and parallel ports. It is controlled by software stored in the on-chip memory. It consists of a processor core, RAM, ROM, and I/O pins used to redirect I/O from sensors/actuators to the processor.

1.3 Advantages of Kernel-based Embedded Systems over the Monolithic implementations

In Microcontrollers, Monolithic implementations imply an infinite loop of instructions. In Real-time Kernels, there is no need to break functionality up into fragments to fit it

within the major and minor cycles, functionality can be executed at the appropriate rate, and sporadic events such as infrequent but short deadline interrupts can be dealt with efficiently. Applications using Real-time Kernels can make significantly more effective use of CPU time than Monoliths. Simple cyclic systems can waste as much as 20% of the CPU through over-sampling and provision for sporadic activities [3].

Real-time operating systems provide basic infrastructure to design the system as individual tasks that can communicate and synchronize amongst themselves. The essential services provided by the Real-time Kernel is listed below [5] :

- External event management
- Timing event management
- Task scheduling and management
- Resource management
- Task synchronization
- Task communication

1.4 Challenges of Real-time Kernels

Implementations of Real-time Kernels are very different from the desktop OS Kernels. They are intended to work on Microcontrollers, which have several constraints [3].

1.4.1 Memory Restrictions

Microcontrollers have two kinds of memory, Flash memory, and RAM. Flash memory is where the Code is stored, while the RAM is where the runtime data-structures are stored. Now the problem is both Flash and RAM are generally in KBs. For example, the board used for testing in this project has a Flash memory of 512 Kb and the RAM 128 Kb, which is very small compared to what general computers have. Thus, the binary size of the whole application (including the Kernel) must be lesser than the Flash memory. These constraints have to be addressed both during design and subsequent implementation phases.

1.4.2 Power Restrictions

Embedded systems are extensively used in battery-powered environments, in systems like TV remotes, Digital watches, Smartwatches, and IoT devices. Thus the end application (Including the Kernel) needs to be power efficient.

1.4.3 Data-structure Overhead: Kernel Data-structures

Due to the tight constraints in Microcontrollers, the Kernel data-structures need to be efficient in terms of memory and execution time. Thus, Real-time Kernel cannot afford to use complex data structures for its operation. They also cannot use Dynamic memory allocation. The issue with dynamic memory is that it is very slow compared to static memory [2]. Real-time Kernels cannot afford this overhead in its operation.

1.5 Advantages of Implementing Real-time Kernels in Rust Programming Language

Rust is a multi-paradigm system programming developed by Mozilla [7]. The language has been designed with performance and reliability in mind. It is a significant advancement in industrial programming languages due to its success in bridging the gap between low-level system programming and high-level application programming languages [**jung_Rustbelt:_2018**]. This success has ultimately empowered programmers to build reliable and efficient software with ease. The safety guarantees made by the Rust have been verified and proven to be correct by using formal methods [8].

Real-time Kernels are typically a part of other larger safety-critical systems [10]. The compile-time guarantees made by Rust make it a perfect fit for the development of Real-time Kernels.

1.5.1 Memory and Concurrency Safety

Memory and Concurrency Safety are typically guarantees made by high-level interpreted programming languages like Perl or Python. These languages achieve them with the help of a runtime platform, which dynamically handles the memory allocated on heap. This approach works well for application software. But System software cannot accept the large execution overheads, especially Real-time systems. Any standard benchmark proves how interpreted languages like Python and Perl are much slower compared to C and C++. But without a runtime, it's impossible to guarantee memory and concurrency safety.

The term memory safety stands for accessing and managing memory in a safe way [**oppermann_Rust_nodate**]. Figure 1.1 shows the share of vulnerabilities in the Linux Kernel.

All the bugs mentioned above other than the "others" section are safety bugs that Rust resolves at compile-time.

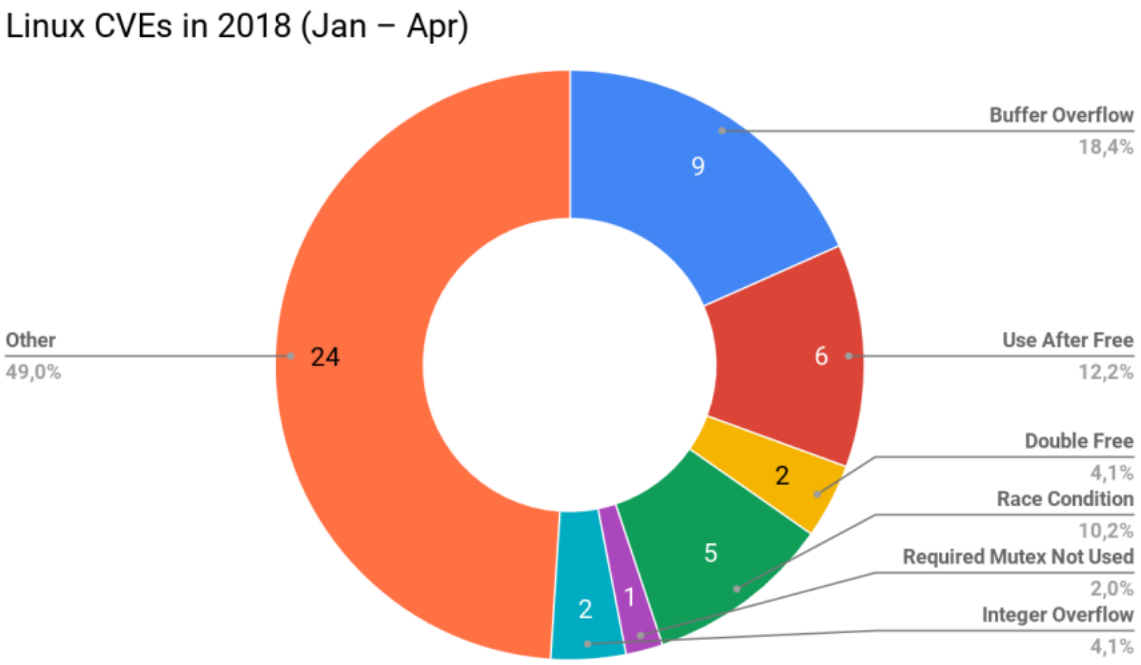

FIGURE 1.1: Linux Kernel CVEs 2018 **[oppermann_Rust_nodate]**

1.5.2 Easy Dependency management

Cargo is the Rust package manager. Cargo downloads the Rust project's dependencies, compiles your packages, makes distributable packages [14]. Dependencies or packages (called crates in Rust) used in a project can be defined in "Cargo.toml" in the project root directory, and Cargo takes care of Downloading, building, and linking the packages with application code irrespective of the target platform.

1.5.3 Performance

In performance benchmarks, idiomatic Rust code's execution time is on par with that of C/C++.

1.5.4 Unsafe Rust

While working with embedded hardware or writing bare-metal Code, many operations cannot be verified by the compiler. For example, explicitly writing to some registers, inlining assembly instructions, etc. But these features are necessary to implement most of the applications that interface with hardware directly. Rust allows writing unsafe blocks of code, inside which few compiler checks are overridden. Unsafe blocks help encapsulate unsafe Code in a program. They allow dereferencing raw pointers, writing

to registers, inlining assembly code, and calling other unsafe functions. These blocks are to be explicitly marked unsafe by the developer [19].

Unsafe blocks allow pointer manipulation, but the compiler still does various other static-analysis to ensure safety. If the unsafe blocks are written carefully and are verified to be correct, the compiler then assumes the unsafe blocks to be working correctly and verifies the rest of the Code for correctness.

Chapter 2

Kernel Architecture and Design

The Kernel Subsystems have been implemented according to the HARTEX design document [5]. The design is also influenced by the features provided by the processor and the software design ideas proposed by Rust. The Kernel has been designed with a focus on predictable execution, low latency, and compile-time Memory and Concurrency safety.

The uniqueness of this implementation in Rust is attributed to the following :

- Software architectural novelties (of HARTEX);
- Hardware features and
- Language features.

2.1 Kernel Architecture

The HARTEX design has been developed, keeping in consideration the tight CPU and memory constraints of Microcontrollers. The novelties in the Architecture are [5, Chapter 2] :

- Due to the usage of *BooleanVectors*, the Kernel does not use complex data-structure like queue or list;
- Scheduling, Software bus, and Resource management is implemented with *BooleanVectors*, which reduce the memory and performance overhead of the Kernel;
- Non-blocking Synchronisation and Communication between tasks are achieved through *BooleanVectors* semaphores;
- Event manager helps writing interrupt handlers with much lower execution times and
- Resource management via Stack-based priority ceiling protocol, which overcomes the problems of both deadlock and priority-inversion.

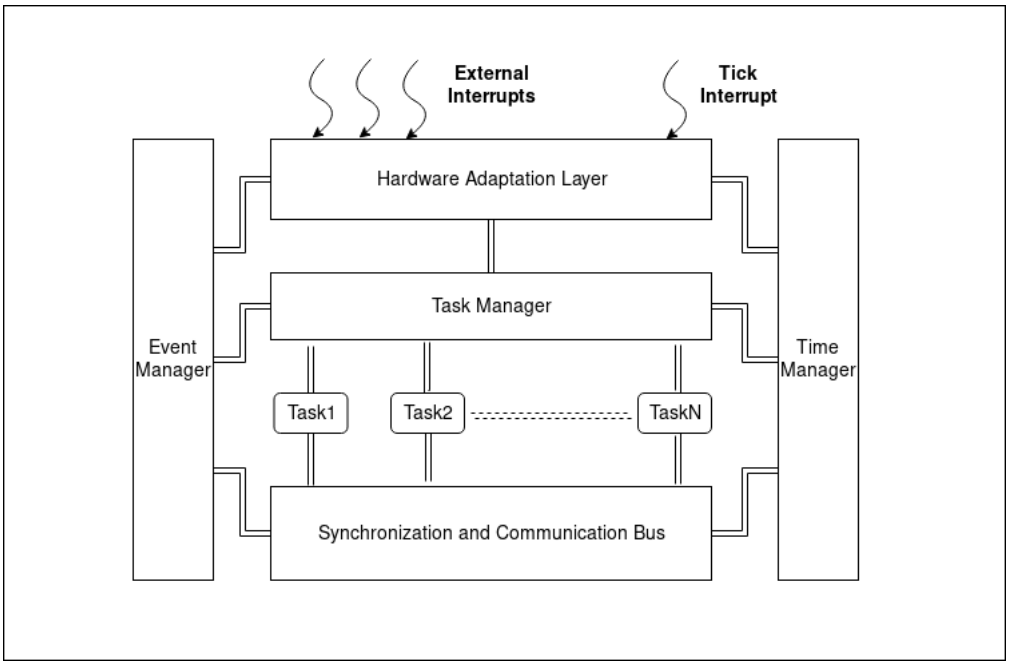

FIGURE 2.1: Kernel Organization

2.1.1 Top-level Organization

The Kernel is organized into various subsystems, as shown in Figure 2.1. Each module encloses the associated data-structures and routines in it. The Subsystems interact with each other via appropriate interfaces; they also provide certain public calls that can be invoked by the end-application [5].

2.1.2 Kernel Subsystems

The Kernel Subsystems are :

- Task Manager;
- Integrated Event Manager;
- Resource Manager;
- Software Bus and
- Hardware Adaptation Layer.

Task Manager

The end-application in real-time systems are composed of several individual tasks. In the implementation, the tasks are defined as functions and are passed on to the Kernel with their corresponding priority. The Task Manager schedules the tasks based on their priority [5]. On activation of a higher priority task, this module handles storing the context of the current task and loads the context of the highest priority task. The Scheduler developed in this project is preemptive and priority-based.

Multi-tasking in single-core processors is done by scheduling multiple tasks one by one on the CPU. A task can exist in one of the following four states [5]:

- Running: The task is currently running on the CPU.
- Ready: The task is currently not running but is ready to be executed.
- Preempted: When a higher priority task is released, the low priority task is preempted; and the high priority task starts execution.
- Inactive: The task has finished execution.

Resource Manager

In multitasking systems, the shared resources are to be used in an atomic fashion; this can be easily achieved by the use of Mutex. But Atomic execution of resources introduces Deadlocks. Deadlocks cannot be accepted in real-time systems as they put the whole system on a halt. HARTEX employs the *Stack-based priority ceiling protocol* (SBPC) [4], which ensures the atomic use of resources without running into deadlocks [5].

The Resource manager ensures atomicity by execution locking, i.e., When a resource is being locked, all competing tasks are blocked right away. These tasks are unblocked when the resource is unlocked [5].

Software Bus

Software bus handles task synchronization and Communication. The Kernel is designed to operate asynchronously. When a task requests for notification, the Kernel doesn't wait for the notification to be dispatched. Instead, it returns false if not available yet and returns the corresponding value in case the notification has arrived. Blocking the Kernel would affect the performance of the system on the whole.

Task Synchronization

Some tasks start or resume execution beyond a certain point (called a synchronization point) only if some other task has completed execution till a certain point. Synchronization is achieved by notifying the tasks on reaching the synchronization point. In HARTEX, it is achieved through semaphores.

Task Communication

Communication is similar to Synchronization. But in Communication, messages can be passed onto the other tasks at the synchronization points. HARTEX Messaging system is designed based on Content Oriented Messaging [5]. It implies that the details of the sender, receivers, and message buffer need not be specified every time to send a message, only the name of the message is required. The details of the sender, receivers, and

the message buffer are provided while initializing the Message primitive at configuration stage.

Time Manager

Time manager can be understood as the system clock. It is tightly coupled with the SysTick timer interrupt, which updates the milli-second field of the time manager, which simultaneously updates the seconds, minutes, and hours fields if required. It is used to dispatch events in the Event Manager, and also can be used by the tasks to get the time elapsed.

Event Manager

This module is used to dispatch events on the occurrence of interrupts and other Timing events [5]. The event manager recognizes interrupts via the Hardware Adaptation Layer (HAL). Every event specified has an event descriptor that specifies how the event should be handled. The event manager uses the event descriptor and notifies other Kernel Subsystems to take appropriate action.

Event Manager provides a significant improvement in end-application predictability. As the interrupt handler itself does not perform the task, the end functionality required is defined as a task. The interrupt handler just notifies the event manager, which further informs other subsystems; this keeps the execution time of interrupt handlers very low. During the execution of interrupt handlers, the occurrence of other interrupts is ignored. Hence, keeping light interrupt handlers increases the number of interrupts that can be serviced in a time interval.

Hardware Adaptation Layer (HAL)

This is the closest layer in the Kernel to the hardware. The purpose of the hardware abstraction layer is to allow upper layers of software to discover and use the hardware through an abstract API [5]. HAL makes it easy to define interrupt-handlers and to configure onboard GPIO pins, clocks, etc. Note that the HAL has not been implemented as a part of this thesis; instead, a standard HAL implementation for cortex-m based boards has been used[12].

2.1.3 Stack-Based Priority Ceiling Protocol (SBPC) [4]

The Kernel being static knows beforehand about all the tasks, resources, and the dependencies between them. SBPC takes advantage of this information to provide mutual exclusion between resources while simultaneously avoiding deadlocks [5, Chapter 6].

For deadlocks to occur, four conditions must be held simultaneously :

- Mutual Exclusion
- Cyclic Waiting
- No Preemption
- hold and wait

To avoid deadlocks, SBPC breaks the cyclic wait condition. The resources and tasks accessing them are known in advance. Every resource has a corresponding Resource Control Block (RCB), which holds the ceiling priority of the resource. The ceiling priority of a resource is the priority of the highest priority task that can access the resource. For example, let's say resource R1 is shared by five tasks with priorities 5, 3, 8, 2, 7 correspondingly. Then the ceiling priority of R1 is 8.

The highest ceiling priority of all the currently locked resources is stored in a variable called the System Ceiling.

Pi(t) stores the value of the System Ceiling at time *t*, *Pi_stack* stores the history of *Pi(t)*. It is initialized to a value Ω, which is a value lower than the priorities of all tasks (for example, -1). When all resources are free, $Pi = \Omega$. Each time a resource is allocated to a task, its ceiling priority is pushed onto the *Pi_stack*. Note that a request for a resource is only processed if the ceiling priority of the resource is greater than the System Ceiling. Tasks with priorities lesser than the System Ceiling at the point in time will not be executed even if the task manager has released them. In this way, cyclic waiting of resources can be avoided.

2.1.4 Dynamic Memory Allocation

The Kernel does not use dynamic data structures. But the end application might need dynamic data structures for providing the required functionality. Thus the Kernel supports an optional allocator which can be used to create dynamic data structures.

2.1.5 Atomic execution in Single-processor multi-tasking systems

In Single processor systems, atomic execution is obtained by disabling interrupts before the critical section and enabling it after them. If interrupts are disabled, the CPU won't interfere with currently executing code, therefore achieving atomic execution [16].

2.2 Hardware

The Kernel implemented is designed for ARM-Based microcontrollers, specifically Cortex-M4 (Armv7E-M) based ones. The reason for choosing ARM is that it is the most popular and widely used embedded platform. It is mainly used in portable devices due to

its low power consumption and reasonably good performance [6]. It's used in smartphones, tablets, multimedia devices, wearables, etc.

The Kernel implemented has been developed and tested on STM32f407VET6[17], which is a Cortex-M4 based microcontroller from STMicroelectronics. The specifications of the platform are as follows :

- Cortex M4 microcontroller (168 MHZ)
- 512K Flash memory
- 196K RAM
- Power Consumption: 238 µA/MHz at 168 MHz
- Up to 17 timers: 16- and 32-bit running

The Kernel implementation is not dependent on features provided by the microcontroller, but it uses the features provided by the Microprocessor. A few features of Cortex-M4 has been very instrumental in the Kernel Design.

2.2.1 Interrupt Handlers

Embedded platforms supports specifying various interrupt handlers; some of them can be configured to the developers' needs (external interrupts). The Kernel uses 3 interrupts for its operation : SysTick, PendSV, and SVC.

SysTick

SysTick is a timer that is a part of the Nested vector interrupt control (NVIC) controller in the Cortex-M microprocessor [6]. It is built and designed to provide a periodic interrupt for Kernels, but it can be used for other simple timing purposes. SysTick interrupt handler is used for time management, scheduling tasks regularly and for dispatching timing events. Note that by default, SysTick is the highest priority interrupt; hence the Time Manager of the Kernel will always stay in sync.

PendSV

PendSV interrupt is a software interrupt, i.e., it is raised by the application and not the hardware [6]. PendSV interrupt handler is used for context switch in the implementation. The reason for performing context switch in PendSV instead of doing it in the SysTick itself is to maintain a low response delay.

Note that Interrupts have priorities, SysTick has the highest priority while PendSV has the lowest while other interrupts like external interrupts lie in between. If an interrupt request IRQ, takes place before the SysTick exception, the SysTick exception will

preempt the IRQ handler. In this case, the OS should not carry out the context switching (Figure 2.2). Otherwise, the IRQ handler process will be delayed [6]. Another case that would cause a problem is when External interrupts are raised while the CPU was serving SysTick. Then the context switch would have occurred before servicing these external interrupts. But if context switch was done in PendSV, then all the external interrupts are serviced before performing the context switch. Thus, by using PendSV interrupts, the Kernel can guarantee better response time.

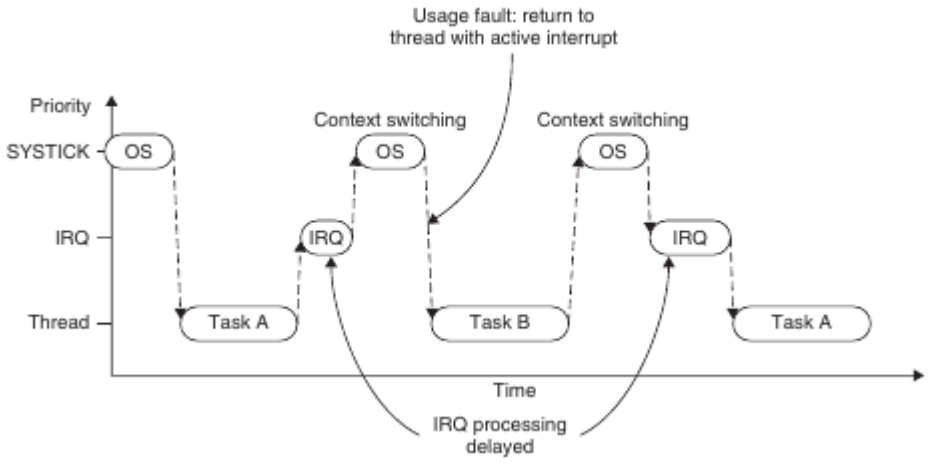

FIGURE 2.2: Illustration of PendSV interrupt [6]

SVC

SVC interrupt is a software interrupt. SVC is for generating operating system function calls. Instead of allowing user programs to access hardware directly, an operating system may provide access to hardware via an SVC interrupt [6]. In the Kernel developed, SVC is used for a few function calls which can only be executed in privileged mode. The unprivileged code creates the SVC interrupt, and the interrupt handler performs the privileged task.

2.2.2 Privileged execution mode

The Cortex-M4 processor has two operation modes and two privilege levels [6].

- The operation modes :
  - Thread mode: the processor is running a normal program.
  - Handler mode: the processor is running an exception handler like an interrupt handler or system exception handler.
- The privilege levels :

– privileged level and user level: it provides a mechanism for safeguarding memory accesses to important memory regions as well as provides a basic security model.

In thread mode, the CPU can either be in a privileged state or in the user state. But while handling exceptions/interrupts, the processor switches to the privileged state [6]. In the privileged state, a program has access to all instructions and registers and can write to any register. But in the case of user state, the program is constrained to ensure safety in the application. When an exception takes place, the processor always switches back to the privileged state and returns to the current state after exiting the exception handler [6].

In Cortex-M4, the following Registers cannot be accessed in the user state [12].

- BASEPRI
- CONTROL
- FAULTMASK
- MSP
- PRIMASK

The separation of privileged and user code improves the security of the system by preventing system configuration registers from being accessed or changed by user tasks.

The Kernel developed exploits this feature to ensure safety in the end application. All the tasks in the Kernel execute in the user-space. Even though many Kernel functions run in the user-space, a few essential Kernel routines can only be executed in the privileged space :

- Hardware Peripheral initialization and configuration;
- Kernel Initialization and setup;
- Tasks configuration;
- Declaration and Initialization of Kernel Primitives (Resources, Messages, Semaphores, and Events) and
- Context switch (which heavily influences the other Kernel modules).

If any of the Kernel routines mentioned are executed from the user level, the Kernel throws an error or halts at a hard fault.

In the case where a thread ends, there is a need to switch from user state to privileged state to start the next task. So this is solved by raising an interrupt(SVC) from the thread mode. When this is done, the processor shifts to the interrupt handler for SVC exception, which in turn does the context switch (as interrupts run in privileged mode) and loads the new task onto the board. Now when the processor returns to the user state, it executes the newly loaded process.

2.2.3 2 Stack Pointers

Cortex-M maintains two stack pointers: a Main Stack Pointer (MSP) and a Process Stack Pointer (PSP). By default, all privileged state (Kernel code and interrupt handlers) code runs on the MSP while the user state code (threads/tasks) runs on PSP [6].

The fact that the operating system and exception handlers use a different stack from the application means that the OS can protect its stack and prevent applications from accessing or corrupting it. Also, it ensures that the OS does not run out of stack if the application consumes all the available stack space. It means that there is always space on the stack to run an exception handler in the case of errors [6].

Similar to the Privileged execution mode, a significant chunk of the OS runs on the main stack.

2.2.4 CLZ Instruction

In the Task Management Module, the decision of the next task to be scheduled uses a function that calculates the Most Significant Bit (MSB) of the value passed to it. This function implemented algorithmically takes O(no_of_tasks) time to execute, which makes it dependent on the number of the tasks in the Kernel. But the goal of the Kernel implementation is to have the all Kernel routines as optimal as possible.

ARM support Count Leading Zeroes (CLZ) CPU instruction, which is faster than any scalable implementation to calculate MSB. The Kernel developed uses this instruction to calculate the MSB to ensure speed and scalability [6].

2.2.5 WFE Instruction

Consider a case in which all the tasks have finished executing. In this case, the Kernel has no task to schedule and just has to wait for some task to be released by an timing/external event. But until this, the Kernel will have to wait. Thus, the Kernel itself defines a task with zero priority called the Idle task. This task just loops and does nothing productive. Thus, whenever the Kernel runs out of tasks to schedule, Idle task is scheduled. But, infinite looping means the CPU is never free. Which directly affects the power consumption of the embedded device on the whole.

ARM supports Wait For Event (WFE) CPU instruction, which on execution puts the CPU to sleep until the occurrence of the next event. The Idle task executes the WFE instruction in a loop. So if there is no task to be scheduled, the Idle task puts the CPU to sleep until the next event. This reduces the power consumption considerably.

[6].

2.3 Language Features

2.3.1 Generics

Generics are used for defining generalized function/structs/types. It reduces code duplication and helps the compiler to ensure type correctness [15].

Listing 1 Rust Lang : Generics

```
struct Sample<T> {
    field1: T,
    field2: T,
}

let x = Sample {field1 : 1, field2 : 5}
let y = Sample {field1 : "hello" , field2 : "all"};
```

In listing 1 Sample has been defined only once, x and y hold instances of Sample with different types in it. But note that in this case, Sample::field1 and Sample::field2 have to be of the same type as they are defined to be of the same Generic Type T. This is how Rust enforces type safety statically. For example, Listing 2 would not compile because *Sample::field1* and *Sample::field2* have to be of different types.

Listing 2 Rust Lang : Generics Example

```
let x = Sample {field1 : 1, field2 : "hello" }
```

In Listing 3, Here *Sample::field1* will be of type T1 and *Sample::Field2* of type T2.

Listing 3 Rust Lang : Generics with Multiple Type parameters

```
struct Sample<T,U> {
    field1: T,
    field2: U
}
```

Similarly, Generic types can also be used to define functions :

Listing 4 Rust Lang : Generic Functions

```
fn generic_function<T> (a: T, b: T) {

}
```

2.3.2 Enums

Rust enums are very similar to C/C++ enums (Listing 5). They help define types by enumerating its possible values [13].

Listing 5 Rust Lang : Enums

```
enum Car {
    Sedan,
    Hatchback,
    SUV
}
```

The developer can define an instance of the values in an Enum, like in Listing 6.

Listing 6 Rust Lang : Enums example

```
let x = Car::Sedan;
let y = Car::Hatchback;
```

A feature that makes Rust enums unique and robust is that the enums can hold values. This allows definition of more powerful types like in Listing 7.

Listing 7 Rust Lang : Enums Example

```
enum IpAddress {
    IpV4(u8, u8, u8, u8),
    IpV6(String)
}

let ip1 = IpAddress::IpV4(192,198,1,1);
let ip2 = IpAddress::IpV6("2001:db8::8a2e:370:7334");
```

2.3.3 Error Handling

Generics and Enums give birth to two powerful types, which makes compile-time error handling and other static checks very robust. The Rust library defines two Enums, *Option* and *Result* (Listing 8.

Listing 8 Rust Lang : Error Handling

```
enum Option<T> {
    Some(T),
    None
}

enum Result<T,E> {
    Ok(T),
    Err(E)
}
```

The Option type encodes the very common scenario in which a value could be something or nothing. The Result Enum helps represent cases where a function might not succeed, so the return value will either contain the value or the Error. Expressing this concept in terms of the type system means the compiler can check whether the developer has handled all the cases that should be handled; this functionality can prevent bugs that are extremely common in other programming languages [13]. Option and Result are better implementations of Null pointers and try-catch.

2.3.4 Traits

Traits in Rust are very similar to Java interfaces. It is a language feature that tells the compiler about the functionality a type must provide [18].

Traits allow a type to make certain promises about their behavior. Generic functions can exploit this to specify constraints on the types they accept. Hence the compiler can now be sure that the function arguments will definitely provide the functionalities that the function expects from it.

Listing 9 Rust Lang : Traits

```
1 trait HasArea {
2     fn area(\&self) -> f64;
3 }
4
5 fn print_area <T: HasArea> (shape: T) {
6     println!("This shape has an area of {}", shape.area());
7 }
```

In Listing 9 though *print_area* doesn't know what would be the type of *shape*. The compiler can be sure it implements *HasArea* trait; hence, area() will be definitely defined by the type *T*.

An Important trait that the Kernel implementation uses is the Sync trait. Any type that implements the Sync trait is understood to be safe to be shared across tasks. The compiler ensures that variables shared across tasks or between tasks and interrupt handlers implement the Sync trait.

Chapter 3

Implementation

The Kernel internally uses only statically allocated structures to guarantee performance. Thus, the configuration variables like maximum no. of tasks, resources, etc. are defined as constants. The Kernel itself is developed as an independent rust library.

Every Kernel Subsystem requires the definition of a few data-structures; these definitions reside in a separate folder. The Kernel routines and primitives are dependent on these data-structures; thus, they are imported and initialized in the file in which the routines are defined. Appendix A Explains the Kernel Code structure.

3.1 Kernel Configuration

Rust allows developers to compile code based on flags passed to the compiler [11]. This feature has been used to provide the configurability in the Kernel. In *Cargo.toml*, along with the package version, the developer can mention additional feature flags; these are used by the library to enable and disable some particular features.

Listing 10 Cargo.Toml for Max 8 tasks.

```
1 hartex-rust = { version = "0.3.0", features=["tasks_8"] }
```

In Listing 10, adding the *tasks_8* feature flag sets the max tasks in the Kernel to 8. Feature flags allow configuration of the Kernel at the compile time without having to tinker with the source code. Developers can also specify multiple feature flags.

Listing 11 Cargo.Toml for Max 8 tasks and 16 resources.

```
1 hartex-rust = { version = "0.3.0", features=["tasks_8", "resources_16"]
  ↪ }
```

Listing 11 Will allocate Kernel data-structures to store at max 16 resources and eight tasks.

Listing 12 Cargo.Toml with conflicts.

```
1 hartex-rust = { version = "0.3.0", features=["tasks_8", "tasks_16"] }
```

Specification according to Listing 12 fails to compile as it is conflicting.

By default, the Kernel supports 32 Messages, Semaphores, Events, Tasks and Resources. The supported feature flags are :

- tasks_8
- tasks_16
- tasks_32 (default)
- resources_16
- resources_32 (default)
- resources_64
- events_16
- events_32 (default)
- events_64
- Semaphores_16
- Semaphores_32 (default)
- Semaphores_64

3.2 Boolean Vector

A Boolean Vector internally translates to a 32-bit integer. Each bit corresponds to the task with TaskID as the position of the bit. Boolean Vectors are used to represent *blocked_tasks* and *active_tasks*.

To understand better, consider Boolean Vectors with only 8 bits. Let's say *active_tasks* is 0b10010101; this implies that Tasks with TaskIds 0,3,5,7 are active tasks. Let *blocked_tasks* be 0b00010001; this means that Tasks with TaskIds 3,7 are blocked tasks.

We can clearly see how this is much more optimal than other Kernel implementations that use queues and lists. For example, for enabling or disabling tasks, the Kernel will only have to set/un-set bits in *active_tasks*.

The Thesis will only discuss the Kernel routines and primitives. For Implementation details, take a look at the API documentation and source code. Links to the same are available in Appendix A.

3.3 Task Manager

3.3.1 Subsystem Data-structures

Listing 13 Task Manager Definition

```
pub struct TaskControlBlock {
    pub sp: usize,
}

pub struct Scheduler {
    pub curr_tid: usize,
    pub is_running: bool,
    pub task_control_blocks: [Option<TaskControlBlock>; MAX_TASKS],
    pub blocked_tasks: BooleanVector,
    pub active_tasks: BooleanVector,
    pub is_preemptive: bool,
    pub started: bool,
}
```

- *TaskControlBlock* : A single tasks's state.
  - *TaskControlBlock::sp* : Holds a reference to the stack pointer for the task.
- Scheduler : Maintains state of all tasks in the Kernel.
  - *Scheduler::curr_tid* : The TaskId of the currently running task.
  - *Scheduler::is_running* : True if the scheduler has started scheduling tasks on the CPU.
  - *Scheduler::task_control_blocks* : An Array of TaskControlBlocks corresponding to each task.
  - *Scheduler::active_tasks* : A boolean vector in which, if a bit at a position is true, it implies that the task is active and to be scheduled.
  - *Scheduler::blocked_tasks* : A boolean vector in which, if a bit at a position is true, it implies that the task is blocked and cannot be scheduled even if it's active.
  - *Scheduler::is_preemptive* : A variable which decided if the scheduler should preemptively schedule tasks or not.
  - *Scheduler::started* : Set true as soon as the first task is scheduled.

3.3.2 Kernel Routines

init(is_preemptive: bool)

Initializes the Kernel scheduler. *is_preemptive* defines if the Kernel should operating preemptively or not. This method sets the *is_preemptive* field of the Scheduler instance and creates the idle task. The idle task is created with zero priority; hence, it is only executed when no other task is in Ready state.

start_Kernel(&self, peripherals: &Peripherals, tick_interval: u32)

Starts scheduling tasks on the system. It also starts the SysTick timer using the reference of the *Peripherals* instance and the *tick_interval*. *Peripherals* is a struct which provides abstract APIs to interact with the hardware. *tick_interval* specifies the frequency of the timer interrupt. The SysTick interrupt updates the Kernel regarding the time elapsed, which is used to dispatch events and schedule tasks.

create_task(&self, priority: usize, stack: &mut [u32], handler_fn: fn(&T) -> !, param: &T)

The program counter for the task is the value of *handler_fn*, which is a function pointer. *param* is a variable whose reference will be made accessible to the task; this helps in sharing variables with other tasks. Both these values are stored in a specific index of the stack . When the context switch routine loads the stack for this task, the appropriate program counter and argument for that function are loaded by the CPU.

An important thing to note is that the task's index in the *task_control_blocks* is the priority of the task. Hence there can be only one task of a priority. Also, another important thing is that the argument *param* is of a generic type *T*.

<T: Sync> informs the compiler that the type T must implement the Sync trait. By implementing the Sync trait, a type becomes safe to be shared across tasks. Hence if a type that doesn't implement Sync trait (like a mutable integer) is passed as *param*, then the code won't compile. This ensures that only concurrency-safe variables are shared between tasks. Kernel primitives like Message and Resource (which are data race safe) implement the Sync trait; hence, they can be passed as *param*.

handler_fn is of type *fn(&T) -> !*, which implies it is a function pointer which takes a parameter of Type *&T* and shall not return. The newly created TaskControlBlock is pushed onto *task_control_blocks*.

block_tasks(&self,task_mask: BooleanVector)

The Kernel blocks the tasks in *tasks_mask*. The bits set in *task_mask* are set in *Scheduler::blocked_tasks*.

unblock_tasks(&self,task_mask: BooleanVector)

The Kernel unblocks the tasks in *tasks_mask*. The bits set in *task_mask* are un-set in *Scheduler::blocked_tasks*.

schedule()

This function is called from both privileged and unprivileged context. If the function is called from privileged context, then *preempt()* is called. Else, *svc_call()* is called, which creates a SVC interrupt. And the SVC interrupt handler (privileged mode) calls schedule again. Thus, the permission level is raised via the interrupt handler.

preempt()

If the scheduler the highest priority task and currently running task aren't the same, then the PendSV interrupt is raised, this interrupt handler required context switch. In case the Kernel has just started and the Kernel has to dispatch its first task, it only loads the context, and does not store the current context. The task with the highest priority is evaluated by *get_next_tid()*.

task_exit()

The *task_exit* function is called just after a task finishes execution. This function un-sets the task's corresponding bit in the *active_tasks* and calls *schedule()*. In this way, the Kernel schedules the next highest priority task, which is in Ready state.

get_curr_tid()

Returns the TaskId of the currently running task (*curr_tid*) in the Kernel.

get_next_tid()

Returns the TaskId of the current highest priority task (in Ready state) in the Kernel. The task with the highest priority in Ready state is evaluated by calling *get_MSB(active_tasks & (blocked_tasks))*.

release(&self, task_mask: BooleanVector)

The Kernel releases the tasks in the *task_mask*, hence these tasks transition from the waiting to the ready state. The set bits in *task_mask* are set in *Scheduler::active_tasks*.

is_preemptive(&self)

Returns if the scheduler is currently operating preemptively or not.

enable_preemption(&self)

Enables preemptive scheduling.

disable_preemption(&self)

Disables preemptive scheduling.

spawn!

According to *create_task(...)*, the tasks must be looping infinitely. But, if the tasks run infinitely, then the scheduler will schedule only that task. Hence the task is defined as the work that will be done in one iteration of the *handler_function* passed on to *create_task(...)*. *task_exit()* must be called at the end of every iteration. The spawn macro makes it easier to define tasks; its expansion takes care of these constraints. It also defines a static variable of type TaskId, which corresponds to the task created.

3.3.3 Task State transition diagram

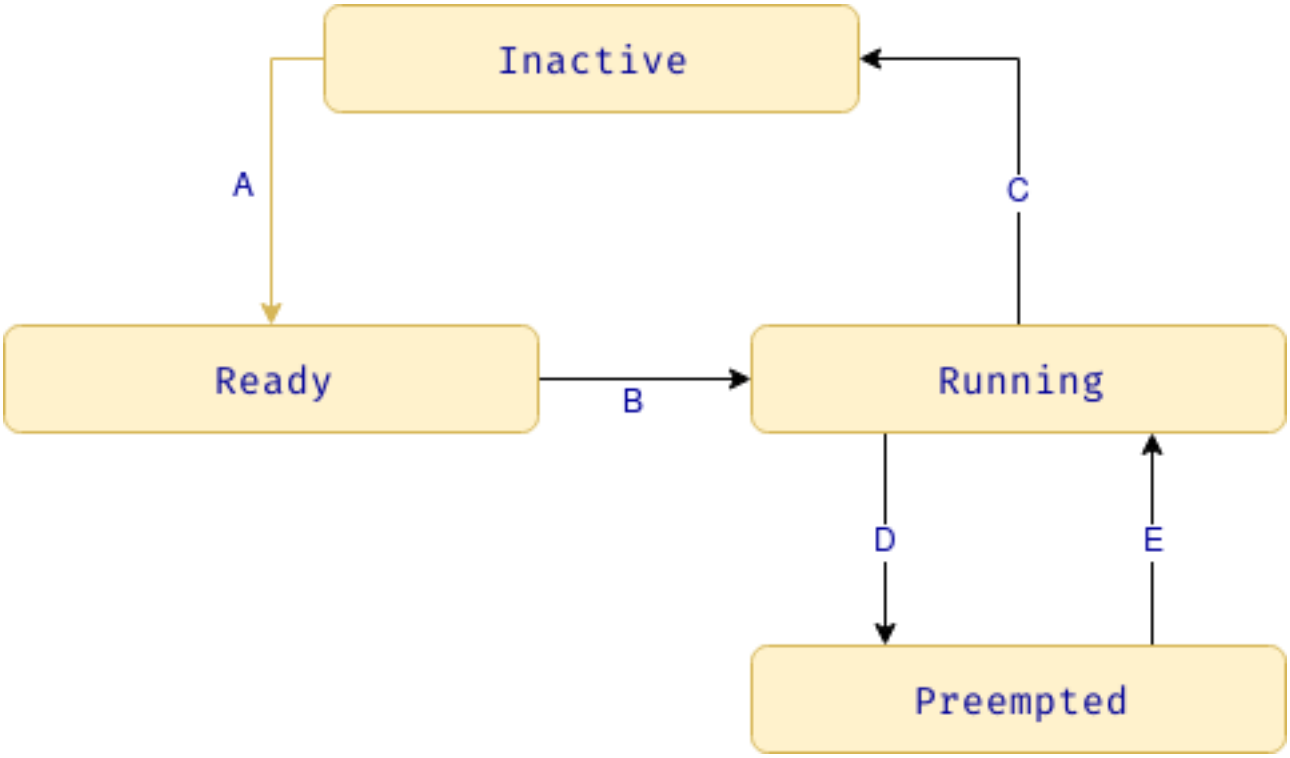

FIGURE 3.1: Task State transition Diagram

The following methods can cause the corresponding transitions

- A : *release()*
- B : *schedule()*
- C : *task_exit()*
- D : *block_task()* and *schedule()* (when a higher priority task is released)
- E : *unblock_task()* and *schedule()* (when all higher priority tasks are preempted)

3.4 Resource Manager

3.4.1 Subsystem Data-structures

Listing 14 Resource Manager Definition

```
const PI: i32 = -1;

pub struct ResourceControlBlock {
    ceiling: TaskId,
    tasks_mask: BooleanVector,
}

pub struct ResourceManager {
    resource_control_blocks: [Option<ResourceControlBlock>;
    ↪ MAX_RESOURCES],
    top: usize,
    pi_stack: [i32; MAX_RESOURCES],
    curr: usize,
    system_ceiling: i32,
}
```

- *ResourceControlBlock* : Describes a single Resource
  - *ResourceControlBlock::tasks_mask* : An boolean vector holding which tasks have access to the resource.
  - *ResourceControlBlock::ceiling* : It holds the priority of the highest priority task that can access that resource.

- *ResourceManager* : Manages the state of all Resources and notifies the task manager regarding blocking and unblocking of tasks.
  - *ResourceManager::resource_control_blocks* : An Array of ResourceControlBlocks.
  - *ResourceManager::curr* : Next empty index in the array where the new ResourceControlBlock can be stored.
  - *ResourceManager::pi_stack* : This stack is used for locking and unlocking of resources.
  - *ResourceManager::top* : Points the top of the pi_stack.
  - *ResourceManager::system_ceiling* : Hold the ceiling of the resource with the highest ceiling amongst the currently locked resources.

In the case where the resource locking/unlocking is being performed outside user tasks (privileged mode), the priority is considered to be 0. This helps as the Kernel would want to ensure that the resource can be accessed in interrupts and main thread only if no actively running task is holding it locked (as 0 is the minimum task priority). Also, 0 priority is assigned to the idle task. The idle task is defined by the Kernel such that it doesn't access any resources; thus, priority 0 can be used for Interrupt handlers without any conflicts.

The ResourceId of a Resource is its index in the *resource_control_blocks*.

3.4.2 Resource Container

Resource Container is a concurrency-safe Kernel primitive to manage the state of each resource individually. It uses a global instance of ResourceManager to handle locking and unlocking of the resources. It provides a wrapper around any generic type and makes it safe to be shared across threads. It guarantees not only atomic access to resources but also deadlock-free resource allocation, which is a stronger guarantee.

Listing 15 Resource Container Definition

```
struct Resource<T> {
    inner: T,
    id: ResourceId,
}
```

- *Resource::inner* : This field holds the actual resource that has to be locked.
- *Resource::id* : Holds the ResourceId allotted by ResourceManager for this resource.

Resource::lock(&self)

Called to lock the resource; once locked, until unlocked, no other task can access this resource. All tasks which are competing for this resource are locked to avoid data races and deadlocks. The competing tasks for a resource are defined as the tasks which have lower priority than the *ceiling* of the Resource.

Resource::unlock(&self)

Called to unlock the resource. It unblocks the tasks which were blocked while locking this resource.

Resource::acquire(&self, handler: F(&T))

Acquire is a helper function that ensures that if a resource is locked, it is unlocked too. It also ensures that the resources are unlocked in the order in which they were locked [5]. It takes one argument, which is function closure, that takes a parameter of type *&T*. If the resource is free, then the closure is executed with *inner* as the parameter.

Resource::access(&self)

There might be cases where the variable has to be accessed without locks. This function is used to access the resource bypassing the locking system, and it returns a reference to *inner*. This function is explicitly marked unsafe.

3.4.3 Kernel Routines

init_peripherals()

This function instantiates the *cortex_m::Peripherals* struct, wraps it in a resource container, and returns it. This peripheral instance is instrumental in configuring the GPIO pins on the board, clock, etc. it also has to be passed on to *start_Kernel()* function.

new(resource: T, tasks_mask: BooleanVector)

Instantiate a new Resource. Also, one important thing is that once a variable is passed onto this function, the variable cannot be accessed in the scope from where it was passed. This function takes ownership of the variable. It returns a resource instantiated with the value. Hence ensuring the value cannot be accessed outside the Resource container.

The 0th bit is set in *tasks_mask* so as to allow privileged code (all user tasks are unprivileged) to be able to access all the resource by default.

3.5 Software Synchronization Bus

3.5.1 SemaphoreControlBlock

Semaphores form the core of synchronization and communication in the Kernel.

Listing 16 SemaphoreControlBlock Definition

```
struct SemaphoreControlBlock {
    flags: u32,
    tasks: u32,
}
```

- *SemaphoreControlBlock::flags* : It is a boolean vector that represents the tasks that have to be notified by the semaphore.
- *SemaphoreControlBlock::tasks* : It is a boolean vector that corresponds to the tasks that are to be released by the semaphore on being signaled.

SemaphoreControlBlock::flags and *SemaphoreControlBlock::tasks* play very important roles in task synchronization. The tasks in the *tasks* field are released whenever the *signal_and_release()* call is made on the semaphore. *flags* is updated on every call to *signal_and_release()* with the parameter passed to it. Whenever *test_and_reset()* is called, it returns true if the TaskId of the current task is enabled in the *flags* field. After that, it un-sets the TaskId of in the *flags* field.

SemaphoreControlBlock::new(tasks_mask: u32)

Creates and returns a new semaphore instance with *tasks* field set to tasks_mask.

SemaphoreControlBlock::signal_and_release(&mut self, tasks_mask: BooleanVector)

This method, when called, appends the *tasks_mask* to the *flags* field. The tasks in the *tasks* field of the Semaphore are released.

SemaphoreControlBlock::test_and_reset(&mut self, curr_tid: TaskId)

This method returns true if the TaskId *curr_tid* is set in the semaphore's *flags* field and then un-sets it.

3.5.2 Subsystem Data-structures

Listing 17 SemaphoresTable Definition

```
struct SemaphoresTable {
    table: [SemaphoreControlBlock; SEMAPHORE_COUNT],
    curr: usize,
}
```

- SemaphoresTable::table : List of SemaphoreControlBlocks.
- SemaphoresTable::curr : Max index in table till which SemaphoreControlBlocks have been allotted.

The SemaphoreId of a Semaphore is its index in *SemaphoresTable::table*.

3.5.3 Kernel Routines

This module instantiates a global instance of SemaphoreTable and then defines functions which provide task synchronization functionality with its help.

new (tasks_mask: BooleanVector)

Creates a new SemaphoreControlBlock and returns its SemaphoreID. The newly create SemaphoreControlBlock is appended to *table* of SemaphoreTable.

signal_and_release(sem_id: SemaphoreId, tasks_mask: BooleanVector)

Calls the *signal_and_release()* method on the Semaphore with SemaphoreID as *sem_id*.

test_and_reset(sem_id: SemaphoreId)

Calls the *test_and_reset()* method on the Semaphore with SemaphoreID as *sem_id*.

3.6 Software Communication Bus

Inter-task communication also utilizes semaphores to release tasks and to keep track of the tasks which can access the message and the tasks that have to be notified about the arrival of messages.

3.6.1 Subsystem Data-structures

Listing 18 Message Table definition

```
pub struct MCB {
    pub receivers: BooleanVector,
}

pub struct MessagingManager {
    pub mcb_table: [Option<MCB>; MESSAGE_COUNT],
    pub scb_table: SemaphoresTable,
}
```

- MCB : Corresponds the details of a single message.
  - MCB::receivers : Boolean vector representing the receiver tasks.
- MessagingManager : The message table stores the metadata of messages, i.e., the receiver tasks and the tasks which to be released when the message has been dispatched.
  - *MessagingManager::mcb_table* : This array stores the MCB corresponding to each message.
  - *MessagingManager::scb_table* : SemaphoresTable

3.6.2 Message Container

It holds a variable of a generic type so that any message can be stored in it. It implements the Sync trait making it safe to be shared across tasks. In the Messaging Subsystem, a global instance of MessageTable is defined and used by the Kernel primitives.

Listing 19 Message Container Definition

```
struct Message<T> {
    inner: T,
    id: MessageId,
}
```

- *Message::inner* : Holds the Message that has to be sent to the receiver tasks.

- *Message::id* : Holds the MessageId corresponding to the message, which will be used to notify and release the tasks on arrival of the message.

Message::new(val: T, id: MessageId)

Creates a new Message of type T with *message_id* as id and returns it.

Message::broadcast(&self, msg: T)

The value of the message is passed as an argument, which is stored in *inner*. Then the updated message is broadcast, which notifies and releases tasks correspondingly. Its called by the sender task.

Message::receive(&self)

The receiver task calls this function. It returns the message value if the message is available for being read, else returns None.

3.6.3 Kernel Routines

new(notify_tasks_mask: BooleanVector, receivers_mask: BooleanVector, msg: T)

Creates a new message container using the parameters passed to it and returns the newly created message.

broadcast(msg_id: MessageId)

Broadcasts the message corresponding to *msg_id*. This function is needed only by event manager.

3.7 Time management

3.7.1 Subsystem Data-structures

Listing 20 Time Manager Definition

```
struct Time {
    m_sec_10: u32,
    sec: u32,
    min: u32,
    hour: u32,
    day: u32,
}

enum TickType {
    MilliSec10,
    Sec,
    Min,
    Hour,
    Day,
}
```

- *Time* : This struct represents a time object.
- *TickType* : This enum represents the highest order time that elapsed in a tick. The tick function returns TickType.

3.7.2 Kernel Routines

Time::tick()

A tick updates the Time object's *m_sec_10* field, which implies 10 MilliSecond has passed, and after every tick, the other fields (Second, Minutes, Hour, and Day) of the Time object are also updated if required. The tick method is called by the SysTick interrupt handler, and the return value is used by the interrupt handler to dispatch events.

Note that if the returned TickType is Hour, it not only implies the current tick caused completion of an hour, but that it caused completion of an Hour, Second, and 10 MilliSecond.

now()

Returns the time object, which corresponds to the uptime of the Kernel.

3.8 Event Manager

3.8.1 Subsystem Data-structures

Listing 21 Event Manager Definition

```
pub enum EventType {
    FreeRunning,
    OnOff,
}

pub enum EventTableType {
    MilliSec,
    Sec,
    Min,
    Hour,
    OnOff,
}

pub struct EventIndexTable {
    table: [usize; EVENT_INDEX_TABLE_COUNT],
    curr: usize,
}

pub struct EventManager {
    event_table: [Event; EVENT_COUNT],
    curr: usize,
    ms_event_table: EventIndexTable,
    sec_event_table: EventIndexTable,
    min_event_table: EventIndexTable,
    hr_event_table: EventIndexTable,
    onoff_event_table: EventIndexTable,
}
```

- *EventTableType* : Events are executed at multiples of different time units. For example, an event can be dispatched once every 40 milliseconds, or 50 seconds, etc. This enum represents each such time units. Note that this includes OnOff also as OnOff events do not belong to any of the time units; hence, it is given as a separate field.
- *EventIndexTable* : An EventIndexTable is created for each of the elements of EventTableType.
  - *table*: Holds the list of EventIds of events that belong to the particular EventTableType this EventIndexTable belongs to.
  - *curr*: Index to the next free location in the table.
- *EventManager* : Holds and Implements all Event management and dispatch functions.
  - *event_table* : This array holds the Event descriptors of all events.
  - *curr* : Points to the current empty slot in the event_table.
  - *ms_event_table* : An instance of EventIndexTable which holds list of EventIds of type EventTableType::MilliSec.
  - *sec_event_table* : An instance of EventIndexTable which holds list of EventIds of type EventTableType::Second.
  - *min_event_table* : An instance of EventIndexTable which holds list of EventIds of type EventTableType::Minute.
  - *hr_event_table* : An instance of EventIndexTable which holds list of EventIds of type EventTableType::Hour.
  - *onoff_event_table* : An instance of EventIndexTable which holds list of EventIds of type EventTableType::OnOff.

3.8.2 Event Descriptor

Listing 22 Event Descriptor Definition

```
pub enum EventType {
    FreeRunning,
    OnOff,
}

pub struct Event {
    is_enabled: bool,
    event_type: EventType,
    threshold: u8,
    counter: u8,
    opcode: u8,
    semaphore: SemaphoreId,
    tasks: BooleanVector,
    msg_index: MessageId,
    next_event: EventId,
}
```

- *EventType* : It is an Enum that represents if an Event is of type FreeRunning or OnOff.
  - *FreeRunning* : Represents events that repeatedly occur after a particular time threshold.
  - *OnOff* : Represents events that are dispatched once and are then disabled. They are executed later only if Some task enables it explicitly.
- *Event* : This object corresponds to an event descriptor.
  - *is_enabled* : If the event is currently enabled or not.
  - *event_type* : Is the event OnOff or FreeRunning.
  - *threshold* : The frequency (of time unit in which it belongs to) in which the Event should run.
  - *counter* : The current time elapsed. On reaching the value of the threshold, it is reset to zero, and the Event is dispatched.
  - *opcode* : This is a 4-bit code that corresponds to what are the operations that this event corresponds to.

- *semaphore* : Hold the SemaphoreId of the Semaphore that has to be signaled when the event is dispatched.
- *tasks* : Holds the BooleanVector of the tasks that have to be released or signaled (in case of semaphore event) when the event is dispatched.
- *msg_index* : Holds the MessageId of the message corresponding to the index.
- *next_event* : Hold the EventId of the next event that has to be triggered by this event.

Opcode

Opcode is a 4-bit code that represents what operations an event does on dispatch.

- 1 : *signal_and_release()* a semaphore.
- 1«1 : *broadcast()* a message.
- 1«2 : *release()* tasks.
- 1«3 : Set next event. Next event is the event that is dispatched by the current event on its dispatch.

It can be clearly seen how a single Opcode can end up doing multiple operations. For e.g., an opcode of 0b1010 would represent both signaling a semaphore and releasing tasks. Hence each Event descriptor holds fields corresponding to the parameters that the Opcode describes :

- *Semaphore* : SemaphoreId(Event::semaphore) and BooleanVector of tasks(Event::tasks)
- *Message* : MessageId(Event::msg_index)
- *Tasks* : BooleanVector of tasks(Event::tasks)
- *Next Event* : EventId(Event::next_event)

Note that In case where an event has to release a Semaphore and release tasks, the field *Event::tasks* is being used to describe both. *set_tasks* function is called to set the *Event::tasks*, and *set_semaphore* is called to set the *Event::semaphore* and *Event::tasks* fields. Thus, in the implementation of the functions configuring the Events, checks have been done to ensure no double writes. If *set_tasks* and *set_semaphore* are called for the same event, then the Kernel throws an error.

3.8.3 Kernel Routines

This Module instantiates a global instance of EventManager which is configured and used by the Kernel to manage Events.

sweep_event_table(event_type: EventTableType)

Dispatches all the events with EventTableType as *event_type*. For a FreeRunning event, when an event is dispatched its *counter* value is reduced by 1. If *counter* reaches zero then it is reset to *threshold* and the operations mentioned in its Opcode are performed.

enable_event(event_id: EventId)

This function is used to enable events if disabled. Useful for dispatching OnOff type events.

new_FreeRunning (is_enabled: bool, threshold: u8, event_counter_type: EventTable-Type)

Creates a new Event of type EventType::FreeRunning.

new_OnOff(is_enabled: bool)

Creates a new Event of type EventType::OnOff.

set_semaphore(event_id: EventId, sem: SemaphoreId, tasks_mask: BooleanVector)

Configure the event with EventId as *event_id* for signalling a semaphore on dispatch. The *Event::semaphore* field of Event is set to *sem*, and *Event::tasks* to *tasks_mask*.

set_tasks(event_id: EventId, tasks_mask: BooleanVector)

Configure the event with EventId as *event_id* for releasing tasks in the *tasks_mask* Boolean-Vector. The *Event::tasks* field of Event is set to *tasks_mask*.

set_message(event_id: EventId, msg_id: MessageId)

Configure the event with EventId as *event_id* to broadcast the message with MessageId as *msg_id*. The *Event::msg_index* field of Event is set to *msg_id*.

set_next_event(event_id: EventId, next_event: EventId)

Configure the event with EventId as *event_id* to dispatch the event with EventId as *next_event*. The *Event::next_event* field of Event is set to *next_event*.

3.8.4 Utils

generate_task_mask(tasks: &[u32])

A helper function that generates a BooleanVector corresponding to the array of TaskIds passed as an argument to it. It returns the BooleanVector. This function is helpful while initializing the Kernel tasks and primitives.

is_privileged()

It plays a significant role in the Permission management. It is written using inline-assembly. It reads the access level by reading the Control register of the CPU; returns true if code is currently running privileged context, else returns false.

get_msb(val: u32)

This function is called very frequently in the Kernel. Hence, it is important that it is as optimal as possible to keep the Kernel routines optimal. Most Machine architectures provide a CPU instruction called CLZ (Count leading Zeroes), this instruction returns the no. of zeroes before the first set bit. This instruction has been used to calculate the MSB of a *val*. This function is also written in inline-assembly; it returns the position of the Most significant bit.

3.9 Dynamic Memory Allocation

On enabling the *alloc* feature flag, the Kernel exports a module called *alloc*, which contains the definition of dynamic data structures. Note that if the *alloc* flag is not enabled, then the allocator code won't be compiled. The allocator has not been implemented as a part of this project, *alloc_cortex_m*[9] library has been used.

Listing 23 listing

```
1 hartex-rust = { version=0.0.5, features=["alloc"] }
```

3.10 Development Flow

The following flowchart explains how to write embedded applications and flash it on any microcontroller.

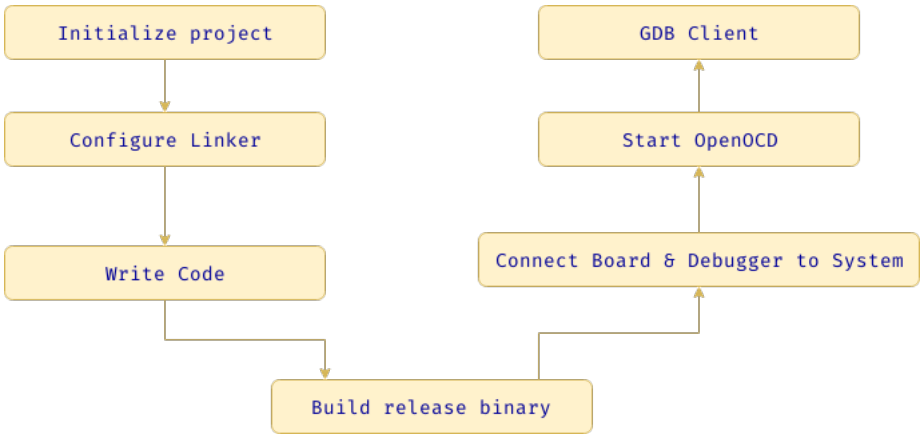

FIGURE 3.2: Development Workflow

3.11 End Application Organization (Using HARTEX)

The following flowchart depicts the general workflow of working with Hartex-rust on Cortex-M based Microcontrollers. Tooling and Development guide for Rust on Embedded can be found in the rust embedded book[16].

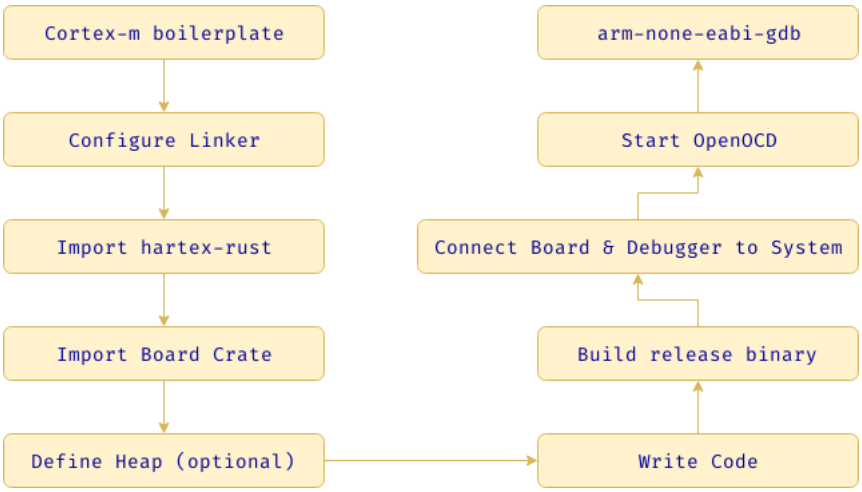

FIGURE 3.3: End Application Organization (Using HARTEX)

- *Cortex-m boilerplate* : Cortex-M projects require a few Linker scripts and other files to be able to compile for the specified machine architecure. Instead of Setting it up manually each time, developers can use this pre-configured repository.
- *Configure Linker* : In the Project Folder, consists of two configuration files: `memory.x` and `.cargo/config`. `memory.x` describes the memory layout of the device being used; this file is used by the Linker while Linking. The `.cargo/config` file describes the compilation target and other Linker configurations. Both these files have to be configured according to the Microcontroller's specifications.
- *Import hartex-rust* : Add `hartex-rust = *` to the Cargo.toml to import the latest version of Hartex-rust in the project.

- *Import Board Crate* : Almost Every Microcontroller has a Peripheral Access crate, which provides abstract APIs to access the board peripherals. Check this page to find the one for a specific board. Importing this crate in *main.rs* will configure the Interrupt vectors on the Microcontroller, else the project won't compile.
- *Define Heap* : Define memory for Heap in case the application uses dynamic data structures.
- *Write Code* : Develop the application.
- *Build Release binary* : Run `cargo build --release`, this will generate the compiled release binary of the application that can now be flashed on to the board.
- *Connect Board & Debugger to System* : Embedded boards require a debugger attached to the board to flash code onto it. After connecting the debugger to the board, connect the debugger to the computer.
- *Start OpenOCD* : OpenOCD is a tool for on-chip debugging. It allows the developer to flash code onto the board. After connecting the debugger to the computer, start openocd; it also instantiates a GDB server via which compiled binary can be loaded onto the board.
- *arm-none-eabi-gdb* : This tool connects with the GDB server started by OpenOCD. On passing it the path to the compiled binary, it flashes it onto the board.

Chapter 4

Testing

Initial implementation of the Kernel was tested on a Machine Architecture emulator called QEMU. Various examples depicting the kernel features have been implemented and tested for the correct output. Below is the code of the examples with the documentation, and output with the explanation.

4.0.1 Tasks (Basic)

This example demonstrates just simple tasks and checks if they execute in the expected order.

LISTING 24: Examples : Tasks (Basic)

```
#![no_std]
#![no_main]

extern crate panic_halt;
extern crate stm32f4;

use core::cell::RefCell;

use cortex_m::peripheral::Peripherals;
use cortex_m_rt::entry;
use cortex_m_semihosting::hprintln;

use hartex_rust::tasks::*;
use hartex_rust::util::generate_task_mask;
use hartex_rust::resources;
use hartex_rust::spawn;
use hartex_rust::types::*;

#[entry]
```

```
20 fn main() -> ! {
21     /*
22     Gets an instance of cortex-m Peripherals struct wrapped in a RefCell
   ↪  inside a Resource container.
23     Peripherals struct provides APIs to configure the hardware beneath.
24     RefCell is used to provide interior mutability read more at :
25     https://doc.rust-lang.org/book/ch15-05-interior-mutability.html
26     */
27     let peripherals: Resource<RefCell<Peripherals>> =
   ↪  resources::init_peripherals().unwrap();
28
29     /*
30     Define the task stacks corresponding to each task.
31     Note to specify the stack size according to the task parameters and
   ↪  local variables etc.
32     */
33     static mut stack1: [u32; 300] = [0; 300];
34     static mut stack2: [u32; 300] = [0; 300];
35     static mut stack3: [u32; 300] = [0; 300];
36
37     /*
38     Task definition.
39     The first parameter corresponds to the name that will be used to
   ↪  refer to the task.
40     The second variable corresponds to the priority of the task.
41     The third variable corresponds to the task stack.
42     The fourth variable corresponds to the task body.
43     */
44     spawn!(task1, 1, stack1, {
45         hprintln!("TASK 1");
46     });
47     spawn!(task2, 2, stack2, {
48         hprintln!("TASK 2");
49     });
50     spawn!(task3, 3, stack3, {
51         hprintln!("TASK 3");
52     });
53
```

```

    // Initializes the kernel in preemptive mode.
    init(true);

    // Releases tasks task1, task2, task3
    release(generate_task_mask(\&[task1, task2, task3]));

    /*
    Starts scheduling tasks on the device.
    It requires a reference to the peripherals to start the SysTick
    timer.
    150_000 corresponds to the tick interval of the SysTick timer.
    */
    start_kernel(
        unsafe { \&mut peripherals.access().unwrap().borrow_mut() },
        150_000,
    );

    loop {}
}
```

The Output of this code snippet is as follows :

```
TASK 3
TASK 2
TASK 1
```

The code runs as expected; the tasks are scheduled in decreasing order of priority.

4.0.2 Tasks (Complete)

The following code defines two tasks which take a parameter and print it to console.

LISTING 25: Examples : Tasks (Complete)

```
// Imports

#[entry]
fn main() -> ! {
    let peripherals = resources::init_peripherals().unwrap();
```

```
6
7       // These are task parameters, they are passed to the task when called
8       let task1_param = "Hello from task 1 !";
9       let task2_param = "Hello from task 2 !";
10
11      static mut stack1: [u32; 300] = [0; 300];
12      static mut stack2: [u32; 300] = [0; 300];
13
14      /*
15      The task definition here is different :
16      arg 1 : task name, this will be used to address the task across the
        ↪ code
17      arg 2 : priority of the task
18      arg 3 : task stack
19      arg 4 : this corresponds to by what name the task body will refer the
        ↪ task argument
20      arg 5 : the task argument
21      arg 6 : task body
22      */
23      spawn!(task1, 1, stack1, param, task1_param, {
24          hprintln!("{}", param);
25      });
26      spawn!(task2, 2, stack2, param, task2_param, {
27          hprintln!("{}", param);
28      });
29
30      init(true);
31      release(generate_task_mask(\&[task1, task2]));
32      start_kernel(
33          unsafe { \&mut peripherals.access().unwrap().borrow_mut() },
34          150_000,
35      );
36
37      loop {}
38  }
```

The Output :

```
Hello from task 2 !
Hello from task 1 !
```

The Tasks task1 and task2 take parameters *task1_param* and *task2_param*, respectively. The parameters are called as *param* in the task body. The output is as expected, according to the priority of the tasks.

4.0.3 Semaphore

The following code demonstrates using a semaphore and testing the working of methods `signal_and_release` and `test_and_reset`.

LISTING 26: Examples : Semaphores

```
// Imports

struct AppState {
    sem1: SemaphoreId,
    sem2: SemaphoreId,
}

#[entry]
fn main() -> ! {
    let peripherals = resources::init_peripherals().unwrap();

    /*
        Instance of AppState, whose reference will be shared with all tasks.
        sem1 is a Semaphore that releases task1 on being signaled, similarly sem2 releases task2.
    */
    let app_inst = AppState {
        sem1: semaphores::new(generate_task_mask(\&[task1])).unwrap(),
        sem2: semaphores::new(generate_task_mask(\&[task2])).unwrap(),
    };

    static mut stack1: [u32; 300] = [0; 300];
    static mut stack2: [u32; 300] = [0; 300];
    static mut stack3: [u32; 300] = [0; 300];

```

```
    spawn!(task1, 1, stack1, params, app_inst, {
        hprintln!("TASK 1: Enter");
        semaphores::signal_and_release(params.sem2,
↪   generate_task_mask(\&[task2]));
        hprintln!("TASK 1: End");
    });

    spawn!(task2, 2, stack2, params, app_inst, {
        hprintln!("TASK 2: Enter");
        if semaphores::test_and_reset(params.sem2).unwrap() {
            hprintln!("TASK 2: sem2 enabled");
        } else {
            hprintln!("TASK 2: sem2 disabled");
        }
        hprintln!("TASK 2: End");
    });

    spawn!(task3, 3, stack3, params, app_inst, {
        hprintln!("TASK 3: Enter");
        semaphores::signal_and_release(params.sem1, 0);
        hprintln!("TASK 3: End");
    });

    init(true);
    release(generate_task_mask(\&[task2, task3]));
    start_kernel(
        unsafe { \&mut peripherals.access().unwrap().borrow_mut() },
        150_000,
    );
    loop {}
}
```

The Output :

```
TASK 3: Enter
TASK 3: End
TASK 2: Enter
TASK 2: sem2 disabled
TASK 2: End
```

```
TASK 1: Enter
TASK 1: End
TASK 2: Enter
TASK 2: sem2 enabled
TASK 2: End
```

The output explained step by step :

- Tasks *task2* and *task3* are released initially; hence, the kernel first schedules *task3*.
- *task1* is released when *sem1* is signaled, but it won't affect the current execution as it has a lower priority.
- After *task3* exits, *task2* is scheduled. It calls *test_and_reset()* on *sem2*, but as *sem2* has not been signaled yet, it returns false. Hence "TASK 2: sem2 disabled" is printed onto the console.
- After *task2*, *task1* is scheduled. *task1* signals *sem2* while passing *task2* as the task to notify and then finishes its execution. Signaling *sem2* releases *task2*.
- *task2* will start its execution. But this time, as *task1* notified *task2*, *sem2* would be enabled. Hence *task2* will print "TASK 2: sem2 enabled" this time.
- No more task to schedule.

4.0.4 Resource

The following examples depict the use of Resource containers and releasing tasks before freeing the resource. Also, note that Resource Container implements the Sync trait as it is safe to be accessed across tasks. If *app_inst* uses any field that does not implement the Sync trait, then the code will not compile.

LISTING 27: Examples : Resource

```rust
// Imports
struct AppState {
    sem2: SemaphoreId,
    sem3: SemaphoreId,
    res1: Resource<[u32; 3]>,
    res2: Resource<[u32; 2]>,
}

#[entry]
fn main() -> ! {
    let peripherals = resources::init_peripherals().unwrap();
```

```
12
13      // app_inst also holds the resource containers res1 and res2.
14      let app_inst = AppState {
15          sem2: semaphores::new(generate_task_mask(\&[task2])).unwrap(),
16          sem3: semaphores::new(generate_task_mask(\&[task3])).unwrap(),
17          res1: resources::new([1, 2, 3], generate_task_mask(\&[task1,
    ↪ task2])).unwrap(),
18          res2: resources::new([4, 5],
    ↪ generate_task_mask(\&[task3])).unwrap(),
19      };
20
21      static mut stack1: [u32; 512] = [0; 512];
22      static mut stack2: [u32; 512] = [0; 512];
23      static mut stack3: [u32; 512] = [0; 512];
24
25      spawn!(task1, 1, stack1, params, app_inst, {
26          hprintln!("TASK 1: Enter");
27          // If res1 is free, then the closure passed on is executed on the
    ↪ resource.
28          params.res1.acquire(|res| {
29              hprintln!("TASK 1 : res1 : {:?}", res);
30              semaphores::signal_and_release(params.sem2, 0);
31              semaphores::signal_and_release(params.sem3, 0);
32              // emulates a long task so that in the next SysTick task3 is
    ↪ scheduled
33              for i in 0..10000 {}
34              hprintln!("TASK 1 : task 2 and 3 dispatched");
35          });
36          hprintln!("TASK 1: End");
37      });
38
39      spawn!(task2, 2, stack2, params, app_inst, {
40          hprintln!("TASK 2: Enter");
41          params.res1.acquire(|res| {
42              hprintln!("TASK 2 : res1 : {:?}", res);
43          });
44          hprintln!("TASK 2: End");
45      });
```

```
46
47      spawn!(task3, 3, stack3, params, app_inst, {
48          hprintln!("TASK 3: Enter");
49          params.res2.acquire(|res| {
50              hprintln!("TASK 3 : res2 :  {:?}", res);
51          });
52          hprintln!("TASK 3: End");
53      });
54
55      init(true);
56      release(generate_task_mask(\&[task1]));
57      start_kernel(
58          unsafe { \&mut peripherals.access().unwrap().borrow_mut() },
59          150_000,
60      );
61      loop {}
62  }
```

The Output :

```
1  TASK 1: Enter
2  TASK 1 : res1 : [1, 2, 3]
3  TASK 3: Enter
4  TASK 3 : res2 :  [4, 5]
5  TASK 3: End
6  TASK 2: Enter
7  TASK 2: End
8  TASK 1 : task 2 and 3 dispatched
9  TASK 1: End
```

- *task1* is the only task that is released initially.
- *task1* acquires resource *res1*; then, it signals *task2* (via *sem2*) and *task3*(via *sem3*), which releases them.
- Next is a loop which emulates a long task to emulate a race condition. Hence before *line no. 24* the scheduler schedules *task3*.
- *task3* tries to acquire resource *res2* and succeeds as no other task is holding it.
- After *task3* exits, *task2* is scheduled, which tries to acquire *res1*. But *res1* is still held locked by *task1*; hence, *task2* does not get hold of the resource.
- Then *task2* finished execution, and *task1* follows and finishes its unfinished code.

4.0.5 Message

The following example depicts the usage of Message Containers. Semaphores have been used to launch tasks to verify the functionality message.receive().

LISTING 28: Examples : Message

```
// Imports
struct AppState {
    sem3: SemaphoreId,
    msg1: Message<[u32; 2]>,
}

#[entry]
fn main() -> ! {
    let peripherals = resources::init_peripherals().unwrap();

    let app_inst = AppState {
        sem3: semaphores::new(generate_task_mask(\&[3])).unwrap(),
        msg1: messages::new(
            generate_task_mask(\&[task2]),
            generate_task_mask(\&[task2]),
            [9, 10],
        )
        .unwrap(),
    };

    static mut stack1: [u32; 300] = [0; 300];
    static mut stack2: [u32; 300] = [0; 300];
    static mut stack3: [u32; 300] = [0; 300];

    spawn!(task1, 1, stack1, params, app_inst, {
        hprintln!("TASK 1: Enter");
        params.msg1.broadcast(Some([4, 5]));
        semaphores::signal_and_release(params.sem3, 0);
        hprintln!("TASK 1: END");
    });

    spawn!(task2, 2, stack2, params, app_inst, {
        hprintln!("TASK 2: Enter");
```

```
        if let Some(msg) = params.msg1.receive() {
            hprintln!("TASK 2: msg received : {:?}", msg);
        }
        hprintln!("TASK 2: END");
    });

    spawn!(task3, 3, stack3, params, app_inst, {
        hprintln!("TASK 3: Enter");
        if let Some(msg) = params.msg1.receive() {
            hprintln!("TASK 3: msg received : {:?}", msg);
        }
        hprintln!("TASK 3: END");
    });

    init(true);
    release(generate_task_mask(\&[task1]));
    start_kernel(
        unsafe { \&mut peripherals.access().unwrap().borrow_mut() },
        150_000,
    );
    loop {}
}
```

The Output :

```
TASK 1: Enter
TASK 1: END
TASK 3: Enter
TASK 3: END
TASK 2: Enter
TASK 2: msg received : [4, 5]
TASK 2: END
```

- First, *task1* is executed; it broadcasts message *msg1* and releases *task3*.
- Note that *task3* is not in the receivers list of *msg1*. Hence when task3 calls `msg1.receive()` it does not return the message.
- Next, when *task2* is executed, it receives the message and prints it to the console.

4.0.6 Events

The following code snippet depicts the usage of the Event manager. The example depicts all type of events and all possible configurations that can be provided.

LISTING 29: Examples : Events

```
// imports
struct AppState {
    sem2: SemaphoreId,
    msg1: Message<[u32; 2]>,
}

#[entry]
fn main() -> ! {
    let peripherals = resources::init_peripherals().unwrap();

    let app_inst = AppState {
        sem2: semaphores::new(generate_task_mask(\&[task2])).unwrap(),
        msg1: messages::new(
            generate_task_mask(\&[task3]),
            generate_task_mask(\&[task3]),
            [9, 10],
        )
        .unwrap(),
    };

    /*
    Creates a FreeRunning Event that occurs once in every second.
    The event releases task1 when its counter expires.
    */
    let e1 = events::new_FreeRunning(true, 1,
↪   EventTableType::Sec).unwrap();
    events::set_tasks(e1, generate_task_mask(\&[task1]));

    /*
    Creates a FreeRunning Event that occurs once in every 2 seconds.
    The event signals semaphore sem2 when its counter expires.
    */
    let e2 = events::new_FreeRunning(true, 2,
↪   EventTableType::Sec).unwrap();
```

```
    events::set_semaphore(e2, app_inst.sem2, generate_task_mask(\&[task1,
↪   task2]));

    /*
    Creates an OnOff Event.
    The event broadcasts message msg1 whenever it is dispatched.
    */
    let e3 = events::new_OnOff(false).unwrap();
    events::set_message(e3, app_inst.msg1.get_id());

    /*
    Creates a FreeRunning Event that occurs once in every 3 seconds.
    The event dispatches event3 when its counter expires.
    */
    let e4 = events::new_FreeRunning(true, 3,
↪   EventTableType::Sec).unwrap();
    events::set_next_event(e4, e3);

    static mut stack1: [u32; 300] = [0; 300];
    static mut stack2: [u32; 300] = [0; 300];
    static mut stack3: [u32; 300] = [0; 300];

    spawn!(task1, 1, stack1, params, app_inst, {
        hprintln!("TASK 1: Enter");
        if let Ok(true) = semaphores::test_and_reset(params.sem2) {
            hprintln!("TASK 1: sem2 enabled");
        }
        hprintln!("TASK 1: End");
    });

    spawn!(task2, 2, stack2, params, app_inst, {
        hprintln!("TASK 2: Enter");
        if let Ok(true) = semaphores::test_and_reset(params.sem2) {
            hprintln!("TASK 2: sem2 enabled");
        }
        hprintln!("TASK 2: End");
    });

```

```
    spawn!(task3, 3, stack3, params, app_inst, {
        hprintln!("TASK 3: Enter");
        if let Some(msg) = params.msg1.receive() {
            hprintln!("TASK 3: msg received : {:?}", msg);
        }
        hprintln!("TASK 3: End");
    });

    init(true);
    release(task1);
    start_kernel(
        unsafe { \&mut peripherals.access().unwrap().borrow_mut() },
        150_000,
    );
    loop {}
}
```

The Output:

```
TASK 3: Enter
TASK 3: msg received : [9, 10]
TASK 3: End
TASK 1: Enter
TASK 1: sem2 enabled
TASK 1: End
TASK 1: Enter
TASK 1: End
TASK 2: Enter
TASK 2: sem2 enabled
TASK 2: End
TASK 1: Enter
TASK 1: sem2 enabled
TASK 1: End
... repeats
```

4.0.7 Heap

The following example depicts the usage of dynamic memory in the application.

LISTING 30: Examples : Heap

```
// Imports
/*
lazy_static is used to define global static variables.

Declaring variables in lazy_static can be useful while sharing kernel primitives to kernel tasks and interrupt
handlers. Resources can be shared with tasks as a parameter but interrupt handlers do not take parameters; hence
the only way to share data with them is via global statics.

The Resource res1 stores a resource of type Vec. Vec is a dynamic memory data structure.
*/
lazy_static! {
    static ref resource1: Resource<RefCell<Vec<u32>>> =
        resources::new(RefCell::new(Vec::new()), generate_task_mask(\&[1, 2])).unwrap();
}

#[entry]
fn main() -> ! {
    // Initialize heap for the application. The argument is the size of the heap.
    init_heap(50);
    let peripherals = resources::init_peripherals().unwrap();

    static mut stack1: [u32; 256] = [0; 256];
    static mut stack2: [u32; 256] = [0; 256];
    static mut stack3: [u32; 256] = [0; 256];

    spawn!(task1, 1, stack1, {
        hprintln!("TASK 1: Enter");
        resource1.acquire(|res| {
            let res = \&mut res.borrow_mut();
            res.push(1);
            hprintln!("TASK 1: Resource : {:?}", res);
        });
```

```
        hprintln!("TASK 1: End");
    });
    spawn!(task2, 2, stack2, {
        hprintln!("TASK 2: Enter");
        resource1.acquire(|res| {
            let res = \&mut res.borrow_mut();
            res.push(2);
            hprintln!("TASK 2: Resource : {:?}", res);
        });
        hprintln!("TASK 2: End");
    });

    init(true);
    release(generate_task_mask(\&[task1, task2]));
    start_kernel(
        unsafe { \&mut peripherals.access().unwrap().borrow_mut() },
        150_000,
    );
    loop {}
}
```

The Output :

```
TASK 2: Enter
TASK 2: Resource : [2]
TASK 2: End
TASK 1: Enter
TASK 1: Resource : [2, 1]
TASK 1: End
```

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The objectives with which the Kernel was designed and implemented are :

- Optimize Kernel operations with the help of simpler data-structures (Boolean Vectors).
- Compile-time Memory and Concurrency Safety.
- Avoiding Deadlocks and Priority-inversion.
- Ensure stronger real-time guarantees with the help of design and Hardware features.
- Modular and Scalable implementation.

All the Kernel subsystems discussed in 2.1 have been implemented and tested manually. The Kernel has been developed as a library and can be imported and used in any embedded application. The source code has been open-sourced under MIT license, and API documentation has been hosted online. Links to the same can be found in Appendix A. Some example applications were compiled, and the Binary size was noted down to be :

- 2 Basic tasks : 15 Kb
- Heap example : 23 Kb

5.2 Future Work

Hartex-rust has great potential for use in Embedded Systems. The future work on this project might include :

- Framework for Testing and Benchmarking.

- Language features like constant functions, constant generics, etc. are currently under heavy development. These features, once on reaching stabilization, can be used for further performance improvement of various kernel routines. These features enable compile-time evaluation of functions, which will help further boost the Kernel' performance.
- Support for additional machine architectures can be added with the help of Conditional Compilation (called feature-flags in Rust).
- Implementation of networking stack would enable the use of Hartex-Rust in IoT projects.
- The Kernel has been designed and developed for single-core/processor systems. The Major work on this project could include modifying the internals to work efficiently on multiprocessor systems.

Appendix A

Kernel Code

The Kernel Source code is hosted on Github. The Kernel user documentation and developer documentation can be found here.

A.1 Code Organization

The Source code of the kernel is organized into sub-modules to help maintain clarity. The File structure is as follows :

LISTING 31: Code Organization

```
.
|_____ config.rs # Kernel Configuration variables
|_____ kernel # Kernel modules
|.. |_____ event_management.rs
|.. |_____ mod.rs
|.. |_____ resource_management.rs
|.. |_____ software_comm_bus.rs
|.. |_____ software_sync_bus.rs
|.. |_____ task_management.rs
|.. |_____ time_management.rs
|_____ lib.rs # Specifies which functions and primitives will be exposed
↪  outside the library.
|_____ macros.rs # Macro definitions
|_____ system # Kernel Data-structures definition
|.. |_____ event_manager.rs
|.. |_____ mod.rs
|.. |_____ resource_manager.rs
|.. |_____ software_comm_bus.rs
|.. |_____ software_sync_bus.rs
|.. |_____ task_manager.rs
```

```
|.. |_____ time_manager.rs
|.. |_____ types.rs
|_____ utils # Utility functions
    |_____ arch.rs # Machine specific code
    |_____ errors.rs # Error type definition
    |_____ heap.rs # Dynamic Memory allocator
    |_____ helpers.rs # Helper Functions
    |_____ interrupts.rs # Interrupt handlers
    |_____ mod.rs
```

The definition of all kernel subsystems is in the folder /src/system. The folder /src/kernel/ holds files corresponding to each sub-module. These files declare a global instance of the subsystem data structures and define public functions on them. For example:

LISTING 32: /src/system/task_manager.rs

```
pub struct TaskManager {
  ...
}

impl TaskManager {
  pub fn init(args..) {}
  pub fn start_kernel(args..) {}
  pub fn release(args..) {}
}
```

LISTING 33: /src/kernel/task_manager.rs

```
let task_manager = TaskManager::new();

pub fn init(args..) {
  task_manager.init(args..);
}

pub fn start_kernel(args..) {
  task_manager.start_kernel(args..);
}

pub fn release(args..) {
```

```
12    task_manager.release(args..);
13  }
```

Bibliography

[1] Phillip A. Laplante. *Real-time systems design and analysis: an engineer's handbook*. New York: Institute of Electrical and Electronics Engineers, 1993. ISBN: 9780780304024.

[2] Andrew W. Appel and Zhong Shao. "Empirical and analytic study of stack versus heap cost for languages with closures". en. In: *Journal of Functional Programming* 6.1 (Jan. 1996), pp. 47–74. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S095679680000157X. URL: https://www.cambridge.org/core/product/identifier/S095679680000157X/type/journal_article (visited on 11/18/2019).

[3] Robert Davis, Nick Merriam, and Nigel Tracey. *How embedded applications using an RTOS can stay within on-chip memory limits*. Jan. 2000.

[4] Jane W. S. Liu. *Real-Time Systems*. Upper Saddle River, NJ: Prentice Hall, 2000. ISBN: 0130996513 9780130996510. URL: http://www.amazon.com/Real-Time-Systems-Jane-W-Liu/dp/0130996513.

[5] Gourinath Banda. "Scalable Real-Time Kernel for Small Embedded Systems". English. MA thesis. Denmark: University of Southern Denmark, June 2003. URL: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=84D11348847CDC13691DFAED09883FCB?doi=10.1.1.118.1909&rep=rep1&type=pdf.

[6] Joseph Yiu. *The Definitive Guide to the ARM Cortex-M3*. en. Elsevier, 2007. ISBN: 9780750685344. DOI: 10.1016/B978-0-7506-8534-4.X5001-5. URL: https://linkinghub.elsevier.com/retrieve/pii/B9780750685344X50015 (visited on 11/17/2019).

[7] Wicher Heldring. "An RTOS for embedded systems in Rust". English. PhD thesis. University of Amsterdam, June 2018. URL: https://esc.fnwi.uva.nl/thesis/centraal/files/f155044980.pdf.

[8] Aaron Weiss et al. "Oxide: The Essence of Rust". In: *arXiv:1903.00982 [cs]* (Mar. 2019). arXiv: 1903.00982. URL: http://arxiv.org/abs/1903.00982 (visited on 11/17/2019).

[9] *alloc_cortex_m - Rust*. URL: https://docs.rs/alloc-cortex-m/0.3.5/alloc_cortex_m/ (visited on 11/18/2019).

[10] *Barua, A., Hoque, M. M., & Akter, R. (2014). Embedded Systems: Security Threats and Solutions. American Journal of Engineering Research (AJER), 03(12), 119–123. Retrieved from www.ajer.org*.

[11] *Conditional Compilation - The Rust Programming Language*. URL: https://doc.rust-lang.org/1.30.0/book/first-edition/conditional-compilation.html (visited on 11/17/2019).

[12] *cortex_m - Rust*. URL: https://docs.rs/cortex-m/0.6.1/cortex_m/ (visited on 11/18/2019).

[13] *Defining an Enum - The Rust Programming Language*. URL: https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html (visited on 11/18/2019).

[14] *Dependencies - Rust By Example*. URL: https://doc.rust-lang.org/rust-by-example/cargo/deps.html (visited on 11/17/2019).

[15] *Generic Data Types - The Rust Programming Language*. URL: https://doc.rust-lang.org/book/ch10-01-syntax.html (visited on 11/18/2019).

[16] Rust Working Group. *Introduction - The Embedded Rust Book*. URL: https://rust-embedded.github.io/book/ (visited on 11/18/2019).

[17] *STM32F407VE*. en. URL: https://www.st.com/en/microcontrollers-microprocessors/stm32f407ve.html (visited on 12/03/2019).

[18] *Traits*. URL: https://doc.rust-lang.org/1.8.0/book/traits.html (visited on 11/18/2019).

[19] *Unsafe Rust - The Rust Programming Language*. URL: https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html (visited on 11/22/2019).