A Lightweight Process Model for FreeRTOS

Session: 2013 - 2015

Submitted by:

Mughees Ahmed Chohan 2013-MS-CE-10

Supervised by: Dr. Muhammad Faisal Hayat

Department of Computer Science and Engineering

University of Engineering and Technology Lahore

A Lightweight Process Model for FreeRTOS

Submitted to the faculty of the Computer Science and Engineering Department
of the University of Engineering and Technology Lahore
in partial fulfillment of the requirements for the Degree of

Master of Science

in

Computer Engineering.

Internal Examiner	External Examiner
Dean Faculty of Computer Science and Engineering	Chairman Computer Science and Engineering Department

Department of Computer Science and Engineering

University of Engineering and Technology Lahore

Declaration

I declare that the work contained in this thesis is my own, except where explicitly stated otherwise. In addition this work has not been submitted to obtain another degree or professional qualification.

Signed: ____________________

Date: ____________________

Acknowledgments

I want to thank Almighty God for all his uncountable blessings. I am extremely grateful to my supervisor, Dr. Muhammad Faisal Hayat, for his invaluable support and guidance throughout this project.

This thesis is dedicated to my parents.

Contents

List of Figures

Abbreviations

ELF	**E**xecuteable and **L**inkable **F**ormat
GOT	**G**lobal **O**ffset **T**able
PLT	**P**rocedure **L**ink **T**able
PIC	**P**osition **I**ndependent **C**ode
RAM	**R**andom **A**ccess **M**emory
ROM	**R**ead **O**nly **M**emory
RTOS	**R**eal **T**ime **O**perating **S**ystem
SMP	**S**ymmetric **M**ulti-processing
MPU	**M**emory **P**rotection **U**nit
MMU	**M**emory **M**anagement **U**nit
SPI	**S**erial **P**eripheral **I**nterface

Abstract

FreeRTOS is a popular Royalty Free and Open Source Real Time Operating System (RTOS) for use in Embedded Systems. Its low footprint is one of the key features that makes it the ultimate choice for memory constrained Embedded Systems. The aim of this research is to develop a processes framework for FreeRTOS to provide support for dynamic loading and unloading of processes at run-time. Some added benefits this process support will provide are Efficient Utilization of Memory resources, Memory Protection, Kernel Reliability and In-field updating of software. Special care will be taken to keep the footprint of the process framework support as low as possible while still providing basic functionalities of the processes.

Chapter 1

Introduction

Embedded Systems are finding their way into more and more applications day by day. Especially with the Internet of Things (IoT) and Wearable concept in place, the number of embedded devices is expected to grow much more rapidly. Their applications are simply uncountable, extending from simple calculators to life critical systems such as flight control of a commercial airliner. With such penetration of embedded systems in daily lives, more elaborate design techniques are needed to make embedded systems more reliable, efficient, and cost-effective. In memory constrained applications, using operating systems such as Linux is totally out of question primarily because of their memory footprint. This is where Real Time Operating Systems come into action. In our research, we highlight a few areas where we can improve an existing embedded software solution, the FreeRTOS, by implementing a processes framework similar to that of Linux but with many variations in order to keep it lightweight but yet a beneficial one.

1.1 Real Time Operating Systems (RTOS)

Operating Systems that implement real time requirements are called Real Time Operating Systems. Their schedulers use scheduling algorithms that guarantee that real time tasks with highest priorities are always scheduled with most importance and meets that their timing requirements are met. Real Time Operating Systems are employed in many embedded systems to meet real time requirements of an application at hand. Usually Real Time Operating Systems are kept small in footprint yet robust in performance to meet both the cost related and real time constraints of the application. But this might not always be the case. They might be used just for the Real Time Performance or just for their light weight footprint or both. They have been deployed on many different hardwares ranging from low-end micro-controllers to high-end uni-core and multi-core processors. Typical applications can include mp3 players, networking routers, sports watches, GPS navigation equipment, calculators, cell phones, automobile, aviation and many more. Examples of popular Real Time Operating Systems are Mentor's Nucleus RTOS [5], Express Logic's ThreadX [6], Windriver's VxWorks [7], BlackBerry's QNX [8] etc. Real Time version on Linux is also available to be used as an RTOS.

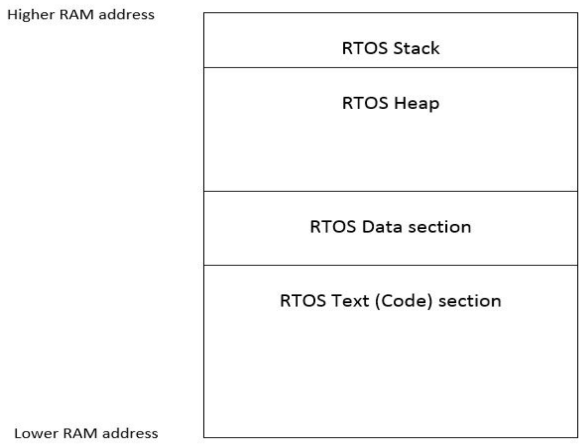


Figure 1.1: RTOS binary image view in RAM

1.2 Processor Memory view of an RTOS

The source code of Real Time Operating Systems is usually compiled as a monolithic binary image. This RTOS binary image has exactly one code section, one data section and an area for heap and stack. This complete image then can be burnt into the ROM of a micro-controller or maybe loaded into the RAM and executed from there. An RTOS usually schedules tasks also called threads. A thread or a task is simply a C function that usually never exits and performs some actions. The scheduler saves context of one task when its time-slice has expired or another high priority task needs attention and restores the context of other task and resumes it from the point where it had left. All the tasks have their code in the code section of the RTOS binary image and all the data of all the tasks is in the data section of the RTOS binary image. The tasks can use memory from the heap of this binary image as their stack. An hypothetical memory view of an RTOS is shown in Figure 1.1. The figure shows the different sections of the RTOS binary image being place at some location in the memory. Actual memory view of different hardware platforms may differ depending upon the memory layout.

1.3 Processes

A process is an executing program in memory [1]. Every process has its own stack, heap, data and text section in its address space. A process can spawn different threads all sharing same address space as the process. All threads spawned by a single process have their code and data sections inside the code and data sections of that process. A process view in memory is shown in Figure 1.2.

1.4 Processes in an RTOS

We can bring this concept of process to an RTOS too. A process can be simply a piece of code, data, stack and heap sections combined into small binary file. This binary file can be loaded into a memory location allocated from the RTOS heap section, given control to and when done can simply be unloaded from there. There is actually more

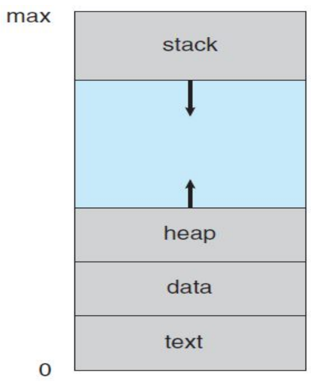


FIGURE 1.2: A Process in memory [1]

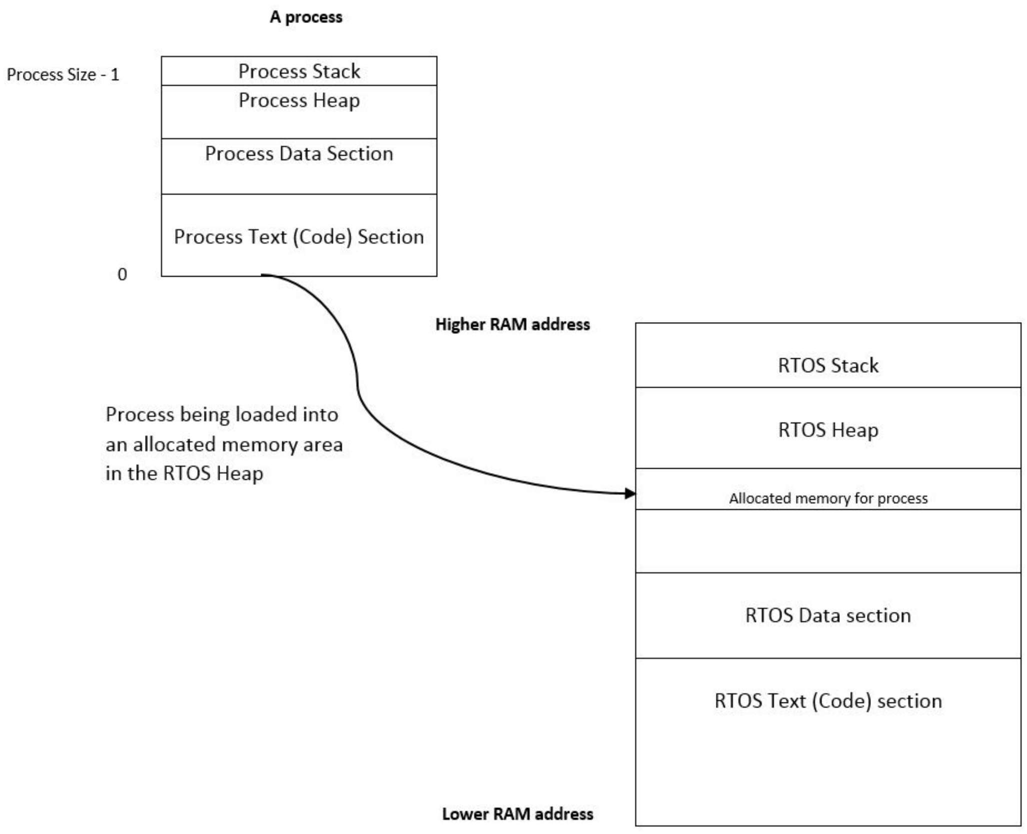


FIGURE 1.3: A Process in an RTOS

to the equation than to simply copy the process file into a memory location as we shall see in the coming chapters. Processes support is offered in some RTOS solutions such as Mentor's Nucleus RTOS [5] and eSOL's extended version of the T-Kernel [9] the eT-Kernel [10]. See Figure 1.3 for a general overview of this concept.

1.5 Benefits of Processes

In this section we will look at the benefits and the capabilities that can be brought to an RTOS solution by implementing a process framework. One very important concern would be the overheads this process framework will introduce in terms of code size and performance. We would like to keep the size of the code compact and the performance overheads at minimum. The most important features and capabilities processes will provide are as follows.

1.5.1 Dynamic Loading and Unloading

A process can be loaded and unloaded at run-time in an already running RTOS. By providing the capability of loading and unloading of process, we can free the memory resources occupied by one process by unloading it and then loading the other required process. The processes could initially reside in a cheaper storage such as SPI Flash from where it can be loaded into the RAM. One obvious drawback is that the unloaded process would not be able to do anything unless it is loaded into memory. But to conserve space, we can use this feature in applications which does not require all of the process to be in memory all the time.

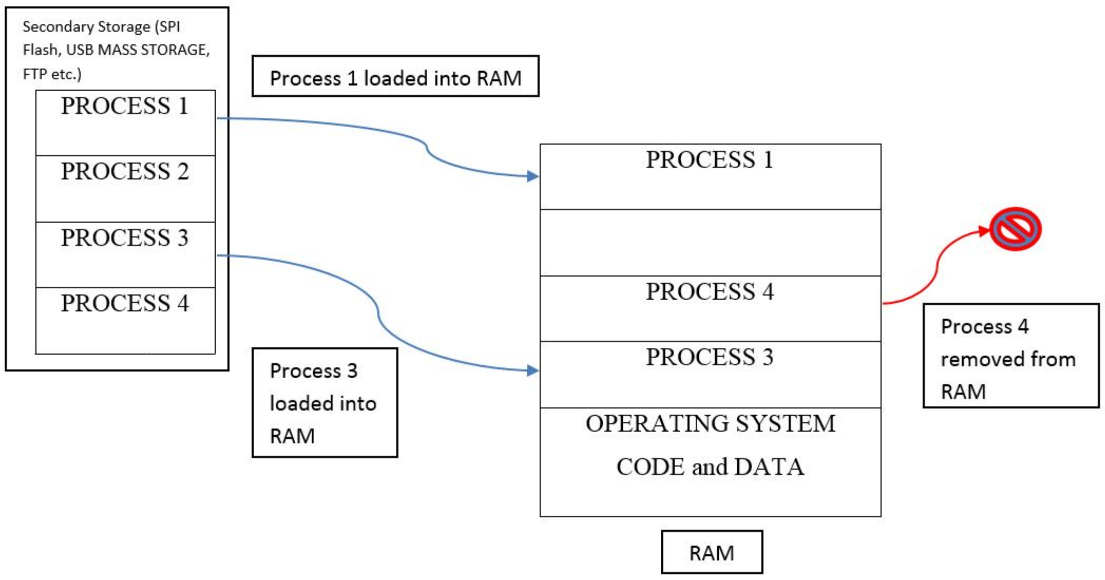

Secondary Storage (SPI Flash, USB MASS STORAGE, FTP etc.)
PROCESS 1
PROCESS 2
PROCESS 3
PROCESS 4
Process 1 loaded into RAM
Process 3 loaded into RAM
PROCESS 1
PROCESS 4
PROCESS 3
OPERATING SYSTEM CODE and DATA
RAM
Process 4 removed from RAM

FIGURE 1.4: Loading and Unloading of Processes

1.5.2 In-field Software Update

An embedded system that has been deployed in the field can use this process framework to update its software at run-time. The application can simply unload existing processes and load new ones containing updated functionalities. The new processes could be loaded from a secondary storage such as flash or even loaded over a network remotely.

1.5.3 Memory Protection

Memory protection can protect one process from accessing the memory of another process accidentally or deliberately. This increases the reliability and security in the system. Every processor provides some kind of Memory Protection Unit MPU for the Operating System to implement memory protection. The whole memory space can be divided into memory regions and these memory regions be assigned to individual processes. If one process tries to access the memory region of another process, the MPU throws some kind of fault to the Operating System. The Operating System can then take further actions such as terminating the process that made the illegal access.

1.5.4 Kernel and User Mode Processes

Processes can be of two types. Kernel Mode and User Mode processes. Kernel Mode processes have direct access to hardware and can execute privileged CPU instructions and can access any memory directly. An example of a kernel mode process could be a loadable device driver. User Mode processes will not have this functionality. The User mode processes can only call Operating System APIs or run general code. If a User Mode process needs to access a kernel resource or execute a kernel API it first needs to execute a special software interrupt instruction that is available in almost every processor architecture. Executing this instruction causes an exception to be generated and control is transferred to the operating system which switches the processor's mode of execution to privileged mode. The operating system then performs the required functionality, switches back to unprivileged mode and then returns to the user-mode process.

1.5.5 Improved Kernel Reliability

In an Operating System that supports processes, the kernel can run as a separate standalone entity. Failures and crashes in processes will be confined to themselves and will not affect the kernel in any way. This greatly increases kernel reliability, that is very important for the functioning of safety critical embedded system.

1.6 Important Concepts for Implementation of Processes

Now, we present some important concepts that would help us understand how processes can be implemented. A general overview of these concepts is presented here. The user is urged to consult [11], [12] and [13] for further understanding.

1.6.1 Linux Processes

The concept of processes is very generic. We can try to look at process model of Linux Operating System to have an idea of the capabilities processes can provide in an Operating Systems environment. Linux Processes provide a lot of features to its users. But for an embedded environment these features maybe unwanted or can be dropped to conserve resources such as memory or to increase performance.

1.6.2 Position Independent Code (PIC)

One of the benefits of introducing processes would be the ability to load and unload processes at run-time in RAM. The processes can be loaded at any available position in the RAM depending upon the location of free memory available and the size requirements of the process. In a typical RTOS environment the code image is compiled as a monolithic binary and is loaded at boot time into the RAM. This code has fixed addresses and code and data references can be made at absolute addresses. But what happens if we want to run a process from any location in the RAM? If the data and function references were done to fixed addresses in the process code, then this process would not be able to run from any location in RAM. The solution to this problem is using Position Independent Code. A position independent code is a kind of code that can run from any physical location in memory. It is not dependent on the absolute physical addresses. A position independent code relies on relative addressing and can also use other indirection mechanisms such as Procedure Linking Table (PLT) and Global Offset Table (GOT) to achieve position independence. For example if we want to branch to a particular function located at an address say X, instead of pushing the actual address of the function into a register and then branching to it, a position independent code will first put the offset of the function address from the current instruction and then branch to it (this technique is an example of Program Counter (PC) relative addressing). To make a piece of code position independent we need to be implement mechanisms for both the data accesses and function calls. These can be accomplished by using the concepts of Global Offset Table (GOT) and Procedure Link Table (PLT) respectively as we explain in the next sections. Though the concepts of GOT and PLT are used to serve much broader use cases, we explore how they can benefit us in running dynamically loadable processes. Both the GOT and the PLT can make linking possible at the time of loading the process. The compiler and linker can be explicitly told by the user to generate Position Independent Code.

1.6.2.1 Global Offset Table (GOT)

A non-position independent code when requires to access global data variables in memory, can access them by using their absolute addresses in the data section. If this code is made to run from a different location in memory then these absolute addresses would simply become invalid. Position Independent Code uses a table know as Global Offset Table (GOT) to achieve position independence for data accesses. The Global Offset Table is a table of absolute addresses. This table is part of the data section of a program image. This table would contain the absolute addresses of all the data variables a position independent program would need to access. Instead of accessing data variable absolute addresses directly, the position independent code would address the entries in this Global Offset Table. The relative address of this Global Offset Table is fixed in the program and is known to the linker and loader. In this way the data accesses in code could be fixed to relative addresses of the entries in the Global Offset Table. From there the code can extract the exact address of the data variable and then can access it. The absolute addresses of the data variables in the Global Offset Table can be calculated and populated during process loading by the loader. Every process will have its own Global Offset Table. See Figure 1.5. The process' code wants to get value of the Data variable 1. Instead of accessing it directly, the code first gets the address of the Data variable 1 from the Global Offset Table and then uses this address to access Data variable 1. The addresses in the GOT are populated at load time of the process.

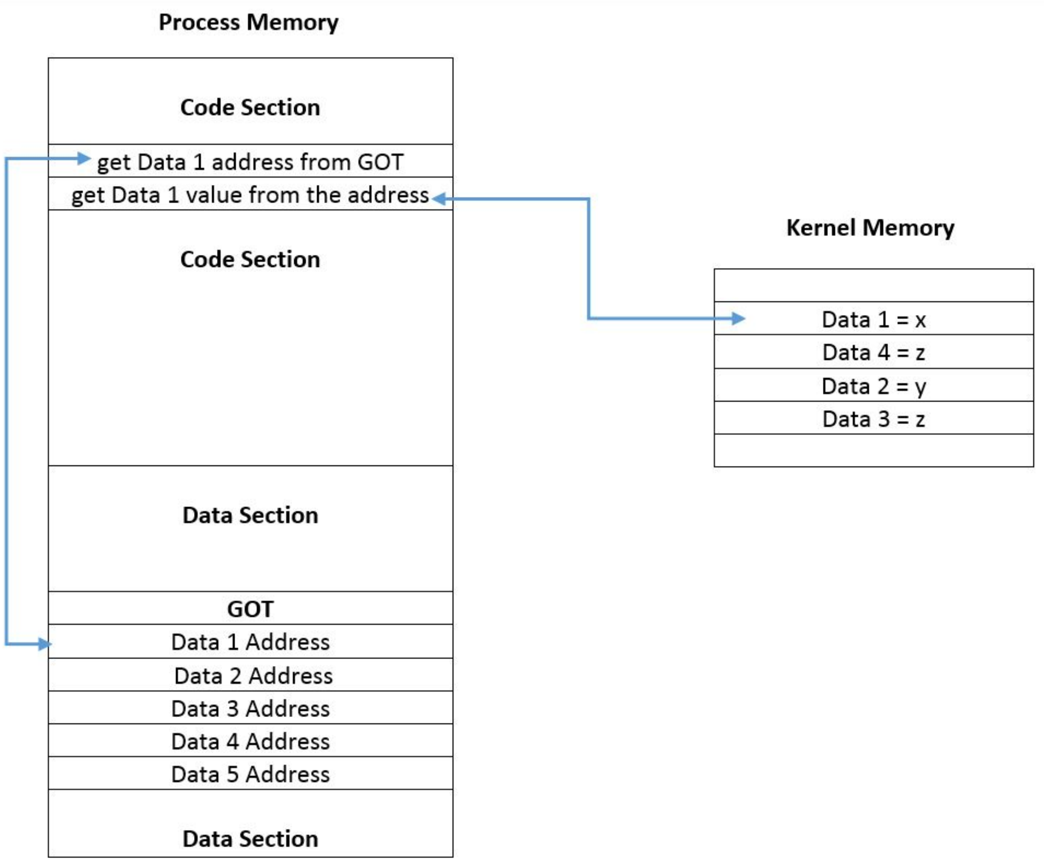


FIGURE 1.5: Global Offset Table (GOT)

1.6.2.2 Procedure Link Table (PLT)

A procedure link table is a table of entries and each entry contains code to fetch some function address from the Global Offset Table and call it. There is one entry for each different function referenced by the position independent code. Instead of calling functions directly, the position independent code initially jumps to the entry in Procedure Link Table for that function. The Procedure Link Table entry then calls the appropriate function by getting its address from the Global Offset Table. The function addresses in the Global Offset Table are populated at process load time by the loader. Every process will have its own Procedure Linking Table in its code section and a Global Offset Table in its data section for use specifically by its PLT. See Figure 1.6. The process' code wants to call Function 2. Instead of calling it directly, the code first jumps to an entry Function2@PLT in the PLT. This entry also contains some code. This code will first get the address of Function 2 from the Global Offset Table and then jump to that address. The addresses in the Global Offset Table can be populated at load time.

1.6.3 Linker Description File (LD File)

A linker description file dictates a linker how code, data and other sections should be generated and placed in memory. This is needed because every hardware platform is different. The memory map of every processor may not be the same. For example

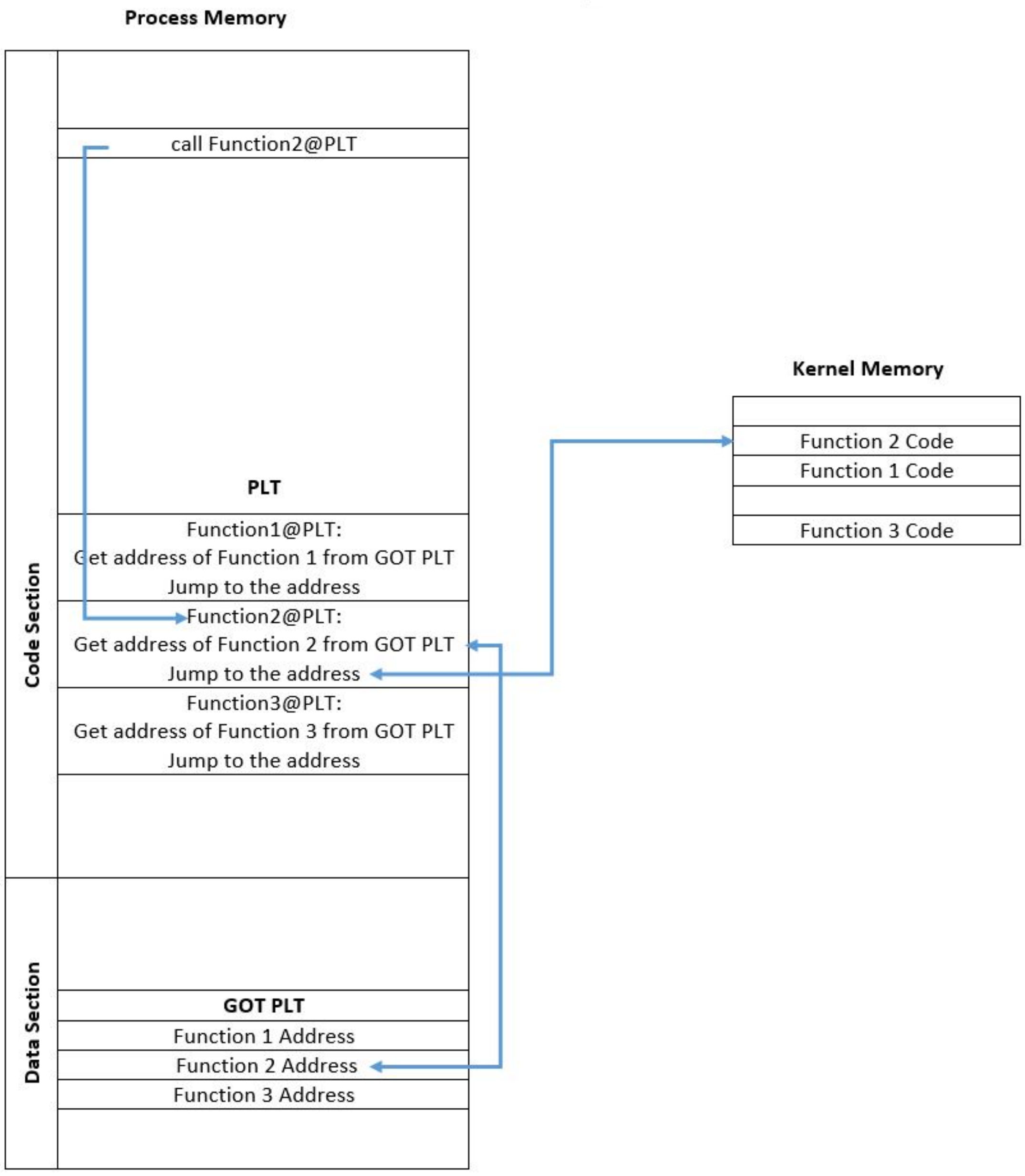


FIGURE 1.6: Procedure Linking Table (PLT)

one processor may map the RAM and ROM at some location in its address space and another processor may map them at different places. Hence, the linker description file for different processors will be different. This linker description file can also help us to add support of processes, by helping us to place the symbol tables, procedure linking table, global offset tables and other sections needed for process development. The syntax of linker description file depends on the toolchain being used. As, for our development we are using the GCC tool-chain, we will use its linker description file syntax. The syntax for linker description file is a language in itself and hence cannot be explained here in every detail. The reader can consult the GCC Linker's manual for complete understanding. We however will explain the functionalities we have used to accomplish our goal.

1.6.4 Dynamic Linking and Loading

A process is a separate program that can be loaded into an already running Operating System's image. The process will be built separately from the Operating System and loaded when the user wants to. What if a process wants to call a function or access a variable that was part of the code of Operating System? The process cannot be linked with the Operating System code, otherwise we will lose benefits such as loading and unloading of processes at run-time or dynamic updating capability. Here is where Dynamic Linking and Loading comes in. While building the process if it accesses a function or a variable that is not defined in its source code, the linker will insert some kind of mechanism to mark such access to be resolved at load time. This mechanism is called a relocation. When the process is being loaded, the loader code will process such relocations. It will resolve such relocations to point to the actual address of the variable or function being accessed. Note that the loader code will always be a part of the Operating System's code and hence will know the addresses of the functions and data variables that it wants to be available for processes. An Operating System can maintain a Symbol Table of Function and Data variable names and their addresses for this purpose. The processes will also have some relocation entries in it that would indicate the names of functions or data variable that need to be resolved at load time and where to resolve them in the process' image. The loader can look up the Symbol Table and fix these address references in the process to make the process able to run. We will explain how we have achieved this in Chapter 5. Implementation and Experimentation.

1.6.5 Executable and Linkable Format (ELF)

The Executable and Linkable Format ELF [14] is a standard file format for object files, executables, shared libraries etc. This format was originally published by UNIX System Laboratories as part of the Application Binary Interface. The purpose was to provide a consistent interface for programmers across multiple hardware and software platforms. The ELF file consists of an ELF Header, different sections, a program header table and a section header table. The ELF header presents general information about the file such as the computer architecture the ELF file was built for, version number and how the file is organized. There are different Sections and each may hold code, data, symbol tables, relocation information etc. The Section Header table describes how different Sections are placed in the file. The ELF File Format Standard has a very elaborate documentation. The documentation explores many use cases such as building executables, shared objects, object files, and performing relocations for dynamic linking and loading purposes. However, we only use a small subset of ideas in it to achieve our goals.

Chapter 2

Related Research

Dynamically adding and removing processes from a running system can serve as a way to update functionality of a running embedded system. A very good example of such use case has been demonstrated by [15]. The authors have demonstrated a use-case where satellite software can be updated dynamically while in orbit. The system is based on VxWorks Real Time Operating System. A protocol was developed to transmit object files remotely to the system and load them. VxWorks Operating System provides functionalities for dynamically loading modules which they had used to accomplish their goal. The paper does not detail how VxWorks implements its dynamic loading, since its source code is not open, but we believe that the techniques used would be much same as we will develop.

For FreeRTOS, a framework to update software at run-time has been demonstrated by [2] and [16]. The framework is designed to insert FreeRTOS tasks in an already running image. The tasks are built as separate ELF images and linked with an already running FreeRTOS kernel at run-time. A high-level depiction of this is shown in Figure 2.1. These ELF images are loaded by the framework, their entry points are recognized and these tasks are then added to the system and are then scheduled by the scheduler. The source code is also available at [17]. The framework only provides dynamic task updating capability. But we aim to bring more capabilities than that such as memory protection and privilege levels. Some advanced or modified methods of dynamically updating software have also been demonstrated by [18], [19] and [20].

The importance of Memory Protection in embedded systems is growing day by day with increasingly complex software. With such complex software it is inevitable that programming errors or malicious software can compromise the security of embedded software. With nanoscale CMOS technology becoming popular, even soft errors can cause such issues [21]. Most of the micro-processors and micro-controllers come equipped with a Memory Management Unit MMU or a Memory Protection Unit MPU in hardware that can be used to protect different regions of memory from accessing each other. [3] has developed a lightweight memory protection mechanism for three different types of

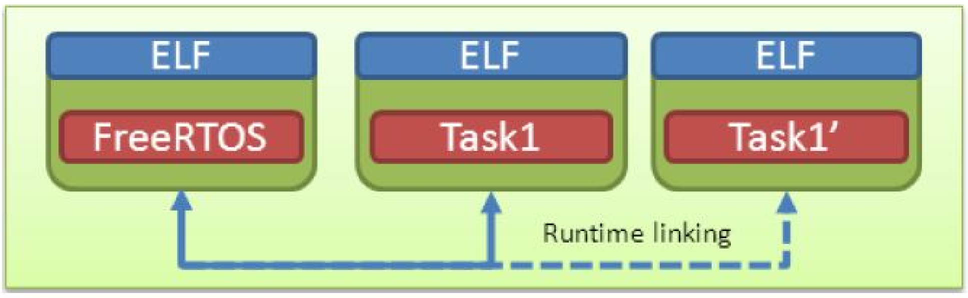


FIGURE 2.1: Run-Time Updating of FreeRTOS Tasks [2]

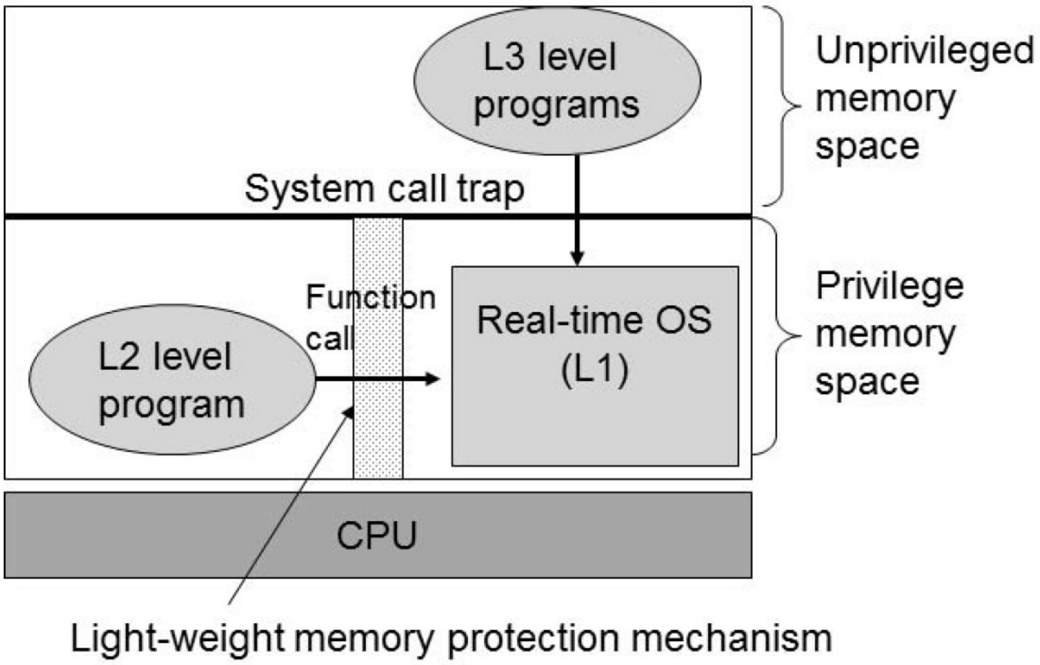


FIGURE 2.2: A Light-weight Memory Protection Mechanism implemented [3]

memory regions in privilege memory space. The author has defined three types of program categories as shown in Figure 2.2 taken from [3]. L1 type is the operating system code and L2 type is the privileged application code. L3 is for unprivileged user level applications. The mechanism developed implements a memory protection mechanism between the L1 and L2 types. If a call has to be made to access L1 from L2, a system call from L2 to L1 would be needed. This system call is not implemented by a software trap. Rather only a software mechanism is designed that changes the memory protection while going from L2 to L1 type region. This kind of software mechanism does away with software trap overhead. Though, we do not use such mechanism to implement our memory protection mechanism, the paper does provide us with valuable information on what has to be done to program a MMU or MPU to implement memory protection between regions.

The concept of process is very commonly used in Microsoft's Windows Operating System and Linux. Some RTOS vendors have also started to offer processes as a part of their solutions. The mechanisms behind processes implementation are generally not well explained in academia. These mechanisms include concepts such as compiling processes, dynamic loading and linking of processes etc. According to [11], these mechanisms are among the difficult to understand areas of software development. It is strongly recommended that the reader of this thesis studies [11] because it explains concepts behind shared libraries and dynamic loading. The authors have explained the differences between static and shared libraries. They have also shown how compilers and linkers can be used to generate code that can be loadable and link-able at run-time. They have also explained the concepts of Position Independent Code PIC, Global Offset Tables GOT and Procedure Linking Table PLT. Our processes would very much be treated as shared objects as far as loading and linking is concerned with one difference being that our processes will not export functionalities to any other processes. Our processes would only be able to access functionalities of the kernel through FreeRTOS APIs and implement their own.

Chapter 3

Motivation and Problem Statement

In this chapter we highlight some key factors that motivated me to carry out this work. We also define our goals that we need to achieve in this research.

3.1 Motivation

I have extensive experience in the field of embedded systems particularly in RTOS Kernels, driver and middle-ware development. Real Time Operating Systems have been deployed on every kind of hardware ranging from high end processors with large memories available to low-end micro-controllers. Real Time isnt always a requirement where resources are limited and the sole purpose is functionality and cost reduction. Many commercially available RTOS solutions now provide ability to dynamically update software. I aim to bring the concept of processes, very much similar to Linux Processes, to FreeRTOS to provide a general purpose framework for the open source community to use freely. Not only it will fulfill the needs for dynamic updating capability but also provide the added benefits of processes such as memory protection, privilege levels and kernel reliability and stability. And since most developers are familiar with the Linuxs process concept, grasping FreeRTOS processes concept would be much easier.

3.2 Problem Statement

By implementing a process framework in this research, we aim to bring following key benefits to the Free RTOS Solution:

1. Dynamic Loading and Unloading of processes for memory reuse.
2. In-field software updating.
3. Memory protection among address spaces of various processes.
4. Separation of processes into User Mode and Kernel Mode processes.
5. Improved reliability of the kernel.

Chapter 4

Proposed Approach

In this chapter, we will outline the step by step approach we will follow to accomplish our goal. The approach discussed is in general terms only. In next chapter we shall go into details.

4.1 Selecting a Hardware Platform

A hardware platform would be selected that would run FreeRTOS. We shall select a platform for which development tools are easily available and are license free. We desire to select a platform that is resource constrained in terms of memory so that the design of the process framework can be shown to be lightweight yet powerful enough to accomplish our goals.

4.2 Selecting a Tool-chain

A tool-chain will be selected that would contain the compiler, linker and other utilities to build FreeRTOS source code and processes. The tool-chain must have support for the processor architecture used for our development.

4.3 Selecting an Integrated Development Environment (IDE)

An Integrated Development Environment would be selected to carry out our work. Eclipse IDE would be the most suitable candidate, mostly because it is free. But we may need another IDE as Eclipse might not have the support to load code into and debug the hardware that we have selected or adding its support might take time and special expertise.

4.4 Building FreeRTOS Source Code

The next step would be to build the FreeRTOS source code for our hardware platform. This would involve setting up the compiler, linker and other utilities and integrating them into the Eclipse IDE. The source code can be downloaded from the FreeRTOS website. We would first try to build some sample applications and run them to become familiarized with FreeRTOS.

4.5 Building Processes

Building processes would be a tricky task. Processes must be built with special settings done for the compiler and linker. We would compile a process's code as a Position Independent code and as a shared object so that a Procedure Linking Table and a Global Offset table is generated. This is needed so that we can dynamically load and link the process into an already running FreeRTOS image.

4.6 Dynamic Loading and Linking of Processes

A framework would be written that would enable us to load the ELF image of the process into memory. The ELF file would be parsed for its different sections and needed sections would be copied into the memory allocated for the process at appropriate locations. The relocation entries in the file would also be processed. At this point we would be able to dynamically load and run a very basic process. This is the step where we would spend a considerable portion of our time. The ELF file format allows for a great deal of flexibility when it comes to dynamic linking and loading. We will need grasp its concepts and chose the best approach to achieve dynamic loading and linking for our processes.

4.7 Implement User and Kernel Mode Processes

At this step, we would implement support for running two different kinds of processes, Kernel Mode and User Mode Processes. Details will follow in next chapter.

4.8 Implement Memory Protection among Processes

Memory Protection would be implemented among the processes in this step. We will utilize the Memory Protection unit available on our hardware to accomplish this.

4.9 Testing

In this step we would test our Process Framework. Different processes will be loaded and unloaded, their privilege levels will be ensured and memory protection would be tested. All this would be done in order to show that kernel reliability has been increased.

Chapter 5

Implementation, Testing and Results

5.1 Tiva TM4C123GHPM ARM Cortex M4 Micro-controller

We selected the Tiva C Series EK-TM4C123GXL Kit [4] from Texas Instruments to carry out our work. This board has an 80 MHz 32-Bit ARM Cortex-M4 based Micro-controller with Floating Point Unit, Memory Protection Unit, 256 KB Flash and 32KB SRAM / 2KB EEPROM on-chip Memory. This micro-controller has very limited resources, yet it is powerful enough to run FreeRTOS. It would be a real challenge to implement our processes support for FreeRTOS and yet keep our resource utilization to a minimum. We would load, unload and run processes from its mere 32KB of on-chip RAM. Its memory protection Unit will be used to protect processes from accessing each others memory. This micro-controller was also selected because Cortex-M4 is a very popular choice for low-cost and low-power embedded systems applications. Also, Texas Instruments, provide their Code Composer Studio IDE to be used with this board free of cost. We will use this IDE to load and debug code only. The code development will be done in Eclipse IDE using freely available GNU ARM compiler as explained in the next sections.

5.2 Eclipse

We use the Eclipse open source IDE for C and C++ development to carry out our work. This IDE can be integrated easily with a wide variety of tool-chains and provides a very intuitive environment for code development. However, to debug and load code into the Tiva C Series TM4C123G Launchpad Evaluation Kit we use the Texas Instruments Code Composer Studio IDE.

5.3 Tool-chain used for Code Development

We use the GNU ARM Embedded Tool-chain available from the ARM Developer website [22]. This tool-chain consists of the GCC Compiler, linker, libraries etc. for software development on ARMs Cortex-M and Cortex-R based devices. The compiler and linker can

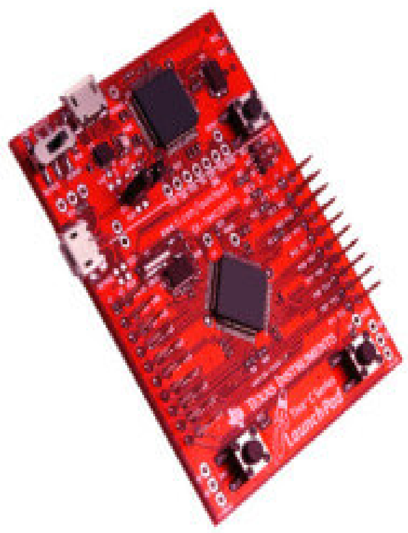
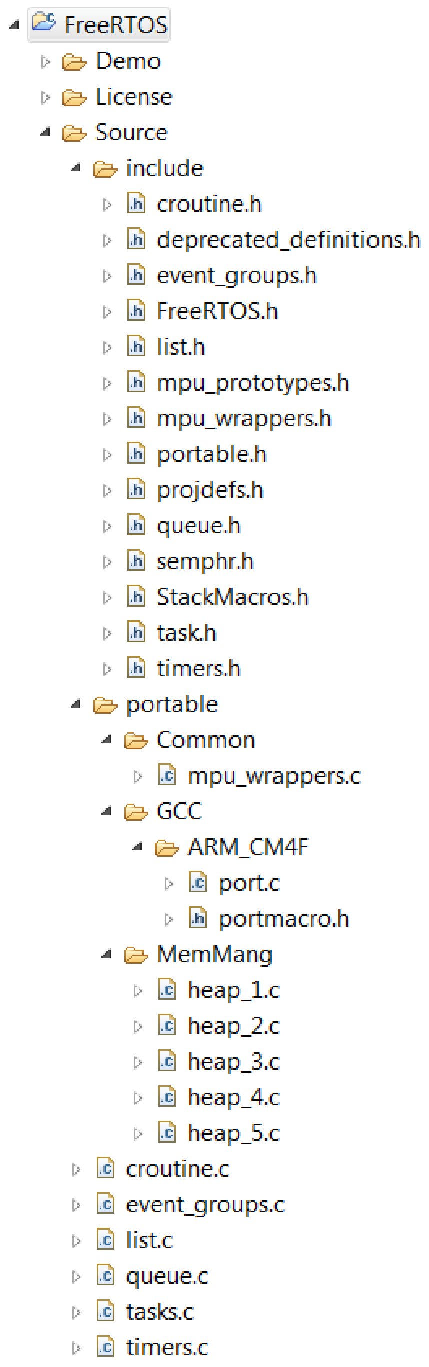


FIGURE 5.1: Tiva C Series TM4C123G Launchpad Evaluation Kit [4]

easily be integrated with Eclipse IDE which we are using to carry out our development. The version of the tool-chain used by us is version 5-2016-q3.

5.4 FreeRTOS Directory Structure

The FreeRTOS Source code can be downloaded from its website. The directory structure is shown in Figure 5.2. The downloaded zip file contains a *Source* folder which contains the source code for the FreeRTOS kernel. The directory structure of FreeRTOS 9.0.0 is shown below. The source folder contains one C file for each functionality of Tasks services, Queue Services, List, Event Groups, Timers and Co-routines. The source folder contains an *include* folder and a *portable* folder. The *include* folder as its name implies contains header files related to FreeRTOS kernel. The *portable* folder contains hardware and architecture dependent files for supported toolchains. For our use case we have a folder for GNU Compiler Collection (GCC) Toolchain and in that folder we have a folder *ARM_CM4F* which contains the portable code for ARMs Cortex-M4 Architecture. The *portable* folder also contains a *MemMang* folder. This folder contains different C files, each containing different heap allocation algorithm. The user can keep any single file according to its needs and application. The *Common* folder in *portable* contains wrappers for privileged functions in case MPU is being used. We will not use MPU the way FreeRTOS uses it, so we can ignore this folder. The *Demo* folder will contain the main application code and the code and linker description file specific to our selected hardware platform. It also contains the *FreeRTOSConfig.h* file which is used to configure the FreeRTOS build using macros.



```
FreeRTOS
  Demo
  License
  Source
    include
      croutine.h
      deprecated_definitions.h
      event_groups.h
      FreeRTOS.h
      list.h
      mpu_prototypes.h
      mpu_wrappers.h
      portable.h
      projdefs.h
      queue.h
      semphr.h
      StackMacros.h
      task.h
      timers.h
    portable
      Common
        mpu_wrappers.c
      GCC
        ARM_CM4F
          port.c
          portmacro.h
      MemMang
        heap_1.c
        heap_2.c
        heap_3.c
        heap_4.c
        heap_5.c
    croutine.c
    event_groups.c
    list.c
    queue.c
    tasks.c
    timers.c
```

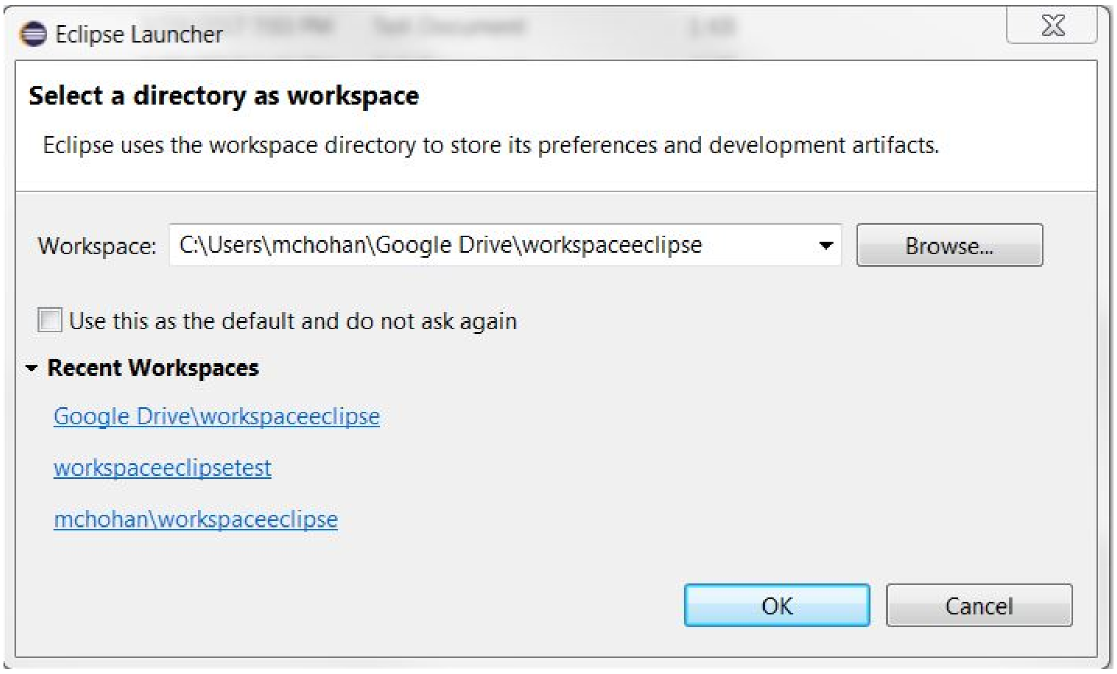
Figure 5.2: FreeRTOS Directory Structure

5.4.1 Files added and modified for Process Framework in FreeRTOS

Two support processes in FreeRTOS we have added two new C files *processes.c* and *FreeRTOSExports.c* in the *Source* folder. Two header files *processes.h* and *elf32.h* are added in the *include*. Apart from adding these files we have modified the *port.c*, *portmacro.h*, *FreeRTOS.h*, *portable.h* and *task.c*. The new code introduced and modifications done have been kept to a minimum in order to keep the footprint of the framework small. The source code of our modified FreeRTOS with process support has been placed at [23] along with some sample processes.

5.5 Setting up Environment in Eclipse

In this section we shall explain how to setup environment in Eclipse IDE to build FreeRTOS Kernel and our processes for our selected hardware. The compiler and linker flags we describe in the next section are crucial for implementing the processes support. Launch Eclipse IDE and select an appropriate workspace. Double-click the installed Eclipse IDE icon. A dialog box box as follow will appear:


Eclipse Launcher

Select a directory as workspace

Eclipse uses the workspace directory to store its preferences and development artifacts.

Workspace: C:\Users\mchohan\Google Drive\workspaceeclipse

☐ Use this as the default and do not ask again

Recent Workspaces

Google Drive\workspaceeclipse

workspaceeclipsetest

mchohan\workspaceeclipse

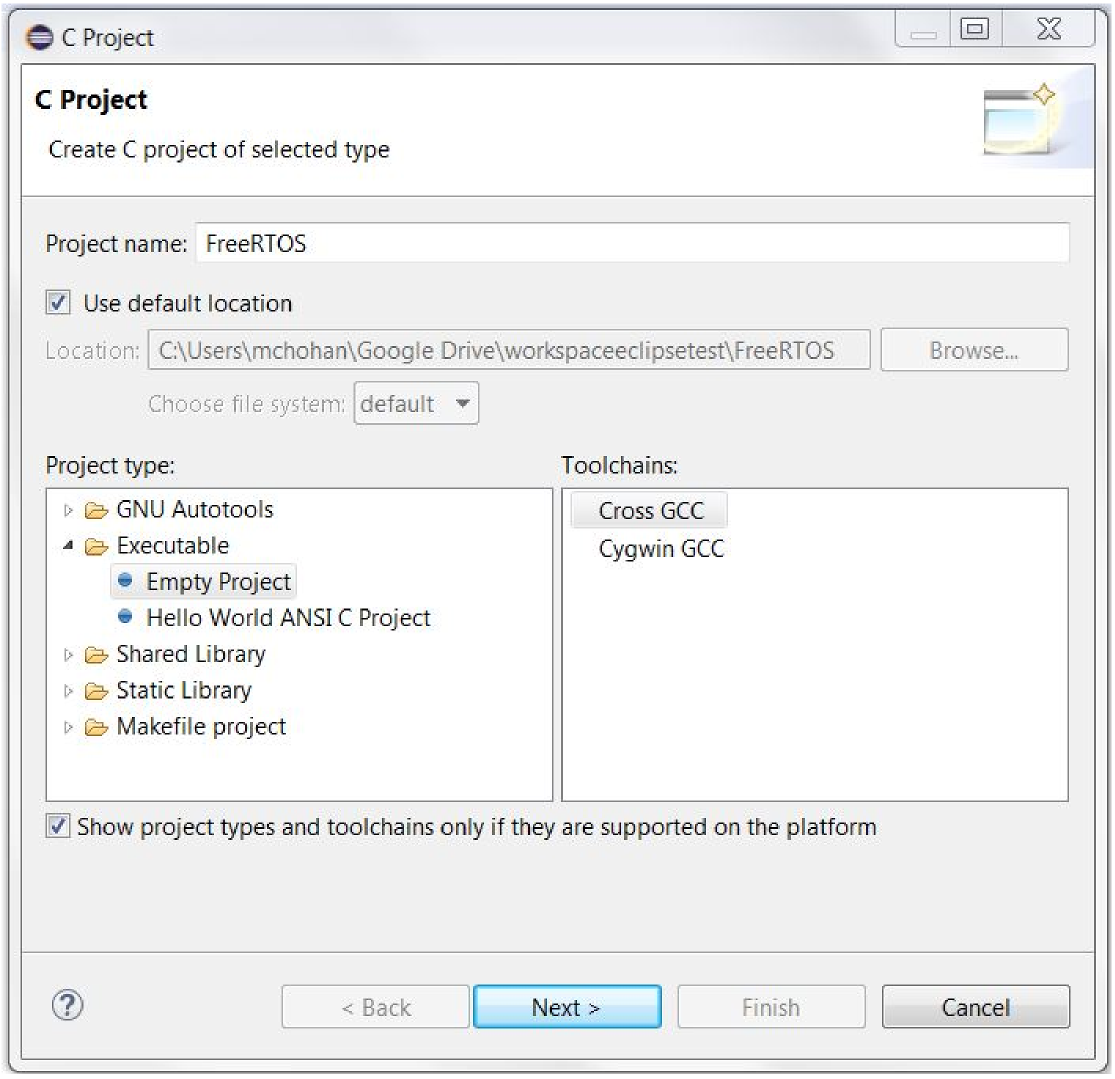
FIGURE 5.3: Select a directory as workspace

Select an appropriate workspace, as we did and click OK.

5.5.1 Setting up Environment in Eclipse to Build Kernel

In this section we show how to create a C project in Eclipse IDE to build FreeRTOS Kernel Image.

1. Click File - New - C Project. A window like following would appear. Select the options as shown and click Next.


C Project

C Project

Create C project of selected type

Project name: FreeRTOS

☑ Use default location

Location: C:\Users\mchohan\Google Drive\workspaceeclipsetest\FreeRTOS

Choose file system: default

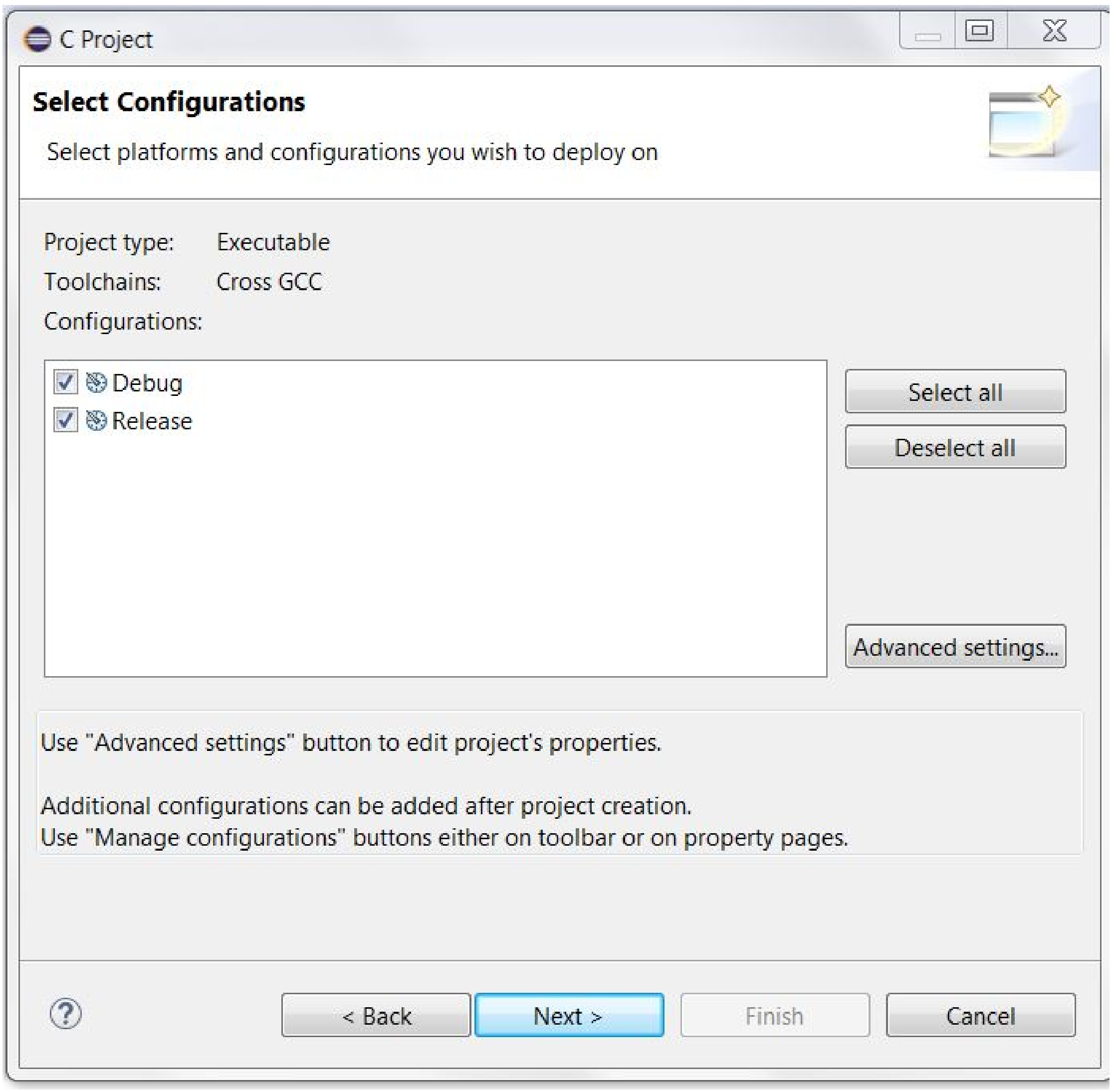
Project type:
- GNU Autotools
- Executable
  - Empty Project
  - Hello World ANSI C Project
- Shared Library
- Static Library
- Makefile project

Toolchains:
- Cross GCC
- Cygwin GCC

☑ Show project types and toolchains only if they are supported on the platform

FIGURE 5.4: Create a C project for building FreeRTOS

2. In the Select Configurations window select options as shown and click Next as shown.


C Project

Select Configurations

Select platforms and configurations you wish to deploy on

Project type: Executable
Toolchains: Cross GCC
Configurations:

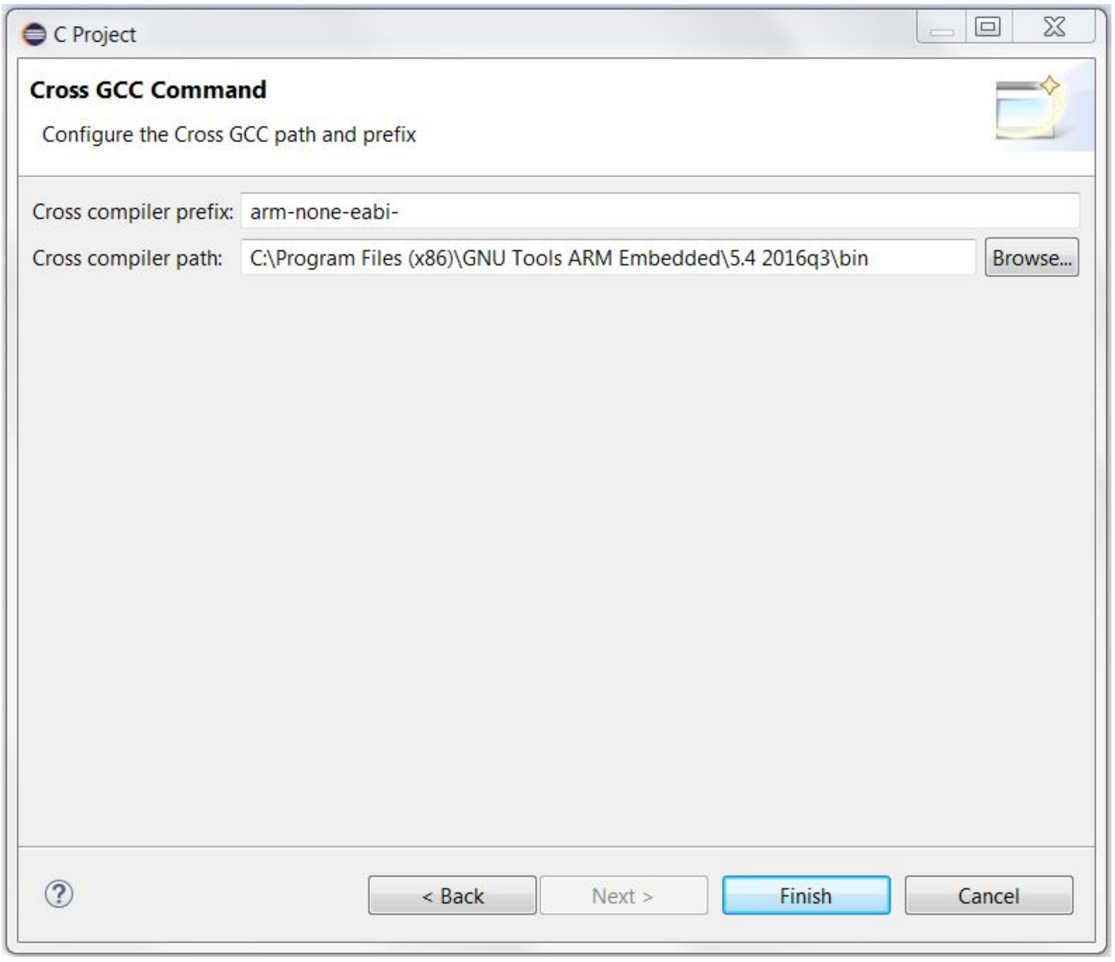
- [✓] Debug
- [✓] Release

Use "Advanced settings" button to edit project's properties.

Additional configurations can be added after project creation.
Use "Manage configurations" buttons either on toolbar or on property pages.

FIGURE 5.5: Select Configurations for building FreeRTOS

3. In Cross GCC Command Window enter the Cross compiler prefix as shown and give the path of the bin folder of the Compiler Tool-chain we installed. Click Finish after this. Now you will see a C Project ”FreeRTOS” created in the Project Explorer window of Eclipse.


C Project

Cross GCC Command

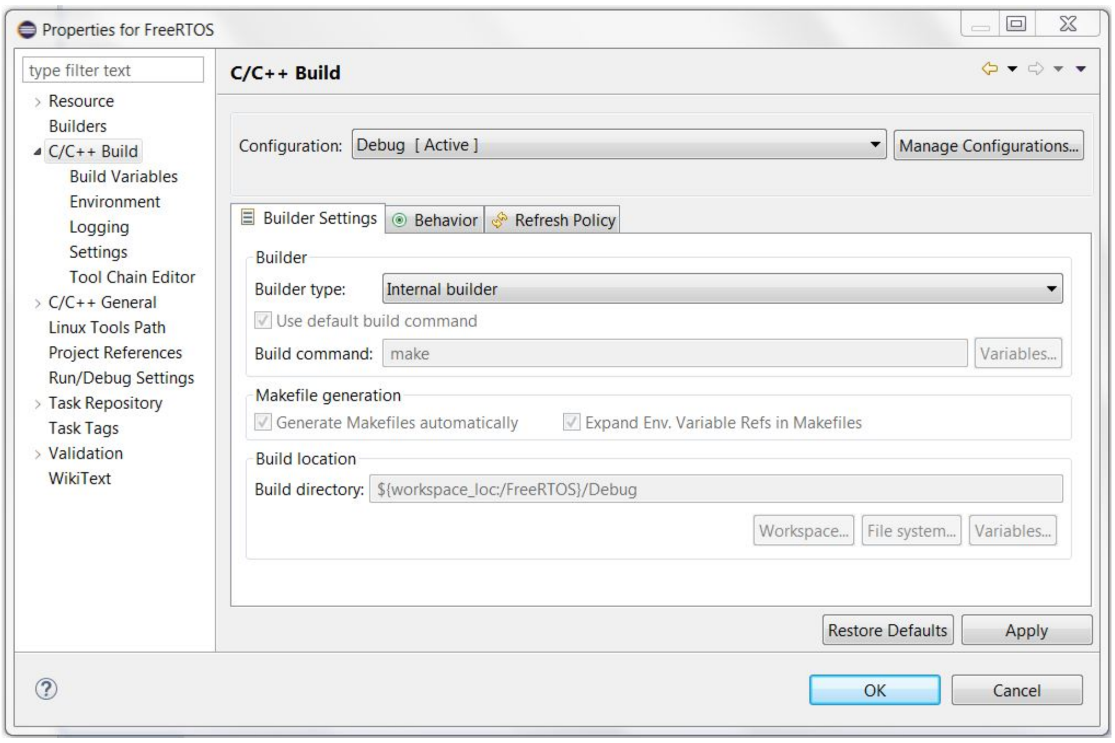
Configure the Cross GCC path and prefix

Cross compiler prefix: arm-none-eabi-

Cross compiler path: C:\Program Files (x86)\GNU Tools ARM Embedded\5.4 2016q3\bin

Figure 5.6: Cross GCC Command for building FreeRTOS

4. Right-Click on the FreeRTOS project and select Properties. A properties dialog box like following would appear. In the C/C++ Build tab, configure the options as shown.



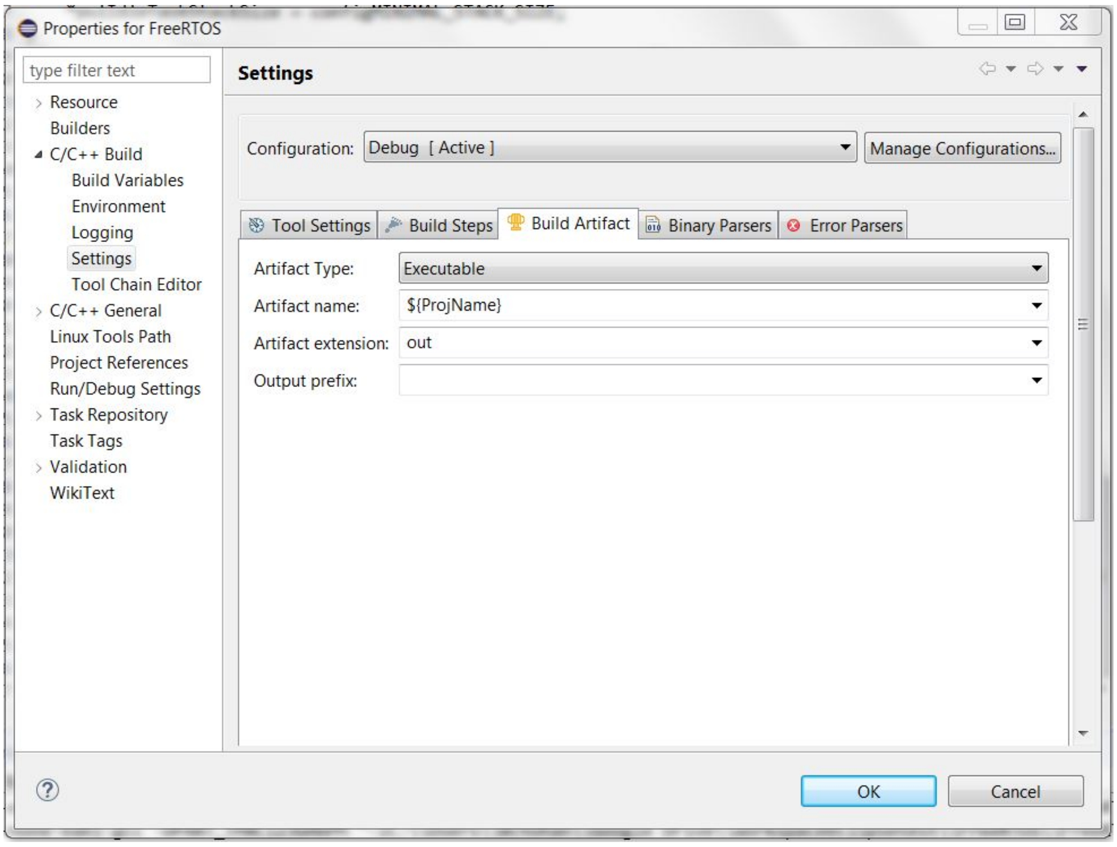
Configuration: Debug [Active]

Builder
Builder type: Internal builder
☑ Use default build command
Build command: make

Makefile generation
☑ Generate Makefiles automatically
☑ Expand Env. Variable Refs in Makefiles

Build location
Build directory: ${workspace_loc:/FreeRTOS}/Debug

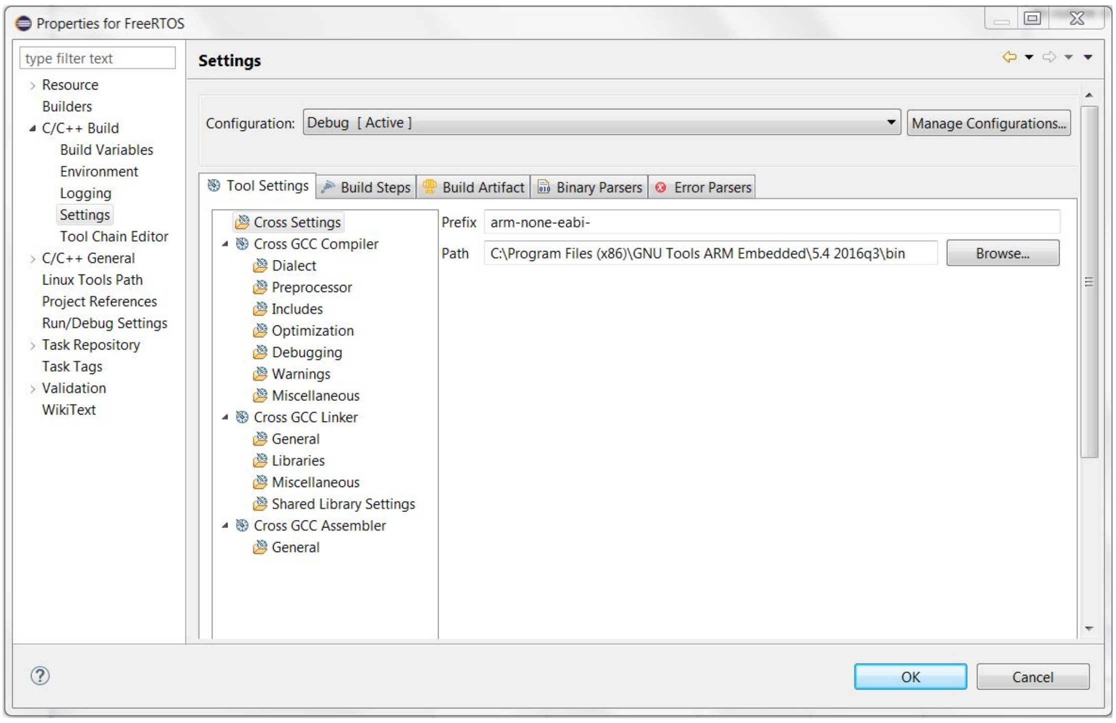
FIGURE 5.7: Configuring C/C++ Build Settings for building FreeRTOS - Step 1



Configuration: Debug [Active]

Artifact Type: Executable
Artifact name: ${ProjName}
Artifact extension: out
Output prefix:

FIGURE 5.8: Configuring C/C++ Build Settings for building FreeRTOS - Step 2

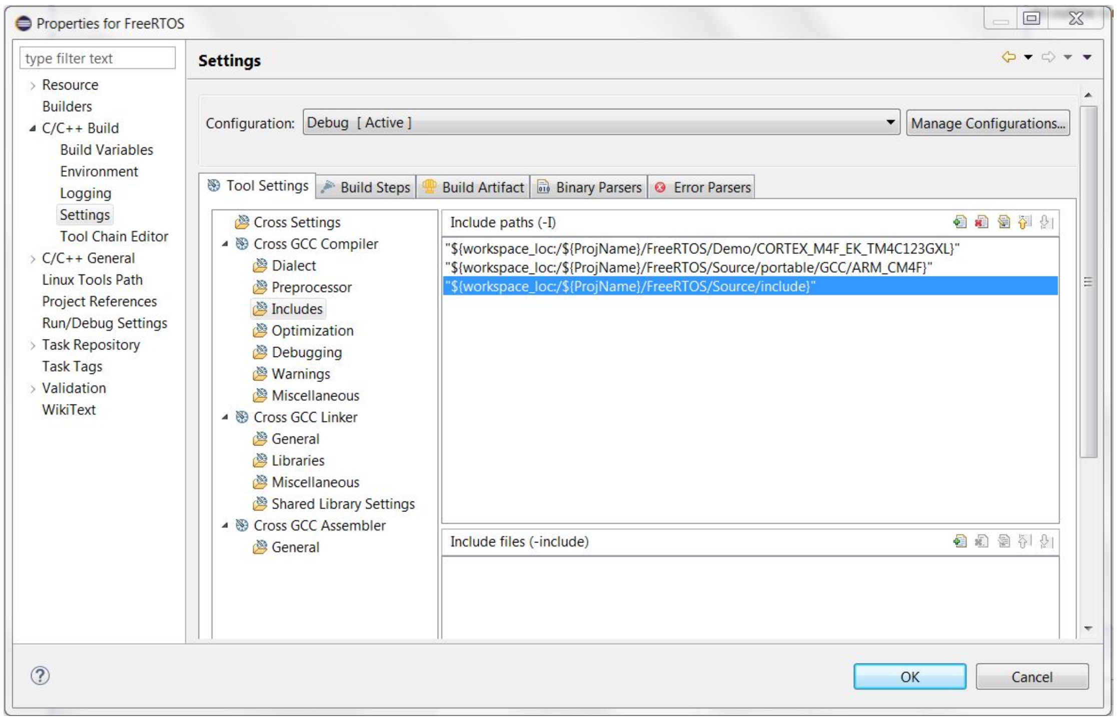


Configuration: Debug [Active]

Prefix arm-none-eabi-
Path C:\Program Files (x86)\GNU Tools ARM Embedded\5.4 2016q3\bin

FIGURE 5.9: Configuring C/C++ Build Settings for building FreeRTOS - Step 3

5. Set the include paths as follows. These are according to the FreeRTOS Source Code Structures.


Properties for FreeRTOS

Settings

Configuration: Debug [Active]

Tool Settings | Build Steps | Build Artifact | Binary Parsers | Error Parsers

Cross GCC Compiler > Includes

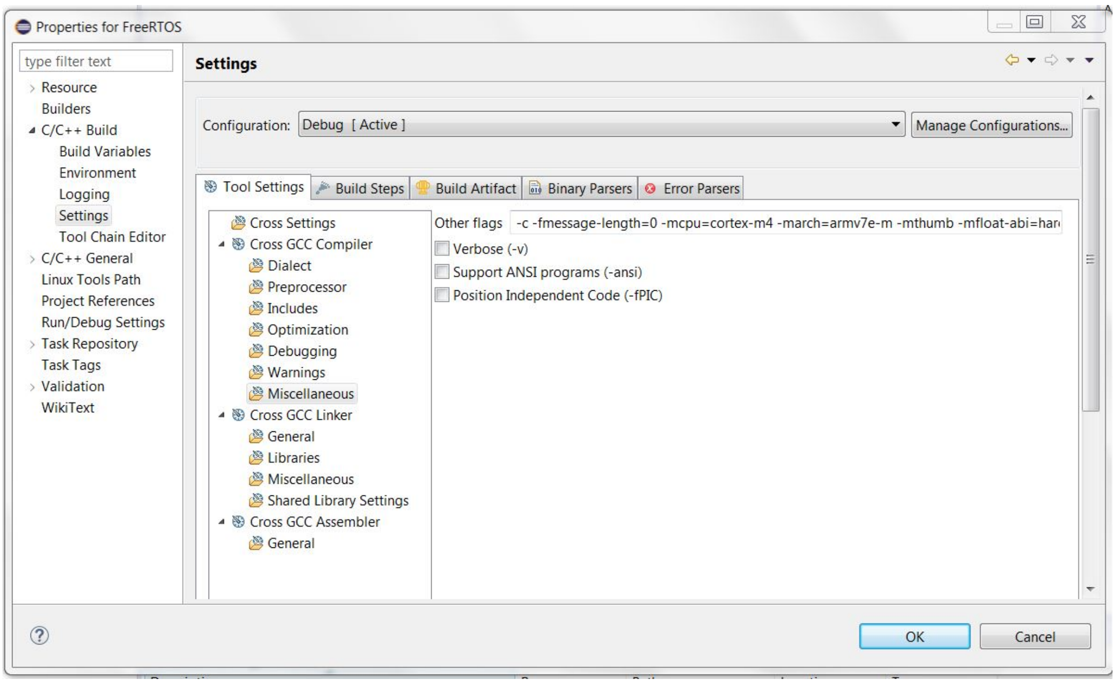
Include paths (-I)

```
"${workspace_loc:/${ProjName}/FreeRTOS/Demo/CORTEX_M4F_EK_TM4C123GXL}"
"${workspace_loc:/${ProjName}/FreeRTOS/Source/portable/GCC/ARM_CM4F}"
"${workspace_loc:/${ProjName}/FreeRTOS/Source/include}"
```

Include files (-include)

FIGURE 5.10: Configuring include paths for building FreeRTOS

6. In the Cross GCC Compiler - Miscellaneous Tab enter these flags in the Other Flags section. ”-c -fmessage-length=0 -mcpu=cortex-m4 -march=armv7e-m -mthumb -mfloat-abi=hard -mfpu=fpv4-sp-d16 -ffunction-sections -fdata-sections -g -gdwarf-3 -gstrict-dwarf -Wall -MD -std=c99”. Have a look at the GCC Toolchain’s Compiler Documentation for details about what each flag does.


Properties for FreeRTOS

Settings

Configuration: Debug [Active]

Tool Settings | Build Steps | Build Artifact | Binary Parsers | Error Parsers

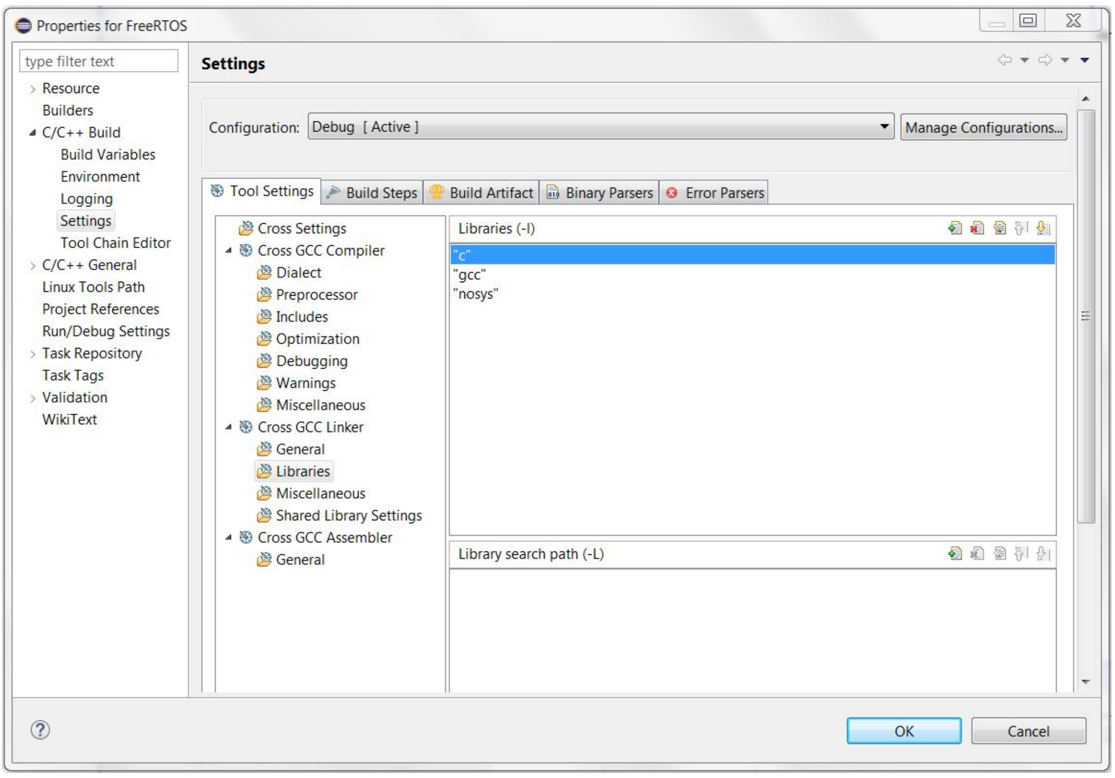
Cross GCC Compiler > Miscellaneous

Other flags: -c -fmessage-length=0 -mcpu=cortex-m4 -march=armv7e-m -mthumb -mfloat-abi=har

☐ Verbose (-v)
☐ Support ANSI programs (-ansi)
☐ Position Independent Code (-fPIC)

FIGURE 5.11: Configuring Compiler Flags for building FreeRTOS

7. Configure the Cross GCC Linker's libraries as shown. Have a look at the GCC Toolchain's Documentation for details about these libraries.



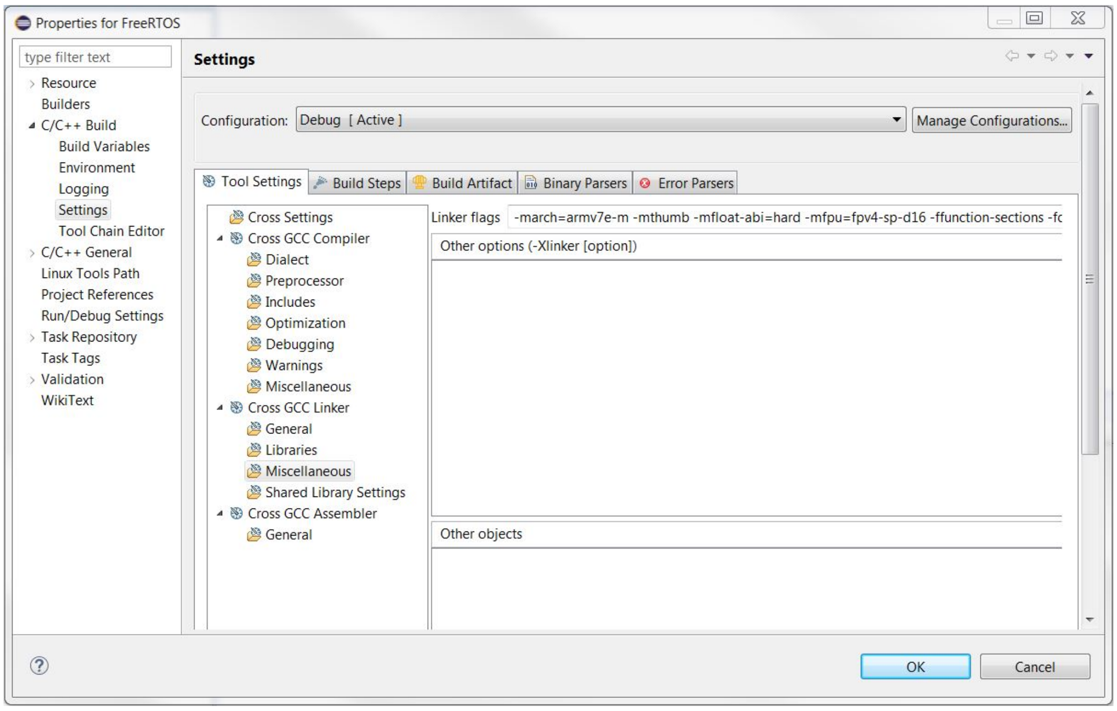
Configuration: Debug [Active]

Libraries (-l)
"c"
"gcc"
"nosys"

Library search path (-L)

FIGURE 5.12: Configuring Linker Libraries for building FreeRTOS

8. In the Cross GCC Linker - Miscellaneous Tab enter these flags in the Other Flags section. "-march=armv7e-m -mthumb -mfloat-abi=hard -mfpu=fpv4-sp-d16 -ffunction-sections -fdata-sections -g -gdwarf-3 -gstrict-dwarf -Wall -Wl,-Map,"FreeRtos9.0.0.map" -Wl, -T "${workspace_loc:/${ProjName}/FreeRTOS/Demo/CORTEX_M4F_EK_TM4C123GXL/tm4c123gh6pm_ld.lds}" ".Have a look at the GCC Tool-chain's Compiler Documentation for details about what these flags do. The -T option specifies the linker description file to use to build the final image. We have created a special linker description file to support processes.



Configuration: Debug [Active]

Linker flags -march=armv7e-m -mthumb -mfloat-abi=hard -mfpu=fpv4-sp-d16 -ffunction-sections -fc

Other options (-Xlinker [option])

Other objects

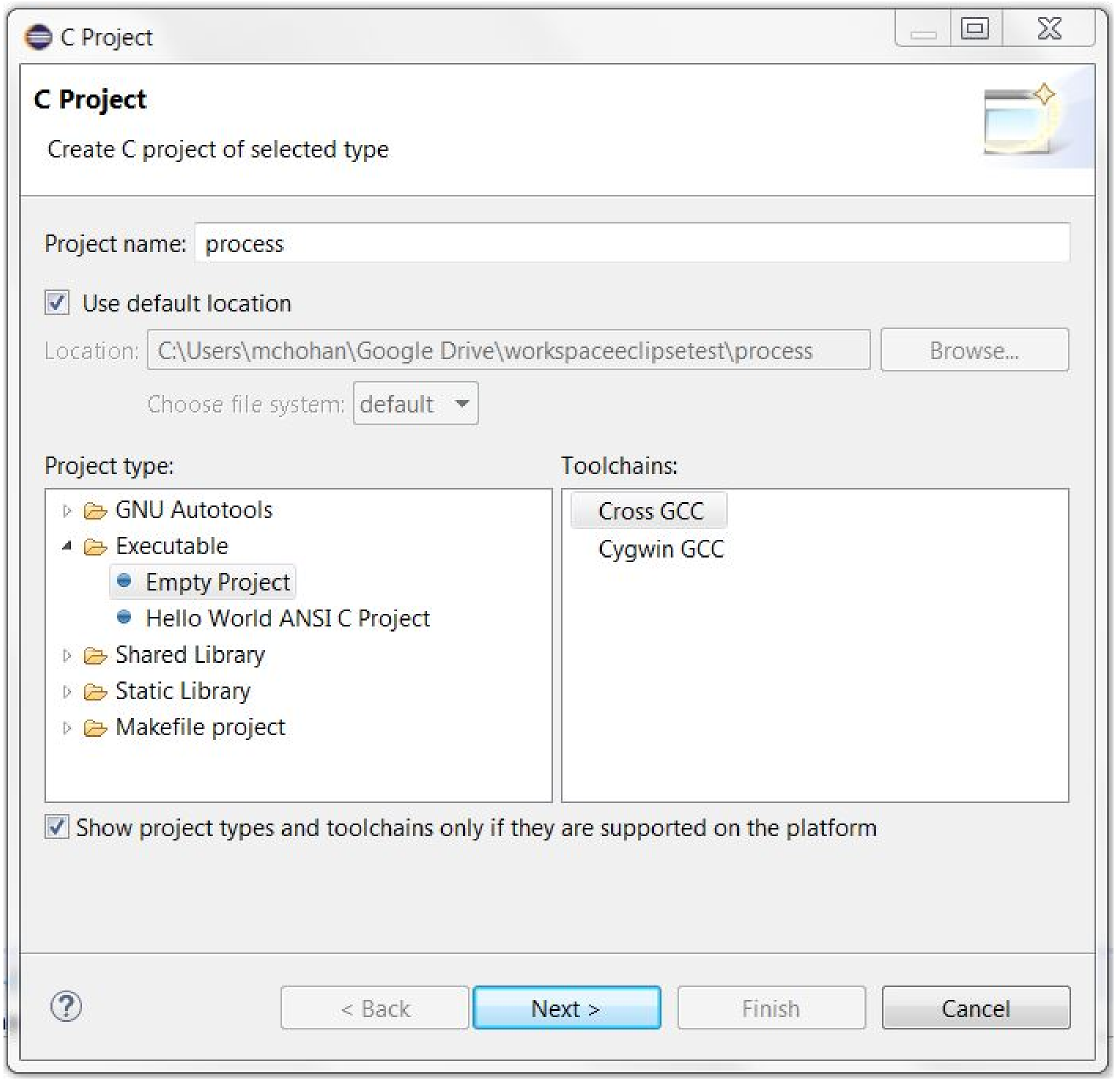
FIGURE 5.13: Configuring Linker Flags for building FreeRTOS

After carrying out these steps, copy the *FreeRTOS* folder from [23] we have provided into this project and build it. The linker description file is contained within the *demo* folder.

5.5.2 Setting up Environment in Eclipse to Build Processes

In this section, we explain how to create our proposed processes in Eclipse IDE. The processes would also be created as C Projects as we did earlier to build the FreeRTOS kernel, but there would be some differences. The processes projects will only contain the code for the processes and the header files of the FreeRTOS source code. There would be no source code of FreeRTOS in it. Also, the processes would be built using special compiler and linker flags and settings and a special linker description file. These flags and compiler settings are crucial for the code to be able to run as processes.

1. Click File - New - C Project. A window like following would appear. Select the options as shown and click Next.



C Project

C Project

Create C project of selected type

Project name: process

☑ Use default location

Location: C:\Users\mchohan\Google Drive\workspaceeclipsetest\process

Choose file system: default

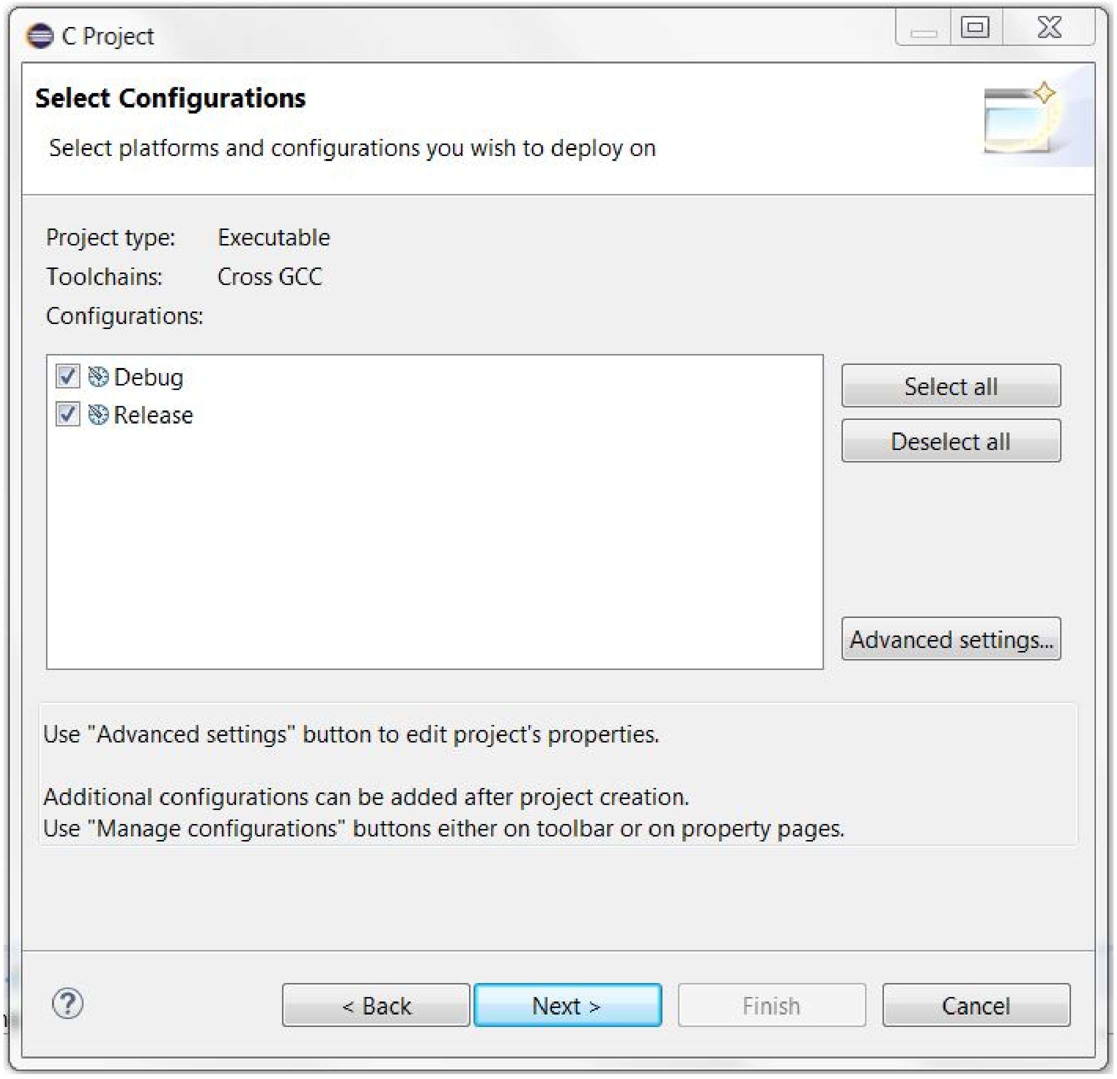
Project type: Executable > Empty Project

Toolchains: Cross GCC

☑ Show project types and toolchains only if they are supported on the platform

FIGURE 5.14: Create C Project for building Process

2. In the Select Configurations window select options as shown and click Next as shown.



Select Configurations

Select platforms and configurations you wish to deploy on

Project type: Executable
Toolchains: Cross GCC
Configurations:

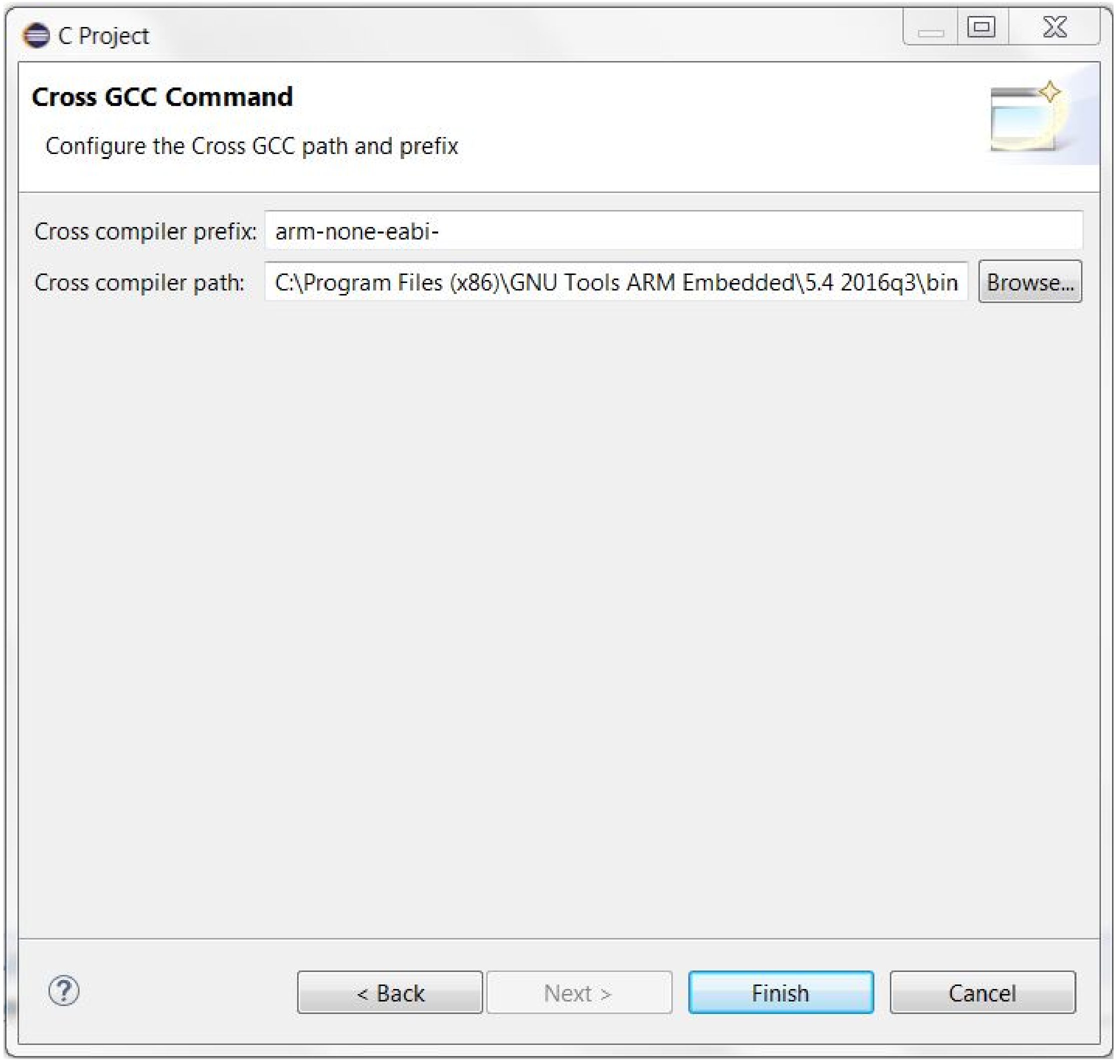
✓ Debug
✓ Release

Use "Advanced settings" button to edit project's properties.

Additional configurations can be added after project creation.
Use "Manage configurations" buttons either on toolbar or on property pages.

FIGURE 5.15: Select Configurations for building Process

3. In Cross GCC Command Window enter the Cross compiler prefix as shown and give the path of the bin folder of the Compiler Tool-chain we installed. Click Finish after this. Now you will see a C Project ”process” created in the Project Explorer window of Eclipse.


C Project

Cross GCC Command

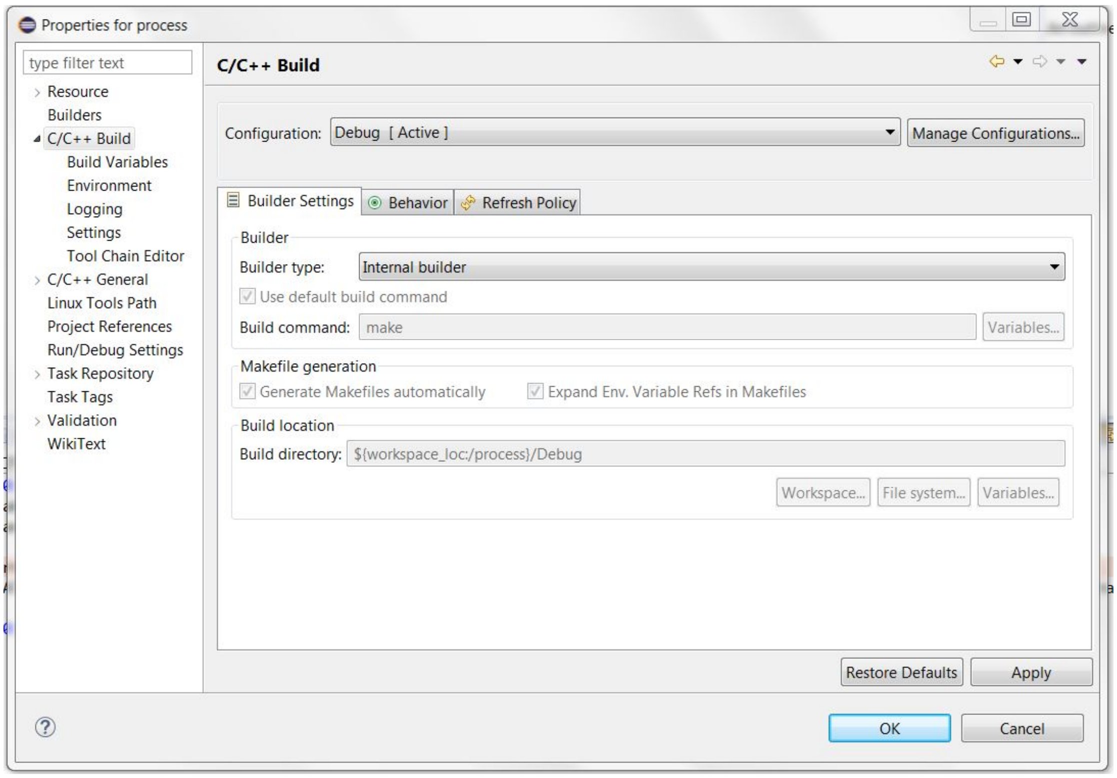
Configure the Cross GCC path and prefix

Cross compiler prefix: arm-none-eabi-

Cross compiler path: C:\Program Files (x86)\GNU Tools ARM Embedded\5.4 2016q3\bin

Figure 5.16: Cross GCC Command for building Process

4. Right-Click on the process project and select Properties. A properties dialog box like following would appear. In the C/C++ Build tab, configure the options as shown.


Properties for process

type filter text

- Resource
- Builders
- C/C++ Build
  - Build Variables
  - Environment
  - Logging
  - Settings
  - Tool Chain Editor
- C/C++ General
- Linux Tools Path
- Project References
- Run/Debug Settings
- Task Repository
- Task Tags
- Validation
- WikiText

C/C++ Build

Configuration: Debug [Active]

Manage Configurations...

Builder Settings | Behavior | Refresh Policy

Builder

Builder type: Internal builder

☑ Use default build command

Build command: make

Variables...

Makefile generation

☑ Generate Makefiles automatically ☑ Expand Env. Variable Refs in Makefiles

Build location

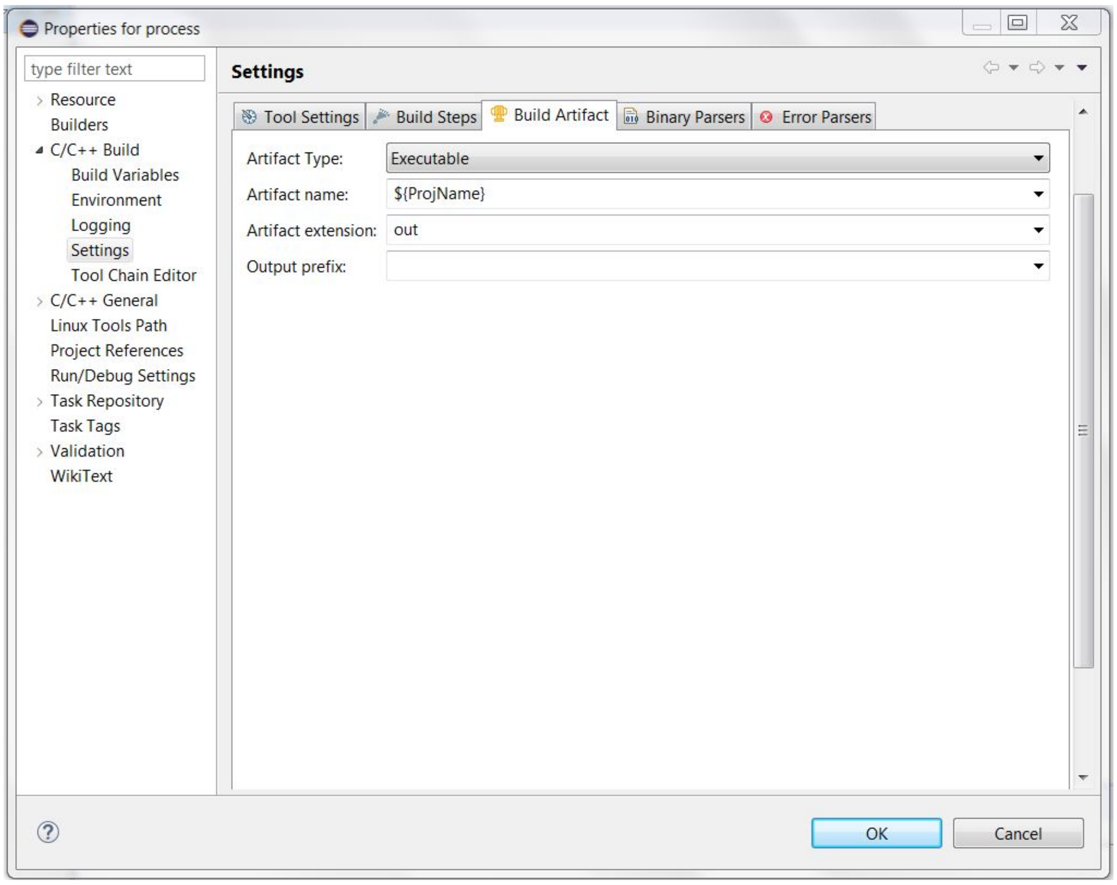
Build directory: ${workspace_loc:/process}/Debug

Workspace... File system... Variables...

Restore Defaults Apply

OK Cancel

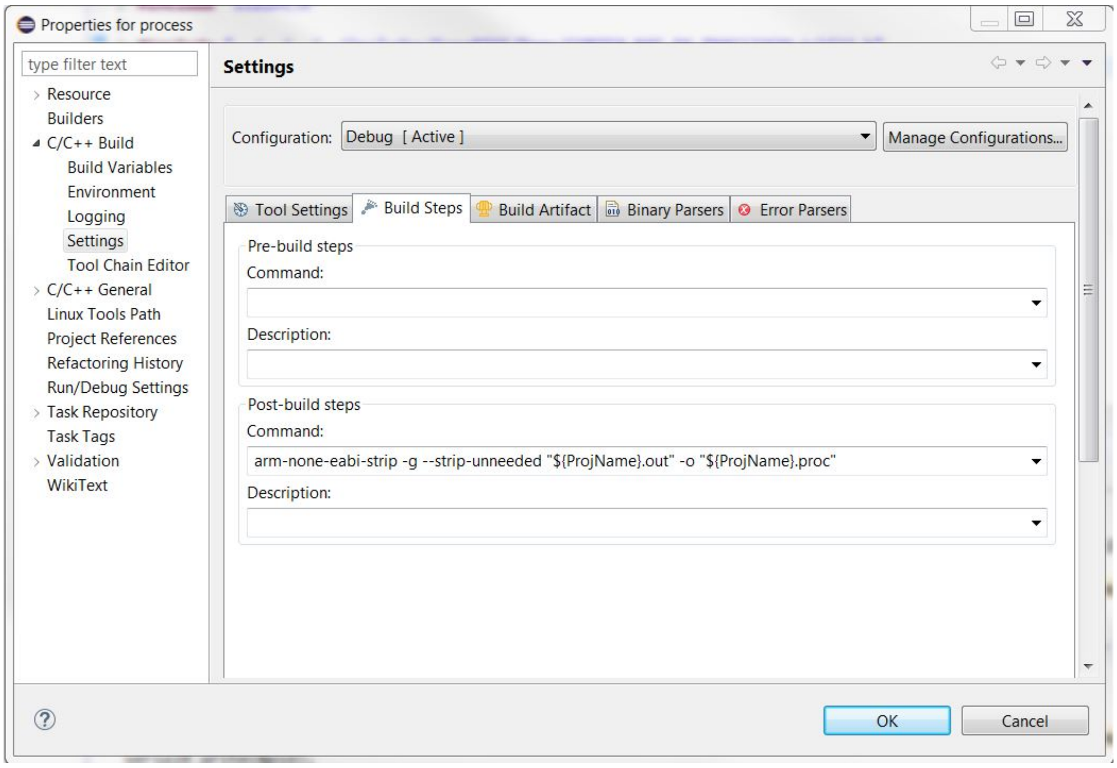
FIGURE 5.17: Configuring C/C++ Build Settings for building Process - Step 1



Artifact Type: Executable
Artifact name: ${ProjName}
Artifact extension: out
Output prefix:

FIGURE 5.18: Configuring C/C++ Build Settings for building Process - Step 2

5. In the Post-Build Steps give the Command as shown. This command will remove debug symbols and symbols not needed for relocation purposes.



Configuration: Debug [Active]

Pre-build steps
Command:
Description:

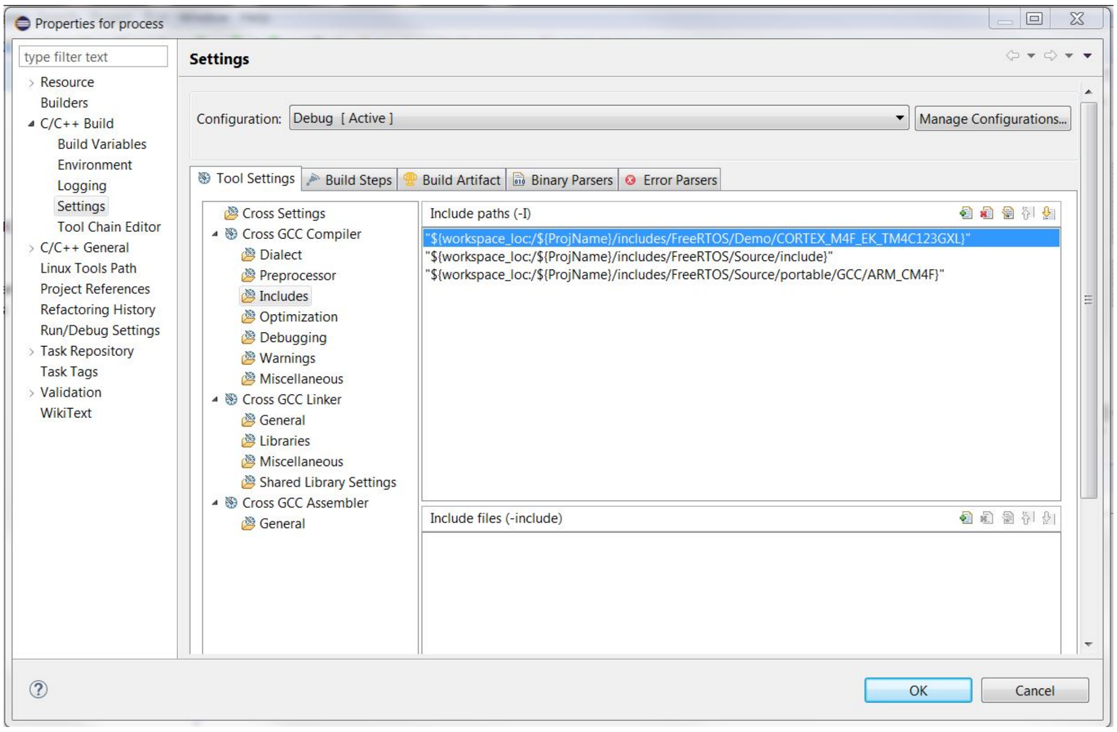
Post-build steps
Command:

```
arm-none-eabi-strip -g --strip-unneeded "${ProjName}.out" -o "${ProjName}.proc"
```

Description:

FIGURE 5.19: Configuring C/C++ Build Settings for building Process - Step 3

6. Setup the include paths as follows. Our process will be needing all the header files of the FreeRTOS source code to build. This requirement will in no way have any code or data size overhead as the header files do not contribute to it. To do this create an *includes* folder in the process project. Then copy the entire FreeRTOS directory from the FreeRTOS project into this ”includes” folder. Now delete all the Source files with .c extension and keep the .h header files intact. In this way we would only end up with the header files. If the include paths are set up as shown below, the process would build successfully. Alternatively, to save time the *includes* folder from [23] can also be copied.



Configuration: Debug [Active]

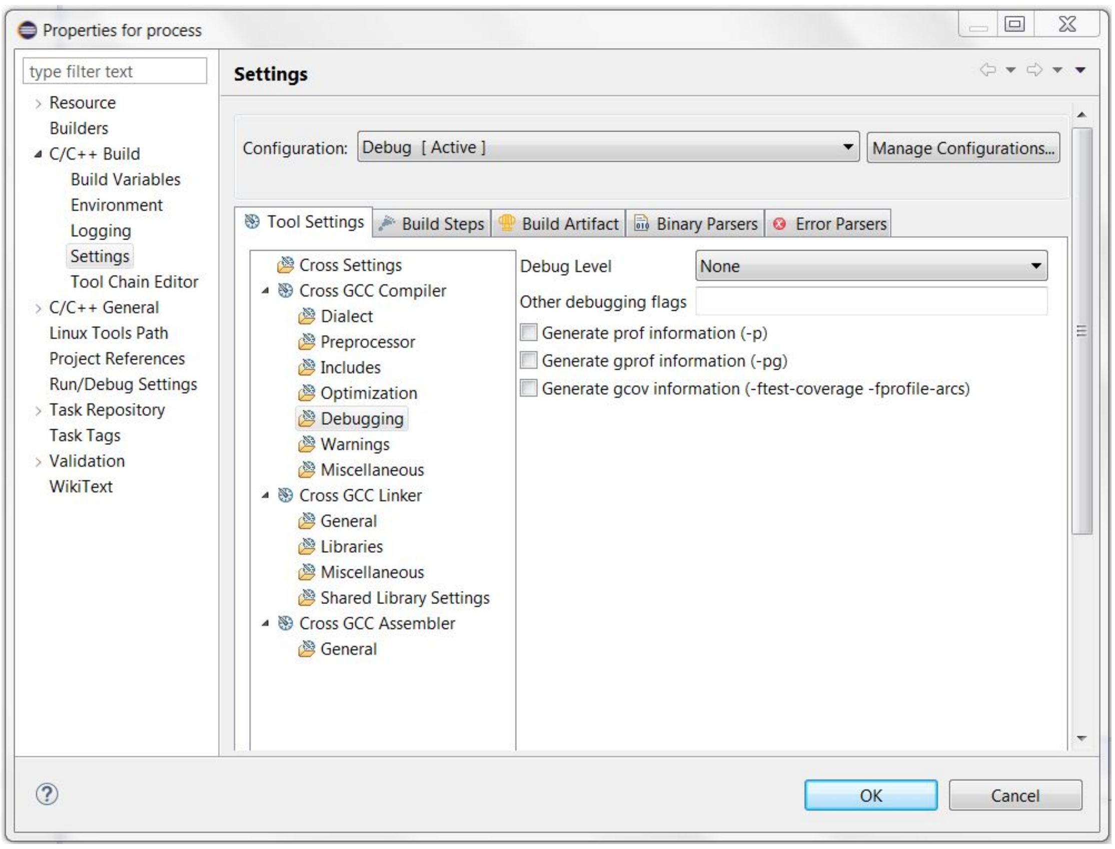
Include paths (-I)

```
"${workspace_loc:/${ProjName}/includes/FreeRTOS/Demo/CORTEX_M4F_EK_TM4C123GXL}"
"${workspace_loc:/${ProjName}/includes/FreeRTOS/Source/include}"
"${workspace_loc:/${ProjName}/includes/FreeRTOS/Source/portable/GCC/ARM_CM4F}"
```

Include files (-include)

FIGURE 5.20: Configuring C/C++ Build Settings for building Process - Step 4

7. Setup the debugging settings as follows.



Configuration: Debug [Active]

Debug Level: None

Other debugging flags

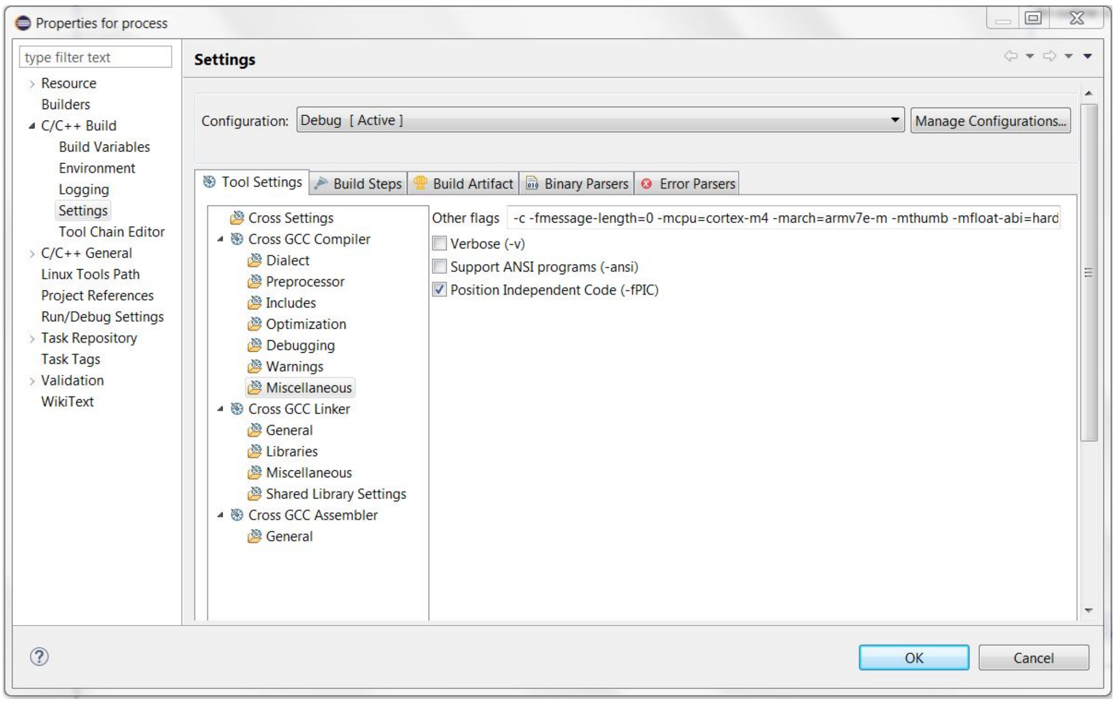
Generate prof information (-p)

Generate gprof information (-pg)

Generate gcov information (-ftest-coverage -fprofile-arcs)

FIGURE 5.21: Configuring C/C++ Build Settings for building Process - Step 5

8. In the Cross GCC Compiler - Miscellaneous Tab enter these flags in the Other Flags section. "-c -fmessage-length=0 -mcpu=cortex-m4 -march=armv7e-m -mthumb -mfloat-abi=hard -mfpu=fpv4-sp-d16 -ffunction-sections -fdata-sections -g -gdwarf-3 -gstrict-dwarf -Wall -MD -std=c99 -nostdlib". Have a look at the GCC Tool-chain's Compiler Documentation for details about what each flag does. Also enable the Position Independent Code -fPIC check box.



Configuration: Debug [Active]

Other flags -c -fmessage-length=0 -mcpu=cortex-m4 -march=armv7e-m -mthumb -mfloat-abi=hard

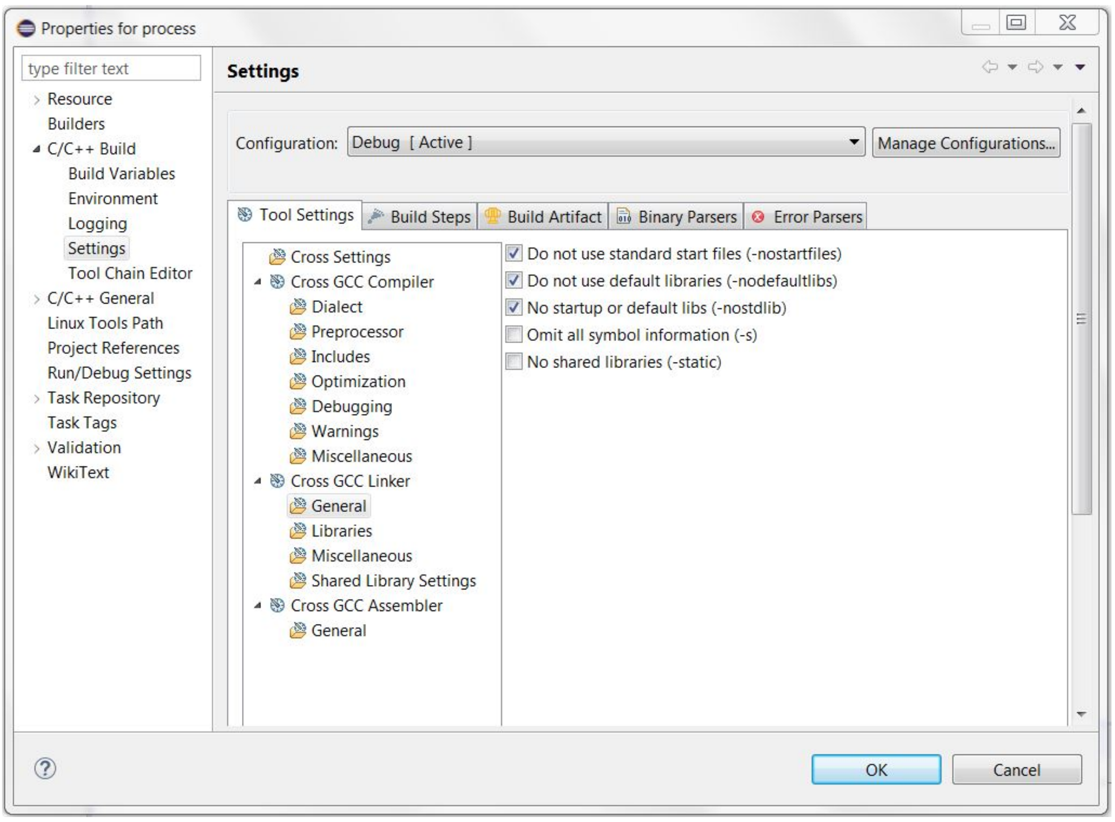
☐ Verbose (-v)

☐ Support ANSI programs (-ansi)

☑ Position Independent Code (-fPIC)

FIGURE 5.22: Configuring Compiler Flags for building Process

9. In the Cross GCC Linker - General Tab configure the options as follows. The -nostdlib, -nodefaultlibs and -nostartfiles flag will exclude the C libraries code or any startup code from the process code. If the process code uses any of the functions of these libraries, these would have to be provided by the kernel through its export mechanism. If a process calls any of functions of these libraries, a relocation would be inserted which would have to be resolved at load time.

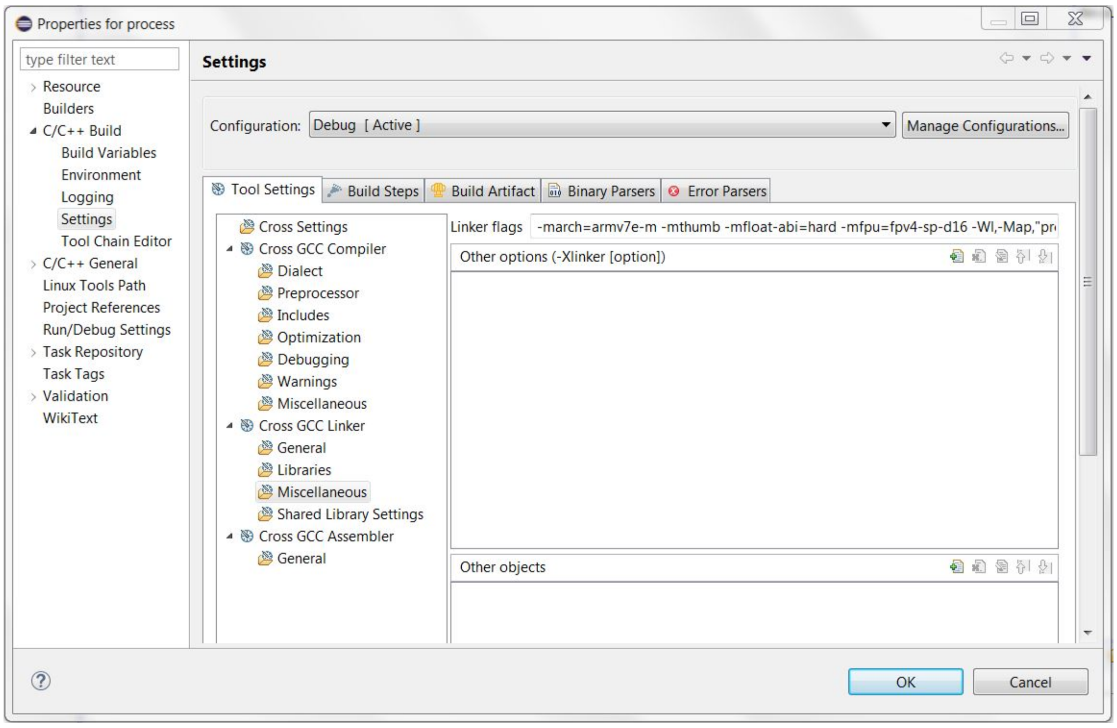


Configuration: Debug [Active]

- [✓] Do not use standard start files (-nostartfiles)
- [✓] Do not use default libraries (-nodefaultlibs)
- [✓] No startup or default libs (-nostdlib)
- [ ] Omit all symbol information (-s)
- [ ] No shared libraries (-static)

FIGURE 5.23: Configuring Linker general settings for building Process

10. In the Cross GCC Linker - Miscellaneous Tab add these linker flags. ”-march=armv7e-m -mthumb -mfloat-abi=hard -mfpu=fpv4-sp-d16 -Wl,-Map,”process.map” -fPIC -Wl,–gc-sections -Wl,-T ”${workspace_loc:/${ProjName}/process.lds}” -Wl,-pie -Wl,–warn-unresolved-symbols -Wl,–no-demangle -Wl,–nmagic”. The -T option will specify the linker description file to use to build the process.



Configuration: Debug [Active]

Linker flags -march=armv7e-m -mthumb -mfloat-abi=hard -mfpu=fpv4-sp-d16 -Wl,-Map,"pr

FIGURE 5.24: Configuring Linker Flags for building Process

After carrying out these steps create a C file containing the implementation of the process and also copy the *process.lds* file in this project from the sample processes we have provided at [23]. Now we can build the project. After building it a file with extension ".proc" will be created in a *Debug* folder. This is the ELF File image of the process.

5.6 Implementation Details

In this section we will explain the details of how we implemented our framework. First we look at the changes we had to do to implement processes and then we look at the changes we did to the FreeRTOS source code to support these processes.

5.6.1 Process Implementation

5.6.1.1 Process Linker Description File

The first step towards implementation of processes is writing its linker description file. This file will define the layout of the process' image. The linker description file of our process, places the start of process image from address 0x0. This can also be called the virtual address. All the code, data and other sections will follow onward from this address. The actual load address would be different, and will be determined by the location of available free memory in the system. The process image would be copied (with some modification) starting from this load address by the loader. Since, the code was compiled as Position Independent Code, it should work from any load address. The *process.lds* file is the Linker Description file we use. This file should reside in the Eclipse project we created for the process. Let us look at some important sections of this file to understand how processes would be built. Note that all these sections would be placed in sequence after one another.

```
ENTRY(main_task);
```

The above directive will tell the linker to place the virtual address of a function main_task() in our process in the *e_entry* field of the ELF File Header of the the output ELF. Control would be transferred to this function when the loader completes loading the process and starts it. The loader will parse the ELF file of the process image and get address of this function. This function has to exist in all processes. For this purpose we have defined a macro in *processes.h* file as FREERTOS_PROCESS. Simply paste this macro in the main C file of the process after including header files. This macro will simply define a function void main_task(void * pvParameters) that would call the main() function of the process. Look at *processes.h* file for details on what this macro does.

```
.text 0x0 :
{
    PROVIDE_HIDDEN (_proc_text_start = .) ;
    *(.text .text*) ;
    *(.glue_7t) ;
    *(.glue_7) ;
    *(.gnu.linkonce.t*)

    /*. = ALIGN(4) ;*/
}

.ARM.extab :
{
    *(.ARM.extab* .gnu.linkonce.armextab.*)
}

.ARM.exidx :
{
    *(.ARM.exidx* .gnu.linkonce.armexidx.*)
    PROVIDE_HIDDEN (_proc_text_end = .) ;
}
```

The first section tells the linker to place the text (code) section at 0x0 virtual address. This section would contain executable code. The next two sections are specific to ARM architecture.

```
.rodata _proc_text_end :
{
    PROVIDE_HIDDEN (_proc_rodata_start = .) ;
    *(.rodata*) ;
}
```

The above section would contain the read-only data of the process.

```
.got.plt :
{
    *(.got.plt)
}
```

The above section would contain the Global Offset Table for the Procedure Link Table of the process. When a process calls a function that is not defined within it, the linker would insert a call to an entry in the Procedure Link Table. That entry would contain code to get the address from an entry in this Global Offset Table. This address would initially be zero and would be populated by the loader at load time. The loader will find the address of the function to be called from the kernel's Symbol Tables and populate this entry.

```
.got :
{
    *(.got)
    PROVIDE_HIDDEN (_proc_rodata_end = .) ;
}
```

The above section would contain the Global Offset Table for the data. When a process would want to access a global data variable that is not a part of the process, the linker would insert an indirection to get the address of that variable from this table. The actual address would be populated at load time by the loader, again by searching the kernel's Symbol Tables.

```
.data _proc_rodata_end :
{
    PROVIDE_HIDDEN (_proc_data_start = .) ;
    *(.data) ;
    *(.data*) ;
    *(.gnu.linkonce.d*)
    PROVIDE_HIDDEN (_proc_data_end = .) ;
}
```

```
.bss _proc_data_end :
{
    PROVIDE_HIDDEN (_proc_bss_start = .) ;
    *(.bss) ;
    *(.bss*) ;
    *(COMMON) ;
    PROVIDE_HIDDEN (_proc_bss_end = .) ;
    PROVIDE_HIDDEN (_proc_load_end = .) ;
}
```

The above two sections will contain the data and the bss sections respectively.

```
proc_end_mark ALIGN(4) :
{
LONG(0xFFFFFFFF);
}
```

We use the above section just as a marker to indicate to the loader, the end of the sections of the process that are needed for execution. The sections that follow after this are only required at the time of loading and are discarded afterwards.

```
.dynsym :
{
    *(.dynsym)
}
```

The above section would contain the symbol table of all the symbols needed for dynamic linking.

```
.dynstr :
{
    *(.dynstr)
}
```

The above section would contain the symbol string table. This table would contain the names of all the symbols (function and data variable names) needed for dynamic linking. The loader would get the names of unresolved symbols used in the process form this table and then search them in the kernel export tables and then perform their relocation.

```
.rel.plt :
```

```
{
    *(.rel.plt)
}
```

The above section would contain the relocation entries for the Procedure Link Table section. ELF specification [14] explains what relocations are and how they can be resolved.

```
.rel.dyn :
{
    *(.rel.dyn)
}
```

The above section would contain the relocation entries for the Global Offset Table section.

5.6.1.2 A Very Basic Process Example

A very basic example of process code would be as follows:

```
#include "FreeRTOS.h"
#include "processes.h"

/* Must always use the FREERTOS_PROCESS macro. */
FREERTOS_PROCESS

int main()
{
   /* Your code here. */

   return 0;
}
```

A process must always contain the above code. There should always be a main() function in which user can enter his code and the FREERTOS_PROCESS macro should always be used. This macro is defined in *processes.h* file as follows:

```
#define FREERTOS_PROCESS                                                        \
                                                                                \
        int main(void);                                                         \
        void main_task( void * pvParameters ) __attribute__ ((noreturn));       \
        void main_task( void * pvParameters )                                   \
        {                                                                       \
            main();                                                    \
                                                                       \
            /* Suspend this task, since main has exited.*/             \
            vTaskSuspend(xTaskGetCurrentTaskHandle());                 \
        }
```

Using the above macro in a process file would define the main_task function that would serve as an entry point for the process. The loader will be able to get the address of this entry point function and will call it when starting the process (actually the loader will make this function to run as a FreeRTOS task). This function then would call the main function of the process.

5.6.2 Process Framework Implementation in FreeRTOS

Now we shall explain the steps taken to modify the FreeRTOS kernel to support processes. We proceed in a step by step fashion to do this.

5.6.2.1 Process ID

Every process that gets loaded in the system would be assigned a unique identifier know as Process ID or PID. The PID value of 0 is reserved for the kernel code. PID values from 1 and onward would be assigned to incoming processes.

5.6.2.2 The Process Control Block

We define a process control block for every executing process in the system. The process control block will store the Process ID, load address, size of memory allocated for process, the stack pointer, the heap pointer, a main task handle and the processes' mode. The process control block is a structure as follows defined in *processes.h* file:

```
typedef struct PROCESS_CONTROL_BLOCK
{
    uint32_t        uPid;
    void*           pvLoadMemAddr;
    void*           pvLoadMemAddrOriginal;
    uint32_t        uLoadSize;
    void*           pvStack;
    void*           pvHeap;
    TaskHandle_t    pxMainTaskHandle;
    uint8_t         uMode;
}ProcessCB;
```

We maintain a global array of process control blocks and assign one to every newly loaded process and free it upon unloading of a process. Note that the FreeRTOS kernel will still schedule tasks instead of processes. A process would just have a number of tasks associated with it. We say that a process has been scheduled when one of its task gets scheduled. We also maintain a global pointer to currently scheduled process' process control block. The scheduler will always keep this value updated with the active process' process control block address.

5.6.2.3 Changes in the FreeRTOS Task Control Block

We add following elements to the Task Control Block tskTCB of FreeRTOS Task defined in *task.c* file.

```
void *pvProcess;
struct tskTaskControlBlock *pxNextTask;
void *pvReturnAddress;
```

The first element points to the Process Control Block to which this task belongs to. Whenever a task is created by a process, this field is populated with the pointer to the current process control block. The second element points to the next task that is associated with the same process this task belongs to. All the tasks associated with a process are maintained in a linked list of pointers to task control blocks. Whenever a process creates a task, the pointer to the task control block of that task is added to this list and whenever a process deletes a task, the pointer to the task control block of that task is removed from the list. The head of this list resides in the task control block of the main task of a process. The last element is used to implement system calls in user-mode processes. We shall explain this third element when we explain the implementation of user-mode processes.

5.6.2.4 Modifications to the Kernel's Linker Description File

As explained earlier, a Linker Description File describes how code, data and other sections would be compiled and linked together to form an executable image. This file is dependent on the underlying processor's memory map and is different for different hardware platforms. Independent of the underlying hardware's memory map, we add four sections in the Linker Description file used to build the FreeRTOS kernel image. These sections deal with the symbols (functions and global data variables) exported by the kernel for use by the processes.

```
procsymstr :
{
    KEEP(*(procsymstr)) ;
} > FLASH
```

The above section will contain the names of the exported symbols as strings one after another. This section will be used only by user-mode processes.

```
procsymtab :
{
    PROVIDE_HIDDEN (_proc_symbol_table_start = .) ;
    KEEP(*(procsymtab)) ;
    PROVIDE_HIDDEN (_proc_symbol_table_end = .) ;
} > FLASH
```

The above section will contain the symbol table entries of the exported symbols. We will explain these symbol table entries in coming subsection. This section will be used only by user-mode processes.

```
kprocsymstr :
{
    KEEP(*(kprocsymstr)) ;
} > FLASH
```

The above section will contain the names of the exported symbols as strings one after another. This section will be used only by kernel-mode processes.

```
kprocsymtab :
{
    PROVIDE_HIDDEN (_kproc_symbol_table_start = .) ;
    KEEP(*(kprocsymtab)) ;
    PROVIDE_HIDDEN (_kproc_symbol_table_end = .) ;
} > FLASH
```

The above section will contain the symbol table entries of the exported symbols. We will explain these symbol table entries in coming subsection. This section will be used only by kernel-mode processes.

5.6.2.5 Symbol Table

If a process needs to call a function or access a global data variable, its address would not be available to it. The process would be built with relocations for these accesses which would be resolved at load time by the loader. The loader would get the names of the symbol to be resolved from relocation information present in the process' image and then search for the addresses of these symbols in a table present in the kernel's image. This table would be known as the Symbol Table. The symbol table would contain one entry for each exported symbol. The symbol would have to be exported from kernel's code using special macros we have defined for this purpose. This mechanism of exports and symbol table is exactly similar to that of Linux. Each symbol table entry would be a structure as follows, defined in *processes.h* file:

```
typedef struct SYM_ENTRY_STRUCT
{
    const char*    pcName;
    void*          pvAddress;
} SymbolEntry;
```

The first element of this structure will point to the string containing the name of an exported symbol. The second element will point to the address of that symbol in the kernel's image. There would be two symbol tables, one available for user-mode processes and the other one for kernel-mode processes. These tables would reside in the sections *procsymtab* and *kprocsymtab* of the kernel's image respectively. The placement of these sections was shown earlier in the linker description file. The string names of the exported symbols for user-mode and kernel-mode processes will reside in the *procsymstr* and *kprocsymstr* sections of the kernel's image respectively. The symbols would be exported by using the following macros in the *processes.h* file.

```
#define EXPORT_KERNEL_SYMBOL(symbol)                                   \
                                                                       \
/* Put string name of the symbol into a string table */                \
static const char __kstr_##symbol[]                                    \
    __attribute__((section("kprocsymstr"), aligned(1))) = #symbol;     \
                                                                       \
/* Put symbol entry into symbol table */                               \
static const SymbolEntry __ksym_##symbol                               \
    __attribute__((section("kprocsymtab"), used)) =                    \
        {__kstr_##symbol, (void *)&(symbol)}
```

The above macro will export a symbol for use by a kernel-mode process. At first the macro declares a character string containing the name of the symbol. The string is placed in the *kprocsymstr* section. All the string names of the exported symbols for kernel-mode processes will go in this section. After this, the macro will populate a symbol table entry containing the address of the name of the string and the address of the symbol. This mechanism is very similar to the export mechanism used by Linux. To export symbols for user-mode processes following macro would be used.

```
#define EXPORT_USER_SYMBOL(symbol)                                     \
                                                                       \
/* Put string name of the symbol into a string table */                \
static const char __str_##symbol[]                                     \
    __attribute__((section("procsymstr"), aligned(1))) = #symbol;      \
                                                                       \
/* Wrap original function in syscall code. */                          \
portSYSCALL_ENTER(symbol)                                              \
                                                                       \
/* Put symbol entry into symbol table */                               \
static const SymbolEntry __sym_##symbol                                \
    __attribute__((section("procsymtab"), used)) =                     \
        {__str_##symbol, (void *)&(__syscall_##symbol)}
```

The above macro used to export symbols for a user-mode process is a bit tricky one and has some differences. First is that it places the symbol strings and the symbol table entry into *procsymstr* and *procsymtab* sections respectively. The second difference is that this macro would first define a new version of the exported symbol using the portSYSCALL_ENTER(symbol) macro and export the address of this symbol instead. The portSYSCALL_ENTER(symbol) macro is defined by us in the architecture specific file *portmacro.h* file. We know that the user-mode processes are unprivileged processes. The portSYSCALL_ENTER(symbol) macros wraps a function exported by the kernel in a wrapper that invokes the software interrupt mechanism to first transfer control to the kernel and elevate the privilege level. Then we export the address of this newly created function into the symbol table. We will explore this in more detail when we explain the implementation of user and kernel mode for processes. Finally, to export data variables we use the EXPORT_DATA_SYMBOL(symbol) macro. For now data variables can only be exported to kernel-mode processes. Following is an example of how to use these macros:

```
void vUserFunction(void);
void vKernelFunction(void);

uint32_t uKernelData = 0xdeadbeef;

EXPORT_DATA_SYMBOL(uKernelData);
EXPORT_KERNEL_SYMBOL(vKernelFunction);
EXPORT_USER_SYMBOL(vUserFunction);

void vUserFunction(void)
{
    /* Do something unprivileged. */
}

void vKernelFunction(void)
{
    /* Do something privileged or unprivileged. */
}
```

Note that these macros can only be used in non-process code i.e code that will become the part of kernel's image. Exporting from processes is not supported yet.

5.6.2.6 Process Loader

The process loader code would be the heart of our processes framework. A process' ELF file can reside in the RAM, ROM or any external available storage. In our case we keep the processes' ELF file as an array in the internal flash of the micro-controller. From there it can be loaded into the RAM by the process loader. The process loader code would first parse the process' ELF file for required sections, copy them into memory allocated to run the process, perform relocations and give control to the main_task function of the process. The process loader code resides in the *processes.c* file.

5.6.2.7 Implementation of User-Mode and Kernel-Mode Processes

Processors and micro-controllers usually provide mechanisms for controlling software privileges. The Cortex-M4 architecture provides Thread-Mode Privilege Level bit in its Control register for this purpose. This bit can only be controlled by privileged software. When the scheduler sees that it has to run a task that was part of the user-mode process, it can set the mode of execution as unprivileged and give control to it. Similarly, when a kernel-mode task has to run the scheduler can set the mode of execution as privileged. The scheduler can control this bit as it always executes in privileged mode. Now what happens if the user-mode process wants to call a kernel API that wants to access privileged resources? This can be achieved by implementation of a system call mechanism. A user-mode function calling a kernel API would get a special exported version of the kernel API as explained earlier. This special exported version of the API contains a wrapper code that invokes the software interrupt mechanism to first transfer control to the kernel and elevate the privilege level. This was achieved using the portSYSCALL_ENTER(symbol) macro inside the EXPORT_USER_SYMBOL(symbol) macro. After elevating the privilege level, the kernel will call the actual kernel API and then return to another function vSysCallReturn. This function would switch the privilege level back to unprivileged mode and then return to the position in the user-mode code from where the actual kernel API was called. The vSysCallReturn function is defined by us in the architecture specific file *port.c*.

5.6.2.8 Implementation of Memory Protection among Processes

Every process will be loaded in one continuous section of memory region allocated for it. By using a Memory Protection Unit we can protect the processes from accessing each other memory regions. If a process does access a memory region of any other process, memory access fault would be generated by the memory protection unit. Control will go to a special handler whose implementation is left up to the user. The user can decide on what actions to take further actions e.g. terminating the process that caused the exception. Whenever a task is to be scheduled, the scheduler will first find out to what process it belongs to, then the scheduler will program its memory regions in the Memory

Protection Unit.

5.6.3 Process Framework Configuration Options

FreeRTOS provides a *FreeRTOSConfig.h* file to configure its capabilities and functionalities depending on hardware and application requirements or to conserve code and data memory. The file contains different macros to configure the kernel. In order to support processes, the configuration macro configSUPPORT_STATIC_ALLOCATION should be defined and set to 1 as our processes framework requires support for static memory allocation. In addition to this, we have introduced the following macros to configure processes support according to the needs.

5.6.3.1 configSUPPORT_PROCESSES

Setting this macro equal to 1 will enable support for processes in the code. Setting it to 0 will disable it. Default value is 1.

5.6.3.2 configMEM_PROTECTED_PROCESSES

Setting this macro to 1 will enable memory protection among the processes in the code. Setting it to 0 will disable memory protection support. Default value is 1.

5.6.3.3 configUSER_PROCESS_SUPPORT

Setting this macro to 1 will enable the processes to be loaded in user-mode or kernel mode based on a parameter passed to the process loading API. Setting it to 0 will make all the processes to load in kernel mode. Default value is 1.

5.6.3.4 configPAGE_SIZE

Usually MMUs and MPUs only allow the programming of memory attributes of memory areas of fixed sizes. One such memory area can be called a page. The memory region of a process can span such multiple pages. When we program the MPU or MMU for a process, we set attributes for all of its pages. This macro configures the size of that page and its value depends on the underlying MMU or MPU.

5.6.3.5 configMAX_PROCESSES

Set this to the number of maximum process that can be loaded in a system. Increasing this option would increase memory consumption. Default value is 4.

5.6.4 Process Framework APIs

In this section, we present our Process Framework APIs a developer can use to run processes and configure their capabilities respectively. The Appendix A also shows a sample process and a sample application to load this process.

5.6.4.1 Initializing FreeRTOS Process Framework

The first step before using the processes framework is to initialize it by calling the following API:

```
void vInitProcessComponent()
```

This API will setup the data structures and create the loader task for the processes component. Only after this API is called, the user can begin to load and unload processes. This function should be called from the main() function before the FreeRTOS scheduler is started.

5.6.4.2 Loading a Process

To load a process into memory use the following API:

```
BaseType_t xLoad(void* pvElfFile, uint32_t uStackSize, uint32_t uHeapSize,
                 uint8_t uMode, uint16_t* puPid, UBaseType_t uxPriority)
```

where,

```
void* pvElfFile - Pointer to memory location of the process' elf file.
uint32_t uStackSize - Stack size of the main task of process.
uint32_t uHeapSize - Heap size of the process' heap.
uint8_t uMode - 1 = User Mode, 0 = Kernel Mode.
uint16_t* puPid - Pointer to store Process ID.
UBaseType_t uxPriority - Priority of the main task of process.
```

This API will return pdPASS if loading succeeded and pdFAIL if it failed. pdPASS and pdFAIL are defined in *projdefs.h* file.

5.6.4.3 Unloading a Process

To remove a process from the system, use the following API

```
BaseType_t xUnload(uint16_t uPid)
```

where,

```
uint16_t uPid - Process ID of the process to unload.
```

This API will return pdPASS if loading succeeded and pdFAIL if it failed. pdPASS and pdFAIL are defined in *projdefs.h* file.

5.6.4.4 Getting Process ID

To get the Process ID of the currently executing process use the following API:

```
uint32_t uxGetProcessId(void)
```

This API will return the Process ID of the currently executing process.

5.6.4.5 Getting Process' Heap Area

A process can request a pointer to the heap area allocated for the process at load time. It is up to the process code to manage the heap area as currently we have not implemented any heap allocation algorithm for processes. To get the address of the heap allocated for a process use the following API:

```
void vGetProcessHeapAddress(void** heap, uint32_t* size)
```

where,

```
void** heap - Pointer to store address of the heap.
uint32_t* size - Pointer to store the size of the heap.
```

5.7 Testing and Results

To test the kernel's reliability after implementation of our Process' Framework, we performed following testing. We had completely achieved our objectives of Dynamic Loading, Privilege Levels and Memory Protection support.

5.7.1 Testing of Loading and Unloading of Processes

A number of processes were loaded and unloaded to test this support. Process images in the form of arrays were made part of the kernel's image and were burnt in the Tiva Microcontroller's flash. From there they were loaded into the RAM of the micro-controller. Since the memory available on the controller was limited, the number of processes that could simultaneously be running was also limited. Even then we have tested at least 4 simultaneous processes running at the same time. Also we have tested the loading and unloading of a number of processes one by one at a time. Some sample processes and sample demos to load and unload processes are provided at [23] which we had used.

5.7.2 Testing of Kernel-Mode and User Mode Privilege Levels

To test this support we made a user-mode process execute a privileged instruction to read the SysTick Control and Status Register of the Cortex-M4 Architecture. This register is the control register for the operating systems timer. Such access should never be allowed to a user-mode process. Since the micro-controller was not in a privileged mode, executing this instruction caused the micro-controller to generate a Hard Fault Exception. If the same instruction was executed from a kernel-mode process, the instruction executed successfully. The instruction was:

```
uint32_t temp = *((volatile uint32_t *)(0xE000E000));
```

5.7.3 Testing of Memory Protection

To test memory protection we made a user-mode process to access a memory area that did not reside in any of its memory regions. For example we tried to modify the starting address of RAM at 0x20000000. We are sure that this address will belong to the kernel's image since it is just the start of the RAM and our process would have been loaded at a higher address. The MPU caught this and generated an exception.

5.7.4 Overheads of the Process Framework

The process framework has brought some overheads to the FreeRTOS solution. The overheads we have found are not severe and reasonable trade-offs between functionality and performance can be done by configuring the system.

5.7.4.1 Code and Data Size Overheads

The symbol table in the kernel would take extra space in the kernel image. The size of the symbol table would depend on the number of function and data symbols exported by kernel. For every exported symbol memory would be used to store its string name and a symbol table entry. So a general guideline would be to keep the names of function and data symbols to be exported small. The process framework code would have its code and data size overhead too but special care has been taken to keep it as low as possible. The processes code would also have larger sizes because of the PLT and GOT code in it and would depend on how many external symbols a process accesses.

5.7.4.2 Processes Speed Overheads

In a process the PLT and GOT would add one extra level of indirection for external function calls or external data accesses. This would have an affect on the speed of execution of code. However, if a process calls no external functions or accesses external data symbols, then there would be no such overhead.

5.7.4.3 User-Mode Processes Overheads

If a user mode process calls a kernel API, a software interrupt would be executed first to transfer control to the kernel. As a result of this interrupt, context would have to be saved and then restored too. Also the wrappers written for implementing the system call would add a few more instructions to the API function call.

5.7.4.4 Memory Protection Overheads

Whenever the scheduler will give control to a task, it will re-program the memory regions of the process this task belongs to in the MPU or the MMU. So the code of reprogramming the MPU or the MMU will have an impact on the speed. Also, as we explained earlier, usually the MMUs and MPUs only allow the programming of memory attributes of memory areas of fixed sizes. One such memory area can be called a page. The memory region of a process can span such multiple pages. When we program the MPU or MMU for a process, we set attributes for all of its pages. This places a requirement that the entire memory of the process should be a multiple of the page size. This is a memory overhead.

Chapter 6

Conclusions and Future Directions

6.1 Conclusions

By implementing a process framework in this research, we have brought following key benefits to the Free RTOS Solution.

1. Dynamic Loading and Unloading of processes for memory reuse.
2. In-field software updating.
3. Memory protection among address spaces of various processes.
4. Separation of processes into User Mode and Kernel Mode processes.
5. Improved reliability of the kernel.

These features have been verified by testing as we had explain in the previous chapter.

6.2 Future Work

6.2.1 Support on Symmetric Multiprocessing Systems (SMP)

Very comprehensive and practical work has been done to develop Symmetric Multiprocessing versions of FreeRTOS by [24] and [25]. Their works can be used to obtain a SMP version of FreeRTOS and our framework can be extended to support SMP.

6.2.2 Support to Export Symbols from Processes

For now, we can only export symbols to processes from the kernel's image. It would be very appealing if support can be added in which processes can export symbols to other processes too. This concept would be much similar to the concept of shared libraries in Linux.

6.2.3 Improved Algorithms for Symbol Searching

When the loader needs to resolve a relocation, it looks for the symbol name in the symbol table one by one in a linear fashion. This can take time if the symbol table becomes very large. A separate tool or script could be written to parse the ELF image of the kernel and sort the symbols in the symbol table. Hash or Binary tree algorithms are perfect candidates for this approach.

6.2.4 Support for Newer Architectures

We have implemented our support for ARM's Cortex-M4 architecture. A number of other architectures such as MIPS, PowerPC and other variants of ARM architecture are widely used in the embedded world. Adding their support would make our solution much more attractive.

6.2.5 Support for 64 Bit Architectures

64 Bit micro-processors and micro-controllers are now available in the market. Current implementation of the process framework only deals with 32-Bit architectures. The support only deals with 32 Bit process ELF images. Support for 64 Bit process ELF images would need to be added. The file parsing mechanisms would need to be updated and methods of performing relocation would also be needed.

Appendix A

Code

A.1 A Sample Process

The following code is an example of a process. This process will create one task. This task would get the Process ID of the current process and print it continuously after every one second.

```
#include "FreeRTOS.h"
#include "task.h"
#include "processes.h"

static StackType_t stack[100];
static TaskHandle_t taskhandle;
static StaticTask_t tasktcb;

extern void serial_write (char* pString);
extern uint32_t uxGetProcessId (void);

FREERTOS_PROCESS

void vTask(void * pvParameters)
{
    while(1)
    {
        /* Get the process id and print it. */
        uint32_t pid = uxGetProcessId() + 48;
        serial_write("This is a task of Process ");
        serial_write(&pid);
        serial_write("\n\r");
        vTaskDelay(100);
    }
}

int main()
{
   serial_write("Creating Tasks \n\r");

   /* Create a task. */
   taskhandle = xTaskCreateStatic(vTask, "task", 100, 0, 0, stack, &tasktcb);

   return 0;
}
```

A.2 Sample Application to Load a Process

The following code will load and run a process. When we build the process project in Eclipse IDE, we get a file with ".proc" extension. This file is the ELF file image of the process. This binary file can be converted into an array, whose address could be passed to the load API to load and run the process. We have made an array of type "static const unsigned char process_proc[]" and placed it into the *process.proc.h* header file included in this code. This is just one way of loading a process. The process could also have been placed in an external memory such as SPI based Flash and loaded. Note that to build a FreeRTOS application such as follows we may have to add some hooks as the FreeRTOS documentation indicates. They have not been shown here for the sake of simplicity.

```
#include "FreeRTOS.h"
#include "task.h"
#include <processes.h>
#include "string.h"
#include "process.proc.h"

extern void platform_init(void);
void mytask( void * pvParameters );

#define STACKSIZE 1024
#define HEAPSIZE   128

int main(void) {

    TaskHandle_t taskhandle;
    /* Call platform init routine specific to Tiva controller. */
    platform_init();

    /* Init process component. */
    vInitProcessComponent();

    /* Create application task. */
    xTaskCreate(mytask,
    "mytask",
    1024,
    0,
    configMAX_PRIORITIES  - 10,
    &taskhandle);

    /* Start the scheduler. This should not return. */
    vTaskStartScheduler();

    /* In case the scheduler returns for some reason,
        print an error and loop forever. */
    while(1)
    {
    }

    return 0;
}

void mytask( void * pvParameters )
{
    uint16_t pid;
    while(1)
    {
        /* Load a process in supervisor (kernel) mode. */
        xLoad((void*)process_proc, STACKSIZE, HEAPSIZE, processSUPERVISOR,
                                     &pid, configMAX_PRIORITIES  - 12);

        vTaskDelay(5000);

        /* Unload the process. */
        xUnload(pid);

        vTaskDelay(5000);
    }
}
```

References

[1] P. B. Galvin, G. Gagne, and A. Silberschatz. *Operating System Concepts*. John Wiley and Sons, 9th edition, 2013.

[2] Simon Holmbacka, Wictor Lund, Sébastien Lafond, and Johan Lilius. Lightweight framework for runtime updating of C-based software in embedded systems. In *HotSWUp*, 2013.

[3] Shimpei Yamada and Yukikazu Nakamoto. Protection mechanism in privileged memory space for embedded systems, real-time os. In *Distributed Computing Systems Workshops (ICDCSW), 2014 IEEE 34th International Conference on*, pages 161–166. IEEE, 2014.

[4] Texas Instruments. ARM Cortex-M4F Based MCU TM4C123G LaunchPad Evaluation Kit. `http://www.ti.com/tool/ek-tm4c123gxl`, Last accessed on April 20, 2017.

[5] Mentor. Nucleus RTOS. `https://www.mentor.com/embedded-software/nucleus`, Last accessed on January 1, 2017.

[6] Express Logic. THREADX. `http://rtos.com/products/threadx/`, Last accessed on January 1, 2017.

[7] Wind River. VxWORKS. `https://www.windriver.com/products/vxworks/`, Last accessed on January 1, 2017.

[8] BlackBerry. QNX. `http://www.qnx.com/content/qnx/en.html`, Last accessed on January 1, 2017.

[9] TRON. T-Kernel. `http://www.tron.org/tron-project/what-is-t-kernel/t-kernel/`, Last accessed on January 1, 2017.

[10] eSOL. Extended T-Kernel RTOS. `http://www.esol.com/embedded/et-kernel.html`, Last accessed on January 1, 2017.

[11] MB David, DW Brian, and RC Ian. The inside story on shared libraries and dynamic loading. *Computing in Science and Engineering, Scientific Programming Journal*, 2001.

[12] J.R. Levine. *Linkers & Loaders.* Morgan Kauffman, 2000.

[13] Michael Lee Scott. *Programming language pragmatics.* Morgan Kaufmann, 2000.

[14] TIS Committee et al. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, May 1995, 2011.

[15] Shao-Ju Wang, Wei Xu, and Xiao-yun Zheng. Research on the technique of module dynamic loading for satellite software. In *Instrumentation, Measurement, Computer, Communication and Control (IMCCC), 2013 Third International Conference on*, pages 813–816. IEEE, 2013.

[16] Wictor Lund. A unified run-time updating and task migration mechanism. Master's thesis, Abo Akademi University, 2012.

[17] Wictor Lund. Run-time dynamic linking for FreeRTOS. https://github.com/ESLab/rtdl, Last accessed on January 1, 2017.

[18] Shubhendu Sinha, Martijn Koedam, Rob Van Wijk, Andrew Nelson, Ashkan Beyranvand Nejad, Marc Geilen, and Kees Goossens. Composable and predictable dynamic loading for time-critical partitioned systems. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pages 285–292. IEEE, 2014.

[19] Shubhendu Sinha, Martijn Koedam, Gabriela Breaban, Andrew Nelson, Ashkan Beyranvand Nejad, Marc Geilen, and Kees Goossens. Composable and predictable dynamic loading for time-critical partitioned systems on multiprocessor architectures. *Microprocessors and Microsystems*, 39(8):1087–1107, 2015.

[20] Nermin Kajtazovic, Christopher Preschern, and Christian Kreiner. A component-based dynamic link support for safety-critical embedded systems. In *Engineering of Computer Based Systems (ECBS), 2013 20th IEEE International Conference and Workshops on the*, pages 92–99. IEEE, 2013.

[21] Vikas Chandra and Robert Aitken. Impact of technology and voltage scaling on the soft error susceptibility in nanoscale CMOS. In *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on*, pages 114–122. IEEE, 2008.

[22] ARM. GNU ARM Embedded Toolchain. https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads, Last accessed on April 20, 2017.

[23] Mughees Chohan. FreeRTOS Process Model. https://github.com/MugheesChohan/FreeRTOS-Process-Model, Last accessed on April 20, 2017.

[24] James Mistry. *FreeRTOS and Multicore.* PhD thesis, University of York, 2011.

[25] Prakash Chandrasekaran, Shibu Kumar KB, Remish L Minz, Deepak D'Souza, and Lomesh Meshram. A multi-core version of FreeRTOS verified for datarace and deadlock freedom. In *Formal Methods and Models for Codesign (MEMOCODE), 2014 Twelfth ACM/IEEE International Conference on*, pages 62–71. IEEE, 2014.