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, January 22, 2021

Contents

iv

List of Tables

List of Figures

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

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 signup 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

2

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

---

[1]You 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:

  ```c
  #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.

---

[2]Grace 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 question [3]. *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.

---

[3] Our 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 is 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 9.5;

- Create your lab1 ARM DS Project from skeleton source code (See Chapter 8);

- Execute the SVC project on the ARM DS by using the VE_Cortex_A9x1 ARM FVP [1]; 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.

---

[1]FVP 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

14

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);
```

---

[2]In 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.

[3]The 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_exfrag()` 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:

$$R_T = \frac{N}{T}.$$ (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

$$R_H = \frac{P}{H}.$$ (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

- 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 of 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` files [4].

- 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

---

[4]For 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 15, the coding listing box at the bottom of the page. The return type of the function has been changed from `void` to `int`.

3. Page 16, first paragraph, "i.e immediate coalescence", added a comma after "i.e.".

4. Page 16, The RETURN VALUE section has been updated to reflect the return type correction of `mem_dealloc` made on page 14.

5. Page 16, last paragraph, "is show as follows" is corrected to "is as follows".

6. Page 18, first paragraph after the bullets, "a non-trivial" is replaced by "A non-trivial".

7. Page 19, Section 2.5.1, original contents is replaced by "None.".

8. Page 19, Section 2.5.2, changed "uVision Project" to "ARM DS project".

9. Page 19, Section 2.5.2, removed "group identification, project manager name and".

Chapter 3

Lab2

To be released.

Chapter 4

Lab3

To be released.

Chapter 5

Lab4

To be released.

Chapter 6

Lab5

To be released.

Part III

Frequently Asked Questions

Part IV

Computing and Software Development Environment Quick Reference Guide

Chapter 7

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 8

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.

8.1 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.

8.2 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`.

8.3 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
```

8.4 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.

8.5   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.
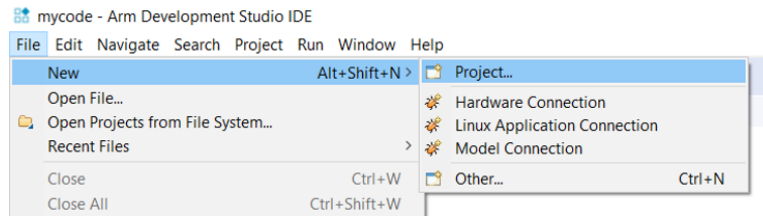
In the main menu, select **File → New → Project** (See Figure 8.1).

Figure 8.1: ARM DS IDE: Create a new project

In the New Project window, select **C/C++ → C Project → Next** (See Figure 8.2).
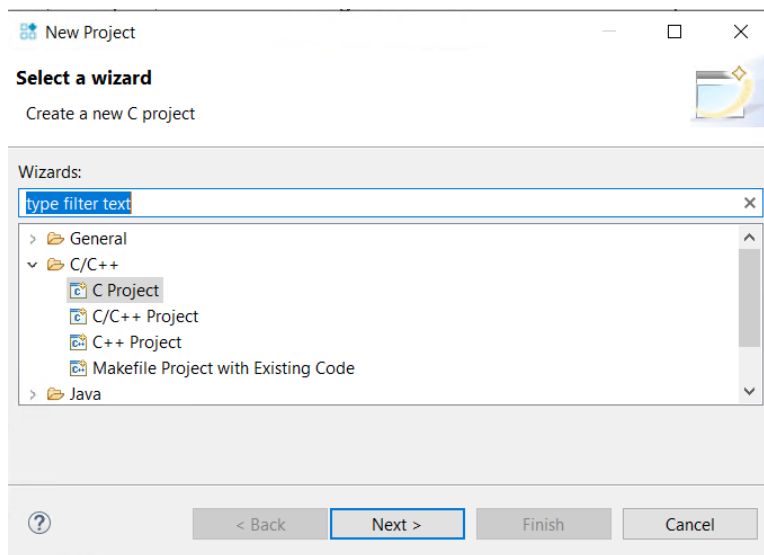

Figure 8.2: ARM DS IDE: Select a wizard

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 8.3). You will see a new empty project MySVC show up in the project explorer (See Figure 8.4).
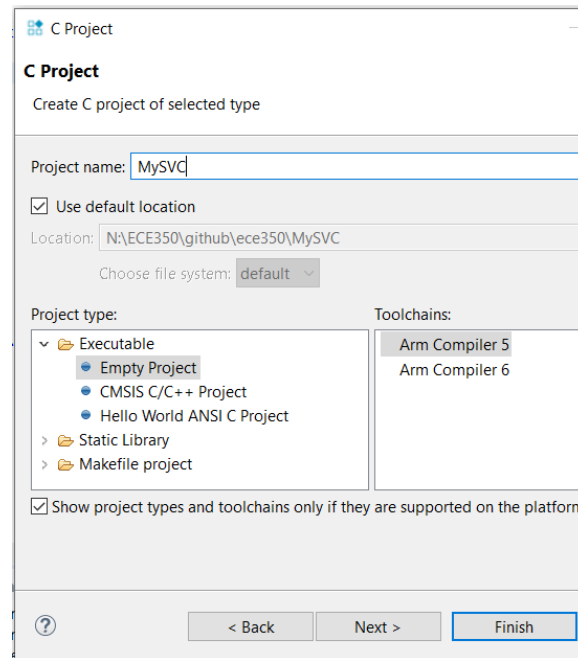
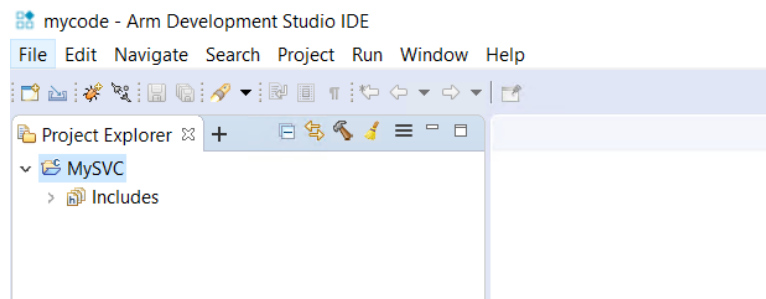Figure 8.3: ARM DS IDE: Select a C project

Figure 8.4: ARM DS IDE: Project explorer

8.6   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 8.5).
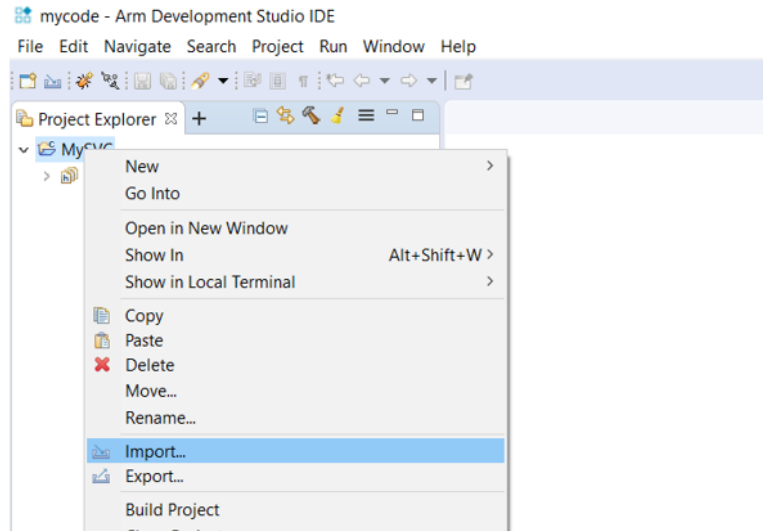

Figure 8.5: ARM DS IDE: Import Files

In the import select wizard window, select **General** → **File System** → **Next** as shown in Figure 8.6
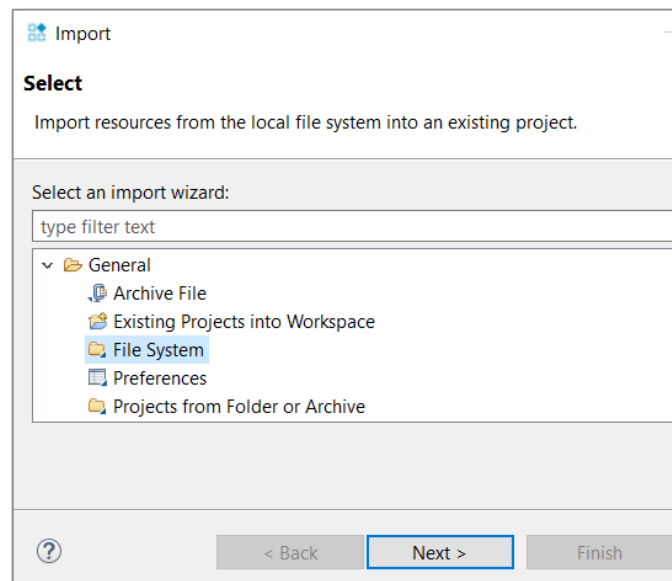

Figure 8.6: ARM DS IDE: Import select wizard

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 8.7. Select the check box of the template folder and leave the "Create top-level-folder" checkbox unchecked. Press **Finish** button.
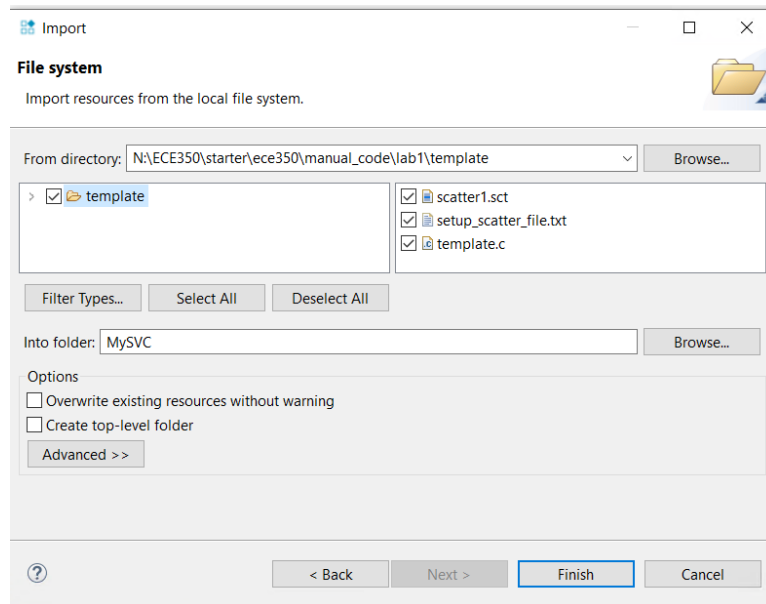

Figure 8.7: ARM DS IDE: Import file system

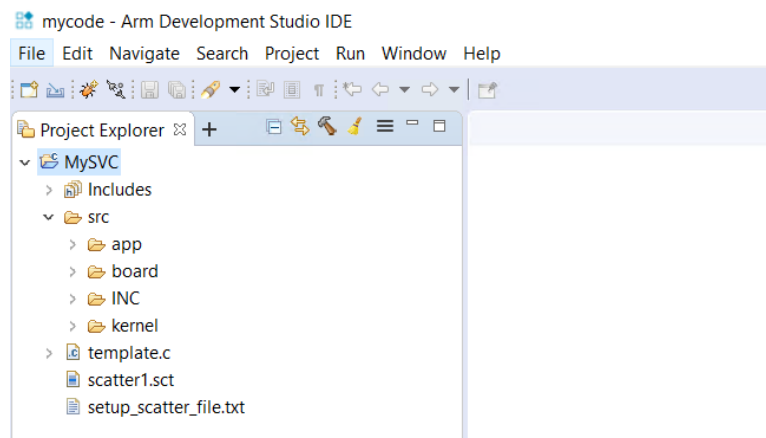Your project should now look like what is shown in Figure 8.8.

Figure 8.8: ARM DS IDE: MySVC project with template files

8.7 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 8.9).
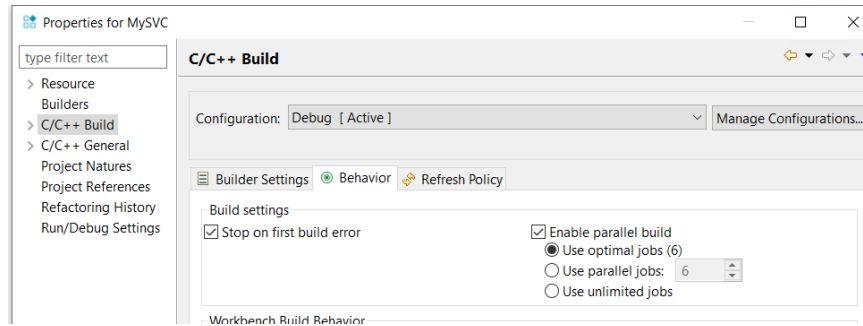

Figure 8.9: ARM DS IDE: MySVC project C/C++ Parallel Build

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 8.10 for details.
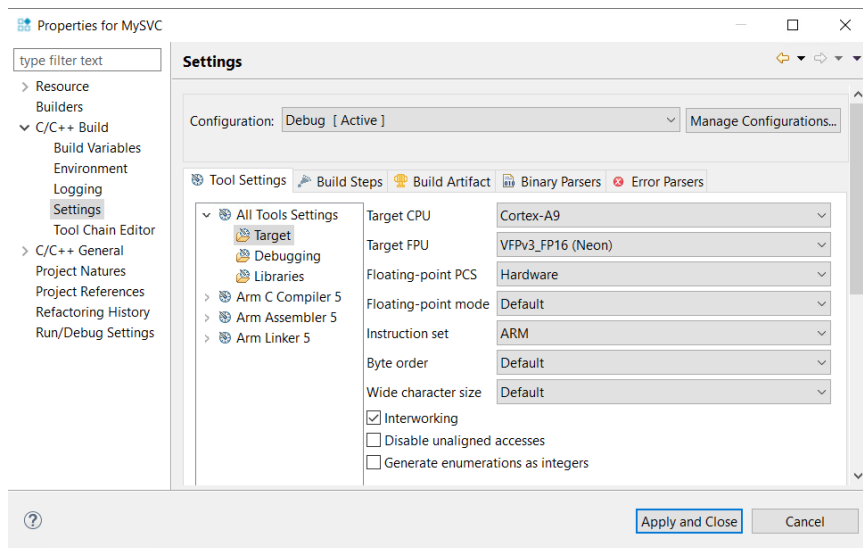

Figure 8.10: ARM DS IDE: MySVC project C/C++ Build Target Setting

Select **Arm C Compiler 5** → **Preprocessor** and add `DEBUG_0` macro (see Figure 8.11).
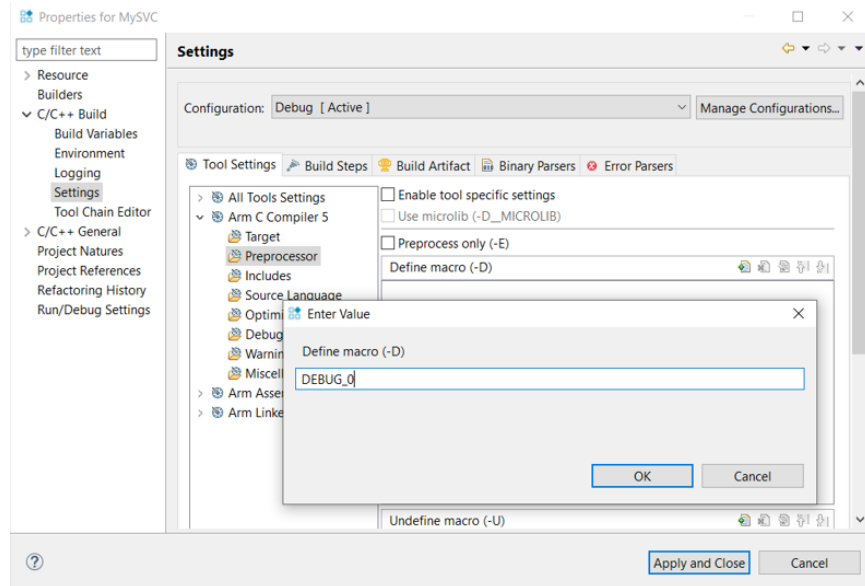

Figure 8.11: ARM DS IDE: MySVC project C/C++ Build Compiler Preprocessor Setting

Select **Arm C Compiler 5** → **Include** (see Figure 8.12) and press **Workspace** button to add include paths from workspace.
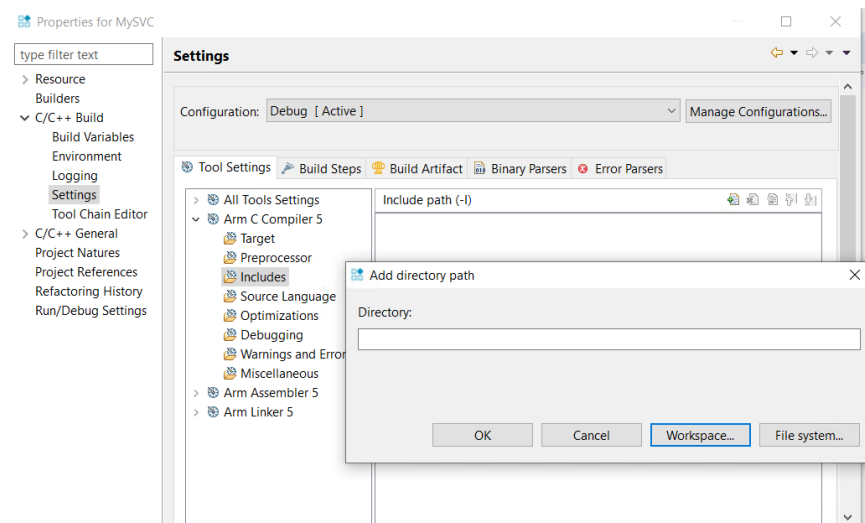

Figure 8.12: ARM DS IDE: MySVC project C/C++ Build Compiler Include Selection

In the folder selection window, select all the folders as shown in Figure 8.13. Select **Arm C Compiler 5** → **Source Language** → **Source language mode** → **C99(–**
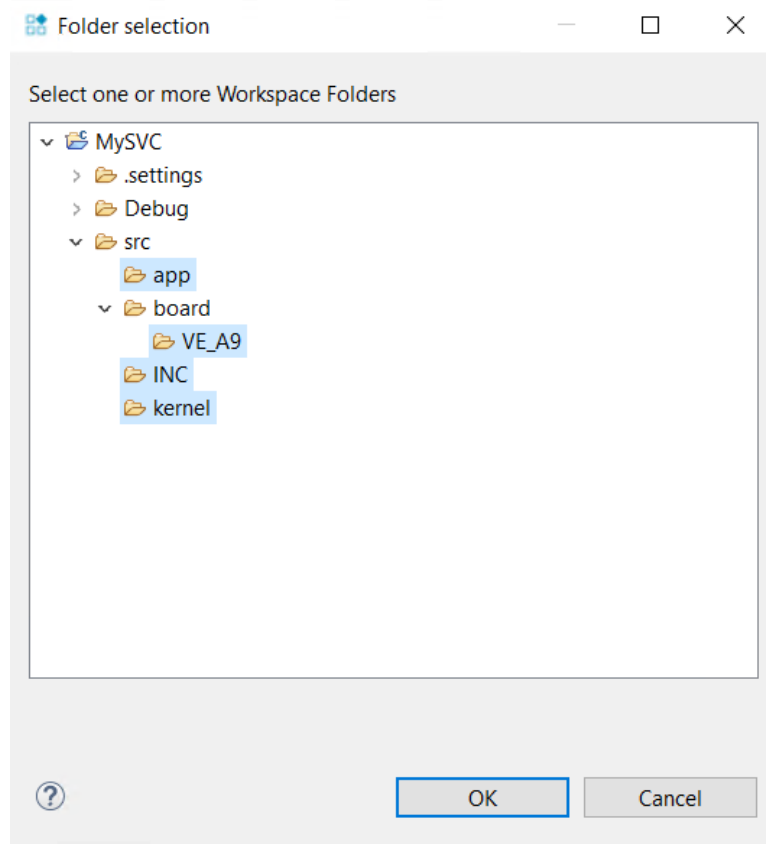
Figure 8.13: ARM DS IDE: MySVC project C/C++ Build Compiler Include Folder Selection

c99) (see Figure 8.14).
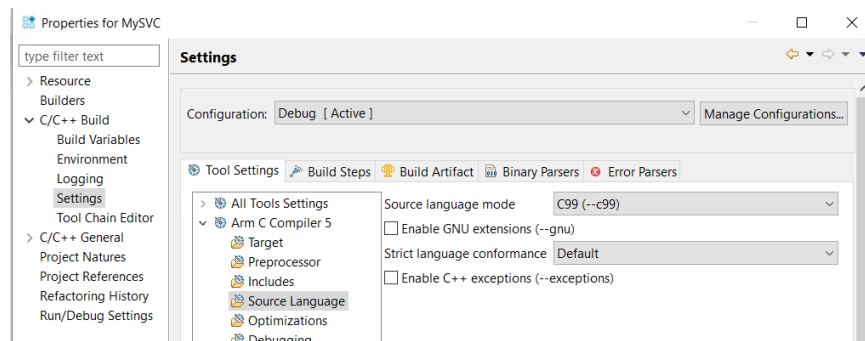
Figure 8.14: ARM DS IDE: MySVC project C/C++ Build Compiler Language C99

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 8.15 for details.
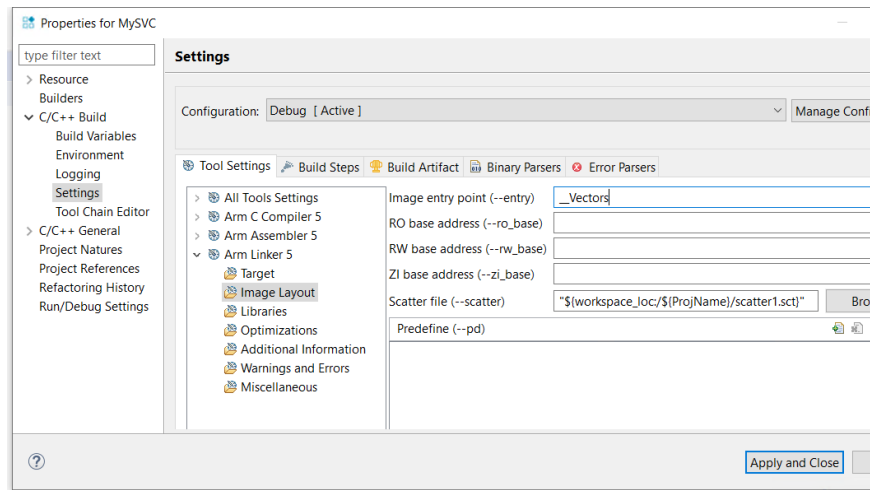
Figure 8.15: ARM DS IDE: MySVC project C/C++ Build Linker Image Layout

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

8.8   Build Project

To build the project, right click the project in the project explorer window and select **Build Project** (See Figure 8.16).
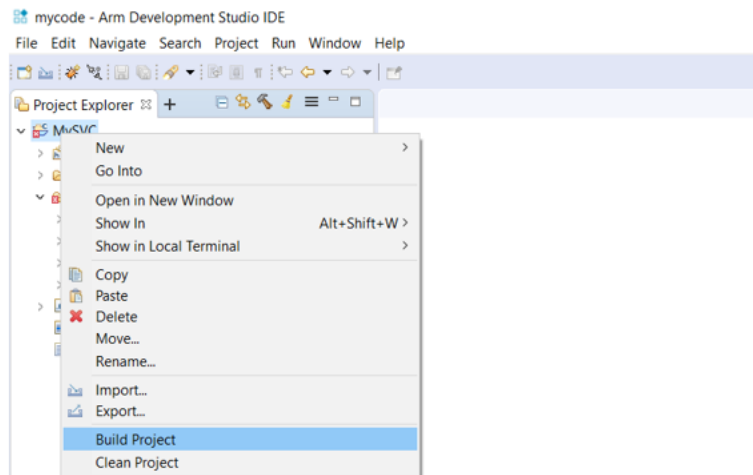
Figure 8.16: ARM DS IDE: Build Project

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 8.17).
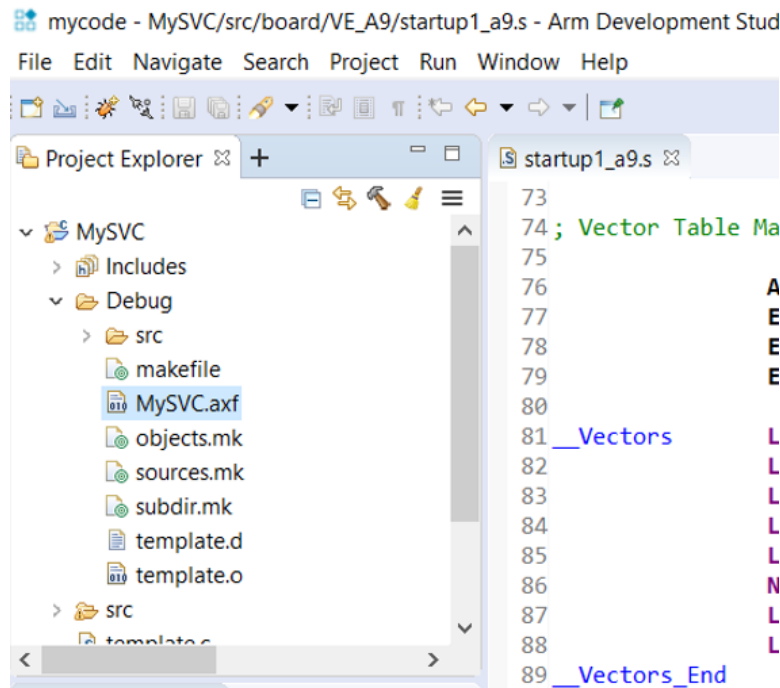
Figure 8.17: ARM DS IDE: Built executable in Debug folder

8.9 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 8.18).
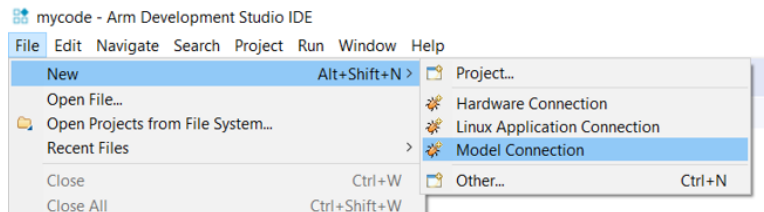

Figure 8.18: ARM DS IDE: Create a new model connection

In the debug connection window, select the MySVC project and give your debug connection a name, say "MySVC1" (See Figure 8.19). Note the debug connection name can be different from the project name. Press the **Next** button to bring up the
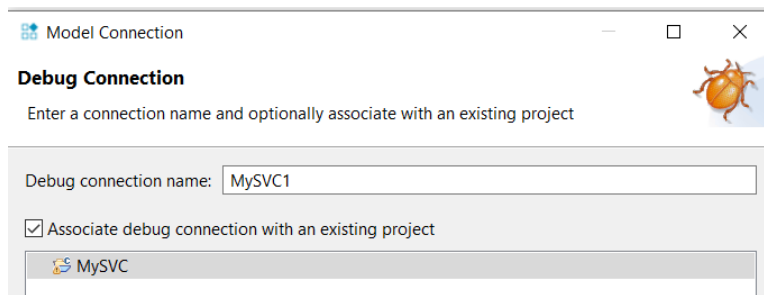
Figure 8.19: ARM DS IDE: Debug Connection Configuration

target selection window.

Select **Arm FVP (Installed with Arm DS)** (See Figure 8.20) and expand it.
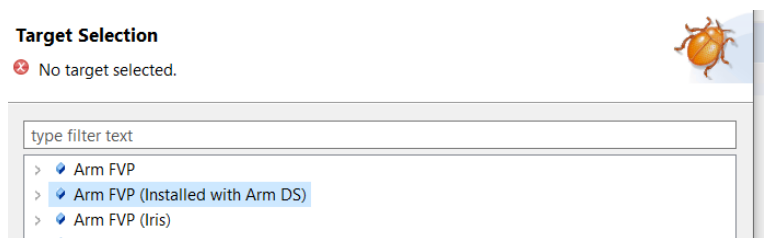
Figure 8.20: ARM DS IDE: Debug Connection Target Selection

Scroll down the list and select **VE Cortex A9x1** as shown in Figure 8.21.

Figure 8.21: 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 8.22).
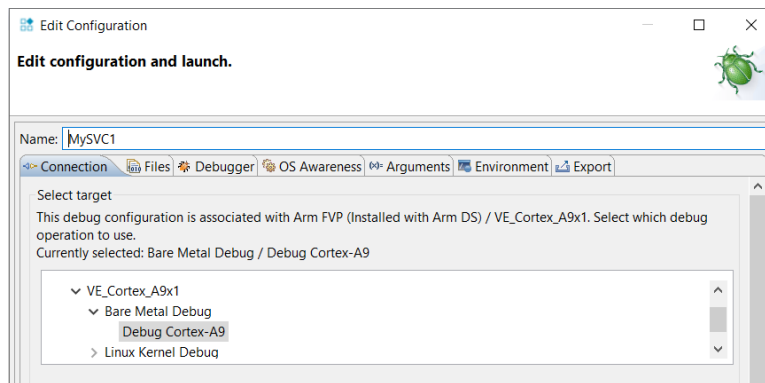
Figure 8.22: ARM DS IDE: Debug Connection Target VE Cortex A9x1 Selection

Select the **Files** tab and press the **Workspace** button for Application on host to download setting as shown in Figure 8.23.
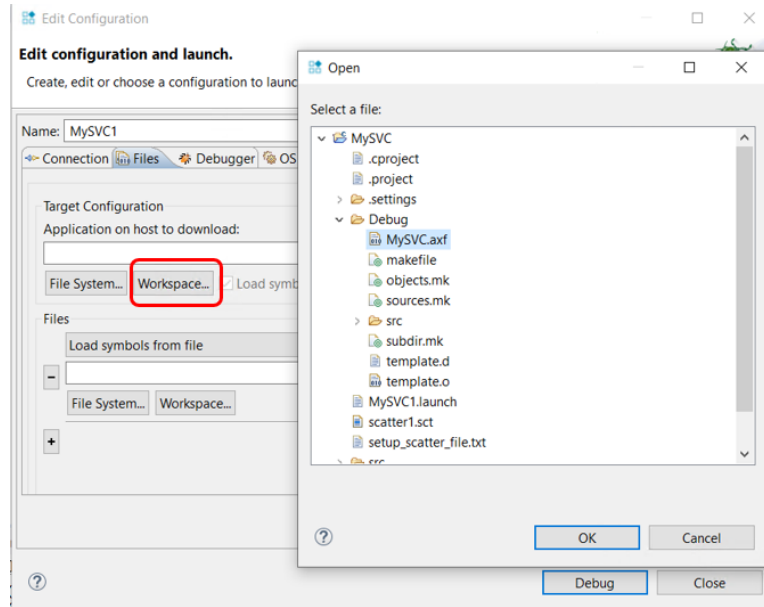
Figure 8.23: ARM DS IDE: Debug Connection Select Target to Download

Expand the **Debug** folder and select **MySVC.axf** as shown in Figure 8.23. Press the **OK** button. You will see some error message as shown in Figure 8.24.
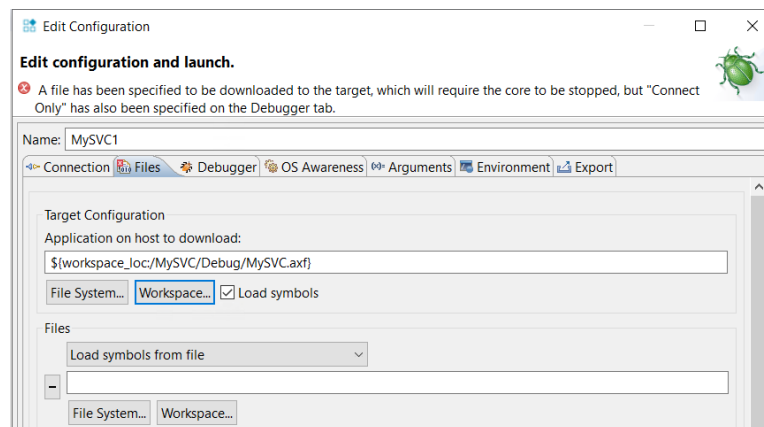
Figure 8.24: ARM DS IDE: Debug Connection Files Tab Setting Error

Select the **Debugger** tab and select **Debug from entry point** as shown in Figure 8.25. Press **Apply** to apply all the debug connection configurations we just made.
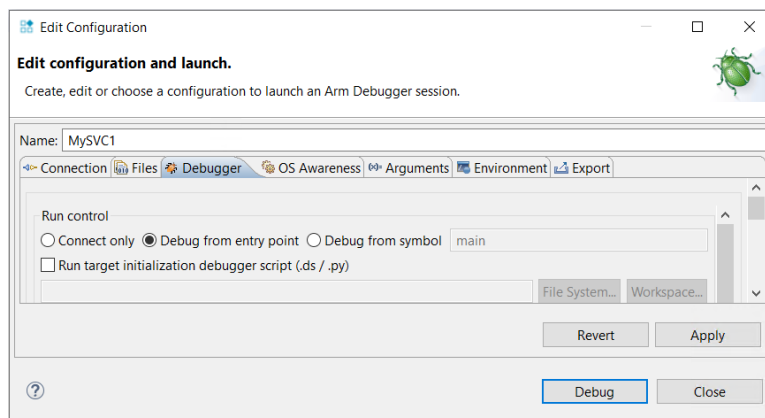
Figure 8.25: ARM DS IDE: Debug Connection Files Tab Setting Error

8.10 Debug the Target

Press the **Debug** button (see Figure 8.25. You will see the FVP starts and the program will stop at its entry point (see Figure 8.26). 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 8.27.
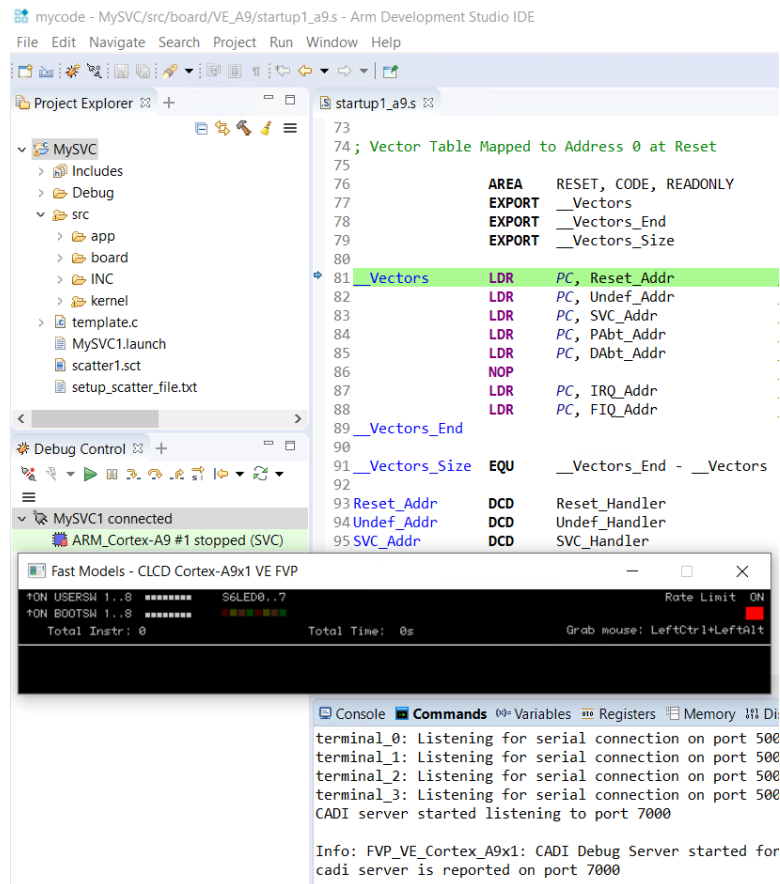
Figure 8.26: ARM DS IDE: Debug Start

```
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 8.27: ARM DS IDE: Debug Output

8.11   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.

8.12   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 9

Programming Cortex-A9

9.1 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 9.1 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].

9.2 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 9.2 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	$Rt, [Rn, \#offset]$	Load Register with word
	LDR R1, [R0, \#24]	Load word value from an memory address R0+24 into R1
LDM	$Rn\{!\}, reglist$	Load Multiple registers
	LDM R4, $\{$R0 $-$ R1$\}$	Load word value from memory address R4 to R0, increment the address, load the value from the updated address to R1.
STR	$Rt, [Rn, \#offset]$	Store Register word
	STR R3, [R2, R6]	Store word in R3 to memory address R2+R6
	STR R1, [SP, \#20]	Store word in R1 to memory address SP+20
MRS	$Rd, spec\_reg$	Move from special register to general register
	MRS R0, MSP	Read MSP into R0
	MRS R0, PSP	Read PSP into R0
MSR	$spec\_reg, Rm$	Move from general register to special register
	MSR MSP, R0	Write R0 to MSP
	MSR PSP, R0	Write R0 to PSP
PUSH	$reglist$	Push registers onto stack
	PUSH $\{$R4 $-$ R11, LR$\}$	push in order of decreasing the register numbers
POP	$reglist$	Pop registers from stack
	POP $\{$R4 $-$ R11, PC$\}$	pop in order of increasing the register numbers
BL	$label$	Branch with Link
	BL funC	Branch to address labeled by funC, return address stored in LR
BLX	Rm	Branch indirect with link
	BLX R12	Branch with link and exchange (Call) to an address stored in R12
BX	Rm	Branch indirect
	BX LR	Branch to address in LR, normally for function call return

Table 9.1: 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 9.2: 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.

9.3 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 9.1). 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 [1]. It has been extended to support Cortex-A series processors.
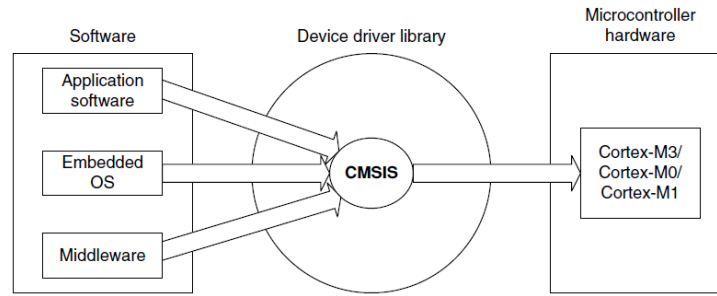
Figure 9.1: Role of CMSIS[4]

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.

9.3.1 CMSIS files

The CMSIS is divided into multiple layers (See Figure 9.2). 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 9.3).

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 9.1.
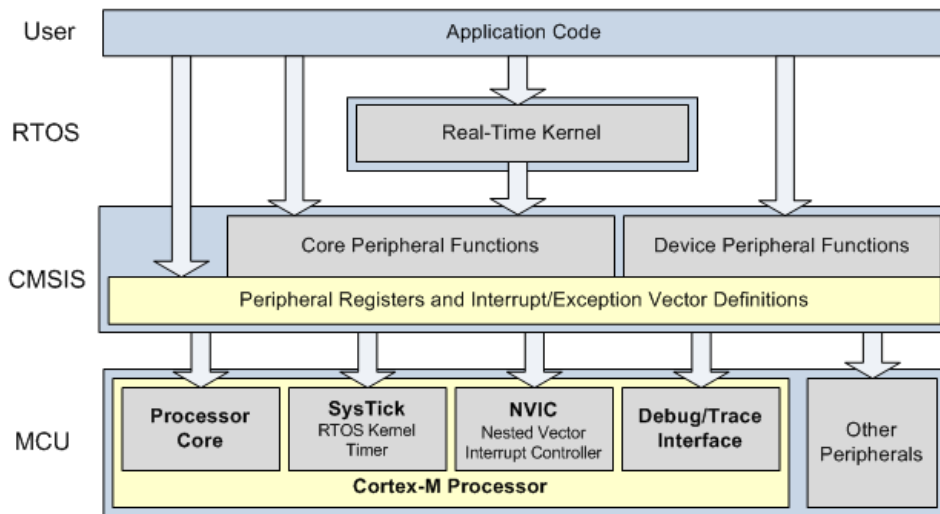
Figure 9.2: CMSIS Organization[1]

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

Listing 9.1: 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.

9.4 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 9.3 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 9.2, we have two C global variables `g_pcb` and `g_var`. We can use the __cpp to access them as shown in Listing 9.3.

```
#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
```
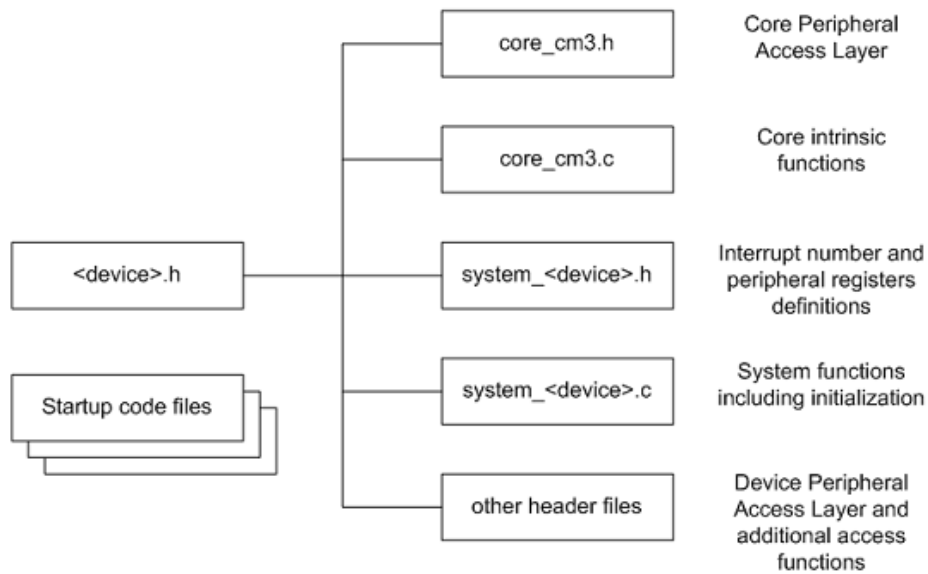
50

Figure 9.3: CMSIS Organization[1]

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

PCB g_pcb;
U32 g_var;
```

Listing 9.2: 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 9.3: Example of accessing global variable from assembly

- A C function. In Listing 9.4, 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 9.4: Example of accessing c function from assembly

- A constant expression in the range of $0 - 255$ defined in C. In Listing 9.5, 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 9.5: 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 9.6).

```
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 9.6: 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.

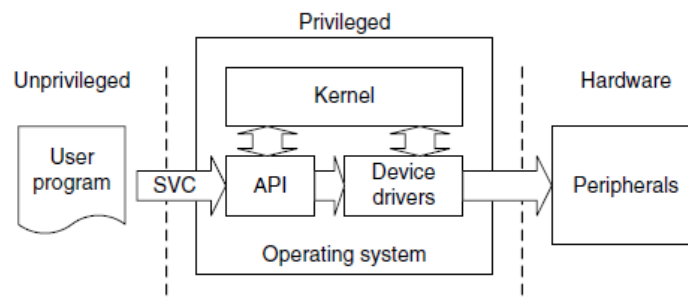## 9.5   SVC Programming: Writing an RTX API Function

Figure 9.4: SVC as a Gateway for OS Functions [4]

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 CM-SIS standardized exception handler for SVC exception will then invoke the kernel function that provides the actual service (see Figure 9.4). 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 9.7.

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

Listing 9.7: 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 9.8 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 9.8: 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[2]. 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 9.8 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] MDK Primer. http://www.keil.com/support/man/docs/gsac.

[2] Realview compilation tools version 4.0: Compiler reference guide, 2007-2010.

[3] Intel Corporation FPGA University Program. Introduction to the ARM Processor Using ARM Toolchain. 2019.

[4] J. Yiu. *The Definitive Guide to the ARM Cortex-M3*. Newnes, 2009.