Lab Manual for ECE 350

by

Seyed Majid Zahedi
Yiqing Huang
Aravind Vellora Vayalapra

Electrical and Computer Engineering Department
University of Waterloo

Contents

List of Tables

List of Figures

Acknowledgment

First, we thank Prof. Paul Dasiewicz who kindly provided detailed notes and sample code for ECE354 lab manual which greatly influenced this lab manual. 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. Our gratitude also goes out to Eric Praetzel for his continuous strong support with regard to the IT infrastructure. We also thank Rasoul Keshavarzi-Valdani for lending us 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. We also sincerely thank the following generous donations:

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

Finally, we thank our ECE354, SE350, and ECE350 students from previous terms who provided us with constructive feedback.

Chapter 1

Lab Administration

1.1 Groups

1.1.1 Group Size

The project is done in groups of four. Five is not allowed and three is not recommended. The workload is fixed regardless of the size of the group. All group members receive the same grade for each project.

1.1.2 Group Sign-up

LEARN is used for group sign-up. Table 1.1 presents the deadline for group sign-up. Please note that grace days do not apply to group sign-up. After the deadline, any student without a group will be randomly assigned to a group.

1.1.3 Quitting Groups

Students can quit their group and join a new one only once. Students need to notify the lab instructor in writing and sign the group split-up form (see the Appendix A) at least one week before the nearest lab deadline. The split-up happens after the lab deadline. If a group member leaves their group, all members of the group loose their group-sign-up points.

1.1.4 Source Code

Groups should maintain the source code and their documents in GitLab. GitLab repositories will be created for each group with group members as "Maintainers" on the project. If a group member leaves the group, they will be removed from the group repository.

Project	Weight (%)	Deadline (EST)
Group sign-up	3	Jan 11 at 23:00
Memory management (P1)	27	Feb 1 at 23:00
Task management (P2)	35	Mar 8 at 23:00
Inter-task comm. and I/O (P3)	35	Mar 29 at 23:00

Table 1.1: Lab projects weights and deadlines.

1.1.5 Collaboration Policy

Explaining concepts to someone in another group, discussing algorithms/testing strategies with other groups, helping someone from another group to debug their code, and searching online for generic algorithms (e.g., hash table) are allowed. Sharing code and test cases with another group, open-sourcing code (e.g., hosting code publicly on GitHub) even after this term, copying/reading other groups' code and test cases, and copying/reading online code and test cases from prior years are not allowed. Any suspected plagiarism or infractions of this honor code will be reported to the appropriate Associate Dean.

1.2 Lab Projects

1.2.1 Late Submissions

Table 1.1 presents the weight and deadline of each project. There are three grace days (including weekends) that can be used for late submissions without incurring any penalty. When all grace days are used, a 15% penalty is applied per day for late submissions. Please be advised that to simplify the book-keeping, late submissions are rounded up. A ten-minute-late submission receives the same penalty as a fifteen-hour-late submission. Submissions after three days are not accepted.

1.2.2 Demo Policy

Every group will demo their projects with a lab teaching staff. Each demo has a time limit. During the demo each group is allowed to make changes to their project. An online link will be posted on the course website for booking demo sessions.

1.2.3 Lab Repeating Policy

For students who are retaking the course, labs need to be re-done with new lab partners. Simply turning in the old lab code is not allowed. It is understood that the student may choose a similar route to the solution chosen last time the course was taken. However, it should not be identical.

1.3 Seeking Help

1.3.1 Discussion Forum

Piazza will be used as the preferred discussion forum. Students are encouraged to ask questions on Piazza instead of sending individual emails to the lab teaching staff.

1.3.2 Office Hours

An online link for booking appointments will be posted on the course website. All group members could attend the same appointment. Each appointment is 15 minutes. Groups could book multiple time slots if needed. Please note that teaching staff are not expected to debug code. Debugging is part of the learning exercise for ECE 350.

1.4 Lab Facility

Labs will be done remotely. If in-person classes resume, students will have access to the lab and after-hours access to the lab might be granted in a case-by-case basis. No food or drink is allowed in the lab. Please be informed that you may share the lab with other classes. When resources become too tight, certain access restrictions may apply.

Chapter 2

Software Development Environment

2.1 GitLab Setup

Each group is expected to maintain their source code using git. We will be using University of Waterloo's GitLab instance to manage git repositories. If you have not previously used GitLab, go to git.uwaterloo.ca, and sign in with your UW credentials. This will create a git account for you.

2.2 Setting up SSH Keys on the lab machine

To setup your SSH keys, you can use the following instructions. We recommend every student to set up SSH keys for their user account.

1. Login to an ECE Lab Machine.
2. Open Git-Bash terminal (**Start Menu** → **Git-Bash**).
3. Generate an SSH key pair and save it in your **network drive (N:)** (set a convenient passphrase).

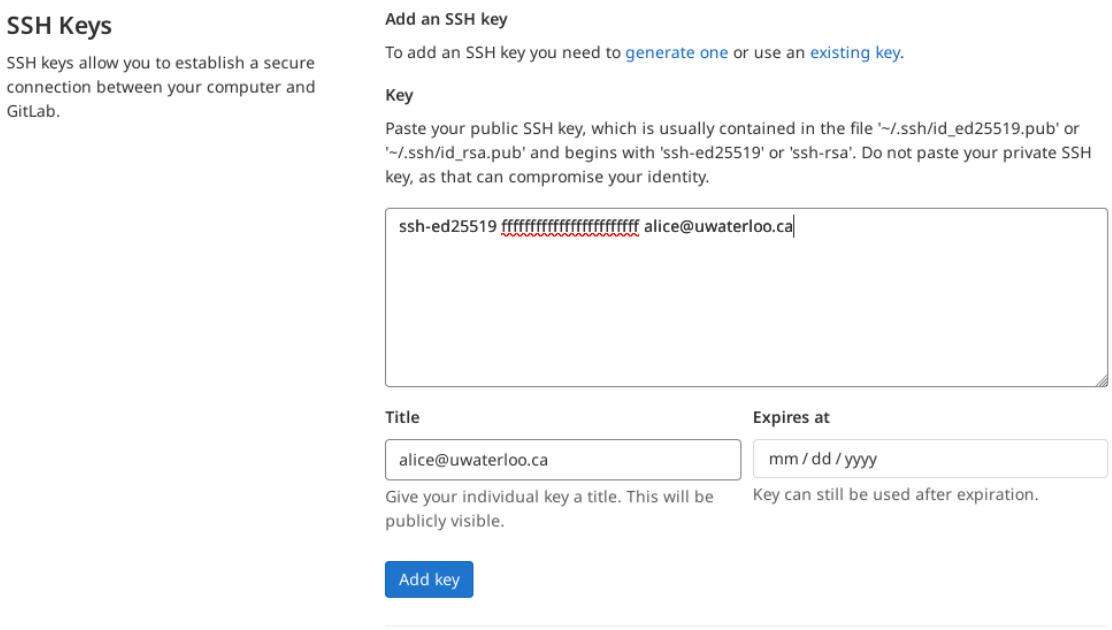
```
$ ssh-keygen -t ed25519 -C "<YOUR UWATERLOO EMAIL>"
...
Enter file in which to save the key: /n/.ssh/id_ed25519
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /n/.ssh/id_ed25519
Your public key has been saved in /n/.ssh/id_ed25519.pub
...
```

4. Copy the contents of the public key file:

```
$ cat /n/.ssh/id_ed25519.pub
ssh-ed25519 fffffffffffffffffffffff alice@uwaterloo.ca
```

5. Log on to GitLab.
6. Click on your user avatar on the top right corner and click **Preferences** to open the Preferences page.

7. On the right hand side pane choose **SSH Keys**.
8. Paste the contents of public key file under key, add a meaningful **Title** and click **Add key**.



2.3 Getting Starter Code from GitLab

1. Open up `Git Bash` terminal (**Start Menu→Git Bash**).
2. Change the directory to your User's Desktop and clone the lab material repository by using the following commands:

```
cd /c/Users/<YOUR UW USERNAME>/Desktop/
git clone ist-git@git.uwaterloo.ca:ece350-w22/student-labs/group<gid>-lab.git
```

2.4 ARM DS Setup

In ECE350, we use ARM DS and Intel DE1-SoC. ARM DS is an eclipse based IDE that allows us to directly program and debug Intel DE1-SoC boards. Further information about ARM DS IDE can be found at the ARM Development Studio User Guide. To setup the IDE of the labs, follow the following instructions.

- Log on to an ECE Lab Machine.
- Click on ARM-DS from the Start Menu.
  - ARM DS will update installed packages. This step usually takes a while — you can let it run in the background. These updates will run daily – every time you open up ARM DS on a lab machine, this process repeats. You may see some error

messages in red saying certain URLs are not accessible, ignore them. You will also see a firewall warning pop up window to ask for granting access, click Cancel.

- Select **File** → **Open Projects from File System...**.
- Click on **Directory...**.
- Navigate to `C:\Users\<USERNAME>\Desktop\<groupid>-lab\Prototype`.
- Select `RTX` folder by clicking on it once.
- Click on the **Select Folder** button.
- Click on **Finish**.
- Right click on the `RTX` folder under **Project Explorer** and click on **Build Project**.
- Double click `RTX.launch` and click **Debug**.

2.5 Setting up NIOS Terminal

- To view the "output" from the DE1-SoC, we need to connect a terminal to the DE1-SoC JTAG UART.
- From the main GUI, open **Window** → **Preferences**.
- In the "Preferences" window, click on **Terminal** → **Local Terminal**.
- Click on **Add** to open the "Add External Executable" window.
- Set the following items, and then click **Add**. (See Figure 2.1).

1. **Name**: "DE1-SoC JTAG UART" (or another name that is meaningful)
2. **Path**: `C:\Software\Altera\20.1\quartus\bin64\nios2-terminal.exe`
3. **Arguments**: `--instance 2`

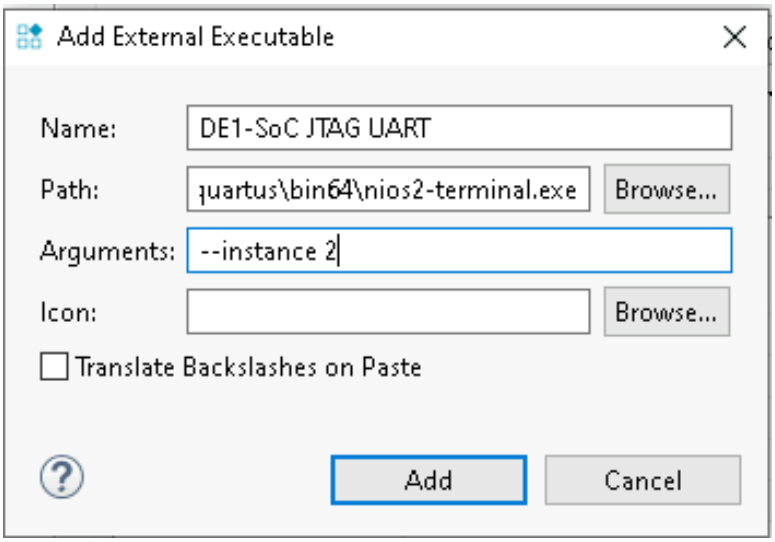


Figure 2.1: ARM DS IDE: Window Preferences Add External Executable

- Click **Apply and Close**.
- From the main GUI, open **Window** → **Terminal**.

- In the Terminal window click on **Open a Terminal** (first icon from the left). In the "Launch Terminal" window, set **Choose Terminal** to "DE1-SoC JTAG UART". Click **OK**.

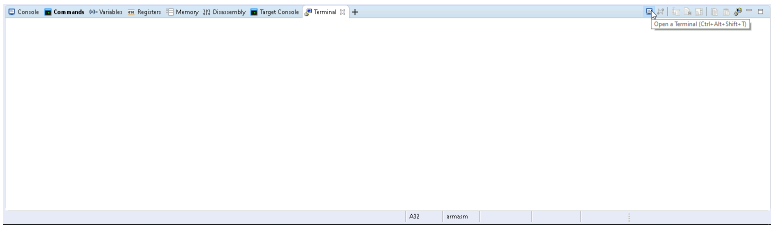


Figure 2.2: ARM DS IDE: Open DE1-SoC JTAG UART Terminal

2.6 Troubleshooting the DE1 SoC Board

The ARM cores on the DE1 SoC board might crash.

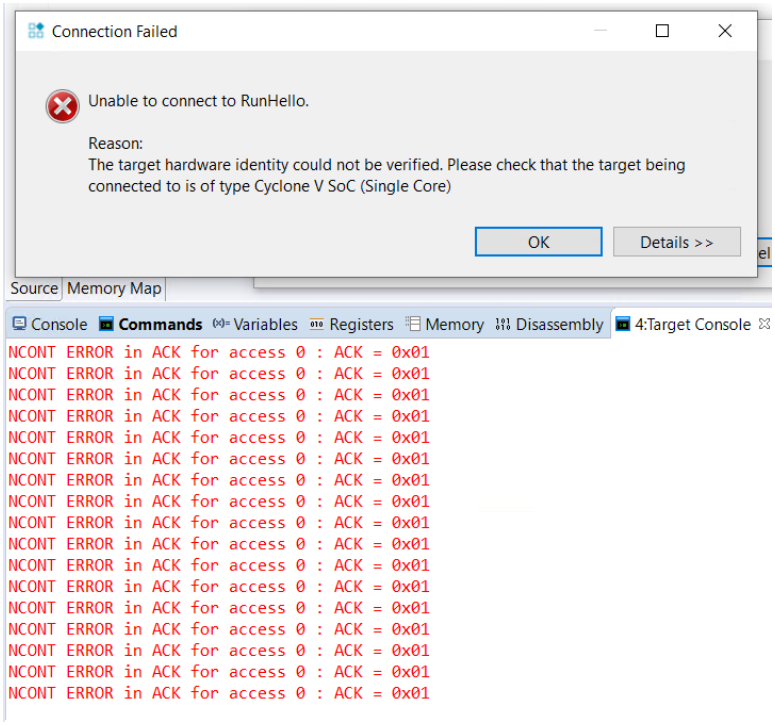


Figure 2.3: ARM DS IDE: Target Errors

If the IDE cannot connect to the target and throws an error (see for example, Figure 2.3), you need to reset the board. To do this, disconnect the hardware connection in ARM DS, and follow the following instructions.

- Open Windows CMD
- Execute the following commands:

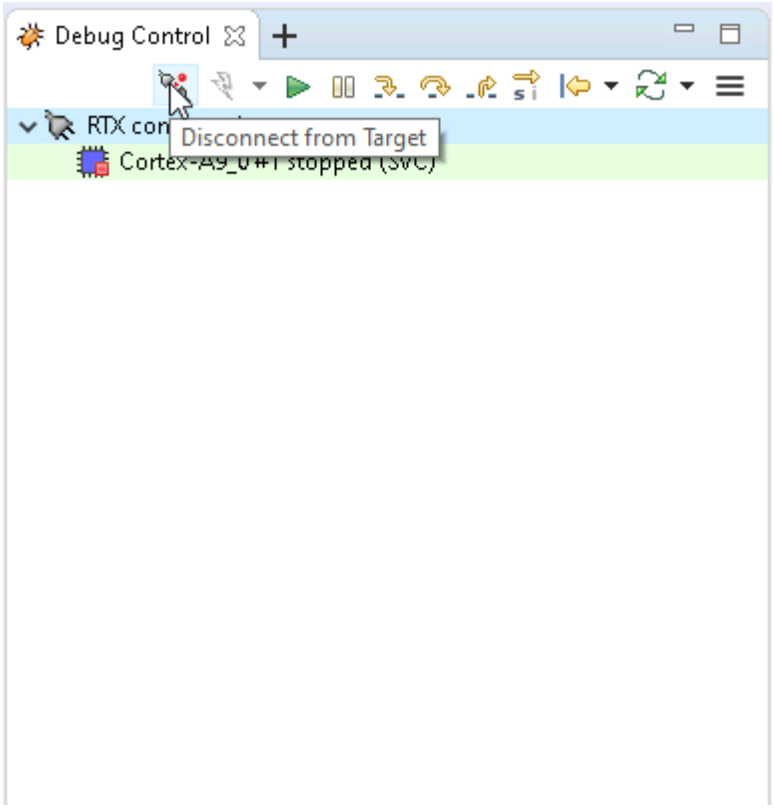


Figure 2.4: ARM DS IDE: Disconnect hardware

```
C:\Software\Altera\20.1\quartus\bin64quartus_pgm.exe -c 1 -m jtag -o "p;N:\ECE350\
    starter\ECE350-Labs\Prototype\DE1-SoC\DE1_SoC_Computer.sof@2"
C:\Software\Altera\20.1\quartus\bin64\quartus_hps.exe -c 1 -o GDBSERVER --gdbport0
    =3008 --preloader=C:\Users\<username>\Desktop\<groupid>-lab\Prototype\DE1-SoC\
    de1-soc.srec --preloaderaddr=0xffff13a0
```

You should see the board programmed and ARM Core Reset successfully. (See Figure 2.6 and Figure 2.5). You can then close the window (you don't need to wait for the GDBSERVER stuff).

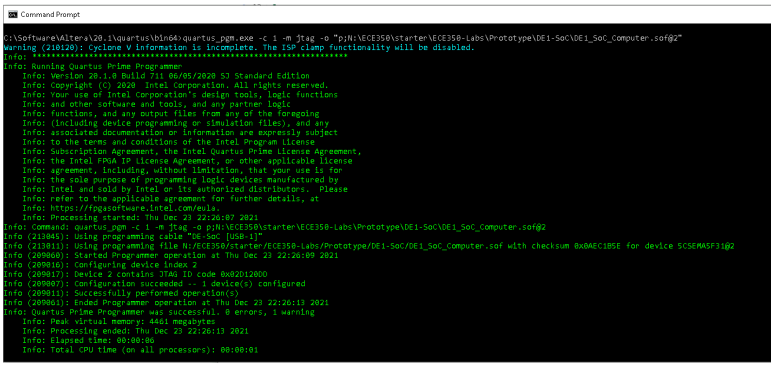


Figure 2.5: ARM DS IDE: Program SoC

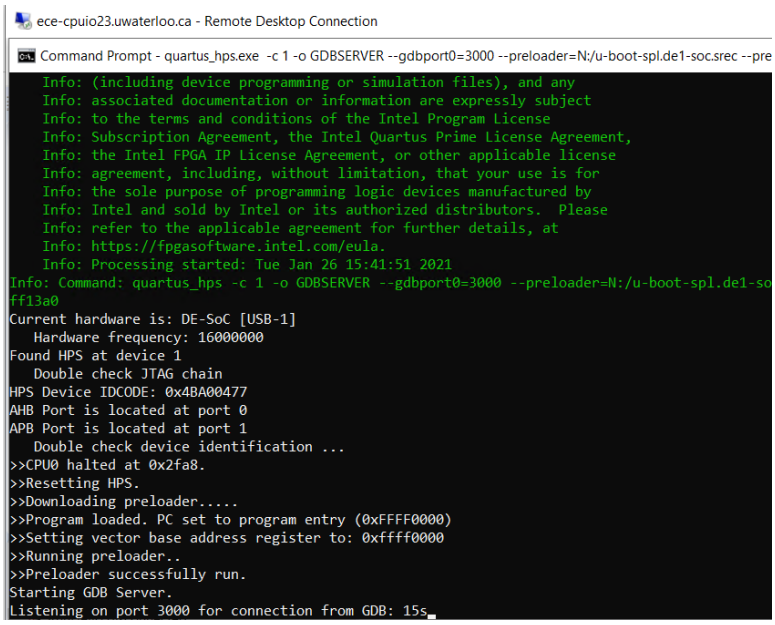


Figure 2.6: ARM DS IDE: Run Preloader

Chapter 3

RTX Overview

3.1 Introduction

In ECE 350 labs, you will design and implement a real-time executive (RTX) on the Intel DE1-SoC board. The DE1-SoC board is powered by Cyclone V SoC chip, which has a dual-core ARM Cortex-A9 Hard Processor System (HPS) and an Altera FPGA. The HPS includes an on-chip RAM of 64 KB and a DDR3 RAM of 1 GB. The board has four Hard Processor System (HPS) timers, two JTAG UARTs and several other peripheral interface devices.

The RTX will provide a basic multi-programming environment that supports priorities, preemption, dynamic memory management, inter-task communications, and basic console I/O. The RTX is designed for a cooperative, non-malicious software environment. The RTX will support privileged and unprivileged modes of computation. Privileged RTX tasks execute under supervisor mode, and unprivileged RTX tasks execute under the user mode of the Cortex-A9 processor.

3.2 RTX Requirements

The RTX requirements are listed as follows.

- **Dynamic memory management.** First-fit dynamic memory allocation will be supported (Chapter (4)).
- **Dynamic task management.** The RTX will support 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 also supports task preemption. Tasks could have different priorities. The RTX will support a simple FIFO (First In, First Out) scheduling policy for each priority level. (Chapter (5)).
- **Inter-task communication and I/O.** The RTX will support mailbox utility for inter-task communication. An interrupt-driven UART will also be supported by the console service. (Chapter (6)).

3.3 RTX Coding

The RTX is expected to have a reasonably lean implementation. No standard C library function will be allowed in the kernel code. The RTX will not support error recovery. It will be assumed that the application programmers deal with errors in their code.

Chapter 4

Memory Management (P1)

4.1 Objective

In this project, you will develop memory management support in kernel. Your will also write test cases for your memory management implementation to evaluate your design. Specifically you will learn:

- How to use the ARM DS IDE to edit, debug, and execute the RTX code,
- How to design and implement data structures and algorithms for a first-fit memory management scheme, and
- To write test cases that exercise your design with appropriate coverage.

4.2 Starter Files

- `scatter_DE1_SoC.sct`: The "scatter file" describes the memory layout of the design target.
- `src/INC`: This directory contains header files with definitions for the RTX API.
  - `common.h`: Contains definitions of common macros and data structures that can be used by the kernel and user programs.
  - `common_ext.h`: Extended header where you can define common macros and data structures.
  - `rtx.h`: Contains function definitions for the RTX API.
- `src/app`: This directory contains test cases.
- `src/board/DE1_SoC_A9`: This directory contains the board support package for the DE1 SoC platform.
- `src/kernel`: This folder contains all the kernel source code.

When making changes to these files, adhere to the following.

Do NOT

- move any file from the `src` directory to any other directories,

- change the file names under the `src` directory,
- make any changes to the contents of the `rtx.h` and `common.h` files,
- change the existing function prototype in the given `k_mem.[ch]` files,
- include any new header files in the `src/app`, and
- modify the `ae.[ch]` files.

You may

- add new self-defined functions to `k_mem.[ch]`,
- create new `.h` and `.c` files[1],
- include newly created `.h` files in `k_mem.c`, or
- put new files in either the `src` directory or other directories you create.

4.3 Preparation

- Read Sections 1, 2, 3, 9, 10, and 11.1 from Introduction to the ARM Processor Using ARM Toolchain [2];
- Read §8.5;
- Read Free-space Management Chapter from OSTEP [5]; and,
- Run `RTX` code on the ARM DE1 SoC development board (see Chapter (2)).

4.4 Assignment

4.4.1 Function Specifications

You will implement dynamic memory management based on first-fit memory allocation scheme. You will first implement a memory-initialization function, which initializes the RTX's memory manager. You will then implement allocation and deallocation 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 test your implementation. Next, we describe the specification of functions to be implemented.

Memory Initialization Function

- **NAME**

  `k_mem_init` - initialize the dynamic memory manager

- **SYNOPSIS**

```c
#include "k_rtx.h"

int k_mem_init();
```

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

- **DESCRIPTION**

  The `k_mem_init()` function 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. As the manager allocates and deallocates memory regions (see `k_mem_alloc` and `k_mem_dealloc`), the memory will be partitioned into free and allocated regions. You need to design appropriate data structures to track free and allocated regions. Note that these data structures will occupy a portion of the free space which is considered as an overhead to each allocation. The size of these data structures need to, therefore, be minimal.

- **RETURN VALUE**

  The function returns `RTX_OK` on success and `RTX_ERR` on failure, which happens if there is no free space in physical memory.

- **DISCUSSION**

  The DE1-SoC has 1 GiB memory ~~starting from physical address of `RAM_START` and ending at physical address of `RAM_END`~~. Your RTX image will occupy some memory starting from ~~this address~~ `RAM_START`. The **end** of your RTX image is the **starting** address of the free space to be managed. The `scatter_DE1_SoC.sct` file makes the linker generate a variable `Image$$ZI_DATA$$ZI$$ZI_Limit` to indicate the end of the OS Image. The end address of the free space to be managed is `RAM_END`. The free space your memory manager will manage starts from the address of this linker defined symbol and ends at ~~`0x400FFFFF`~~ `RAM_END`. You are responsible for designing and implementing data structures used to track free and allocated memory regions. Please note that in Lab 2, you will need to modify your memory manager to track ownership of each allocated memory region.

Allocation Function

- **NAME**

  `k_mem_alloc` - allocate dynamic memory

- **SYNOPSIS**

```c
#include "k_rtx.h"

void *k_mem_alloc(size_t size);
```

- **DESCRIPTION**

  The `k_mem_alloc()` function allocates `size` bytes according to the first-fit algorithm and returns a pointer to the beginning of the allocated memory region. The first-fit iteration should start from the beginning of the free memory region. The `k_mem_init()` should be called before `k_mem_alloc` is called. Otherwise, the function returns `NULL`. The `size` argument is the number of bytes requested. The function 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 `k_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.

Deallocation Function

- **NAME**

  `k_mem_dealloc` - Free dynamic memory

- **SYNOPSIS**

```
#include "k_rtx.h"

int k_mem_dealloc(void *ptr);
```

- **DESCRIPTION**

  The `k_mem_dealloc()` function frees the memory space pointed to by `ptr`, which must have been returned by a previous call to `k_mem_alloc()`. Otherwise, or if `k_mem_dealloc(ptr)` has already been called before to free up the memory space pointed to by `ptr`, the function returns `RTX_ERR`. If `ptr` is `NULL`, no action is performed. If the newly freed memory region is adjacent to other free memory regions, they 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 `RTX_OK` on success and `RTX_ERR` on failure. Failure happens when the RTX cannot successfully free the memory region for some reason (some of which are explained above).

Utility Function

- **NAME**

  `k_mem_count_extfrag` - Count externally fragmented memory regions

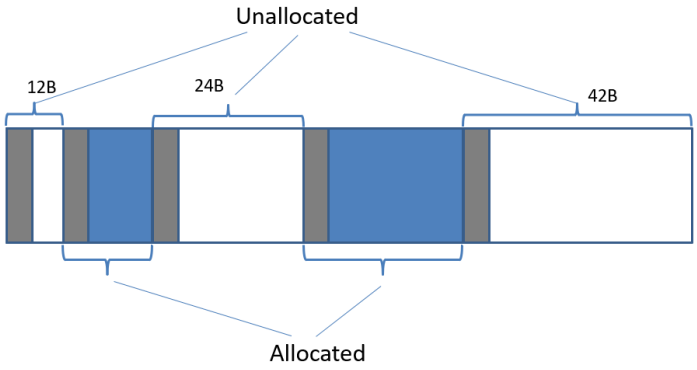
- **SYNOPSIS**

```
#include "k_rtx.h"

int k_mem_count_extfrag(size_t size);
```

- **DESCRIPTION**

  The `k_mem_count_extfrag` function 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 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 `k_mem_count_extfrag` with 12, 42, and 43 as inputs should return 0, 2, and 3, respectively.

4.4.2 Test Cases

In order to test your implementation, you need to write at least three test cases in `src/app/ae_mem.c`. 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). 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 external fragmentation. Allocate and deallocate memory with different sizes and see how external fragmentation is affected. You can use the function `k_mem_count_exfrag()` to quantify the level of external fragmentation.

4.5 Grading

You will have to push your code to your group's repository on Gitlab. We will run several test cases to verify the correctness of your implementation. In `main_svc_cw.c`, the `main` function calls `ae_init` and `ae_start`. These are testing software implements. These functions are responsible for setting up task(s) that will exercise the test cases. You can see the various test cases for this project in `ae_mem.c`. During our testing, the files under the `app` directory will be replaced by more complicated test cases.

4.5.1 Performance Metric

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 N/T. 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. To time your memory manager, you could use the Private A9 timer. For more information on the timer please read Section 2.4 of [1]. Timer functionality is available through `src\board\DE1_SoC\timer.[c|h]` files.

- **Heap utilization ratio**. This metric measures the overhead of the data structures used to implement the memory manager. Let $P = \sum_i^N p_i$ be the total number of bytes allocated after a sequence of N allocation requests (i.e., `k_mem_alloc(p_i)` for $i \in \{1, \ldots, N\}$). Let H be the entire heap size (i.e., initial free memory). The heap utilization ratio for the sequence is defined as P/H. Please note that heap utilization depends on the testing sequence. For it to be meaningful, the allocation sequence should fill up the entire memory. When this happens, heap utilization can be used to measure the overhead of the memory manager. In our test cases, we measure the heap utilization for different allocation sequences that fill up the entire available memory.

4.6 Marking Rubric

The Rubric for marking the submitted source code and report is listed in Table 4.1. The functionality and performance of your implementation will be tested by a third-party testing program. We might also conduct random code inspection.

Points	Description
10	Code compiles without errors
90	Third-party testing Code inspection

Table 4.1: Lab1 Marking Rubric

Chapter 5

Task Management (P2)

Chapter 6

Inter-task Communications and I/O (P3)

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. Visit https://uwaterloo.ca/information-systems-technology/services/virtual-private-network-vpn for 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 a machine under **ECE** → **ece-mcu***. This will download a Remote Desktop File.
- Open this file with a Remote Desktop Client of your choice (Microsoft Remote Desktop can be used on Window and on Mac OS).
- When prompted for user name, input `Nexus\userid`, where the `userid` is your quest ID.
- The password is your Quest password. T

You should be 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

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 [2].

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

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 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 SB TR	Platform register. The meaning of this register is defined by platform standard.
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

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 [3]. It has been extended to support Cortex-A series processors.

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

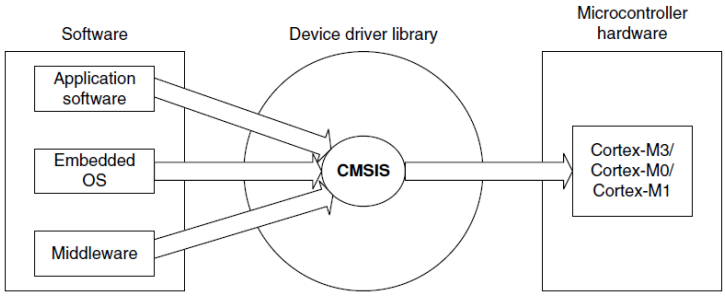


Figure 8.1: Role of CMSIS[6]

- **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.

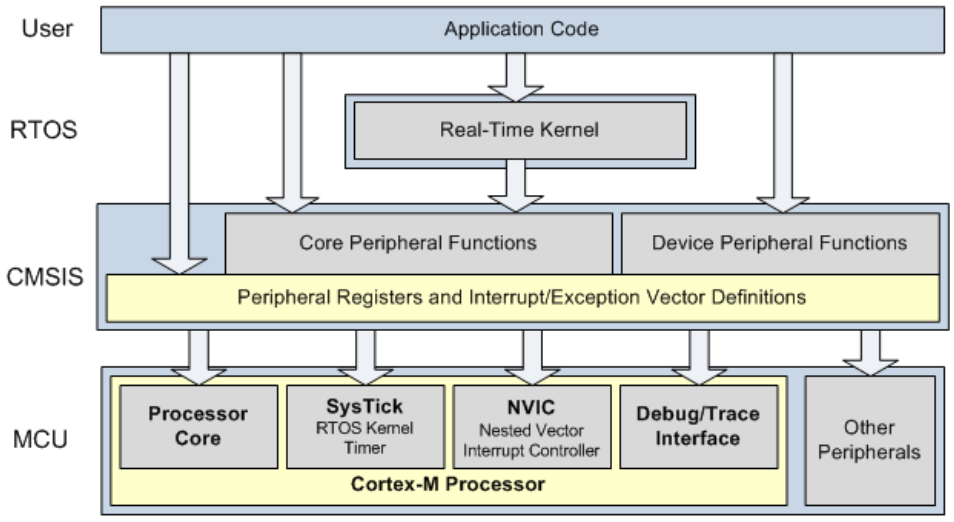


Figure 8.2: CMSIS Organization[3]

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).

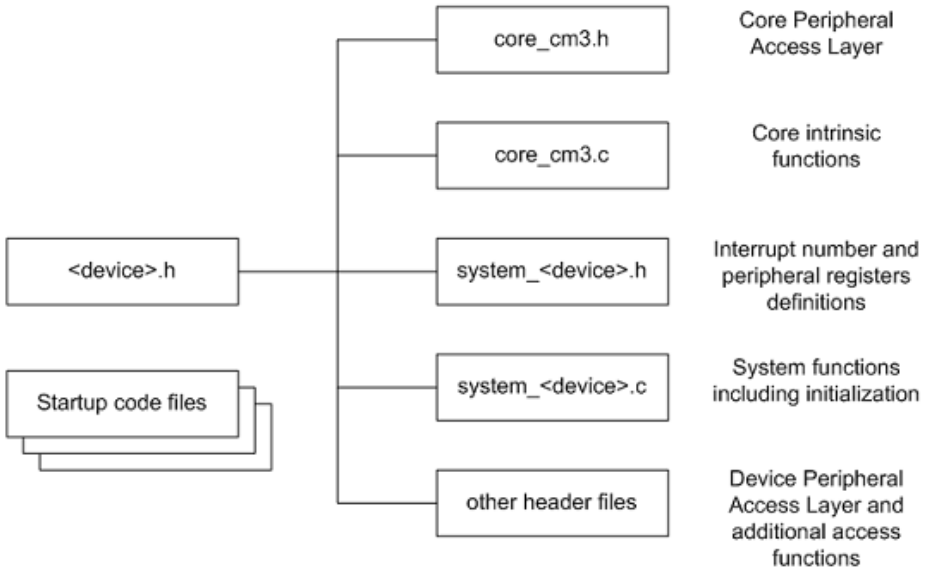


Figure 8.3: CMSIS Organization[3]

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 below.

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

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. 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 following.

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

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

PCB g_pcb;
U32 g_var;
```

```
__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
}
```

- A C function. For instance, `a_c_function` is a function written in C. We can invoke this function in assembly.

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

- A constant expression in the range of $0 - 255$ defined in C. Below, `g_flag` is 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
    ;......
}
```

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.

```
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
}
```

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

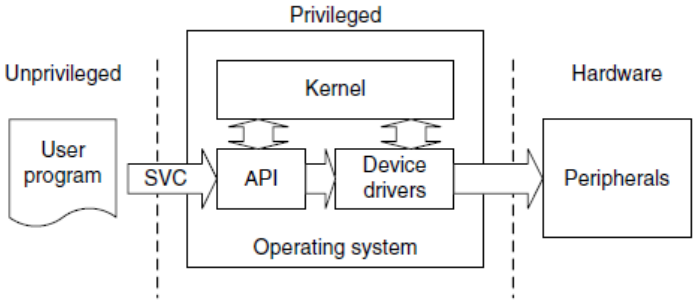


Figure 8.4: SVC as a Gateway for OS Functions [6]

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 below.

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

```
    BX   LR
    ALIGN
}
```

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`. Below 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
}
```

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`[4]. 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` can then 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] DE1-SoC Computer System with ARM* Cortex* A9. https://ece.uwaterloo.ca/~smzahedi/crs/ece350/resources/DE1-SoC_Computer_ARM.pdf.

[2] Intel Corporation FPGA University Program. Introduction to the ARM Processor Using ARM Toolchain. 2019. https://ece.uwaterloo.ca/~smzahedi/crs/ece350/resources/ARM_A9_intro_alt.pdf.

[3] MDK Primer. http://www.keil.com/support/man/docs/gsac.

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

[5] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Operating Systems: Three Easy Pieces. Arpaci-Dusseau Books, 1.00 edition, August 2018.

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