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, October 26, 2020

Contents

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 Keil MCB1700 board populated with an NXP LPC1768 microcontroller.

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: Memory Protection and Stress Testing

Part III is frequently asked questions.

Part IV introduces the computing environment and the development tools. It includes a Keil MCB 1700 hardware and software reference guide. The topics are as follows.

- Keil MCB1700 Hardware Environment
- Keil Software Development Tools
- Programming MCB1700

Acknowledgements

Our project is inspired by the original ECE354 RTX course project created by Professor Paul Dasiewicz. Professor Dasiewicz provided detailed notes and sample code to us. We sincerely thank ARM University Program for providing us with lab teaching materials and development software licenses. We also own many thanks to our students who did ECE354 and SE350 course projects in the past and provided constructive feedback. Professor Sebastian Fischmeister made the Keil Boards donations possible. Our gratitude also goes out to Eric Praetzel who supports the IT infrastructure of RTOS lab hardware and the ARM-MDK software. Special thanks go to our teaching assistants: Ali Hossein Abbasi Abyaneh, Maizi Liao and Weitian Xing. Their talents and strong support tremendously help the delivery of this lab.

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 is due by 8:30 am on Monday of the second academic week (see 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.

- **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 least one week before the nearest lab project deadline.

Deliverable	Weight	Due Date	File Name
P0 Group Sign-up	2%	08:30 Sep 14	
P1 Memory Management	18%	20:30 Sep 25	p1_Gid.zip
P2 Task Management	20%	20:30 Oct 09	p2_Gid.zip
P3 Synchronization and Console I/O	25%	20:30 Nov 02	p3_Gid.zip
P4 Timing and Real-Time Scheduling	20%	20:30 Nov 16	p4_Gid.zip
P5 Memory Protection and Stress Tests	15%	20:30 Nov 30	p5_Gid.zip

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

Lab Project Submission and Grading 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 and a lab report (in pdf) file. Create a directory and name it "labN", where N is 1, 2, ..., 5. Create a sub-directory named "code". Put your uVision 5 application folder under the code directory. Name your lab report file "pN_report.pdf", where N is 1, 2, ..., N 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.

- **Project Grading.** Submissions will be evaluated on the board connected to a Microsoft Windows 10 lab machine. Lab machines are accessible through ENGLab remote desktop session when connected to the campus virtual private network (VPN). A **15%** penalty will be applied to a deliverable that is only able to function inside the simulator but not on the actual hardware.

  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 group member received and assign individual lab grade to each team member

Peer Rating	Contribution Factor CF
[8, 10]	100%
[7, 8)	80%
[6, 7)	60%
[4, 6)	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).

by multiplying the project grade with a contribution percentage factor listed in Table 0.2.

- **Project Re-grading.** To initiate a re-grading process, contact the grading TA in charge first. The re-grading is a rigid process. The entire project will be re-graded. Your new grades may be lower, unchanged or higher than the original grade received. If you are still not satisfied with the grades received after the re-grading, escalate your case to the lab instructor to request a review and the lab instructor will finalize the case by re-grading the entire project a second time.

- **Late Submissions.** Late submission is accepted within three days after the deadline of the lab project. There are *five grace days*[1] 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.* Please be advised that to simplify the book-keeping, 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.

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.

[1]Grace days are calendar days. Days in weekends are counted.

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 [2]. *There is no guarantee on the response time to questions of a lab that passes the submission deadline*.

- **Office Hours.** The Learn system calendar gives the office hour details.

- **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.

  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.

[2] 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.

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 Keil MCB1700 board populated with an NXP LPC1768 microcontroller . The executive will provide a basic multiprogramming environment, with five priority levels, preemption, dynamic memory management, shared memory and semaphore for inter-task communication and synchronization, a basic timing service, system console I/O and debugging support.

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. Applications and non-kernel RTX tasks must execute in the unprivileged level of LPC1768. The RTX kernel will execute in the privileged level. There is 32K of RAM for use by the RTX and application tasks. The microcontroller has four timers, four UARTs and several other peripheral interface devices. The board has two RS-232 interfaces, from which 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 semaphore utility for inter-task communication 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 system tasks.

User Tasks
These tasks are operating at a unprivileged level in thread 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 Lab3, you will add a wall clock user task and a set process priority task once you have the console I/O service ready.

System Tasks
These tasks are operating in thread mode. Some may require a privileged level of operation and some may be sufficient to operate at a unprivileged 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 to introduce kernel programming for the Keil MCB1700 board. You will become familiar with the Keil uVision5 IDE (Integrated Development Environment). You will create a set of memory management system calls that are part of the services that your RTX provides. After this lab you will be able to answer the following questions:

- How to use the Keil uVision5 IDE to edit, debug, simulate and execute a bare-metal uVision project
- How to use SVC as a gateway to program a system call in the kernel space for ARM Cortex-M3 processor
- How to design and implement first fit memory management data structure and algorithm

2.2 Starter Files

The starter file is on GitHub at http://github.com/yqh/ece350/manual_code/. It contains the following files:

- `util/printf_uart`: printf source code and the uart polling source code. The printf outputs to the UART1 by polling.
- `util/debug_script`: RAM.ini that initializes debugger to load code for in-memory execution.

- `HelloWorld/`: a bare-metal project that outputs strings to UART0 and UART1 by polling.
- `SVC/`: a bare-metal project that uses the SVC as a gateway to transfer from user space to kernel space. It is the skeleton project file for your lab1. Aside from the printf and uart polling source code, it contains the following under the `src/` folder:
  - `k_mem.h`: the kernel memory management routine header file;
  - `k_mem.c`: the kernel memory management C source code template file to be completed by students;
  - `main_svc.c`: a main method file for writing testing cases;
  - `common.h`: the header file both the kernel and the user can include; and
  - `rtx.h`: the user RTX API file. You should not modify this file.

2.3 Pre-lab Preparation

- Create a Hello World application for Keil MCB1700 using MDK uVision (See Chapter 9)
- Read "Keil MCB1700 Hardware Environment" in Chapter 8
- Read "SVC Programming: Writing an RTX API Function" in Section 10.5.
- Execute the `HelloWorld` project in the simulator and the board.
- Execute the `SVC` project in the simulator and on the board.
- Read "Free-Space Management" at http://pages.cs.wisc.edu/~remzi/OSTEP/vm-freespace.pdf.

2.4 Assignment

There is a programming project and a report. The report documents the data structures, algorithms and testing scenarios of the programming project.

2.4.1 Programming Project

You are to implement the first fit memory allocation algorithm. You will first implement a memory initialization function, which initializes the RTX with the smallest size of memory manageable. Then you will implement an allocation function and a de-allocation function. One utility function will also be implemented to help analyze the efficiency of allocation algorithm and its implementation. You will also write tests 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(size_t blk_size, int algo);
```

DESCRIPTION

The `mem_init()` system call initializes the system's memory manager with the smallest memory block size that the manager is able to manage and the allocation algorithm used.

The granularity of the memory that is manageable is determined by the blk_size. During initialization, the memory manager divides the space into blk_size blocks. The blk_size is a multiple of word size. In our case, we have a 32 bit machine. Our word size is 4 bytes. Any management data structure placed inside the manageable memory block is counted into the blk_size.

The input parameter algo specifies the memory allocation algorithm. The full list of memory allocation algorithms are as follows:

FIRST_FIT

The first fit memory allocation algorithm is used.

RETURN VALUE

The function returns `0` on success and `-1` on failure. For example, some causes of failure could be that the blk_size value is too small or an unsupported memory allocation algorithm is specified.

The system call traps into the kernel and then initializes the memory allocation data structure based on the specified memory allocation algorithm. You are responsible for designing and implementing data structures used to track what memory is currently allocated and what is free[1].

Allocation Function:

NAME

[1]In Lab2, where multi-tasking is supported, you will also need to start making your memory management data structures to track which task holds which allocated memory. This is not required in Lab1 since multi-tasking is not in your kernel yet.

`mem_alloc` - allocate dynamic memory

SYNOPSIS

```
#include "rtx.h"

void *mem_alloc(size_t size);
```

DESCRIPTION

The `mem_alloc()` system call allocates size bytes and returns a pointer to the allocated memory [2]. The allocated memory is not initialized. If size is 0, then `mem_alloc()` returns NULL.

The input parameter size is the number of bytes requested from the allocator. The allocator then returns the starting address of a consecutive region of memory of the appropriate size. The memory address should be four bytes aligned.

Memory requests may be of any size from one byte all the way up to the maximum size of the physical memory on the board. Because the `mem_init()` divides the space into equally spaced memory blocks to manage, you may need to return multiple of these blocks that are in a consecutive location in the memory if one block is not big enough to serve the requested size. In case of multiple blocks in a consecutive location is returned, there will be additional space that is not asked by the caller. This space is the internal fragmentation (and the user will not be told).

RETURN VALUE

The function returns a pointer to the allocated memory or NULL if the request fails.

Deallocation Function:

NAME

`mem_dealloc` - Free dynamic memory

SYNOPSIS

```
#include "rtx.h"

void mem_dealloc(void *ptr);
```

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()`.

[2] The `mem_init()` needs to be invoked before calling `mem_alloc()`.

Otherwise, or if `mem_dealloc(ptr)` has already been called before, undefined behaviour occurs. If ptr is NULL, no operation is performed.

If the freed memory block is adjacent to other free memory blocks, it is merged with them immediately (i.e. immediate coalescence) and the combined block is then re-integrated into the memory under management. You are not required to clear the block (that is, to fill the memory with zeros).

RETURN VALUE

This function returns no value.

Utility Function:

NAME

`mem_count_extfrag` - Count external fragmented memory blocks

SYNOPSIS

```
#include "rtx.h"

int mem_count_extfrag(size_t size);
```

DESCRIPTION

This system call counts the number of free (i.e. unallocated) memory blocks in memory that are of a size less than the input value of size. The input value size is in bytes. The space your structures need to maintain un-allocated (free) memory is considered as free in this context.

Testing

In order to test your implementation of the required functions, write an application to allocate and de-allocate memory in different ways and possibly multiple times. The provided `main_svc.c` file is for writing your testing code. Create a set of testing scenarios to verify functions implemented. Document the testing specification in the report (See 2.4.2). Implement the tests in `main_svc.c` file.

There is no hard requirement on what tests to be implemented. The rule of thumb is that the tests will convince yourself that your implementation is correct. However to provide more leads to testing ideas, you may want to consider repeatedly requesting and then releasing 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 test you want to include is about the external fragmentation. Allocate and de-allocate memory with different sizes and see how external fragmentation is affected. The utility function `mem_count_exfrag()` is a useful tool.

Since the tests have no knowledge of your detailed internal design, they only invoke the functions specified by the RTX API. We require the testing results to comply with the following format and you output the results to the UART terminal by polling (i.e. UART1):

```
Gid_test: START
Gid_test: test n OK
Gid_test: test m FAIL
Gid_test: x/N tests OK
Gid_test: y/N tests FAIL
Gid_test: END
```

For example, assume that you are in group G099 and you have 3 testing cases in total. If two of the testing cases pass and one of the testing cases does not pass, the final testing results should be output to the putty terminal as follows:

```
G099_test: START
G099_test: total 3 tests
G099_test: test 1 OK
G099_test: test 2 OK
G099_test: test 3 FAIL
G099_test: 2/3 tests OK
G099_test: 1/3 tests FAIL
G099_test: END
```

Performance

Three metrics are used to measure the performance of implementation.

- **Throughput**. Let T be the time that a sequence of N requests are completed. A request can be an allocation request or a deallocation request. Throughput is defined as

$$R_T = \frac{N}{T} \, . \tag{2.1}$$

  For example, assume in one second, the system served 100 allocation requests and 100 deallocation requests. The Throughput is 200 operations per second.

- **Heap utilization ratio**. This metric is to measure the overhead of the memory management data structures. Given a request sequence and let P be the accumulated bytes that user space can see and H be the entire heap size. The heap utilization ratio is defined as

$$R_H = \frac{P}{H} \, . \tag{2.2}$$

- **Image footprint ratio**. Embedded systems have very limited memory resources. We want our operating system to be as small as possible. Assume your OS image size is F bytes, the image footprint ratio is defined as

$$R_F = \frac{P}{F} \,. \tag{2.3}$$

2.4.2 Report

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

- Statement of the problem to be discussed in the report
- Descriptions of the data structures and algorithms used to implement the allocation strategy
- Testing scenario descriptions

To illustrate key algorithm, pseudocode is a good way to describe the algorithm from a high-level point of view. For testing, include five or more non-trivial testing scenarios.

2.4.3 Third-party Testing and Source Code File Organization

We will write a third-party testing program to verify the correctness of your implementation of the functions. In order to do so, we will need to enforce certain source code file organization standard. You should keep the file organization in the provided skeleton code, which has a src sub-directory inside. There are dos and don'ts you need to follow.

Don'ts

- Keep all existing files inside the src directory. Do not move any one of them to other directories.
- Do not change the file names under the src directory.
- Do not make any changes of the contents of the `rtx.h` file.
- Do not change the existing function prototype in the given `k_mem.[ch]` files.
- Do not include any new header files in the `main_svc.c`.

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 [3].
- 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 the memory management functions you will implement. During the third-party testing, the `main_svc.c` file will be replaced by a third-party testing code with the same file name. Using the provided template `main_svc.c` to write your own testing code.

2.5 Deliverable

2.5.1 Pre-Lab Deliverables

Fill the project_manager.csv template file and submit it to Lab1 Dropbox on Learn.

2.5.2 Post-Lab Deliverables

Create a directory named "lab1". Then create a sub-directory named "code" under "lab1". Put your uVision 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 is lost or extra memory appears after repeating allocation and de-allocation function calls. We will also conduct manual random code inspection.

[3] 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.

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

Table 2.1: Lab1 Marking Rubric

Chapter 3

Lab2 Task Management

3.1 Objective

This lab is to learn about, and gain practical experience in task management kernel programming. In particular, you will implement system calls to make the RTX a multi-tasked kernel. You will also implement a utility system call to return task information from the RTX at runtime to the users. In this lab, you will design and implement a preemptive multi-tasked kernel for single Cortex-M3 processor board, such as MCB1700. More specifically, you will learn:

- How to design and implement a multi-tasked kernel
- How to design and implement kernel support for task priority
- How to design and implement kernel support for task preemption

3.2 Starter Files

The starter file is uploaded on GitHub and is available at https://github.com/yqh/ece350/tree/master/manual_code/Context_Switching. The uploaded code is a rudimentary kernel project that supports context switching between two **kernel** tasks running at privileged thread mode and using statically allocated kernel stacks.

Important Note

In the starter code, all tasks are kernel tasks. Kernel tasks only use kernel stacks. You will implement user-space tasks. User-space tasks must have both kernel and user-space stacks (MSP and PSP should be set differently). The kernel stack is statically allocated in kernel code and only has to be assigned to each created task. The user-space stack, however, must be allocated dynamically when a task is created. Your

tcb (task-control-block) structure should include pointers to both stacks. The user-space stack must be used only for user-space code and the kernel stack must be only used for kernel code (e.g., system calls). This means that you need to carefully set processor's sp to point to PSP or MSP depending on the code that is to be executed.

3.3 Pre-lab Preparation

- Review the lecture notes on creating thread, context switching between threads, and yielding the processor.
- Review processor operation mode in 8.2.2, processor's two stacks in 8.2.3 and exception return privilege level and stack setting in 8.4.3.
- Read C and assembly programming in Section 10.4.
- Run the given https://github.com/yqh/ece350/tree/master/manual_code/Context_Switching starter project on the simulator and then on the real hardware.
- Work through the context switching code and understand what they do and how they work.

3.4 Assignment

You should submit your programming project and a report. The report should clearly document all the data structures, algorithms, and testing scenarios that are used in your project.

3.4.1 Programming Project

You should design and implement a priority-based preemptive multi-tasked kernel. The maximum number of (kernel and user) tasks that could co-exist in the system should be specified at compile time. First, you will implement the `rtx_init()` system call which initializes the kernel with the memory initialization parameters and a list of tasks that should be created at boot time. Next, you will implement `tsk_create()` and `tsk_exit()` system calls to create and terminate tasks at runtime. Next, you will then implement `tsk_yield()` and `tsk_set_prio()` system calls to add support for processor and task priority management. You will also implement `tsk_get()` system call that returns task information. You will enhance the `mem_alloc` and `mem_dealloc` functions you implemented in lab1 so that the kernel keeps track of the ownership of each allocated memory (i.e., the task that invokes `mem_alloc` owns the allocated memory). When a task invokes `mem_dealloc` , it will fail if the input memory is not owned by the calling task.

Finally, you will create a number of unprivileged user-space tasks – a "null task" and some testing tasks to verify your design and implementation.

Macros and User Task Data Structure

In the starter code, we have a `common.h` file. This file contains data structures and macro definitions that can be used by both kernel and user-space tasks. The file is included by the `rtx.h` in the user space and by the `k_rtx.h` in the kernel space. For this project, the relevant macros are as follows.

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

/* Task Priority. The bigger the number is, the lower the priority is*/
#define HIGH 0
#define MEDIUM 1
#define LOW 2
#define LOWEST 3
#define PRIO_NULL 4 /* hidden priority for null task */

/* task state macro */
#define DORMANT   0  /* terminated task state*/
#define READY     1
#define RUNNING   2
#define NEW       15
```

An important data structure is the `rtx_task_info`.

```
typedef struct rtx_task_info {
   void (*ptask)(); /* Task entry address          */
   U32  k_sp;      /* The task current kernel stack pointer */
   U32  k_stack_hi; /* The kernel stack starting addr. (high addr.)*/
   U32  u_sp;      /* The task current user stack pointer */
   U32  u_stack_hi; /* The user stack starting addr. (high addr.) */
   U16  k_stack_size; /* The task total kernel stack space in bytes */
   U16  u_stack_size; /* The task total user stack space in bytes*/
   task_t tid;     /* Task ID                     */
   U8   prio;      /* Execution priority          */
   U8   state;     /* Task state                  */
   U8   priv;      /* = 0 unprivileged, =1 priviliged */
} RTX_TASK_INFO;
```

Note that for each unprivileged user-space task, you are required to have two separate stacks: a user-space stack and a kernel stack. For each privileged kernel task, you only need a kernel stack.

Description of Functions

Specifications of each function to be implemented are described below:

RTX Initialization Function

NAME

`rtx_init` - initialize the kernel

SYNOPSIS

```
#include "rtx.h"

int rtx_init(size_t blk_size, int algo,
             RTX_TASK_INFO *tasks, int num_tasks);
```

DESCRIPTION

The `rtx_init` function initializes the kernel. It then creates `num_tasks` tasks to be executed. The `blk_size` argument specifies the smallest manageable memory size under the RTX management. The `algo` argument specifies the memory allocation algorithm. The `tasks` argument points to a memory location of an array of `num_tasks` `RTX_TASK_INFO` elements. The maximum number of tasks that can co-exist in the kernel including the null task is set at compile time by the `MAX_TASKS` macro defined in the `common.h` file. All kernel stacks and kernel task-control data structures are statically allocated at compile time. Kernel stacks for all (kernel and user-space) tasks have the same size that is determined by the `KERN_STACK_SIZE` macro defined in `common.h`.

RETURN VALUE

The `rtx_init` function returns `0` on success and `1` on failure. On success, the kernel will execute one of the tasks identified by the `tasks` argument. Which task to run is controlled by the scheduling policy of the kernel. When input parameters are invalid, the function fails.

Task Creation Function The kernel has one primitive to create a user-space unprivileged task at runtime. Each task is uniquely identified by a task ID. The task ID is an integer value of $0, 1, 2, \ldots, N-1$, where N is the maximum number of tasks (including the null task) the kernel supports and is decided by the `MAX_TASKS` macro defined in the `common.h`. The task ID 0 is reserved for the null task (see 3.4.1).

NAME

`tsk_create` - create an unprivileged task

SYNOPSIS

```
#include "rtx.h"

int tsk_create(task_t *task, void (*task_entry) (void),
               U8 prio, U16 stack_size);
```

DESCRIPTION

The `tsk_create()` system call adds a new user-space unprivileged task to the system. When executed, the new task starts from `task_entry()`; The `prio` argument sets the initial priority of the new task. There are four user-visible priorities – `LOWEST`, `LOW`, `MEDIUM` and `HIGH`, which are macros that are defined in the `commoh.h`. The `stack_size` argument specifies the user stack size in bytes. The kernel is responsible for allocating the space for the user-space stack and freeing the stack space when the task terminates. The function should be non-blocking. However, if priority of newly-created task is higher than other tasks, the newly-created task should preempt all other tasks and start running immediately. The task must invoke `tsk_exit()` before it terminates. Before returning, a successful call to `tsk_create()` stores the ID of the new task in the buffer pointed to by `task`.

RETURN VALUE

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

The system call traps into the kernel and sets up the task-control-block data structure for the new task. You are responsible for designing and implementing the task-control-block data structure. The structure will be used to track task information including, but not limited to, the state of task, user stack pointer and kernel stack pointer.

Task Termination Function The kernel has one primitive to terminate a task at runtime.

NAME

`tsk_exit` - terminate the calling task

SYNOPSIS

```
#include "rtx.h"

void tsk_exit(void);
```

DESCRIPTION

The `tsk_eixt()` system call stops and deletes the currently running task. Once a task is terminated, its state becomes `DORMANT` if its task-control-block data structure still exists in the system.

RETURN VALUE

The function does not return.

Processor Management Function The kernel has one system call to yield the CPU.

NAME

`tsk_yield` - yield the processor

SYNOPSIS

```
#include "rtx.h"

int tsk_yield(void);
```

DESCRIPTION

The `tsk_yield()` system call enables a task to relinquish the CPU. The task is moved to the ready queue and the kernel will make new scheduling decision to determine the next task to run.

RETURN VALUE

The function returns 0 on success and -1 on failure.

Task Priority Function

The RTX scheduling policy is priority based. You need to design and implement a primitive to set the priority of the task.

NAME

`tsk_set_prio` - set task priority at runtime

SYNOPSIS

```
#include "rtx.h"

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

DESCRIPTION

The `tsk_set_prio()` system call changes the priority of the task identified by `task_id` to `prio`. The full list of task priority values are `HIGH`, `MEDIUM`, `LOW`, `LOWEST`, and `PRIO_NULL`. The priority level `PRIO_NULL` is reserved for the null task and cannot be assigned to any other task.

An unprivileged task may change the priority of any other unprivileged task (including itself). A privileged task may change the priority of any other task (including itself). The priority of the null task cannot be changed and remains at level `PRIO_NULL`.

The caller of this primitive never blocks, but could be preempted. If the value of `prio` is higher than the priority of the current running task, and the task identified by `task_id` is ready to run, then the task identified by the `task_id` preempts the current running task. Otherwise, the current running task continues its execution.

RETURN VALUE

The function returns 0 on success and -1 on failure. Example causes of failure could be an invalid task ID or an invalid priority level.

Task Utility Function

NAME

`tsk_get` - obtain task status information from the kernel

SYNOPSIS

```
#include "rtx.h"

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

DESCRIPTION

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

RETURN VALUE

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

Memory Management Functions You will add the notion of ownership to the memory management module in your RTX. When a task invokes `mem_alloc` and the system returns a valid memory address to it, the returned memory block is owned by the calling task. Only the owner of a memory block can successfully deallococate that memory block. If a task calls `mem_dealloc` with a memory that it does not own, the function will return -1. Other functionalities of the memory management functions remain the same as described in lab1 manual.

Required Tasks

The Null Task

The kernel has to run a task at any given time. If there are no ready task, then kernel will run the unprivileged task named `null task`, which is created at boot time (the user won't be told that this task exists). The null task operates at the priority level `PRIO_NULL`. The `PRIO_NULL` is a hidden priority level reserved for the null task only. Task ID 0 is reserved for the null task. So when there is no user-space tasks that the kernel can execute, then the `null task` is scheduled to run. Initially, the following pseudo-code can be used to design the `null task`:

```
loop forever
      relinquish the CPU
end loop
```

Once your kernel supports preemption, 'relinquish the CPU' line could be removed from the infinite loop. You may want to think why this is true.

User Test Tasks In order to test your implementation of the required functions, write an application that uses your kernel primitives. The provided `main_svc.c` and `usr_tasks.[ch]` files are for writing test codes. Create a set of testing scenarios to test your implemented functions and document the testing specification in the report (See 2.4.2). Implement the tests in `main_svc.c` and `usr_tasks.[ch]` files.

There is no hard requirement on what tests to be implemented. The rule of thumb is that the tests should be comprehensive enough to convince you that your implementation is correct. However, to provide more leads to testing ideas, you may want to consider repeatedly creating and then terminating tasks while making sure that no extra task is created or no task gets lost. The most important objective is to insure that tasks are able to resume correctly when they get blocked and later on become unblocked. Another testing objective that you may want to consider is preemption. You could create multiple tasks with different priorities and change their priority at runtime to test preemption.

The utility functions `mem_count_extfrag` and `tsk_get()` are useful tools for checking system memory and task status information.

Similar to lab1, you will use the polled UART terminal to output the testing results, and you will format the testing results as described in *Testing* subsection in Section 2.4.1.

Performance

Context switching time is the metric we use to measure the performance. We will measure the following times:

- The time from the `task_yield()` being invoked until the next ready to run task being executed
- The time from the `set_prio()` being invoked until the next ready to run task being executed
- The time from the `tsk_create()` (with lower priority than itself) being invoked until the system call returns and the calling task resumes its execution

3.4.2 Report

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

- Statement of the problem to be discussed in the report
- Descriptions of the data structures and algorithms used to implement all the functions listed in Section 3.4.1
- Testing scenario descriptions

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

3.4.3 Third-party Testing and Source Code File Organization

We will write a third-party testing program to verify the correctness of your implementations. In order to do so, we will need to enforce certain source code file organization standards. You should keep the file organization in the provided skeleton code, which has a src sub-directory inside. There are dos and don'ts you need to follow.

Don'ts

- Keep all existing files inside the src directory. Do not move any one of them to other directories.
- Do not change the file names under the src directory.
- Do not make any changes to the content of the `rtx.h` file.
- Do not change the existing function prototypes in the `k_mem.[ch]` and `k_task.[ch]` files.
- Do not include any new header files in the `main_svc.c`.

Dos

- You are allowed to add new self-defined functions to `k_mem.[ch]` and `k_task.[ch]` files.
- You are also allowed to create new `.h` and `.c` files [1].
- The newly created `.h` files are allowed to be included in the `k_mem.c` and `k_task.c` files.
- Any new file 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 system call functions defined in the `rtx.h`. During the third-party testing, the `main_svc.c` file will be replaced by a third-party testing code with the same file name. Use the provided template `main_svc.c` to write your own test codes.

3.5 Deliverables

3.5.1 Pre-Lab Deliverables

Fill the project_manager.csv template file and submit it to Lab2 Dropbox on Learn.

3.5.2 Post-Lab Deliverables

Create a directory named "lab2". Then create a sub-directory named "code" under "lab2". Put your uVision Project folder under "lab2/code". Put the `p2_report.pdf` under the "lab2" directory. Include a README file with group identification, project manager name and a description of directory contents. Put the README file under the "lab2" directory. Zip everything inside the lab2 directory and submit it to Learn Lab2 Dropbox.

3.6 Marking Rubric

The Rubric for marking the submitted source code and report is listed in Table 3.1. The functionality and performance of your implementation will be tested by a third-party test program. A minimum of **20 points** will be deducted if memory is lost or extra memory appears after calls to allocate and de-allocate memory. A minimum of **20 points** will be deducted if tasks are lost or extra tasks mysteriously appear after

[1] 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.

calls to create and delete tasks. A minimum of **30 points** will be deducted if tasks cannot be resumed correctly. Your grade will be relative to the amount of error the third-party testing program has identified.

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

Table 3.1: Lab2 Marking Rubric

Chapter 4

Lab3 Inter-task Communications and Console I/O

In this lab you will work on inter-task communications and interrupt-driven I/O. In particular, you will design and implement system calls to create and manage task mailboxes. You will also design and implement daemon tasks and a UART interrupt handler to enable the RTX console. After this lab, you will learn:

- How to design and implement mailbox API to support inter-task communications.
- How to block and unblock a task.
- How to design and implement interrupt-driven I/O services.

4.1 Starter Files

The starter file is uploaded on GitHub and is available at https://github.com/yqh/ece350/tree/master/manual_code/Context_Switching_UART0_IRQ. The uploaded code is a rudimentary kernel project that supports context switching between two **kernel** tasks. The kernel tasks run in privileged mode and use statically allocated kernel stacks. A context switch happens when "s" key is pressed on the keyboard or when a task relinquishes the CPU by calling `tsk_yield()`.

Additionally, a separate UART_IRQ project is available at https://github.com/yqh/ece350/tree/master/manual_code/UART0_irq, which demonstrates how to echo a key press.

Important Note

The kernel should implement a simple priority-based, first-come-first-served scheduling policy. This means that the ready queue for the processor sorts ready-to-run tasks

based on their priorities. If there are multiple tasks with the same priority, they are sorted based on first-come-first-serve policy. Every time that the kernel needs to make a scheduling decision, it picks the task on top of the queue. The kernel has to make scheduling decisions every time the state of a task changes (e.g., a task is created, or a task exists, or a task yields, or a task's priority changes, or a task is blocked).

Additionally, similar to Lab 2, in the starter code, all tasks are kernel tasks. Kernel tasks only use kernel stacks. You will have to implement user-space tasks with both kernel and user-space stacks (i.e., MSP and PSP should be set differently). The kernel stack is statically allocated in kernel code and only has to be assigned to each newly-created task. The user-space stack, however, must be allocated dynamically when a task is created. Your tcb (task-control-block) structure should include pointers to both stacks. The user-space stack must be used only for user-space code and the kernel stack must be only used for kernel code (e.g., system calls or interrupt handler). This means that you will need to carefully set processor's stack pointer to point to PSP or MSP depending on the code that is to be executed.

4.2 Pre-lab Preparation

- Refresh your memory on inter-process communications (ECE 252).
- Refresh your memory on producer-consumer design pattern and ring buffer (i.e., circular queue).
- Read Chapter 14 of [4]
- Run the given https://github.com/yqh/ece350/tree/master/manual_code/Context_Switching_UART starter project on the simulator and the real hardware.
- Work through the context switching and the UART_IRQ code and understand what they do and how the work.

4.3 Assignment

You should submit your programming project and a report. The report should clearly document all the data structures, algorithms, and testing scenarios that are used in your project.

4.3.1 Programming Project

Overview

The first objective is to make the RTX support message-based inter-task communication. Tasks will be able to request a mailbox to receive messages from other tasks. Sending and receiving messages to and from other tasks will be done through system calls. The second objective is to enable the RTX console. The console will be used to provide direct communication with the RTX and running tasks on the microcontroller.

In this lab, we work with two external devices: a keyboard and an monitor. These two devices communicate serially with the microcontroller using the receive and transmit lines of one of the two RS-232 ports. The RTX will run two daemon tasks– the Keyboard Command Decoder (KCD) task and the LCD task. These two daemon tasks work in cooperation with the UART0 interrupt handler and never terminate. Sending and receiving characters to and from serial port will trigger UART0 interrupts. The UART0 handler forwards received characters from the keyboard to the KCD task. It also responds to UART0 output requests that are received from the LCD task by transmitting characters to serial port. Other tasks can communicate with the LCD task to output characters to a terminal emulator (e.g. PuTTy) that runs on the host computer.

Description of Functions

Specifications of each function to be implemented are described below.

Mailbox Creation Function

NAME

`mbx_create` - create a mailbox

SYNOPSIS

```
#include "rtx.h"

int mbx_create(size_t size);
```

DESCRIPTION

Upon calling `mbx_create`, kernel creates a mailbox for the calling task. Each task is allowed to have only one mailbox. The `size` argument specifies the capacity of the mailbox in bytes. This capacity is used for the messages and any meta data that kernel might need for each message to manage the mailbox. Each mailbox serves messages using the first-come-first-served policy. Once a task exits, the memory for its mailbox must be deallocated by the kernel.

RETURN VALUE

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

- The calling task already has a mailbox.
- The `size` argument is less than `MIN_MBX_SIZE`.
- The `size` argument is more than available memory at run time.

Please note that any task can send messages to a mailbox. However, only one task can receive messages from each mailbox. This is a classical multiple-producer-single-consumer problem. For this problem, implementing the mailbox as a ring buffer (i.e., circular queue) is strongly encouraged.

Send Message Function

NAME

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

SYNOPSIS

```
#include "rtx.h"

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

DESCRIPTION

The `send_msg` function delivers the message specified by `buf` to the mailbox of the task identified by the `receiver_tid`. If the task identified by the `receiver_tid` is blocked on its mailbox (i.e., task's state is `BLK_MSG`), the state of task should be changed to `READY`, and the task should be put back onto the ready queue. The message starts with a message header followed by the actual message data (see Figure 4.1).

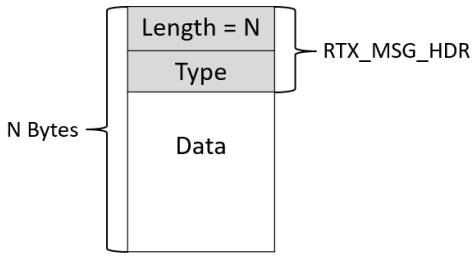

Figure 4.1: Structure of a message buffer

The message header data structure is as follows.

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

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

`DEFAULT`

A general purpose message.

`KCD_REG`

A message to register a command with the KCD task (see Section 4.3.1).

`KCD_CMD`

A message that contains a command to be handled by the receiving task (see Section 4.3.1)

`DISPLAY`

A message to display data on the console.

`KEY_IN`

A message that contains an input key (does not need to be only one character, e.g., control keys) from keyboard.

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

RETURN VALUE

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

- The task identified by the `receiver_tid` does not exist or is in DORMANT state.
- The task identified by the `receiver_tid` exists but does not have a mailbox.
- The `buf` argument is a null pointer.
- The length field in the `buf` specifies a size that is less than `MIN_MSG_SIZE` (macros are defined in the `common.h`).
- The `receiver_tid`'s mailbox does not have enough free space for the message.

Receive Message Function

NAME

`recv_msg` - receive a message

SYNOPSIS

```
#include "rtx.h"

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

DESCRIPTION

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

RETURN VALUE

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

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

Task Utility Function

NAME

`tsk_ls` - return the task IDs of tasks that are not in DORMANT state

SYNOPSIS

```
#include "rtx.h"

int tsk_ls(task_t *buf, int count);
```

DESCRIPTION

The `tsk_ls` function returns the task ID of the first `count` tasks (in decreasing order of task IDs) in the system that are not in the DORMANT state.I The `buf` points to a memory location allocated by the calling task.

RETURN VALUE

On success, the `tsk_ls` function returns minimum of `count` and the number of non-DORMANT tasks in the system. On failure, the function returns `-1`. Possible causes of failure are listed below.

- The buffer pointed by `buf` is a null pointer.
- The `count` is non-positive.

Mailbox Utility Function NAME

`mbx_ls` - list task IDs of tasks that have mailbox

SYNOPSIS

```
#include "rtx.h"

int mbx_ls(task_t *buf, int count);
```

DESCRIPTION

The `mbx_ls` function returns the task ID of the first `count` tasks (in decreasing order of task IDs) in the system that are not in the DORMANT state and have a mailbox. The `buf` points to a memory location allocated by the calling task.

RETURN VALUE

On success, the `mbx_ls` function returns minimum of `count` and the number of non-DORMANT tasks in the system with a mailbox. On failure, the function returns `-1`. Possible causes of failures are listed below.

- The buffer pointed by `buf` is a null pointer.
- The `count` is non-positive.

System Console I/O Tasks

In this Lab, you will also enable the RTX console to accept input commands from the keyboard. A command starts with symbol `%` followed by a charachter that is the command's identifier. The identifier is then followed by command's data if there is any.

- **The Keyboard Command Decoder (KCD) Task**
  The keyboard command decoder task is an unprivileged user-space task. The kernel should reserve `TID_KCD` for KCD's task ID. KCD should request a mailbox when it first starts to run. Once the mailbox is created, in an infinite loop, KCD calls `recv_msg` to receive messages from its mailbox. KCD only responds to two types of messages (and ignores the rest): (a) command registration (`KCD_REG`) and console keyboard input (`KEY_IN`). KCD processes received messages as follows.
  - Command Registration
    To register a command with KCD, any task can send a `KCD_REG` message to KCD. The message's data is only the command's identifier. Listing 4.1 shows code snippets of registering `%W` command.

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

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

Listing 4.1: Command Registration

KCD will forward any input command with a registered identifier to the mailbox of its corresponding registered task. The `L` command identifier is reserved for the KCD itself (see below). Tasks can register an already registered command identifier. KCD will always forward a command to the mailbox of the latest task that has registered the command's identifier.

- Keyboard Input
  The UART0_IRQ will forward any key input to the mailbox of the KCD using a `KEY_IN` message. The KCD will forward this message to the LCD Display task's mailbox to be echoed back to the console. KCD also should queue the input keys. Upon receiving "enter" key, KCD dequeues all previous keys to construct a single string. If the string starts with `%LT`, then KCD will print the task ID and state of all non-DORMANT tasks on the concole. If the string starts with `%LM`, then KCD will print the task ID and state of all non-DORMANT tasks with a mailboxe on the concole. You have the freedom to design the output format of these two commands.

  If the string starts with `%` followed by a registered command, then the KCD will forward this string to the mailbox of the corresponding registered task using a `KCD_CMD` message. The receiving task is responsible for handling the command. The message body contains the command string (excluding the `%` charachter and the "enter" key). If the command identifier is not registered or the registered task no longer exists, then KCD ignores the string and sends a message to LCD's mailbox to print "Command cannot be processed." If the string does not start with `%` or the length of string is longer than what is allowed to be sent as a message, then KCD will ignore the string and send a message to LCD to print "Invalid Command.".

- **The LCD Task**
  The LCD task is a privileged kernel task. The kernel reserves `TID_DISPLAY` for LCD's task ID. Similar to KCD, LCD also creates a mailbox when it first runs and then starts listening on its mailbox. LCD responds only to one message type (and ignores the rest): a `DISPLAY` message. The message body contains the character string to be displayed. The string may contain control characters (e.g. newline). LCD prints the string to the console terminal. As a privileged kernel task, LCD can disable the UART0 transmit interrupt when all elements of the string have been

displayed. LCD and the UART0_IRQ use an internal kernel buffer allocated by the kernel to communicate strings to be transmitted by the UART0 transmit register.

Interrupt Handler

The UAR0 uses interrupts for transmitting and receiving characters to and from the serial port. It sends a `KEY_IN` message to the KCD when a keyboard input is received. The task ID `TID_UART0_IRQ` is reserved to indicate the message is from the UART0 IRQ handler. To let UART0 to communicate with the LCD task, kernel creates an internal buffer for the LCD task to put the strings to be displayed in the buffer. The UART0 gets the data from the buffer and writes the data to is UART0 transmit register. Please note that UART0 IRQ handler is not a task. It is an interrupt handler that can send messages to KCD. The UART0 IRQ handler does not have a mailbox and cannot receive messages from other tasks.

Clarification: Interrupt handler should use the kernel stack of the interrupted task. After interrupt handler is done, RTX should resume the interrupted task by default unless the state of some other tasks has changed in which case a scheduling decision has to be made.

User Test Tasks

In order to test your implementation of the required functions, write an application that uses your kernel primitives. The provided `main_svc.c` and `usr_tasks.[ch]` files are for writing test codes. Create a set of testing scenarios to test your implemented functions and document the testing specification in the report (See 4.3.2). Implement the tests in `main_svc.c` and `usr_tasks.[ch]` files.

There is no hard requirement on what tests to be implemented. The rule of thumb is that the tests should be comprehensive enough to convince you that your implementation is correct.

Performance

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

- **Message passing time**. The time from the `send_msg()` being invoked by a MEDIUM priority task until the message is received by a receiver, a HIGH priority task that has called or will call `recv_msg()`.
- **Mailboxes utilization**. The overhead of saving meta data for messages should be minimum. We will evaluate your mailbox implementation by measuring its achieved utilization, which is defined as the ratio of the total size of the successfully delivered messages to the size of the mailbox.

4.3.2 Report

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

- Statement of the problem to be discussed in the report
- Descriptions of the data structures and algorithms used to implement all the functions listed in Section 4.3.1
- Testing scenario descriptions

To illustrate key algorithms, pseudo-code is acceptable. For testing, include five or more non-trivial testing scenarios.

4.3.3 Third-party Testing and Source Code File Organization

We will write a third-party testing program to verify the correctness of your implementations. In order to do so, we will need to enforce certain source code file organization standards. You should keep the file organization in the provided skeleton code, which has a src sub-directory inside. There are dos and don'ts you need to follow.

Don'ts

- Keep all existing files inside the src directory. Do not move any one of them to other directories.
- Do not change the file names under the src directory.
- Do not make any changes to the content of the `rtx.h` and `common.h` files.
- Do not change the existing function prototypes of any `k_*` functions appeared in `rtx.h` in `*.c` files.
- Do not include any new header files in the `main_svc.c`.
- Do not change the function name and prototype of the kcd, lcd and null tasks in `kcd_task.c`, `lcd_task.c` and `null_task.c`.

Dos

- You are allowed to add new self-defined functions to `k_*.[ch]` files.
- You are also allowed to create new `.h` and `.c` files [1].

[1] 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.

- The newly created `.h` files are allowed to be included in the `k_*.c` files.
- Any new file 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 system call functions defined in the `rtx.h`. During the third-party testing, the `main_svc.c` file will be replaced by a third-party testing code with the same file name. Use the provided template `main_svc.c` to write your own test codes.

4.4 Deliverables

4.4.1 Pre-Lab Deliverables

Fill the project_manager.csv template file and submit it to Lab3 Dropbox on Learn.

4.4.2 Post-Lab Deliverables

Create a directory named "lab3". Then create a sub-directory named "code" under "lab3". Put your uVision Project folder under "lab3/code". Put the `p3_report.pdf` under the "lab3" directory. Include a README file with group identification, project manager name and a description of directory contents. Put the README file under the "lab3" directory. Zip everything inside the lab3 directory and submit it to Learn Lab3 Dropbox.

4.5 Marking Rubric

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

Table 4.1: Lab3 Marking Rubric

The Rubric for marking the submitted source code and report is listed in Table 4.1. The functionality and performance of your implementation will be tested by a third-party test program. A minimum of **20 points** will be deducted if messages are lost or mysterious appear. A minimum of **35 points** will be deducted if the system

console I/O does not function at all. Your grade will be relative to the amount of error the third-party testing program has identified.

4.6 Errata

1. Page 30, Section 4.1 starter file urls have been corrected to "https://github.com/yqh/ece350/tree/master/manual_code/Context_Switching_UART0_IRQ" and "https://github.com/yqh/ece350/tree/master/manual_code/UART0_irq".
2. Page 32, the line before "RETURN VALUE": "Once a task exists" has been corrected to "Once a task exits".
3. Page 33: "task's state is `BLK_MBX`" has been corrected to "task's state is `BLK_MSG`".
4. Page 35: "The state of a blocked task is set to `BLK_MBX`" has been corrected to "The state of a blocked task is set to `BLK_MSG`".
5. Page 36, last line: "`%W` with task1." has been corrected to "`%W` command.".
6. Page 37, last line in Listing 4.1: "`buf)`" has been corrected to "`ptr);`".
7. Page 38, Interrupt Handler section: added "Clarification: Interrupt handler should use the kernel stack of the interrupted task. After interrupt handler is done, RTX should resume the interrupted task by default unless the state of some other tasks has changed in which case a scheduling decision has to be made.".
8. Page 39, under "Don'ts": added last item "Do not change the function name and prototype of the kcd, lcd and null tasks in `kcd_task.c`, `lcd_task.c` and `null_task.c`.".

Chapter 5

Lab4 Timing Service and Real-Time Scheduling

To be released by Nov 02, 2020.

Chapter 6

Lab5 Memory Protection and Stress Testing

To be released by Nov 16, 2020.

Part III

Frequent Asked Questions

Chapter 7

Frequently Asked Questions

We list frequently asked questions with answers in this chapter.

7.1 Lab1 Memory Management

7.2 Lab2 Task Management

Q1. *Reading through the source code provided, the only instance of NEW being used is when it's created, and the only difference is that it sets the MSP (the same way as if the task is not the same as the prior one). Is the only intent of the NEW state to initialize the MSP?*

A1. NEW is used when we first context switch to a newly-created task. The kernel stack of a newly-created task is different from the kernel stack of a task that has been preempted and this difference makes us treat the two differently in the context-switch function.

Q2. *What is MSP and PSP? What should they point to?*

A2. SP, MSP and PSP are processor registers. PSP should point to a user-space stack and MSP should point to a kernel stack. Setting MSP and PSP registers is the kernel's job. You probably want to keep track of both stacks in each task's TCB structure. When an interrupt or a system call happens, hardware automatically switches sp from pointing to PSP to pointing to MSP. That is, inside interrupt and system-call handlers, sp points to MSP which points to interrupted task's kernel stack. PSP also points to the interrupted task's user-space stack.

Q3. *What is the difference between MSP and K_SP? And what is the difference between PSP and U_SP?*

A3. MSP = K_SP. In the tutorial slides, we use the approach of letting USP = PSP and the kernel task does not use PSP. From the hardware point of view, you

could have all four possible combinations of stack and privilege level settings. The requirement of lab2 is that when you are in the kernel space (exception handler + privileged tasks), then you should use the kernel stack. The exception handler automaticaly uses MSP (set by the hardware, no choice). But for the privleged tasks, it can use either PSP or MSP when it is in the thread mode.

Q4. *tsk_yield seems to be a complete method as it is, and as far as I can tell from the lab manual, there isn't much for us to do beyond what is given already?*

A4. tsk_yield invokes dummy_scheduler and task_switch. You want your own scheduler instead of the dummy_scheduler. You also mostly likely need to change the task_switch (depending how you implement scheduler and the priority queues) since you have to deal with 4 priorities and it is not just two dummy tasks. In this sense, you need to change the tsk_yield.

Q5. *Can we change other functions in that file that are already given to us? For example, k_tsk_init to init some of the TCB memory?*

A5. You need to change the k_tsk_init, actually we put a comment there saying user stack allocation is not implemented. there are many other things are missing in this function if you want to support multiple priorities.

Q6. *How do we know which task is calling mem_alloc and dealloc?*

A6. When you are calling alloc/dealloc, you are the current running task. The kernel keeps track of the current running task's TCB. Go through the k_task.c file and see how the starter code implemented current task tracking.

Q7. *When would we call task_switch without calling the scheduler?*

A7. We separate them for modularity reason. The scheduler makes the policy and the task_switch is the mechanism.

Q8. *Where is PC actually changed in the starter code?*

A8. Please refer to slide 18 of the lab2 tutorial slides (in Learn). Line 11 is where we assign the PC the entry point of the task. We manually create the exception stack frame.

Q9. *What is a task? In the context of lab 2, is a task a process, a thread, or something else entirely?*

A9. Similar to a general-purpose thread, but specialized for our RTX.

Q10. *Should the user stack size include our allocated memory header?*

A10. It does not include the the header. It is what you asked for when calling mem_alloc.

Q11. *Who takes the owenership of the user stack?*

A11. Since it is the kernel's responsibility to tear down the user stack, so technically this is owned by the kernel.

Q12. *Do we need to update the RTX_TASK_INFO?*

A12. You are not required to update the RTX_TASK_INFO.

Q13. *When we allocate memory in k_mem.c, the address we return is at the bottom of the memory (since a heap grows from bottom to top). So when we allocate memory for the user stack, should u_stack_hi be the address returned by k_mem_alloc? Or do we need to get that address and add its size so that u_stack_hi actually points to the end of the memory allocated by k_mem_alloc?*

A13. The latter. A allocator should always return the starting address (low address) of the block of memory. If you want to use the returned memory block as a stack, then you should add the stack size to the returned address. Or you could do a stack allocator function to make the program more modular.

Q14. *Do we have to replace the dummy_scheduler with our own scheduler?*

A14. Yes, you have to implement a priority-driven scheduler as described in the lab manual. Note that the dummy_scheduler implements a simple scheduling algorithm for only two tasks.

Q15. *Can I set PRIO_NULL to the element in tcb array, which are not been initialized? I want to use PRIO_NULL to differentiate between the task that has been created and the vacant block.*

A15. This is not desirable, though from a blackbox testing point of view, we probably will have no way to capture this if you do it carefully. I would recommend to think another way to differentiate those tasks.

Q16. *I am a bit confused about kernel tasks vs user tasks.*

A16. When a unprivileged task executes, it uses its user stack in the user space and its kernel stack in the kernel space.

Q17. *Suppose there is one high priority thread and one low priority thread at the end of rtx initialization. The high priority thread will run first. Let's say that this high priority thread eventually yields the processor. Should our scheduler continue to let the high priority thread run because there are no other threads ready at the same (high) priority? Or do we let the lower priority thread run since we yielded the processor?*

A17. High-priority thread should continue running

Q18. *If the currently running thread calls tsk_set_prio on itself with a lower priority than it is running currently, should we be prempting the processor from the running thread in favor of a higher priority thread?*

A18. Yes, if there are ready tasks with higher priorities, they should run.

Q19. *Is the stack_hi address in rtx_task_info referring to the bottom of the stack or is it the bottom of the stack + one word? i.e. If we have a user stack with low address of A, and stack size of S (in words), is the high address A + S or A + S - 1?*

A19. A + S

Q20. *Do we need to check for stack overflow?*

A20. No.

Q21. *What are xPSR used in tsk_init in the starter code?*

A21. Special Purpose Program Status Register. The three program status registers APSR, EPSR and IPSR can be accessed as one combined register, referred to as xPSR.

Q22. *What is the difference between k_tsk_init and k_tsk_create?*

A22. They both do the exact same thing (create a new task) but k_tsk_init is called by the rtx initializer, whereas k_tsk_create is called by a user task. We on purpose did not separte the k_tsk_create from k_tsk_init so that we do not do too much in the starter code.

Q23. *What changes do we need to make to EXC_RETURN?*

A23. You change EXC_RETURN based on the task privilege level, which is a field in the TCB.

Q24. *Why does EXC_RETURN uses MVN instead of MOV?*

A24. MOV can only load values from 0-0xFFFF, so we use MVN trick here.

Q25. *What should the size of the user stack be for RTX_TASK_INFO?*

A25. User stack size is dynamically set, see the k_tsk_create function.

Q26. *When would we want to call C code from Assembly code?*

A26. You may want to perform certain operations depenidng on the value of a TCB field in assembly code. Say privilege level, or accessing the psp field in the TCB.

Q27. *Where is the heap and the scheduling data structure stored in memory?*

A27. There is no heap inside the OS Image. The heap is between the end of the image and the end of IRAM1. The ellipsis represents some other global variables inside the image. If the scheduling data structure can be decided at compile time, then usually we just statically allocate it.

Q28. *Where should we define/implement the NULL task?*

A28. The null task is a user-space task (see slide 9 of the lab2 tutorial).

Q29. *What will happend if all tasks have terminated?*

A29. The processor will run the null task indefinitely until a new task is created.

Q30. *So let's say I make 1st test case with 10 user tasks, so my task array is 11 tasks total including the null task. I then plan to run 2nd test case with 2 user tasks. In this example, how do I know when my 1st test case is over so that I can start the 2nd test case?*

A30. You cannot create the 2nd test case if all user tasks are terminated since null will forever run.

Q31. *Am I only allowed to call rtx_init once for the entire test harness?*

A31. The rtx_init is only called once. If you want to test whether null gets run properly, you need to make this test case the last one. Or you can have multiple main_svc.c for different testing combinations.

7.3 Lab3 Inter-task Communications and Console I/O

7.4 Lab4 Timing Service and Real-Time Scheduling

7.5 Lab5 Memory Protection and Stress Testing

Part IV

Software Development Environment Quick Reference Guide

Chapter 8

Keil MCB1700 Hardware Environment

8.1 MCB1700 Board Overview

The Keil MCB1700 board is populated with *NXP LPC1768* Microcontroller. Figure 8.1 shows the important interface and hardware components of the MCB1700 board.

Figure 8.2 is the hardware block diagram that helps you to understand the MCB1700 board components. Note that our lab will only use a small subset of the components which include the LPC1768 CPU, COM and Dual RS232.

The LPC1768 is a 32-bit ARM Cortex-M3 microcontroller for embedded applications requiring a high level of integration and low power dissipation. The LPC1768 operates at up to an 100 MHz CPU frequency. The peripheral complement of LPC1768 includes 512KB of on-chip flash memory, 64KB of on-chip SRAM and a variety of other on-chip peripherals. Among the on-chip peripherals, there are system control block, pin connect block, 4 UARTs and 4 general purpose timers, some of which will be used in your RTX course project. Figure 8.3 is the simplified LPC1768 block diagram [4], where the components to be used in your RTX project are circled with red. Note that this manual will only discuss the components that are relevant to the RTX course project. The LPC17xx User Manual is the complete reference for LPC1768 MCU.

8.2 Cortex-M3 Processor

The Cortex-M3 processor is the central processing unit (CPU) of the LPC1768 chip. The processor is a 32-bit microprocessor with a 32-bit data path, a 32-bit register bank, and 32-bit memory interfaces. Figure 8.4 is the simplified block diagram of the Cortex-M3 processor [5]. The processor has private peripherals which are system control block, system timer, NVIC (Nested Vectored Interrupt Controller) and MPU (Memory Protection Unit). The processor includes a number of internal debugging components which provides debugging features such as breakpoints and

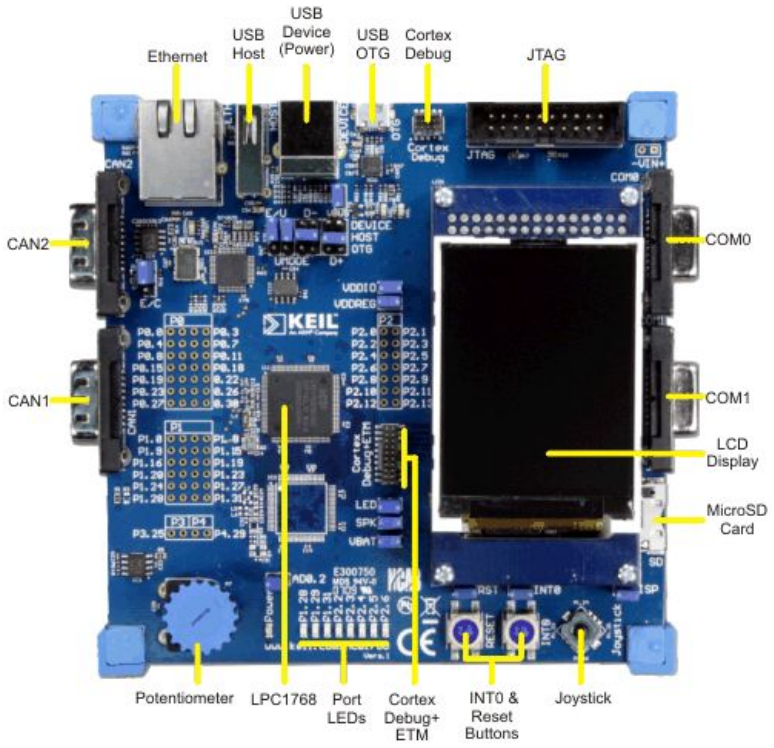

Figure 8.1: MCB1700 Board Components [1]

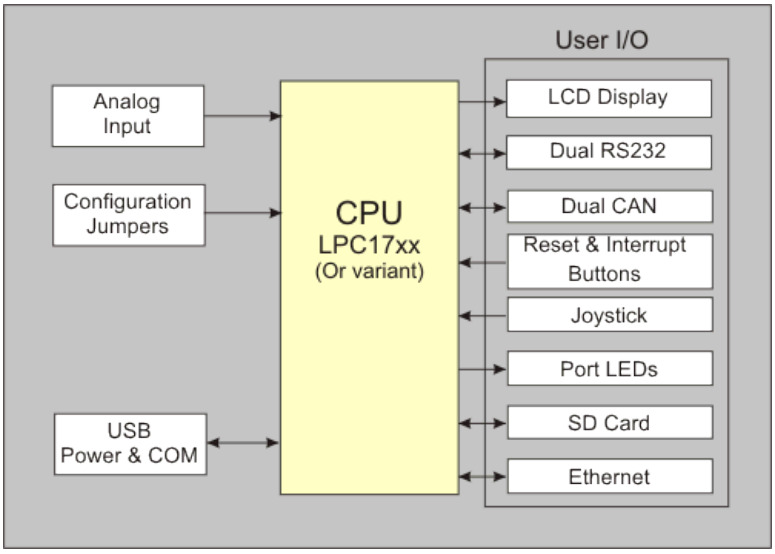

Figure 8.2: MCB1700 Board Block Diagram [1]

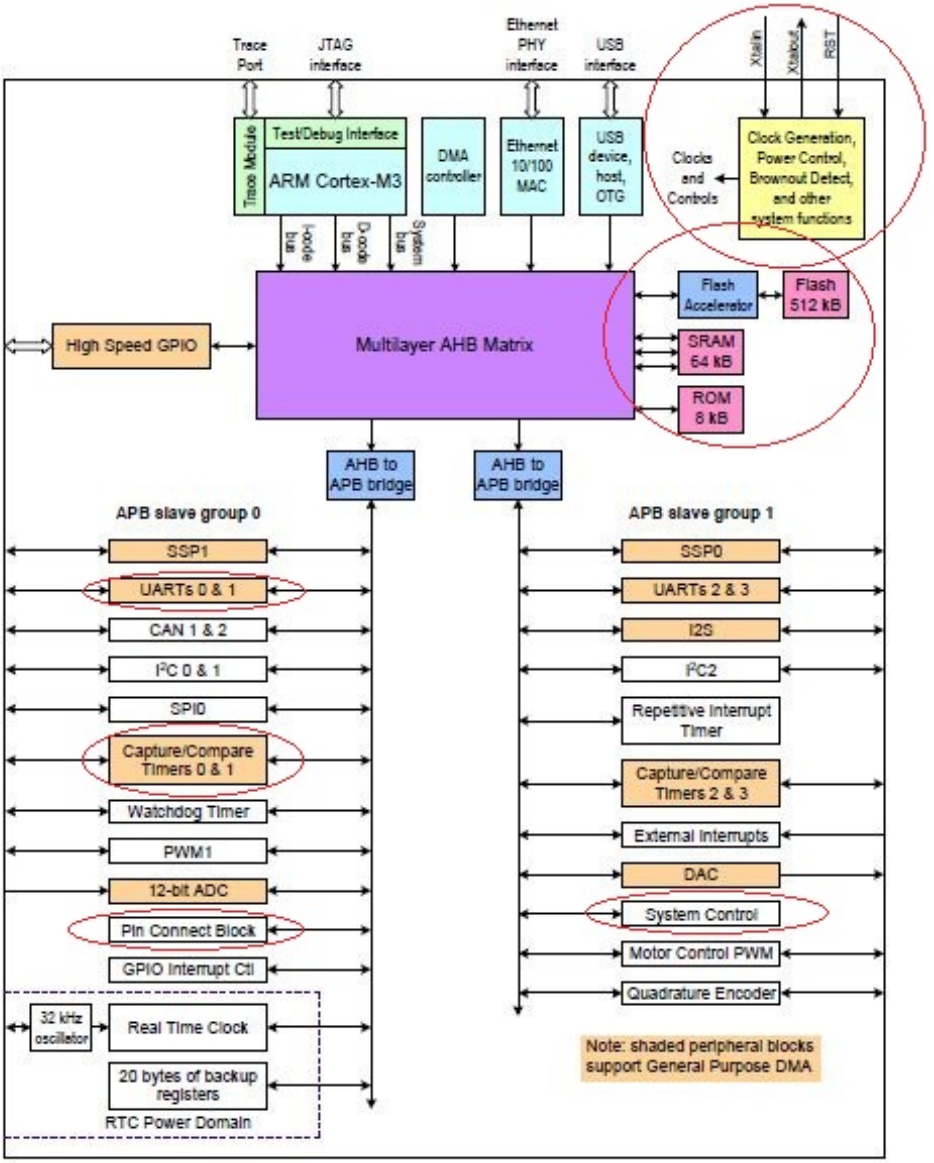

Figure 8.3: LPC1768 Block Diagram. The circled blocks are the ones that we will use in the lab project.

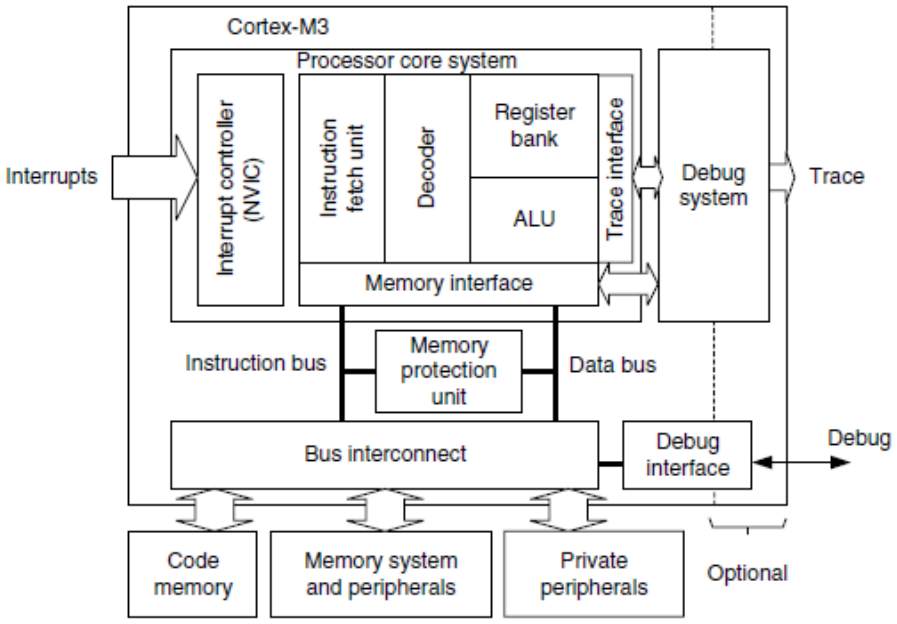

Figure 8.4: Simplified Cortex-M3 Block Diagram[5]

watchpoints.

8.2.1 Registers

The processor core registers are shown in Figure 8.5. For detailed description of each register, Chapter 34 in [4] is the complete reference.

- R0-R12 are 32-bit general purpose registers for data operations. Some 16-bit Thumb instructions can only access the low registers (R0-R7).
- R13(SP) is the stack pointer alias for two banked registers shown as follows:
  - *Main Stack Pointer (MSP)*: This is the default stack pointer and also reset value. It is used by the OS kernel and exception handlers.
  - *Process Stack Pointer (PSP)*: This is used by user application code.

  On reset, the processor loads the MSP with the value from address 0x00000000. The lowest 2 bits of the stack pointers are always 0, which means they are always word aligned.

  In Thread mode, when bit[1] of the CONTROL register is 0, MSP is used. When bit[1] of the CONTROL register is 1, PSP is used.
- R14(LR) is the link register. The return address of a subroutine is stored in the link register when the subroutine is called.
- R15(PC) is the program counter. It can be written to control the program flow.

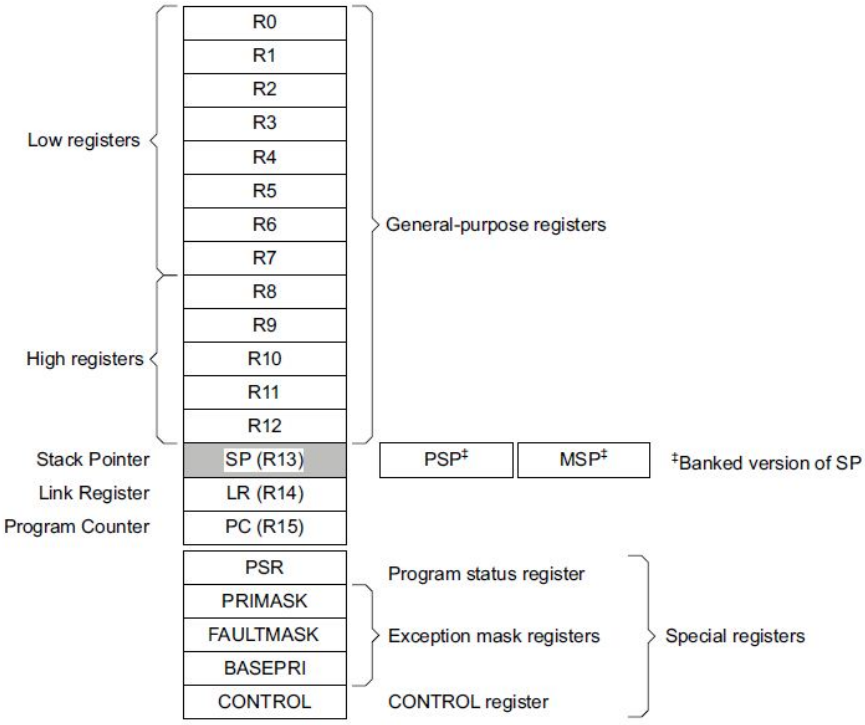

Figure 8.5: Cortex-M3 Registers[4]

- Special Registers are as follows:
  - Program Status registers (PSRs)
  - Interrupt Mask registers (PRIMASK, FAULTMASK, and BASEPRI)
  - Control register (CONTROL)

When at privilege level, all the registers are accessible. When at unprivileged (user) level, access to these registers are limited.

8.2.2 Processor mode and privilege levels

The Cortex-M3 processor supports two modes of operation, Thread mode and Handler mode.

- Thread mode is entered upon Reset and is used to execute application software.
- Handler mode is used to handle exceptions. The processor returns to Thread mode when it has finished exception handling.

Software execution has two access levels, Privileged level and Unprivileged (User) level.

- Privileged
  The software can use all instructions and has access to all resources. Your RTOS

kernel functions are running in this mode.

- Unprivileged (User)
  The software has limited access to `MSR` and `MRS` instructions and cannot use the `CPS` instruction. There is no access to the system timer, NVIC , or system control block. The software might also have restricted access to memory or peripherals. User processes such as the wall clock process should run at this level.

When the processor is in Handler mode, it is at the privileged level. When the processor is in Thread mode, it can run at privileged or unprivileged (user) level. The bit[0] in CONTROL register determines the execution privilege level in Thread mode. When this bit is 0 (default), it is privileged level when in Thread mode. When this bit is 1, it is unprivileged when in Thread mode.Figure 8.6 illustrate the mode and privilege level of the processor.

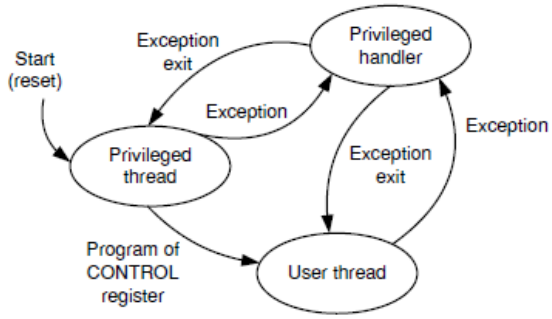

Figure 8.6: Cortex-M3 Operating Mode and Privilege Level[5]

Note that only privileged software can write to the CONTROL register to change the privilege level for software execution in Thread mode. Unprivileged software can use the `SVC` instruction to make a supervisor call to transfer control to privileged software. When we are in the privileged thread mode, we can directly set the control register to change to unprivileged thread mode. We also can change to unprivileged thread mode by calling `SVC` to raise an exception first and then inside the exception handler we set the privilege level to unprivileged by setting the control register. Then we modify the `EXC_RETURN` value in the LR (R14) to indicate the mode and stack when returning from an exception. This mechanism is often used by the kernel in its intialization phase and also context switching between privileged processes and unprivileged processes.

8.2.3 Stacks

The processor uses a full descending stack. This means the stack pointer indicates the last stacked item on the stack memory. When the processor pushes a new item

onto the stack, it decrements the stack pointer and then writes the item to the new memory location.

The processor implements two stacks, the *main stack* and the *process stack*. One of these two stacks is banked out depending on the stack in use. This means only one stack is visible at a time as R13. In Handler mode, the main stack is always used. The bit[1] in CONTROL register reads as zero and ignores writes in Handler mode. In Thread mode, the bit[1] setting in CONTROL register determines whether the main stack or the process stack is currently used. Table 8.1 summarizes the processor mode, execution privilege level, and stack use options.

Processor mode	Used to execute	Privilege level for software execution	CONTROL Bit[0]	CONTROL Bit[1]	Stack used
Thread	Applications	Privileged	0	0	Main Stack
		Privileged	0	1	Process Stack
		Unprivileged	1	1	Process Stack
Handler	Exception handlers	Privileged	-	0	Main Stack

Table 8.1: Summary of processor mode, execution privilege level, and stack use options

8.3 Memory Map

The Cortex-M3 processor has a single fixed 4GB address space. Table 8.2 shows how this space is used on the LPC1768.

Address Range	General Use	Address range details	Description
0x0000 0000 to 0x1FFF FFFF	On-chip non-volatile memory	0x0000 0000 – 0x0007 FFFF	512 KB flash memory
	On-chip SRAM	0x1000 0000 – 0x1000 7FFF	32 KB local SRAM
	Boot ROM	0x1FFF 0000 – 0x1FFF 1FFF	8 KB Boot ROM
0x2000 0000 to 0x3FFF FFFF	On-chip SRAM (typically used for peripheral data)	0x2007 C000 – 0x2007 FFFF	AHB SRAM - bank0 (16 KB)
		0x2008 0000 – 0x2008 3FFF	AHB SRAM - bank1 (16 KB)
	GPIO	0x2009 C000 – 0x2009 FFFF	GPIO
0x4000 0000 to 0x5FFF FFFF	APB Peripherals	0x4000 0000 – 0x4007 FFFF	APB0 Peripherals
		0x4008 0000 – 0x400F FFFF	APB1 Peripherals
	AHB peripherals	0x5000 0000 – 0x501F FFFF	DMA Controller, Ethernet interface, and USB interface
0xE000 0000 to 0xE00F FFFF	Cortex-M3 Private Peripheral Bus (PPB)	0xE000 0000 – 0xE00F FFFF	Cortex-M3 private registers(NVIC, MPU and SysTick Timer et. al.)

Table 8.2: LPC1768 Memory Map

Note that the memory map is not continuous. For memory regions not shown in the table, they are reserved. When accessing reserved memory region, the processor's behavior is not defined. All the peripherals are memory-mapped and the LPC17xx.h file defines the data structure to access the memory-mapped peripherals in C.

8.4 Exceptions and Interrupts

The Cortex-M3 processor supports system exceptions and interrupts. The processor and the Nested Vectored Interrupt Controller (NVIC) prioritize and handle all exceptions. The processor uses *Handler mode* to handle all exceptions except for reset.

8.4.1 Vector Table

Exceptions are numbered 1-15 for system exceptions and 16 and above for external interrupt inputs. LPC1768 NVIC supports 35 vectored interrupts. Table 8.3 shows system exceptions and some frequently used interrupt sources. See Table 50 and Table 639 in [4] for the complete exceptions and interrupts sources. On system reset, the vector table is fixed at address `0x00000000`. Privileged software can write to the VTOR (within the System Control Block) to relocate the vector table start address to a different memory location, in the range `0x00000080` to `0x3FFFFF80`.

8.4.2 Exception Entry

Exception entry occurs when there is a pending exception with sufficient priority and either

- the processor is in Thread mode
- the processor is in Handler mode and the new exception is of higher priority than the exception being handled, in which case the new exception preempts the original exception (This is the nested exception case which is not required in our RTOS lab).

When an exception takes place, the following happens

- Stacking

  When the processor invokes an exception (except for tail-chained or a late-arriving exception, which are not required in the RTOS lab), it automatically stores the following eight registers to the SP:
  - R0-R3, R12
  - PC (Program Counter)

Exception number	IRQ number	Vector address or offset	Exception type	Priority	C PreFix
1	-	0x00000004	Reset	-3, the highest	
2	-14	0x00000008	NMI	-2,	NMI_
3	-13	0x0000000C	Hard fault	-1	HardFault_
4	-12	0x00000010	Memory management fault	Configurable	MemManage_
⋮					
11	-5	0x0000002C	SVCall	Configurable	SVC_
⋮					
14	-2	0x00000038	PendSV	Configurable	PendSVC_
15	-1	0x0000003C	SysTick	Configurable	SysTick_
16	0	0x00000040	WDT	Configurable	WDT_IRQ
17	1	0x00000044	Timer0	Configurable	TIMER0_IRQ
18	2	0x00000048	Timer1	Configurable	TIMER1_IRQ
19	3	0x0000004C	Timer2	Configurable	TIMER2_IRQ
20	4	0x00000050	Timer3	Configurable	TIMER3_IRQ
21	5	0x00000054	UART0	Configurable	UART0_IRQ
22	6	0x00000058	UART1	Configurable	UART1_IRQ
23	7	0x0000005C	UART2	Configurable	UART2_IRQ
24	8	0x00000060	UART3	Configurable	UART3_IRQ
⋮					

Table 8.3: LPC1768 Exception and Interrupt Table

  - PSR (Processor Status Register)
  - LR (Link Register, R14)

  Figure 8.7 shows the exception stack frame. Note that by default the stack frame is aligned to double word address starting from Cortex-M3 revision 2. The alignment feature can be turned off by programming the STKALIGN bit in the System Control Block (SCB) Configuration Control Register (CCR) to 0. On exception entry, the processor uses bit[9] of the stacked PSR to indicate the stack alignment. On return from the exception, it uses this stacked bit to restore the correct stack alignment.

- Vector Fetching
  While the data bus is busy stacking the registers, the instruction bus fetches the exception vector (the starting address of the exception handler) from the vector table. The stacking and vector fetch are performed on separate bus interfaces, hence they can be carried out at the same time.

- Register Updates
  After the stacking and vector fetch are completed, the exception vector will start to execute. On entry of the exception handler, the following registers will be updated as follows:
  - SP: The SP (MSP or PSP) will be updated to the new location during stack-

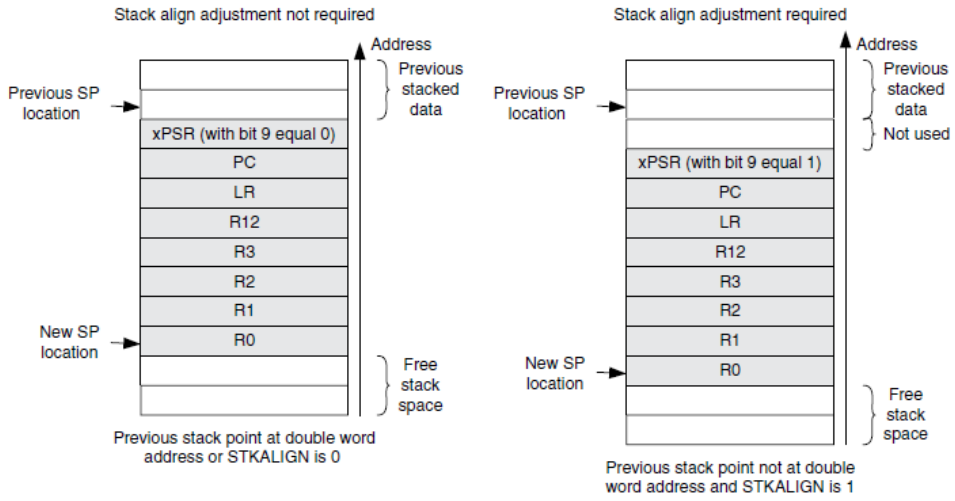

Figure 8.7: Cortex-M3 Exception Stack Frame [5]

ing. Stacking from the privileged/unprivileged thread to the first level of the exception handler uses the MSP/PSP. During the execution of exception handler routine, the MSP will be used when stack is accessed.

- PSR: The IPSR will be updated to the new exception number
- PC: The PC will change to the vector handler when the vector fetch completes and starts fetching instructions from the exception vector.
- LR: The LR will be updated to a special value called `EXC_RETURN`. This indicates which stack pointer corresponds to the stack frame and what operation mode the processor was in before the exception entry occurred.
- Other NVIC registers: a number of other NVIC registers will be updated .For example the pending status of exception will be cleared and the active bit of the exception will be set.

8.4.3 `EXC_RETURN` Value

`EXC_RETURN` is the value loaded into the LR on exception entry. The exception mechanism relies on this value to detect when the processor has completed an exception handler. The `EXC_RETURN` bits $[31 : 4]$ is always set to `0xFFFFFFF` by the processor. When this value is loaded into the PC, it indicates to the processor that the exception is complete and the processor initiates the exception return sequence. Table 8.4 describes the `EXC_RETURN` bit fields. Table 8.5 lists Cortex-M3 allowed `EXC_RETURN` values.

Bits	31:4	3	2	1	0
Description	0xFFFFFFF	Return mode (Thread/Handler)	Return stack	Reserved; must be 0	Process state (Thumb/ARM)

Table 8.4: EXC_RETURN bit fields [5]

Value	Description		
	Return Mode	Exception return gets state from	SP after return
0xFFFFFFF1	Handler	MSP	MSP
0xFFFFFFF9	Thread	MSP	MSP
0xFFFFFFFD	Thread	PSP	PSP

Table 8.5: EXC_RETURN Values on Cortex-M3

8.4.4 Exception Return

Exception return occurs when the processor is in Handler mode and executes one of the following instructions to load the EXC_RETURN value into the PC:

- a POP instruction that includes the PC. This is normally used when the EXC_RETURN in LR upon entering the exception is pushed onto the stack.
- a BX instruction with any register. This is normally used when LR contains the proper EXC_RETURN value before the exception return, then BX LR instruction will cause an exception return.
- a LDR or LDM instruction with the PC as the destination. This is another way to load PC with the EXC_RETURN value.

Note unlike the ColdFire processor which has the RTE as the special instruction for exception return, in Cortex-M3, a normal return instruction is used so that the whole interrupt handler can be implemented as a C subroutine.

When the exception return instruction is executed, the following exception return sequences happen:

- Unstacking: The registers (i.e. exception stack frame) pushed to the stack will be restored. The order of the POP will be the same as in stacking. The SP will also be changed back.
- NVIC register update: The active bit of the exception will be cleared. The pending bit will be set again if the external interrupt is still asserted, causing the processor to reenter the interrupt handler.

8.5 Data Types

The processor supports 32-bit words, 16-bit halfwords and 8-bit bytes. It supports 64-bit data transfer instructions. All data memory accesses are managed as little-endian.

Chapter 9

Keil Software Development Tools

The Keil MDK-ARM development tools are used for MCB1700 boards in our lab. The tools include

- uVision5 IDE which combines the project manager, source code editor and program debugger into one environment;
- ARM compiler, assembler, linker and utilities;
- ULINK USB-JTAG Adapter which allows you to debug the embedded programs running on the board.

The MDK-Lite is the evaluation version and does not require a license. It has a code size limit of 32KB, which is adequate for the lab projects. The MDK-Lite version 5 is installed on all lab computers. If you want to install the software on your own computer. MDK 5.30 installation file is in Learn Lab/RTX Project section. The downloading link for the latest version is https://www2.keil.com/mdk5/editions/lite.

9.1 Creating an Application in uVision5 IDE

To get started with the Keil IDE, the Getting Started with MDK Guide at https://www.keil.com/support/man/docs/mdk_gs/ is a good place to start. We will walk you through the IDE by developing a simple HelloWorld application which displays Hello World through the UART0 and UART1 that are connected to the lab PC. Note the HelloWorld example uses polling on both UART0 and UART1 rather than interrupt.

9.1.1 Getting Starter Code from the GitHub

The ECE 350 lab starter github is at https://github.com/yqh/ece350. Let's first make a clone of this repository by using the following command:

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

9.1.2 Create a New Project

1. Create a directory named "HelloWorld" on your computer. The folder path name should not contain spaces on Nexus computers.
2. Create a sub-directory "src" under the "HelloWorld" directory. This sub-folder is where we want to put our source code of the project.
3. Copy the following files to "src" folder:
   - `manual_code/util/printf_uart/uart_def.h`
   - `manual_code/util/printf_uart/uart_polling.h`
   - `manual_code/util/printf_uart/uart_polling.c`
4. Create a new uVision project.
   Open the file explorer and navigate to `C:\Software\Keil_v5\UV4`. Double click the `UV4.exe` program to start the IDE.
   - Click Project → New uVision Project (See Figure 9.1).

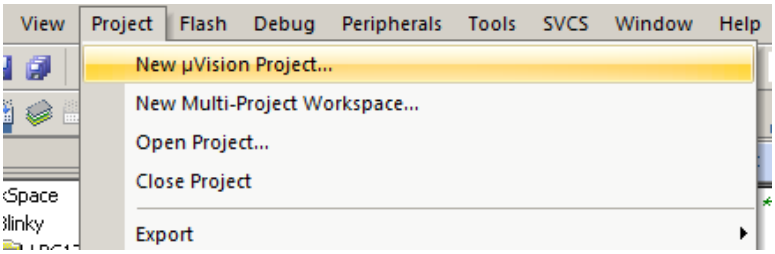

Figure 9.1: Keil IDE: Create a New Project

   - Select NXP → LPC1700 Series → LPC176x → LPC1768 (See Figure 9.2).
   - Select CMSIS → CORE and Device → Startup (See Figure 9.3).

9.1.3 Managing Project Components

You just finished creating a new project. One the left side of the IDE is the Project window. Expand all objects. You will see the default project setup as shown in Figure 9.4.

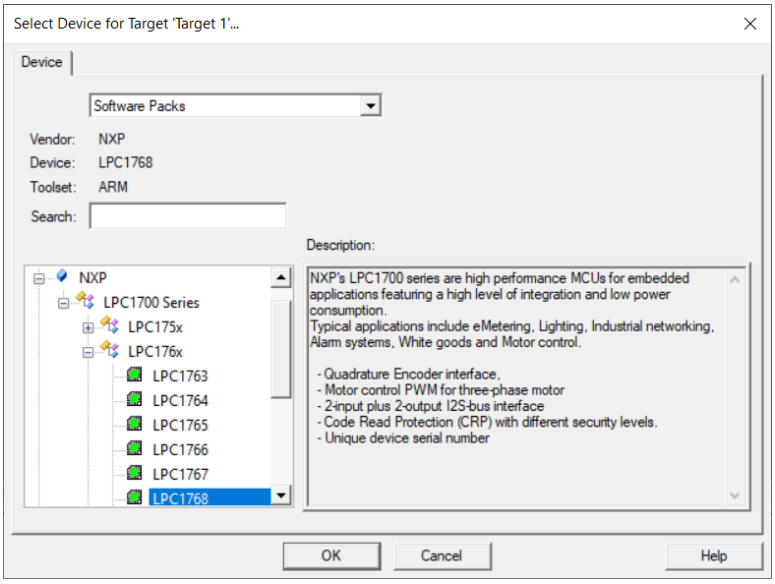

Figure 9.2: Keil IDE: Choose MCU

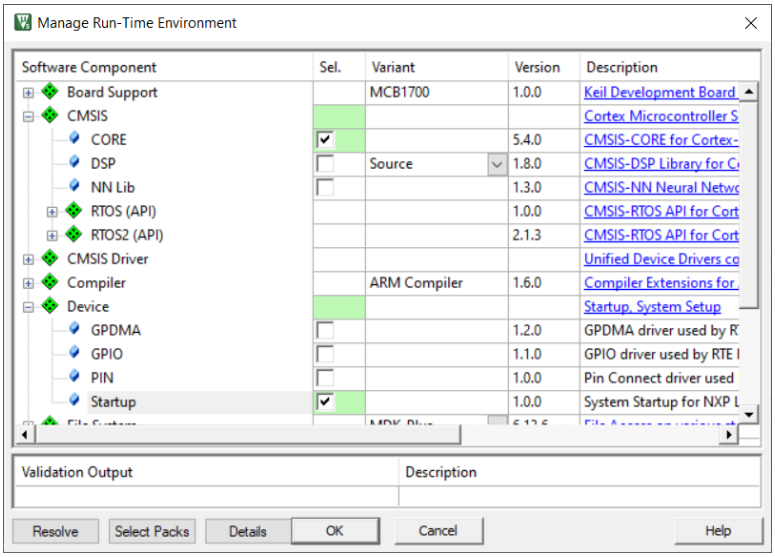

Figure 9.3: Keil IDE: Manage Run-Time Environment

1. Rename the Target
   The "Target 1"is the default name of the project build target and you can rename it. Select the target name to highlight it and then long press the left button of the mouse to make the target name editable. Input a new target name, say "HelloWorld SIM".

2. Rename the Source Group
   The IDE allows you to group source files to different groups to better manage the source code. By default "Source Group 1" is created and it contains no file.

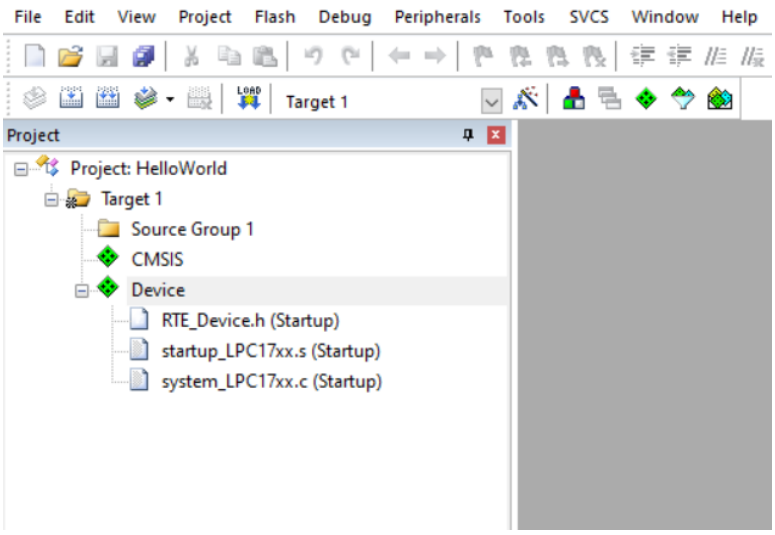

Figure 9.4: Keil IDE: A default new project

Let's rename the source group to "System Code" [1].

3. Add a New Source Group
   We can also add new source group in our project. Select the HelloWorld SIM item and right click to bring up the context window and select "Add Group..." (See Figure 9.5).

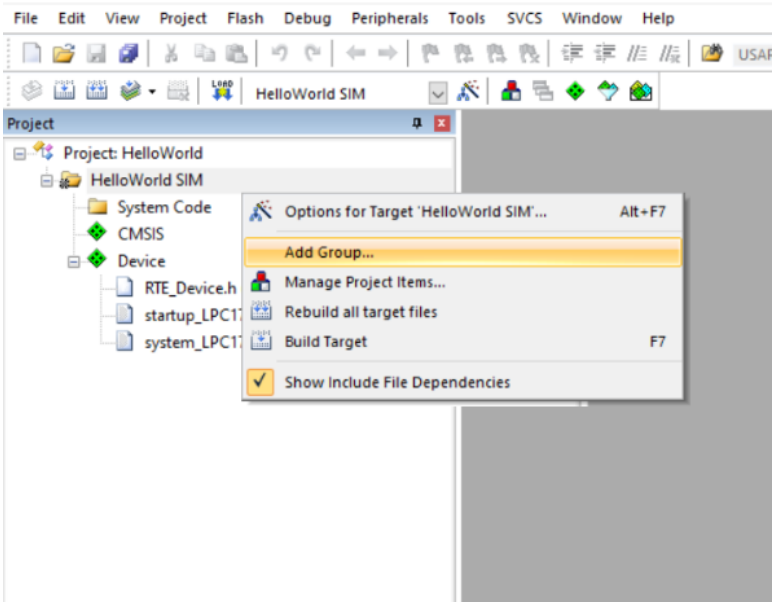

Figure 9.5: Keil IDE: Add Group

[1] To rename a source group, select the source group to highlight it and long press the left mouse button to make the name editable.

A new source group named “New Group” is added to the project. Let’s rename it to “User Code“. Your project will now look like Figure 9.6.

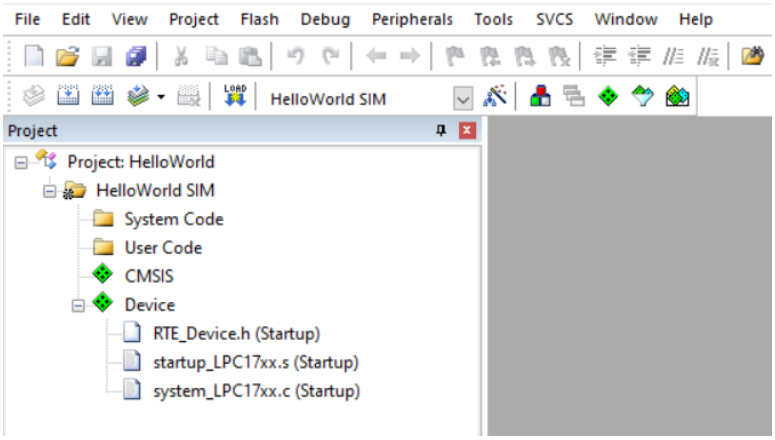

Figure 9.6: Keil IDE: Updated Project Profile

4. Add Source Code to a Source Group
   Let’s add `uart_polling.c` to “System Code” group by double clicking the source group and choose the file from the file window. Double clicking the file name will add the file to the source group. Or you can select the file and click the “Add” button at the lower right corner of the window (See Figure 9.7).

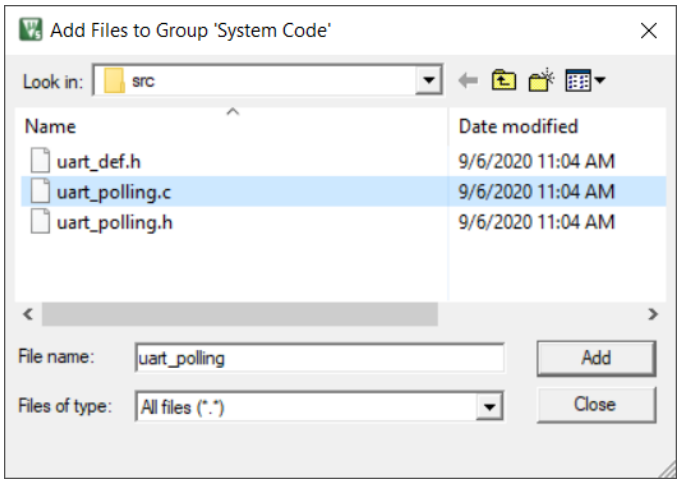

Figure 9.7: Keil IDE: Add Source File to Source Group

   Your project will now look like Figure 9.8.

5. Create a new source file
   The project does not have a main function yet. We now create a new file by selecting File → New (See Figure 9.9). Before typing anything to the file, save the file and name it “main.c”. Add main.c to the “Source Code” group. Type

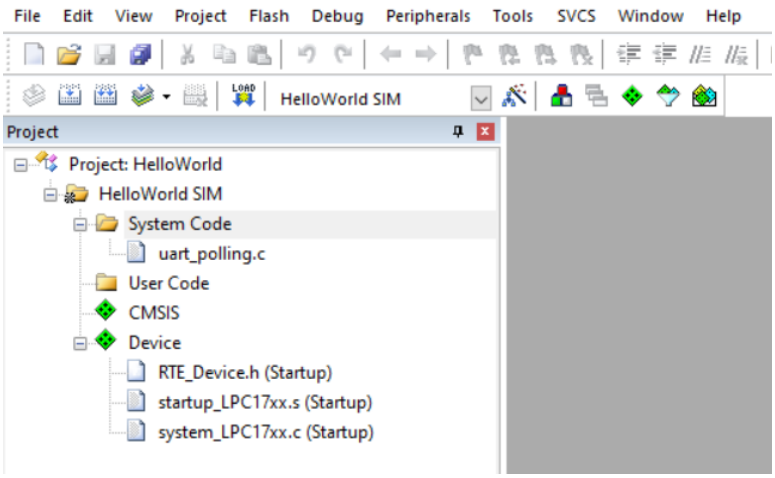

Figure 9.8: Keil IDE: Updated Project Profile

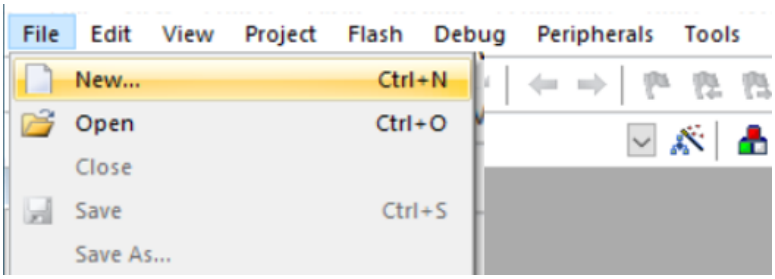

Figure 9.9: Keil IDE: Create New File

the source code as shown in Figure 9.10. Your final project would look like the screen shot in Figure 9.10.

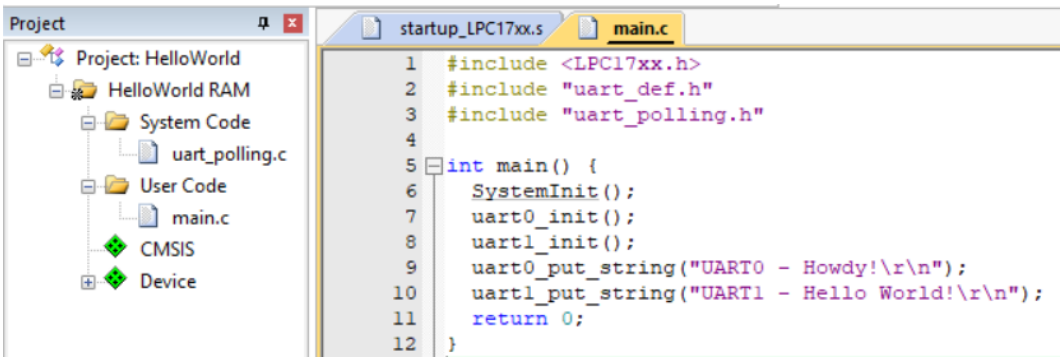

Figure 9.10: Keil IDE: Final Project Setting

9.1.4 Build the Application

To build a target, the main work is to configure the target options.

Configure Target Options

Most of the default settings of the target options are good. There are a few options that we need to modify.

1. Bring up the target option configuration window by pressing the target options button (See Figure 9.11).

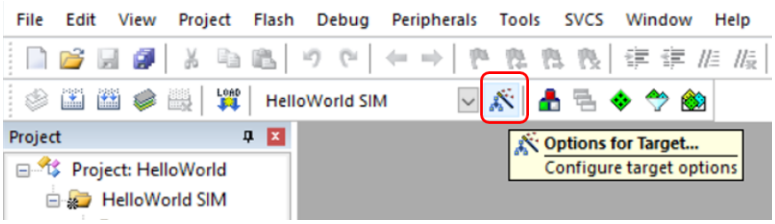

Figure 9.11: Keil IDE: Target Options Configuration

2. Configure the C/C++ tab as shown in Figure 9.12. We do not need c99 for this example. So we leave it unchecked. If you click OK to finish the Target option setting now, you will find the red cross at line 2 (See Figure 9.10) is now gone. Now re-open the target options configuration window and proceed to the next step.

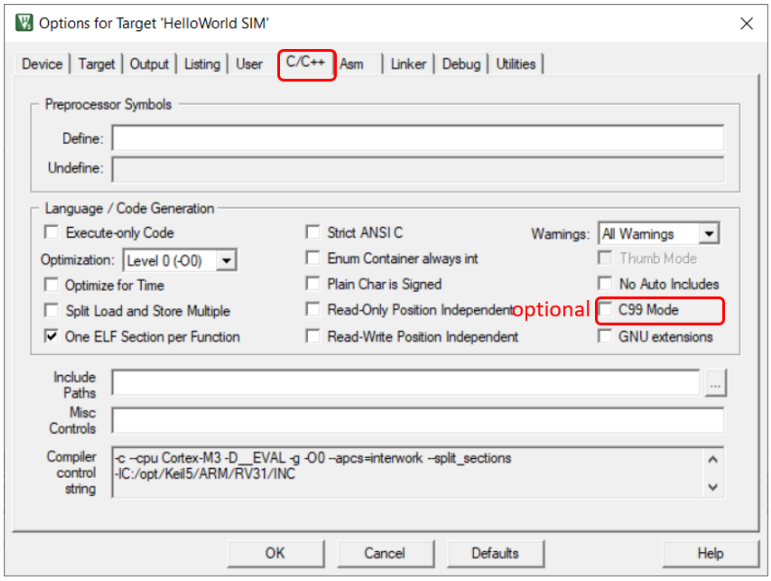

Figure 9.12: Keil IDE: Target Options C/C++ Tab Configuration

3. Configure the Target tab as shown in Figure 9.13. We want to use the default version 5 arm compiler. We also want remove the IRAM2 from the default setting.

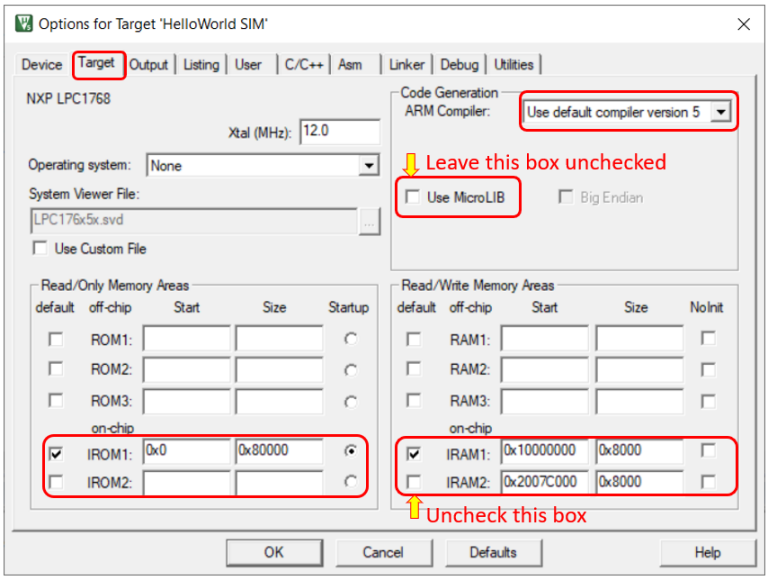

Figure 9.13: Keil IDE: Target Options Target Tab Configuration

4. Configure the Linker tab as shown in Figure 9.14. This is to instruct the linker to use the memory layout from the Target tab rather than default memory layout.

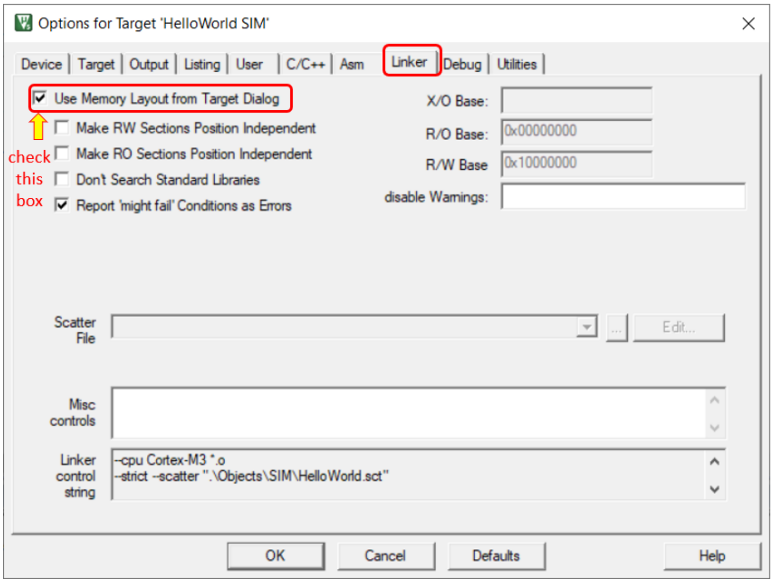

Figure 9.14: Keil IDE: Target Options Linker Tab Configuration

Build the Target

To build the target, click the "Build" button (see Figure 9.15). If nothing goes wrong,

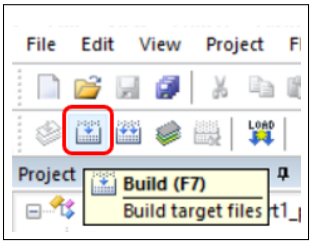

Figure 9.15: Keil IDE: Build Target

the build output window at the bottom of the IDE will show a log similar like the one shown in Figure 9.16.

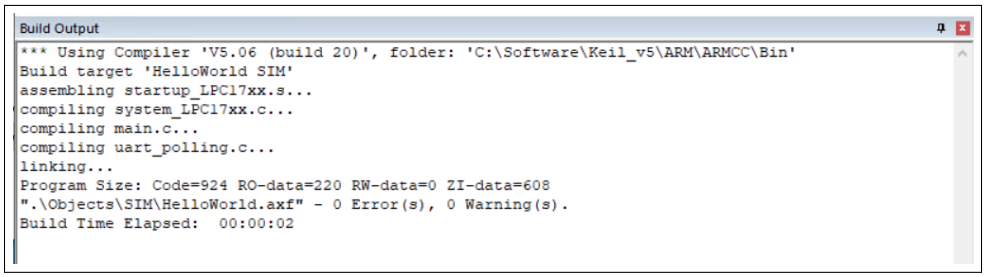

Figure 9.16: Keil IDE: Build Target

9.2 Debug the Target

In theory, you may now load the target by pressing the LOAD button. However please *pause* before you attempt to do it. Our final goal is to build a project that is ready to be released and then load it to the on-chip flash to ship it to the customer. However we will need to do lots of debugging before we reach this goal. Keep flashing the board will greatly shorten the life of the on-chip memory since there is a limited number of times one can flash it. So for development purpose, developers rarely press the LOAD button in the IDE to load the image to the flash memory since each load action writes to the flash memory cells. Most of the time we use the simulator to debug and execute our project. We will also show you a commonly used technique to load the target to RAM, which has a lot longer life span than flash memory, and debug the target on the board by using the ULINK-ME hardware debugger in Section 9.2.2.

9.2.1 Debug the Project in Simulator

We will configure our project to use the simulator as the debugger.

1. Open up the target option window and select "Use Simulator" in the Debug tab and set the Dialog DLL and Parameters as shown Figure 9.17.

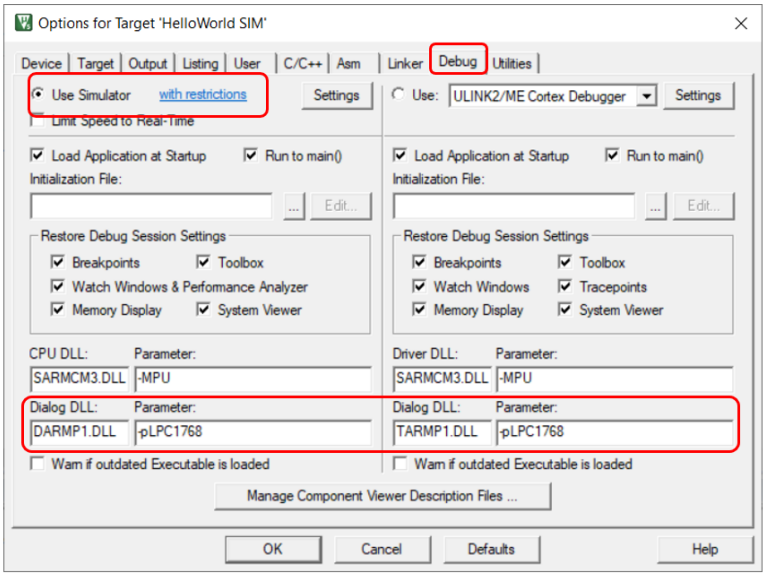

Figure 9.17: Keil IDE: Target Options Debug Tab Configuration

2. Press the "debug" button to bring up the debugger interface (See Figure 9.18).

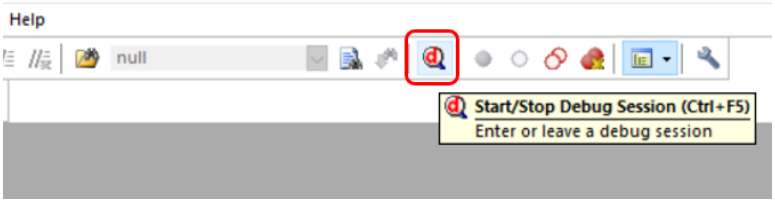

Figure 9.18: Keil IDE: Debug Button

3. Select UART1 and UART2 (see Figure 9.19) from the serial window drop down list so that they appear in simulator (see Figure 9.20). Note that the hardware UART index starts from 0 and the simulator UART index starts from 1. So the UART1 window in simulator is for the UART0 on the board. The UART2 window in simulator is for the UART1 on the board.

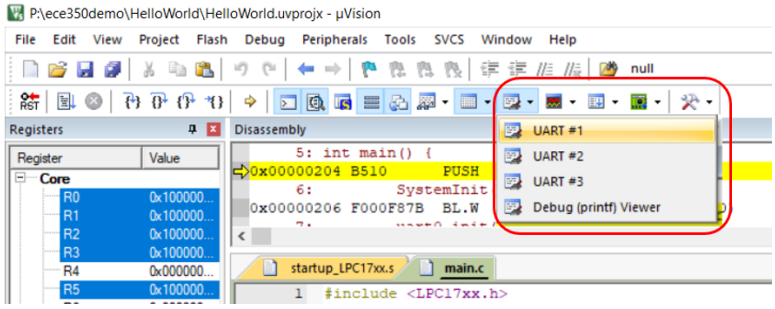

Figure 9.19: Keil IDE: Debugging. Enable Serial Window View.

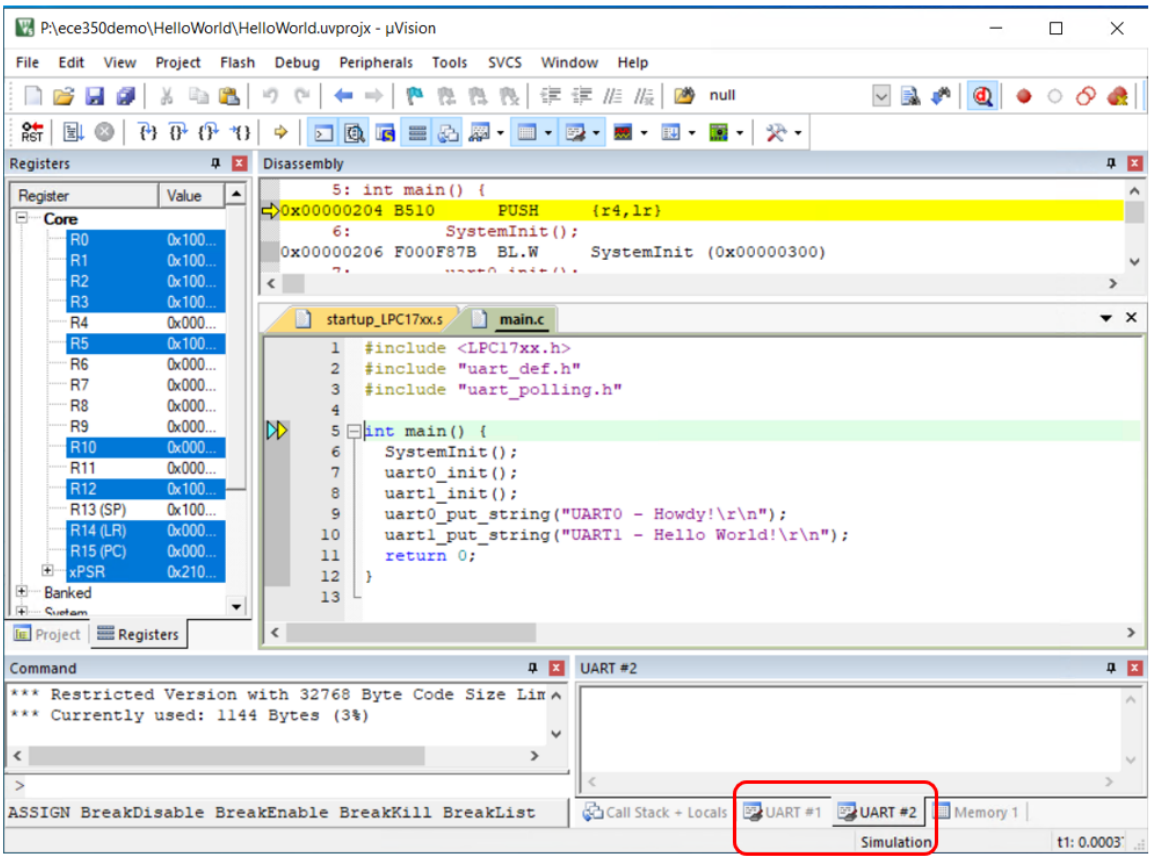

Figure 9.20: Keil IDE: Debugging. Both UART0 and UART1 views are enabled in simulator.

4. Press the "Run" button on the menu to let the program execute (see Figure 9.21). You will see the output of UART0 appearing in UART1 simulator window and the output of UART1 appearing in UART2 simulator window (see Figure 9.22). Note that we moved the UART windows from their default positions in the simulator for better view.

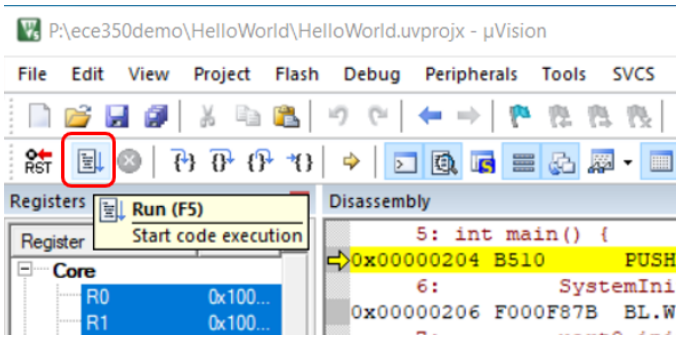

Figure 9.21: Keil IDE: Debugging. The Run Button.

5. To exit the debugging session, press the "debug" button again (see Figure 9.18).

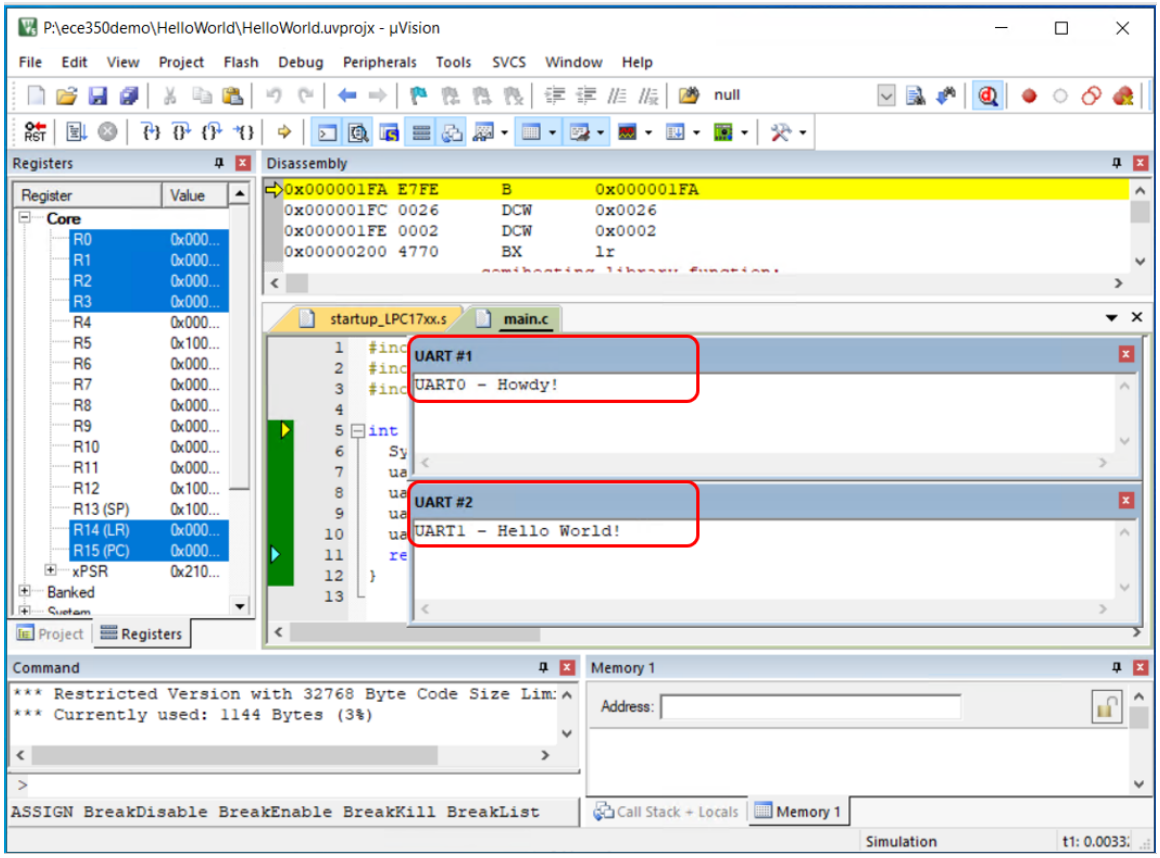

Figure 9.22: Keil IDE: Debugging Output.

9.2.2 Debug the Project on the Board

When debugging the code on the board, we use the ULINK-ME Cortex Debugger. The code will execute on the board. You will find creating a separate hardware debug target makes the development process easier.

1. Press the Managing Project Item button (see Figure 9.23).

2. Press the New icon to create a new target and name it "HelloWorld RAM"(see Figure 9.24). The new target duplicates the HelloWorld SIM target configuration.

3. Switch your target to the newly created RAM target (See Figure 9.25). Configure in-memory code execution as shown in Figure 9.26.

   The default image memory map setting is that the code is executed from the ROM (see Figure 9.13). Since the ROM portion of the code needs to be flashed in order to be executed on the board, this incurs wear-and-tear on the on-chip flash of the LPC1768. Since most attempts to write a functioning RTX will eventually require some more or less elaborate debugging, the flash memory might wear out quickly. Unlike the flash memory stick file systems where the

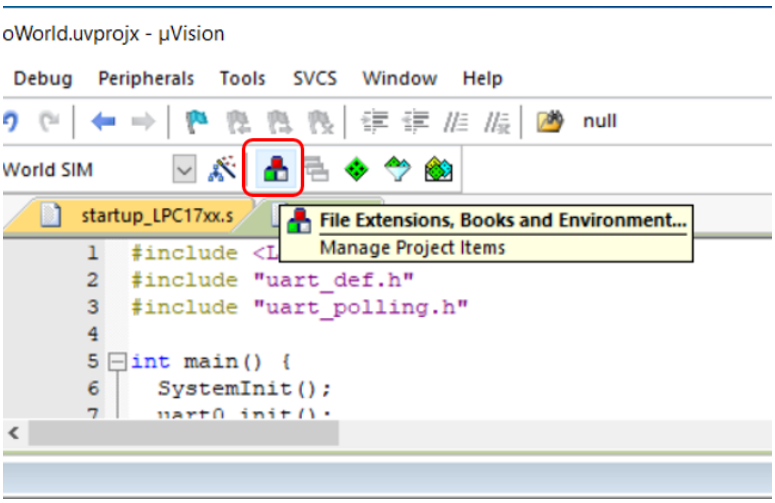

Figure 9.23: Keil IDE: Manage Project Items Button

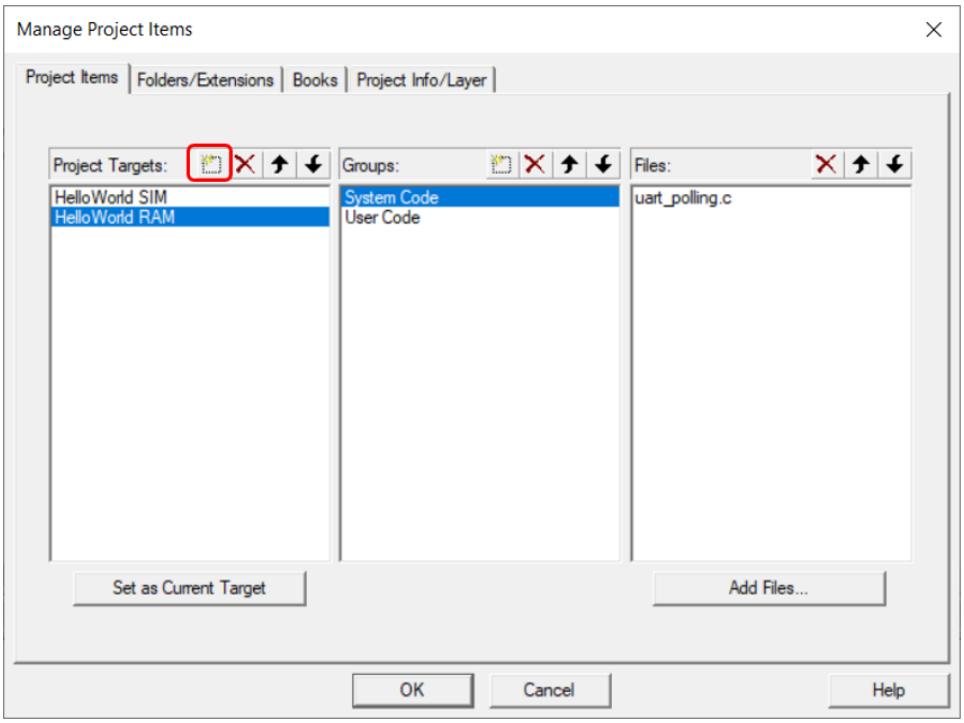

Figure 9.24: Keil IDE: Manage Project Items Window.

wear is aimed to be uniformly distributed across the memory portion, this flash memory will get used over and over again in the same portion.

The ARM compiler can be configured to have a different starting address. The configuration in Figure 9.26 makes code starting address in RAM.

4. Select the ULINK2/ME Cortex Debugger in the target options Debug tab and use an debug script `RAM.ini` provided in the starter code (See Figure 9.27) as

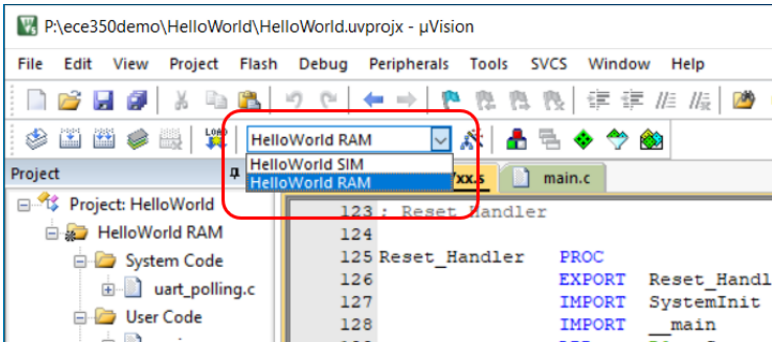

Figure 9.25: Keil IDE: Select HelloWorld RAM Target.

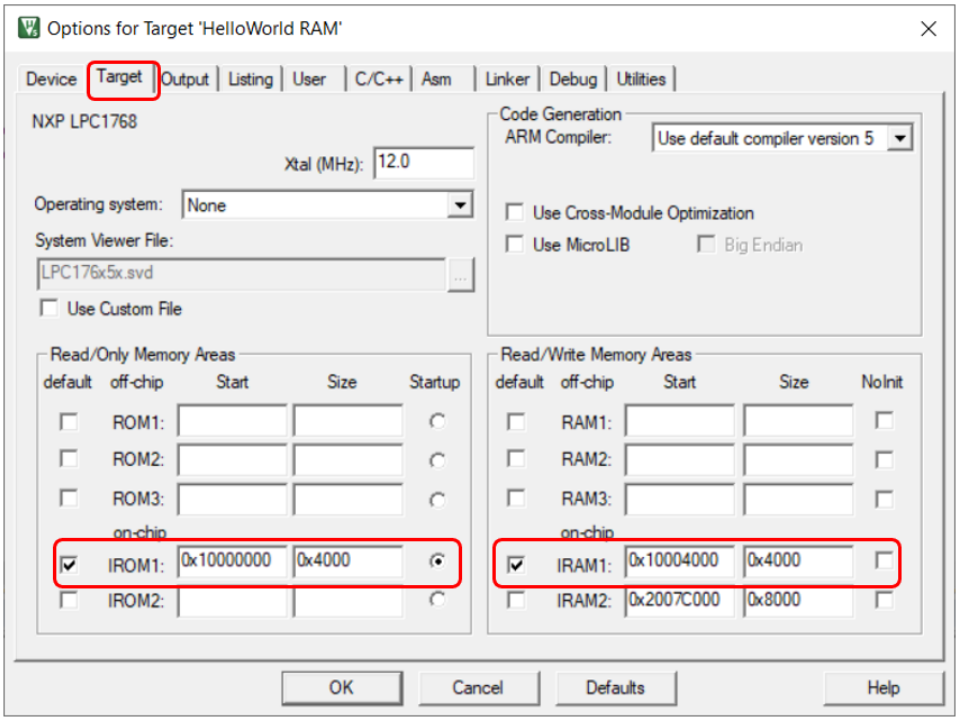

Figure 9.26: Keil IDE: Configure Target Options Target Tab for In-memory Execution.

a initialization file. An initialization file `RAM.ini` (see Listing B.1 in Appendix B) is needed to do the proper setting of SP, PC and vector table offset register.

5. Press the settings button beside the ULINK2/ME Cortex Debugger (see Figure 9.27) and select the Flash Download tab (see Figure 9.28). You should add the LPC17xx IAP 512kB Flash algorithm to the Programming Algorithm field if it is not already there.

6. Open the PuTTY terminals to see the output.

   You will need a terminal emulator such as PuTTY that talks directly to COM ports in order to see output of the serial port. To find out the two COM ports,

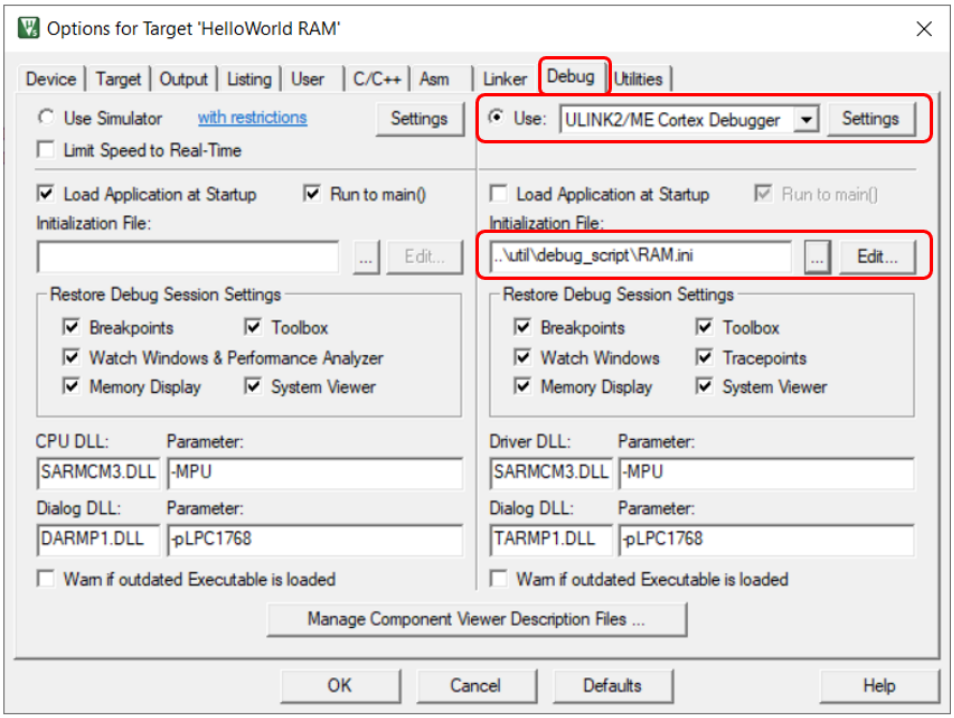

Figure 9.27: Keil IDE: Configure ULINK-ME Hardware Debugger.

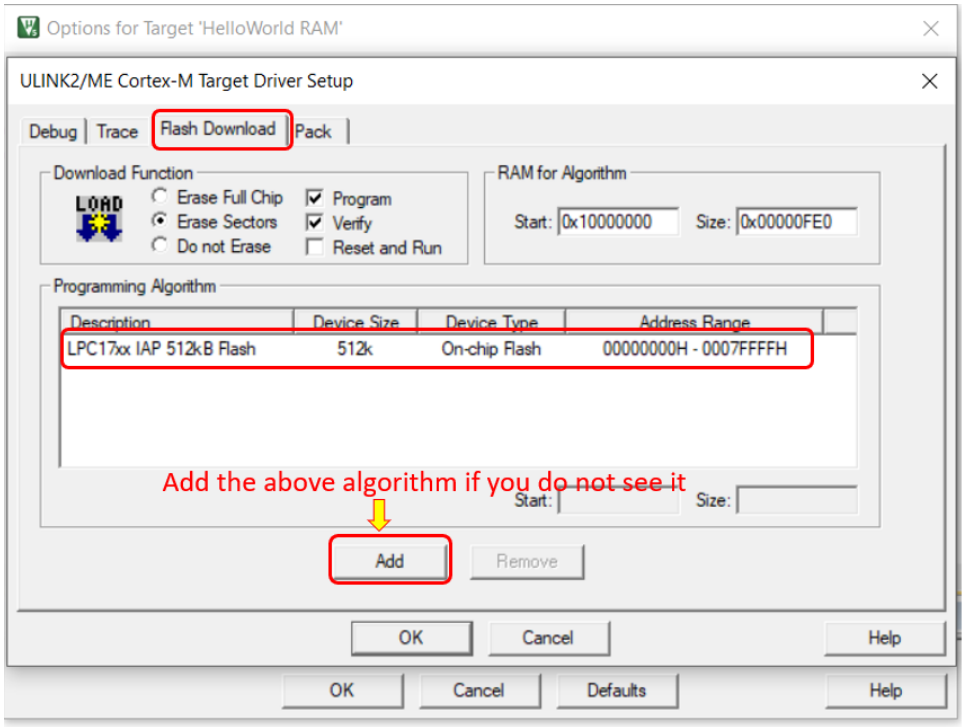

Figure 9.28: Keil IDE: Flash Download Programming Algorithm Configuration.

open up the device manager and expand the Ports (COM & LPT) line (see Figure 9.29). Note the COM port numbers are different for each lab computer. The COM port numbers may also change after a reboot of the computer. An example PuTTY Serial configuration is shown in Figures 9.30 and 9.31.

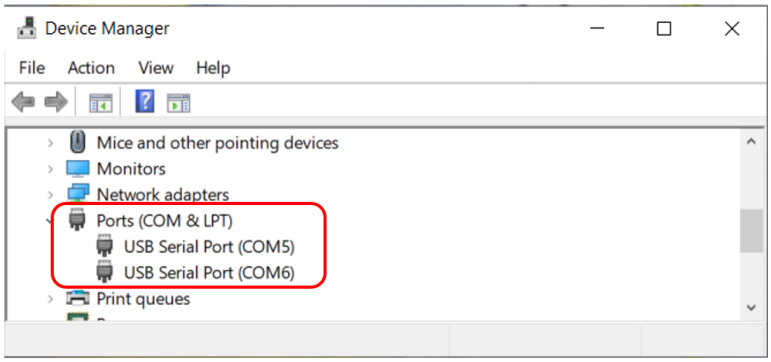

Figure 9.29: Device Manger COM Ports

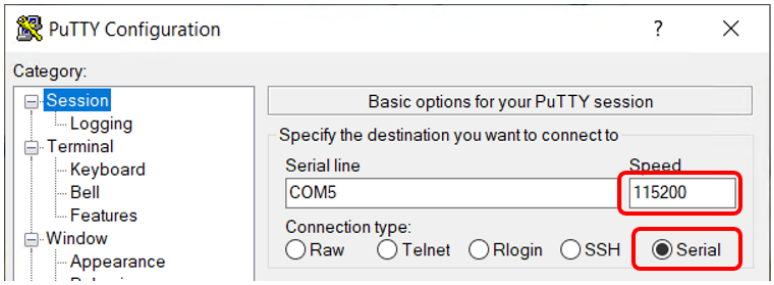

Figure 9.30: PuTTY Session for Serial Port Communication

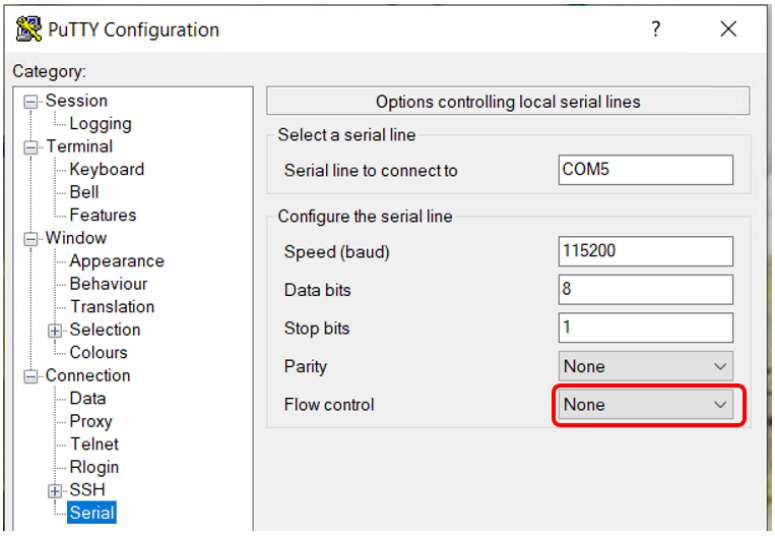

Figure 9.31: PuTTY Serial Port Configuration

7. To download the code to the board, *do not press the LOAD button*. Instead, the *debug button* is pressed to initiate a debug session and the `RAM.ini` file will load the code to the board.

8. Either step through the code or just press the Run button to execute the code till the end. You will see output from your PuTTY terminals (see Figure 9.32).

Figure 9.32: PuTTY Output

9.3 Download to ROM

Though we keep discouraging you to download the image to ROM, we walk you through the steps on how to do it to give you a feel of how a project that is ready to be released is loaded to the ROM. We expect that you already fixed your code by debugging the code on board by using the in-memory execution technique we showed you earlier. You should only do the following experiment once or twice. Please use the ROM sparingly.

Switch your target to the "HelloWorld SIM" target (see Figure 9.34). Open up the target option. Select the Debug tab and press the "Settings" button beside the ULINK2/ME Debugger (upper right portion of the window). Select the "Flash Download" tab and check the box "Reset and Run" in the Download Function section (See Figure 9.33). This will execute the code automatically without the need to press the physical reset button on the board. Apply all the changes and close the target options configuration window.

To download the code to the on-chip ROM, click the "Load" button (see Figure 9.34). The download is through the ULINK-ME. The code automatically runs. You should see the output from PuTTY terminals.

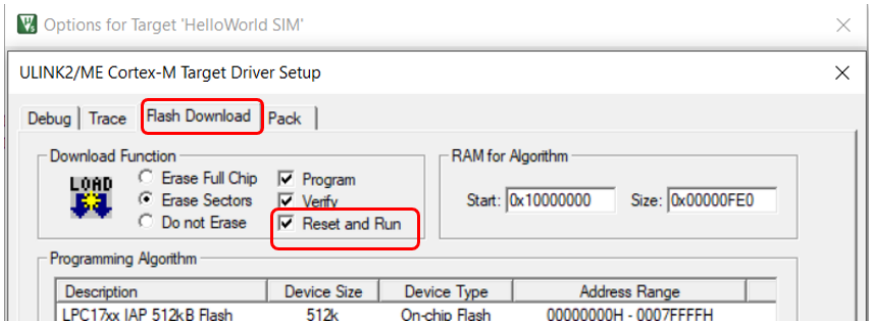

Figure 9.33: Flash Download Reset and Run Setting

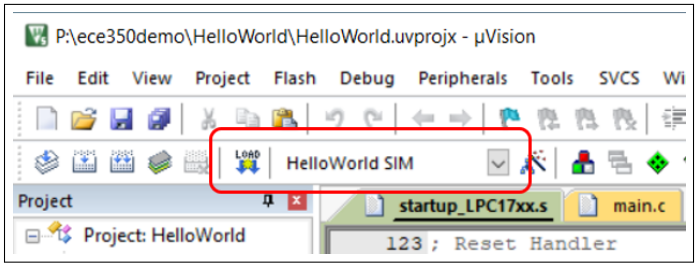

Figure 9.34: Keil IDE: Download Target to Flash

Chapter 10

Programming MCB1700

10.1 The Thumb-2 Instruction Set Architecture

The Cortex-M3 supports only the Thumb-2 (and traditional Thumb) instruction set. With support for both 16-bit and 32-bit instructions in the Thumb-2 instruction set, there is no need to switch the processor between Thumb state (16-bit instructions) and ARM state (32-bit instructions).

In the RTOS lab, you will need to program a little bit 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 10.1 lists some assembly instructions that the RTX project may use. For complete instruction set reference, we refer the reader to Section 34.2 (ARM Cortex-M3 User Guide: Instruction Set) in [4].

10.2 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 10.2 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

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 10.1: 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 10.2: Core Registers and AAPCS Usage

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.

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.

10.3 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 10.1). This improves software portability and re-usability. It enables soft-

ware solution suppliers to develop products that can work seamlessly with device libraries from various silicon vendors [2].

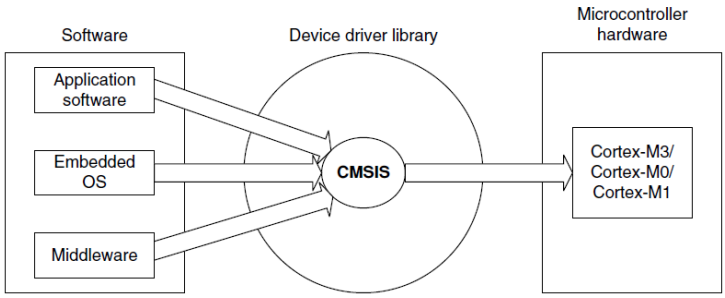

Figure 10.1: Role of CMSIS[5]

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. `LPC17xx.h`) and system startup code files (e.g., `startup_LPC17xx.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.

10.3.1 CMSIS files

The CMSIS is divided into multiple layers (See Figure 10.2). For each device, the MCU vendor provides a device header file `<device>.h` (e.g., `LPC17xx.h`) which pulls in additional header files required by the device driver library and the Core Peripheral Access Layer (see Figure 10.3).

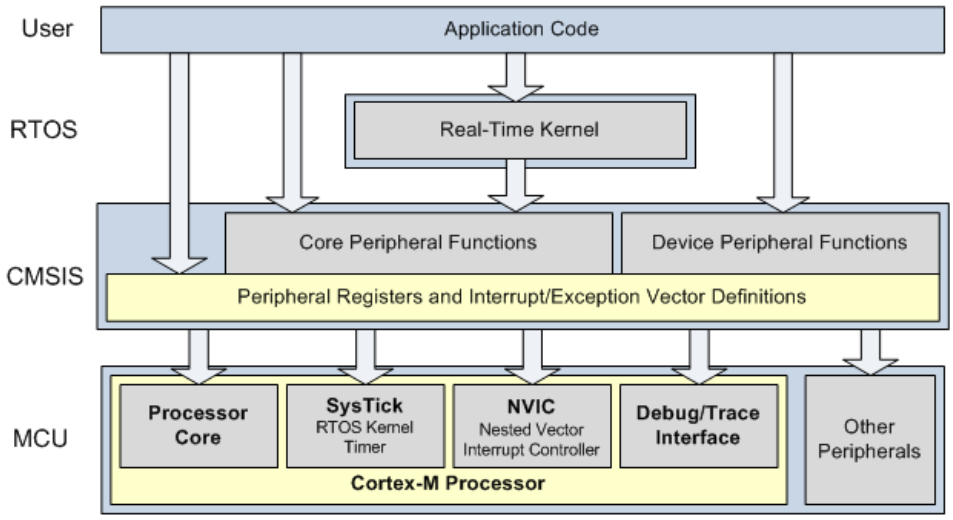

Figure 10.2: CMSIS Organization[2]

By including the `<device>.h` (e.g., `LPC17xx.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 10.1.

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

Listing 10.1: CMSIS SystemInit()

The CMSIS compliant device drivers also contain a startup code (e.g., `startup_LPC17xx.s`), which include the vector table with standardized exception handler names (See Section 10.3.3.

10.3.2 Cortex-M Core Peripherals

We only introduce the NVIC programming in this section. The Nested Vectored Interrupt Controller (NVIC) can be accessed by using CMSIS functions (see Figure 10.4). As an example, the following code enables the UART0 and TIMER0 interrupt

```
NVIC_EnableIRQ(UART0_IRQn); // UART0_IRQn is defined in LPC17xx.h
NVIC_EnableIRQ(TIMER0_IRQn); // TIMER0_IRQn is defined in LPC17xx.h
```

10.3.3 System Exceptions

Writing an exception handler becomes very easy. One just defines a function that takes no input parameter and returns void. The function takes the name of the standardized exception handler name as defined in the startup code (e.g., `startup_LPC17xx.s`).

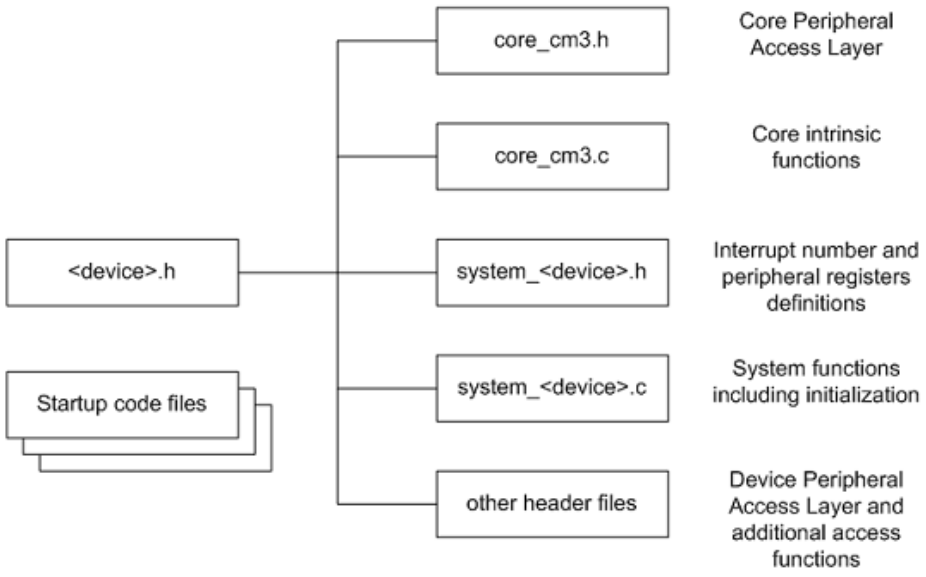

Figure 10.3: CMSIS Organization[2]

	Function definition	Description
void	**NVIC_SystemReset** (void)	Resets the whole system including peripherals.
void	**NVIC_SetPriorityGrouping** (uint32_t priority_grouping)	Sets the priority grouping.
uint32_t	**NVIC_GetPriorityGrouping** (void)	Returns the value of the current priority grouping.
void	**NVIC_EnableIRQ** (IRQn_Type IRQn)	Enables the interrupt IRQn.
void	**NVIC_DisableIRQ** (IRQn_Type IRQn)	Disables the interrupt IRQn.
void	**NVIC_SetPriority** (IRQn_Type IRQn, int32_t priority)	Sets the priority for the interrupt IRQn.
uint32_t	**NVIC_GetPriority** (IRQn_Type IRQn)	Returns the priority for the specified interrupt.
void	**NVIC_SetPendingIRQ** (IRQn_Type IRQn)	Sets the interrupt IRQn pending.
IRQn_Type	**NVIC_GetPendingIRQ** (IRQn_Type IRQn)	Returns the pending status of the interrupt IRQn.
void	**NVIC_ClearPendingIRQ** (IRQn_Type IRQn)	Clears the pending status of the interrupt IRQn, if it is not already running or active.
IRQn_Type	**NVIC_GetActive** (IRQn_Type IRQn)	Returns the active status for the interrupt IRQn.

Figure 10.4: CMSIS NVIC Functions[2]

The following listing shows an example to write the UART0 interrupt handler entirely in C.

```
void UART0_Handler (void)
{
   // write your IRQ here
}
```

Another way is to use the embedded assembly code:

Instruction		CMSIS Intrinsic Function
CPSIE I		void __enable_irq(void)
CPSID I		void __disable_irq(void)
Special Register	**Access**	**CMSIS Function**
CONTROL	Read	uinit32_t __get_CONTROL(void)
	Write	void __set_CONTROL(uint32_t value)
MSP	Read	uinit32_t __get_MSP(void)
	Write	void __set_MSP(uint32_t value)
PSP	Read	uinit32_t __get_PSP(void)
	Write	void __set_PSP(uint32_t value)

Table 10.3: CMSIS intrinsic functions defined in core_cmFunc.h

```
__asm void UART0_Handler(void)
{
   ; do some asm instructions here
   BL __cpp(a_c_function) ; a_c_function is a regular C function
   ; do some asm instructions here,
}
```

10.3.4 Intrinsic Functions

ANSI cannot directly access some Cortex-M3 instructions. The CMSIS provides intrinsic functions that can generate these instructions. The CMSIS also provides a number of functions for accessing the special registers using MRS and MSR instructions. The intrinsic functions are provided by the RealView Compiler. Table 10.3 lists some intrinsic functions that your RTOS project most likely will need to use. We refer the reader to Tables 613 and 614 one page 650 in Section 34.2.2 of [4] for the complete list of intrinsic functions.

10.3.5 Vendor Peripherals

All vendor peripherals are organized as C structure in the <device>.h file (e.g., LPC17xx.h). For example, to read a character received in the RBR of UART0, we can use the following code.

```
unsigned char ch;
ch = LPC_UART0->RBR; // read UART0 RBR and save it in ch
```

10.4 Accessing C Symbols from Assembly

Only embedded assembly is support in Cortex-M3. To write an embedded assembly function, you need to use the __asm keyword. For example the the function "embedded_asm_function" in Listing 10.3 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 10.2, we have two C global variables g_pcb and g_var. We can use the __cpp to access them as shown in Listing 10.3. Note to access the value of a variable, it needs to be a constant variable. For a non-constant variable, the assembly code access the address of the variable.

```
#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;
const U32 g_var;
```

Listing 10.2: 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, which is
      a constant
   STR R4, [R3, #SP_OFFSET] ; write R4 value to g_pcb.mp_sp
}
```

Listing 10.3: Example of accessing global variable from assembly

- A C function. In Listing 10.4, 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 10.4: Example of accessing c function from assembly

- A constant expression in the range of $0 - 255$ defined in C. In Listing 10.5, `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 10.5: 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 10.6).

```
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 10.6: 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.

10.5 SVC Programming: Writing an RTX API Function

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 10.5). Effectively, the RTX API function is a wrapper that invokes `SVC` exception handler and passes corresponding kernel service operation information to the SVC handler.

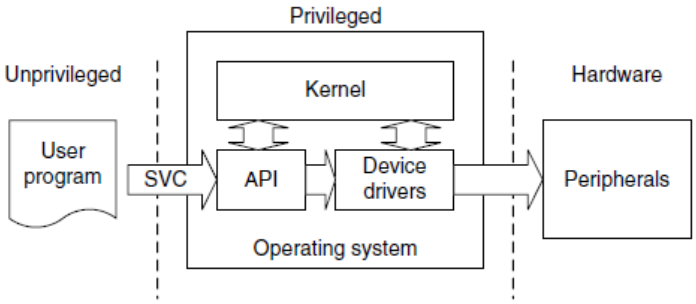

Figure 10.5: SVC as a Gateway for OS Functions [5]

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 10.7.

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

Listing 10.7: 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 10.8 is an excerpt of the HAL.c from the starter code.

```
__asm void SVC_Handler(void) {
   MRS R0, PSP

   ;Extract SVC number, if SVC 0, then do the following
   LDM R0, {R0-R3, R12}; Read R0-R3, R12 from stack

   ; code to save cpu registers omitted

   BLX R12 ; R12 contains the kernel function entry point

   ;Code to restore registers omitted

   MVN LR, #:NOT:0xFFFFFFFD; set EXC_RETURN, thread mode, PSP
   BX  LR              ;
}
```

Listing 10.8: 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`[3]. 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 10.8 then can be used to handle the `SVC 0` exception.

10.6 UART Programming

To program a UART on MCB1700 board, one first needs to configure the UART by following the steps listed in Section 15.1 in [4] (referred as LPC17xx_UM in the sample code comments). Listings 10.9, 10.10 and 10.11 give one sample implementation of programming UART0 interrupts.

```
/**
 * @brief: UART defines
 * @file: uart_def.h
 * @author: Yiqing Huang
 * @date: 2014/02/08
 */

#ifndef UART_DEF_H_
#define UART_DEF_H_

/* The following macros are from NXP uart.h */
#define IER_RBR  0x01
#define IER_THRE 0x02
#define IER_RLS  0x04

#define IIR_PEND 0x01
#define IIR_RLS  0x03
#define IIR_RDA  0x02
#define IIR_CTI  0x06
#define IIR_THRE 0x01

#define LSR_RDR  0x01
#define LSR_OE  0x02
#define LSR_PE  0x04
#define LSR_FE  0x08
#define LSR_BI  0x10
#define LSR_THRE 0x20
#define LSR_TEMT 0x40
#define LSR_RXFE 0x80

#define BUFSIZE  0x40
/* end of NXP uart.h file reference */


/* convenient macro for bit operation */
#define BIT(X) ( 1 << X )

/*
  8 bits, no Parity, 1 Stop bit

  0x83 = 1000 0011 = 1 0 00 0 0 11
  LCR[7] =1 enable Divisor Latch Access Bit DLAB
  LCR[6] =0 disable break transmission
  LCR[5:4]=00 odd parity
  LCR[3] =0 no parity
  LCR[2] =0 1 stop bit
  LCR[1:0]=11 8-bit char len
  See table 279, pg306 LPC17xx_UM
*/
#define UART_8N1 0x83

#ifndef NULL
#define NULL 0
```

```
#endif


#endif /* !UART_DEF_H_ */
```

Listing 10.9: UART0 IRQ Sample Code uart_def.h

```
/**
 * @brief: uart.h
 * @author: Yiqing Huang
 * @date: 2014/02/08
 */

#ifndef UART_IRQ_H_
#define UART_IRQ_H_

/* typedefs */
#include <stdint.h>
#include "uart_def.h"

/* The following macros are from NXP uart.h */
/*
#define IER_RBR  0x01
#define IER_THRE 0x02
#define IER_RLS  0x04

#define IIR_PEND 0x01
#define IIR_RLS  0x03
#define IIR_RDA  0x02
#define IIR_CTI  0x06
#define IIR_THRE 0x01

#define LSR_RDR  0x01
#define LSR_OE  0x02
#define LSR_PE  0x04
#define LSR_FE  0x08
#define LSR_BI  0x10
#define LSR_THRE 0x20
#define LSR_TEMT 0x40
#define LSR_RXFE 0x80

#define BUFSIZE  0x40
*/
/* end of NXP uart.h file reference */


/* convenient macro for bit operation */
//#define BIT(X) ( 1 << X )

/*
  8 bits, no Parity, 1 Stop bit

  0x83 = 1000 0011 = 1 0 00 0 0 11
  LCR[7] =1 enable Divisor Latch Access Bit DLAB
```

```
  LCR[6] =0 disable break transmission
  LCR[5:4]=00 odd parity
  LCR[3] =0 no parity
  LCR[2] =0 1 stop bit
  LCR[1:0]=11 8-bit char len
  See table 279, pg306 LPC17xx_UM
*/
//#define UART_8N1 0x83


#define uart0_irq_init() uart_irq_init(0)
#define uart1_irq_init() uart_irq_init(1)

/* initialize the n_uart to use interrupt */
int uart_irq_init(int n_uart);

#endif /* ! UART_IRQ_H_ */
```

Listing 10.10: UART0 IRQ Sample Code uart.h

```
/**
 * @brief: uart_irq.c
 * @author: NXP Semiconductors
 * @author: Y. Huang
 * @date: 2014/02/08
 */

#include <LPC17xx.h>
#include "uart.h"
#include "uart_polling.h"
#ifdef DEBUG_0
#include "printf.h"
#endif


uint8_t g_buffer[]= "You Typed a Q\n\r";
uint8_t *gp_buffer = g_buffer;
uint8_t g_send_char = 0;
uint8_t g_char_in;
uint8_t g_char_out;

/**
 * @brief: initialize the n_uart
 * NOTES: It only supports UART0. It can be easily extended to support
    UART1 IRQ.
 * The step number in the comments matches the item number in Section 14.1
     on pg 298
 * of LPC17xx_UM
 */
int uart_irq_init(int n_uart) {

  LPC_UART_TypeDef *pUart;

  if ( n_uart ==0 ) {
```

```
  /*
  Steps 1 & 2: system control configuration.
  Under CMSIS, system_LPC17xx.c does these two steps

  -----------------------------------------------------
  Step 1: Power control configuration.
        See table 46 pg63 in LPC17xx_UM
  -----------------------------------------------------
  Enable UART0 power, this is the default setting
  done in system_LPC17xx.c under CMSIS.
  Enclose the code for your refrence
  //LPC_SC->PCONP |= BIT(3);

  -----------------------------------------------------
  Step2: Select the clock source.
       Default PCLK=CCLK/4 , where CCLK = 100MHZ.
       See tables 40 & 42 on pg56-57 in LPC17xx_UM.
  -----------------------------------------------------
  Check the PLL0 configuration to see how XTAL=12.0MHZ
  gets to CCLK=100MHZin system_LPC17xx.c file.
  PCLK = CCLK/4, default setting after reset.
  Enclose the code for your reference
  //LPC_SC->PCLKSEL0 &= ~(BIT(7)|BIT(6));

  -----------------------------------------------------
  Step 5: Pin Ctrl Block configuration for TXD and RXD
        See Table 79 on pg108 in LPC17xx_UM.
  -----------------------------------------------------
  Note this is done before Steps3-4 for coding purpose.
  */

  /* Pin P0.2 used as TXD0 (Com0) */
  LPC_PINCON->PINSEL0 |= (1 << 4);

  /* Pin P0.3 used as RXD0 (Com0) */
  LPC_PINCON->PINSEL0 |= (1 << 6);

  pUart = (LPC_UART_TypeDef *) LPC_UART0;

} else if ( n_uart == 1) {

  /* see Table 79 on pg108 in LPC17xx_UM */
  /* Pin P2.0 used as TXD1 (Com1) */
  LPC_PINCON->PINSEL4 |= (2 << 0);

  /* Pin P2.1 used as RXD1 (Com1) */
  LPC_PINCON->PINSEL4 |= (2 << 2);

  pUart = (LPC_UART_TypeDef *) LPC_UART1;

} else {
  return 1; /* not supported yet */
}
```

```
/*
----------------------------------------------------
Step 3: Transmission Configuration.
      See section 14.4.12.1 pg313-315 in LPC17xx_UM
      for baud rate calculation.
----------------------------------------------------
    */

/* Step 3a: DLAB=1, 8N1 */
pUart->LCR = UART_8N1; /* see uart.h file */

/* Step 3b: 115200 baud rate @ 25.0 MHZ PCLK */
pUart->DLM = 0; /* see table 274, pg302 in LPC17xx_UM */
pUart->DLL = 9; /* see table 273, pg302 in LPC17xx_UM */

/* FR = 1.507 ~ 1/2, DivAddVal = 1, MulVal = 2
  FR = 1.507 = 25MHZ/(16*9*115200)
  see table 285 on pg312 in LPC_17xxUM
*/
pUart->FDR = 0x21;



/*
----------------------------------------------------
Step 4: FIFO setup.
     see table 278 on pg305 in LPC17xx_UM
----------------------------------------------------
    enable Rx and Tx FIFOs, clear Rx and Tx FIFOs
Trigger level 0 (1 char per interrupt)
*/

pUart->FCR = 0x07;

/* Step 5 was done between step 2 and step 4 a few lines above */

/*
----------------------------------------------------
Step 6 Interrupt setting and enabling
----------------------------------------------------
*/
/* Step 6a:
  Enable interrupt bit(s) wihtin the specific peripheral register.
      Interrupt Sources Setting: RBR, THRE or RX Line Stats
  See Table 50 on pg73 in LPC17xx_UM for all possible UART0 interrupt
     sources
  See Table 275 on pg 302 in LPC17xx_UM for IER setting
*/
/* disable the Divisior Latch Access Bit DLAB=0 */
pUart->LCR &= ~(BIT(7));

//pUart->IER = IER_RBR | IER_THRE | IER_RLS;
pUart->IER = IER_RBR | IER_RLS;
```

```
  /* Step 6b: enable the UART interrupt from the system level */

  if ( n_uart == 0 ) {
    NVIC_EnableIRQ(UART0_IRQn); /* CMSIS function */
  } else if ( n_uart == 1 ) {
    NVIC_EnableIRQ(UART1_IRQn); /* CMSIS function */
  } else {
    return 1; /* not supported yet */
  }
  pUart->THR = '\0';
  return 0;
}


/**
 * @brief: use CMSIS ISR for UART0 IRQ Handler
 * NOTE: This example shows how to save/restore all registers rather than
     just
 *     those backed up by the exception stack frame. We add extra
 *     push and pop instructions in the assembly routine.
 *     The actual c_UART0_IRQHandler does the rest of irq handling
 */
__asm void UART0_IRQHandler(void)
{
  PRESERVE8
  IMPORT c_UART0_IRQHandler
  PUSH{r4-r11, lr}
  BL c_UART0_IRQHandler
  POP{r4-r11, pc}
}
/**
 * @brief: c UART0 IRQ Handler
 */
void c_UART0_IRQHandler(void)
{
  uint8_t IIR_IntId; // Interrupt ID from IIR
  LPC_UART_TypeDef *pUart = (LPC_UART_TypeDef *)LPC_UART0;

#ifdef DEBUG_0
  uart1_put_string("Entering c_UART0_IRQHandler\n\r");
#endif // DEBUG_0

  /* Reading IIR automatically acknowledges the interrupt */
  IIR_IntId = (pUart->IIR) >> 1 ; // skip pending bit in IIR
  if (IIR_IntId & IIR_RDA) { // Receive Data Avaialbe
    /* read UART. Read RBR will clear the interrupt */
    g_char_in = pUart->RBR;
#ifdef DEBUG_0
    uart1_put_string("Reading a char = ");
    uart1_put_char(g_char_in);
    uart1_put_string("\n\r");
#endif // DEBUG_0

    g_buffer[12] = g_char_in; // nasty hack
```

```
   g_send_char = 1;
  } else if (IIR_IntId & IIR_THRE) {
  /* THRE Interrupt, transmit holding register becomes empty */

   if (*gp_buffer != '\0' ) {
     g_char_out = *gp_buffer;
#ifdef DEBUG_0
     //uart1_put_string("Writing a char = ");
     //uart1_put_char(g_char_out);
     //uart1_put_string("\n\r");

     // you could use the printf instead
     printf("Writing a char = %c \n\r", g_char_out);
#endif // DEBUG_0
     pUart->THR = g_char_out;
     gp_buffer++;
   } else {
#ifdef DEBUG_0
     uart1_put_string("Finish writing. Turning off IER_THRE\n\r");
#endif // DEBUG_0
     pUart->IER ^= IER_THRE; // toggle the IER_THRE bit
     pUart->THR = '\0';
     g_send_char = 0;
     gp_buffer = g_buffer;
   }

  } else { /* not implemented yet */
#ifdef DEBUG_0
     uart1_put_string("Should not get here!\n\r");
#endif // DEBUG_0
   return;
  }
}
```

Listing 10.11: UART0 IRQ Sample Code uart_irq.c

Listings 10.12 and 10.13 give one sample implementation of programming UART0 by polling.

```
/**
 * @brief: uart_polling.h
 * @author: Yiqing Huang
 * @date: 2014/01/05
 */

#ifndef UART_POLLING_H_
#define UART_POLLING_H_

#include <stdint.h> /* typedefs */
#include "uart_def.h"

#define uart0_init() uart_init(0)
#define uart0_get_char() uart_get_char(0)
#define uart0_put_char(c) uart_put_char(0,c)
```

```
#define uart0_put_string(s) uart_put_string(0,s)

#define uart1_init() uart_init(1)
#define uart1_get_char() uart_get_char(1)
#define uart1_put_char(c) uart_put_char(1,c)
#define uart1_put_string(s) uart_put_string(1,s)

int uart_init(int n_uart); /* initialize the n_uart */
int uart_get_char(int n_uart); /* read a char from the n_uart */
int uart_put_char(int n_uart, unsigned char c); /* write a char to n_uart
    */
int uart_put_string(int n_uart, unsigned char *s);/* write a string to
   n_uart */
void putc(void *p, char c); /* call back function for printf, use uart1
   */

#endif /* ! UART_POLLING_H_ */
```

Listing 10.12: UART0 IRQ Sample Code uart_polling.h

```
/**
 * @brief: uart_polling.c, polling UART to send and receive data
 * @author: Yiqing Huang
 * @date: 2014/01/05
 * NOTE: the code only handles UART0 for now.
 */

#include <LPC17xx.h>
#include "uart_polling.h"


/**
 * @brief: initialize the n_uart
 * NOTES: only tested uart0 so far, but can be easily extended to other
    uarts.
 *      it should work with uart1, but no testing was done.
 */
int uart_init(int n_uart) {

 LPC_UART_TypeDef *pUart; /* ptr to memory mapped device UART, check */
                     /* LPC17xx.h for UART register C structure overlay
                        */

 if (n_uart == 0 ) {
   /*
   Step 1: system control configuration

   step 1a: power control configuration, table 46 pg63
   enable UART0 power, this is the default setting
   also already done in system_LPC17xx.c
   enclose the code below for reference
   LPC_SC->PCONP |= BIT(3);
```

```
   step 1b: select the clock source, default PCLK=CCLK/4 , where CCLK =
      100MHZ.
   tables 40 and 42 on pg56 and pg57
   Check the PLL0 configuration to see how XTAL=12.0MHZ gets to CCLK=100
      MHZ
   in system_LPC17xx.c file
   enclose code below for reference
   LPC_SC->PCLKSEL0 &= ~(BIT(7)|BIT(6)); // PCLK = CCLK/4, default
      setting after reset

   Step 2: Pin Ctrl Block configuration for TXD and RXD
   Listed as item #5 in LPC_17xxum UART0/2/3 manual pag298
   */
   LPC_PINCON->PINSEL0 |= (1 << 4); /* Pin P0.2 used as TXD0 (Com0) */
   LPC_PINCON->PINSEL0 |= (1 << 6); /* Pin P0.3 used as RXD0 (Com0) */

   pUart = (LPC_UART_TypeDef *) LPC_UART0;

  } else if (n_uart == 1) {
   LPC_PINCON->PINSEL4 |= (2 << 0); /* Pin P2.0 used as TXD1 (Com1) */
   LPC_PINCON->PINSEL4 |= (2 << 2); /* Pin P2.1 used as RXD1 (Com1) */

   pUart = (LPC_UART_TypeDef *) LPC_UART1;

  } else {
   return -1; /* not supported yet */
  }

  /* Step 3: Transmission Configuration */

  /* step 3a: DLAB=1, 8N1 */
  pUart->LCR = UART_8N1;

  /* step 3b: 115200 baud rate @ 25.0 MHZ PCLK */
  pUart->DLM = 0;
  pUart->DLL = 9;
  pUart->FDR = 0x21;  /* FR = 1.507 ~ 1/2, DivAddVal = 1, MulVal = 2 */
                      /* FR = 1.507 = 25MHZ/(16*9*115200) */
  pUart->LCR &= ~(BIT(7)); /* disable the Divisior Latch Access Bit DLAB=0
      */

  return 0;
}


/**
 * @brief: read a char from the n_uart, blocking read
 */

int uart_get_char(int n_uart)
{
  LPC_UART_TypeDef *pUart;

  if (n_uart == 0) {
```

```
    pUart = (LPC_UART_TypeDef *) LPC_UART0;
  } else if (n_uart == 1) {
    pUart = (LPC_UART_TypeDef *) LPC_UART1;
  } else {
    return -1; /* UART2,3 not supported yet */
  }

  /* polling the LSR RDR (Receiver Data Ready) bit to wait it is not empty
      */
  while (!(pUart->LSR & LSR_RDR));
  return (pUart->RBR);
}

/**
 * @brief: write a char c to the n_uart
 */

int uart_put_char(int n_uart, unsigned char c)
{
  LPC_UART_TypeDef *pUart;

  if (n_uart == 0) {
    pUart = (LPC_UART_TypeDef *)LPC_UART0;
  } else if (n_uart == 1) {
    pUart = (LPC_UART_TypeDef *)LPC_UART1;
  } else {
    return -1; // UART2,3 not supported
  }

  /* polling LSR THRE bit to wait it is empty */
  while (!(pUart->LSR & LSR_THRE));
  return (pUart->THR = c); /* write c to the THR */
}

/**
 * @brief write a string to UART
 */
int uart_put_string(int n_uart, unsigned char *s)
{
  if (n_uart >1 ) return -1; /* only uart0, 1 are supported for now */
  while (*s !=0) {       /* loop through each char in the string */
    uart_put_char(n_uart, *s++);/* print the char, then ptr increments */
  }
  return 0;
}

/**
 * @brief call back function for printf
 * NOTE: first paramter p is not used for now. UART1 used.
 */
void putc(void *p, char c)
{
  if ( p != NULL ) {
    uart1_put_string("putc: first parameter needs to be NULL");
```

```
  } else {
     uart1_put_char(c);
  }
}
```

Listing 10.13: UART0 IRQ Sample Code uart_polling.c

10.7 Timer Programming

To program a TIMER on MCB1700 board, one first needs to configure the TIMER by following the steps listed in Section 21.1 in [4]. Listings 10.14 and 10.15 give one sample implementation of programming TIMER0 interrupts. The timer interrupt fires every one millisecond.

```
/**
 * @brief timer.h - Timer header file
 * @author Y. Huang
 * @date 2013/02/12
 */
#ifndef _TIMER_H_
#define _TIMER_H_

extern uint32_t timer_init ( uint8_t n_timer ); /* initialize timer
    n_timer */

#endif /* ! _TIMER_H_ */
```

Listing 10.14: Timer0 IRQ Sample Code timer.h

```
/**
 * @brief timer.c - Timer example code. Tiemr IRQ is invoked every 1ms
 * @author T. Reidemeister
 * @author Y. Huang
 * @author NXP Semiconductors
 * @date 2012/02/12
 */

#include <LPC17xx.h>
#include "timer.h"

#define BIT(X) (1<<X)

volatile uint32_t g_timer_count = 0; // increment every 1 ms

/**
 * @brief: initialize timer. Only timer 0 is supported
 */
uint32_t timer_init(uint8_t n_timer)
{
  LPC_TIM_TypeDef *pTimer;
```

```
if (n_timer == 0) {
  /*
  Steps 1 & 2: system control configuration.
  Under CMSIS, system_LPC17xx.c does these two steps

  -----------------------------------------------------
  Step 1: Power control configuration.
        See table 46 pg63 in LPC17xx_UM
  -----------------------------------------------------
  Enable UART0 power, this is the default setting
  done in system_LPC17xx.c under CMSIS.
  Enclose the code for your refrence
  //LPC_SC->PCONP |= BIT(1);

  -----------------------------------------------------
  Step2: Select the clock source,
       default PCLK=CCLK/4 , where CCLK = 100MHZ.
       See tables 40 & 42 on pg56-57 in LPC17xx_UM.
  -----------------------------------------------------
  Check the PLL0 configuration to see how XTAL=12.0MHZ
  gets to CCLK=100MHZ in system_LPC17xx.c file.
  PCLK = CCLK/4, default setting in system_LPC17xx.c.
  Enclose the code for your reference
  //LPC_SC->PCLKSEL0 &= ~(BIT(3)|BIT(2));

  -----------------------------------------------------
  Step 3: Pin Ctrl Block configuration.
        Optional, not used in this example
        See Table 82 on pg110 in LPC17xx_UM
  -----------------------------------------------------
  */
  pTimer = (LPC_TIM_TypeDef *) LPC_TIM0;

} else { /* other timer not supported yet */
  return 1;
}

/*
-----------------------------------------------------
Step 4: Interrupts configuration
-----------------------------------------------------
*/

/* Step 4.1: Prescale Register PR setting
   CCLK = 100 MHZ, PCLK = CCLK/4 = 25 MHZ
   2*(12499 + 1)*(1/25) * 10^(-6) s = 10^(-3) s = 1 ms
   TC (Timer Counter) toggles b/w 0 and 1 every 12500 PCLKs
   see MR setting below
*/
pTimer->PR = 12499;

/* Step 4.2: MR setting, see section 21.6.7 on pg496 of LPC17xx_UM. */
pTimer->MR0 = 1;
```

```
  /* Step 4.3: MCR setting, see table 429 on pg496 of LPC17xx_UM.
    Interrupt on MR0: when MR0 mathches the value in the TC,
                    generate an interrupt.
    Reset on MR0: Reset TC if MR0 mathches it.
  */
  pTimer->MCR = BIT(0) | BIT(1);

  g_timer_count = 0;

  /* Step 4.4: CSMSIS enable timer0 IRQ */
  NVIC_EnableIRQ(TIMER0_IRQn);

  /* Step 4.5: Enable the TCR. See table 427 on pg494 of LPC17xx_UM. */
  pTimer->TCR = 1;

  return 0;
}

/**
 * @brief: use CMSIS ISR for TIMER0 IRQ Handler
 * NOTE: This example shows how to save/restore all registers rather than
    just
 *     those backed up by the exception stack frame. We add extra
 *     push and pop instructions in the assembly routine.
 *     The actual c_TIMER0_IRQHandler does the rest of irq handling
 */
__asm void TIMER0_IRQHandler(void)
{
  PRESERVE8
  IMPORT c_TIMER0_IRQHandler
  PUSH{r4-r11, lr}
  BL c_TIMER0_IRQHandler
  POP{r4-r11, pc}
}
/**
 * @brief: c TIMER0 IRQ Handler
 */
void c_TIMER0_IRQHandler(void)
{
  /* ack inttrupt, see section 21.6.1 on pg 493 of LPC17XX_UM */
  LPC_TIM0->IR = BIT(0);

  g_timer_count++ ;
}
```

Listing 10.15: Timer0 IRQ Sample Code timer.c

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 _____

Appendix B

The RAM.ini File

The RAM.ini file in the starter code can be found in Listing B.1. It relocates the vector table to RAM and load the code for in-memory execution (i.e. not to the ROM). This will avoid wear-and-tear on the on-chip flash memory.

```
FUNC void Setup (void) {
  // Setup Stack Pointer
  SP = _RDWORD(0x10000000);
  // Setup Program Counter
  PC = _RDWORD(0x10000004);
  // Set Thumb bit
  XPSR = 0x01000000;
  // Setup Vector Table Offset Register
  _WDWORD(0xE000ED08, 0x10000000);
  // Enable ADC Power
  _WDWORD(0x400FC0C4, _RDWORD(0x400FC0C4) | 1<<12);
  // Setup ADC Trim
  _WDWORD(0x40034034, 0x00000F00);
}
// Download
LOAD %L INCREMENTAL

// Setup for Running
Setup();
g, main
```

Listing B.1: The RAM.ini file

Bibliography

[1] MCB1700 User's Guide. http://www.keil.com/support/man/docs/mcb1700.

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

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

[4] LPC17xx User Manual, Rev2.0, 2010.

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