

ECE 350

Laboratory Project Manual for

Real-Time Operating Systems

by

Yiqing Huang
Seyed Majid Zahedi

Electrical and Computer Engineering Department
University of Waterloo

Waterloo, Ontario, Canada, April 3, 2021

© Y. Huang and S.M. Zahedi 2020 - 2021

Contents

List of Tables	vi
List of Figures	viii
Preface	1
I Lab Administration	1
II Lab Project	8
1 Introduction	9
1.1 Overview	9
1.2 Summary of RTX Requirements	9
1.2.1 RTX Primitives and Services	9
1.2.2 RTX Tasks	10
1.2.3 RTX Footprint and Processor Loading	11
1.2.4 Error Detection and Recovery	11
2 Lab1: Introduction to Kernel Programming and Memory Management	12
2.1 Objective	12
2.2 Starter Files	12
2.3 Pre-lab Preparation	13
2.4 Assignment	13
2.4.1 Programming Project	14
2.4.2 Report	17
2.4.3 Third-party Testing and Source Code File Organization	18

2.5	Deliverable	19
2.5.1	Pre-Lab Deliverables	19
2.5.2	Post-Lab Deliverables	19
2.6	Marking Rubric	19
2.7	Errata	20
3	Lab2 Task Management	21
3.1	Objective	21
3.2	Starter Files	21
3.3	Pre-lab Preparation	22
3.4	Assignment	22
3.4.1	Programming Project	22
3.4.2	Report	28
3.4.3	Third-party Testing and Source Code File Organization	29
3.5	Deliverables	30
3.5.1	Post-Lab Deliverables	30
3.6	Marking Rubric	30
3.7	Errata	30
4	Lab3 Inter-task Communications and UART Interrupts	32
4.1	Starter Files	32
4.2	Pre-lab Preparation	33
4.3	Assignment	33
4.3.1	Programming Project	33
4.3.2	Report	38
4.3.3	Third-party Testing and Source Code File Organization	38
4.4	Deliverables	39
4.4.1	Post-Lab Deliverables	39
4.5	Marking Rubric	40
4.6	Errata	40
5	Lab4 Real-time Tasks	41
5.1	Starter Files	41
5.2	Pre-lab Preparation	42

5.3	Assignment	43
5.3.1	Overview	43
5.3.2	Programming Project	43
5.3.3	Report	48
5.3.4	Third-party Testing and Source Code File Organization	48
5.4	Deliverables	50
5.4.1	Post-Lab Deliverables	50
5.5	Marking Rubric	50
5.6	Errata	50
III	Frequently Asked Questions	52
IV	Computing and Software Development Environment Quick Reference Guide	53
6	Windows 10 Remote Desktop	54
7	Software Development Environment	55
7.1	Getting Started with ARM DS	55
7.2	Creating a ECE350 Workspace Structure	55
7.3	Getting Starter Code from the GitHub	56
7.4	Start the ARM DS	56
7.5	Create a New Empty C Project	57
7.6	Import Source Code	59
7.7	Setup Build Properties	61
7.8	Build Project	65
7.9	Create a Debug Connection of VE_Cortex_A9x1	66
7.10	Import Project and Switch between Devices	70
7.10.1	Build for VE_Cortex_A9x1	71
7.10.2	Build for DE1-SoC	72
7.11	Create a Debug Connection for DE1 SoC Board	73
7.12	Debug the Target	75
7.12.1	Debugging with VE_Cortex_A9x1	75

7.12.2	Debugging with DE1 SoC	77
7.12.3	Troubleshooting the DE1 SoC Board	79
7.13	Import an ARM DS Project	81
7.14	Errata	81
8	Programming Cortex-A9	82
8.1	The ARM Instruction Set Architecture	82
8.2	ARM Architecture Procedure Call Standard (AAPCS)	82
8.3	Cortex Microcontroller Software Interface Standard (CMSIS)	84
8.3.1	CMSIS files	85
8.4	Accessing C Symbols from Assembly	86
8.5	SVC Programming: Writing an RTX API Function	88
A	Forms	91
	References	93

List of Tables

0.1	Project Deliverable Weight and Deadlines. Replace the “id” in “Gid” with the two digit group ID number.	3
0.2	Group Project contribution factor table. Each student’s lab grade is their group project grade multiplied by the CF (Contribution Factor). .	4
2.1	Lab1 Marking Rubric	20
3.1	Lab2 Marking Rubric	31
4.1	Lab3 Marking Rubric	40
5.1	Lab3 Marking Rubric	50
8.1	Assembler instruction examples	83
8.2	Core Registers and AAPCS Usage	84

List of Figures

4.1	Structure of a message buffer	35
7.1	ARM DS IDE: Create a new project	57
7.2	ARM DS IDE: Select a wizard	57
7.3	ARM DS IDE: Select a C project	58
7.4	ARM DS IDE: Project explorer	58
7.5	ARM DS IDE: Import Files	59
7.6	ARM DS IDE: Import select wizard	59
7.7	ARM DS IDE: Import file system	60
7.8	ARM DS IDE: MySVC project with template files	60
7.9	ARM DS IDE: MySVC project C/C++ Parallel Build	61
7.10	ARM DS IDE: MySVC project C/C++ Build Target Setting	61
7.11	ARM DS IDE: MySVC project C/C++ Build Compiler Preprocessor Setting	62
7.12	ARM DS IDE: MySVC project C/C++ Build Compiler Include Selection	62
7.13	ARM DS IDE: MySVC project C/C++ Build Compiler Include Folder Selection	63
7.14	ARM DS IDE: MySVC project C/C++ Build Compiler Language C99 .	63
7.15	ARM DS IDE: MySVC project C/C++ Build Linker Image Layout . . .	64
7.16	ARM DS IDE: Build Project	65
7.17	ARM DS IDE: Built executable in Debug folder	65
7.18	ARM DS IDE: Create a new model connection	66
7.19	ARM DS IDE: Debug Connection Configuration	66
7.20	ARM DS IDE: Debug Connection Target Selection	66
7.21	ARM DS IDE: Debug Connection Target VE_Cortex_A9x1 Selection . .	67
7.22	ARM DS IDE: Debug Connection Target VE_Cortex_A9x1 Selection . .	67
7.23	ARM DS IDE: Debug Connection Select Target to Download	68

7.24	ARM DS IDE: Debug Connection Files Tab Setting Error	68
7.25	ARM DS IDE: Debug Connection Files Tab Setting Error	69
7.26	ARM DS IDE: Open Project from File System	70
7.27	ARM DS IDE: Import Project from Folder	70
7.28	ARM DS IDE: Exclude Other Driver Files	71
7.29	ARM DS IDE: Exclude or Include Driver Files	72
7.30	ARM DS IDE: Create New Hardware Connection	73
7.31	ARM DS IDE: Select Cyclone V (Dual Core) for DE1 SoC	73
7.32	ARM DS IDE: Select Target for DE1 SoC	74
7.33	ARM DS IDE: Select Connection for DE1 SoC	74
7.34	ARM DS IDE: Debug Start	75
7.35	ARM DS IDE: Debug Output	76
7.36	ARM DS IDE: PuTTY Setup	77
7.37	ARM DS IDE: PuTTY Outputs	78
7.38	ARM DS IDE: Target Errors	79
7.39	ARM DS IDE: Run Preloader	80
8.1	Role of CMSIS	85
8.2	CMSIS Organization	86
8.3	CMSIS Organization	87
8.4	SVC as a Gateway for OS Functions [6]	88

Preface

Who Should Read This Lab Manual?

This lab manual is written for students who will design and implement a small Real-Time Executive (RTX) for Intel DE1-SoC board populated with a Cyclone V SoC chip, which has a dual-core ARM Cortex-A9 Hard Processor System (HPS) and Altera FPGA.

What is in This Lab Manual?

The first purpose of this document is to provide the descriptions and notes for the laboratory project. The second purpose of this document is a quick reference guide of the relevant development tools for completing laboratory projects. This manual is divided into three parts.

Part I describes the lab administration policies.

Part II is the project description. We break the project into the following five laboratory projects.

- P1: Introduction to Kernel Programming and Memory Management
- P2: Task Management
- P3: Inter-task Communications and Console I/O
- P4: Timing Service and Real-Time Scheduling
- P5: TBD

Part III is frequently asked questions.

Part IV introduces the computing environment and the development tools. It includes a DE1-SoC hardware and software reference guide. The topics are as follows.

- Windows 10 Remote Desktop
- Software Development Environment

- DE1-SoC Hardware Environment
- Programming DE1-SoC

Acknowledgements

Our project is inspired by the original ECE354 RTX course project created by Professor Paul Dasiewicz. Professor Dasiewicz provided detailed notes and sample code to us. We sincerely thank the following generous donations, without which the lab will not be possible:

- ARM University Program for providing us with lab teaching materials and ARM DS gold edition software licenses.
- Intel University Program for providing us with 50 DE1-SoC FPGA boards.
- TerasIC, the manufacture, for shipping the boards in a timely manner.
- Imperas Software for providing us one evaluation license to experiment with their software tools during the lab development.

We gratefully thank our graduate teaching assistants: Zehan Gao, Ali H. A. Abyaneh, Weitian Xing, and Maizi Liao for their help in developing important parts of the lab and the lab manual. Our gratitude also goes out to Eric Praetzel for his continuous strong support of the IT infrastructure of RTOS lab hardware and the ARM DS software, Rasoul Keshavarzi-Valdani for lending us the a DE1-SoC board to experiment with during the initial board selection phase of the lab development. Kim Pope and Reinier Torres Labrada both provided helpful FPGA tips and we gratefully acknowledge their expertise and help.

Finally we own many thanks to our students who did ECE354, SE350 and ECE350 course projects in the past and provided constructive feedback. The lab projects won't exist without our students.

Part I

Lab Administration

Lab Administration Policy

Group Lab Policy

- **Group Size.** All labs are done in groups of *four*. A group size of less than four is not recommended. There is no reduction in project deliverables regardless the size of the project group. The Learn system (<http://learn.uwaterloo.ca>) is used to sign up for groups. The lab group sign-up deadline is in Table 0.1). Late group sign-up is not accepted and will result in losing the entire lab sign-up mark, which is 2% of the total lab project grade. Grace days do not apply to Group Sign-up. Any student without a lab group after the sign-up deadline will be randomly assigned to a lab group by the lab teaching staff.
- **Group Project Manager.** The group elects one member as the group project manager. The project manager can be the same person for all deliverables or a different person for a different deliverable. Rotating project manager's role gives each group member an opportunity to practice group project management. However this role rotation is a choice rather than requirement. It is up to the group to decide. You need to submit the group information in .csv file every time there is an update of the project manager or group membership. A `group.csv` template file can be found at <https://github.com/yqh/ece350> under the submission sub-directory.
- **Quitting from a Group.** If you notice workload imbalance, try to solve it as soon as possible within your group. Quitting from the group should be used as the last resort. Group quitting is only allowed once. You are allowed to join another group which has three or less number of students. You are not allowed to quit from the newly formed group again. There is *one grace day deduction penalty* to be applied to each member in the old group. We highly recommend everyone to stay with your group members as much as possible, for the ability to do team work will be an important skill in your future career. Please choose your lab partners carefully and wisely. The code and documentation completed before the group split-up are the intellectual property of each students in the old group.
- **Group Quitting Deadline.** To quit from your group, you need to notify the lab instructor in writing and sign the group split-up form (see the Appendix A) at

Deliverable	Weight	Due Date	File Name
P0 Group Sign-up	2%	16:30 EST Jan 14	group.csv
P1 Memory Management	18%	16:30 EST Jan 28	p1_Gid.zip
P2 Task Management	20%	16:30 EST Feb 11	p2_Gid.zip
P3 Inter-task Communications and Console I/O	25%	16:30 EST Mar 11	p3_Gid.zip
P4 Timing and Real-Time Scheduling	20%	16:30 EST Mar 25	p4_Gid.zip
P5 TBD	15%	16:30 EST Apr 08	p5_Gid.zip

Table 0.1: Project Deliverable Weight and Deadlines. Replace the “id” in “Gid” with the two digit group ID number.

least one week before the nearest lab project deadline.

Project Submission Policy.

- **Project Deliverables.** The lab project is divided into five deliverables. For each deliverable, there is a pre-lab deliverable and a post-lab deliverable. Students are required to finish the pre-lab deliverable before attempting the lab assignments. For the terms we have scheduled lab sessions, pre-lab is due by the time your scheduled lab session starts. For the terms we do not have scheduled lab sessions, pre-lab is due by the deadline of the previous lab’s post-lab.

Each post-lab deliverable includes the source code, a lab report (in pdf) file and the `group.csv`¹. Create a directory and name it “labN”, where N is 1, 2, ..., 5. Create a sub-directory named “code”. Put your ARM DS application folder under the code directory. Name your lab report file “pN_report.pdf”, where N is 1, 2, 3, 4, 5 and put it under labN directory. Include a README file with group identification, project manager name and a description of directory contents. Put the README file under labN directory. Archive all files for each deliverable in a single file and submit it to the corresponding Learn Dropbox. Table 0.1 gives the weight, deadline and naming convention of each post-lab deliverable.

For each deliverable, we will conduct an anonymous peer review within a group. A student will rate how satisfied he/she is with every other group member’s contribution from 0 to 10, where the higher the rating, the more satisfied the student feels about the contribution the other member has done for the project. This is to make sure everyone in the group will contribute their fair share to the project. We will use simple arithmetic average ratings each

¹You are required to submit the `group.csv` whenever there is an update of your group membership or project manager role. When there is no update, you are welcome to submit it again, though not required.

Peer Rating	Contribution Factor CF
[7, 10]	100%
[6, 7)	80%
[5, 6)	60%
[4, 5)	40%
[0, 4)	0%

Table 0.2: Group Project contribution factor table. Each student's lab grade is their group project grade multiplied by the CF (Contribution Factor).

group member received and assign individual lab grade to each team member by multiplying the project grade with a contribution percentage factor listed in Table 0.2. The peer review submission deadline is one hour after the project deadline or one hour after the project submission if the project submission is after the project deadline.

- **Late Submissions**

Late submission is accepted within three days after the deadline. Please be advised that late submission is counted in a unit of day rather than hour or minute. An hour late submission is one day late, so does a fifteen hour late submission. Unless notified otherwise, we always take the latest submission from the Learn dropbox. Assume the submission is H hours after the deadline. Then the number of days you are late is computed by the following function given the hours your are late:

```
#include <math.h>

int get_late_days(double late_hours) {
    return (int) (ceil(late_hours/24));
}
```

- **Late Project Submissions.** There are *five grace days*² that can be used for project deliverables late submissions without incurring any penalty. A group split-up will consume one grace day. When you use up all your grace days, a 15% per day late penalty will be applied to a late submission. *Submission is not accepted if it is more than three days late.*
- **Late Peer Review Submissions.** Late peer review submission is accepted within three days after the peer review submission deadline. A 5% per day late penalty will be applied to individual's lab grade when the peer review submission is late.

²Grace days are calendar days. Days in weekends are counted.

Project Grading Policy

- **Project Grading Procedure.** The project is graded by automated testing framework. For each deliverable, we publish a small set of testing cases. We require students to pass these testing cases before they submit. If you are not able to pass these testing cases, then you will be given a chance to demo your project and the maximum grade you will get would be capped to 70.
- **Hardware vs. Simulator.** Submissions will be evaluated on a Windows 10 lab machine that has a board attached to it. Lab machines are accessible through [ENGLab remote desktop session](#) when connected to the campus virtual private network (VPN). If a lab requires the program to run on the board, but the program only functions inside the simulator, a 15% penalty will be applied the particular lab's grade.
- **Project Demo Policy.** Every group will have a chance to request a demo session with a grading TA after lab grades are released. The purpose of project demo is mainly for re-grading your project. If you have no concerns with your lab grade, then you are not required to attend the demo. Each demo has a time limit. You are allowed to make up to 1024 bytes of change of the source. Any change of the source code will consume one grace day or a 15% late submission penalty should all grace days are exhausted. Note change of the source code needs to be done through a text editor. Directly cut and paste from a file not within your submission or replace a source file with an file not in your submission is strictly prohibited.

If you are not interested in re-grading your project, but want to ask grading TA some questions or advises, you may also request a demo after the grades are released.

- **Second Project Re-grading after the Demo.** If you are still not satisfied with the grades received after the demo, escalate your case to the lab instructor to request a review and the lab instructor will finalize the case. Please note any re-grading (including the demo re-grading) is a rigid process. The entire project is subject to be re-graded. Your new grades may be lower, unchanged or higher than the original grade received.

Lab Repeating Policy

For a student who repeats the course, labs need to be re-done with new lab partners. Simply turning in the old lab code is not allowed. We understand that the student may choose a similar route to the solution chosen last time the course was taken. However it should not be identical. The labs will be done a second time, we expect that the student will improve the older solutions. Also the new lab partners should be contributing equally, which will also lead to differences in the solutions.

Note that the policy is course specific to the discretion of the course instructor and the lab instructor.

Lab Projects Solution Internet Policy

Publishing your lab projects solution source code or lab report on the internet for public to access is a violation of academic integrity. Because this potentially enabling other groups to cheat the system in the current and future offerings of the course. For example, it is not acceptable to host a public repository on GitHub that contains your lab project solutions. A lab grade zero will automatically be assigned to the offender.

Seeking Help

- **Discussion Forum.** We recommend students to use the Piazza discussion forum to ask the teaching team questions instead of sending individual emails to lab teaching staff. For questions related to lab projects, our target response time is one business day before the deadline of the particular lab in question³. *There is no guarantee on the response time to questions of a lab that passes the submission deadline.*
- **Office Hours.** The lab office hours are for group project consultation. Your entire group may attend the same appointment. Each appointment is a 15 minute time slot. Book multiple consecutive time slots if you need more time. All appointments require a minimum 1 hour lead time. The maximum lead time you can book a lab office hour is 5 days. You should cancel your booked appointment if you are not able to attend it. There are other students that may need to slots, so please do not book appointments if you are not able to make the appointment.
- **Appointments.** Students can also make appointments with lab teaching staff should their problems are not resolved by discussion forum or during office hours. The appointment booking is by email.

To make the appointment efficient and effective, when requesting an appointment, please specify three preferred times and roughly how long the appointment needs to be. On average, an appointment is fifteen minutes per project group. Please also summarize the main questions to be asked in your appointment requesting email. If a question requires teaching staff to look at a code fragment, please make sure your code is accessible by the lab teaching staff.

³Our past experiences show that the number of questions spike when deadline is close. The teaching staff will not be able to guarantee one business day response time when workload is above average, though we always try our best to provide timely response.

Please note that teaching staff will not debug student's program for the student. Debugging is part of the exercise of finishing a programming assignment. Teaching staff will be able to demonstrate how to use the debugger and provide case specific debugging tips. Teaching staff will not give direct solution to a lab assignment. Guidances and hints will be provided to help students to find the solution by themselves.

Part II

Lab Project

Chapter 1

Introduction

1.1 Overview

In this project, you will design a small real-time executive (RTX) and implement it on a Intel DE1-SoC board populated with the Cyclone V SoC chip, which has a dual-core ARM Cortex-A9 Hard Processor System (HPS) and Altera FPGA. The executive will provide a basic multiprogramming environment, with five priority levels, preemption, dynamic memory management, mailbox for inter-task communications and synchronization, a basic timing service, system console I/O and debugging support, and finally real-time scheduling.

Such an RTX is suitable for embedded computers which operate in real time. A cooperative, non-malicious software environment is assumed. The design of the RTX should allow its placement in ROM. Unprivileged RTX tasks must execute under the user mode of the Cortex-A9 processor. The RTX kernel and kernel tasks will execute in the privileged level under supervisor mode.

On the HPS side, there is an on-chip RAM of 64 KB and a DDR3 RAM of 1 GB for use by the RTX and application tasks. The board has four Hard Processor System (HPS) timers, two JTAG UARTs and several other peripheral interface devices. The UART0 is used for your RTX system console and UART1 is used for your RTX debug terminal.

1.2 Summary of RTX Requirements

The RTX requirements are listed as follows:

1.2.1 RTX Primitives and Services

The RTX provides primitives and services for as following.

Memory Management

First fit dynamic memory allocation is supported. Refer to Chapter 2 for details.

Task Management

The RTX fixed number of tasks. The maximum number of tasks that can run is decided at compile time. The RTX supports task creation and deletion during run time. The RTX supports task preemption. There are four user priority levels plus an additional “hidden” priority level for the Null task. There is no time slicing. FIFO (First In, First Out) scheduling policy at each priority level is supported. Refer to Chapter 3 for details.

Synchronization, Timing Service and Console I/O

The RTX provides mailbox utility for inter-task communication and synchronization. An interrupt-driven UART provides the console service. The RTX provides a primitive for a task to pause itself and for a primitive to query the kernel internal clock ticks. Refer to Chapter 4 for details.

Real-Time Dynamic Scheduling

The EDF (Earliest Deadline First) scheduling policy at each priority level is supported. Refer to Chapter 5 for details.

1.2.2 RTX Tasks

You are required to implement two types of tasks by using the RTX primitives and services. They are user tasks and kernel tasks.

User Tasks

These tasks are operating at a unprivileged level in user mode. They are user applications that perform certain user defined functions. For each lab project, you will implement test tasks to help you test the RTX primitives and services you have designed and implemented. In later labs, you will add tasks that require console I/O services once you have the console I/O service ready.

System Tasks

These tasks are operating in user mode or supervisor mode. Some may require a privileged level of operation and some may be sufficient to operate at a unprivi-

leged level. It is your design decision to justify which task will be operating at what privilege level. Three system tasks are required and they are null task (see Chapter 2), console display task and keyboard command decoder task (see Chapter 4).

1.2.3 RTX Footprint and Processor Loading

A reasonably *lean* implementation is expected. No standard C library function call is allowed in the kernel code.

1.2.4 Error Detection and Recovery

The primitive will return an error code (a non-zero integer value) upon an error. No error recovery is required. It may be assumed that the application processes can deal with this situation.

Chapter 2

Lab1: Introduction to Kernel Programming and Memory Management

2.1 Objective

This lab is an introduction to kernel programming for the ARM Cortex-A9 processor. You will become familiar with the ARM Development Studio (DS) Integrated Development Environment (IDE). You will implement a set of system calls related to memory management. In this lab, you will learn:

- How to use the ARM DS IDE to edit, debug, simulate and execute a bare-metal project;
- How to use SVC as a gateway to program a system call in the kernel space for ARM Cortex-A9 processor; and
- How to design and implement first-fit memory management data structure and algorithm.

2.2 Starter Files

The starter files are uploaded on GitHub at <http://github.com/yqh/ece350/> under the `manual_code/lab1` sub-directory. The sub-directory contains the following sub-directories:

- `template`: This sub-directory contains
 - `src`: the skeleton source code,
 - `scatter1.sct`: the scatter file,
 - `setup_scatter_file.txt`: instructions to setup the scatter file, and

- `template.c`: an empty c file with documentation template.
- `SVC/`: This sub-directory contains a bare-metal ARM DS skeleton project for lab1. It contains all the functions that you need to implement in this lab. It also contains sample third-party test cases:
 - `src/INC`: This folder contains header files for the RTX API:
 - * `common.h`: the header file that both kernel and user-space code can include,
 - * `common_ext.h`: the extended header file that both the kernel and the user can include, and
 - * `rtx.h`: the RTX user API file.
 IMPORTANT: You should not modify the `rtx.h` or the `common.h` file.
 - `src/app`: This folder contains sample third-party test cases.
 - `src/board/VE_A9`: This folder contains the board support package for the ARM VE_A9 fixed virtual platform.
 - `src/kernel`: This folder contains all the kernel source code. You will mainly work on the `k_mem.c` file. See Section 2.4.3 for more details on what you can and cannot modify in this folder.

2.3 Pre-lab Preparation

- Read Sections 1, 2, 3, 9, 10, and 11.1 in “Introduction to the ARM* Processor Using ARM Toolchain” [3];
- Read “SVC Programming: Writing an RTX API Function” in Section 8.5;
- Create your lab1 ARM DS Project from skeleton source code (See Chapter 7);
- Execute the `SVC` project on the ARM DS by using the `VE_Cortex_A9x1` ARM FVP¹; and
- Read “Free-space Management” at <http://pages.cs.wisc.edu/~remzi/OSTEP/vm-freespace.pdf>.

2.4 Assignment

You should submit a programming project and a report. Your report should document the data structures, algorithms (if applicable), and test scenarios that you develop for your programming project.

¹FVP stands for Fixed Virtual Platform, which is the simulator.

2.4.1 Programming Project

You should implement the first-fit memory allocation. You will first implement the memory-initialization function, which initializes the RTX's memory manager. You will then implement the allocation and de-allocation functions. You will also implement a utility function to analyze the efficiency of the allocation algorithm and its implementation. Finally, you will write test cases to verify your implementation.

Description of Functions

The specification of each function to be implemented are described below:

Memory Initialization Function

NAME

`mem_init` - initialize the dynamic memory manager

SYNOPSIS

```
#include "rtx.h"

int mem_init();
```

DESCRIPTION

The `mem_init()` system call initializes the RTX's memory manager. A memory region is a set of consecutive bytes in physical memory. Initially, there is only one free region ~~that includes all the memory regions~~. As the manager allocates and deallocates memory regions (see `mem_alloc` and `mem_dealloc`), the memory will be partitioned into free and allocated regions. You need to design appropriate data structures to easily track the free and allocated regions. Please note that the size of the data structures that you will design for this purpose has to be minimal because they will occupy a portion of the free space and will be considered as overhead.

RETURN VALUE

The function returns 0 on success and -1 on failure, which happens if there is no free space in physical memory.

The ARM Versatile Express Cortex-A9 Fixed Virtual Platform has 2 GiB memory starting from physical address of 0x80000000. We will only use the first 1 GiB of the memory in this lab, which ends at the physical address of 0xBFFFFFFF. Your OS image will occupy some memory from this address. The end of your OS image is the starting address of the free space to be managed. The end address of the free space to be managed is 0xBFFFFFFF. The `scatter1.sct` file makes the linker generate a variable `Image$$ZI_DATA$$ZI$$ZI_Limit` to indicate the end of the OS Image. The free space your memory manager

will manage starts from the address of this linker defined symbol and ends at 0xBFFFFFFF.

The system call traps into the kernel and then initializes the memory manager. You are responsible for designing and implementing data structures used to track free and allocated memory regions ².

Allocation Function:

NAME

`mem_alloc` - allocate dynamic memory

SYNOPSIS

```
#include "rtx.h"

void *mem_alloc(size_t size);
```

DESCRIPTION

In short, the `mem_alloc()` system call allocates `size` bytes according to the first-fit algorithm and returns a pointer to the beginning of the allocated memory region ³. The `size` argument is the number of bytes requested from the memory manager. The memory manager then returns the starting address of a consecutive region of memory with the requested size. The memory address should be four-byte aligned. If `size` is 0, then `mem_alloc()` returns `NULL`. The allocated memory is not initialized (i.e., RTX does not need to set the content of the allocated region to zero). Memory requests may be of any size.

RETURN VALUE

The function returns a pointer to the allocated memory or `NULL` if the request fails. Failure happens if RTX cannot allocate the requested memory for whatever reason.

Deallocation Function: NAME

`mem_dealloc` - Free dynamic memory

SYNOPSIS

```
#include "rtx.h"

int mem_dealloc(void *ptr);
```

²In Lab2, where we add multi-tasking support to the RTX, you will need to track memory regions that are allocated to each task. This is not required in Lab1, because multi-tasking is not yet supported in your RTX.

³The `mem_init()` needs to be invoked before calling `mem_alloc()`. Otherwise, the behaviour is undefined.

DESCRIPTION

The `mem_dealloc()` system call frees the memory space pointed to by `ptr`, which must have been returned by a previous call to `mem_alloc()`. Otherwise, or if `mem_dealloc(ptr)` has already been called before, undefined behaviour occurs. If `ptr` is `NULL`, no action is performed. If the newly-freed memory region is adjacent to other free memory regions, they all have to be merged immediately (i.e., immediate coalescence) and the combined region is then re-integrated into the free memory under management. The RTX does not clear the content of the newly-freed region.

RETURN VALUE

This function returns 0 on success and -1 on failure. Failure happens when the RTX cannot successfully free the memory region for whatever reason.

Utility Function:

NAME

`mem_count_extfrag` - Count externally-fragmented memory regions

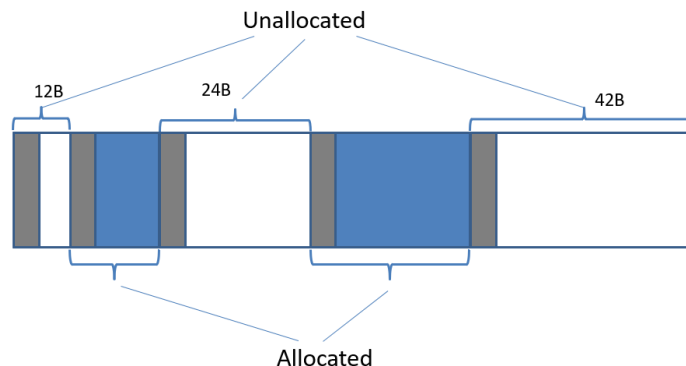
SYNOPSIS

```
#include "rtx.h"

int mem_count_extfrag(size_t size);
```

DESCRIPTION

The `mem_count_extfrag` system call counts the number of free (i.e. unallocated) memory regions that are of size strictly less than `size`. The `size` argument is in bytes. The space that your memory-management data structures occupy inside each free region is considered to be free in this context. For example, assume that the memory status is ~~show~~ as follows.



The grey regions are occupied by the memory manager's data structures. The white regions indicate free spaces to be allocated. And blue regions indicate already-allocated memory regions. Calling `mem_count_extfrag` with 12, 42, and 43 as inputs should return 0, 2, and 3, respectively.

Testing

In order to test your implementation, you need to write at least three test cases in `lab1/SVC/src/app/ae_mem.c` file. To get some ideas, you could look into the sample test cases that are provided with the starter code (your test cases should be different for the sample test cases). You also need to document the specification of your test cases in the report (see 2.4.2). There is no hard requirement on the exact testing scenarios. The rule of thumb is that the tests should convince you that your implementation is correct. For example, you may want to consider repeatedly allocating and then deallocating memory and make sure no extra memory appears or no memory gets lost. The sum of free memory and allocated memory should always be constant. Another aspect to consider is the external fragmentation. Allocate and deallocate memory with different sizes and see how external fragmentation is affected. You will find the utility function `mem_count_extfrag()` to be a useful tool.

Performance

Two metrics are used to measure the performance of your implementation.

- **Throughput.** Let T be the time it takes for a sequence of N requests to be completed (a request can be an allocation request or a deallocation request). Throughput is defined as:

$$R_T = \frac{N}{T}. \quad (2.1)$$

For example, if your RTX can serve 100 allocation requests and 100 deallocation requests in one second, then the Throughput of your memory manager is 200 operations per second.

- **Heap utilization ratio.** This metric measures the overhead of the data structures used to implement the memory manager. Let P be the total number of bytes allocated after a sequence of requests is served. Let H be the entire heap size. The heap utilization ratio is defined as

$$R_H = \frac{P}{H}. \quad (2.2)$$

2.4.2 Report

Write the following items in your report and name it `p1_report.pdf`.

- Descriptions of the data structures and algorithms (if applicable) used to implement the allocation strategy;
- Test-case descriptions; and
- Test results.

If you use specific algorithms that need to be described, then use pseudocode to highlight the algorithms' main ideas. For test cases, include three or more non-trivial testing scenarios. **A** non-trivial test case should test some important aspects of your implementation. For example, your test cases should verify that your code at least does the following correctly:

- coalescing free regions,
- reusing newly-freed regions,
- not leaking memory (size of heap should not increase or decrease),
- utilizing memory with minimum overheads, and
- returning correct number of externally-fragmented regions.

For test results, include throughput and heap utilization results for all your test cases.

2.4.3 Third-party Testing and Source Code File Organization

We will write third-party test cases to verify the correctness of your implementation. In order to do so, you will need to maintain the file organization of the project skeleton in the starter code. There are dos and don'ts that you need to follow.

Don'ts

- Do not move any file from the `src` directory to any other directories;
- Do not change the file names under the `src` directory;
- Do not make any changes to the contents of the `rtx.h` and `common.h` files;
- Do not change the existing function prototype in the given `k_mem.[ch]` files; and
- Do not include any new header files in the `lab1/SVC/src/app/ae_mem.c`.
- Do not modify the `ae.[ch]` files.

Dos

- You are allowed to add new self-defined functions to `k_mem.[ch]`.
- You are also allowed to create new `.h` and `.c` files⁴.
- The newly created `.h` file is allowed to be included in the `k_mem.c` file.
- Any new files you add to the project can be put into either the `src` directory or other directories you will create.

Note that the `main_svc.c` calls third-party testing by calling `ae_init` and `ae_start` functions which the third-party testing software implements. The function prototypes of these two functions do not change. But the implementation of these two functions may change in real testing. Do not delete the lines in the `main_svc.c` where these two functions are invoked. During the third-party testing, the files under the `app` directory will be replaced by more complicated testing cases than the ones published on GitHub.

2.5 Deliverable

2.5.1 Pre-Lab Deliverables

~~Fill the `project_manager.csv` template file and submit it to Lab1 Dropbox on Learn. None.~~

2.5.2 Post-Lab Deliverables

Create a directory named “lab1”. Then create a sub-directory named “code” under “lab1”. Put your ~~uVision Project~~ **ARM DS project** folder under “lab1/code”. Put the `p1_report.pdf` under the “lab1” directory. Include a README file with ~~group identification, project manager name~~ and a description of directory contents. Put the README file under the “lab1” directory. Zip everything inside the lab1 directory and submit it to Learn Lab1 Dropbox.

2.6 Marking Rubric

The Rubric for marking the submitted source code and report is listed in Table 2.1. The functionality and performance of your implementation will be tested by a third-party testing program and a minimum **20 points** will be deducted if we find memory

⁴For example, you may want to create linked list data structure functions or some helper functions. You may want to create new files to hold these functions for better file organization.

is lost or extra memory appears after repeating allocation and de-allocation function calls. We will also conduct manual random code inspection.

Points	Sub-points	Description
90		Source Code
	10	Code compilation
	80	Third-party testing Manual code inspection
10		Report
	3	Description of data structures and algorithms
	3	Testing scenario descriptions
	4	Testing results

Table 2.1: Lab1 Marking Rubric

2.7 Errata

1. Page 14, in the DESCRIPTION section: “that includes all the memory retions” is removed.
2. Page 14, last paragraph, 0XBFFFFFFFFF is updated to 0xBFFFFFFFFF.
3. Page 15, the coding listing box at the bottom of the page. The return type of the function has been changed from `void` to `int`.
4. Page 16, first paragraph, “i.e immediate coalescence”, added a comma after “i.e.”.
5. Page 16, The RETURN VALUE section has been updated to reflect the return type correction of `mem_dealloc` made on page 14.
6. Page 16, last paragraph, “is show as follows” is corrected to “is as follows”.
7. Page 18, first paragraph after the bullets, “a non-trivial” is replaced by “A non-trivial”.
8. Page 19, Section 2.5.1, original contents is replaced by “None.”.
9. Page 19, Section 2.5.2, changed “uVision Project” to “ARM DS project”.
10. Page 19, Section 2.5.2, removed “group identification, project manager name and”.

Chapter 3

Lab2 Task Management

3.1 Objective

In this lab, you will create a preemptive multi-tasked kernel. In particular, you will design and implement system calls to manage tasks ¹. Additionally, you will design and implement a utility system call to return tasks information. Finally, you will modify the memory management system calls from lab1 to support memory ownership. More specifically, you will learn:

- How to design and implement kernel support for multi-tasking,
- How to design and implement kernel support for scheduling tasks with different priorities, and
- How to design and implement kernel support for task preemption.

3.2 Starter Files

The starter files are uploaded on GitHub at <http://github.com/yqh/ece350/> under the `manual_code/lab2` directory. The directory contains `TM` sub-directory which is a bare-metal ARM DS skeleton project for lab2. The sub-directory follows the same organization as the `SVC` sub-directory in lab1. You will mainly work on the `src/kernel/k_mem.c` and `src/kernel/k_task.c` files. See Section 3.4.3 for more details on what you can and cannot modify.

¹A task in our RTX resembles a single-threaded process in general-purpose OS. But, there are several differences between our tasks and general-purpose processes. The most important difference is that in our RTX we do not have isolated address spaces for tasks. All tasks share the same address space with themselves and the kernel. We assume that programmers write well-behaved tasks that are not malicious.

3.3 Pre-lab Preparation

- Run your lab1 code on the actual board (see Chapter 7.10 for more details).
- Review the lecture notes on multi-threaded kernels.
- Review Sections 9, 10, and 11 in [3].
- Run the lab2 starter code on the simulator and on the actual board.
- Work through the context switching code and understand what they do and how they work.

3.4 Assignment

You should submit a programming project and a report. Your report should document the data structures, algorithms (if applicable), and test scenarios that you develop for your programming project.

3.4.1 Programming Project

You will design and implement a priority-based, preemptive multi-tasked kernel. The maximum number of (kernel and user-mode) tasks that could co-exist in the system will be fixed and specified at compile time. First, you will implement system calls to create a task, delete a task, yield processor, set priority of a task, and get information about a task. Next, you will implement a simple strict-priority scheduler to schedule ready tasks. Then, you will enhance the `mem_alloc` and `mem_dealloc` system calls that you implemented in lab1 so that ~~the~~ your RTX keeps track of the ownership of each allocated memory (i.e., the task that invokes `mem_alloc` will own the allocated memory.). When a task invokes `mem_dealloc`, it will fail if the input memory is not owned by the calling task. Finally, you will design and implement a number of task cases to verify your design and implementation.

Macros and User Task Data Structure

For this lab, the relevant macros from the `common.h` file are as follows.

```
#define TID_NULL 0    /* pre-defined Task ID for null task*/
#define MAX_TASKS 16 /* maximum number of tasks in the system */
#define KERN_STACK_SIZE 0x200 /* kernel stack size in bytes*/
#define PROC_STACK_SIZE 0x200 /* proc space stack size in bytes*/

/* Priorities (greater numbers indicate lower priorities)*/
#define HIGH 100
#define MEDIUM 101
```



```

#define LOW      102
#define LOWEST   103
#define PRIO_NULL 255 /* reserved priority for null task */

/* task state macro */
#define DORMANT   0    /* terminated task state*/
#define READY    1
#define RUNNING  2

```

An important data structure that will be used in this lab is the `rtx_task_info`. The relevant elements of the data structure are as follows.

```

typedef struct rtx_task_info {
    void (*ptask)(); /* Entry address */
    U32 k_sp; /* Current kernel stack pointer */
    U32 k_stack_hi; /* Kernel stack starting addr. (high addr.) */
    U32 u_sp; /* Current user-space stack pointer */
    U32 u_stack_hi; /* User stack starting addr. (high addr.) */
    U16 k_stack_size; /* Size of kernel stack in bytes */
    U16 u_stack_size; /* Size of user-space stack in bytes */
    task_t tid; /* Task ID */
    U8 prio; /* Priority */
    U8 state; /* Task state */
    U8 priv; /* Privilege (0 unprivileged and 1 privileged) */
} RTX_TASK_INFO;

```

This structure is used to launch initial tasks during the RTX initialization (see `k_tsk_init` function in `k_task.c` file). Please note that each unprivileged user-mode task, has two separate stacks: a user-space stack and a kernel stack. Each privileged kernel task, only has one stack: a kernel stack. User-mode tasks are created by the `k_task_create` function (see Chapter 3.4.1). Kernel tasks are created during the rtx initialization (see for example the `k_tsk_init` and `k_tsk_create_new` functions in the `k_task.c` file).

Description of Functions

This section provides specifications of each function that needs to be implemented.

Task Creation Function

NAME

`k_tsk_create` - create an unprivileged task

SYNOPSIS

```

#include "k_rtx.h"

int k_tsk_create(task_t *task, void (*task_entry)(void),

```

```
U8 prio, U16 stack_size);
```

DESCRIPTION

The `k_tsk_create` function creates a new user-mode task at runtime. Once created, each task is given a unique task id (TID). A TID is an integer between 0 and $N - 1$, where N is the maximum number of tasks (including the null task) the kernel supports and is decided by the `MAX_TASKS` macro defined in the `common.h` file. The task ID 0 is reserved for the null task (see 3.4.1). Before returning, a successful call to `k_tsk_create` stores the TID of the new task in the buffer pointed to by `task`. The `task_entry` argument point to the entry point of the task (i.e., when the newly-created task runs for the first time, it starts running from `task_entry`). The `prio` argument sets the initial priority of the new task. For this lab, your RTX should support four priority levels—`LOWEST`, `LOW`, `MEDIUM` and `HIGH`, which are macros that are defined in the `commoh.h` file. The `stack_size` argument is a multiple of 8, and it specifies the size of the user-space stack in bytes. The kernel is responsible for allocating the space for the user-space stack and freeing the stack space when the task terminates. Once allocated, the owner of the user-space stack for the newly-created task becomes the kernel. Please not that the caller of `k_tsk_create` never blocks, but it could be preempted (see the description of `scheduler()` function for more details).

RETURN VALUE

The `k_tsk_create` function returns 0 on success and -1 on failure. Failure happens when the number of tasks has reached its maximum, or when the stack size is too small (i.e., less than `PROC_STACK_SIZE`) or too big for the system to support, or when the `prio` is invalid. This is not a complete list of failure causes. Your implementation should cover all other possible failure causes when you design your RTX.

User-level tasks can create other user-level tasks only by calling `tsk_create` (and including `rtx.h`), which then traps into the kernel and runs `k_tsk_create`. Kernel tasks can create user-level tasks by directly calling `k_tsk_create`. You are responsible for designing and implementing the task-control-block (TCB) data structure (a preliminary version of TCB is provided in the starter code; you will have to add/remove elements as you see fit). The structure will be used to track task information including, but not limited to, the state of task, user stack pointer and kernel stack pointer.

Important Note. You will have to implement user-mode tasks with both kernel and user-space stacks. The kernel stack is statically allocated in kernel code and only has to be assigned to each created task. The user-space stack, however, must be allocated dynamically when a task is created. To do this, you will have to modify the `k_alloc_p_stack` function in `k_mem.c` file. Your TCB

structure should include pointers to both stacks. The user-space stack must be used only for user-mode code, and the kernel stack must be only used for kernel code (e.g., system calls).

Task Termination Function

NAME

`k_tsk_exit` - terminate the calling task

SYNOPSIS

```
#include "k_rtx.h"

void k_tsk_exit(void);
```

DESCRIPTION

The `k_tsk_exit()` function stops and deletes the currently running task. Once a task is terminated, its state becomes `DORMANT` if its TCB data structure still exists in the system. Once a running test terminates, the RTX should schedule another ready task to run (the null task will always be ready to run if there are no other ready tasks).

RETURN VALUE

The function does not return.

Similar to `tsk_create`, user-mode tasks can terminate only by calling `tsk_exit`, which then traps into the kernel and runs `k_tsk_exit`. And kernel tasks can terminate by directly calling `k_tsk_exit`.

Task Priority Function

NAME

`k_tsk_set_prio` - set task priority at runtime

SYNOPSIS

```
#include "k_rtx.h"

int k_tsk_set_prio(task_t task_id, U8 prio);
```

DESCRIPTION

The `k_tsk_set_prio()` function changes the priority of the task identified by `task_id` to `prio`. The full list of task priority values are `HIGH`, `MEDIUM`, `LOW`, `LOWEST`, and `PRIO_NULL`. The priority level `PRIO_NULL` is reserved for the null task and cannot be assigned to any other task. A user-mode task can change the priority of any other user-mode task (including itself), but it cannot change the priority of any kernel task. A kernel task,

however, can change the priority of any kernel or user-mode task (including itself). The priority of the null task cannot be changed and remains at level `PRIO_NULL`. The caller of `k_tsk_set_prio` never blocks, but it could be preempted (see the description of `scheduler()` function for more details).

RETURN VALUE

The function returns 0 on success and -1 on failure. Example causes of failure are an invalid TID, an invalid priority level, or a priority change that is not allowed (as described in the description of the function).

Similar to `tsk_create`, user-mode tasks can only call `tsk_set_prio`, which then traps into the kernel and runs `k_tsk_set_prio`. And kernel tasks can directly call `k_tsk_set_prio`.

Task Utility Function

NAME

`k_tsk_get` - obtain task status information from the kernel

SYNOPSIS

```
#include "k_rtx.h"

int k_tsk_get(task_t task_id, RTX_TASK_INFO *buffer);
```

DESCRIPTION

The `k_tsk_get()` function obtains system information and stores it in the buffer pointed by `buffer` before returning. The buffer is a `rtx_task_info` structure defined in `common.h`

RETURN VALUE

The function returns 0 on success and -1 on failure. Example causes of failure are an invalid TID or a `buffer` which is a null pointer.

Scheduling Name

`scheduler` - return the highest-priority runnable task

SYNOPSIS

```
#include "k_rtx.h"

TCB *scheduler(void);
```

DESCRIPTION

This function returns the highest-priority task among all runnable tasks (i.e., ready tasks plus currently-running task). Runnable tasks are scheduled based on a simple strict-priority scheduling algorithm. This means that every time that the kernel needs to make a scheduling decision, it picks the runnable task with highest priority. To implement this, the ready queue for the processor should maintain a sorted list of ready tasks based on their priorities. If there are multiple tasks with the same priority, they are sorted based on first-come-first-serve policy (i.e., among same-priority tasks, the task that was added to the ready queue first, has higher priority). The kernel has to make scheduling decisions every time the state of any task changes (e.g., a task is created, a task exists, a task yields, or a task's priority changes). For example, suppose that a running task, A, creates a new task, B. If the priority of B is higher than that of A, then B preempts A and starts running immediately. The preempted task, A, is added to the front of the ready queue. In other words, when A goes back to the ready queue, it maintains its previous position among tasks with the same priority (i.e., A will be sorted as the first task among all tasks with the same priority as A). If the priority of B is ~~lower than~~ **not higher than** that of A, then B will be added to the back of the ready queue (i.e., it will be sorted as the last task among all tasks with the same priority as B). As another example, assume that a running task, C, changes the priority of a ready task, D. If the new priority of D is higher than that of C, then D preempts C and runs immediately (note that D has to be in the ready state to be able to preempt C). The preempted task, C, is added to the front of the ready queue (i.e., it will be sorted as the first task among all tasks with the same priority as C). If the new priority of D is not higher than that of C, then D will be (re-)added to the back of the ready queue (i.e., it will be sorted as the last task among all tasks with the same priority as D). **If C changes its own priority, it should continue running unless the new priority is lower than the priority of the highest-priority task in the ready queue, in which case C is added to the back of the ready queue (i.e., it will be sorted as the last task among all tasks with the same priority).** As a final example, suppose that a running task, E, calls `task_yield`. If the priority of the highest-priority task in the ready queue, F, is strictly less than that of E, then E should continue to run. Otherwise, F is scheduled, and E is added to the back of the ready queue (i.e., it will be sorted as the last task among all tasks with the same priority as E).

RETURN VALUE

The function returns TCB of the the highest-priority ready task.

Please note that the performance of the scheduler is one of the most important aspects of a real-time operating system. In future labs, you will measure and optimize your scheduler. It might be hard to change your code later on. Therefore, you might want to spend some time now thinking about efficient data structures and algorithms to add, remove, and sort ready tasks in the ready

queue.

Memory Management Functions You will add the notion of ownership to your memory manager. When a task invokes `k_mem_alloc` and the system returns a valid memory address to it, the returned memory block is owned by the calling task (`gp_current_task`). Only the owner of a memory block can successfully deallocate that memory block. If a task calls `k_mem_dealloc` with a memory that it does not own, the function will return -1. **Additionally, you will have to modify your `k_mem_alloc` function to return 8-byte aligned addresses.** Other functionalities of the memory management functions remain the same as described in lab1 manual.

Required Tasks

The Null Task The kernel has to run a task at any given time. If there are no ready tasks, then kernel will run the null task, which is created during the initialization (see `k_tsk_init` function in `k_task.c` file). The null task operates at the priority level `PRIO_NULL`. The `PRIO_NULL` is a hidden priority level reserved for the null task only. Task ID 0 is reserved for the null task. So when there is no other ready tasks, the null task is scheduled to run.

Testing Tasks In order to test your implementation of the required functions, write an application that uses your kernel primitives. The provided `ae_priv_tasks.[ch]` and `ae_usr_tasks.[ch]` files are for writing kernel and user-mode tasks, respectively. To set the initial tasks that have to be created during initialization, you will have to modify the `ae_set_task_info` function in `ae.c` file and the `main` function in `main_svc_cw.c` file. Please note that you will have to keep the interfaces defined in `ae.h` file unchanged. Create a set of testing scenarios to test your implemented functions and document the testing specification in the report (See 2.4.2). There is no hard requirement on what tests to be implemented. The rule of thumb is that the tests should be comprehensive enough to convince you that your implementation is correct. For example, you may want to consider repeatedly creating and then terminating tasks while making sure that no extra task is created or no task gets lost. Another testing objective that you may want to consider is preemption. You could create multiple tasks with different priorities and change their priority at runtime to test preemption. The utility functions `mem_count_extfrag` and `tsk_get()` are useful tools for checking system memory and task status information.

3.4.2 Report

Write the following items in your report and name it `p2_report.pdf`.

- Descriptions of the data structures and algorithms used to implement all the functions listed in Chapter 3.4.1

- Testing scenario descriptions

To illustrate key algorithms, pseudocode is acceptable. For testing, include three or more non-trivial testing scenarios. A non-trivial test case should test some important aspects of your implementation. For example, your test cases should verify that your code at least does the following correctly:

- creating and terminate user-mode tasks,
- scheduling tasks with the same priorities (calling `tsk_yield` and `tsk_set_prio`),
- scheduling tasks with different priorities (calling `tsk_yield` and `tsk_set_prio`),
- not deallocating memory that is not owned, and
- getting correct information on tasks.

3.4.3 Third-party Testing and Source Code File Organization

We will write third-party test cases to verify the correctness of your implementations. In order to do so, you will need to maintain the file organization of the project skeleton in the starter code. There are dos and don'ts you need to follow.

Don'ts

- Do not move any file from the `src` directory to any other directories;
- Do not change the file names under the `src` directory;
- Do not make any changes to the contents of the `rtx.h` and `common.h` files;
- Do not change the existing function prototype in the given `k_mem.[ch]` and `k_task.[ch]` files;
- Do not include any new header files in the `app/ae_mem.c` file;
- Do not modify the `ae.h` file; and
- Do not change the `ae.c` file except the body of `ae_set_task_info` function.

Dos

- You are allowed to add new self-defined functions to `k_mem.[ch]` and `k_task.[ch]`.
- You are also allowed to create new `.h` and `.c` files ².

²For example, you may want to create linked list data structure functions or some helper functions. You may want to create new files to hold these functions for better file organization.

- The newly-created `.h` file is allowed to be included in the `k_mem.c` and `k_task.c` files.
- Any new files you add to the project can be put into either the `src` directory or other directories you will create.
- You are allowed to change the body of functions in `k_task.c` without changing the prototype defined in `k_task.h`.

Please note that the `main_svc_cw.c` calls third-party testing by calling `ae_init` function which then calls `ae_set_task_info` function that implements the third-party testing software. The function prototypes of `ae_set_task_info` function does not change. But the implementation of it may change in real testing. During the third-party testing, the files under the `app` directory will be replaced by more complicated testing cases than the ones published on GitHub.

3.5 Deliverables

3.5.1 Post-Lab Deliverables

Create a directory named “lab2”. Then create a sub-directory named “code” under “lab2”. Put your ARM DS project folder under “lab2/code”. Put the `p2_report.pdf` under the “lab2” directory. Include a README file with a description of directory contents. Put the README file under the “lab2” directory. Zip everything inside the lab2 directory and submit it to Learn Lab2 Dropbox.

3.6 Marking Rubric

The Rubric for marking the submitted source code and report is listed in Table 3.1. The functionality of your implementation will be tested by a third-party test program. A minimum of **20 points** will be deducted if memory is lost or extra memory appears after calls to allocate and deallocate memory. A minimum of **20 points** will be deducted if tasks are lost or extra tasks mysteriously appear after calls to create and delete tasks. A minimum of **30 points** will be deducted if tasks are not created, terminated, or preempted correctly.

3.7 Errata

1. In page 22, “the your RTX” is changed to “your RTX.”
2. In page 28, the following requirement is added: “Additionally, you will have to modify your `k_mem_alloc` function to return 8-byte aligned addresses.”

Points	Sub-Points	Description
90		Source Code
	10	Code compilation
	80	Third-party testing Manual code inspection
10		Report

Table 3.1: Lab2 Marking Rubric

3. In page 30, the following item is added: "You are allowed to change the body of functions in `k_task.c` without changing the prototype defined in `k_task.h`."
4. In page 24, the following requirement is added: "The `stack_size` argument is a multiple of 8 (...)."
5. In page 27, new clarifications are added to the scheduler.

Chapter 4

Lab3 Inter-task Communications and UART Interrupts

In this lab you will work on inter-task communications and handling UART interrupts. In particular, you will design and implement system calls to create and manage mailboxes for tasks. You will also design and implement a task to enable the RTX terminal. After this lab, you will learn:

- How to design and implement mailbox API to support inter-task communications,
- How to block and unblock a task,
- How to work with UART interrupts, and
- How to design and implement a simple terminal task.

4.1 Starter Files

The starter files are uploaded on GitHub at <http://github.com/yqh/ece350/> under the `manual_code/lab3` directory. The directory contains `IPC` sub-directory which is a bare-metal ARM DS skeleton project for lab3. The sub-directory follows the same organization as the `TM` sub-directory in lab2. In fact, `IPC` project is mainly the `TM` project with minor changes. The main change is the addition of the interrupt handler code (i.e., `IRQ_Handler` function) in `HAL_CA.c` file. The interrupt handler is invoked when an interrupt happens. By default, interrupts are disabled in the `SVC` mode and enabled in the `USR` mode. If the interrupt is a UART interrupt, the interrupt handler calls `SER_Interrupt` function. In the starter code, this function simply reads the input from the serial port, echos the input back to the serial port, and calls `k_tsk_run_new` function. In this lab, you will need to modify this function to communicate the input to the terminal task. Similar to previous labs, please see Section 4.3.3 for more details on what you can and cannot modify.

4.2 Pre-lab Preparation

- Refresh your memory on inter-process communications (see for example ECE 252 lecture notes),
- Skim through Chapter 22 of [1],
- Run the lab3 starter code on the actual board (the simulator does not support interrupts), and
- Work through the `IRQ_Handler` code and understand what it does and how it works.

4.3 Assignment

You should submit a programming project and a report. Your report should document the data structures, algorithms (if applicable), and test scenarios that you develop for your programming project.

4.3.1 Programming Project

Overview

You will design and implement message-based inter-task communications. Tasks will be able to request a mailbox to receive messages from other tasks. Tasks will also be able to send and receive messages to and from other tasks. You will also implement a simple terminal task, called the Keyboard Command Decoder (KCD) task. The KCD task will be used to provide direct communication between the end user and the running tasks over the RS232 UART serial port. To this end, you will modify the UART handler to forwards received characters to the KCD task's mailbox.

Description of Functions

This section provides specifications of each function that needs to be implemented.

Mailbox Creation Function

NAME

`k_mbx_create` - create a mailbox

SYNOPSIS

```
#include "k_rtx.h"

int k_mbx_create(size_t size);
```

DESCRIPTION

`k_mbx_create` creates a mailbox for the calling task. The `size` argument specifies the capacity of the mailbox in bytes. This capacity is used for the messages and any meta data that kernel might need for each message to manage the mailbox. Each mailbox serves messages using the first-come-first-served policy. Please note that each task will have at most one mailbox. The owner of the memory allocated to each task's mailbox is the kernel. Once a task exits, the memory for its mailbox must be deallocated by the kernel.

RETURN VALUE

The `k_mbx_create` function returns 0 on success and -1 on failure. Possible causes of failure are listed below.

- The calling task already has a mailbox.
- The `size` argument is less than `MIN_MBX_SIZE`.
- The available memory at run time is not enough to create the requested mailbox.

Implementing the mailbox as a ring buffer (i.e., circular queue) is strongly encouraged.

Send Message Function

NAME

`k_send_msg` - send a message to the mailbox of a task.

SYNOPSIS

```
#include "k_rtx.h"

int k_send_msg(task_t receiver_tid, const void* buf);
```

DESCRIPTION

The `k_send_msg` function delivers the message that is specified by `buf` to the mailbox of the task identified by the `receiver_tid`. If the task identified by the `receiver_tid` is blocked on its mailbox (i.e., task's state is `BLK_MSG`), ~~the state of the task should be changed to READY, and the task should be added to the back of the ready queue~~ it becomes unblocked. **In this case, if the priority of the unblocked task is higher than that of the currently-running task, then the unblocked task preempts the currently-running task, and the preempted task is added to the front of the ready queue. If the priority of the unblocked task is not higher than that of the currently-running task, then the unblocked task is added to the back of the ready queue.** The message starts with a message header followed by the actual message data (see Figure 4.1). The message header data structure is as follows.

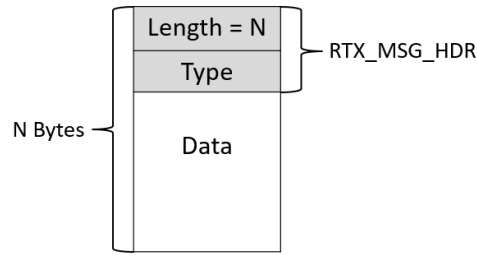


Figure 4.1: Structure of a message buffer

```
typedef struct rtx_msg_hdr {
    U32 length; /* length of message including header size */
    U32 type; /* type of message */
} RTX_MSG_HDR;
```

The `length` field in the structure is the size of the message including the message header size. The `type` field is the message type defined in the `common.h`:

- `DEFAULT`: A general purpose message.
- `KCD_REG`: A message to register a command with the KCD task (see Section 4.3.1).
- `KCD_CMD`: A message that contains a command to be handled by the receiving task (see Section 4.3.1)
- `KEY_IN`: A message that contains an input key (does not need to be only one character, e.g., control keys) from keyboard.

For the data part of the message, please note that both the host computer and the board are little-endian systems. Also note that kernel has to copy the actual message data into the receiver task's mailbox. In addition to the actual message data, kernel might want to store some meta data (e.g., task ID of sender or some information from the message header).

RETURN VALUE

The `k_send_msg` function returns 0 on success and -1 on failure. Possible causes of failures are listed below.

- The task identified by the `receiver_tid` does not exist or is in `DORMANT` state.
- The task identified by the `receiver_tid` exists but does not have a mailbox.
- The `buf` argument is a null pointer.
- The `length` field in the `buf` specifies a size that is less than `MIN_MSG_SIZE`.

- The `receiver_tid`'s mailbox does not have enough free space for the message.

Receive Message Function

NAME

`k_recv_msg` - receive a message

SYNOPSIS

```
#include "k_rtx.h"

int K_recv_msg(task_t *sender_tid, void *buf, size_t len);
```

DESCRIPTION

The task calling `k_recv_msg` receives a message from its mailbox if there are any and gets blocked if there are none. The `sender_tid` will be filled with the sender task ID if it is not a null pointer. When the `sender_tid` is a null pointer, it indicates that the calling task is not interested in obtaining the sender identification. The `buf` will be filled with the received message. The `len` argument specifies the length of `buf` in bytes. The incoming message starts with a message header followed by the actual message data (see Figure 4.1). Messages should be received in the same order that they were delivered to the mailbox (i.e., first-come-first-served). The calling task should allocate enough memory for the `buf` to hold the incoming message. Otherwise, the top message is discarded and the function returns failure. If the mailbox is empty, the calling task is blocked. The state of a blocked task is set to `BLK_MSG` (the task does not return to ready queue). When a running task becomes blocked, the kernel should run the highest-priority task in the ready queue.

RETURN VALUE

The `k_recv_msg` function returns 0 on success and -1 on failure. Possible causes of failure are listed below.

- The calling task does not have a mailbox.
- The `buf` argument is a null pointer.
- The buffer is too small to hold the message.

Keyboard Command Decoder Task

The keyboard command decoder task is an unprivileged user-space task created during initialization. The kernel should reserve `TID_KCD` for KCD's task ID. KCD should request a mailbox when it first starts to run. Once the mailbox is created, in an infinite loop, KCD calls `recv_msg` to receive messages from its mailbox. KCD

only responds to two types of messages (and ignores the rest): (a) command registration (KCD_REG) and terminal keyboard input (KEY_IN). KCD processes received messages as follows.

- Command Registration

A command starts with symbol % followed by 1 or more characters. The single character after % is the command's identifier (identifiers are case sensitive and alphanumeric). The identifier is then followed by command's data if there is any. To register a command with KCD, any task can send a KCD_REG message to KCD. The message's data is only the command's identifier. If data contains more than one character, then KCD ignores the message. Listing 4.1 shows code snippets of registering %W command.

```
size_t msg_hdr_size = sizeof(struct rtx_msg_hdr);
U8 *buf = buffer; /* buffer is allocated by the caller somewhere else
    */
struct rtx_msg_hdr *ptr = (void *)buf;

ptr->length = msg_hdr_size + 1;
ptr->type = KCD_REG;
buf += msg_hdr_size;
*buf = 'W';
send_msg(TID_KCD, (void *)ptr);
```

Listing 4.1: Command Registration

KCD will forward any input command with a registered identifier to the mailbox of its corresponding registered task. Each task can register as many commands as it wants with the KCD task. Tasks can (re-)register an already registered command identifier (tasks will never un-register a command). KCD will always forward a command to the mailbox of the latest task that has registered the command's identifier.

- Keyboard Input

The UART0_IRQ will forward any key input to the mailbox of the KCD using a (possibly multi-character) KEY_IN message (in addition to echoing the key back to the UART serial port). KCD then queues the input keys. Upon receiving "enter" key, KCD dequeues all previous keys to construct a single string. If the string starts with % followed by a registered command, then the KCD will forward this string to the mailbox of the corresponding registered task using a KCD_CMD message. The receiving task is responsible for handling the command. The message body contains the command string (excluding the % character and the "enter" key). If the command identifier is not registered or the registered task no longer exists or sending the message fails, then KCD ignores the string and sends "Command cannot be processed" message to the UART port. If the string does not start with % or the length of the command is more than 64B, then KCD will ignore the string and sends "Invalid Command" to the UART port.

Interrupt Handler

The UART uses interrupts for receiving characters from the serial port. It sends a `KEY_IN` message to the KCD when a keyboard input is received. The task ID `TID_UART0_IRQ` is reserved to indicate the message is from the UART IRQ handler. Please note that UART IRQ handler is not a task. It is an interrupt handler that can send messages to KCD. The UART IRQ handler does not have a mailbox and cannot receive messages from other tasks. The UART IRQ handler uses the kernel stack of the interrupted task. After interrupt handler is done, RTX should resume the interrupted task by default unless the state of some other tasks has changed in which case a scheduling decision has to be made.

Testing Tasks

In order to test your implementation of the required functions, write an application that uses your kernel primitives. There is no hard requirement on what tests to be implemented. The rule of thumb is that the tests should be comprehensive enough to convince you that your implementation is correct.

4.3.2 Report

Write the following items in your report and name it `p3_report.pdf`.

- Descriptions of the data structures and algorithms used to implement all the functions listed in Chapter 4.3.1; and
- Testing scenario descriptions.

To illustrate key algorithms, pseudocode is acceptable. For testing, include three or more non-trivial testing scenarios.

4.3.3 Third-party Testing and Source Code File Organization

We will write third-party test cases to verify the correctness of your implementations. In order to do so, you will need to maintain the file organization of the project skeleton in the starter code. There are dos and don'ts you need to follow.

Don'ts

- Do not change any file in `src/board` directory;
- Do not move any file from the `src` directory to any other directories;
- Do not change the file names under the `src` directory;

- Do not make any changes to the contents of the `rtx.h` and `common.h` files;
- Do not change the existing function prototype in the given `k_mem.[ch]`, `k_task.[ch]`, and `k_msg.[ch]` files;
- Do not include any new header files in the `app/ae_mem.c` file;
- Do not modify the `ae.h` file; and
- Do not change the `ae.c` file except the body of `ae_set_task_info` function.

Dos

- You are allowed to add new self-defined functions to `k_mem.[ch]`, `k_task.[ch]`, and `k_msg.[ch]`.
- You are also allowed to create new `.h` and `.c` files ¹.
- The newly-created `.h` file is allowed to be included in the `k_mem.c`, `k_task.c`, and `k_msg.c` files.
- Any new files you add to the project can be put into either the `src` directory or other directories you will create.
- You are allowed to change the body of functions in `k_task.c` without changing the prototype defined in `k_task.h`.

Please note that the `main_svc_cw.c` calls third-party testing by calling `ae_init` function which then calls `ae_set_task_info` function that implements the third-party testing software. The function prototypes of `ae_set_task_info` function does not change. But the implementation of it may change in real testing. During the third-party testing, the files under the `app` directory will be replaced by more complicated testing cases than the ones published on GitHub.

4.4 Deliverables

4.4.1 Post-Lab Deliverables

Create a directory named “lab3”. Then create a sub-directory named “code” under “lab3”. Put your ARM DS project folder under “lab3/code”. Put the `p3_report.pdf` under the “lab3” directory. Include a README file with a description of directory contents. Put the README file under the “lab3” directory. Zip everything inside the lab3 directory and submit it to Learn Lab3 Dropbox.

¹For example, you may want to create linked list data structure functions or some helper functions. You may want to create new files to hold these functions for better file organization.

4.5 Marking Rubric

Points	Sub-Points	Description
90		Source Code
	10	Code compilation
	80	Third-party testing Manual code inspection
10		Report

Table 4.1: Lab3 Marking Rubric

The Rubric for marking the submitted source code and report is listed in Table 4.1. The functionality of your implementation will be tested by a third-party test program. A minimum of **20 points** will be deducted if messages are lost or mysterious appear. A minimum of **35 points** will be deducted if the system console I/O does not function at all. Your grade will be relative to the amount of error the third-party testing program has identified.

4.6 Errata

1. In page 34, new clarifications are added to the `k_send_msg` function.
2. In page 38, `p2_report.pdf` and Chapter 3.4.1 are changed to `lab3_report.pdf` and Chapter 4.3.1 function.

Chapter 5

Lab4 Real-time Tasks

In this lab you will work on adding timing services to the RTX. In particular, you will design and implement system calls to create periodic real-time tasks. You will also design and implement the earliest-deadline-first (EDF) scheduler to schedule real-time tasks. After this lab, you will learn:

- How to work with timers to provide timing services, and
- How to design and implement periodic, real-time tasks.

5.1 Starter Files

The starter files are uploaded on GitHub at <http://github.com/yqh/ece350/> under the `manual_code/lab4` directory. The directory contains RT sub-directory which is a bare-metal ARM DS skeleton project for lab4. The sub-directory follows the same organization as in lab1, lab2, and lab3. The main change is the addition of the timers in `src/DE1_SoC_A9/timer.[c|h]` and modification of the interrupt handler code in `HAL_CA.c` file. Similar to lab3, the interrupt handler is invoked when an interrupt happens. In lab4, in addition to the UART interrupt, we support timer interrupts from timer 1 and timer 2. Similar to lab3, interrupts are disabled by default in the SVC mode and enabled in the USR mode. In this lab, you will need to modify the interrupt handler to meet the requirements outlined below. Similar to previous labs, please see Section 5.3.4 for more details on what you can and cannot modify.

The starter code supports three different timers—two HPS timers and one A9 private timer (see Section 2.4 in [2]). Each cycle takes 10 nsec for the two HPS timers and 5 nsec for the A9 private timer. The two HPS timers could be set to run in two different modes—user-defined count mode and free-running mode. In user-defined count mode, the timer starts from a user-defined value and counts down every cycle. In the free-running mode, the timer starts from `0xFFFFFFFF` and counts down every cycle. If the interrupt mask bit of any of the HPS timers is set to zero, that timer

raises an interrupt when its counter reaches zero.

The A9 private timer also has two running modes—auto-loading mode and one-time mode. In auto-loading mode, the user-defined value is loaded to the counter every time the counter reaches zero. In the one-time mode, the timer stops running once the counter reaches zero. Additionally, the A9 private timer has a prescaler field (8 bits) that can be used to slowdown the counting rate. The timer counter decrements every (prescaler + 1) cycles. With prescaler set to 0, the A9 private timer decrements every clock cycle. If prescaler is set to 1, the A9 private timer decrements every second clock, and so on. Similar to HPS timers, if the interrupt bit of the A9 timer is set to one, the timer raises an interrupt when the counter reaches zero.

The simple test case in the starter code uses the A9 private timer and one HPS timer (both initialized in `k_rtx_init`). The A9 private timer is set up in the auto-loading mode to decrement from ~~0xFFFFFFFF~~ `0xFFFFFFFF` every 1 usec (counter reaches zero every about 1.2 hrs). The interrupt bit of the A9 private timer is set to zero so that it does not raise an interrupt when the counter reaches zero. The HPS timer 0 is set up in the user-defined count mode to decrement from `10000` every 10 nsec. The interrupt mask bit of the timer is set to zero so that it raises an interrupt every 100 usec.

Apart from the null task, there are two user tasks and one kernel task. The kernel task, in an infinite loop, prints some data and yields the processor. The two user tasks have infinite loops and do not yield the processor by themselves. The interrupt handler is coded to force a context switch every time a key is pressed. Note that interrupts (UART0 and HPS timer 0) are disabled in SVC mode. On HPS timer 0 interrupts, the interrupt handler is coded to use the A9 private timer count to calculate the time that has passed from the previous interrupt. If the elapsed is more than 500 msec, the interrupt handler prints the elapsed time.

5.2 Pre-lab Preparation

- Review the lecture notes on real-time systems,
- Read Section 2.4 of [2] (short read but an important one),
- Skim through Chapter 24 of [1] (read at least the first four pages),
- Run the lab4 starter code on the actual board (the simulator does not support interrupts), and
- Work through the modified `IRQ_Handler` code and understand how it is different from lab3.

5.3 Assignment

You should submit a programming project and a report. Your report should document the data structures, algorithms (if applicable), and test scenarios that you develop for your programming project.

5.3.1 Overview

The first objective is to add support for timing services. The RTX will provide a system call to create periodic real-time tasks that execute the same job at regular intervals. The RTX will also provide a system call to suspend the calling task for a specified period of time. The second objective is to make mailboxes more predictable. In particular, for real-time tasks, the RTX can create mailboxes that only receive predefined messages. Additionally, the RTX will provide tasks with a non-blocking system call to receive a message from their mailbox. The third objective is to make the RTX support EDF scheduling policy for real-time tasks.

5.3.2 Programming Project

- The RTX supports both real-time and non-real-time tasks.
- The priority level of the NULL task is 255 (PRIO_NULL).
- The priority level of normal, non-real-time tasks is between 1 and 254.
- The priority level of real-time tasks is 0 (PRIO_RT).
- Similar to the previous labs, smaller priority levels indicate higher priorities (i.e., 0 is the highest priority, and 255 is the lowest priority).
- Real-time tasks are scheduled using EDF scheduling policy.
- When there is no runnable real-time task, the scheduler starts scheduling runnable non-real-time tasks using the policy outlined in lab2 (and lab3).

Macros and Data Structures

For this project, the relevant macros are as follows.

```
/* Task priorities */
#define PRIO_RT    0      /* Priority level for real-time tasks */
#define PRIO_NULL 255    /* hidden priority for null task */

/* Task states */
#define DORMANT    0      /* Terminated task state */
#define READY     1
```

```
#define RUNNING 2
#define BLK_MSG 4
#define SUSPENDED 5 /* Suspended task */
```

RTX also has a time value structure `struct timeval_rt`.

```
struct timeval_rt {
    U32 tv_sec; /* seconds */
    U32 tv_usec; /* microseconds */
} TIMEVAL;
```

This data structure is used to account for time. For example, to capture 19.05 seconds in this structure, we set the `tv_sec` field to 19 and set the `tv_usec` field to 50000. New fields in the `rtx_task_info` structure that are used for this lab are as follows.

```
typedef struct rtx_task_info { /* fields defined in previous labs are not
    shown */
    TIMEVAL p_n; /* period in seconds and microseconds */
    size_t rt_mbx_size; /* real-time task's mailbox size */
} RTX_TASK_INFO;
```

Description of Functions

Real-time Task Creation Function

NAME

`k_tsk_create_rt` - create real-time task

SYNOPSIS

```
#include "k_rtx.h"

int k_tsk_create_rt(task_t *tid, TASK_RT *task);
```

DESCRIPTION

The `k_tsk_create_rt()` is a non-blocking function that creates a new real-time task. The priority of the new task is set to `PRIOR_RT`. The `task` argument defines the real-time task parameters.

```
typedef struct task_rt {
    TIMEVAL p_n; /* Period = relative deadline */
    void (*task_entry)(); /* Entry address */
    U16 u_stack_size; /* User-space stack size in bytes */
    size_t rt_mbx_size; /* Size of mailbox */
} TASK_RT;
```

The `p_n` argument specifies the period (and relative deadline) of real-time tasks. The time specified by `p_n` argument has to be a multiple of 500 microseconds. The `task_entry` argument points to the beginning of the code that the task will start its execution from. The `u_stack_size` argument specifies the user-space stack size in bytes. Once allocated, the owner of the user-space stack is the RTX. As before, the RTX is responsible for freeing the stack space once the task terminates. If `mbx_size` is not zero, then the RTX creates a mailbox for the real-time task with capacity of `mbx_size` bytes. Before returning, a successful call to `k_tsk_create_rt()` stores the task ID of the newly-created task in the buffer pointed to by `tid`.

RETURN VALUE

The function returns 0 on success and -1 on failure. Possible causes of failures are listed below.

- The time specified by `p_n` is not a multiple of 500 microseconds.
- The number of tasks has reached its maximum in the system.
- The requested user-space stack size is too small (or more than available free memory).
- There is not enough memory to allocate to the mailbox.
- The requested period is zero (second and microsecond).

This is not a complete list of failure causes. Your implementation should cover all other possible failure causes when you design your RTX.

Please note that similar to Lab 3, tasks only can have one mailbox. You will need to make few modifications to your code from previous labs:

- Modify your code to support priority leveles from 1 to 254;
- Modify the `k_rtx_init` function to allow real-time tasks to be created during system initialization;
- Modify the `k_tsk_get` function to include real-time tasks' period in the returned information;
- Modify the `k_tsk_set_prio` function so that it does not allow a non-real-time task's priority to be changed to `PRIO_RT`; and
- Modify the `j_tsk_set_prio` function so that it does not allow a real-time task's priority to be changed.

Done Function

NAME

`k_tsk_done_rt` - suspend a periodic task until its next period

SYNOPSIS

```
#include "k_rtx.h"

void k_tsk_done_rt(void);
```

DESCRIPTION

This function can only be used by real-time tasks to indicate the end of the execution of one job. When the function is called, the RTX resets the calling task's stack to its starting address. The RTX also resets the task's program counter to its `task_entry`. If the finished job has met the task's deadline, then the RTX suspends the task (i.e., sets the task's state to `SUSPENDED`). It is the responsibility of each task to free its allocated memories before calling `tsk_done_rt`. When the task's next period starts, the RTX wakes up the task by setting its state to `READY`. When the task runs again (i.e., new job of the task), it start from its original `task_entry`.

RETURN VALUE

The function does not return.

Important note. In this lab, the finest time granularity of the RTX should be 100 usec. This means that for the RTX, time is divided into 100-usec time quanta. For example, suppose that real-time task A with period 500 usec is created at some point in the middle of the 50th time quantum. In this case, the deadline for jobs A.1, A.2, A.3, ..., A.N will be the end of the 55th, 60th, ..., (50 + 5 × N)th time quantum, respectively. To implement this feature, you will have to use the hardware timers. You have to remember that interrupts are disabled in the SVC mode (system call handling or running kernel tasks). To determine the time that is spent in the SVC mode, similar to the starter code, you could use the A9 timers.

The RTX should keep track of the number of jobs that have finished for each real-time task and their deadlines. If a job exceeds its deadline, the RTX does not preempt the job and waits for the `tsk_done` to be called by that job. Once `tsk_done` is called, RTX checks to see if the deadline is missed. If the j-th finished job of a real-time task with TID i misses its deadline, then an error message of "Job XX of task YY missed its deadline" is sent to the **JTAG** UART port (~~Putty not JTAG UART~~), where XX is j and YY is i. When the RTX is overloaded, a real-time task might not be able to run any of its jobs during multiple of its periods. In such a case, the error messages are only shown after each job runs. This way, the RTX allows tasks to ~~cache~~ **catch** up with their periods once the load on the system winds down.

Real-time tasks can be created by both real-time and non-real-time tasks. If a non-real-time task creates a real-time tasks, then the newly-created real-time task will preempt the currently-running non-real-time task (because the priority of real-time tasks is higher than that of non-real-time tasks). The preempted non-real-time task is added to the front of the ready queue (i.e., sorted first among other

tasks with the same priority). If a real-time task creates another real-time task, then the preemption happens based on EDF scheduling (among real-time task, the scheduler always runs the one with earliest deadline).

Please note that real-time tasks could call `tsk_exit`. For example, a real-time task could be programmed to call `tsk_exit` if it receives a specific message. In this case, the real-time task is terminated (i.e., the state of the task becomes DORMANT), and the RTX no longer wakes up the task at task's regular periods. **Please note that when a real-time task calls `tsk_exit`, the RTX checks for the deadline of the current job and prints an error message if the deadline is missed.** Real-time tasks could also call other functions that were outlined in lab2 and lab3 (e.g., `tsk_yield` or `set_prio`). This means that real-time tasks could potentially be blocked. In our test cases, however, real-time tasks only call non-blocking system calls.

Task Suspend Function

NAME

`k_tsk_suspend` - suspend the calling task for a specified time

SYNOPSIS

```
#include "k_rtx.h"

void k_tsk_suspend(TIMEVAL *tv);
```

DESCRIPTION

The `k_tsk_suspend()` function suspends the calling task for the time specified by `tv`. The time specified by `tv` has to be multiple of 500 usec. The state of the task is set to `SUSPENDED`. The kernel picks another ready-to-run task to execute after the calling task is suspended. After `tv` time has elapsed, the RTX wakes up the suspended task by setting its state to `READY`. **If the priority of the newly-unsuspended task is strictly higher than that of the currently-running task, then the newly-unsuspended task runs immediately, and the currently-running task is added to the front of the ready queue. Otherwise, the newly-suspended task is added to the back of the ready queue.** If `tv` indicates zero seconds and zero microseconds, then the function returns immediately without suspending the calling task. The `k_tsk_suspend` can be called by both real-time and non-real-time tasks.

RETURN VALUE

The function does not return any value. If the time specified by `tv` is not a multiple of 500 usec, the function returns immediately without suspending the caller.

As mentioned before, the finest granularity of the RTX is 100 usec. If task B calls `suspend` at some point in the middle of the 20th time quantum to be suspended for 1 ms, it should remain suspended until the end of the 30th time quantum.

Non-blocking Receive Message Function

NAME

`k_recv_msg_nb` - receive a message , non-blocking

SYNOPSIS

```
#include "k_rtx.h"

int k_recv_msg_nb(task_t *sender_tid, void *buf, size_t len);
```

DESCRIPTION

This function can be called by real-time and non-real-time tasks. The specification of this function is similar to the `k_recv_msg` function outlined in lab3. The only difference is that the task calling `recv_msg_nb` receives a message from its mailbox if there are any and returns if there are none (i.e., it does not get blocked if its mailbox is empty).

RETURN VALUE

The `k_recv_msg_nb` function returns 0 on success and -1 on failure. Please see the section on `k_recv_msg` function in lab3 for possible causes of failure.

Testing Tasks

In order to test your implementation of the required functions, write an application that uses your kernel primitives. There is no hard requirement on what tests to be implemented. The rule of thumb is that the tests should be comprehensive enough to convince you that your implementation is correct.

5.3.3 Report

Write the following items in your report and name it `p4_report.pdf`.

- Descriptions of the data structures and algorithms used to implement all the functions listed in Chapter 5.3.2; and
- Testing scenario descriptions.

To illustrate key algorithms, pseudocode is acceptable. For testing, include three or more non-trivial testing scenarios.

5.3.4 Third-party Testing and Source Code File Organization

We will write third-party test cases to verify the correctness of your implementations. In order to do so, you will need to maintain the file organization of the project skeleton in the starter code. There are dos and don'ts you need to follow.

Don'ts

- Do not change any file in `src/board` directory;
- Do not move any file from the `src` directory to any other directories;
- Do not change the file names under the `src` directory;
- Do not make any changes to the contents of the `rtx.h` and `common.h` files;
- Do not change the existing function prototype in the given `k_mem.[ch]`, `k_task.[ch]`, and `k_msg.[ch]` files;
- Do not include any new header files in the `app/ae_mem.c` file;
- Do not modify the `ae.h` file; and
- Do not change the `ae.c` file except the body of `ae_set_task_info` function.

Dos

- You are allowed to include the `timer.h` file (in `src/board/DE1_SoC_A9` folder) in any kernel file you need.
- You are allowed to add new self-defined functions to `k_mem.[ch]`, `k_task.[ch]`, and `k_msg.[ch]`.
- You are also allowed to create new `.h` and `.c` files ¹.
- The newly-created `.h` file is allowed to be included in the `k_mem.c`, `k_task.c`, and `k_msg.c` files.
- Any new files you add to the project can be put into either the `src` directory or other directories you will create.
- You are allowed to change the body of functions in `k_task.c` without changing the prototype defined in `k_task.h`.

Please note that the `main_svc_cw.c` calls third-party testing by calling `ae_init` function which then calls `ae_set_task_info` function that implements the third-party testing software. The function prototypes of `ae_set_task_info` function does not change. But the implementation of it may change in real testing. During the third-party testing, the files under the `app` directory will be replaced by more complicated testing cases than the ones published on GitHub.

¹For example, you may want to create linked list data structure functions or some helper functions. You may want to create new files to hold these functions for better file organization.

5.4 Deliverables

5.4.1 Post-Lab Deliverables

Create a directory named “lab4”. Then create a sub-directory named “code” under “lab4”. Put your ARM DS project folder under “lab4/code”. Put the `p4_report.pdf` under the “lab4” directory. Include a README file with a description of directory contents. Put the README file under the “lab4” directory. Zip everything inside the lab4 directory and submit it to Learn Lab4 Dropbox.

5.5 Marking Rubric

Points	Sub-Points	Description
95		Source Code
	5	Code compilation
	10	Performance of scheduler
	80	Third-party testing Manual code inspection
5		Report

Table 5.1: Lab3 Marking Rubric

The Rubric for marking the submitted source code and report is listed in Table 5.1. The functionality and **performance** of your implementation will be tested by a third-party test program. The performance of your scheduler (i.e., the scheduling overhead) will be measured for third-party test programs. For performance, we grade groups relative to each other. Your grade will be relative to the amount of error the third-party testing program has identified.

5.6 Errata

1. On page 42, `0xFFFFFFFF` was changed to `0xFFFFFFFF`.
2. On page 43, `sing` was changed to `using`.
3. On page 46, `cache` was changed to `catch`.
4. On page 46, the JTAG UART is specified as the right port for printing missed-deadline error messages.
5. On page 47, a new requirement is added for RT tasks that call `tsk_exit`.

6. On page 47, some new detail is added for scheduling tasks calling `k_tsk_suspend`.

Part III

Frequently Asked Questions

Part IV

Computing and Software Development Environment Quick Reference Guide

Chapter 6

Windows 10 Remote Desktop

The lab machines are accessible by windows 10 remote desktop. You will need to be on the campus virtual private network (VPN) first. The <https://uwaterloo.ca/information-systems-technology/services/virtual-private-network-vpn> gives detailed instructions on how to connect to the campus VPN. If you are in China, a special instruction can be found at <https://wiki.uwaterloo.ca/display/ISTKB/Accessing+Waterloo+learning+technologies+from+China+using+special+VPN>.

The Englab at <https://englab.uwaterloo.ca/> is the main gateway. Choose **ECE** → **ece-cpuio*** or **ECE** → **ece-public*** machines. When prompt for user name, input `Nexus\userid`, where the `userid` is your quest ID. The password is your Quest password. Then you are connected to one of the lab machines that as the software and hardware installed for this lab.

Please be advised that if you are idle on a lab machine for an extended period of time, your session will automatically times out and your account will be locked from using this computer for a period of time. While your account is locked for a machine, you may still be able to login onto the machine. But most of the software installed on the machine will become inaccessible.

Once you finish using the lab computer, remember to close all your programs and logout from the remote desktop session.

Chapter 7

Software Development Environment

The ARM Development Studio (DS) is used for the Intel DE1-SoC board development. The ARM DS includes the following:

- An eclipse-based IDE which combines the project manager, source code editor and program debugger into one environment;
- ARM compiler, assembler, linker and utilities;

The ARM DS requires a license. The license information can be found in Learn. If you want to install the software on your own computer. You need to download the Arm Development Studio for Intel SoC FPGA from <https://fpgasoftware.intel.com/armds/> and select release 2020.0. You will then need to configure your license manager to add the University of Waterloo license server information provided in Learn.

7.1 Getting Started with ARM DS

To get started with the ARM DS IDE, the ARM Development Studio User Guide at <https://developer.arm.com/documentation/101470/2000/?lang=en> is a good place to start. We will walk you through the IDE by creating lab1 skeleton project using the provided skeleton source code on GitHub.

7.2 Creating a ECE350 Workspace Structure

We notice some problem with ARM DS saving projects on P: Drive. So your should be on N: drive if you want to access your workspace from any one of the lab machines. Create a ECE350 folder on N: Drive as our workspace. Create a sub-folder under the ECE350 folder and name it `starter`. Create another sub-folder under the ECE350 folder and name it `mycode`.

7.3 Getting Starter Code from the GitHub

The ECE 350 lab starter github is at <https://github.com/yqh/ece350>. pen up Git Bash terminal, change directory to the N:\ECE350LAB\starter directory. Clone the lab material repository by using the following command:

```
git clone https://github.com/yqh/ece350
```

7.4 Start the ARM DS

The ARM DS IDE 2020.0 shortcut should be accessible from the start menu on school computers. If not, then go to Program Files folder on C Drive and navigate to the Arm\Arm Development Studio 2020.0\bin folder. Find the armds_ide.exe and double click it.

The first time you start the ARM DS. You will encounter the following events:

- A license setup pop up window asks you to configure the license. Just press next and then finish button.
- The ARM DS will update installed package and it usually takes long, you can let it run in the background. You may see some error messages in red saying certain URLs are not accessible, ignore them. Note this update is on a daily basis. So every day the first time you open up ARM DS on a lab machine, this process repeats.
- You will also see a firewall warning pop up window to ask for granting access, just click the cancel button.

7.5 Create a New Empty C Project

When you see the ARM DS splash screen, you are in the default work space of `C:\Users\<user_id>\Development Studio Workspace\`. Let's switch the workspace to `N:\ECE350\mycode`. Select **File** → **Switch Workspace** → **Other** to bring up the workspace selection window. Select `N:\ECE350LAB\mycode` as your ARM DS workspace and then press the launch button. You will see ARM DS IDE launches.

In the main menu, select **File** → **New** → **Project** (See Figure 7.1).

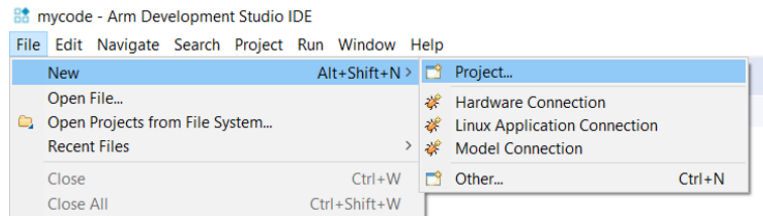


Figure 7.1: ARM DS IDE: Create a new project

In the New Project window, select **C/C++** → **C Project** → **Next** (See Figure 7.2).

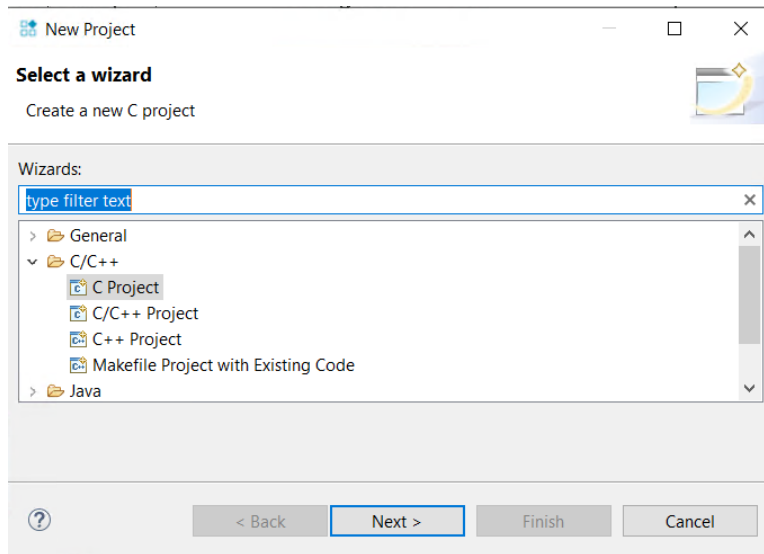


Figure 7.2: ARM DS IDE: Select a wizard

In the C project selection window, name the project **MySVC**. Leave the rest of the settings with default values. Press the **Finish** button (See Figure 7.3). You will see a new empty project MySVC show up in the project explorer (See Figure 7.4).

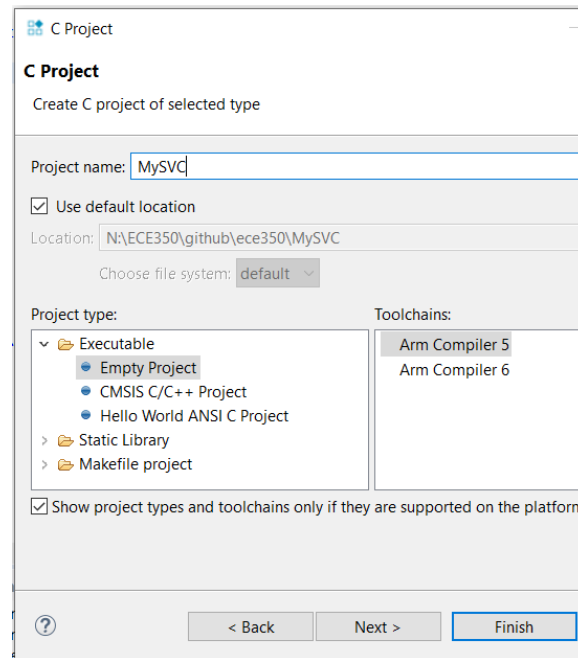


Figure 7.3: ARM DS IDE: Select a C project

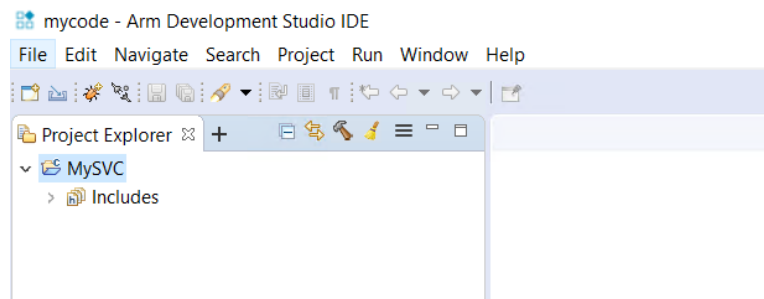


Figure 7.4: ARM DS IDE: Project explorer

7.6 Import Source Code

The starter code has a template folder which contains source code and documentation of lab1. We import these files from the file system by right click the project name and then select **Import** (see Figure 7.5).

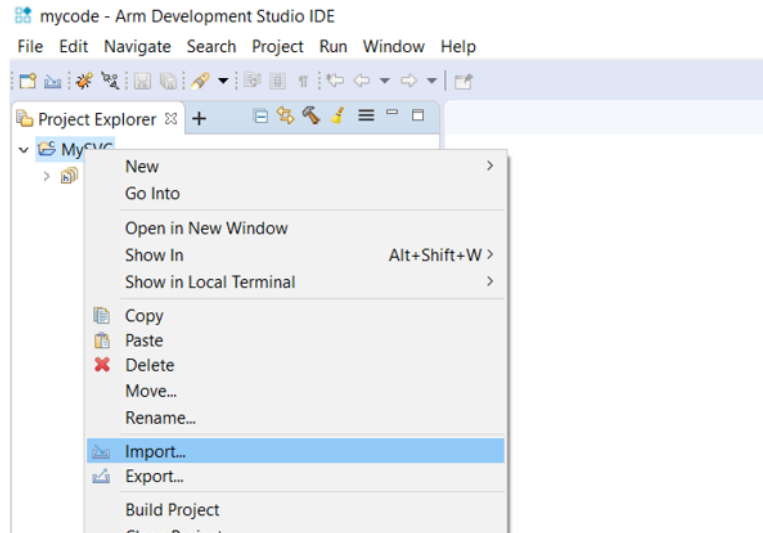


Figure 7.5: ARM DS IDE: Import Files

In the import select wizard window, select **General** → **File System** → **Next** as shown in Figure 7.6

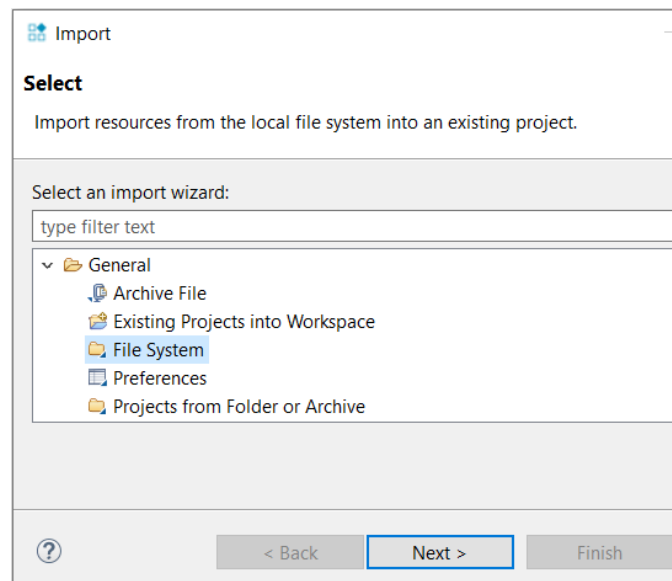


Figure 7.6: ARM DS IDE: Import select wizard

In the import file system window, press the **Browse** button to navigate to the

directory where we have the starter code template as shown in Figure 7.7. Select the check box of the template folder and leave the “Create top-level-folder” checkbox unchecked. Press **Finish** button.

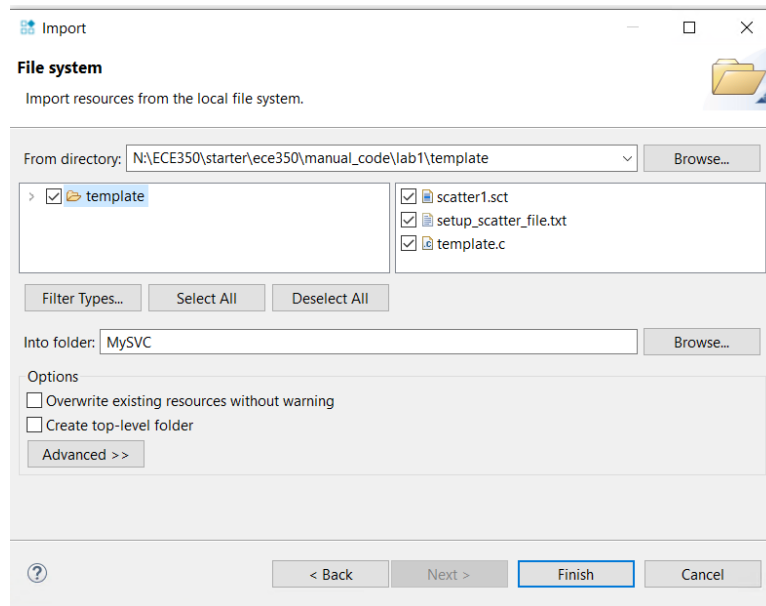


Figure 7.7: ARM DS IDE: Import file system

Your project should now look like what is shown in Figure 7.8.

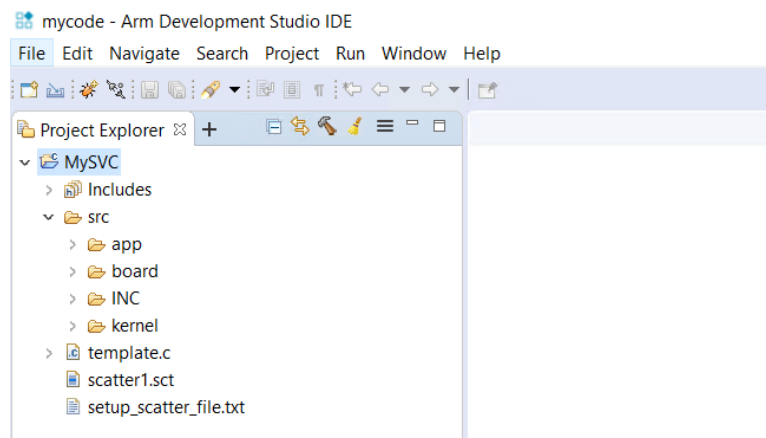


Figure 7.8: ARM DS IDE: MySVC project with template files

7.7 Setup Build Properties

Before we build the project, we need to configure the project C/C++ build. Select the MySVC project in the project explorer window and right click to bring up the context window. Select the **Properties**, which is the last item on the list. A project properties window pops up, select the **C/C++ Build** on the left side. Select the **Behavior** tab and check the **Enable parallel build** option (See Figure 7.9).

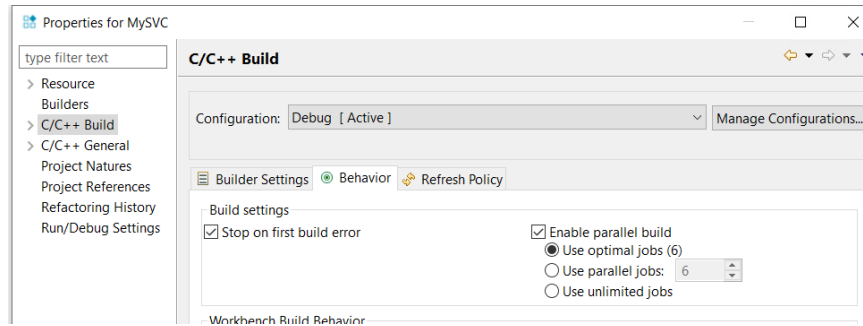


Figure 7.9: ARM DS IDE: MySVC project C/C++ Parallel Build

Expand the C/C++ Build item on the left side and select **Settings** → **All Tools Settings** → **Target** → **TargetCPU** → **Cortex-A9**. Check the **Interworking** option. See Figure 7.10 for details.

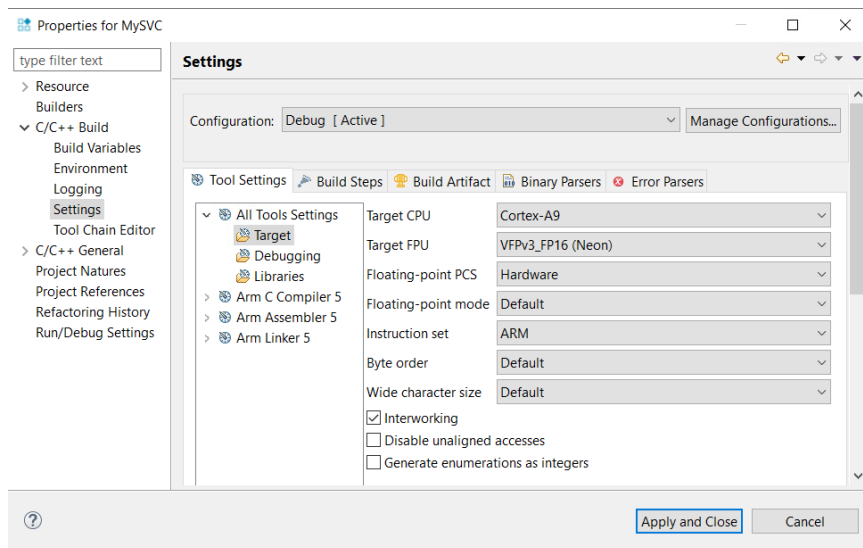


Figure 7.10: ARM DS IDE: MySVC project C/C++ Build Target Setting

Select **Arm C Compiler 5** → **Preprocessor** and add `DEBUG_0` macro (see Figure 7.11).

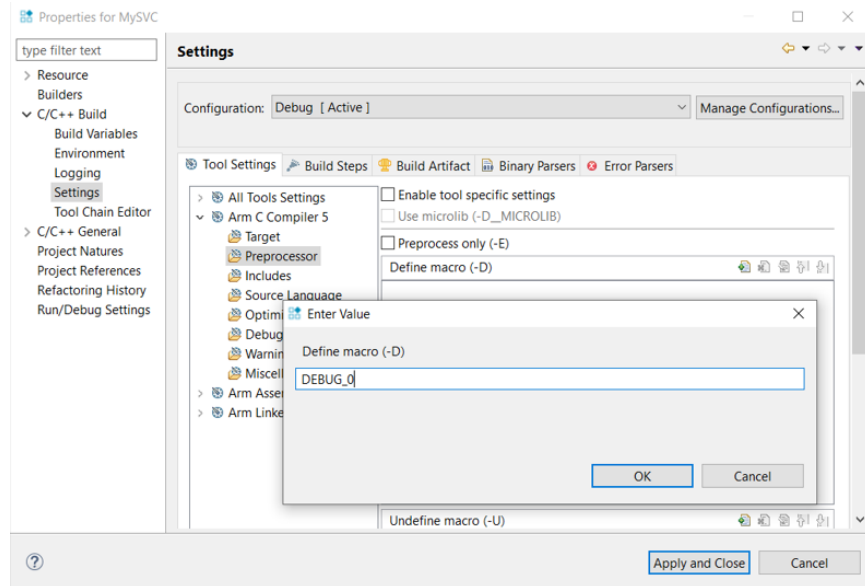


Figure 7.11: ARM DS IDE: MySVC project C/C++ Build Compiler Preprocessor Setting

Select **Arm C Compiler 5** → **Include** (see Figure 7.12) and press **Workspace** button to add include paths from workspace.

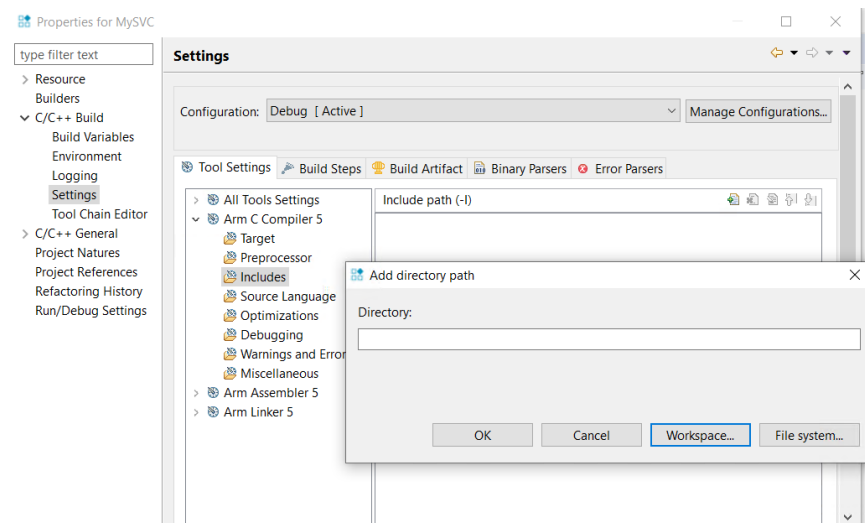


Figure 7.12: ARM DS IDE: MySVC project C/C++ Build Compiler Include Selection

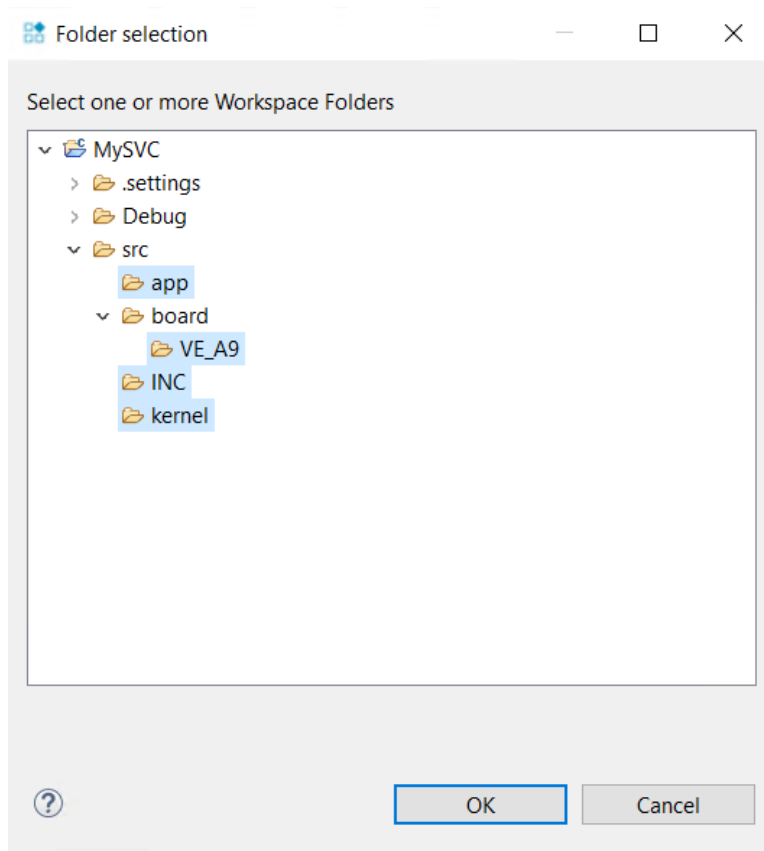


Figure 7.13: ARM DS IDE: MySVC project C/C++ Build Compiler Include Folder Selection

In the folder selection window, select all the folders as shown in Figure 7.13. Select **Arm C Compiler 5** → **Source Language** → **Source language mode** → **C99(–c99)** (see Figure 7.14).

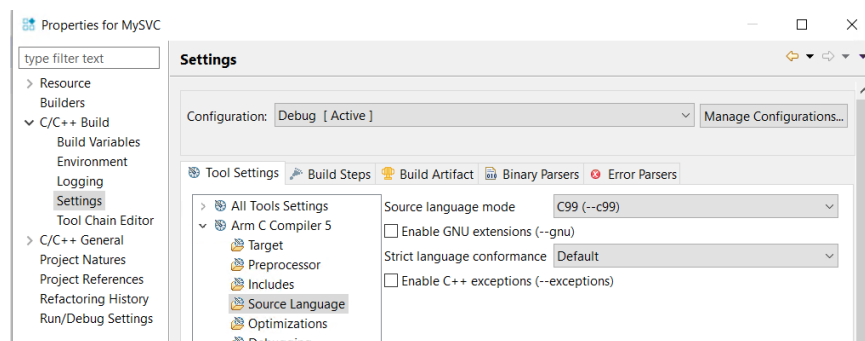


Figure 7.14: ARM DS IDE: MySVC project C/C++ Build Compiler Language C99

Select **Arm Linker 5** → **Image Layout**. Type `__Vectors` in the **Image entry point (–entry)** text field. Type `"${workspace_loc}/${ProjName}/scatter1.sct}"` in the **Scatter file (–scatter)** text field. See Figure 7.15 for details.

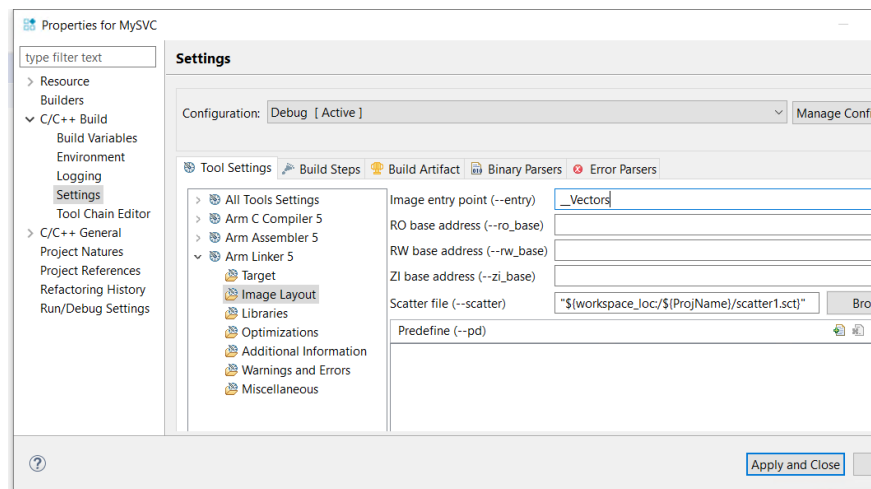


Figure 7.15: ARM DS IDE: MySVC project C/C++ Build Linker Image Layout

Finally, press the **Apply and Close** button to finish the project properties set up.

7.8 Build Project

To build the project, right click the project in the project explorer window and select **Build Project** (See Figure 7.16).

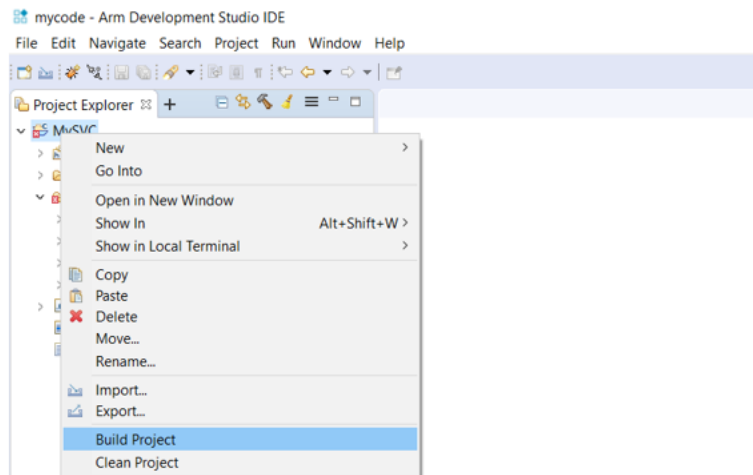


Figure 7.16: ARM DS IDE: Build Project

If the build is successful, you will see a new **Debug** folder in your project explorer window. Expand the contents, the `MySVC.axf` is the executable (See Figure 7.17).

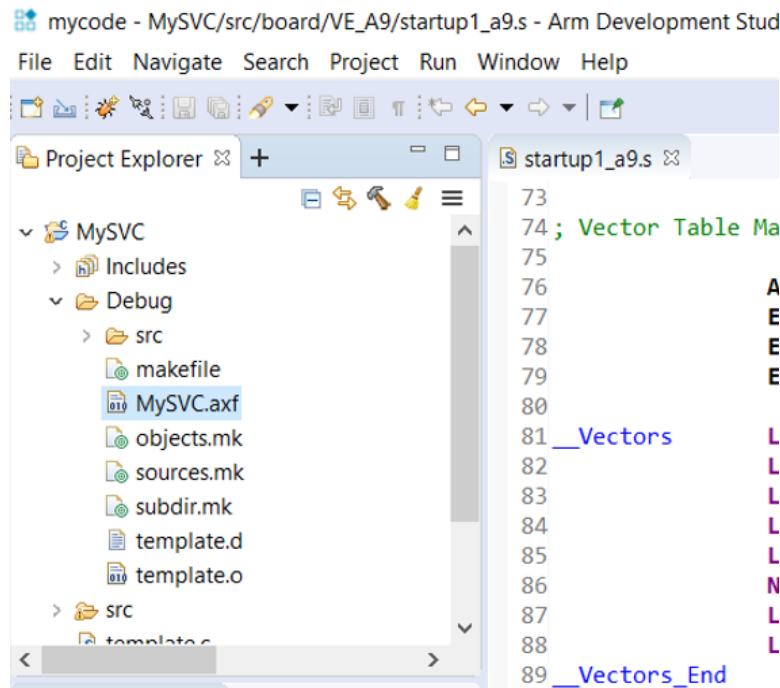


Figure 7.17: ARM DS IDE: Built executable in Debug folder

7.9 Create a Debug Connection of VE_Cortex_A9x1

To run our build, we need to create a debug connection. Specifically we will create a new model connection to use for the ARM Versatile Express Cortex A9 Fixed Virtual Platform. This allows us to run our program without the need of real hardware and is a great way to start the development independent from the hardware.

In the main menu, select **File** → **New** → **Model Connection** (See Figure 7.18).

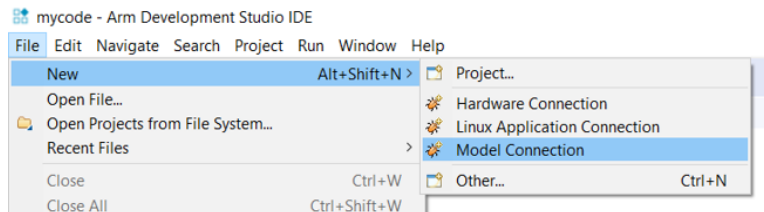


Figure 7.18: ARM DS IDE: Create a new model connection

In the debug connection window, select the MySVC project and give your debug connection a name, say “MySVC1” (See Figure 7.19). Note the debug connection name can be different from the project name. Press the **Next** button to bring up the

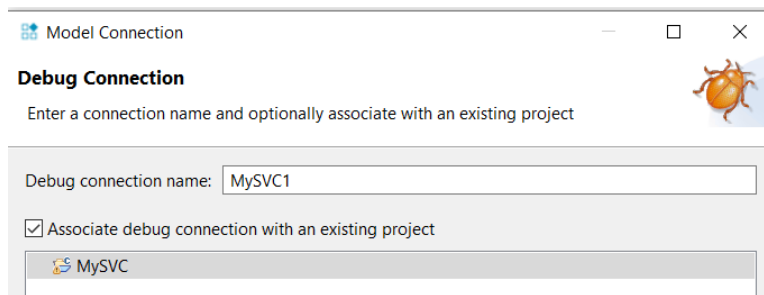


Figure 7.19: ARM DS IDE: Debug Connection Configuration

target selection window.

Select **Arm FVP (Installed with Arm DS)** (See Figure 7.20) and expand it.

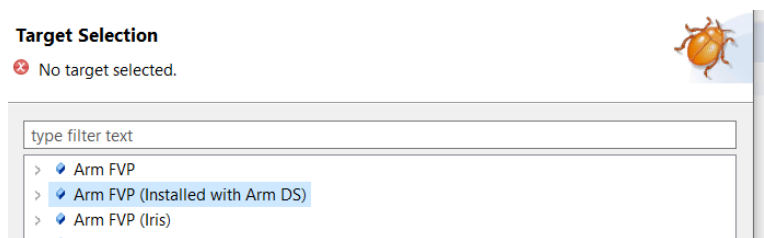


Figure 7.20: ARM DS IDE: Debug Connection Target Selection

Scroll down the list and select **VE_Cortex_A9x1** as shown in Figure 7.21.



Figure 7.21: ARM DS IDE: Debug Connection Target VE_Cortex_A9x1 Selection

In the Edit Configuration window, you should see the Connection tab shows **VE_Cortex_A9x1** → **Bare Metal Debug** → **Debug Cortex-A9** is currently selected (See Figure 7.22).

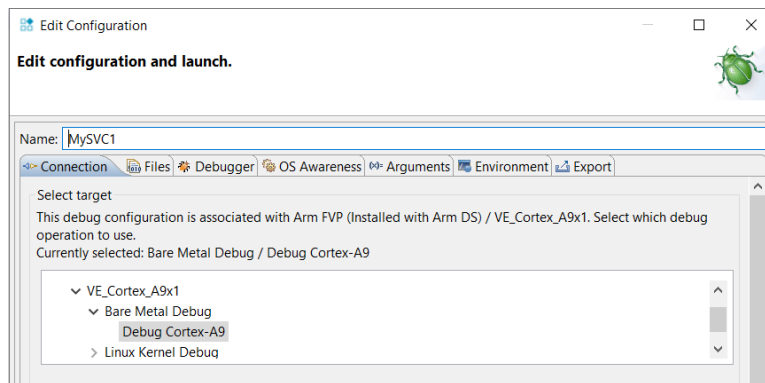


Figure 7.22: ARM DS IDE: Debug Connection Target VE_Cortex_A9x1 Selection

Select the **Files** tab and press the **Workspace** button for Application on host to download setting as shown in Figure 7.23.

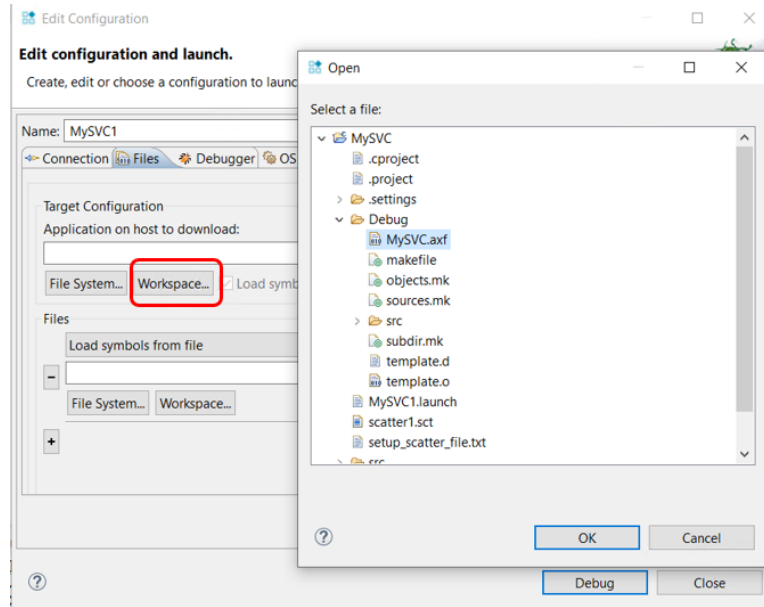


Figure 7.23: ARM DS IDE: Debug Connection Select Target to Download

Expand the **Debug** folder and select **MySVC.axf** as shown in Figure 7.23. Press the **OK** button. You will see some error message as shown in Figure 7.24.

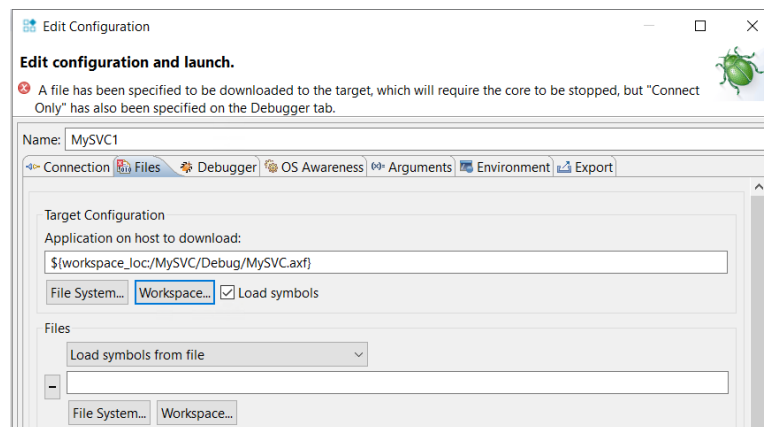


Figure 7.24: ARM DS IDE: Debug Connection Files Tab Setting Error

Select the **Debugger** tab and select **Debug from entry point** as shown in Figure 7.25. Press **Apply** to apply all the debug connection configurations we just made.

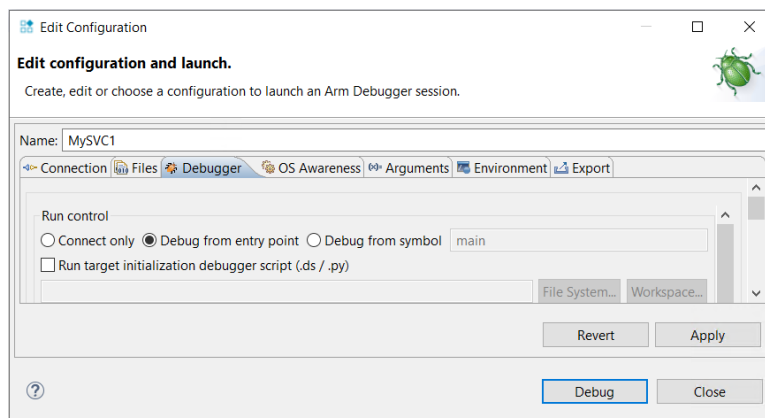


Figure 7.25: ARM DS IDE: Debug Connection Files Tab Setting Error

7.10 Import Project and Switch between Devices

First, download the project files to the computer. To open it with ARM DS-5 IDE, click on **File** → **Open Projects from File System** (See Figure 7.26).

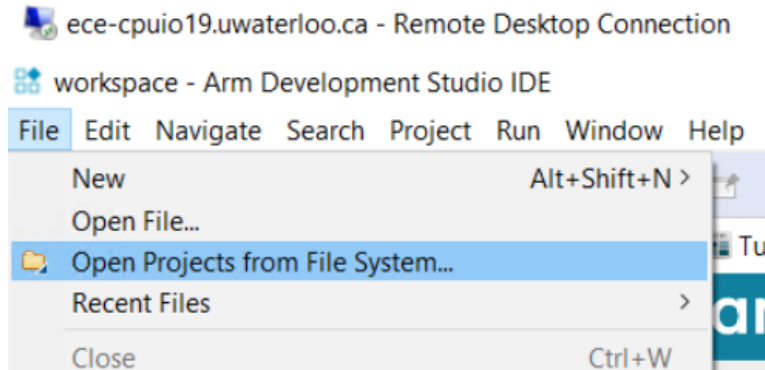


Figure 7.26: ARM DS IDE: Open Project from File System

Then, click on **Directory** (See Figure 7.27), browse and select your project directory. After that, click "Finish".

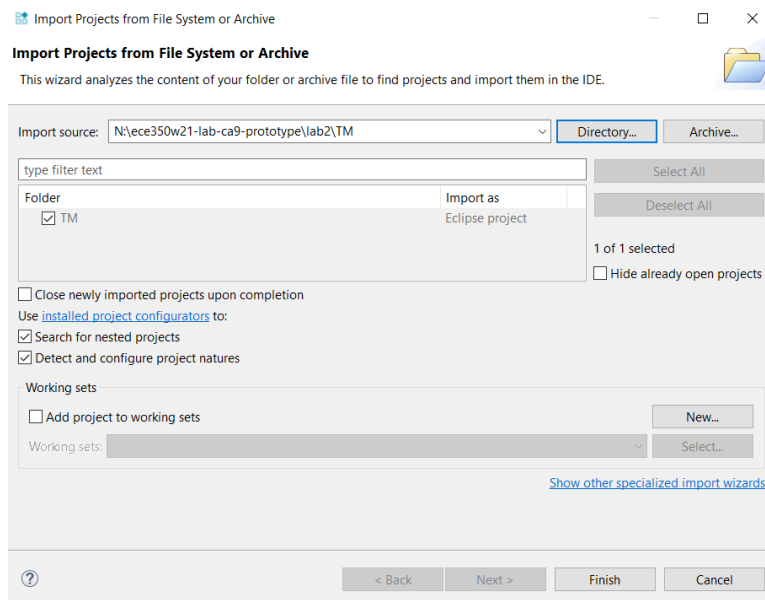


Figure 7.27: ARM DS IDE: Import Project from Folder

The IDE may prompt to rebuild the target, but it will fail unless you do the next steps.

7.10.1 Build for VE_Cortex_A9x1

First, you need to exclude the `board/DE1_SoC_A9` folder. To do this, right click on the folder and select **Resource Configuration** → **Exclude from Build** (See Figure 7.28).

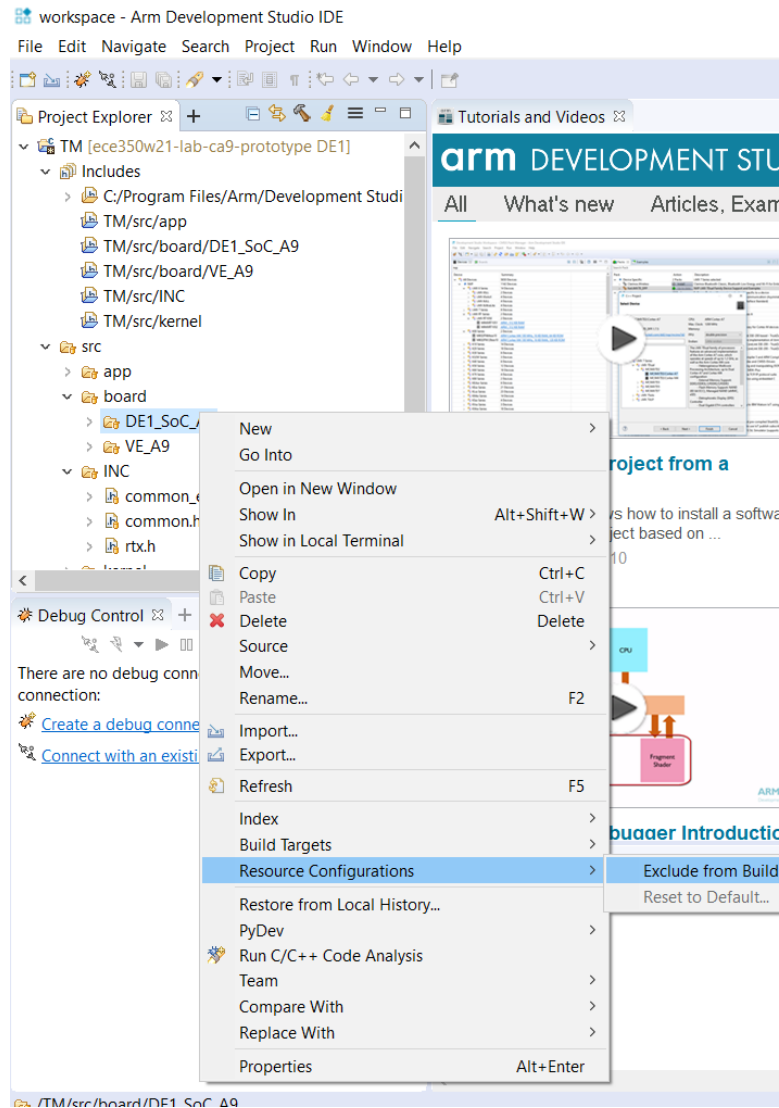


Figure 7.28: ARM DS IDE: Exclude Other Driver Files

Checking the box will exclude the folder and clear the box will include the folder (see Figure 7.29).

Then, follow the steps described in Figure 7.15. Set the scatter file to `scatter1.sct`.

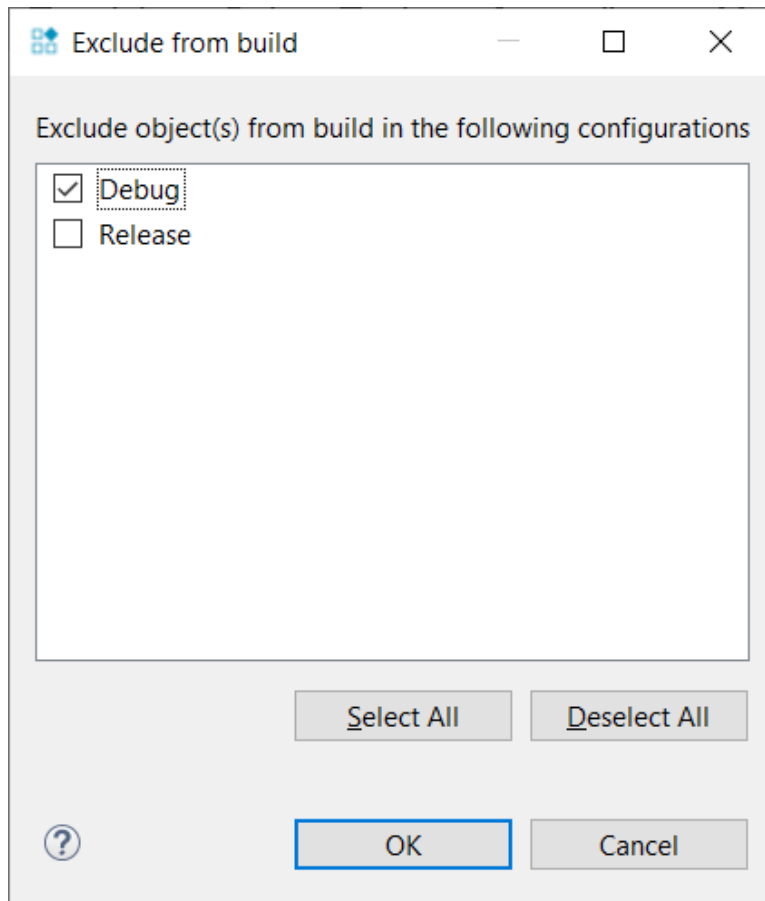


Figure 7.29: ARM DS IDE: Exclude or Include Driver Files

7.10.2 Build for DE1-SoC

Exclude the `board/VE_A9` folder and include the `DE1_SoC_A9` folder (refer to the steps outlined in [7.10.1](#)).

Then, follow the steps described in [Figure 7.15](#). Set the scatter file to `scatter_DE1_SoC.sct`.

7.11 Create a Debug Connection for DE1 SoC Board

First, when you create your project (see Figure 7.13), include `src/board/DE1-SoC` folder instead of `src/board/VE_A9` folder. Second, configure your build settings to use the scatter file for the DE1 SoC Board (`scatter_DE1_SoC.sct` included in the starter files). Next, to use a hardware device, you need to create a hardware connection. In the main menu, select **File** → **New** → **Hardware Connection** (See Figure 7.30). In the debug connection window, select the MySVC project and give

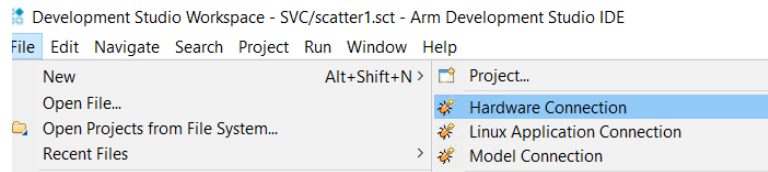


Figure 7.30: ARM DS IDE: Create New Hardware Connection

your debug connection a name, say “MySVC1” (See Figure 7.19). Note the debug connection name can be different from the project name. Next, select “Intel Cyclone V SoC (Dual Core)” as shown in Figure 7.31.

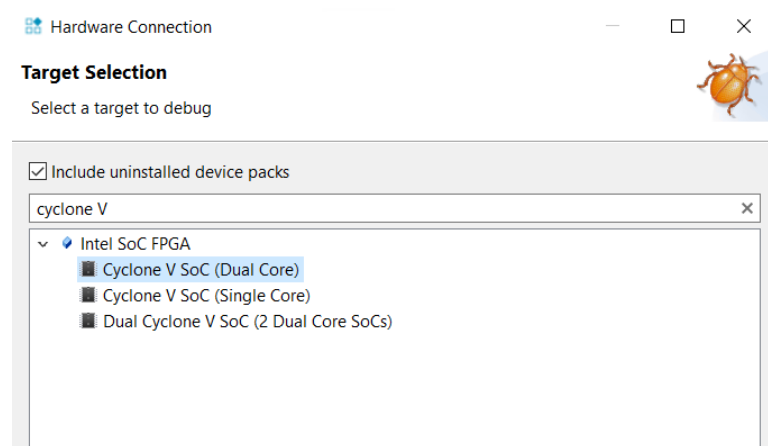


Figure 7.31: ARM DS IDE: Select Cyclone V (Dual Core) for DE1 SoC

Next, select “Debug Cortex-A9_0” as our target, and change “Target Connection” to USB-Blaster, as shown in Figure 7.32.

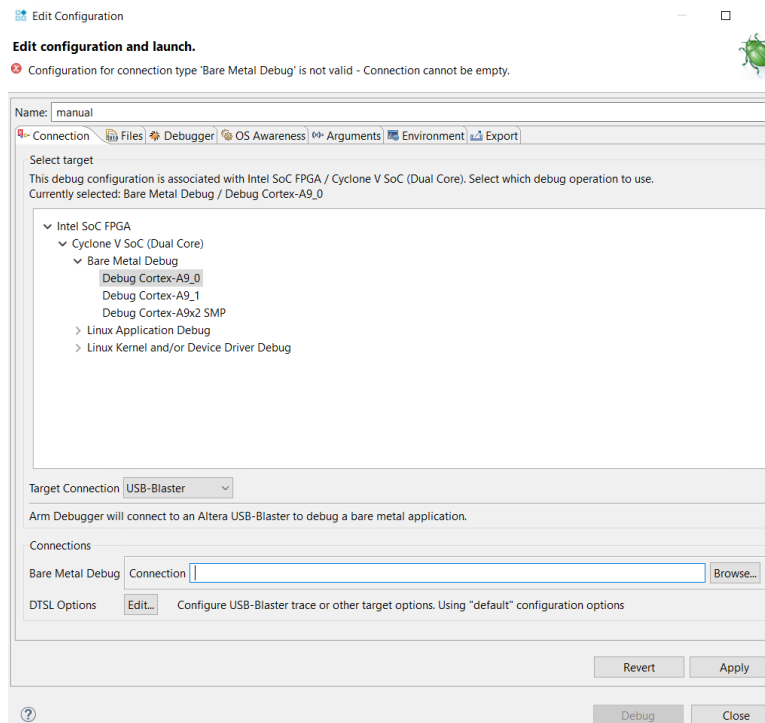


Figure 7.32: ARM DS IDE: Select Target for DE1 SoC

Click on the “Browse” button to set up the connection for our board, as shown in Figure 7.33.

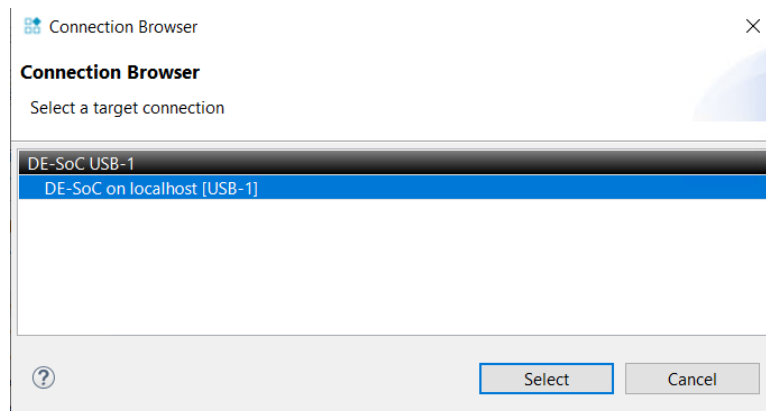


Figure 7.33: ARM DS IDE: Select Connection for DE1 SoC

Then proceed to the same steps to set up “Files” and “Debugger” tabs, as shown in Figure 7.23, 7.24, and 7.25.

7.12 Debug the Target

7.12.1 Debugging with VE_Cortex_A9x1

Press the **Debug** button (see Figure 7.25. You will see the FVP starts and the program will stop at its entry point (see Figure 7.34). In the Debug Control window, you have the usual debug control buttons such as continue, step into, step over, step out et. al.. Press the **Continue** button (green triangle icon) to let the program continue to execute. You should see the output in Figure 7.35.

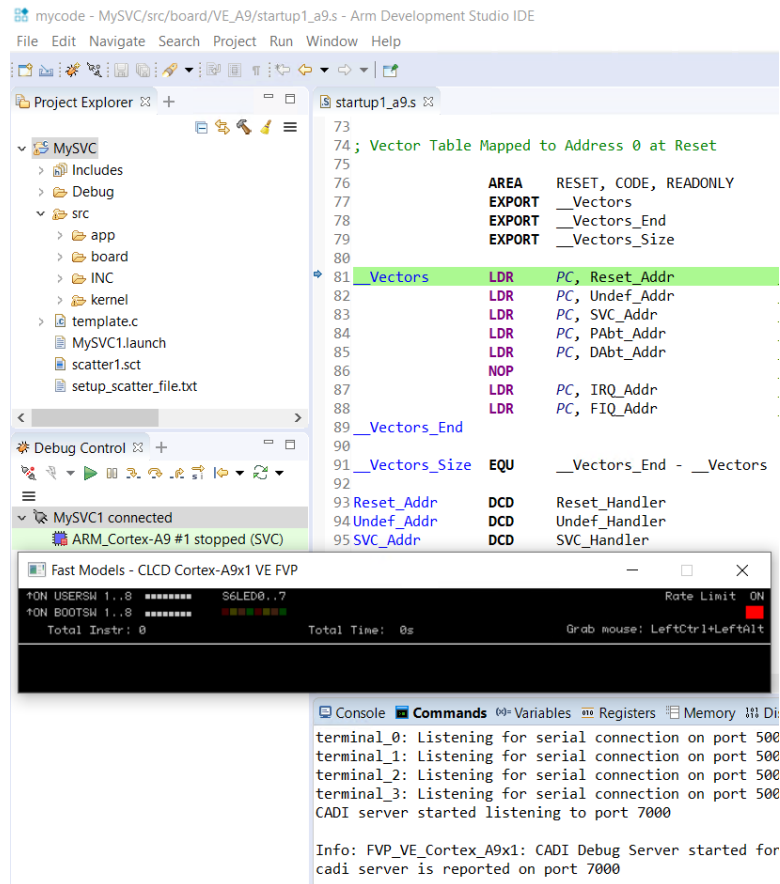


Figure 7.34: ARM DS IDE: Debug Start

```
Telnet localhost
mode = 0x13
mode = 0x1f
mode = 0x10
mode = 0x10
k_mem_init: image ends at 0x80302a60
k_mem_init: RAM ends at 0xffffffff
k_mem_alloc: requested memory size = 8
k_mem_alloc: requested memory size = 8
k_mem_dealloc: freeing 0x0
k_mem_extfrag: size = 128
k_mem_dealloc: freeing 0x0
k_mem_extfrag: size = 128
test result = 8
=====Task NULL=====
=====Task NULL=====
=====Task NULL=====
=====Task NULL=====
=====Task NULL=====
```

Figure 7.35: ARM DS IDE: Debug Output

7.12.2 Debugging with DE1 SoC

The steps to start running your program and debug it on the DE1 SoC board are very similar to what is outlined in Section 7.12.1 for VE_Cortex_A9x1. However, please note that you have to select the correct debug connection for DE1 SoC. To see the outputs of your program while running on the actual board, you will need to use the PuTTY application (search and start it from the Start Menu) to connect to the serial port. Select "Serial" and use COM3 as shown in Figure 7.36.

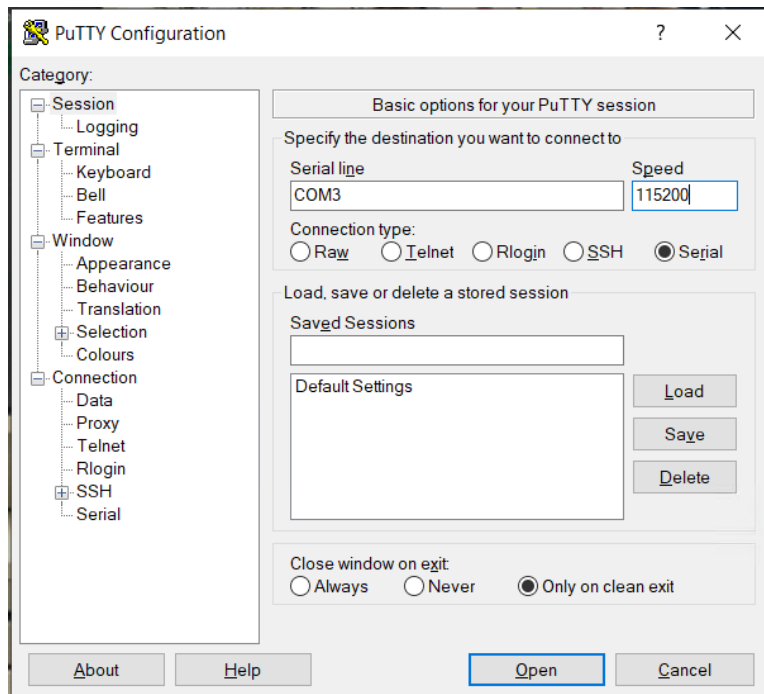
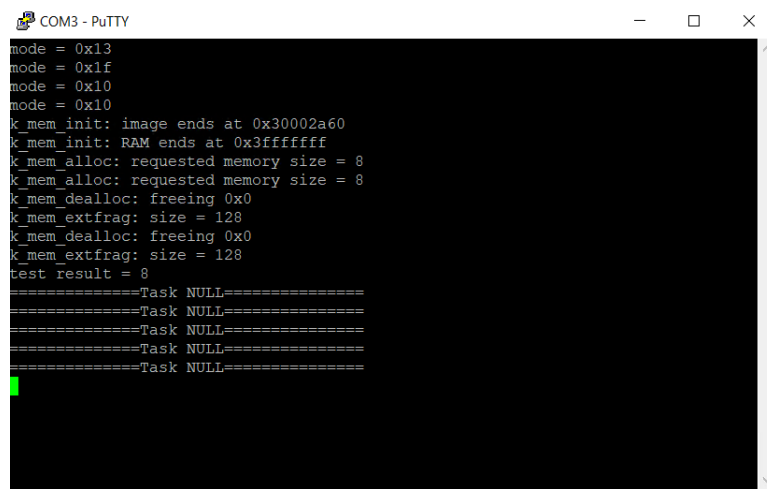


Figure 7.36: ARM DS IDE: PuTTY Setup

Please note that only one connection to the serial port is allowed at any given time. If everything is set up correctly, you should see outputs like Figure 7.37.

A screenshot of a PuTTY terminal window titled 'COM3 - PuTTY'. The window has a black background with white text. The text shows the output of an ARM DS IDE, including memory initialization and allocation details. The output is as follows:

```
mode = 0x13
mode = 0x1f
mode = 0x10
mode = 0x10
k_mem_init: image ends at 0x30002a60
k_mem_init: RAM ends at 0x3fffffff
k_mem_alloc: requested memory size = 8
k_mem_alloc: requested memory size = 8
k_mem_dealloc: freeing 0x0
k_mem_extfrag: size = 128
k_mem_dealloc: freeing 0x0
k_mem_extfrag: size = 128
test result = 8
=====Task NULL=====
=====Task NULL=====
=====Task NULL=====
=====Task NULL=====
=====Task NULL=====
```

A green cursor is visible on the line following the last 'Task NULL' output.

Figure 7.37: ARM DS IDE: PuTTY Outputs

7.12.3 Troubleshooting the DE1 SoC Board

The ARM cores on the DE1 SoC board might crash in some cases. If the IDE can-

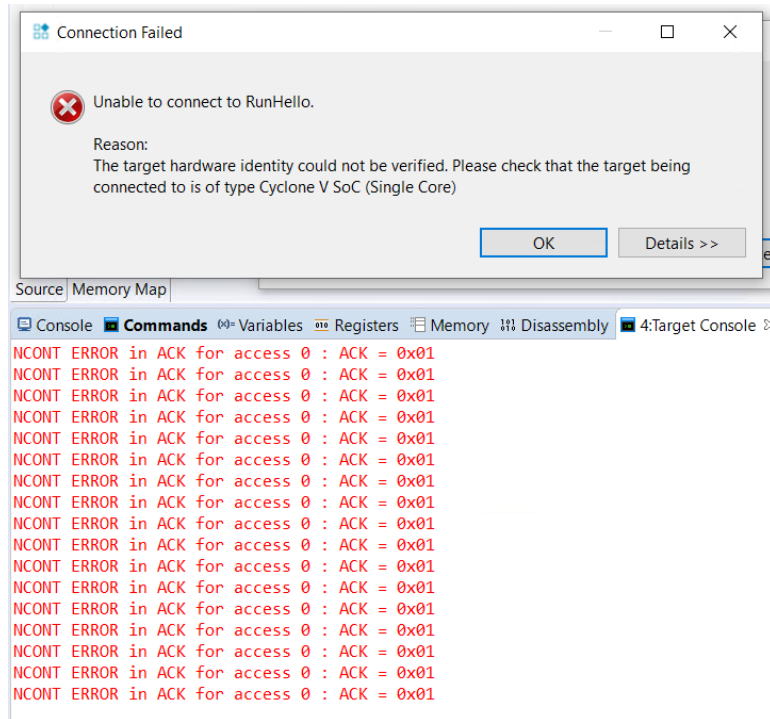


Figure 7.38: ARM DS IDE: Target Errors

not connect to the target and throws an error that indicates that the target cannot be identified (see for example Figure 7.38), you need to reset the ARM core and re-install the preloader. To do that, disconnect the hardware connection in ARM DS-5. Start Windows CMD in path C:\Software\Altera\20.1\quartus\bin64. Execute the following commands:

```
C:\n> cd C:\Software\Altera\20.1\quartus\bin64\n> quartus_hps.exe -c 1 -o GDBSERVER --gdbport0=3008 --preloader=PATH/TO/u-\n> boot-spl.del-soc.srec --preloaderaddr=0xfffff13a0
```

Please note that for `--preloader`, you need to provide a path to the `u-boot-spl.del-soc.srec` file from that is included in the lab starter files (<http://github.com/yqh/ece350/DE1-SoC>). This command will reset the processor, load the preloader (to configure the hardware including the SDRAM controller), and run it. You should see the preloader running successfully (See Figure 7.39). You can then close the window (you don't need to wait for the GDBSERVER stuff).

```
ece-cpuio23.uwaterloo.ca - Remote Desktop Connection
Command Prompt - quartus_hps.exe -c 1 -o GDBSERVER --gdbport0=3000 --preloader=N:/u-boot-spl.de1-soc.srec --pre
Info: (including device programming or simulation files), and any
Info: associated documentation or information are expressly subject
Info: to the terms and conditions of the Intel Program License
Info: Subscription Agreement, the Intel Quartus Prime License Agreement,
Info: the Intel FPGA IP License Agreement, or other applicable license
Info: agreement, including, without limitation, that your use is for
Info: the sole purpose of programming logic devices manufactured by
Info: Intel and sold by Intel or its authorized distributors. Please
Info: refer to the applicable agreement for further details, at
Info: https://fpgasoftware.intel.com/eula.
Info: Processing started: Tue Jan 26 15:41:51 2021
Info: Command: quartus_hps -c 1 -o GDBSERVER --gdbport0=3000 --preloader=N:/u-boot-spl.de1-soc.srec
ff13a0
Current hardware is: DE-SoC [USB-1]
Hardware frequency: 16000000
Found HPS at device 1
Double check JTAG chain
HPS Device IDCODE: 0x4BA00477
AHB Port is located at port 0
APB Port is located at port 1
Double check device identification ...
>>CPU0 halted at 0x2fa8.
>>Resetting HPS.
>>Downloading preloader.....
>>Program loaded. PC set to program entry (0xFFFF0000)
>>Setting vector base address register to: 0xffff0000
>>Running preloader..
>>Preloader successfully run.
Starting GDB Server.
Listening on port 3000 for connection from GDB: 15s_
```

Figure 7.39: ARM DS IDE: Run Preloader

7.13 Import an ARM DS Project

To import an existing ARM DS project into the workspace, in the main menu, select **File** → **Import** to start. Under the General folder, either **Existing Projects into Workspace** or **Projects from Folder or Archive** would do the job. At the <https://github.com/yqh/ece350>, the `manual_code/lab1/SVC` is an ARM DS project that you can directly import into your workspace.

7.14 Errata

1. Page 44, Figure 8.27, the output line “test_result = 0” has been corrected to “test_result = 8” due to a bug fix.

Chapter 8

Programming Cortex-A9

8.1 The ARM Instruction Set Architecture

The Cortex-A9 supports ARM, Thumb, and Thumb-2 instruction sets. By default, the processor uses ARM instruction set. In the RTOS lab, you will need to program some code (5% - 10%) in the assembler language. We introduce a few assembly instructions that you most likely need to use in your project in this section.

The general formatting of the assembler code is as follows:

```
label
    opcode operand1, operand2, ... ; Comments
```

The `label` is optional. Normally the first operand is the destination of the operation (note STR is one exception).

Table 8.1 lists some assembly instructions that the RTX project may use. For more details on instruction set reference, we refer the reader to Sections 4, 6 and 7 (Introduction to the ARM Processor Using ARM Toolchain) in [3].

8.2 ARM Architecture Procedure Call Standard (AAPCS)

The AAPCS (ARM Architecture Procedure Call Standard) defines how subroutines can be separately written, separately compiled, and separately assembled to work together. The C compiler follows the AAPCS to generate the assembly code. Table 8.2 lists registers used by the AAPCS.

Registers R0-R3 are used to pass parameters to a function and they are not preserved. The compiler does not generate assembler code to preserve the values of these registers. R0 is also used for return value of a function.

Registers R4-R11 are preserved by the called function. If the compiler generated assembler code uses registers in R4-R11, then the compiler generate assembler code

Mnemonic	Operands/Examples	Description
LDR	$Rt, [Rn, \#offset]$ LDR R1, [R0, #24]	Load Register with word Load word value from an memory address R0+24 into R1
LDM	$Rn\{!\}, reglist$ LDM R4, {R0 – R1}	Load Multiple registers Load word value from memory address R4 to R0, increment the address, load the value from the updated address to R1.
STR	$Rt, [Rn, \#offset]$ STR R3, [R2, R6] STR R1, [SP, #20]	Store Register word Store word in R3 to memory address R2+R6 Store word in R1 to memory address SP+20
MRS	$Rd, spec_reg$ MRS R0, MSP MRS R0, PSP	Move from special register to general register Read MSP into R0 Read PSP into R0
MSR	$spec_reg, Rm$ MSR MSP, R0 MSR PSP, R0	Move from general register to special register Write R0 to MSP Write R0 to PSP
PUSH	$reglist$ PUSH {R4 – R11, LR}	Push registers onto stack push in order of decreasing the register numbers
POP	$reglist$ POP {R4 – R11, PC}	Pop registers from stack pop in order of increasing the register numbers
BL	$label$ BL funC	Branch with Link Branch to address labeled by funC, return address stored in LR
BLX	Rm BLX R12	Branch indirect with link Branch with link and exchange (Call) to an address stored in R12
BX	Rm BX LR	Branch indirect Branch to address in LR, normally for function call return

Table 8.1: Assembler instruction examples

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer (full descending stack).
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6	Platform register.
		SB	The meaning of this register is defined by platform standard.
		TR	
r8	v5		Variable-register 5.
r7	v4		Variable-register 4.
r6	v3		Variable-register 3.
r5	v2		Variable-register 2.
r4	v1		Variable-register 1.
r3	a4		argument / scratch register 4
r2	a3		argument / scratch register 3
r1	a2		argument / result / scratch register 2
r0	a1		argument / result / scratch register 1

Table 8.2: Core Registers and AAPCS Usage

to automatically push/pop the used registers in R4-R11 upon entering and exiting the function.

R12-R15 are special purpose registers. A function that has the `_svc_indirect` keyword makes the compiler put the first parameter in the function to R12 followed by an SVC instruction. R13 is the stack pointer (SP). R14 is the link register (LR), which normally is used to save the return address of a function. R15 is the program counter (PC).

Note that the exception stack frame automatically backs up R0-R3, R12, LR and PC together with the xPSR. This allows the possibility of writing the exception handler in purely C language without the need of having a small piece of assembly code to save/restore R0-R3, LR and PC upon entering/exiting an exception handler routine.

8.3 Cortex Microcontroller Software Interface Standard (CMSIS)

The Cortex Microcontroller Software Interface Standard (CMSIS) was developed by ARM. It provides a standardized access interface for embedded software products (see Figure 8.1). This improves software portability and re-usability. It enables software solution suppliers to develop products that can work seamlessly with device libraries from various silicon vendors [4]. It has been extended to support Cortex-A series processors.

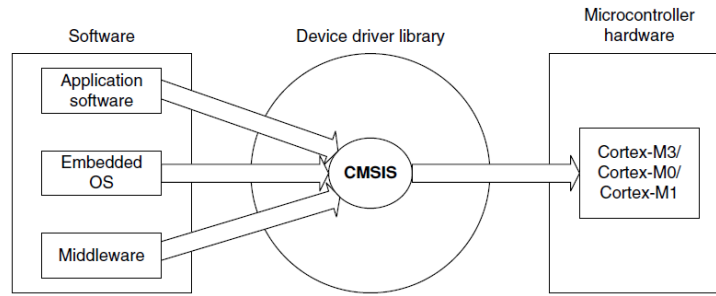


Figure 8.1: Role of CMSIS[6]

The CMSIS uses standardized methods to organize header files that makes it easy to learn new Cortex-M microcontroller products and improve software portability. With the `<device>.h` (e.g. `device_a9.h`) and system startup code files (e.g., `startup_a9.s`), your program has a common way to access

- **Cortex-M processor core registers** with standardized definitions for NVIC, SysTick, MPU registers, System Control Block registers , and their core access functions (see `core_cm*.ch` files).
- **system exceptions** with standardized exception number and handler names to allow RTOS and middleware components to utilize system exceptions without having compatibility issues.
- **intrinsic functions with standardized name** to produce instructions that cannot be generated by IEC/ISO C.
- **system initialization** by common methods for each MCU. Fore example, the standardized `SystemInit()` function to configure clock.
- **system clock frequency** with standardized variable named as `SystemFrequency` defined in the device driver.
- **vendor peripherals** with standardized C structure.

8.3.1 CMSIS files

The CMSIS is divided into multiple layers (See Figure 8.2). For each device, the MCU vendor provides a device header file `<device>.h` (e.g., `device_a9.h`) which pulls in additional header files required by the device driver library and the Core Peripheral Access Layer (see Figure 8.3).

By including the `<device>.h` (e.g., `device_a9.h`) file into your code file. The first step to initialize the system can be done by calling the CMSIS function as shown in Listing 8.1.

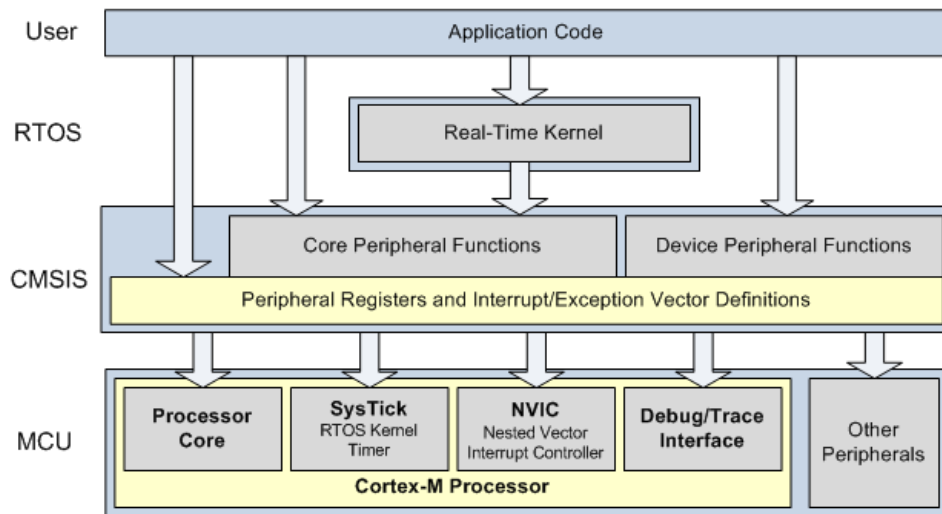


Figure 8.2: CMSIS Organization[4]

```
SystemInit(); // Initialize the MCU clock
```

Listing 8.1: CMSIS SystemInit()

The CMSIS compliant device drivers also contain a startup code (e.g., `startup_a9.s`), which include the vector table with standardized exception handler names.

8.4 Accessing C Symbols from Assembly

Embedded assembly is support by ARM compiler. To write an embedded assembly function, you need to use the `__asm` keyword. For example the the function “`embedded_asm_function`” in Listing 8.3 is an embedded assembly function. You can only put assembly instructions inside this function. Note that inline assembly is not supported in Cortex-M3.

The `__cpp` keyword allows one to access C compile-time constant expressions, including the addresses of data or functions with external linkage, from the assembly code. The expression inside the `__cpp` can be one of the followings:

- A global variable defined in C. In Listing 8.2, we have two C global variables `g_pcb` and `g_var`. We can use the `__cpp` to access them as shown in Listing 8.3.

```
#define U32 unsigned int
#define SP_OFFSET 4

typedef struct pcb {
    struct pcb *mp_next;
    U32 *mp_sp; // 4 bytes offset from the starting address of
```

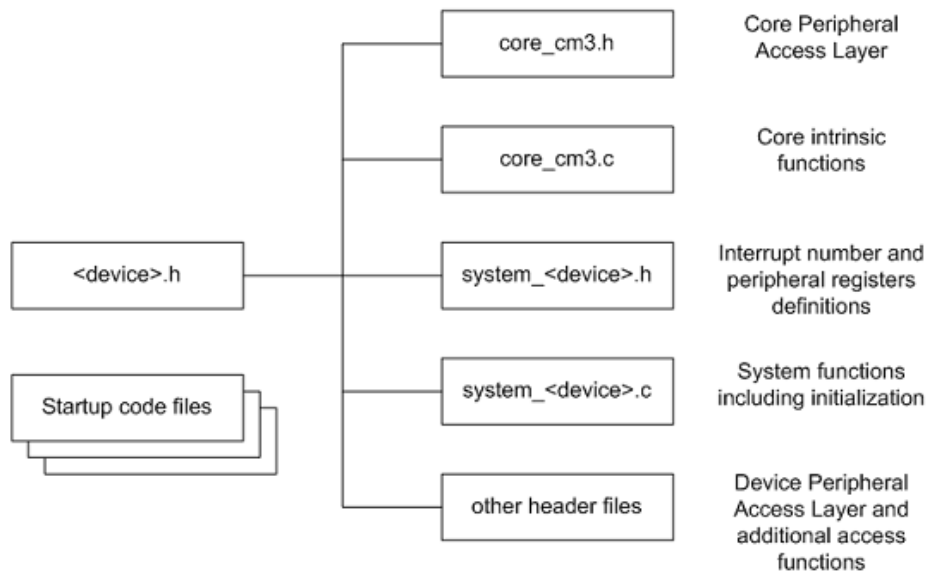



Figure 8.3: CMSIS Organization[4]

```

        // this structure
        //other variables...
    } PCB;

    PCB g_pcb;
    U32 g_var;

```

Listing 8.2: Example of accessing C global variables from assembly. The C code.

```

__asm embedded_asm_function(void) {
    LDR R3, =__cpp(&g_pcb) ; load R3 with the address of g_pcb
    LDM R3, {R1, R2}      ; load R1 with g_pcb.mp_next
                          ; load R2 with g_pcb.mp_sp
    LDR R4, =__cpp(g_var) ; load R4 with the value of g_var
    STR R4, [R3, #SP_OFFSET] ; write R4 value to g_pcb.mp_sp
}

```

Listing 8.3: Example of accessing global variable from assembly

- A C function. In Listing 8.4, `a_c_function` is a function written in C. We can invoke this function by using the assembly language.

```

extern void a_c_function(void);
...
__asm embedded_asm_function(void) {
    ;.....
    BL __cpp(a_c_function) ; a_c_function is regular C function
    ;.....
}

```

Listing 8.4: Example of accessing c function from assembly

- A constant expression in the range of 0 – 255 defined in C. In Listing 8.5, `g_flag` is such a constant. We can use `MOV` instruction on it. Note the `MOV` instruction only applies to immediate constant value in the range of 0 – 255.

```
unsigned char const g_flag;

__asm embedded_asm_function(void) {
    ;.....
    MOV R4, #__cpp(g_flag) ; load g_flag value into R4
    ;.....
}
```

Listing 8.5: Example of accessing constant from assembly

You can also use the `IMPORT` directive to import a C symbol in the embedded assembly function and then start to use the imported symbol just as a regular assembly symbol (see Listing 8.6).

```
void a_c_function (void) {
    // do something
}

__asm embedded_asm_add(void) {
    IMPORT a_c_function ; a_c_function is a regular C function
    BL a_c_function ; branch with link to a_c_function
}
```

Listing 8.6: Example of using `IMPORT` directive to import a C symbol.

Names in the `__cpp` expression are looked up in the C context of the `__asm` function. Any names in the result of the `__cpp` expression are mangled as required and automatically have `IMPORT` statements generated from them.

8.5 SVC Programming: Writing an RTX API Function

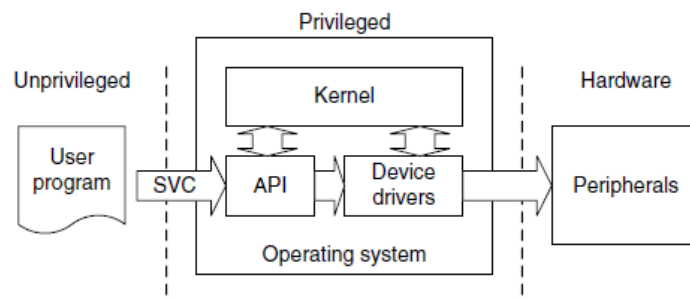


Figure 8.4: SVC as a Gateway for OS Functions [6]

A function in RTX API requires a service from the operating system. It needs to be implemented through the proper gateway by *trapping* from the user level into the kernel level. On Cortex-M3, the SVC instruction is used to achieve this purpose.

The basic idea is that when a function in RTX API is called from the user level, this function will trigger an SVC instruction. The SVC_Handler, which is the CMSIS standardized exception handler for SVC exception will then invoke the kernel function that provides the actual service (see Figure 8.4). Effectively, the RTX API function is a wrapper that invokes SVC exception handler and passes corresponding kernel service operation information to the SVC handler.

To generate an SVC instruction, there are two methods. One is a direct method and the other one is an indirect method.

The direct method is to program at assembly instruction level. We can use the embedded assembly mechanism and write SVC assembly instruction inside the embedded assembly function. One implementation of `void *mem_alloc(size_t size)` is shown in Listing 8.7.

```
__asm void *mem_alloc(size_t size) {  
    LDR R12,=__cpp(k_mem_alloc)  
    ; code fragment omitted  
    SVC 0  
    BX LR  
    ALIGN  
}
```

Listing 8.7: Code Snippet of mem_alloc

The corresponding kernel function is the C function `k_mem_alloc`. This function entry point is loaded to register `r12`. Then `SVC 0` causes an SVC exception with immediate number 0. In the SVC exception handler, we can then branch with link and exchange to the address stored in `r12`. Listing 8.8 is an excerpt of the `HAL_CA.c` from the starter code.

```
__asm void SVC_Handler(void) {  
    ; save registers  
    ; Extract SVC number, if SVC 0, then do the following  
  
    BLX R12 ; R12 contains the kernel function entry point  
  
    ;restore registers  
}
```

Listing 8.8: Code Snippet of SVC_Handler

The indirect method is to ask the compiler to generate the SVC instruction from C code. The ARM compiler provides an intrinsic keyword named `__svc_indirect` which passes an operation code to the SVC handler in `r12[5]`. This keyword is a function qualifier. The two inputs we need to provide to the compiler are

- `svc_num`, the immediate value used in the SVC instruction and

- `op_num`, the value passed in `r12` to the handler to determine the function to perform. The following is the syntax of an indirect SVC.

```
__svc_indirect(int svc_num)
    return_type function_name(int op_num[, argument-list]);
```

The system handler must make use of the `r12` value to select the required operation. For example, the `mem_alloc` is a user function with the following signature:

```
#include <rtx.h>
void *mem_alloc(size_t size);
```

In `rtx.h`, the following code is revelent to the implementation of the function.

```
#define __SVC_0 __svc_indirect(0)
extern void *k_mem_alloc(size_t size);
#define mem_alloc(size) _mem_alloc((U32)k_mem_alloc, size);
extern void *_mem_alloc(U32 p_func, size_t size) __SVC_0;
```

The compiler generates two assembly instructions

```
LDR.W r12, [pc, #offset]; Load k_mem_alloc into r12
SVC 0x00
```

The `SVC_handler` in Listing 8.8 then can be used to handle the `SVC 0` exception.

Appendix A

Forms

Lab administration related forms are given in this appendix.

ECE 350 Request to Leave a Project Group Form

Name	
Quest ID	
Student ID	
Lab Project ID	
Group ID	
Name of Other Group Member 1	
Name of Other Group Member 2	
Name of Other Group Member 3	

Provide the reason for leaving the project group here:

Signature _____

Date _____

Bibliography

- [1] Cyclone V Hard Processor System Technical Reference Manual. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_54001.pdf.
- [2] DE1-SoC Computer System with ARM* Cortex* A9. ftp://ftp.intel.com/pub/fpgaup/pub/Intel_Material/17.0/Computer_Systems/DE1-SoC/DE1-SoC_Computer_ARM.pdf.
- [3] Intel Corporation FPGA University Program. Introduction to the ARM Processor Using ARM Toolchain. 2019. ftp://ftp.intel.com/pub/fpgaup/pub/Intel_Material/14.0/Tutorials/ARM_A9_intro_alt.pdf.
- [4] MDK Primer. <http://www.keil.com/support/man/docs/gsac>.
- [5] Realview compilation tools version 4.0: Compiler reference guide, 2007-2010.
- [6] J. Yiu. *The Definitive Guide to the ARM Cortex-M3*. Newnes, 2009.