The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor

By

Er Pei Qing

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF INFORMATION TECHNOLOGY (HONOURS) COMPUTER ENGINEERING

Faculty of Information and Communication Technology

(Kampar Campus)

JANUARY 2022

Report Status Declaration Form

REPORT STATUS DECLARATION FORM

Title: The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor

Academic Session: JAN 2022

I ER PEI QING

(CAPITAL LETTER)

declare that I allow this Final Year Project Report to be kept in

Universiti Tunku Abdul Rahman Library subject to the regulations as follows:

1. The dissertation is a property of the Library.
2. The Library is allowed to make copies of this dissertation for academic purposes.

Verified by,

(Author's signature)

(Supervisor's signature)

Address:

No 9, Jalan 27, Taman Kota Paloh

86600, Paloh, Kluang, Johor

Chang Jing Jing

Supervisor's name

Date: 15/4/2022

Date: 22/4/2022

FYP THESIS SUBMISSION FORM

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY
UNIVERSITY TUNKU ABDUL RAHMAN

Date: 21/04/2022

SUBMISSION OF FINAL YEAR PROJECT

It is hereby certified that Er Pei Qing (ID No: 18ACB04358) has completed this final year project entitled "The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor" under the supervision of Ts Dr. Chang Jing Jing (Supervisor) from the faculty of Information and Communication Technology (FICT).

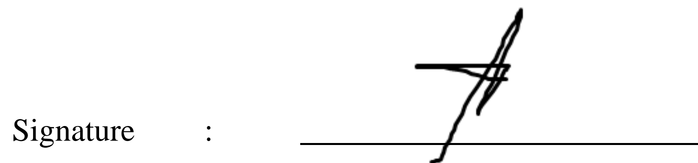
I understand that University will upload softcopy of my final year project in pdf format into UTAR Institutional Repository, which may be made accessible to UTAR community and public.

Yours truly,

(Er Pei Qing)

DECLARATION OF ORIGINALITY

I declare that this report entitled "**The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor**" is my own work except as cited in the references. The report has not been accepted for any degree and is not being submitted concurrently in candidature for any degree or other award.



Signature : ____________________________

Name : _______Er Pei Qing_________

Date : ______14/4/2022____________

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisors, Mr. Mok Kai Ming who has given me this bright opportunity to engage in RISC32 RTOS development project. It is my first step to establish a career in RTOS development. A million thanks to you.

To a very special person in my life, Teo Sei Hau, for his patience, unconditional support, and love, and for standing by my side during hard times. Finally, I must say thanks to my parents and my family for their love, support, and continuous encouragement throughout the course.

ABSTRACT

Real-Time Operating System (RTOS) is a software component that is able to rapidly switches the tasks, making the user have the impression of running multiple programs simultaneously on a single processor. An RTOS provides a highly deterministic reaction and hard real time response to the external events. Because of hard real time response, it is a must for a system to meet its deadline or an unacceptable damage may occur.

So, development of an RTOS for RISC32 processor plays an important role to improve its performance. RISC32 was developed by a group of FICT programmer and the processor is involved in this project as academic purpose. Up to the current stage, the processor supported Interrupt Service Routine (ISR) and exception handler. To improve the processor, an RTOS software code written in C programming language is used to improve the processor performance. To master the RTOS code, FreeRTOS is used as a reference and guidelines to assist us in creating an RTOS code for this project. The RTOS code is divided into few partitions storing in individual file to help the reader to understand the code easily. The code obtained from FreeRTOS was modified in order to port the processor used in this project. Therefore, we expect that the processor is able to switch the task rapidly and must meet the deadline after the implementation of the compiled code.

Table of Contents

The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor i

Report Status Declaration Form ii

REPORT STATUS DECLARATION FORM ii

FYP THESIS SUBMISSION FORM i

DECLARATION OF ORIGINALITY i

ACKNOWLEDGEMENTS ii

ABSTRACT iii

Table of Contents iv

List of Tables vii

List of Figures i

List of Abbreviations iv

CHAPTER 1: Introduction 1

1.1 Background Information 1

1.1.1 MIPS 1

1.1.2 UART 2

1.1.3 RTOS 3

1.2 Problem Statement and Motivation 4

1.3 Project Scope 4

1.4 Project Objective 4

1.5 Impact, Significance and Contribution 5

1.6 Report Organization 5

CHAPTER 2: Literature Review 6

2.1 RISC32 6

2.1.1 Memory Map 6

2.1.2 Coprocessor 0 8

2.2 LLVM 9

2.3 FreeRTOS 9

2.4 RT-Thread 11

2.5 Comparison between FreeRTOS and RT-Thread 13

CHAPTER 3: Proposed Method / Approach 14

3.1 Methodologies and General Work Procedures 14

3.2 Analysis of RTOS Architecture and Components 14

3.2.1 Analysis of RTOS Behaviour: Multiprogramming vs Multiprocessing 15

3.2.2 Analysis of RTOS Behaviour: Task and Scheduling Algorithm 16

3.2.3 Analysis of RTOS Behaviour: Process Switching 17

3.3 Comparison between LLVM and GCC 18

3.4 RISC32 Components Involved 19

3.5 Exception Handling Registers 20

3.5.1 Status Register 20

3.5.2 Cause Register 21

3.5.3 EPC Register 22

3.6 Tools Involved 22

3.6.1 LLVM 22

3.6.2 Xilinx Vivado 23

3.7 Implementation Issues and Challenge 23

3.8 Timeline 24

3.8.1 Timeline of FYP 1 24

3.8.2 FYP2 Timeline 24

Chapter 4 Analysis and Modification of FreeRTOS 26

4.1 Analysis of FreeRTOS Architecture 26

4.2 Demo Path 26

4.3 Source Path **27**
4.3.1 The include Folder 28
4.3.2 The portable Folder 28
4.4 FreeRTOS Kernel Architecture and its Usage **29**
4.5 FreeRTOS Functions and Code Analysis and Modification **30**
4.5.1 Source Files of FreeRTOS Core 30
4.5.2 Configuring RTOS Scheduler 33
4.5.3 Portable Layer 35
CHAPTER 5: FreeRTOS Implementation **41**
5.1 Setup LLVM compiler as Toolchain **41**
5.2 Compilation results of LLVM **41**
5.2.1 Testing the LLVM Compilation via UART Communication 41
5.2.2 FreeRTOS Source Code Compilation and Setup 43
5.3 FreeRTOS Simulation on RISC32 .... **46**
5.3.1 RISC32 Testbench 46
5.3.2 FreeRTOS Assembly Code Debugging 50
CHAPTER 6: CONCLUSION **52**
6.1 Conclusion **52**
6.2 Future Work **52**
Bibliography .... **A**
Appendix B – RISC 32 Coprocessor 0 Register .... **C**
Biweekly Report .... **D**
Poster .... **K**
Plagiarism Check Result .... **L**
FYP2 Checklist .... **N**

List of Tables

Table 2.5.1 Comparison of FreeRTOS and RT-Thread..13

List of Figures

Figure 1.1.1 MIPS 5-stage pipeline 1
Figure 1.1.2 Two UART communicating with each other[1] 2
Figure 1.1.3 RTOS Task State (Yasen.S) 3

Figure 2.1.1 Structural View of RISC32 Microarchitecture [2] 6
Figure 2.1.2 Virtual to physical memory mapping based on 32-bit MIPS architecture. The mapped memory segment is mapped to the Memory Management Unit (MMU) while the cached segment used the cache memory to enhance the data accessing speed[3]. 7
Figure 2.1.3 Memory allocation on kseg0 and kseg1[3] 8
Figure 2.1.4 Exception handling registers in Coprocessor 0[4] 8

Figure 2.2.2 Internals of LLVM[3] 9

Figure 2.3.1 Architecture of FreeRTOS[5] 10

Figure 2.4.1 RT-Thread architecture 12

Figure 3.2.1 Process Control Block[6] 14

Figure 3.3.1 The internals of GCC[3] 18

Figure 3.4.1 Task 1 information is stored inside the .heap segment of RISC32 20

Figure 3.5.1 Layout of status register[4] 20
Figure 3.5.2 Graphical view of cause register[11] 21
Figure 3.5.3 Exception codes[11] 21

Figure 3.6.1 Simplified architecture of LLVM of RISC32[3] 23

Figure 3.9.1 Project Progress from Week 1 to Week 5 24
Figure 3.9.2 Project Progress from Week 6 to Week 10 24
Figure 3.9.3 Project Progress from Week 1 to Week 5 24
Figure 3.9.4 Project Progress from Week 6 to Week 10 25
Figure 3.9.5 Project Progress from Week 11 to Week 13 25

Figure 4.1.1 FreeRTOS kernel directory structure[11]...26

Figure 4.2.1 Six RISC-V example projects 26

Figure 4.3.1 "Source" folder structure [11] 27
Figure 4.3.2 Minimal files needed to build real time kernel...27
Figure 4.3.3 Header files of FreeRTOS kernel 28
Figure 4.3.4 RISC-V uses GCC compiler..28

Figure 4.4.1 FreeRTOS Kernel Architecture..29

Figure 4.5.1 Example code of task.c...30
Figure 4.5.2 Example code of list.c 31
Figure 4.5.3 Example codes of queue.c 32
Figure 4.5.4 Part 1 settings 33
Figure 4.5.5 Mutex Example 34
Figure 4.5.6 Part 2 setting..34
Figure 4.5.7 Example function prototypes of heap_4.c 35
Figure 4.5.8 Example function prototypes defined in portable.h 36
Figure 4.5.9 Function definition of xPortStartFirstTask defined in port.c 37
Figure 4.5.10 Difference between RISC32 and RISC-V...38
Figure 4.5.11 Example assembly code in portASM.s...39
Figure 4.5.12 Difference between RISC-V and RISC32 ISA 40
Figure 4.5.13 The functions of memcpy, memset, and strlen defined at string2.h..................40
Figure 4.5.14 The files are compiled step by step 41

Figure 5.2.1 Source code to be implemented 42
Figure 5.2.2 Assembly code of testing_llvm.c 42
Figure 5.2.3 The result of testing_llvm.c via UART 43
Figure 5.2.4 Queue is created and the tasks created are inserted into queue 44
Figure 5.2.5 The task will be executed in prvQueueReceiveTask() 44
Figure 5.2.6 Part of assembly codes generated 45

Figure 5.3.1 The flow of the program leading to infinite looping 50
Figure 5.3.2 The result of FreeRTOS simulation on RISC32 51

List of Abbreviations

API	Application Programming Interface
BIOS	Basic Input/Output System
BRK	Program Break Address
CISC	Complex Instruction Set Computer
FPGA	Field Programmable Gate Array
GCC	GNU compiler collection
GPIO	General Purpose Input/Output
ID	Instruction Decode
IDE	Integrated Development Environment
IF	Instruction Fetch
I/O	Input/Output
IOT	Internet Of Things
IR	Immediate Representation
IAR	Ingenjörsfirman Anders Rundgren (Anders Rundgren Engineering Company)
MEM	Memory
MEMCPY	Copy Memory Block
MEMSET	Set Block Values
MIPS	Microprocessor without Interlocked Pipeline Stages
MTC0	Move to coprocessor 0
MFC0	Move from coprocessor 0
OS	Operating System
PC	Program Counter
RAM	random access memory
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RTOS	Real Time Operating System
SBRK	Short for Program Break Address
UART	Universal Asynchronous Receiver and Transmitter

CHAPTER 1: Introduction

1.1 Background Information

A computer consists of several components. Among the components, the main components are the processor and I/O devices. A processor performance's maybe robust, but without an interactive interface with the user, the processor might not be fully utilized. The I/O devices act as the interactive interface between the user and the processor. The interconnect between I/O devices and the processor would be the bus system.

1.1.1 MIPS

MIPS is known as Microprocessor without Interlocked Pipeline Stage, which is based on RISC architecture developed by MIPS technologies, previously known as MIPS Computer Systems. RISC processor supports simple instruction set compared to CISC[1]. RISC architecture emphasizes on using register rather than memory. Instead of using Intel 80x86, MIPS is used because it has a simple design and yet high performance as embedded processor. It also has large market for embedded app. After years of development, MIPS architecture nowadays can support 64-bit addressing and operation and high-performance floating point making it popular in the embedded systems implementation such as router, game machine and so on. The instruction execution is broken by the operation of MIPS processor into multiple small independent stages (Integrated Device Technology. Inc, 1994, pg1-2). The word "stages" implies the datapath resources at each stage.

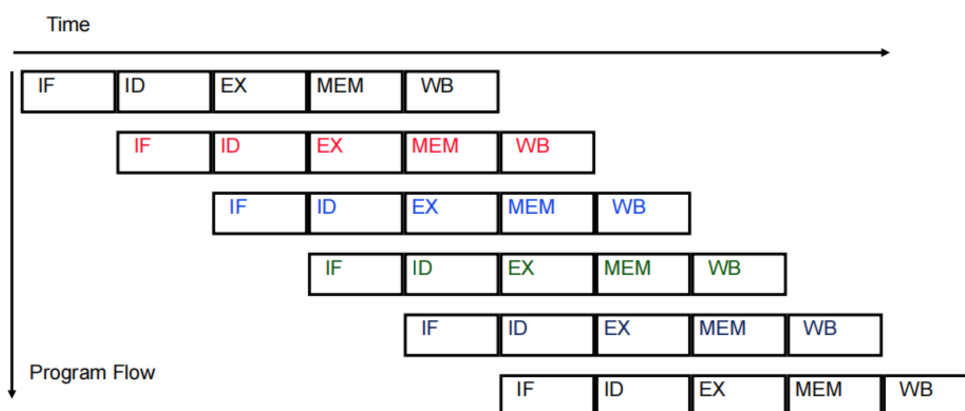


Figure 1.1.1 MIPS 5-stage pipeline

From Figure 1.1.1, the execution of an instruction is done in 5 basic stages including:

- IF: Instruction fetch and update PC
- ID: Instruction decode and register fetch
- EX: Execute R-type, calculate memory address
- MEM: Read data from memory or write data to memory
- WB: Write the result data into register file

1.1.2 UART

UART stands for Universal Asynchronous Receiver/Transmitter, it is used for asynchronous serial communication of data over peripheral device serial port. Most embedded systems use UART for data communication as it is a hardware communication protocol that only uses 2 wires for transmitting end (TX) and receiving end (RX). Figure 1.1.2 shows that there are 2 UART communicating with each other.

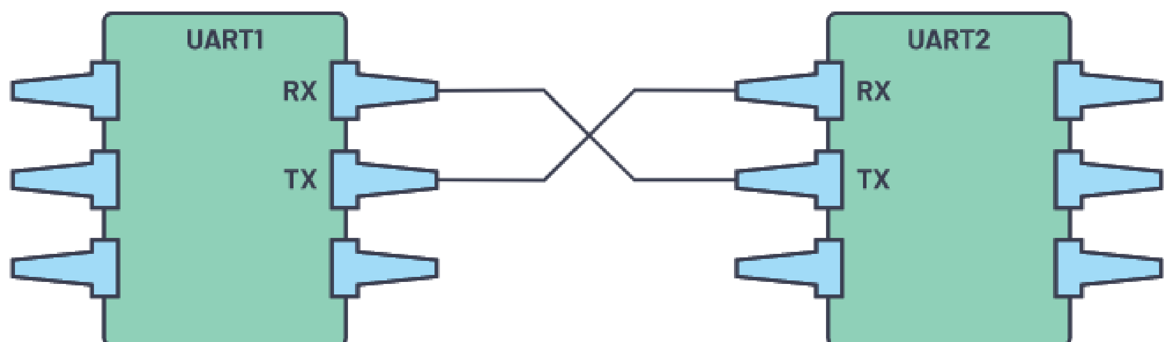


Figure 1.1.2 Two UART communicating with each other[2]

1.1.3 RTOS

Real Time Operating System is an operating system intended to serve real time applications. It is a software component that rapidly switches between tasks, make the user have an impression that multiple programs are executing simultaneously on a single processor. Operating system consists of many different parts such as file system, I/O, memory allocation, network, and scheduler. RTOS provides a hard real time response and a highly deterministic reaction to external event. Hard real time is a system that must always meet all deadlines or the system will fail if the deadline is missed. RTOS can be time-sharing or event-driven. Time-sharing system switch the task based on the timer interrupt while event-driven system switches the task according to the task priority. The value of a real-time operating system depends on how fast it can respond compared to the amount of work it can perform in given period of time. Most RTOS is using a pre-emptive algorithm. A basic RTOS has 3 states which the task might be assigned.

Figure 1.1.3 RTOS Task State (Yasen.S)

From Figure 1.1.3, a RTOS task usually has one main state such as:

- Ready: The task is ready to be executed by processor but not yet occupy the processor.
- Running: The task is currently executed by the processor.

- Blocked: When the task is making an I/O request, the task will go to blocked state until the event it is waiting occurs.

1.2 Problem Statement and Motivation

So far, a MIPS-ISA compatible RISC32 processor had been developed. With the peripheral interface to it, firmware is also built to test out the customizability of the RISC32 processor. However, the backend of RISC32 processor is not completed yet. Despite RISC32 processor already has Interrupt Service Routine and the exception handler, but the operating system still needs some improvement. To guarantee the response time and the deterministic behaviour of RISC32, a project is initiated to develop the Real Time Operating System.

1.3 Project Scope

The project scope of the project mainly concentrates on using an open source RTOS, FreeRTOS to perform multitasking and enable guaranteed response time. In fact, RTOS is an embedded software as it interfaces with the hardware and dealing with numerous simultaneously interrupts as well as scheduling concurrent task. To achieve the goal, the code provided will be port over into the RTOS chosen. To make sure llvm compiler supports some special function library which is not shown in the file developed in the previous work, we convert the extra instruction and add it into that file. Therefore, a multitasking feature of processor will be developed at the end the project.

1.4 Project Objective

The following are the objectives which are set for the project:

- Analyse the open source RTOS architecture and components.
- Develop a suitable RTOS for the RISC32 architecture which is compilable by LLVM compiler.
- LLVM setup and RTOS compilation and debugging. By compiling the RTOS into MIPS assembly instruction, the LLVM compiler can readjust the address.
- Simulate the RTOS in RISC32.

1.5 Impact, Significance and Contribution

With the assisting of RTOS for RISC32, the processor can interleave many periodic tasks in an easy way. As RTOS also supports priority-based scheduling, low priority tasks can be scheduled round-robin, interrupted at a specified time interval while high priority task will pre-empt those low priority tasks. The scheduling algorithm used is "First Come First Serve" scheduling. So, RTOS will simplify the software and the improve the predictability of the application. By having this contribution, we can utilize the processor more wisely.

1.6 Report Organization

The details of this project are shown in the following chapters.

- Chapter 1 : Introduction. Related information is included to help the reader to understand easily and give a basic concept about the project. Problem statement, project scope and its objective are discussed in this chapter.
- Chapter 2 : Literature Review. History of RTOS and its related information are discussed. Knowledge about the open source code used will also be discussed.
- Chapter 3 : Methodology. This chapter discuss about the procedure of the project and the project's timeline.
- Chapter 4 : Analysis and Modification of FreeRTOS
- Chapter 5: FreeRTOS Implementation
- Chapter 6: Conclusion and Future Work.

CHAPTER 2: Literature Review

2.1 RISC32

RISC32[3] is a MIPS Instruction Set Architecture (ISA) compatible 5-stage pipeline 32-bit IoT processor. It decodes and executes MIPS instruction in 5 stages which are Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Read/Write data from/to Memory (MEM), and Write Back (WB). The word "Stages" means the datapath resources at each stage. Figure 2.1.1 shows the structural view of RISC32 microarchitecture.

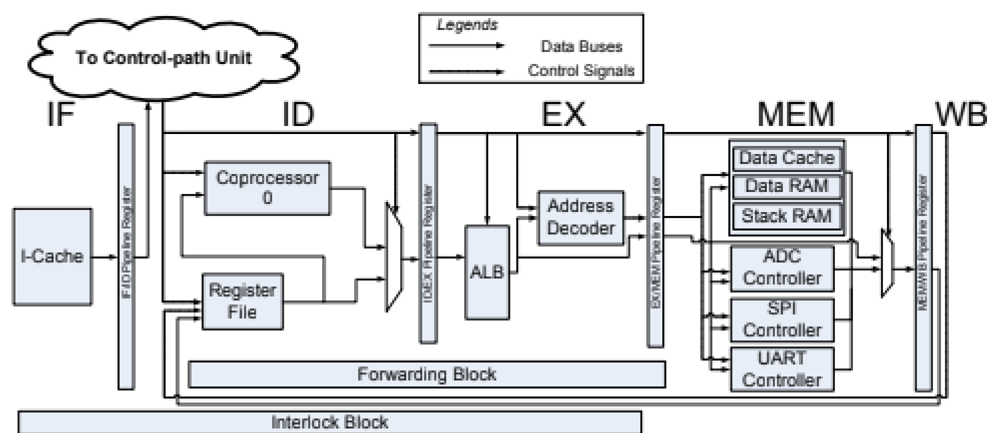


Figure 2.1.1 Structural View of RISC32 Microarchitecture [4]

From Figure 2.1.1, RISC32 has a Coprocessor 0 (CP0) which is providing some necessary functions to support Operating System such as monitoring hardware interrupt caused by I/O controller, and software exceptions[4]. Software exceptions are abnormal events that occur after executing a software program such as illegal instructions and arithmetic overflow. Besides, RISC32 is integrated with I/O controllers like UART, ADC, SPI, and GPIO to provide an interface for IoT applications. These I/O controllers are used to interact with external devices like sensors, wireless module, printer, and so on.

2.1.1 Memory Map

RISC32 Memory can be defined as a large and one-dimensional array with 32-bit address, it can support up to 4GB memory space. The Physical Memory of RISC32 is the actual size of the memory to store or access the instruction and operand which include Flash Memory, Data and Stack RAM, boot RAM, and I/O registers[3]. The Virtual Memory is the logical view of the address space. It is useful for planning of distribution of the various address segments

throughout RISC32 address space for programmer's use. Figure 2.1.2 shows the virtual to physical memory mapping based on MIPS32 architecture.

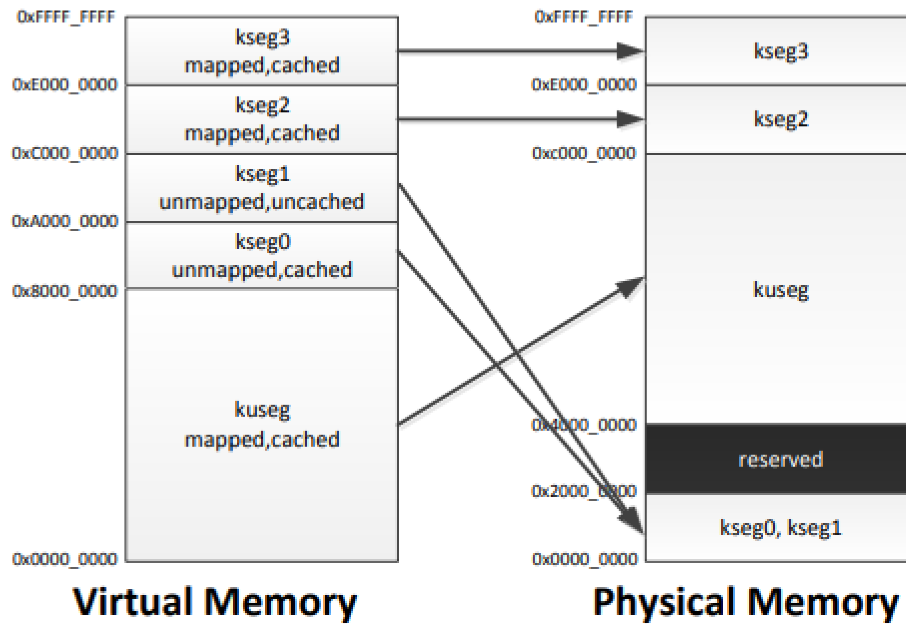


Figure 2.1.2 Virtual to physical memory mapping based on 32-bit MIPS architecture. The mapped memory segment is mapped to the Memory Management Unit (MMU) while the cached segment used the cache memory to enhance the data accessing speed[3]**.**

From Figure 2.1.2, there are 5 segments distributing in virtual memory which are kernel user segment (kuseg), kernel segment 0 (kseg0), kernel segment 1 (kseg1), kernel segment 2 (kseg2), and kernel segment 3 (kseg3). kuseg, kseg2, and kseg3 are mapped segment and they should not be used by the processor when there is no Memory Management Unit (MMU) because MMU takes the responsibility of the translation of virtual addresses to physical addresses. Therefore, only kseg0 and kseg1 are available for the implementation. From Figure 2.1.2, kseg0 and kseg1 share the same physical addresses but different virtual addresses if kseg0 is not accessed through the cache. Figure 2.1.3 shows the memory allocation on kseg0 and kseg1.

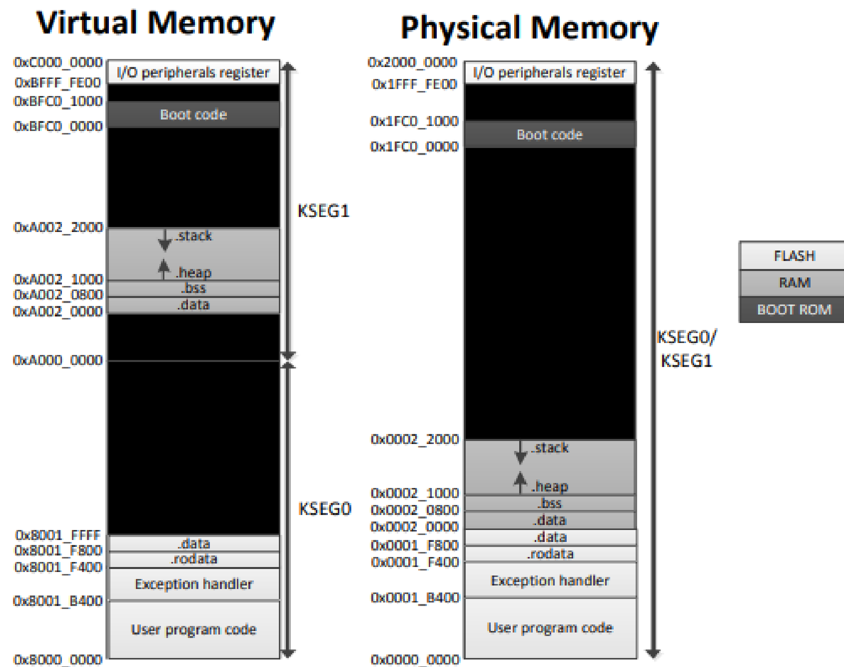


Figure 2.1.3 Memory allocation on kseg0 and kseg1[3]

2.1.2 Coprocessor 0

MIPS has 2 Coprocessors which are Coprocessor 0 (c0) and Coprocessor 1 (c1). In this project, only Coprocessor 0 will be used as it handles the exceptions and stores the information of the corresponding exception event. Figure 2.1.4 shows the relevant exception handling registers in Coprocessor 0. Further details about the exception handling registers will be discussed in Chapter 3.

Register Number	Register Name	Usage
8	BadVAddr	Memory address where exception occurred
12	Status	Interrupt mask, enable bits, and status when exception occurred
13	Cause	Type of exception and pending interrupt bits
14	EPC	Address of instruction that caused exception

Figure 2.1.4 Exception handling registers in Coprocessor 0[5]

2.2 LLVM

LLVM has a series of modularized compiler components and tool chains while GCC is a static program language compiler for the GNU and Linux systems. Unlike GCC, LLVM is not a compiler for any programming language, but it is a framework to generate object code from any programming language source code.

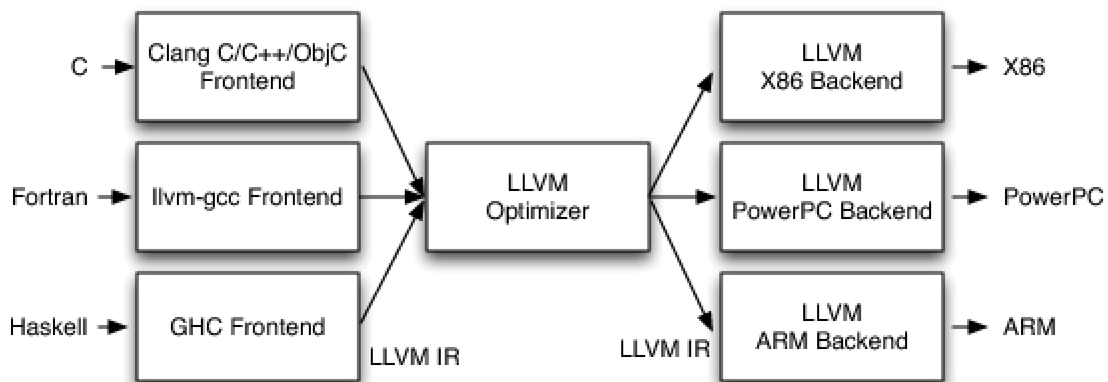


Figure 2.2.1 Internals of LLVM[4]

From Figure 2.2.2, the LLVM internals is known as three-phase design as it consists of 3 main components which are frontend, optimizer, and backend. The frontend takes the responsibility of parsing the source codes such as C/C++ and checking for the error. The LLVM frontend is unique to its supported programming language, for example Clang is the frontend to C while LLVM-GCC is the frontend of Fortran. The parsed source code will be translated into LLVM Intermediate Representation (IR) as the output of the frontend passing to the LLVM optimizer. LLVM IR is a low-level RISC assembly language used by LLVM compiler framework for transformation. The LLVM optimizer will then do a variety of transformation in order to improve the run-time of the code. As a result, the LLVM IR produced from the optimizer will be a more optimized version and pass to the backend. The backend will map the code into the targeted machine code. Each backend can only be written for a single target family and they are independent of each other[4].

2.3 FreeRTOS

FreeRTOS is an open source real-time operating system for microcontroller and microprocessor. It supports more than 35 architectures and it is distributed under MIT License.

FreeRTOS was developed by Richard Berry around 2004, and was maintained by Richard's company, Real Time Engineers Ltd. The design goals of FreeRTOS are easy to use, small footprint and robust. It also supports plenty of hardware architectures, making it a better choice to be used with different IoT application. FreeRTOS is strictly quality managed and is professional developed, it does not contain any ambiguous intellectual property and it is totally free to use without any exposure of personal code. People are allowed to use FreeRTOS code to create their product having market value without informing the company of FreeRTOS. In order to make the code readable and easy to portable, it is developed in C programming language and some assembly functions. FreeRTOS is a real-time kernel which the embedded system can be built to meet their hard real-time requirements. Hard real-time requires us to set a time deadline and fail to meet the deadline will result in system malfunction. For example, the car's air bag will be more harm compared to good if it does not respond on time to the sensor input.

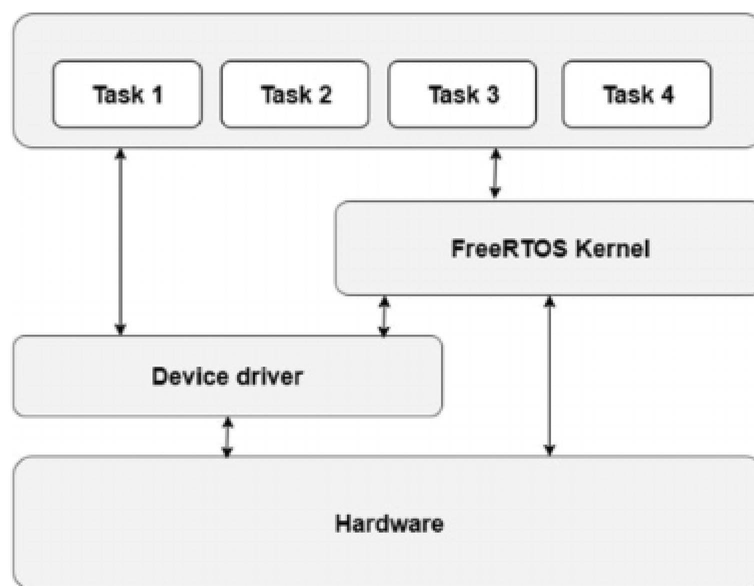


Figure 2.3.1 Architecture of FreeRTOS[6]

From Figure 2.3.1, the FreeRTOS Kernel is contacting with all the components such as device driver, and hardware. The FreeRTOS Kernel will schedule the tasks coming from device driver and hardware. Device driver can be known as a software driver as it is a small piece of software that allowing the hardware to interact with operating system or with another hardware. Device driver plays an important rule as it keeps the system running efficiently when a computer having the correct device drivers. Without device driver, the OS will not be able to

communicate with I/O device because the OS works with device driver and BIOS to perform hardware task.

For those single core processors, only a single task is allowed to occupy and run on the processor at one time. FreeRTOS is a real-time kernel that decides which task to be executed according to their priority assigned by the application designer. Thus, the task implementing the hard real-time requirements can be assigned with a high priority. The kernel has the responsibility for timing execution and provides a time-related API to the application. So, the application structure is maintainability as it is simple and having smaller code size. Therefore, most of the FreeRTOS code involves prioritizing, scheduling and running the task defined by the user (Christopher.S).

Besides, software is said to be totally event-driven if kernel is used in the project. It is because no more polling for the event and no time are wasted. Event polling is the process where the computer is waiting for an external device to check for the task state. When the processor is idle, the scheduler will create an idle task automatically to perform background checks and the processor is in a low-power mode. Therefore, there are numerous reasons of using the RTOS kernel because it improves efficiency, enable code reuse and power management.

Apart from that, FreeRTOS is feature rich and still having continuous active development. It is pre-emptive and the core of the RTOS kernel can be built by having only three .c files. It is pre-emptive and using First-fit algorithm. First-fit algorithm scans memory from the beginning and selects the available space which is sufficient for a task to be allocated. The code style created by FreeRTOS is simple and reliable.

2.4 RT-Thread

RT-Thread is a real time operating system for embedded devices which is developed by the RT-Thread Development Team based in China. The goal is to change the situation of has no well-used open source RTOS in the microcontroller area in China. It is written in C programming language in order to understand easily as well as easy to port. To making the code elegant and structured, object-oriented programming methods was applied to real time system design.

There are 2 versions in RT-Thread which are Standard and Nano. For MCU system, the Nano kernel version which needs 3KB Flash and 1.2KB RAM memory resources is tailored with ease-to-use tools. The RT-Thread architecture has real-time kernel and rich components.

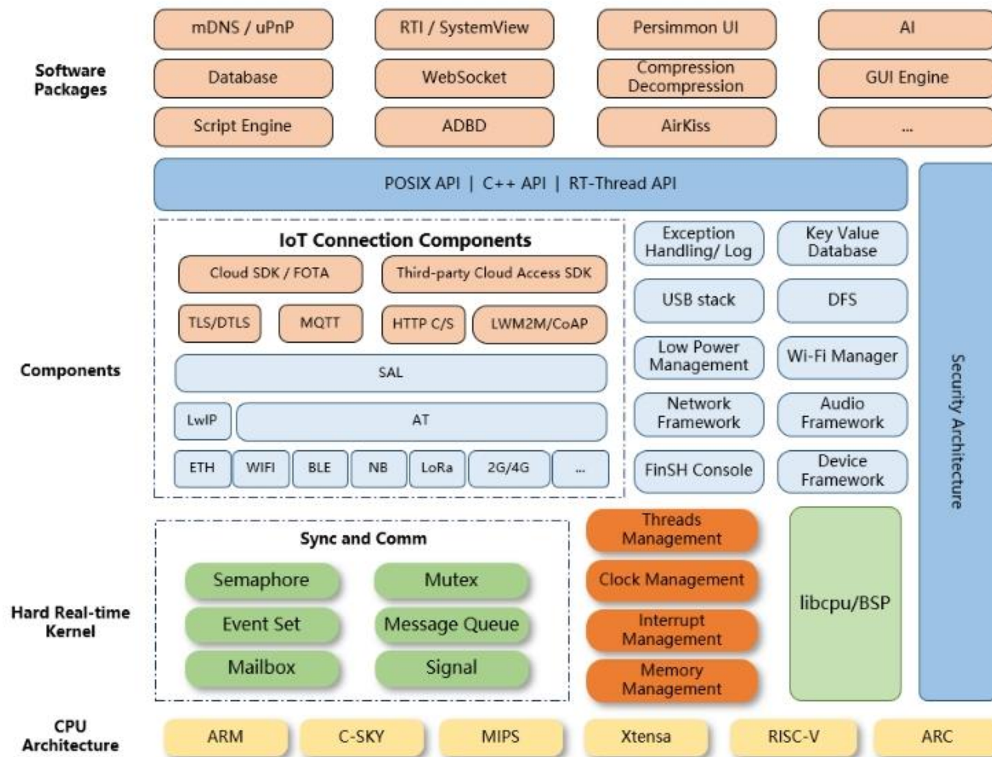


Figure 2.4.1 RT-Thread architecture

From the figure above, we know that the architecture includes:

- Kernel layer. The core part of RT-Thread which includes the implementation of objects in the kernel system like multi-threading, semaphore, memory management, timer, etc.
- Components and service layer. For examples, virtual file systems, device framework, network frameworks and so on are the components that is on top of the RT-Thread kernel. Thus, it allows high internal cohesion inside the components and low coupling between components.
- RT-Thread software package. It acts as a general-purpose software component running on IoT OS platform for different application areas such as source code. RT-Thread provides open package platform with officially available packages so that there is a

choice of reusable packages which is an important part in RT-Thread ecosystem. RT-Thread can support up to 180 software packages.

RT-Thread has ported for almost 90 development boards, most BSPs support, GCC compiler and provided default MDK and IAR project, which allows the users to add their application code directly based on the project. Also, RT-Thread supported many architectures and covered the major architectures in current applications. The architecture manufacturer involved are MIPS32, RISC-V, ARM Cortex-R4 and more. While the main compilers which are supported by RT-Thread is GCC, IAR and RT-Thread Studio.

2.5 Comparison between FreeRTOS and RT-Thread

	FreeRTOS	RT-Thread
Scheduler	Pre-emptive, optional priority	Full-preemptive priority based
Compiler used	GCC, IAR, Clang and so on	GCC, IAR
Kernel type	microkernel	Single kernel
Language Support	C	C

Table 2.5.1 Comparison of FreeRTOS and RT-Thread

From Table 2.5.1, it seems like a quite difference only between FreeRTOS and RT-Thread. However, FreeRTOS is more secure and it provides plenty of demo project for various of architectures and compilers. Its scalable size with program memory footprint as low as 9KB makes the kernel tiny and power-saving.

CHAPTER 3: Proposed Method / Approach

3.1 Methodologies and General Work Procedures

In order to make the RISC32 support multi-tasking, has the deterministic behaviour, a real time operating system will be developed. By having RTOS, task can be prioritized which depends on the importance and facilitates application expansion. So, the processor can run more efficiently even on a limited hardware resource. it consumes little power and memory because the kernel size is small and it able to fit the limited ROM storage of embedded systems.

3.2 Analysis of RTOS Architecture and Components

The kernel is a core component which running at all times in the system. Each executing process is known as task and consists of an executable program. There are 3 components for a process:

- An executable program
- Associated data needed by a program
- Execution context (task state) of a program. All information on how a process will be controlled by the system is included in Process Control Block (PCB).

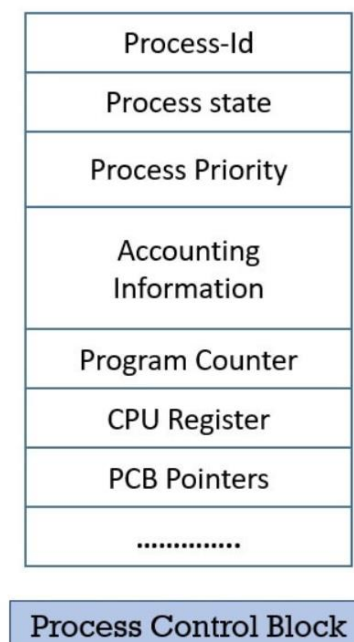


Figure 3.2.1 Process Control Block[1]

3.2.1 Analysis of RTOS Behaviour: Multiprogramming vs Multiprocessing

Multiprogramming is running a group of tasks concurrently while multiprocessing is running a group of tasks simultaneously. In multiprogramming, the processor will switch to another task if the current running task is waiting for I/O, because there is only one processor in the system. In multiprocessing, many processes are running on different processors when the system has multiple processors. In fact, multiprogramming gives an impression of running the tasks simultaneously as the scheduler helps the processor to switch the tasks rapidly when the task is waiting for I/O. Doing so can prevent the processor from wasting time in waiting for the task to be ready running. From Figure 3.2.2, we know that the processor is fully utilized without idle although there is only one processor for multiprogramming.

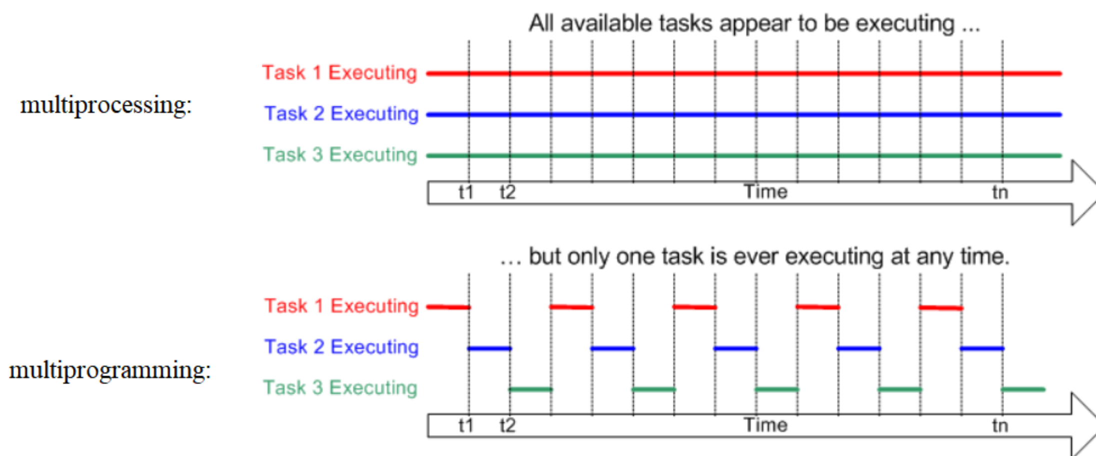


Figure 3.2.1 Multiprocessing and multiprogramming

Since RISC32 is a single core processor, the concept of multiprogramming will be applied in RISC32 by using FreeRTOS source codes. From Figure 3.2.2, only one task will be executed on the processor because there is only a processor available. The scheduler will schedule the time slice for each task to make sure that all tasks have the chance to be executed on the processor.

3.2.2 Analysis of RTOS Behaviour: Task and Scheduling Algorithm

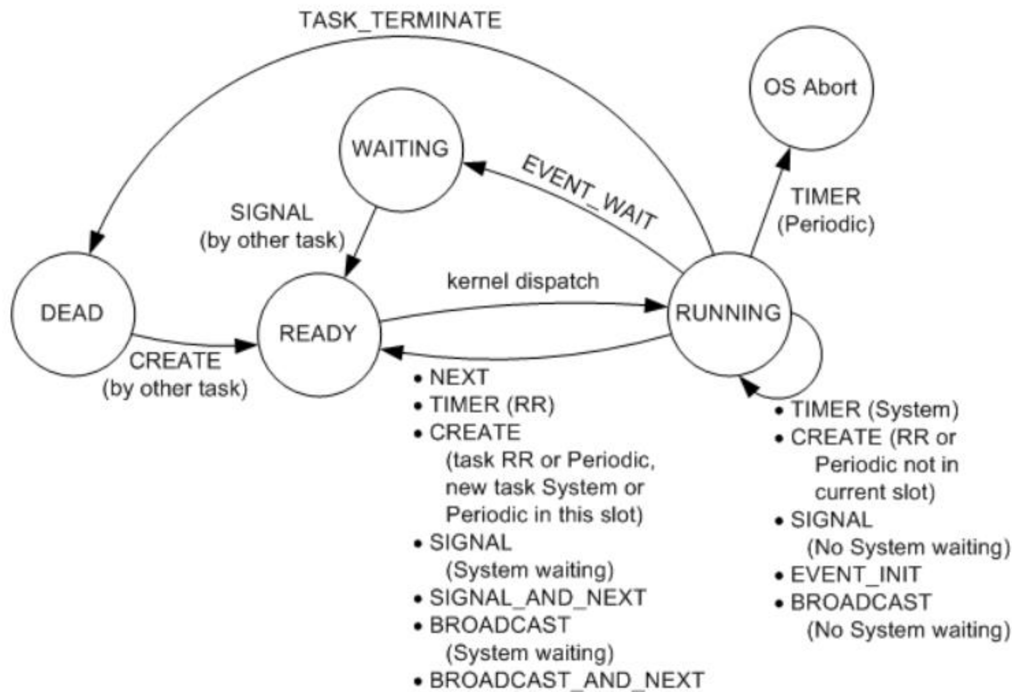


Figure 3.2.2 Task state[7]

Before creating an RTOS for the project, we need to understand on how a task is scheduled from queuing, executing and finally exiting the processor. Each process has many states which will change when there is an interrupt occurs. To make multiprogramming possible, an interrupt is needed to send a signal to processor in order to switch task. According to Figure 3.2.3, CPU scheduling occurs when a task:

i. Ready to Running state
ii. Running to Ready state
iii. Running to Blocked state
iv. Blocked to Ready state
v. Running to Terminates

When only conditions i and v occur in CPU scheduling, it is a non-preemptive scheduling; otherwise, it is pre-emptive. A process scheduler schedules different processes into CPU based on the scheduling algorithm. There are some process scheduling algorithms which can be pre-emptive or non-preemptive:

- First Come First Serve scheduling

- Shortest Job Next scheduling
- Shortest Remaining Time
- Round Robin Scheduling

The kernel of FreeRTOS supports 2 types of scheduling algorithms which are Round-robin scheduling and Fixed Priority Pre-emptive algorithm. Round Robin algorithm uses pre-emption based on time quantum which means each process is allowed to use the processor based on the amount of time determined. Fixed Priority Pre-emptive algorithm selects task to use the processor according to the task's priority value. So, a higher priority task always occupies the processor than a lower priority task.

3.2.3 Analysis of RTOS Behaviour: Process Switching

Task consists of a sequence of code which will be executed in the processor. When the task is waiting for an event occurs, the task will be swapped out to Block State by the kernel. According to the figure below, there are 7 steps required to switch a process and scheduler the following task to be executed.

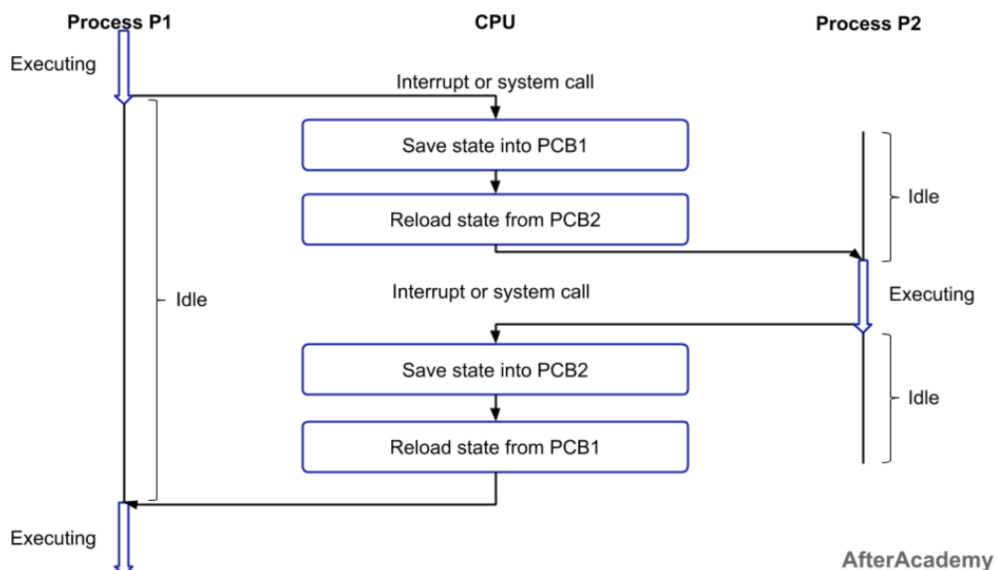


Figure 3.2.3 Process Switching steps[8]

- Step 1: Save the context of processor including the program counter and the other registers.
- Step 2: Update the process's PCB which is running by changing the process state.
- Step 3: Move PCB to an appropriate queue such as ready queue, block queue or ready(suspended) queue.
- Step 4: Select another process to be executed.
- Step 5: Update the selected process's PCB.
- Step 6: Update the memory management data structures.
- Step 7: Restore the context of the selected process.

3.3 Comparison between LLVM and GCC

Before starting the comparison, GCC compilation process will be introduced first. The compilation processes are read the source file, pre-process the source file, transform the source file into GCC IR, optimize and generate an assembly file, and finally an object file is created by the assembler. Figure 3.3.1 shows the internals of GCC compiler.

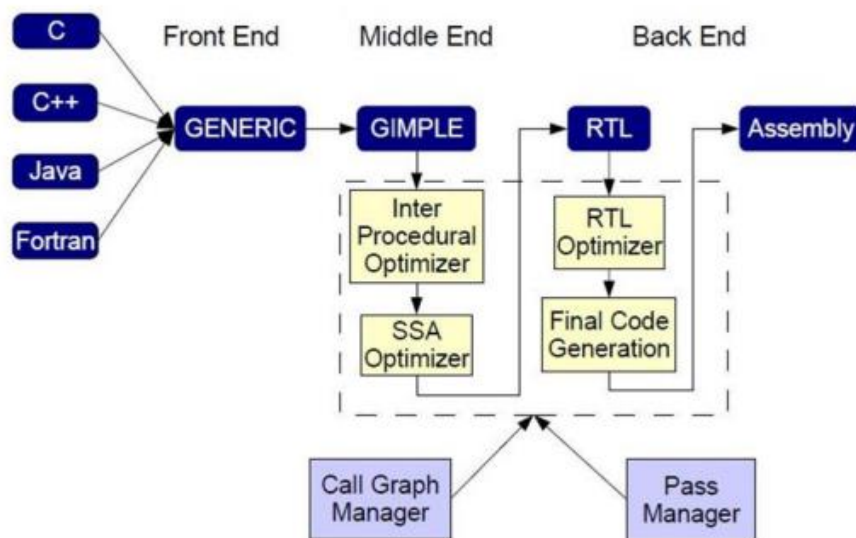


Figure 3.3.1 The internals of GCC[4]

The data structure of LLVM IR is more concise than the GCC IR data structure, meaning that less memory is occupied by LLVM IR during compilation[9]. Thus, LLVM

compiler has greater compilation performance than GCC compiler as LLVM has shorter compilation time.

In terms of code complexity, GCC has well-defined frontend and backend stages, leading it become a complicated software while LLVM is modular in design. Therefore, LLVM is a better choice as it has rather straightforward LLVM internals and the design is easy to understand. In this project, LLVM will be used to compile FreeRTOS source codes.

3.4 RISC32 Components Involved

Understanding where the data will be stored in memory will help to prevent stack overflow and memory leaks. From Figure 2.3.2, a simple C program allocating in memory will be separated into user program code (.text), initialised data (.data), uninitialized data (.bss), stack data (.stack), and heap data (.heap) [3]. In RISC32, those initialised global variables, and static variables are stored in flash memory. While the .bss segment stores those uninitialized variables in Data RAM. When the processor is powered, the bootloader will copy the content of .data storing in flash memory to the .data segment in Data RAM. The .stack segment stores the functions and their local variables. Due to .stack is a Last-In-First-Out (LIFO) system, the variables can be continuously pushed to the stack when there are nested function calls. When returning to the caller function from the called function, the variables will be popped out of the stack as the return data or the variables will be deleted entirely when they are no longer used. Besides, the .stack can grow in size when the compiler reserves as much stack as needed for local variables.

The .heap segment is similar to .stack segment as it also can grow when the program is running, it grows toward the .stack segment. The .heap is known as dynamic data segment because the memory space can be explicitly created to store what the programmer is desired to store by calling a function called malloc(). There are 2 functions can be used to adjust the memory space allocated for the calling task which are brk() and sbrk(). The brk() set the task's break value to a higher address while the sbrk() add an increment of storage to the heap segment of a task[10]. When finish using the dynamic memory location, the free() function is called to release the location or the .heap will continue growing indefinitely.

FreeRTOS will allocate the memory of RISC32 at runtime when a new task is created and it is assigned a portion of memory from the .heap segment. That portion of memory is

distributed into 2 parts, task control block (TCB) and a stack which is exclusive to the task. TCB is a data structure storing important information of a task such as the task's priority level, the location of the task's stack, and so on. Figure 3.4.1 shows the information of Task 1 stored in the .heap segment.

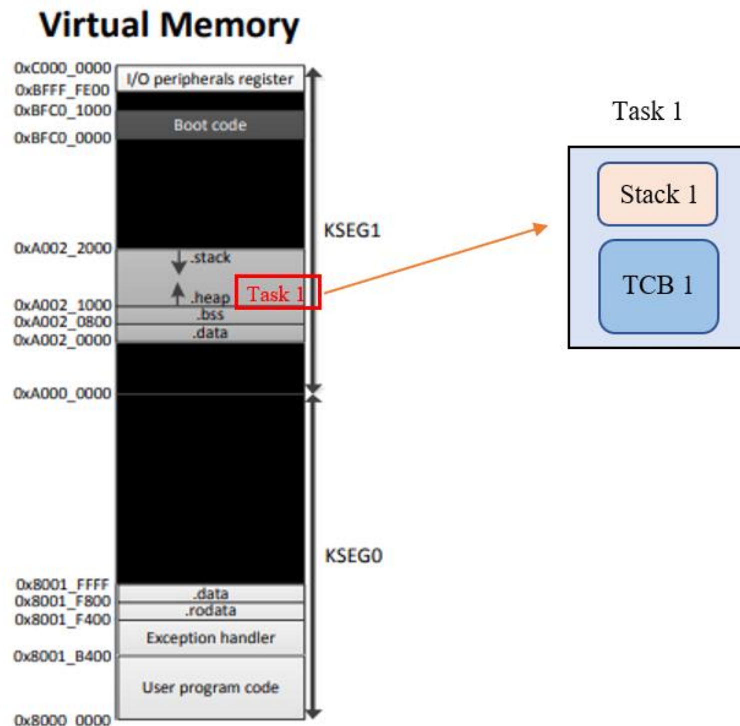


Figure 3.4.1 Task 1 information is stored inside the .heap segment of RISC32

3.5 Exception Handling Registers

3.5.1 Status Register

Status register ($stat or $12) is a read/write register containing the information of exception. Figure 3.5.1 shows the layout of status register where bit 0 – bit 5 represent status information while bit 8 – bit 15 represent interrupt mask.

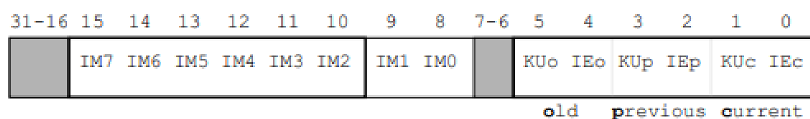


Figure 3.5.1 Layout of status register[5]

From Figure 3.5.1, bit 8 – bit 9 are used for software interrupt level while bit 10 – bit 15 is responsible for hardware interrupt level. Bit 0 (IE) in status register is known as interrupt enable. When it is set to 1, meaning that an interrupt is enable.

3.5.2 Cause Register

Cause Register ($cause or $13) stores the information the pending interrupts as well as the causes of the exception[5]. Figure 3.5.2 shows the graphical view of the cause register.

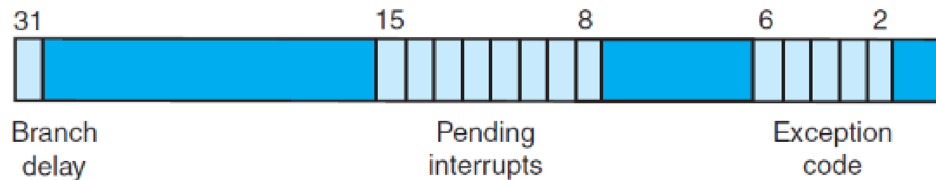


Figure 3.5.2 Graphical view of cause register[11]

In cause register, its bit 31 is used for branch delay, meaning that there is an exception occurring inside in branch/jump instruction. Bit 8 – bit 15 represent the pending interrupt. If the pending bit is set to 1 means that there is an exception occurring and it is in the pending state. Bit 2 – Bit 6 are exception code which is used to indicate what caused the exception. Figure 3.5.3 shows the exception codes in cause register.

Number	Name	Cause of exception
0	Int	interrupt (hardware)
4	AdEL	address error exception (load or instruction fetch)
5	AdES	address error exception (store)
6	IBE	bus error on instruction fetch
7	DBE	bus error on data load or store
8	Sys	syscall exception
9	Bp	breakpoint exception
10	RI	reserved instruction exception
11	CpU	coprocessor unimplemented
12	Ov	arithmetic overflow exception
13	Tr	trap
15	FPE	floating point

Figure 3.5.3 Exception codes[11]

3.5.3 EPC Register

The Exception Program Counter (EPC) register is used to store the exception return address. For example, a "jal" instruction is executed to call a procedure and the return address should be stored in a saved register, $ra. However, the return address cannot be stored in $ra because it may overlap the address which has been stored before the exception occurs. The EPC register solve the problem by storing the address of the executing instruction when there is an exception occurring[5].

3.6 Tools Involved

3.6.1 LLVM

LLVM compiler is split into 3 parts which are front-end, middle-end, back-end. LLVM intermediate representation (LLVM IR) is used by LLVM compiler framework. Clang is the front-end processing the C source code while MIPS is in the back-end compiles IR to machine code (MIPS). In middle-end, LLVM contains opt(optimizer), llc(compiler), lld(linker). Clang provides optimization which optimizes from high level language to IR.

First, opt is a modular LLVM optimizer which it takes LLVM source file as input and run specified optimizations to output the optimized file. It performs various analyses of the input source and print the results on standard output.

Second, llc is a second module in LLVM. It compiles LLVM IR into assembly codes(.s) or it will generate object file(.o). The assembly code output is then passed through the linker.

Third, lld is the last module in LLVM and it acts as a linker which links all the object files into an Executable and Linkable file(.elf). Then, the file is passed to last stage which will strip the .elf file to raw binary files such as .txt, .dat, and .rotdata segment. Therefore, MIPS assembly instruction can be obtained for execution. Figure 3.6.1 shows the simplified LLVM architecture of RISC32[4].

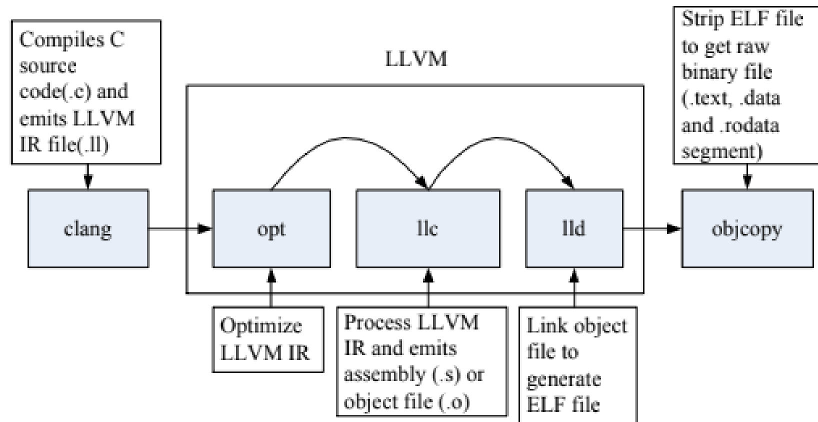


Figure 3.6.1 Simplified architecture of LLVM of RISC32[4]

3.6.2 Xilinx Vivado

Xilinx Vivado is a complex integrated development environment (IDE) tool to program FPGA and the implementation process. To create a Verilog files, the source codes can be written in the text editor supported by Vivado. Testbench code can be added as simulation source code. By having the project source code, the user can choose different types of simulation for their project. For example, behavioural simulation will launch the built-in simulator. After simulating the project, there is schematic view of the design according to the user source code.

3.7 Implementation Issues and Challenge

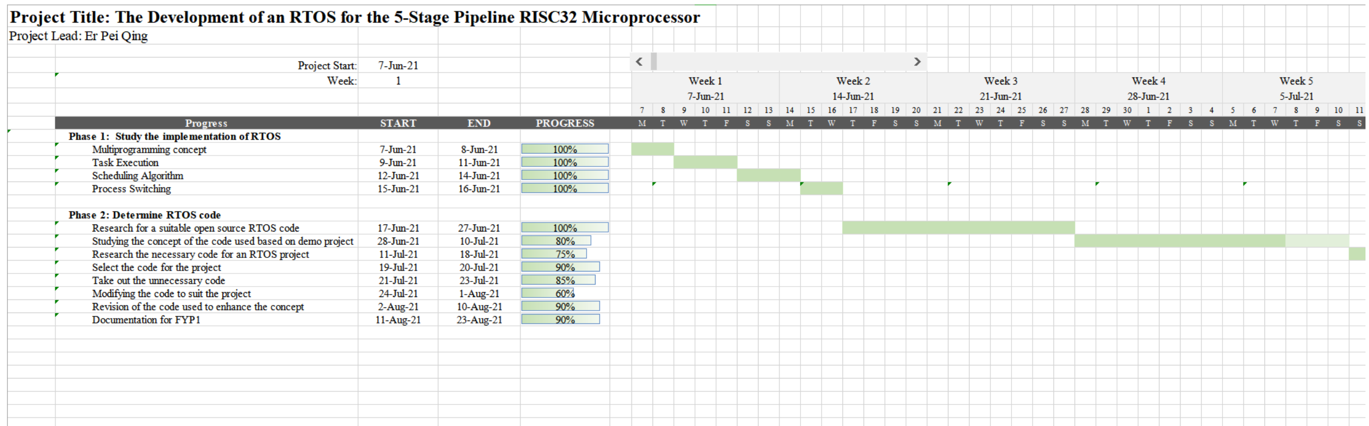
In this project, we may add new instruction in the file which is done in the previous work if the RISC32 processor does not support the special library function. Once the compiler compiles successfully, the address of the MIPS instruction can be readjusted in order to be read by the RISC32 processor.

Besides, the modification of the existing code may occur logic error due to the setting of code for our project are imprecise. Repeating error checking is needed when compiling the code.

3.8 Timeline

Gantt Chart is used to describe our project:

3.8.1 Timeline of FYP 1

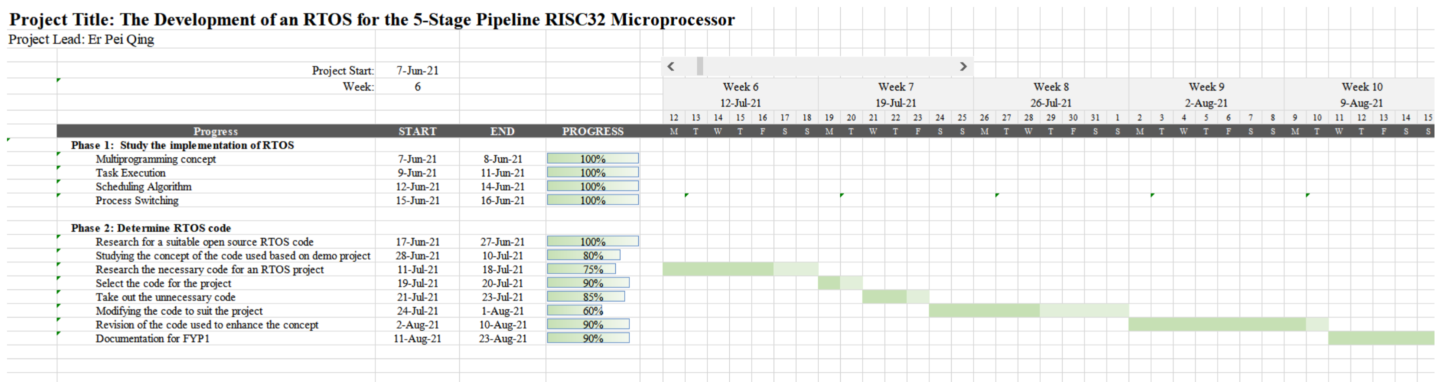


Project Title: The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor
Project Lead: Er Pei Qing

Project Start: 7-Jun-21
Week: 1

Progress	START	END	PROGRESS
Phase 1: Study the implementation of RTOS			
Multiprogramming concept	7-Jun-21	8-Jun-21	100%
Task Execution	9-Jun-21	11-Jun-21	100%
Scheduling Algorithm	12-Jun-21	14-Jun-21	100%
Process Switching	15-Jun-21	16-Jun-21	100%
Phase 2: Determine RTOS code			
Research for a suitable open source RTOS code	17-Jun-21	27-Jun-21	100%
Studying the concept of the code used based on demo project	28-Jun-21	10-Jul-21	80%
Research the necessary code for an RTOS project	11-Jul-21	18-Jul-21	75%
Select the code for the project	19-Jul-21	20-Jul-21	90%
Take out the unnecessary code	21-Jul-21	23-Jul-21	85%
Modifying the code to suit the project	24-Jul-21	1-Aug-21	60%
Revision of the code used to enhance the concept	2-Aug-21	10-Aug-21	90%
Documentation for FYP1	11-Aug-21	23-Aug-21	90%

Figure 3.8.1 Project Progress from Week 1 to Week 5



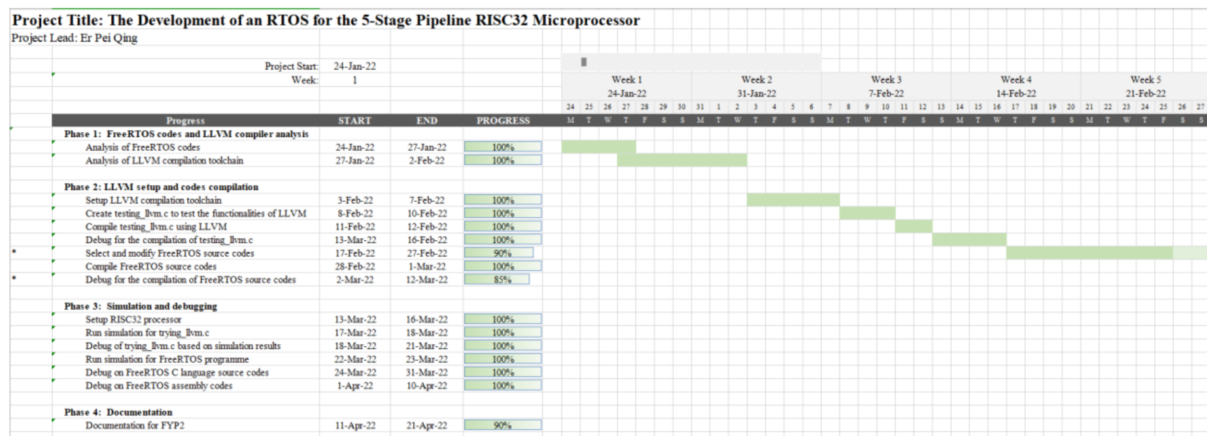
Project Title: The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor
Project Lead: Er Pei Qing

Project Start: 7-Jun-21
Week: 6

Progress	START	END	PROGRESS
Phase 1: Study the implementation of RTOS			
Multiprogramming concept	7-Jun-21	8-Jun-21	100%
Task Execution	9-Jun-21	11-Jun-21	100%
Scheduling Algorithm	12-Jun-21	14-Jun-21	100%
Process Switching	15-Jun-21	16-Jun-21	100%
Phase 2: Determine RTOS code			
Research for a suitable open source RTOS code	17-Jun-21	27-Jun-21	100%
Studying the concept of the code used based on demo project	28-Jun-21	10-Jul-21	80%
Research the necessary code for an RTOS project	11-Jul-21	18-Jul-21	75%
Select the code for the project	19-Jul-21	20-Jul-21	90%
Take out the unnecessary code	21-Jul-21	23-Jul-21	85%
Modifying the code to suit the project	24-Jul-21	1-Aug-21	60%
Revision of the code used to enhance the concept	2-Aug-21	10-Aug-21	90%
Documentation for FYP1	11-Aug-21	23-Aug-21	90%

Figure 3.8.2 Project Progress from Week 6 to Week 10

3.8.2 FYP2 Timeline

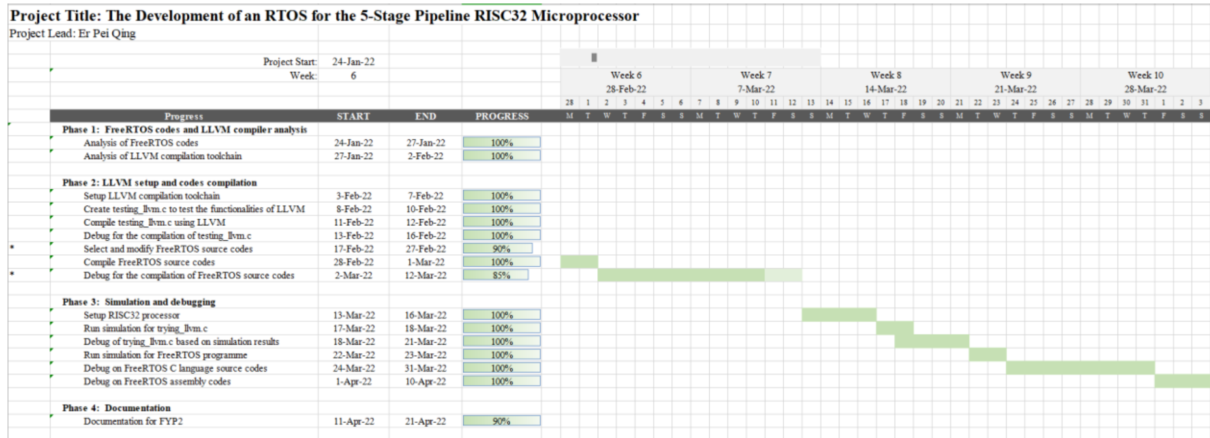


Project Title: The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor
Project Lead: Er Pei Qing

Project Start: 24-Jan-22
Week: 1

Progress	START	END	PROGRESS
Phase 1: FreeRTOS codes and LLVM compiler analysis			
Analysis of FreeRTOS codes	24-Jan-22	27-Jan-22	100%
Analysis of LLVM compilation toolchain	27-Jan-22	2-Feb-22	100%
Phase 2: LLVM setup and codes compilation			
Setup LLVM compilation toolchain	3-Feb-22	7-Feb-22	100%
Create testing_llvm.c to test the functionalities of LLVM	8-Feb-22	10-Feb-22	100%
Compile testing_llvm.c using LLVM	11-Feb-22	12-Feb-22	100%
Debug for the compilation of testing_llvm.c	13-Mar-22	16-Feb-22	100%
Select and modify FreeRTOS source codes	17-Feb-22	27-Feb-22	90%
Compile FreeRTOS source codes	28-Feb-22	1-Mar-22	100%
Debug for the compilation of FreeRTOS source codes	2-Mar-22	12-Mar-22	85%
Phase 3: Simulation and debugging			
Setup RISC32 processor	13-Mar-22	16-Mar-22	100%
Run simulation for trying_llvm.c	17-Mar-22	18-Mar-22	100%
Debug of trying_llvm.c based on simulation results	18-Mar-22	21-Mar-22	100%
Run simulation for FreeRTOS programme	22-Mar-22	23-Mar-22	100%
Debug on FreeRTOS C language source codes	24-Mar-22	31-Mar-22	100%
Debug on FreeRTOS assembly codes	1-Apr-22	10-Apr-22	100%
Phase 4: Documentation			
Documentation for FYP2	11-Apr-22	21-Apr-22	90%

Figure 3.8.3 Project Progress from Week 1 to Week 5



Project Title: The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor

Project Lead: Er Pei Qing

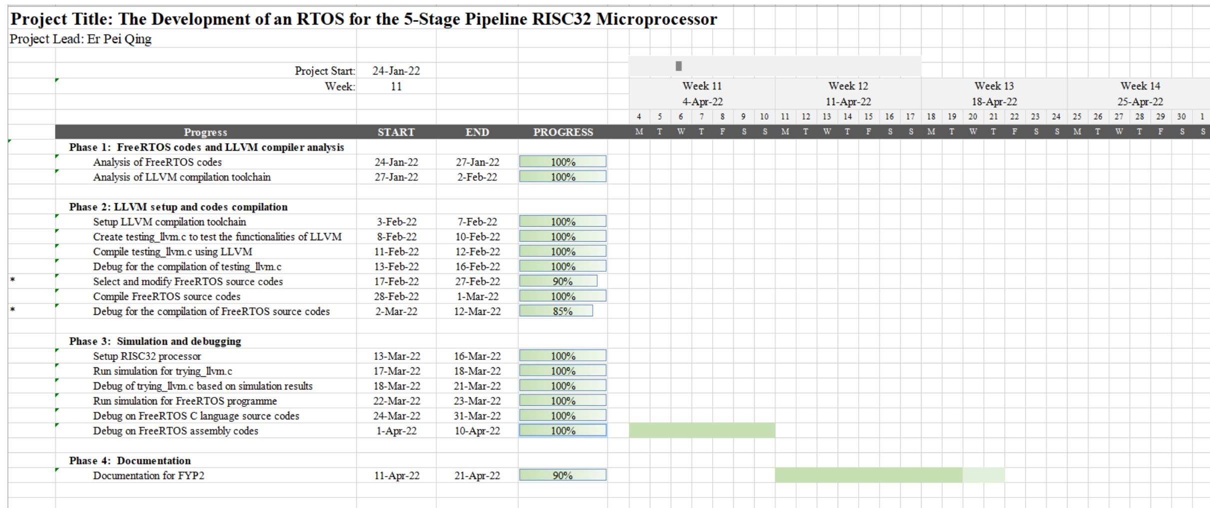
Project Start: 24-Jan-22

Week: 6

Week 6 (28-Feb-22) | Week 7 (7-Mar-22) | Week 8 (14-Mar-22) | Week 9 (21-Mar-22) | Week 10 (28-Mar-22)

Progress	START	END	PROGRESS
Phase 1: FreeRTOS codes and LLVM compiler analysis			
Analysis of FreeRTOS codes	24-Jan-22	27-Jan-22	100%
Analysis of LLVM compilation toolchain	27-Jan-22	2-Feb-22	100%
Phase 2: LLVM setup and codes compilation			
Setup LLVM compilation toolchain	3-Feb-22	7-Feb-22	100%
Create testing_llvm.c to test the functionalities of LLVM	8-Feb-22	10-Feb-22	100%
Compile testing_llvm.c using LLVM	11-Feb-22	12-Feb-22	100%
Debug for the compilation of testing_llvm.c	13-Feb-22	16-Feb-22	100%
Select and modify FreeRTOS source codes	17-Feb-22	27-Feb-22	90%
Compile FreeRTOS source codes	28-Feb-22	1-Mar-22	100%
Debug for the compilation of FreeRTOS source codes	2-Mar-22	12-Mar-22	85%
Phase 3: Simulation and debugging			
Setup RISC32 processor	13-Mar-22	16-Mar-22	100%
Run simulation for trying_llvm.c	17-Mar-22	18-Mar-22	100%
Debug of trying_llvm.c based on simulation results	18-Mar-22	21-Mar-22	100%
Run simulation for FreeRTOS programme	22-Mar-22	23-Mar-22	100%
Debug on FreeRTOS C language source codes	24-Mar-22	31-Mar-22	100%
Debug on FreeRTOS assembly codes	1-Apr-22	10-Apr-22	100%
Phase 4: Documentation			
Documentation for FYP2	11-Apr-22	21-Apr-22	90%

Figure 3.8.4 Project Progress from Week 6 to Week 10CHAPTER 4: Analysis and Modification of FreeRTOS



Project Title: The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor

Project Lead: Er Pei Qing

Project Start: 24-Jan-22

Week: 11

Week 11 (4-Apr-22) | Week 12 (11-Apr-22) | Week 13 (18-Apr-22) | Week 14 (25-Apr-22)

Progress	START	END	PROGRESS
Phase 1: FreeRTOS codes and LLVM compiler analysis			
Analysis of FreeRTOS codes	24-Jan-22	27-Jan-22	100%
Analysis of LLVM compilation toolchain	27-Jan-22	2-Feb-22	100%
Phase 2: LLVM setup and codes compilation			
Setup LLVM compilation toolchain	3-Feb-22	7-Feb-22	100%
Create testing_llvm.c to test the functionalities of LLVM	8-Feb-22	10-Feb-22	100%
Compile testing_llvm.c using LLVM	11-Feb-22	12-Feb-22	100%
Debug for the compilation of testing_llvm.c	13-Feb-22	16-Feb-22	100%
Select and modify FreeRTOS source codes	17-Feb-22	27-Feb-22	90%
Compile FreeRTOS source codes	28-Feb-22	1-Mar-22	100%
Debug for the compilation of FreeRTOS source codes	2-Mar-22	12-Mar-22	85%
Phase 3: Simulation and debugging			
Setup RISC32 processor	13-Mar-22	16-Mar-22	100%
Run simulation for trying_llvm.c	17-Mar-22	18-Mar-22	100%
Debug of trying_llvm.c based on simulation results	18-Mar-22	21-Mar-22	100%
Run simulation for FreeRTOS programme	22-Mar-22	23-Mar-22	100%
Debug on FreeRTOS C language source codes	24-Mar-22	31-Mar-22	100%
Debug on FreeRTOS assembly codes	1-Apr-22	10-Apr-22	100%
Phase 4: Documentation			
Documentation for FYP2	11-Apr-22	21-Apr-22	90%

Figure 3.8. 5 Project Progress from Week 11 to Week 13

Chapter 4 Analysis and Modification of FreeRTOS

4.1 Analysis of FreeRTOS Architecture

Real Time Engineering Ltd has distributed each of the source file and header file into several parts according to their functionality. There are 4 folders stored in FreeRTOS folder which are “Demo”, “License”, “Source” and “Test”. For the project, only “Demo” and “Source” files are needed. They are used as FreeRTOS kernel structure.

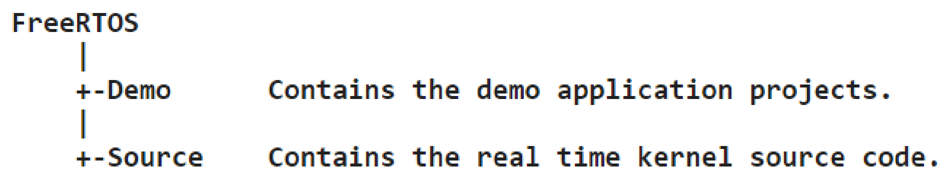


Figure 4.1.1 FreeRTOS kernel directory structure[12]

4.2 Demo Path

This folder includes around 200 examples for every microarchitecture and compiler. For our project, we refer to the 6 examples of RISC-V which are tested to different devices.

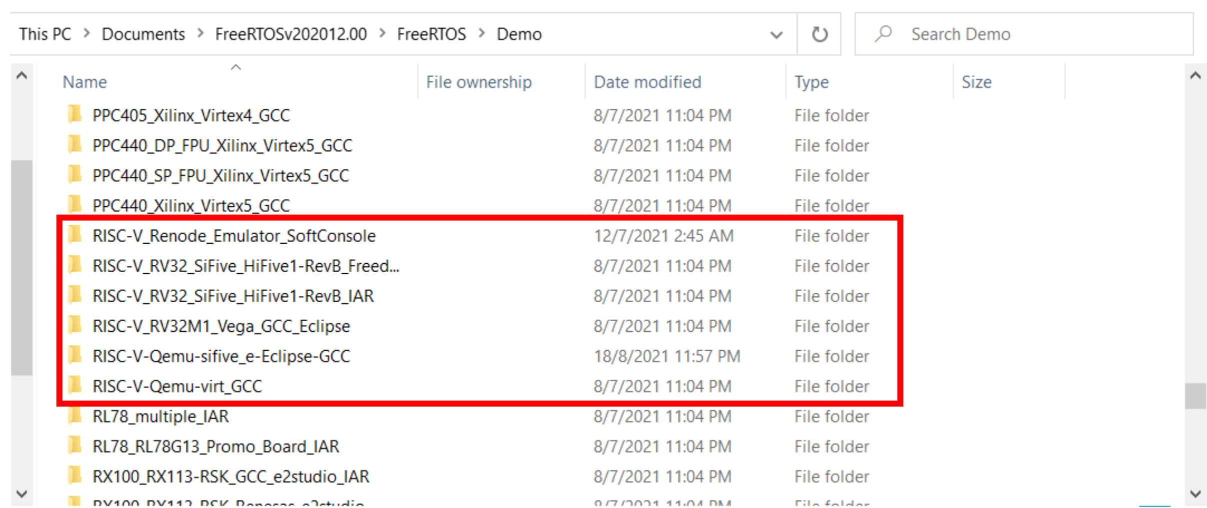


Figure 4.2.1 Six RISC-V example projects

4.3 Source Path

This directory contains 7 source (.c)files for the RTOS core but we only take those necessary source files for RISC32. In "Source" directory, the "include" file contains all the header files used by C source files. The "portable" file consists of a lot of compilers and each of the compiler has their own port layer source files.

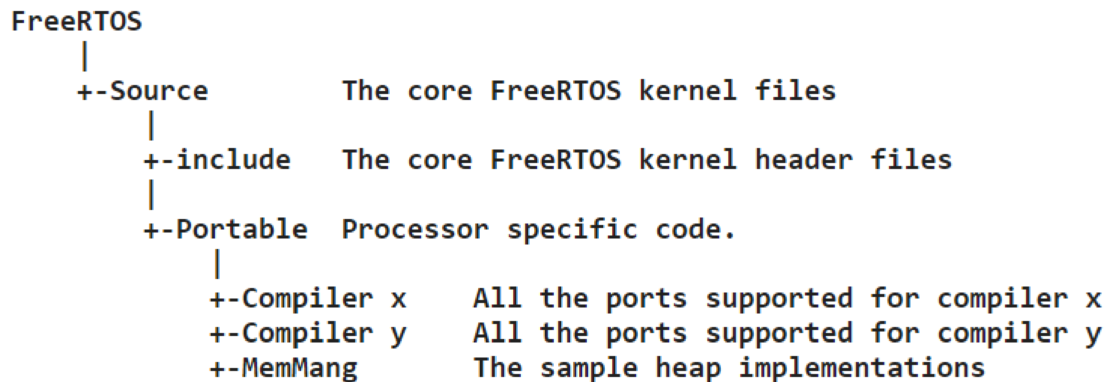


Figure 4.3.1 "Source" folder structure [12]

There is numerous source file which are used for the core RTOS. Each source file contains their own functionality. For example, task.c file is used to create task, set the task priority value and scheduling the task. Due to our project only need these 3 files, the other two optional files are not included to make sure the code size is small. The functionality of source files of list.c, queue.c and tasks.c are discussed in Chapter 5.

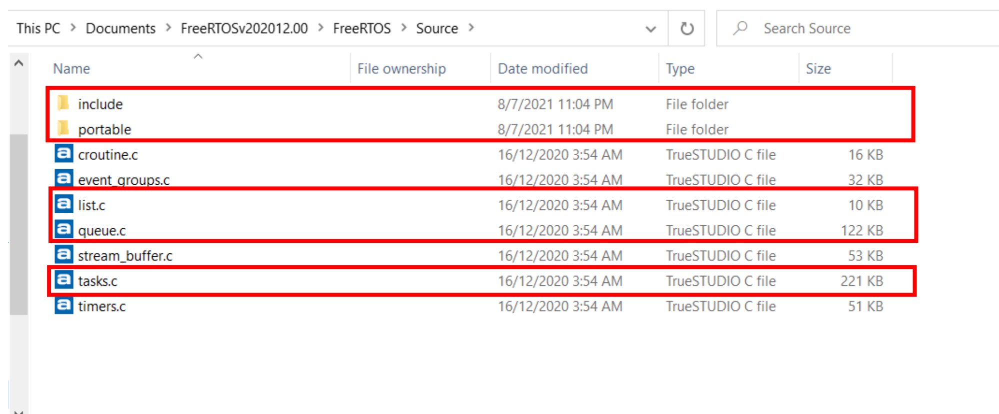


Figure 4.3.2 Minimal files needed to build real time kernel

4.3.1 The include Folder

From Figure 4.3.3, the folder "include" contains the header files used by C source files.

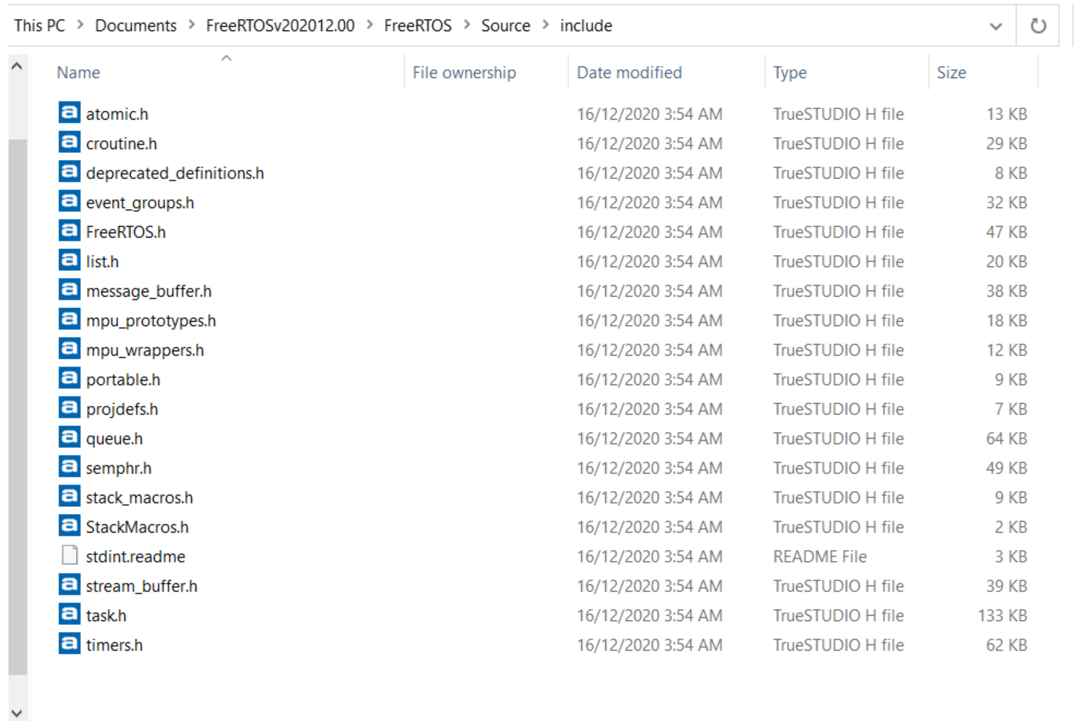


Figure 4.3.3 Header files of FreeRTOS kernel

4.3.2 The portable Folder

This folder contains specific code to particular microcontroller and the compiler. For our project, we use GCC compiler for RISC-V as the reference.

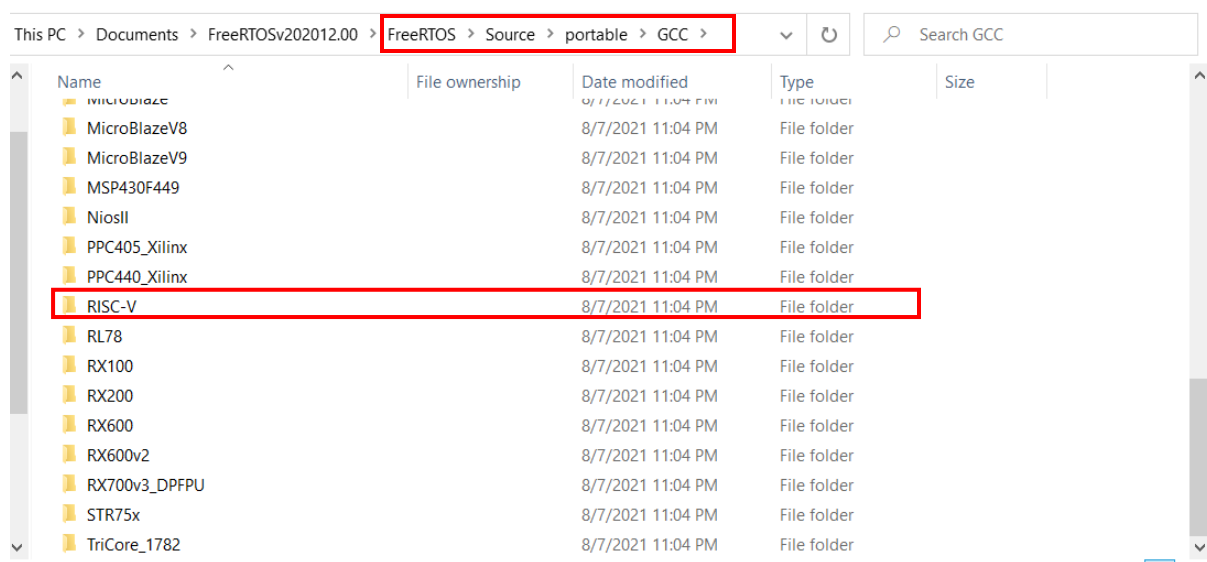


Figure 4.3.4 RISC-V uses GCC compiler

4.4 FreeRTOS Kernel Architecture and its Usage

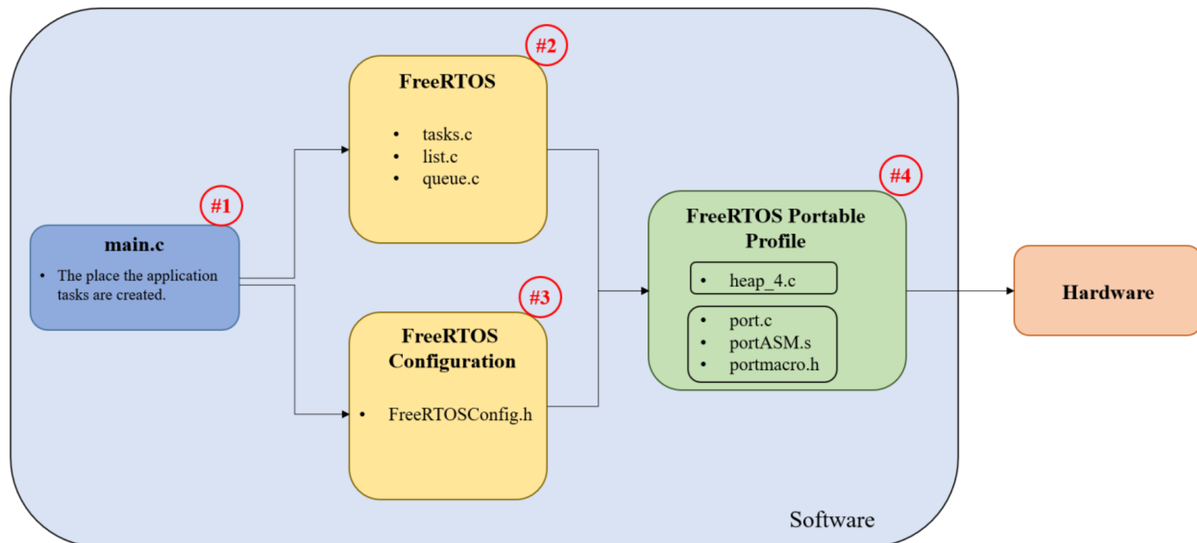


Figure 4.4.1 FreeRTOS Kernel Architecture

Demo application tasks can be created in main() function in main.c because it acts as the entry point to FreeRTOS. The source files contained in #2 and #3 provides the FreeRTOS API to the application. The source files in #2 are the basic files for FreeRTOS core. FreeRTOS can be configured by tailoring FreeRTOSConfig.h in order to match RISC32 specific application. Source codes in #4 play important roles to connect FreeRTOS to the hardware model and compiler. There are 2 specific data type in portmacro.h that are the TickType_t and BaseType_t which will be discussed later. Besides, inline assembly are used as it enables the assembly instruction to be embedded within C code. The syntax of inline assembly is "__asm" associated with volatile keyword.

4.5 FreeRTOS Functions and Code Analysis and Modification

4.5.1 Source Files of FreeRTOS Core

FreeRTOS can be defined as a small application, the core of FreeRTOS consists of 6 source files and their respective header files, and the total code needed is under 10k lines. These 6 files are tasks.c, list.c, and queue.c, they are common to all the FreeRTOS ports. The header files are stored in another folder named "include"[12].

- task.c - Almost 50% of FreeRTOS's core code are dealing with task in an operating system. Task is user-defined C function as the user assigns the priority value to the task. This part involves in creating, scheduling and maintaining the tasks.

```
/* pxDelayedTaskList and pxOverflowDelayedTaskList are switched when the tick
 * count overflows. */
#define taskSWITCH_DELAYED_LISTS()                                              \
    {                                                                           \
        List_t * pxTemp;                                                        \
                                                                                \
        /* The delayed tasks list should be empty when the lists are switched. */ \
        configASSERT( ( listLIST_IS_EMPTY( pxDelayedTaskList ) ) );             \
                                                                                \
        pxTemp = pxDelayedTaskList;                                             \
        pxDelayedTaskList = pxOverflowDelayedTaskList;                          \
        pxOverflowDelayedTaskList = pxTemp;                                     \
        xNumOfOverflows++;                                                      \
        prvResetNextTaskUnblockTime();                                          \
    }

/*-----------------------------------------------------------*/

/*
 * Place the task represented by pxTCB into the appropriate ready list for
 * the task.  It is inserted at the end of the list.
 */
#define prvAddTaskToReadyList( pxTCB )                                                          \
    traceMOVED_TASK_TO_READY_STATE( pxTCB );                                                    \
    taskRECORD_READY_PRIORITY( ( pxTCB )->uxPriority );                                         \
    vListInsertEnd( &( pxReadyTasksLists[ ( pxTCB )->uxPriority ] ), &( ( pxTCB )->xStateListItem ) ); \
    tracePOST_MOVED_TASK_TO_READY_STATE( pxTCB )
/*-----------------------------------------------------------*/
```

Figure 4.5.1 Example code of task.c

- list.c - It is a data structure and the list implementation is used by the scheduler. Its concept is similar to linked list that the items is stored in the list, but it is used to track tasks in FreeRTOS. list.c defines the structures and functions used by task.c [6]. There are 2 types of list items used in FreeRTOS which are list items and mini-list items.

```
void vListInitialise( List_t * const pxList )
{
    /* The list structure contains a list item which is used to mark the
     * end of the list.  To initialise the list the list end is inserted
     * as the only list entry. */
    pxList->pxIndex = ( ListItem_t * ) &( pxList->xListEnd );

    /* The list end value is the highest possible value in the list to
     * ensure it remains at the end of the list. */
    pxList->xListEnd.xItemValue = portMAX_DELAY;

    /* The list end next and previous pointers point to itself so we know
     * when the list is empty. */
    pxList->xListEnd.pxNext = ( ListItem_t * ) &( pxList->xListEnd );
    pxList->xListEnd.pxPrevious = ( ListItem_t * ) &( pxList->xListEnd );

    pxList->uxNumberOfItems = ( UBaseType_t ) 0U;

    /* Write known values into the list if
     * configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
    listSET_LIST_INTEGRITY_CHECK_1_VALUE( pxList );
    listSET_LIST_INTEGRITY_CHECK_2_VALUE( pxList );
}
/*-----------------------------------------------------------*/

void vListInitialiseItem( ListItem_t * const pxItem )
{
    /* Make sure the list item is not recorded as being on a list. */
    pxItem->pxContainer = NULL;

    /* Write known values into the list item if
     * configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
    listSET_FIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE( pxItem );
    listSET_SECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE( pxItem );
}
/*-----------------------------------------------------------*/
```

Figure 4.5.2 Example code of list.c

- queue.c – Perform thread-safe queue for synchronisation and task communication[6]. In order to make the task can communicate with each other, queue.c and queue.h are used to handle FreeRTOS task communication. Using queue enables task and interrupt to send data between each other and to signal the other tasks that the critical resources are used by editing the value of semaphores and mutexes.

```
typedef struct QueuePointers
{
    int8_t * pcTail;     /*< Points to the byte at the end of the queue storage area.
    int8_t * pcReadFrom; /*< Points to the last place that a queued item was read from
} QueuePointers_t;

typedef struct SemaphoreData
{
    TaskHandle_t xMutexHolder;        /*< The handle of the task that holds the mutex.
    UBaseType_t uxRecursiveCallCount; /*< Maintains a count of the number of times a re
} SemaphoreData_t;

/* Semaphores do not actually store or copy data, so have an item size of
 * zero. */
#define queueSEMAPHORE_QUEUE_ITEM_LENGTH    ( ( UBaseType_t ) 0 )
#define queueMUTEX_GIVE_BLOCK_TIME          ( ( TickType_t ) 0U )

#if ( configUSE_PREEMPTION == 0 )

/* If the cooperative scheduler is being used then a yield should not be
 * performed just because a higher priority task has been woken. */
    #define queueYIELD_IF_USING_PREEMPTION()
#else
    #define queueYIELD_IF_USING_PREEMPTION()    portYIELD_WITHIN_API()
#endif
```

Figure 4.5.3 Example codes of queue.c

4.5.2 Configuring RTOS Scheduler

To make FreeRTOS can be implemented in the project, a configuration file called FreeRTOSConfig.h should be modified to tailor the RTOS kernel. Figures below will display the code which contributes to the settings.

```
#define configUSE_PREEMPTION            1
#define configUSE_IDLE_HOOK             1
#define configUSE_TICK_HOOK             1
#define configCPU_CLOCK_HZ              ( ( uint32_t ) ( SYS_CLK_FREQ ) )
#define configTICK_RATE_HZ              ( ( TickType_t ) 1000 )
#define configMAX_PRIORITIES            ( 5 )
#define configMINIMAL_STACK_SIZE        ( ( uint32_t ) 170 )
#define configTOTAL_HEAP_SIZE           ( ( size_t ) ( 4096 ) )
#define configMAX_TASK_NAME_LEN         ( 16 )
#define configUSE_TRACE_FACILITY        0
#define configUSE_16_BIT_TICKS          0
#define configIDLE_SHOULD_YIELD         0
#define configUSE_MUTEXES               1
#define configQUEUE_REGISTRY_SIZE       8
#define configCHECK_FOR_STACK_OVERFLOW  2
#define configUSE_RECURSIVE_MUTEXES     1
#define configUSE_MALLOC_FAILED_HOOK    1
#define configUSE_APPLICATION_TASK_TAG  0
#define configUSE_COUNTING_SEMAPHORES   1
#define configGENERATE_RUN_TIME_STATS   0
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 1
```

Figure 4.5.4 Part 1 settings

According to the setting above, the pre-emptive RTOS scheduler is used. Idle hook and Tick hook functions are set to 1. It is because idle task can be created automatically when the scheduler begins to check there is at least one task is executed on the processor. The stack size of idle task is set by using configMINIMAL_STACK_SIZE(). In RISC32, the heap segment size is 0x1000 meaning that 4096 of size in terms of decimal base is assigned to the label configTOTAL_HEAP_SIZE(). Tick interrupt is able to implement timer functionality by calling vApplicationTickHook() function. RTOS tick interrupt frequency can be modified by setting the value for configTICK_RATE_HZ() function.

Each task is assigned with a priority value from 0 to 4. The maximum value set cannot be higher than 32 to ensure RAM usage efficiency. The scheduler makes sure that the task is given processor time in preference although it has lower priority in ready state while the higher priority task is running on the processor.

Mutex functionality is included in RTOS scheduler. Mutex is a locking mechanism which ensures that only 1 thread can acquire the critical session at one time. Mutex will available when the thread finishes using the critical session and release it. Mutex is different than semaphore as semaphore is a signalling mechanism. Figure 4.2.2 is an example of mutex.

```
wait (mutex);
…..
Critical Section
…..
signal (mutex);
```

Figure 4.5.5 Mutex Example

Task may come to deadlock when the processes is competing for the system resources or communicating with each other. When a task tries to acquire the mutex for more than once without returning the mutex first. As a result, the task in Blocked state is waiting for the mutex to be returned but the mutex is holding by the task itself. Therefore, to avoid deadlock from happening, a function named configUSE_RECURSIVE_MUTEX is set to 1. Doing so allows a task can take more than 1 mutex.

```
/* Co-routine definitions. */
#define configUSE_CO_ROUTINES           0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

/* Software timer definitions. */
#define configUSE_TIMERS                1
#define configTIMER_TASK_PRIORITY       ( configMAX_PRIORITIES - 1 )
#define configTIMER_QUEUE_LENGTH        4
#define configTIMER_TASK_STACK_DEPTH    ( configMINIMAL_STACK_SIZE )

/* Task priorities.  Allow these to be overridden. */
#ifndef uartPRIMARY_PRIORITY
    #define uartPRIMARY_PRIORITY        ( configMAX_PRIORITIES - 3 )
#endif

/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */
#define INCLUDE_vTaskPrioritySet            1
#define INCLUDE_uxTaskPriorityGet           1
#define INCLUDE_vTaskDelete                 1
#define INCLUDE_vTaskCleanUpResources       1
#define INCLUDE_vTaskSuspend                1
#define INCLUDE_vTaskDelayUntil             1
#define INCLUDE_vTaskDelay                  1
#define INCLUDE_eTaskGetState               1
#define INCLUDE_xTimerPendFunctionCall      1
#define INCLUDE_xTaskAbortDelay             1
#define INCLUDE_xTaskGetHandle              1
#define INCLUDE_xSemaphoreGetMutexHolder    1

/* Normal assert() semantics without relying on the provision of an assert.h
header file. */
#define configASSERT( x ) if( ( x ) == 0 ) { taskDISABLE_INTERRUPTS(); __asm volatile( "ebreak" ); for( ;; ); }

/* Defined in main.c and used in main_blinky.c and main_full.c. */
void vSendString( const char * const pcString );
```

Figure 4.5.6 Part 2 setting

From Figure 5.3.3, configUSE_CO_ROUTINE is set to 0 because the croutine.c source file is excluded from "Source" directory and it is not used in RISC-V. The co-routines priority value is set to 2 meaning that only 2 priorities is available and each co-routine share the same priority. Software timer is used in the scheduler to allow function executes at a set of time in future.

There are macros "INCLUDE" where the function name is actually the API function. From the figure above, all the API function is set to 1.

4.5.3 Portable Layer

The processor architecture containing specific RTOS code is stored in a folder called "portable" as it acts as RTOS portable layer. Memory management code is also included in portable file.

- heap_4.c (in Memory Management path) – The RTOS kernel needs RAM when a task, queue, or semaphore is created. It uses the first-fit algorithm and combine adjacent free memory blocks into a large block. Using heap_4.c allows the application can delete tasks, queue, semaphore or mutex repeatedly. To set the total amount of available heap size, configTOTAL_HEAP_SIZE in FreeRTOSConfig.h is configured to the values of 4096 with unsigned integer data type. Calling pvPortMalloc() when the kernel needs RAM while calling vPortFree to free RAM.

```c
/*
 * Inserts a block of memory that is being freed into the correct position in
 * the list of free memory blocks.  The block being freed will be merged with
 * the block in front it and/or the block behind it if the memory blocks are
 * adjacent to each other.
 */
static void prvInsertBlockIntoFreeList( BlockLink_t * pxBlockToInsert ) PRIVILEGED_FUNCTION;

/*
 * Called automatically to setup the required heap structures the first time
 * pvPortMalloc() is called.
 */
static void prvHeapInit( void ) PRIVILEGED_FUNCTION;

/*-----------------------------------------------------------*/

/* The size of the structure placed at the beginning of each allocated memory
 * block must by correctly byte aligned. */
static const size_t xHeapStructSize = ( sizeof( BlockLink_t ) + ( ( size_t ) ( portBYTE_ALIGNMENT - 1 ) ) ) & ~( ( size_t ) portBYTE_ALIGNMENT_MASK );

/* Create a couple of list links to mark the start and end of the list. */
PRIVILEGED_DATA static BlockLink_t xStart, * pxEnd = NULL;

/* Keeps track of the number of calls to allocate and free memory as well as the
 * number of free bytes remaining, but says nothing about fragmentation. */
PRIVILEGED_DATA static size_t xFreeBytesRemaining = 0U;
PRIVILEGED_DATA static size_t xMinimumEverFreeBytesRemaining = 0U;
PRIVILEGED_DATA static size_t xNumberOfSuccessfulAllocations = 0;
PRIVILEGED_DATA static size_t xNumberOfSuccessfulFrees = 0;
```

Figure 4.5.7 Example function prototypes of heap_4.c

- port.c (in GCC compiler path) – port.c file implements the functions specified in portable.h file. portable.h file is the portable layer API and the function must be defined for each port. Figure 4.4.8 shows the function prototypes defined in portable.h and the function definitions are created at port.c.

```
/*
 * Returns a HeapStats_t structure filled with information about the current
 * heap state.
 */
void vPortGetHeapStats( HeapStats_t * pxHeapStats );

/*
 * Map to the memory management routines required for the port.
 */
void * pvPortMalloc( size_t xSize ) PRIVILEGED_FUNCTION;
void vPortFree( void * pv ) PRIVILEGED_FUNCTION;
void vPortInitialiseBlocks( void ) PRIVILEGED_FUNCTION;
size_t xPortGetFreeHeapSize( void ) PRIVILEGED_FUNCTION;
size_t xPortGetMinimumEverFreeHeapSize( void ) PRIVILEGED_FUNCTION;

/*
 * Setup the hardware ready for the scheduler to take control.  This generally
 * sets up a tick interrupt and sets timers for the correct tick frequency.
 */
BaseType_t xPortStartScheduler( void ) PRIVILEGED_FUNCTION;

/*
 * Undo any hardware/ISR setup that was performed by xPortStartScheduler() so
 * the hardware is left in its original condition after the scheduler stops
 * executing.
 */
void vPortEndScheduler( void ) PRIVILEGED_FUNCTION;
```

Figure 4.5.8 Example function prototypes defined in portable.h

```
BaseType_t xPortStartScheduler( void )
{
extern void xPortStartFirstTask( void );

    #if( configASSERT_DEFINED == 1 )
    {
        volatile uint32_t mtvec = 0;

        /* Check the least significant two bits of mtvec are 00 - indicating
        single vector mode. */

        //__asm volatile( "csrr %0, mtvec" : "=r"( mtvec ) );
        //configASSERT( ( mtvec & 0x03UL ) == 0 );

        /* Check alignment of the interrupt stack - which is the same as the
        stack that was being used by main() prior to the scheduler being
        started. */
        configASSERT( ( xISRStackTop & portBYTE_ALIGNMENT_MASK ) == 0 );

        #ifdef configISR_STACK_SIZE_WORDS
        {
            // memset is used to fill a block of memory with particular value.
            memset( ( void * ) xISRStack, portISR_STACK_FILL_BYTE, sizeof( xISRStack ) );
        }
        #endif  /* configISR_STACK_SIZE_WORDS */
    }
    #endif /* configASSERT_DEFINED */

    /* If there is a CLINT then it is ok to use the default implementation…
    vPortSetupTimerInterrupt();

    #if( ( configMTIME_BASE_ADDRESS != 0 ) && ( configMTIMECMP_BASE_ADDRESS != 0 ) )…
    #else
    {
        /* Enable external interrupts. */
        //__asm volatile( "csrs mie, %0" :: "r"(0x800) );
        portENABLE_INTERRUPTS();
    }
    #endif /* ( configMTIME_BASE_ADDRESS != 0 ) && ( configMTIMECMP_BASE_ADDRESS != 0 ) */

    xPortStartFirstTask();

    /* Should not get here as after calling xPortStartFirstTask() only tasks
    should be executing. */
    return pdFAIL;
}
```

#1

#2

Figure 4.5.9 Function definition of xPortStartFirstTask defined in port.c

Since the label configASSERT_DEFINED is set to 1 in FreeRTOS.h, the codes inside the label will be executed. In #1, the inline assembly is used for RISC-V to check whether the register mtvec is single vector mode. When RISC-V is on single vector mode, the register mtvec will point to the ISR base address. Then, the meaning of single vector mode in RISC32 is allowing every interrupt jumps to a single general routine in order to overcome the problem of multiple interrupts occur at the same time. An EXL bit of $stat in CP0 is set to 1 to disable further interrupts. Since the concept of single vector mode on RISC-V and RISC32 is similar, and RISC32 is using single vector mode, the code of checking mode can be ignored.

After analysing the code, the default inline assembly codes were written for RISC-V chips instead of RISC32. Thus, the code has to be modified according to the default assembly codes. For example, the coding styles of portDISABLE_INTERRUPTS() are different in RISC32 and RISC-V. In #2 of Figure 4.5.9, RISC-V define a label for portDISABLE_INTERRUPT() by creating a line of inline assembly. However, 2 lines of inline assembly are needed for RISC32 to disable interrupts. Therefore, the way of defining a label to disable interrupt does not work in RISC32. To solve the problem, a void function prototype is created in portmacro.h and its function definition is placed at port.c. To disable interrupts, status register($12) is used because its bit 0 is interrupt enable bit. An instruction called "mtc0", move to coprocessor 0, is used to assign the value into status register in coprocessor 0. Figure 4.5.10 shows the resulting code for RISC32.

RISC32

```
void portDISABLE_INTERRUPTS()
{
    __asm volatile("addi $t0,0x0000\n\t"
            "mtc0 $t0, $12\n\t");
}

void portENABLE_INTERRUPTS()
{
    __asm volatile("addi $t0,0x0001\n\t"
            "mtc0 $t0, $12\n\t");
}
```

RISC-V

```
//#define portDISABLE_INTERRUPTS()  __asm volatile( "csrc mstatus, 8" )
//#define portENABLE_INTERRUPTS()       __asm volatile( "csrs mstatus, 8" )
```

Figure 4.5.10 Difference between RISC32 and RISC-V

- portASM.s – it is an assembler file containing functions that will be used by the source file. For example, main.c will call a function named vTaskStartScheduler() to start the scheduler and the scheduler will call xPortStartFirstTask(). Its function definition is created in portASM.s. Due to portASM.s is taken from RISC-V example, conversion of coding from RISC-V ISA to MIPS2 ISA is needed. There is a big difference between RISC-V ISA and MIPS2 ISA such as register used.

```
.text
xPortStartFirstTask:

    lw $sp, pxCurrentTCB            /* Load pxCurrentTCB. */
    lw $sp, 0( $sp )                    /* Read sp from first TCB member. */

    lw  $ra, 0( $sp ) /* Note for starting the scheduler the exception return address is used as the function return address. */

    lw $t1, 3 * portWORD_SIZE( $sp )
    lw $t2, 4 * portWORD_SIZE( $sp )
    lw $s0, 5 * portWORD_SIZE( $sp )
    lw $s1, 6 * portWORD_SIZE( $sp )
    lw $a0, 7 * portWORD_SIZE( $sp )
    lw $a1, 8 * portWORD_SIZE( $sp )
    lw $a2, 9 * portWORD_SIZE( $sp )
    lw $a3, 10 * portWORD_SIZE( $sp )
    lw $s2, 11 * portWORD_SIZE( $sp )
    lw $s3, 12 * portWORD_SIZE( $sp )
    lw $s4, 13 * portWORD_SIZE( $sp )
    lw $s5, 14 * portWORD_SIZE( $sp )
    lw $s6, 15 * portWORD_SIZE( $sp )
    lw $s7, 16 * portWORD_SIZE( $sp )
    lw $t3, 17 * portWORD_SIZE( $sp )
    lw $t4, 18 * portWORD_SIZE( $sp )
    lw $t5, 19 * portWORD_SIZE( $sp )
    lw $t6, 20 * portWORD_SIZE( $sp )

    lw $t0,21 * portWORD_SIZE( $sp )        // Load $status into $t0
    addi $t0, $t0, 0x01                     // set IE bit of $status

    // Enable Timer Interrupt
    li $t4,0x8000       // timer interrupt enable bit
    ori $t4, $t4,0x1000 // bonk interrupt bit
    ori $t4, $t4,1
    mtc0 $t4, $12

    // Request Timer Interrupt
    lw $v0, 0xffff001c  // read current time
    addi $v0, $v0, 50   // add 50 to current time
    sw $v0, 0xffff001c  // request timer interrupt in 50 cycles
    li $a0, 10
    sw $a0,  0xffff0010($zero)  //derive

    addi    $sp, $sp, portCONTEXT_SIZE
    jr $ra
```

Figure 4.5.11 Example assembly code in portASM.s

```
// RISC-V
* Register     ABI Name     Description                        Saver
* x0           zero         Hard-wired zero                    -
* x1           ra           Return address                     Caller
* x2           sp           Stack pointer                      Callee
* x3           gp           Global pointer                     -
* x4           tp           Thread pointer                     -
* x5-7         t0-2         Temporaries                        Caller
* x8           s0/fp        Saved register/Frame pointer       Callee
* x9           s1           Saved register                     Callee
* x10-11       a0-1         Function Arguments/return values Caller
* x12-17       a2-7         Function arguments                 Caller
* x18-27       s2-11        Saved registers                    Callee
* x28-31       t3-6         Temporaries                        Caller
```

```
// MIPS
 * NAME        NUMBER      Description                                          PRESERVED ACROSS A CALL?
 * $zero        0          The Constant Value 0                                 -
 * $at          1          Assembler Temporary                                  No
 * $v0-$v1      2-3        Values for Function Results&Expression Evaluation    No
 * $a0-$a3      4-7        Arguments                                            No
 * $t0-$t7      8-15       Temporaries                                          No
 * $s0-$s7      16-23      Saved Temporaries                                    Yes
 * $t8-$t9      24-25      Temporaries                                          No
 * $k0-$k1      26-27      Reserved for OS kernel                               No
 * $gp          28         Global pointer                                       Yes
 * $sp          29         Stack pointer                                        Yes
 * $fp          30         Frame pointer                                        Yes
 * $ra          31         Return address                                       No
```

Figure 4.5.12 Difference between RISC-V and RISC32 ISA

- string2.h – Due to RISC32 does not support the default functions of string.h such as memcpy, memset, strlen, and so on, a header file named string2.h is created to solve the problem. Figure 4.5.13 shows the functions used by FreeRTOS are created explicitly. The function named memcpy is used to copy a memory block from one location to the another while memset function is used to fill a block of memory with particular value. The strlen function is used to get the length of a string.

```
void* memcpy(void *dest, void *src, size_t n) {
    int i;
    //cast src and dest to char*
    char *src_char = (char *)src;
    char *dest_char = (char *)dest;
    for (i=0; i<n; i++)
        dest_char[i] = src_char[i]; //copy contents byte by byte

    return dest;
}

void* memset(void *s, int c, size_t len) {
     unsigned char *dst = s;
     while (len > 0) {
         *dst = (unsigned char) c;
         dst++;
         len--;
     }
     return s;
}


unsigned int strlen(const char *s)
{
     unsigned int count = 0;
     while(*s!='\0')
     {
         count++;
         s++;
     }
     return count;
}
#endif
```

Figure 4.5.13 The functions of memcpy, memset, and strlen defined at string2.h

CHAPTER 5: FreeRTOS Implementation

5.1 Setup LLVM compiler as Toolchain

RISC32 has its own compiler called LLVM compiler, it is used to compile the RTOS code. Before installing LLVM compiler, install “cmake” in order to install LLVM into Ubuntu16.04 LTS operating system. To make sure the source file can be compiled by LLVM, focusing on the standard library function that will be used in RTOS code. A bash script called compile.sh is created to compile the source files and it makes debugging phase easier as the bash script content is arranged in correct order and readable.

```
# text address
text_addr=80000000

# codes start
clang  -"$o_level" --target=mips-unknown-linux -mips32 $rt_evt_grp $rt_list $rt_main $rt_queue $rt_tasks $rt_timers $rt_heap4 $rt_port -emit-llvm -S

## Assembly file creation
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_evt_grp" -o "asm_files/$asm_evt_grp"-debug &> debug00.log
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_list" -o "asm_files/$asm_list"-debug &> debug00.log
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_main" -o "asm_files/$asm_main"-debug &> debug00.log
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_queue" -o "asm_files/$asm_queue"-debug &> debug00.log
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_tasks" -o "asm_files/$asm_tasks"-debug &> debug00.log
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_timers" -o "asm_files/$asm_timers"-debug &> debug00.log
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_heap4" -o "asm_files/MemMang/$asm_heap4"-debug &> debug00.log
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_port" -o "asm_files/RISC-V/$asm_port"-debug &> debug00.log
//llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_portASM" -o "asm_files/RISC-V/$asm_portASM"-debug &> debug00.log
### object files creation
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_evt_grp" -filetype=obj -o "obj_files/$obj_evt_grp"
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_list" -filetype=obj -o "obj_files/$obj_list"
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_main" -filetype=obj -o "obj_files/$obj_main"
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_queue" -filetype=obj -o "obj_files/$obj_queue"
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_tasks" -filetype=obj -o "obj_files/$obj_tasks"
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_timers" -filetype=obj -o "obj_files/$obj_timers"
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_heap4" -filetype=obj -o "obj_files/$obj_heap4"
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_port" -filetype=obj -o "obj_files/$obj_port"
//llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_portASM" -filetype=obj -o "obj_files/$obj_portASM"
llvm-mc -assemble -arch=mips -mcpu=risc32 -mattr=+risc32 "$rt_portASM" -filetype=obj -o "obj_files/$obj_portASM"
## link files
ld.lld --Bstatic -script script.ld "obj_files/$obj_evt_grp" "obj_files/$obj_list" "obj_files/$obj_main" "obj_files/$obj_queue" "obj_files/$obj_tasks" "obj_files/$obj_timers"
"obj_files/$obj_heap4" "obj_files/$obj_port" "obj_files/$obj_portASM" -o "elf/$elf_file"
```

Figure 4.5.14 The files are compiled step by step

5.2 Compilation results of LLVM

5.2.1 Testing the LLVM Compilation via UART Communication

Once LLVM compiler is setup as a toolchain to compile FreeRTOS code, a test program is needed to make sure that LLVM compiler is functioning well. A source file named testing_llvm.c is created by using UART. 4 characters(P,Q,R,S) are inserted into UART and the results are shown in below Figure 5.2.1.

```
int main(){
uint8_t status_byte=0; //For UART status reading

//Setup UART
UCR = UCR | 0x82; //UARTCR = 10000010, UARTEN=>1, TXEIE->0, 9600->010

//Send Data to UART
UDR = 0x50;
UDR = 0x51;
UDR = 0x52;
UDR = 0x53;

//Read TX Empty Flag
status_byte = USR & 0x40;

//Poll UART Status before transmitting next word


while(1);
return 0;
}
```

Figure 5.2.1 Source code to be implemented

The LLVM compiler will compile testing_llvm.c into assembly code stored in testing_llvm_o0_o0_dis.txt. The symbol "o0" means least optimization level. Figure 5.2.2 is the assembly code generated by LLVM with least optimization level.

```
testing_llvm_O0_O0:     file format elf32-tradbigmips


Disassembly of section .text:

80000000 <main>:
80000000:       27bdfff0        addiu   sp,sp,-16
80000004:       afbe000c        sw      s8,12(sp)
80000008:       03a0f025        move    s8,sp
8000000c:       afc00008        sw      zero,8(s8)
80000010:       24010000        li      at,0
80000014:       a3c00004        sb      zero,4(s8)
80000018:       3c02bfff        lui     v0,0xbfff
8000001c:       3443fe28        ori     v1,v0,0xfe28
80000020:       90640000        lbu     a0,0(v1)
80000024:       34840082        ori     a0,a0,0x82
80000028:       a0640000        sb      a0,0(v1)
8000002c:       3443fe2a        ori     v1,v0,0xfe2a
80000030:       24040050        li      a0,80
80000034:       a0640000        sb      a0,0(v1)
80000038:       24040051        li      a0,81
8000003c:       a0640000        sb      a0,0(v1)
80000040:       24040052        li      a0,82
80000044:       a0640000        sb      a0,0(v1)
80000048:       24040053        li      a0,83
8000004c:       a0640000        sb      a0,0(v1)
80000050:       3442fe29        ori     v0,v0,0xfe29
80000054:       90420000        lbu     v0,0(v0)
80000058:       30420040        andi    v0,v0,0x40
8000005c:       a3c20004        sb      v0,4(s8)
80000060:       afc10000        sw      at,0(s8)
80000064:       0800001a        j       80000068 <main+0x68>
80000068:       0800001a        j       80000068 <main+0x68>
```

Figure 5.2.2 Assembly code of testing_llvm.c

When LLVM compiles the source codes, object file will also be generated as the requirement to implement the source codes on RISC32 is object file which is storing the hexadecimal codes. Figure 5.2.3 shows the result of the object file being imported into RISC32.

Figure 5.2.3 The result of testing_llvm.c via UART

5.2.2 FreeRTOS Source Code Compilation and Setup

To make sure that FreeRTOS source codes are executable on RISC32, a testing code is written on main.c of FreeRTOS. UART and GPIO are used to sent out the data. Theoretically, main() function will create a queue before inserting the task information into the queue. Once the queue is created, xTaskCreate() is executed to start the task. To implement the task, prvQueueReceiveTask() is called to in order to receive task information into queue. Once the queue has the tasks inside, vTaskStartScheduler() is called to start the scheduler and to run the task on RISC32.

```
xQueue = xQueueCreate( mainQUEUE_LENGTH, sizeof( uint32_t ) );

if( xQueue != NULL )
{
    /* Start the two tasks as described in the comments at the top of this
    file. */
    xTaskCreate( prvQueueReceiveTask,               /* The function that implements the task. */
                "Rx",                               /* The text name assigned to the task - for debug only as it is not used by the kernel. */
                configMINIMAL_STACK_SIZE * 2U,          /* The size of the stack to allocate to the task. */
                NULL,                               /* The parameter passed to the task - not used in this case. */
                mainQUEUE_RECEIVE_TASK_PRIORITY,    /* The priority assigned to the task. */
                NULL );                             /* The task handle is not required, so NULL is passed. */

    xTaskCreate( prvQueueSendTask, "TX", configMINIMAL_STACK_SIZE * 2U, NULL, mainQUEUE_SEND_TASK_PRIORITY, NULL );

    /* Start the tasks and timer running. */
    vTaskStartScheduler();
}
```

Figure 5.2.4 Queue is created and the tasks created are inserted into queue

```
static void prvQueueReceiveTask( void *pvParameters )
{
unsigned long ulReceivedValue;
const unsigned long ulExpectedValue = 100UL;
const char * const pcPassMessage = "Blink\r\n";
const char * const pcFailMessage = "Unexpected value received\r\n";
extern void vToggleLED( void );

    /* Remove compiler warning about unused parameter. */
    ( void ) pvParameters;
    UCR = UCR | 0x82;
    for( ;; )
    {
        /* Wait until something arrives in the queue - this task will block
        indefinitely provided INCLUDE_vTaskSuspend is set to 1 in
        FreeRTOSConfig.h. */
        xQueueReceive( xQueue, &ulReceivedValue, portMAX_DELAY );

        /*  To get here something must have been received from the queue, but
        is it the expected value?  If it is, toggle the LED. */
        if( ulReceivedValue == ulExpectedValue )
        {
            vSendString( pcPassMessage );
            vToggleLED();
            GPIO_DATA = 0x00001111;
            //Send Data to UART
            UDR = 0x50; //Read ciphertext from memory
            UDR = 0x51; //Read ciphertext from memory
            UDR = 0x51; //Read ciphertext from memory
            UDR = 0x51;//Read ciphertext from memory

            ulReceivedValue = 0U;
        }
        else
        {
            vSendString( pcFailMessage );
        }
    }
}
```

Figure 5.2.5 The task will be executed in prvQueueReceiveTask()

For better understanding, FreeRTOS has distributed its functions into several C language source files such as tasks.c, queue.c, list.c, main.c. and the others. To make them interconnected to each other, a linker (lld) is used to link them up. A bash script (compile.sh) stores the commands to be used to compile files. For example, a command with “clang” is used to compile the source files independently. Then, the lld will link those compiled files according to the labels which are the function names inside the files. Figure 5.2.6 shows the example part of assembly codes generated from FreeRTOS source codes.

```
80000d04 <main>:
80000d04:       27bdffe0        addiu   sp,sp,-32
80000d08:       afbf001c        sw      ra,28(sp)
80000d0c:       afbe0018        sw      s8,24(sp)
80000d10:       03a0f025        move    s8,sp
80000d14:       24020000        li      v0,0
80000d18:       a7c00014        sh      zero,20(s8)
80000d1c:       afc20010        sw      v0,16(s8)
80000d20:       0c000350        jal     80000d40 <prvSetupHardware>
80000d24:       0c000357        jal     80000d5c <main_blinky>
80000d28:       8fc20010        lw      v0,16(s8)
80000d2c:       03c0e825        move    sp,s8
80000d30:       8fbe0018        lw      s8,24(sp)
80000d34:       8fbf001c        lw      ra,28(sp)
80000d38:       27bd0020        addiu   sp,sp,32
80000d3c:       03e00008        jr      ra

80000d40 <prvSetupHardware>:
80000d40:       27bdfff8        addiu   sp,sp,-8
80000d44:       afbe0004        sw      s8,4(sp)
80000d48:       03a0f025        move    s8,sp
80000d4c:       03c0e825        move    sp,s8
80000d50:       8fbe0004        lw      s8,4(sp)
80000d54:       27bd0008        addiu   sp,sp,8
80000d58:       03e00008        jr      ra

80000d5c <main_blinky>:
80000d5c:       27bdffd0        addiu   sp,sp,-48
80000d60:       afbf002c        sw      ra,44(sp)
80000d64:       afbe0028        sw      s8,40(sp)
80000d68:       03a0f025        move    s8,sp
80000d6c:       24040001        li      a0,1
80000d70:       24050004        li      a1,4
80000d74:       24060000        li      a2,0
80000d78:       0c0004d0        jal     80001340 <xQueueGenericCreate>
80000d7c:       3c04a002        lui     a0,0xa002
80000d80:       ac820010        sw      v0,16(a0)
80000d84:       8c820010        lw      v0,16(a0)
80000d88:       1040001f        beqz    v0,80000e08 <main_blinky+0xac>
80000d8c:       08000364        j       80000d90 <main_blinky+0x34>
```

Figure 5.2.6 Part of assembly codes generated by LLVM

Figure 5.2.6 shows the part of assembly codes generated by LLVM after compilation of FreeRTOS source code completed. The total number of lines of assembly codes generated are 10070 lines, meaning that they will use 10070 words of memory space in I-cache which is the segment for user program code. From Figure 2.1.3, the memory space for storing the instructions in I-cache is starting from the address of 0x8000_0000 to 0x8001_b400, which

means that I-cache can support up to 111617 words. Therefore, the assembly codes of FreeRTOS can be fit into I-cache as the memory space it used does not exceed the memory space if I-cache. Caching for the instruction cache memory will not occur when the program is running.

5.3 FreeRTOS Simulation on RISC32

5.3.1 RISC32 Testbench

Testbench is a Verilog module which is used for simulation purpose. A testbench of RISC32 named tb_r32_pipeline.sv is created to test the functional behaviour of RISC32. There is a program.txt file storing the object code (hex code) generated and the content of program.txt will be loaded into instruction memory when the process of simulation starts. While the exception handler object code is stored in exc_handler.txt which its content will be loaded into an address starting from 0x8001_b400. The section below will show the testbench codes of RISC32.

```
`timescale 1ns / 1ps
`default_nettype none
`define demo003_UART 1

module tb_r32_pipeline();
//declaration
//===== INPUT =====
//System signal
reg     tb_u_clk;
reg     tb_u_rst;
//~~~~~~~~~~~~~~~~~~~
wire    tb_u_spi_mosi_dut;
wire    tb_u_spi_miso_dut;
wire    tb_u_spi_sclk_dut;
wire    tb_u_spi_ss_n_dut;

wire    tb_u_fc_sclk_dut;
wire    tb_u_fc_ss_dut;
wire    tb_u_fc_MOSI_dut;
wire    tb_u_fc_MISO1_dut;
wire    tb_u_fc_MISO2_dut;
wire    tb_u_fc_MISO3_dut;

wire    tb_ua_tx_rx_dut;
wire    tb_ua_RTS_dut, tb_ua_CTS_dut;
wire[31:0]  tb_u_GPIO_dut;

//~~~~~~~~~~~~~~~~~~~
wire    tb_u_spi_mosi_client;
wire    tb_u_spi_miso_client;
wire    tb_u_spi_sclk_client;
wire    tb_u_spi_ss_n_client;

wire    tb_u_fc_sclk_client;
wire    tb_u_fc_ss_client;
wire    tb_u_fc_MOSI_client;
wire    tb_u_fc_MISO1_client;
wire    tb_u_fc_MISO2_client;
wire    tb_u_fc_MISO3_client;

wire    tb_ua_tx_rx_client;
wire    tb_ua_RTS_client, tb_ua_CTS_client;
wire[31:0]  tb_u_GPIO_client;

crisc c_risc_dut(
//*********** INSTANTIATION *************
//======= INPUT =======
//GPIO
.urisc_GPIO(tb_u_GPIO_dut),

//SPI controller
.uiorisc_spi_mosi(tb_u_spi_mosi_dut),
.uiorisc_spi_miso(tb_u_spi_miso_dut),
.uiorisc_spi_sclk(tb_u_spi_sclk_dut),
.uiorisc_spi_ss_n(tb_u_spi_ss_n_dut),

//UART controller
```

```
.uorisc_ua_tx_data(tb_ua_tx_rx_dut),
//.uorisc_ua_rts(tb_ua_RTS_dut),
.uirisc_ua_rx_data(tb_ua_tx_rx_client),
//.uirisc_ua_cts(tb_ua_CTS_dut),

//FLASH controller
.uorisc_fc_sclk(tb_u_fc_sclk_dut),
.uiorisc_fc_MOSI(tb_u_fc_MOSI_dut),
.uirisc_fc_MISO1(tb_u_fc_MISO1_dut),
.uirisc_fc_MISO2(tb_u_fc_MISO2_dut),
.uirisc_fc_MISO3(tb_u_fc_MISO3_dut),
.uorisc_fc_ss(tb_u_fc_ss_dut),

// System signal
.uirisc_clk_100mhz(tb_u_clk),
.uirisc_rst(tb_u_rst));

//-----------------------------------------------------
s25fl128s
SPI_flash_dut(
.SI(tb_u_fc_MOSI_dut), //IO0
.SO(tb_u_fc_MISO1_dut), //IO1
.SCK(tb_u_fc_sclk_dut),
.CSNeg(tb_u_fc_ss_dut),
.RSTNeg(tb_u_rst),
.WPNeg(tb_u_fc_MISO2_dut), //IO2
.HOLDNeg(tb_u_fc_MISO3_dut));

//============================================================
==============================

crisc
c_risc_client(
//*********** INSTANTIATION *************
//======= INPUT =======
//GPIO
.urisc_GPIO(tb_u_GPIO_client),

//SPI controller
.uiorisc_spi_mosi(tb_u_spi_mosi_client),
.uiorisc_spi_miso(tb_u_spi_miso_client),
.uiorisc_spi_sclk(tb_u_spi_sclk_client),
.uiorisc_spi_ss_n(tb_u_spi_ss_n_client),

//UART controller
.uorisc_ua_tx_data(tb_ua_tx_rx_client),
//.uorisc_ua_rts(tb_ua_RTS_client),
.uirisc_ua_rx_data(tb_ua_tx_rx_dut),
//.uirisc_ua_cts(tb_ua_CTS_client),

//FLASH controller
.uorisc_fc_sclk(tb_u_fc_sclk_client),
.uiorisc_fc_MOSI(tb_u_fc_MOSI_client),
.uirisc_fc_MISO1(tb_u_fc_MISO1_client),
.uirisc_fc_MISO2(tb_u_fc_MISO2_client),
.uirisc_fc_MISO3(tb_u_fc_MISO3_client),
.uorisc_fc_ss(tb_u_fc_ss_client),
```

```
// System signal
.uirisc_clk_100mhz(tb_u_clk),
.uirisc_rst(tb_u_rst));

//-----------------------------------------------------
s25fl128s
SPI_flash_client(
.SI(tb_u_fc_MOSI_client), //IO0
.SO(tb_u_fc_MISO1_client), //IO1
.SCK(tb_u_fc_sclk_client),
.CSNeg(tb_u_fc_ss_client),
.RSTNeg(tb_u_rst),
.WPNeg(tb_u_fc_MISO2_client), //IO2
.HOLDNeg(tb_u_fc_MISO3_client));

assign tb_u_spi_mosi_dut = tb_u_spi_mosi_client;
assign tb_u_spi_miso_dut = tb_u_spi_miso_client;
assign tb_u_spi_ss_n_dut = tb_u_spi_ss_n_client;
assign tb_u_spi_sclk_dut = tb_u_spi_sclk_client;

assign tb_ua_CTS_dut = tb_ua_RTS_client;
assign tb_ua_CTS_client = tb_ua_RTS_dut;

//*********************Clock***********************
initial tb_u_clk = 1'b1;
always #5 tb_u_clk =~ tb_u_clk;

//For Vivado: remember to add the text files into the simulation source
//add Source -> Simulation sources -> add Files -> select the files (find "All file" in file type)
initial begin
//For client: copy the right test program and exc handler into FPGA flash.
$readmemh(`EXC_HANDLER_CLIENT, tb_r32_pipeline.SPI_flash_client.Mem);
$readmemh(`TEST_CODE_PATH_CLIENT, tb_r32_pipeline.SPI_flash_client.Mem);

//For dut: copy the right test program and exc handler into FPGA flash.
$readmemh("exc_handler.txt", tb_r32_pipeline.SPI_flash_dut.Mem);
$readmemh("program.txt", tb_r32_pipeline.SPI_flash_dut.Mem);
//test instruction 1st


//2nd test IO seperately
//SPI
//UART
//GPIO

//3rd exception handler

tb_u_rst = 1'b1;
repeat(1)@(posedge tb_u_clk);
tb_u_rst = 1'b0;
repeat(30000)@(posedge tb_u_clk);
tb_u_rst = 1'b1;
repeat(12000000)@(posedge tb_r32_pipeline.c_risc_dut.urisc_clk);
end


endmodule
```

5.3.2 FreeRTOS Assembly Code Debugging

However, the actual result is unexpected because the simulation goes into an infinite loop at the address of 0x8000_15f8. In order to trace the program flow, UART is used as a communication mean by sending out the values assigned. There is a function called main_blinky() containing the source code of implementing UART. Figure 5.3.1 shows the steps that is leading the simulation to infinite loop behaviour in xQueueGenericCreate() data structure. The highlighted label is where the simulation goes into an infinite loop as the jump instruction is jump back to its address .

Program starts from main.c

```
80000d04 <main>:
80000d04:       27bdffe0        addiu   sp,sp,-32
80000d08:       afbf001c        sw      ra,28(sp)
80000d0c:       afbe0018        sw      s8,24(sp)
80000d10:       03a0f025        move    s8,sp
80000d14:       24020000        li      v0,0
80000d18:       a7c00014        sh      zero,20(s8)
80000d1c:       afc20010        sw      v0,16(s8)
80000d20:       0c000350        jal     80000d40 <prvSetupHardware>
80000d24:       0c000357        jal     80000d5c <main_blinky>
80000d28:       8fc20010        lw      v0,16(s8)
80000d2c:       03c0e825        move    sp,s8
80000d30:       8fbe0018        lw      s8,24(sp)
80000d34:       8fbf001c        lw      ra,28(sp)
80000d38:       27bd0020        addiu   sp,sp,32
80000d3c:       03e00008        jr      ra
```

function call

```
80000d5c <main_blinky>:
80000d5c:       27bdffd0        addiu   sp,sp,-48
80000d60:       afbf002c        sw      ra,44(sp)
80000d64:       afbe0028        sw      s8,40(sp)
80000d68:       03a0f025        move    s8,sp
80000d6c:       24040001        li      a0,1
80000d70:       24050004        li      a1,4
80000d74:       24060000        li      a2,0
80000d78:       0c0004d0        jal     80001340 <xQueueGenericCreate>
```

Jump to the label's data structure

```
80001340 <xQueueGenericCreate>:
80001340:       27bdff48        addiu   sp,sp,-184
80001344:       afbf00b4        sw      ra,180(sp)
80001348:       afbe00b0        sw      s8,176(sp)
8000134c:       03a0f025        move    s8,sp
80001350:       00c00825        move    at,a2
80001354:       00a01025        move    v0,a1
80001358:       00801825        move    v1,a0
8000135c:       afc400ac        sw      a0,172(s8)
80001360:       afc500a8        sw      a1,168(s8)
80001364:       a3c600a4        sb      a2,164(s8)
80001368:       8fc400ac        lw      a0,172(s8)
8000136c:       afc10094        sw      at,148(s8)
80001370:       afc30090        sw      v1,144(s8)
80001374:       afc2008c        sw      v0,140(s8)
                        ...
                        ...
                        ...
80001388:       080004e3        j       8000138c <xQueueGenericCreate+0x4c>
8000138c:       080004e3        j       8000138c <xQueueGenericCreate+0x4c>
```

Figure 5.3.1 The flow of the program leading to infinite looping

Figure 5.3.2 shows the result of FreeRTOS simulation on RISC32. The task of UART is not created as the program goes into infinite looping during the process of creating a queue.

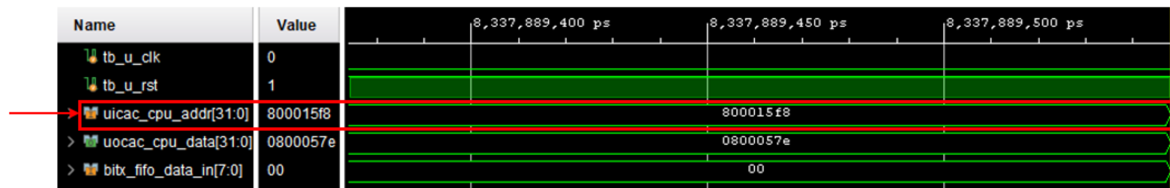


Figure 5.3.2 The result of FreeRTOS simulation on RISC32

Figure 5.3.1 and Figure 5.3.2 have proved that the program goes into infinite looping when the program executes the data structure of C program. There are 2 typed memory allocation instructions helping the LLVM compiler in memory allocation pattern which are malloc and alloca. The malloc instruction is used to dynamically allocate memory space on the heap, then a typed pointer pointing to new memory is returned. When more heap memory are allocated, the heap segment goes up. While the alloca instruction is having the similar concept of the malloc but the alloca instruction allocates the memory on the current function's stack frame[13].

In LLVM IR, the functions are defined with their name, arguments, and return type[14]. To call the function, an instruction named "call" is used associated with the function type, function name, and the name and the type of arguments. The "call" instruction will take the pointer to the function and the arguments as well. Thus, when a function is called, some stack memory will be allocated for to that function. If there are local variables declaring inside the function, more stack memory will be allocated for it. Therefore, the stack will go down according to its allocation pattern[15].

After analysing how a data structure and a function call will be executed in LLVM and stored in memory, the cause of the problem of infinite looping behaviour occurring during data structure is the LLVM is compiling the data structure onto stack segment. When the data structure is used, it should be heap access instead of stack access. Therefore, the LLVM has generated a function call type of instruction for the data structure which is totally wrong, leading it storing at stack segment.

CHAPTER 6: CONCLUSION

6.1 Conclusion

In a nutshell, the first 3 objectives are met in this project while the last objective which is the simulation of RTOS on RISC32 is done partially. The open source RTOS architecture and components are analysed so that the modification process can be proceed in order to make the RTOS suitable for RISC32 since the default version of FreeRTOS source codes are built for RISC-V examples. Second objective is met by modifying and developing RTOS for RISC32 and the RTOS source code is compilable by LLVM compiler. In order to compile the RTOS code, LLVM has to be setup. Thus, third objective is met because the LLVM compiler is setup successfully and a testing file is created to make sure the LLVM compiler is functioning well. LLVM compiler has converted the C language source file into assembly code as well as the object code. Besides, FreeRTOS source files is compilable by the LLVM after carrying out numerous debugging process. Until this stage, the RTOS codes can be said that it is mature enough to be port over RISC32. Simulating the RTOS in RISC32 is the last objective and it is done partially as there are problems coming out in the assembly codes as well as the hex codes generated by the LLVM compiler. Because of the duplication of assembly codes and the hex codes generated, the program runs inside a loop infinitely. There are jump instructions which is jumping back to its address are generated twice. Thus, LLVM compiler might not get the coding style of FreeRTOS. Simulation of RTOS failed on RISC32.

6.2 Future Work

For future work, a heap segment needs to be created in physical memory for FreeRTOS in order to prevent mis-generated instructions from occuring in LLVM. Some functions that may not be supported by LLVM as well as RISC32, modification of FreeRTOS codes is required until FreeRTOS can be simulated on RISC32. To prove the assumption, compiling FreeRTOS codes by using GCC compiler is a required action. In addition, comparing the codes of FreeRTOS of RISC32 and the original version of FreeRTOS of RISC-V in order to make sure there is nothing missing out. Besides, heap segment to be used by RTOS need to be configured to avoid overlapping or misusing memory problems occur. Therefore, checking the functionality of LLVM, RISC32, and FreeRTOS are considered as huge processes which can be taken as another new project. Modification of codes will be done and simulating RTOS again on RISC32.

Bibliography

[1] C. Architecture and S. Engineering, "Process Control Block (PCB)," pp. 1–9, 2019.

[2] E. Peña and M. G. Legaspi, "UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter," *Visit Analog.*, vol. 54, no. 4, 2020.

[3] W. P. Kiat, "THE DESIGN OF AN FPGA-BASED PROCESSOR WITH RECONFIGURABLE PROCESSOR EXECUTION STRUCTURE FOR INTERNET OF THINGS (IoT) APPLICATIONS KIAT WEI PAU MASTER OF SCIENCE (COMPUTER SCIENCE) FACULTYOF INFORMATION AND COMMUNICATION TECHNOLOGY UNIVERSITI TUNKU ABD," no. December, 2018.

[4] J. C. See, "TOOLCHAIN DEVELOPMENT AND QUEUE SYSTEM ENHANCED SECURITY COPROCESSOR FOR FPGA- BASED INTERNET OF THINGS (IOT) PROCESSOR SEE JIN CHUAN MASTER OF SCIENCE (COMPUTER SCIENCE) FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY UNIVERSITI TUNKU ABDUL RAHMA," no. August, 2019.

[5] V. Bistriceanu, "Exception handling registers in coprocessor 0," 1996.

[6] M. H. Qutqut, A. Al-Sakran, F. Almasalha, and H. S. Hassanein, "Comprehensive survey of the IoT opensource OSs," *IET Wirel. Sens. Syst.*, vol. 8, no. 6, pp. 323–339, 2018, doi: 10.1049/iet-wss.2018.5033.

[7] O. File and T. Cases, "State Diagram for a Task," pp. 21–22.

[8] E. Now, "What is Context Switching in Operating System?," *AfterAcademy*, pp. 1–12, 2019, [Online]. Available: https://afteracademy.com/blog/what-is-context-switching-in-operating-system.

[9] A. Tech, "GCC vs. Clang/LLVM: An In-Depth Comparison of C/C++ Compilers," pp. 1–12, 2019.

[10] "Oracle Homeage test Documentation Home > Programming Interfaces Guide > Chapter 1 Memory Management > Library-Level Dynamic Memory > Other Memory Control Interfaces > brk and sbrk Programming Interfaces Guide," p. 19683.

[11] "Computer Organization and Design," *Computer Organization and Design*. 1994, doi: 10.1016/c2013-0-08305-3.

[12] "Mastering the FreeRTOS ™ Real Time Kernel."

[13] C. Lattner and V. Adve, "The LLVM Instruction set and compilation strategy," *CS Dept., Univ. Illinois Urbana-Champaign, …*, pp. 1–20, 2002, [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.58.40&rep=rep1&type=pdf.

[14] A. M. Si and C. L. Ir, "/SI413/lab/l13/Compilingto LLVM IR," pp. 1–8.

[15] J. Chen and R. Guo, "Stack and Heap Allocation," pp. 3–4.

Appendix B – RISC 32 Coprocessor 0 Register

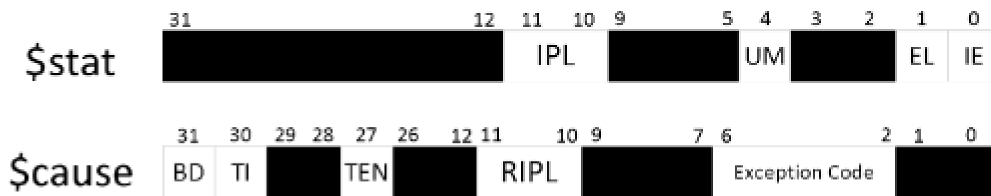


Figure 3.95: Graphical view of CP0 $stat and $cause registers

Register	bit	usage
$stat	[31:12]	RESERVED
	IPL[11:10]	store current interrupt priority level
	[9:5]	RESERVED
	UM[4]	1=user mode, 0=kernel mode
	[3:2]	RESERVED
	EL[1]	Exception level 1=exception occurs, disable further exception to occur 0=no exception occurs
	IE[0]	1=Interrupt enable 0=Interrupt disable
$cause	BD[31]	Indicate branch delay
	TI[30]	1=enable timer interrupt 0=disable timer interrupt
	[29:28]	RESERVED
	TEN[27]	CP0 Timer, $count disable control
	[26:12]	RESERVED
	RIPL[11:10]	Request interrupt priority level
	[9:7]	RESERVED
	Exception code [6:2]	encodes reasons for the exception 0=Interrupt 4=AdEL, address error trap (load or instruction fetch) 5= AdES, address error trap (store) 6=IBE, bus error on instruction fetch trap 7=DBE, bus error on data load or store trap 8=Sys, syscall trap 9=Bp, breakpoint trap 10=RI, undefined instruction trap 12=Ov, arithmetic overflow trap
	[1:0]	RESERVED

Biweekly Report

FINAL YEAR PROJECT WEEKLY REPORT

(Project II)

Trimester, Year: Jan, 2022	**Study week no.: 1**
Student Name & ID: Er Pei Qing	
Supervisor: Ts Dr. Chang Jing Jing	
Project Title: The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor	

1. WORK DONE

- Analysis FreeRTOS source codes in order to do modification.
- Install Ubuntu 16.04 for compilation purpose.
- Setup LLVM compiler as toolchain for the project
- Setup RISC32 on Vivado
- Run the demo codes available on the folder provided by Mr.Mok

2. WORK TO BE DONE

- Analysis LLVM compiler.
- Compiling FreeRTOS source codes and debugging.
- Writing a testing code to test LLVM functionality.

3. PROBLEMS ENCOUNTERED

- The demo projects provided by FreeRTOS is not working due to lack of the demo components.
- Need to learn how to trace the data coming out from RISC32 on Xilinx Vivado

4. SELF EVALUATION OF THE PROGRESS

- 70% in progress as FreeRTOS is a huge document and trying to write a testing code.

Supervisor's signature

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project II)

Trimester, Year: Jan, 2022	**Study week no.: 3**
Student Name & ID: Er Pei Qing	
Supervisor: Ts Dr. Chang Jing Jing	
Project Title: The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor	

1. WORK DONE
• Analysis of FreeRTOS architecture. • Write a testing code to check the LLVM functionality. • After writing the testing code, write a bash script to do compilation. • Analysis LLVM to get the compilation step.
2. WORK TO BE DONE
• Compiling the testing code and debugging. • Compiling the testing code. • Modification on FreeRTOS to suit RISC32.
3. PROBLEMS ENCOUNTERED
• No sure how to use UART to send data out. • There are many commands needed to compile the source code but not sure which one is the correct version to compile the testing code written.
4. SELF EVALUATION OF THE PROGRESS
• 90% as the codes had been complete written.

Supervisor's signature

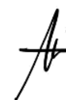
Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project II)

Trimester, Year: Jan, 2022	**Study week no.: 5**
Student Name & ID: Er Pei Qing	
Supervisor: Ts Dr. Chang Jing Jing	
Project Title: The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor	

1. WORK DONE
• The testing code has been compiled and there are several files obtained. • Analysis the assembly codes of the testing code generated by LLVM. • Move the hex code generated to Xilinx Vivado for simulation. • The testing code has been compiled and there are several files obtained. • Analysis the assembly codes of the testing code generated by LLVM. • Move the hex code generated to Xilinx Vivado for simulation. • Observe the simulation result of UART.
2. WORK TO BE DONE
• Compiling the testing code. • Modification on FreeRTOS to suit RISC32. • Compiling FreeRTOS which has been modified for RISC32. • Debugging if errors come out.
3. PROBLEMS ENCOUNTERED
• There are many commands needed to compile the source code but not sure which one is the correct version to compile the testing code written.
4. SELF EVALUATION OF THE PROGRESS
• 90% as the codes had been complete written.



Supervisor's signature

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project II)

Trimester, Year: Jan, 2022	**Study week no.: 7**
Student Name & ID: Er Pei Qing	
Supervisor: Ts Dr. Chang Jing Jing	
Project Title: The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor	

1. WORK DONE
• FreeRTOS has been compiled. • Debugging for the undefined errors. • Replace the original inline assembly code to RISC supported version. • Searching for the codes that are written for RISC-V for replacement purpose.
2. WORK TO BE DONE
• Search for the information of RISC-V and do comparison of RISC-V and RISC32 for modification purpose. • Study those codes usage to make the modification easy.
3. PROBLEMS ENCOUNTERED
• Lack of knowledge about the inline assembly used in C program. • The instruction set architectures are different in RISC-V and RISC32, making modification hard.
4. SELF EVALUATION OF THE PROGRESS
• 60% as insufficient knowledge to modify the codes which is good enough for RISC32 and modification still in progess.

Supervisor's signature

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project II)

Trimester, Year: Jan, 2022	**Study week no.: 9**
Student Name & ID: Er Pei Qing	
Supervisor: Ts Dr. Chang Jing Jing	
Project Title: The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor	

1. WORK DONE
• Create the replaced functions for RISC32 as it does not support string.h files. • Inline assembly code modification completed. • Compiled the codes. • Debugging as there are still some coding which is undefined for RISC32.
2. WORK TO BE DONE
• Compiling the codes again. • Debugging and modifying the codes to suit RISC32.
3. PROBLEMS ENCOUNTERED
• To do function replacement part, knowledge about string.h internal function definition is required. • Lack of information about the instruction supported by RISC32. • The registers used in RISC-V are different for the registers used in RISC32. • RISC-V uses "x" to represent the register while RISC32 uses "$" to represent the register.
4. SELF EVALUATION OF THE PROGRESS
• 80% as modification almost done.

Supervisor's signature

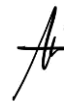
Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project II)

Trimester, Year: Jan, 2022	**Study week no.: 11**
Student Name & ID: Er Pei Qing	
Supervisor: Ts Dr. Chang Jing Jing	
Project Title: The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor	

1. WORK DONE
• The porting codes have been modified. • Compiling the codes and debugging. • Modified the ISR and the Interrupt Enable bit for RISC32 in FreeRTOS source code. • Compiled the codes to get the hex code in order to do simulation on Xilinx Vivado. • Simulate the codes on RISC32 in Xilinx Vivado.
2. WORK TO BE DONE
• Analysis the interrupt service routine (ISR) used in RISC32 and modified the ISR in FreeRTOS source codes. • Start writing the report.
3. PROBLEMS ENCOUNTERED
• Not enough knowledge about ISR of RISC32.
4. SELF EVALUATION OF THE PROGRESS
• 100% as the source codes are compiled successfully.



Supervisor's signature

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project II)

Trimester, Year: Jan, 2022	**Study week no.: 13**
Student Name & ID: Er Pei Qing	
Supervisor: Ts Dr. Chang Jing Jing	
Project Title: The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor	

1. WORK DONE
• Debugging the problems coming out from the simulation result. • Writing the report.
2. WORK TO BE DONE
• Writing the report
3. PROBLEMS ENCOUNTERED
• The program goes to infinite looping due to memory and the compiler issues.
4. SELF EVALUATION OF THE PROGRESS
• 75% as solving the memory issue and the LLVM compiler problems are huge project.

Supervisor's signature

Student's signature

Poster

UNIVERSITI TUNKU ABDUL RAHMAN

Faculty of Information and Communication Technology (FICT)

The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor

Introduction:

- RTOS - operating system intended to serve real time applications, rapidly switches between tasks, make the user have an impression that multiple programs are executing simultaneously on a single processor.
- FreeRTOS – an open-source code real-time operating system for microcontroller and microprocessor.
- Compiling code using llvm compiler.

Method:

- Refer to FreeRTOS source code.
- Compare the code used by RISC-V demo projects.
- Create a file to be used by RISC32.

Discussion:

- Why do we need RTOS for RISC32?
- How does RTOS interact with the hardware component?

Conclusion:

- RTOS code can be implemented in RISC32.
- RISC performance will be improved.

Plagiarism Check Result

Document Viewer

Turnitin Originality Report

Processed on: 22-Apr-2022 04:41 +08
ID: 1815750828
Word Count: 8450
Submitted: 3

RTOS of RISC32 By Pei Qing Er

Similarity Index	Similarity by Source	
1%	Internet Sources:	N/A
	Publications:	1%
	Student Papers:	N/A

include quoted include bibliography exclude small matches mode: quickview (classic) report Change mode print download

<1% match (publications)
Jin-Chuan See, Kai-Ming Mok, Wai-Kong Lee, Hock-Guan Goh. "RISC32-E: Field programmable gate array based sensor node with queue system to support fast encryption in Industrial Internet of Things applications", International Journal of Circuit Theory and Applications, 2020

<1% match (publications)
Ruohuang Zheng, Michael C. Huang. "Redundant Memory Array Architecture for Efficient Selective Protection", Proceedings of the 44th Annual International Symposium on Computer Architecture, 2017

<1% match (publications)
Panagiotis Manolios. "<![CDATA[A Refinement-Based Compositional Reasoning Framework for Pipelined Machine Verification]]>", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 4/2008

<1% match (publications)
Fulvio Corno, Luigi De Russis, Juan Pablo Saenz. "How is Open Source Software Development Different in Popular IoT Projects?", IEEE Access, 2020

<1% match (publications)
I-Chun Liu, I-Wei Wu, Jean Jyh-Jiun Shann. "Instruction Emulation and OS Supports of a Hybrid Binary Translator for x86 Instruction Set Architecture", 2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), 2015

<1% match (publications)
"Tools and Algorithms for the Construction and Analysis of Systems", Springer Science and Business Media LLC, 2015

<1% match (publications)
Ofelia Rodríguez Alzueta. "2'-Methoxyacethopenone as DNA Photosensitiser for Mono and Biphotonic Processes", Universitat Politecnica de Valencia, 2020

CHAPTER 1: Introduction 1.1 Background Information A computer consists of several components. Among the components, the main components are the processor and I/O devices. A processor performance's maybe robust, but without an interactive interface with the user, the processor might not be fully utilized. The I/O devices act as the interactive interface between the user and the processor. The interconnect between I/O devices and the processor would be the bus system. 1.1.1 MIPS MIPS is known as Microprocessor without Interlocked Pipeline Stage, which is based on RISC architecture developed by MIPS technologies, previously known as MIPS Computer Systems. According to Neha T (2019), RISC processor supports simple instruction set compared to CISC. RISC architecture emphasizes on using register rather than memory. Instead of using Intel 80x86, MIPS is used because it has a simple design and yet high performance as embedded processor. It also has large market for embedded app. (Junko 2010) After years of development, MIPS architecture nowadays can support 64-bit addressing and operation and high- performance floating point making it popular in the embedded systems implementation such as router, game machine and so on. The instruction execution is broken by the operation of MIPS processor into multiple small independent stages (Integrated Device Technology. Inc, 1994, pg1-2). The word "stages" implies the datapath resources at each stage. Figure 1.1.1 MIPS 5-stage pipeline From Figure 1.1, the execution of an instruction is done in 5 basic stages including: ? IF: Instruction fetch and update PC ? ID: Instruction decode and register fetch ? EX: Execute R-type, calculate memory address ? MEM: Read data from memory or write data to memory ? WB: Write the result data into register file 1.1.2 UART UART stands for Universal Asynchronous Receiver/Transmitter, it is used for asynchronous serial communication of data over peripheral device serial port. Most embedded systems use UART for data communication as it is a hardware communication protocol that only uses 2 wires for transmitting end (TX) and receiving end (RX). Figure 1.1.2 shows that there are 2 UART communicating with each other. Figure 1.1.2 Two UART communicating with each other[1] 1.1.3 RTOS Real Time Operating System is an operating system intended to serve real time applications. It is a software component that rapidly switches between tasks, make the user have an impression that multiple programs are executing simultaneously on a single processor. Operating system consists of many different parts such as file system, I/O, memory allocation, network, and scheduler. RTOS provides a hard real time response and a highly deterministic reaction to external event. Hard real time is a system that must always meet all deadlines or the system will fail if the deadline is missed. RTOS can be time-sharing or event-driven. Time-sharing system switch the task based on the timer interrupt while event-driven system switches the task according to the task priority. The value of a real-time operating system depends on how fast it can respond compared to the amount of work it can perform in given period of time. Most RTOS is using a pre-emptive algorithm. A basic RTOS has 3 states which the task might be assigned. Figure 1.1.3 RTOS Task

Form Title: Supervisor's Comments on Originality Report Generated by Turnitin for Submission of Final Year Project Report (for Undergraduate Programmes)			
Form Number: FM-IAD-005	Rev No.: 0	Effective Date: 01/10/2013	Page No.: 1of 1

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Full Name(s) of Candidate(s)	Er Pei Qing
ID Number(s)	18ACB04358
Programme / Course	CT
Title of Final Year Project	The Development of an RTOS for the 5-Stage Pipeline RISC32 Microprocessor

Similarity	**Supervisor's Comments** **(Compulsory if parameters of originality exceed the limits approved by UTAR)**
Overall similarity index:__1____ % **Similarity by source** Internet Sources: __0__ % Publications: __1__ % Student Papers: __0__ %	
Number of individual sources listed of more than 3% similarity: _0____	
Parameters of originality required, and limits approved by UTAR are as Follows: **(i) Overall similarity index is 20% and below, and** **(ii) Matching of individual sources listed must be less than 3% each, and** **(iii) Matching texts in continuous block must not exceed 8 words** ***Note: Parameters (i) – (ii) shall exclude quotes, bibliography and text matches which are less than 8 words.***	

Based on the above results, I hereby declare that I am satisfied with the originality of the Final Year Project Report submitted by my student(s) as named above.

____

Signature of Supervisor

Name: Ts Dr Chang Jing Jing

Date: 22/4/2022

____

Signature of Co-Supervisor

Name: ________________________

Date: ________________________

FYP2 Checklist

UNIVERSITI TUNKU ABDUL RAHMAN

FACULTY OF INFORMATION & COMMUNICATION TECHNOLOGY
(KAMPAR CAMPUS)

CHECKLIST FOR FYP2 THESIS SUBMISSION

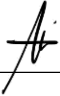
Student Id	18ACB04358
Student Name	Er Pei Qing
Supervisor Name	Ts Dr. Chang Jing Jing

TICK (√)	DOCUMENT ITEMS Your report must include all the items below. Put a tick on the left column after you have checked your report with respect to the corresponding item.
	Front Plastic Cover (for hardcopy)
√	Title Page
√	Signed Report Status Declaration Form
√	Signed FYP Thesis Submission Form
√	Signed form of the Declaration of Originality
√	Acknowledgement
√	Abstract
√	Table of Contents
√	List of Figures (if applicable)
√	List of Tables (if applicable)
	List of Symbols (if applicable)
√	List of Abbreviations (if applicable)
√	Chapters / Content
√	Bibliography (or References)
√	All references in bibliography are cited in the thesis, especially in the chapter of literature review
√	Appendices (if applicable)

√	Weekly Log
√	Poster
√	Signed Turnitin Report (Plagiarism Check Result - Form Number: FM-IAD-005)
√	I agree 5 marks will be deducted due to incorrect format, declare wrongly the ticked of these items, and/or any dispute happening for these items in this report.

*Include this form (checklist) in the thesis (Bind together as the last page)

I, the author, have checked and confirmed all the items listed in the table are included in my report.



(Signature of Student)

Date: 15/4/2022