

ECE 350

Laboratory Project Manual for

Real-Time Operating Systems

Keil MCB1700 Edition

by

Yiqing Huang
Seyed Majid Zahedi
Rodolfo Pellizzoni

Electrical and Computer Engineering Department
University of Waterloo

Waterloo, Ontario, Canada, June 29, 2021

© Y. Huang, S.M. Zahedi and R. Pellizzoni 2020 - 2021

Contents

List of Tables	vii
List of Figures	xi
Preface	1
I Lab Administration	1
II Lab Project	7
1 Introduction	8
1.1 Overview	8
1.2 Summary of RTX Requirements	8
1.2.1 RTX Tasks	9
1.2.2 RTX Footprint and Processor Loading	10
1.2.3 Error Detection and Recovery	10
1.3 Errata	10
2 Lab0 Group Signup and Introduction to Keil MDK5	11
2.1 Objective	11
2.2 Starter Files	11
2.3 Pre-Lab	12
2.4 Lab Tasks	12
2.4.1 Task #1: Group Signup	12
2.4.2 Task #2: Create a Hello World Application	12
2.5 Deliverable	12

3	Lab1 Kernel Memory Management	13
3.1	Objective	13
3.2	Starter Files	13
3.3	Pre-lab Preparation	13
3.4	Lab Project	14
3.4.1	The Memory Map	14
3.4.2	Dynamic Memory Allocator	16
3.4.3	Design and Implement Issues	17
3.4.4	Implementation Tips	19
3.4.5	Specifications of Functions	20
3.5	Source Code File Organization and Third-party Testing	24
3.5.1	Testing	25
3.6	Lab Report	26
3.7	Deliverable	27
3.7.1	Pre-Lab Deliverables	27
3.7.2	Post-Lab Deliverables	27
3.8	Marking Rubric	27
3.9	Errata	28
4	Lab2 Task Management	29
4.1	Objective	29
4.2	Starter Files	29
4.3	Pre-lab Preparation	30
4.4	Lab Project Part A - Memory System Calls	30
4.4.1	Overview	30
4.4.2	Memory Map	31
4.4.3	Specifications of Functions	33
4.5	Lab Project Part B - Task Management System Calls	35
4.5.1	Overview	36
4.5.2	Data Structures and Algorithm	36
4.5.3	Scheduler	37
4.5.4	Context Switching	39
4.5.5	Macros and User Task Data Structure	41

4.5.6	Specifications of Functions	42
4.5.7	Required Tasks	47
4.6	Source Code File Organization and Third-party Testing	48
4.7	Report	49
4.8	Deliverables	50
4.8.1	Pre-Lab Deliverables	50
4.8.2	Post-Lab Deliverables	50
4.9	Marking Rubric	51
4.10	Errata	51
5	Lab3 Inter-task Communications and Console I/O	53
5.1	Starter Files	53
5.2	Pre-lab Preparation	54
5.3	Lab Project	54
5.3.1	Overview	55
5.3.2	Data Structures and Algorithm	56
5.3.3	Scheduler	57
5.3.4	Context Switching	58
5.3.5	Interrupt Handler and System Tasks	58
5.3.6	Types, Macros and Mssage Header Structure	59
5.3.7	Specifications of Functions	60
5.3.8	System Console I/O Tasks	65
5.3.9	UART0 Interrupt Handler	67
5.3.10	User Tasks	68
5.4	Source Code File Organization and Third-party Testing	68
5.4.1	Report	70
5.5	Deliverables	70
5.5.1	Pre-Lab Deliverables	70
5.5.2	Post-Lab Deliverables	70
5.6	Marking Rubric	71
6	Lab4	72

III Computing Environment and Development Tools Quick Reference Guide

	73
7 Windows 10 Remote Desktop	74
8 Keil Software Development Tools	75
8.1 Getting Started with uVision5 IDE	75
8.2 Getting Starter Code from the GitHub	76
8.3 Start the Keil uVision5 IDE	76
8.4 Create a New uVision5 Project	76
8.5 Managing Project Components	78
8.6 Build the Project Target	81
8.6.1 Configure Target Options	81
8.6.2 Build the Target	83
8.7 Debug the Target	84
8.7.1 Debug the Project in Simulator	84
8.7.2 Debug the Project on the Board by In-Memory Execution	87
8.8 Download to ROM	93
8.9 Create a Library Project	94
8.9.1 Preparing Directory Structure	95
8.9.2 Create a New Library uVision Project	95
8.9.3 Managing Library Project Component	96
8.9.4 Configure a Library Target Options	97
8.9.5 Build the Library Target	98
8.10 Create an Application that Links with a Library	99
8.11 Create a Multi-Project Workspace	100
8.12 Batch Build	102
8.13 Using the Library	104
8.14 Errata	106
9 Programming MCB1700	107
9.1 The Thumb-2 Instruction Set Architecture	107
9.2 ARM Architecture Procedure Call Standard (AAPCS)	107
9.3 Cortex Microcontroller Software Interface Standard (CMSIS)	109

9.3.1	CMSIS files	110
9.3.2	Cortex-M Core Peripherals	111
9.3.3	System Exceptions	111
9.3.4	Intrinsic Functions	113
9.3.5	Vendor Peripherals	113
9.4	Accessing C Symbols from Assembly	114
9.5	SVC Programming: Writing an RTX API Function	115
9.5.1	Programming in Assembly Language	116
9.5.2	Programming in C with ARM Compiler Keywords	117
9.6	UART Programming	118
9.7	Timer Programming	129
10	Keil MCB1700 Hardware Environment	132
10.1	MCB1700 Board Overview	132
10.2	Cortex-M3 Processor	132
10.2.1	Registers	135
10.2.2	Processor mode and privilege levels	136
10.2.3	Stacks	137
10.3	Memory Map	138
10.4	Exceptions and Interrupts	139
10.4.1	Vector Table	139
10.4.2	Exception Entry	139
10.4.3	EXC_RETURN Value	141
10.4.4	Exception Return	142
10.5	Data Types	143
A	Forms	144
B	The Debugger Initialization Files	146
References		147

List of Tables

0.1	Project Deliverable Weight and Deadlines	3
0.2	Group Project contribution factor table. Each student's lab grade is their group project grade multiplied by the CF (Contribution Factor). This scheme only applies to groups who need peer reviews.	5
3.1	Lab1 Marking Rubric	28
4.1	Lab2 Marking Rubric	51
5.1	Lab3 Marking Rubric	71
9.1	Assembler instruction examples	108
9.2	Core Registers and AAPCS Usage	109
9.3	CMSIS intrinsic functions	113
10.1	Summary of processor mode, execution privilege level, and stack use options	138
10.2	LPC1768 Memory Map	138
10.3	LPC1768 Exception and Interrupt Table	140
10.4	EXC_RETURN bit fields	142
10.5	EXC_RETURN Values on Cortex-M3	142

List of Figures

3.1	NXP LPC1768 Memory Map. RAM regions are highlighted.	14
3.2	NXP LPC1768 IRAM1 Memory Execution View. The RTX Image has two libraries (RTX-Lib and AE-Lib) built into it together with the application (RTX-App) that uses the libraries.	15
3.3	NXP LPC1768 IRAM2 Memory Execution View.	15
3.4	A Free Block Format	17
3.5	A memory map.Figure is not drawn to scale.	23
3.6	Lab1 Submission Directory Layout	27
4.1	Lab2 NXP LPC1768 IRAM1 Memory Execution View. There is a block of free space that is not managed.	31
4.2	Lab2 NXP LPC1768 IRAM1 SIM Target Memory Map Configuration. .	32
4.3	Lab2 In-memory Execution RAM Target Memory Map Configuration.	33
4.4	Lab2 Three State Transition Diagram	38
4.5	Lab2 Submission Directory Layout	50
4.6	Lab2 Git Commit History	51
5.1	UART and Host Computer Connection	55
5.2	Lab3 State Transition Diagram	57
5.3	Structure of a message buffer	62
5.4	Message passing among system console I/O tasks, UART interrupt handler and a command handling task communication.	67
5.5	Lab3 Submission Directory Layout	71
8.1	Keil IDE: Create a New Project	76
8.2	Keil IDE: Create a New Project	77
8.3	Keil IDE: Choose MCU	77
8.4	Keil IDE: Manage Run-time Environment	78

8.5	Keil IDE: A default new project	78
8.6	Keil IDE: Add Group	79
8.7	Keil IDE: Updated Project Profile	79
8.8	Keil IDE: Add Source File to Source Group	80
8.9	Keil IDE: Updated Project Profile	80
8.10	Keil IDE: Create New File	81
8.11	Keil IDE: Final Project Setting	81
8.12	Keil IDE: Target Options Configuration	81
8.13	Keil IDE: Target Options Target Tab Configuration	82
8.14	Keil IDE: Target Options C/C++ Tab Configuration	82
8.15	Keil IDE: Target Options Linker Tab Configuration	83
8.16	Keil IDE: Target Options Output Tab Configuration for SIM Target	83
8.17	Keil IDE: Build Target	84
8.18	Keil IDE: Build Target	84
8.19	Keil IDE: Target Options Debug Tab Configuration	85
8.20	Keil IDE: Debug Button	85
8.21	Keil IDE: Debugging. Enable Serial Window View.	85
8.22	Keil IDE: Debugging. Both UART0 and UART1 views are enabled in simulator.	86
8.23	Keil IDE: Debugging. The Run Button.	86
8.24	Keil IDE: Debugging Output.	87
8.25	Keil IDE: Manage Project Items Button	88
8.26	Keil IDE: Manage Project Items Window.	88
8.27	Keil IDE: Select HelloWorld RAM Target.	89
8.28	Keil IDE: Configure Target Options Target Tab for In-memory Execution.	89
8.29	Keil IDE: RAM Target Asm Configuration.	90
8.30	Keil IDE: Configure ULINK-ME Hardware Debugger.	90
8.31	Keil IDE: Flash Download Programming Algorithm Configuration.	91
8.32	Keil IDE: Target Option Utilities Configuration for RAM Target.	91
8.33	Keil IDE: Target Options Output Tab Configuration for RAM Target	92
8.34	Device Manger COM Ports	92
8.35	PuTTY Session for Serial Port Communication	92
8.36	PuTTY Serial Port Configuration	93

8.37	PuTTY Output	93
8.38	Flash Download Reset and Run Setting	94
8.39	Keil IDE: Download Target to Flash	94
8.40	Directory Structure of a Multi-Project Workspace. The HelloWorld directory layout is omitted.	95
8.41	Directory Structure of a Multi-Project Workspace. The HelloWorld directory layout is omitted.	96
8.42	Keil IDE: A Library Project Profile	97
8.43	Keil IDE: Target Options Output Tab Library Creation Configuration	97
8.44	Keil IDE: Target Options C/C++ Tab Configuration	98
8.45	Keil IDE: Build Library Target	98
8.46	Keil IDE: HelloWorld Application that uses a Library	99
8.47	Keil IDE: Removing source code files from HelloWorld inside the Helloworld-Multi folder	99
8.48	Keil IDE: Build Output of HelloWorld Application Linked with a Library	100
8.49	Keil IDE: Create a New Multi-Project Workspace Menu Item	100
8.50	Keil IDE: Create a New Multi-Project Workspace Window	100
8.51	Keil IDE: Final New Multi-Project Workspace Window	101
8.52	Keil IDE: Multi-Project Workspace Explorer	101
8.53	Keil IDE: Batch Setup Menu Item	102
8.54	Keil IDE: Batch Setup Window	103
8.55	Keil IDE: Manage Multi-Project Workspace Button	103
8.56	Keil IDE: Batch Build Button	103
8.57	Keil IDE: Batch Build Output	104
8.58	The main.c code that uses printf	105
8.59	Keil IDE: demonstration of printf using simulator	105
8.60	Keil IDE: demonstration of printf on board	105
9.1	Role of CMSIS	110
9.2	CMSIS Organization	111
9.3	CMSIS Organization	112
9.4	CMSIS NVIC Functions	112
9.5	SVC as a Gateway for OS Functions [6]	116
10.1	MCB1700 Board Components	133

10.2 MCB1700 Board Block Diagram	133
10.3 LPC1768 Block Diagram	134
10.4 Simplified Cortex-M3 Block Diagram	135
10.5 Cortex-M3 Registers	136
10.6 Cortex-M3 Operating Mode and Privilege Level	137
10.7 Cortex-M3 Exception Stack Frame	141

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 four 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

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

- Windows 10 Remote Desktop
- 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 the following generous donations, without which the lab will not be possible:

- ARM University Program for providing us with lab teaching materials and ARM DS gold edition software licenses.
- ARM University Program for providing us with 50 Keil MCB 1700 boards.
- Intel University Program for providing us with 50 DE1-SoC FPGA boards.
- TerasIC, the manufacturer, for shipping the boards in a timely manner.
- Imperas Software for providing us one evaluation license to experiment with their software tools during the lab development.

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

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

Part I

Lab Administration

Lab Administration Policy

Group Lab Policy

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

least one week before the nearest lab project deadline.

Project Submission Policy.

- **Project Deliverables.** The lab project is divided into four 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. Table 0.1 gives the weight, deadline and naming convention of each post-lab deliverable.

Deliverable	Weight	Due Date	File Name
Lab0 Group Sign-up	2%	23:59 EST May 14	group.csv
Lab1 Memory Management	18%	23:59 EST May 28	lab1.zip
Lab2 Task Management	30%	23:59 EST Jun 25	lab2.zip
Lab3 Inter-task Communications and Console I/O	25%	23:59 EST Jul 16	lab3.zip
Lab4 Real-Time Scheduling	25%	23:59 EST Jul 30	lab4.zip

Table 0.1: Project Deliverable Weight and Deadlines.

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

```
#include <math.h>

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

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.*

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

Project Grading Policy

- **Project Grading Procedure.** The project is graded by automated testing framework. For each deliverable, we publish a small set of testing cases. We require students to pass these testing cases before they submit. If you are not able to pass these testing cases, then your project will be graded manually by spot checking the source code, which usually will result in a very undesirable lab grade.
- **Hardware vs. Simulator.** Submissions will be evaluated on a Windows 10 lab machine that has a board attached to it. Lab machines are accessible through [ENGLab remote desktop session](#) when connected to the campus virtual private network ([VPN](#)). If a lab requires the program to run on the board, but the program only functions inside the simulator, a 15% penalty will be applied to the particular lab's grade. If you are not interested in re-grading your project, but want to ask grading TA some questions or advises, you may also request a demo after the grades are released.
- **Project Re-grading.** If you want to appeal your lab grade, you need to initiate a re-grading process by contacting the grading TA in charge first. The re-grading is a rigid process. The entire lab 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.
- **Individual Lab Grade.** Normally everyone in the same group gets the same lab grade, which is the group project grade. If your group has serious workload distribution issue, you should submit the peer review form (available on Learn) to the dropbox on Learn and notify the lab instructor by email to initiate a peer review process. Each group member 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 review each group member's contribution to the project. We will use simple arithmetic average ratings each group member received and assign individual lab grade to each team member by multiplying the group project grade with a contribution percentage factor listed in Table [0.2](#).

Note peer review is optional and only applies to those groups that have group dynamic issues that need to be escalated. Majority of our students work well in their groups.

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

Table 0.2: Group Project contribution factor table. Each student's lab grade is their group project grade multiplied by the CF (Contribution Factor). This scheme only applies to groups who need peer reviews.

Lab Repeating Policy

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

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

Lab Projects Solution Internet Policy

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

Seeking Help

- **Discussion Forum.** We recommend students to use the Piazza discussion forum to ask the teaching team questions instead of sending individual emails to lab teaching staff. For questions related to lab projects, our target response time is one business day before the deadline of the particular lab in question ². There

²Our 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.

is no guarantee on the response time to questions of a lab that passes the submission deadline.

- **Office Hours.** The lab office hours are for group project consultation. Your entire group may attend the same appointment. Each appointment is a 15 minute time slot. Book multiple consecutive time slots if you need more time. All appointments require a minimum 1 hour lead time. The maximum lead time you can book a lab office hour is 5 days. You should cancel your booked appointment if you are not able to attend it so other students who need it can book it. Please do not book an appointment if you are not able to make it.
- **Appointments.** Students can also arrange 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.

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 256 priority levels, preemption, dynamic memory management, mailbox for inter-task communications and synchronization, a basic timing service, system console I/O and debugging support, and finally real-time scheduling.

Such an RTX is suitable for embedded computers which operate in real time. A cooperative, non-malicious software environment is assumed. The design of the RTX should allow its placement in ROM. Applications and non-kernel RTX tasks must execute at the unprivileged level of LPC1768. The RTX kernel will execute at the privileged level. There are two banks of 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:

Memory Management

Binary buddy system dynamic memory allocation is supported. Refer to Chapter 3 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 255 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 [4](#) for details.

Synchronization and Console I/O

The RTX provides mailbox utility for inter-task communication and synchronization. An interrupt-driven UART provides the console service. Refer to Chapter [5](#) for details.

Timing Service and Real-Time Scheduling

The Polling Server with RM (Rate Monotonic) scheduling policy is supported. Refer to Chapter [6](#) for details.

1.2.1 RTX Tasks

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

User Tasks

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

System Tasks

These tasks are operating in user mode or supervisor mode. Some may require a privileged level of operation and some may be sufficient to operate at a 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 [3](#)), console display task and keyboard command decoder task (see Chapter [5](#)).

1.2.2 RTX Footprint and Processor Loading

A reasonably *lean* implementation is expected. No standard C library function call is allowed in the kernel code. An implementation of simplified c library function of `printf` is provided in the starter code.

1.2.3 Error Detection and Recovery

The primitive will return a non-zero integer value upon an error and set the `errno` accordingly. No error recovery is required. It may be assumed that the application processes can deal with this situation.

1.3 Errata

1. Page 9, first paragraph. The first sentence is removed. The “is” is removed from the sentence “The RTX is supports task preemption.”.
2. Page 9, section 1.2.1, the first sentence in User Tasks sub-section, “a privileged level” changed to “the unprivileged level”.

Chapter 2

Lab0 Group Signup and Introduction to Keil MDK5

2.1 Objective

This lab is to get you prepared for the lab project development. After this lab, students will be able to

- find a lab project group on Learn
- use Keil uVision (Integrated Development Environment) to edit, debug, simulate and execute a bare-metal uVision project written in C and assembly;

2.2 Starter Files

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

- submission/group.csv: lab0 deliverable template file
- manual_code/util/printf_uart: The printf source code and the uart polling source code. The printf outputs to the UART1 by polling.
- manual_code/util/debug_script: RAM.ini that initializes debugger to load code for in-memory execution; and the SIM.ini that maps the second bank of memory region on the board for the simulator to have read and write access.
- manual_code/lab0>HelloWorld/: a bare-metal project that outputs strings to UART0 and UART1 by polling. This is the solution of Task #2 of lab tasks for your reference.

2.3 Pre-Lab

Read Part I, Chapter 1, Chapter 7 and Chapter 8 up to Section 8.8 of the lab manual.

2.4 Lab Tasks

2.4.1 Task #1: Group Signup

Enroll yourself to a lab project group on Learn. One of your group members should submit the `group.csv` by the deadline.

2.4.2 Task #2: Create a Hello World Application

Follow the steps in Sections 8.1 - 8.8. Create a HelloWorld application for NXP LPC1768 microcontroller using MDK5 uVision. Create two different targets:

- A SIM target that can be debugged inside the simulator; and
- A RAM target that uses the debugger to load the image to RAM and executes on the physical hardware.

Perform the following experiments:

- Build the SIM target and execute it inside the simulator by using the debugger.
- Build the RAM target and load it to RAM to execute it on the board by using the hardware debugger.
- Download the SIM target to ROM and execute it on the board without using any debugger.

2.5 Deliverable

The filled `group.csv` file needs to be submitted by one of your group members. Note this is a group submission, only one of the group members need to submit it.

Chapter 3

Lab1 Kernel Memory Management

3.1 Objective

This lab is to introduce kernel memory programming for the NXP LPC1768 microcontroller populated on the Keil MCB1700 board. You will create a set of kernel memory management functions that will be used in future labs to allocate kernel memory as well as to construct system calls that your RTX provides to user space. After this lab you will be able to

- Design and implement a kernel memory allocator using the binary buddy system.

3.2 Starter Files

The starter file is on GitHub at http://github.com/yqh/ece350/tree/master/manual_code/lab1/. It contains a multi-project workspace profile which contains the following individual projects.

- RTX-Lib/: The RTX kernel library project template for your lab1.
- AE-Lib/: The testing suite library project which contains some testing cases written by the lab teaching staff.
- RTX-App/: A uVision application project that links with the RTX-Lib and the AE-Lib libraries.

3.3 Pre-lab Preparation

- Create a software library (see Section 8.9).
- Create an application that links with a Library (See Section 8.10).

- Create a multi-project workspace (see Sections 8.11 - 8.13).
- Execute the RTX-App project in the simulator and on the board.
- Review the binary buddy system memory allocator algorithm in lecture materials.
- Read the Section “Finding integer log base 2 of an integer (aka the position of the highest bit set)” at <https://graphics.stanford.edu/~seander/bithacks.html> and program a helper function to compute the $\lceil \log_2(S) \rceil$ and $\lfloor \log_2(S) \rfloor$, where S is an unsigned integer.

3.4 Lab Project

You are to implement the binary buddy system dynamic memory allocator for the kernel to manage physical on-chip random access memory (RAM) on the board. So let’s first look at the memory map of the board.

3.4.1 The Memory Map

When we get a new board, one of the most important technical specifications is the memory map. The NXP LPC1768 board has two banks 32 KiB RAM (see Figure 3.1). We name the first bank of memory as IRAM1 and the second bank of memory of IRAM2.

0x2008 4000		
0x2007 C000	32 KiB	AHB SRAM (2 blocks of 16 KiB) (IRAM2)
0x1FFF 2000		Reserved
0x1FFF 0000	8 KiB	Boot ROM
0x1000 8000		Reserved
0x1000 0000	32 KiB	Local SRAM (IRAM1)
0x0008 0000		Reserved
0x0000 0000	512 KiB	On-chip flash

Figure 3.1: NXP LPC1768 Memory Map. RAM regions are highlighted.

The starting address and the size of each bank of the RAM are defined in the

`lpc1768_mem.h` file. The IRAM1 is mainly for keeping the RTX image¹. in residence. The space between the end of the RTX image and the end of IRAM1 is free for kernel to manage (see Figure 3.2). To find out the ending address of the RTX image, we use the address of the `Image$$RW_IRAM1$$ZI$$Limit`, a linker defined symbol.

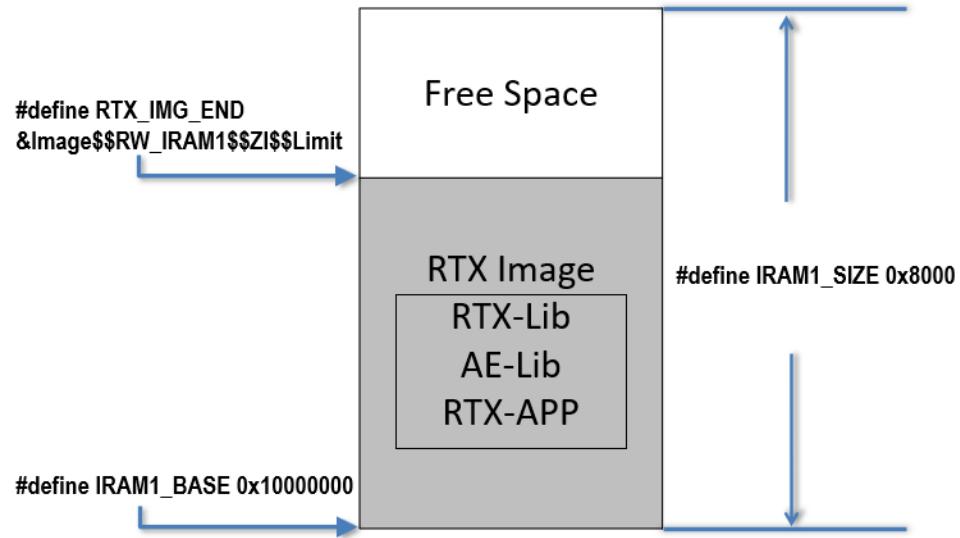


Figure 3.2: NXP LPC1768 IRAM1 Memory Execution View. The RTX Image has two libraries (RTX-Lib and AE-Lib) built into it together with the application (RTX-App) that uses the libraries.

The entire IRAM2 is a free space for the kernel to manage (see Figure 3.3).

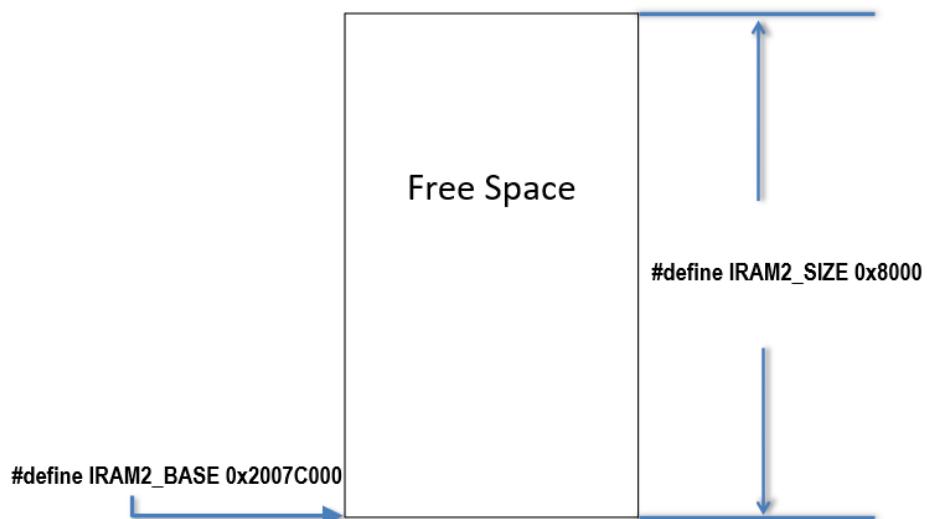


Figure 3.3: NXP LPC1768 IRAM2 Memory Execution View.

¹We build the kernel and the application that uses the kernel into one single image and refer it as the RTX Image.

Another important specification we need is the word size. The Cortex-M3 is a 32-bit processor. The word size is 32-bit. The `WORD_SIZE` macro is defined as 32 in the `lpc1768_mem.h` for this purpose.

3.4.2 Dynamic Memory Allocator

A memory allocator manages free spaces. We refer to a free space that resides in a continuous memory area as a memory pool. In this lab, we have two pools of real memory (i.e. physical memory)² for the allocator to manage. Each memory pool is a linear collection of contiguous bytes, where each byte has an address. The bytes are ordered from a starting address to an ending address, where the starting address is no greater than the ending address and the addresses are contiguous. The dynamic memory allocator reserves and frees variable-sized blocks of memory in an arbitrary order from the memory pools it manages. Each block is a contiguous chunk of memory (i.e. consecutive memory locations). An allocator manages the memory as a collection of variable-sized blocks. Its job is to keep track of allocated and free blocks.

The allocator provides the following functions to kernel application programs³:

- `mpool_t k_mpool_create(int algo, U32 start, U32 end);`
- `void *k_mpool_alloc(mpool_t mpid, size_t size);`
- `int *k_mpool_dealloc(mpool_t mpid, void *ptr);`
- `int k_mpool_dump(mpool_t mpid);`

The `k_mpool_create` function creates a memory pool. It initializes the data structures that the allocator uses to keep track of which parts of memory are in use (i.e. allocated) and which parts of memory are free in the memory pool. It returns a non-negative memory pool ID on success and -1 on failure. The `k_mpool_alloc` function is to reserve a block for an application from a memory pool. A block allocated by the application remains allocated until it is explicitly deallocated by the application. The `k_mpool_dealloc` function is to deallocate (i.e. free) a block an application releases from the memory pool. A deallocated block is free and remains free until it is explicitly allocated by the application. The `k_mpool_dump` function dumps address and sizes of free memory blocks in the memory pool to the debug terminal and returns the number of free memory blocks in the memory pool.

²The allocator we write can also be used to manage virtual memory.

³The term “application” is used in a general sense. Any program that uses the memory allocator to mange its memory is considered as an application. A kernel that uses the allocator to manage its memory is one example of an application.

3.4.3 Design and Implement Issues

An allocator needs to keep track of which blocks are free and which blocks are allocated by using some data structures. The data structures encode information to distinguish free blocks from allocated blocks and their boundaries. How do we encode the information and where do we store the encoded information? This brings us to the first design and implementation issue of a memory allocator, which is the block format design.

Issue #1: Block Format

Figure 3.4 shows one design of a free block. We use doubly linked list to keep track of them. The two pointers are only needed to keep track of blocks on the free list. Hence we can safely discard them once we remove the block from the free list⁴. Padding might be needed for reasons such as alignment requirement and allocator's strategy for coalescing et. al..

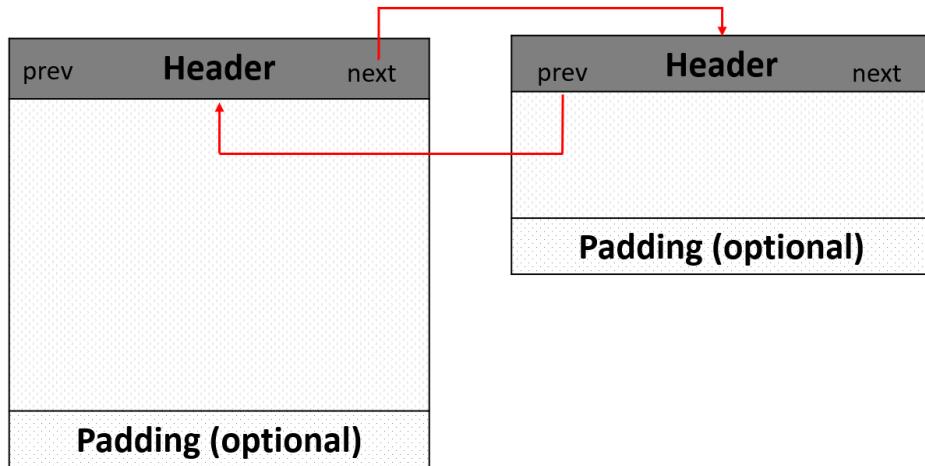


Figure 3.4: A Free Block Format

The header size determines the lower bound of the minimum block size that the allocator can support. Any size that is smaller than the header size is impossible since there is no way to keep track of free blocks. In this lab, we have defined `MIN_BLK_SIZE` macro in `common.h` as 32. This macro puts an upper bound of the header size in your data structure design. When you create the `k_mpool_create` function, this is one of the main design issues you need to consider.

⁴There are also designs that we need to encode more information in the header and some part of the information needs to be kept in the header after the memory block is removed from the free list.

Issue #2: Free List Organization

The allocator data structures need to keep track of free blocks. These free blocks are organized by using free list(s). How do we organize the free list(s) is the second design and implementation issue. A linked list, an array of linked lists, or a tree are just some possible organization mechanisms. In this lab we implement the binary buddy system. Assume the memory pool size is 2^m bytes, the idea of this method is to keep separate lists of available blocks each of the size 2^k bytes, where $0 \leq k \leq m$ [5]. The memory for free list(s) themselves can be statically allocated to simplify the implementation.

What information do we need to encode for allocated memory blocks? As you may notice that the `k_mpool_deallocate` function does not specify the size of the block to be deallocated. This implies the kernel needs to book keep this piece of information. For the binary buddy system, we can create a perfect binary bit tree and then compute the size. To simplify the implementation, we will statically allocate memory at compile time for creating this binary bit tree. Please check the lecture materials for details.

The space the data structures need is the overhead. There is a trade off to be made between space and speed. The more space your structures require, the less space is available for allocator to return to application programs. On the other hand, larger data structures may result in faster operations, if the data structures are efficient. When you create the `k_mpool_create` function, this is one of the main design issues you need to consider.

Issue #3 Placement And Splitting Policy

Once we have free list(s), we need a placement policy to determine how to search for an appropriate free block to place a newly allocated block. It answers the question that when there are multiple free blocks that are big enough, which one we should choose? After we place a newly allocated block in some free block, there might be some space left. The splitting policy determines what do we do with the remaining block. Do we make a new free block or do we pad the free space into the allocated block and hence create internal fragmentation? When you create the `k_mpool_alloc` function, these are main issues you need to consider.

Assume the memory pool the allocator manages is of size 2^m bytes, in the binary buddy system , when the requested size is S bytes, we want to find a free block of size of 2^k bytes, where $k = \lceil \log_2(S) \rceil$. If no 2^k byte free block is found, we try to find the smallest free block of a large enough size that has a size of 2^{k+i} bytes, where $i = 1, 2, \dots, m - k$. The found larger block keeps splitting itself in half until we get a free block with size of 2^k bytes. Every time we split a larger block into two halves, they become buddies. One buddy is for further splitting if it is too big (i.e. bigger than 2^k bytes) or we return it to the application if it is the right size. The other buddy goes onto the free list of its size.

Issue #4 Coalescing Policy

When a block is deallocated by the application program, what operations do we need to do about this block? Do we just put the block back to its free list and mark it as free? What if its adjacent blocks are free? Do we combine these blocks to make a bigger free block? If we do, how should we combine them and when do we combine them? All these are determined by the coalescing policy. In the buddy systems, the freed block can only be merged with its buddy block when the buddy is free. The coalescing continues to the next level of memory block size hierarchy until no more free buddy block is found. In this lab, we require immediate coalescing.

When you create the `k_mpool_dealloc` function, this is the main issue you need to consider.

3.4.4 Implementation Tips

Tip #1

What makes the buddy system useful practically is that a buddy's address can be easily computed given the address and size of the other buddy block. All buddies are aligned on a power-of-two boundary offset from the beginning of the memory pool. If we look at the address offset values from the beginning of the memory pool of two buddies, they differ exactly by one bit in binary number representation and the bit position is a function of the buddy block size (we trust you can easily figure out this function). Hence the “exclusive or” operation makes computing the buddy’s address a very easy and quick operation.

Tip #2

One common mistake that most likely will cause excessive amount of debugging time is to forget that pointer arithmetic operations are performed in the units that are the size of the objects that pointers point to. For example, assume I have the following code excerpt to increment a pointer `p` by one unit, the value of `p` will be `n` after `p++`, *not one*.

```
int *p = 0;
unsigned int n = sizeof(int);
p++;
```

Listing 3.1: Pointer Arithmetic Code Excerpt

If you want `p` to be incremented by one, then you need to cast `p` to a pointer that points to a byte size object. Two ways of doing this is as follows.

```

p = (void *) ((char *) p + 1); // This is one way of casting.
p = (int *) ((char *) p + 1); // This is another way of casting.

```

Listing 3.2: Pointer Increment by One using Casting

Note when you assign a variable to a pointer variable, you need to cast it to the proper data type. Either `void *` or the data type of the pointer is allowed. Checkout https://www.keil.com/support/man/docs/armcc/armcc_chr1359124216794.htm for additional data alignment constraints the arm compiler has.

3.4.5 Specifications of Functions

The specifications of each function to be implemented are described in this section.

Memory Pool Creation Function

NAME

`k_mpool_create` - create a memory pool and initialize the dynamic memory allocator data structures

SYNOPSIS

```

#include "k_rtx.h"

mpool_t k_mpool_create(int algo, U32 start, U32 end);

```

DESCRIPTION

The `k_mpool_create()` creates a memory pool and initializes the memory allocator with the data structures that the allocation algorithm uses. The input parameter `algo` specifies the memory allocation algorithm. The `start` and `end` are the starting and ending addresses of the pool of memory space that the allocator manages. The full list of memory allocation algorithms are as follows ⁵:

BUDDY

The binary buddy system memory allocation algorithm is used. If the memory space size is not a power of two, then the allocator finds the number N which is the largest power of two that is not greater than the memory pool size and only manages these N bytes from the starting address. The rest of the bytes will never be used.

The function initializes the memory allocator data structures used by the specified algorithm. Initially, there is only one free memory block. As the allocator

⁵BUDDY is the only supported algorithm in this lab.

serves the allocation and deallocation requests, the memory will be partitioned into allocated and free memory blocks. The allocator uses its data structures to track which parts of memory are allocated and which parts are free.

RETURN VALUE

The function returns a non-negative memory pool ID on success and -1 if an error occurred (`errno` is set). `MPID_IRAM1` and `MPID_IRAM2` are two reserved memory pool IDs (see `common.h`) for the two on-chip free spaces inside IRAM1 and IRAM2 respectively (see Figures 3.2 and 3.3).

ERRORS

`EINVAL` The `algo` is invalid.

`ENOMEM` There is not enough space to support the operation.

SEE ALSO

`k_mpool_alloc`, `k_mpool_dealloc`

Memory Allocation Function

NAME

`k_mpool_alloc` - allocate dynamic memory from a memory pool

SYNOPSIS

```
#include "k_rtx.h"

void *k_mpool_alloc(mpool_t mpid, size_t size);
```

DESCRIPTION

The `k_mpool_alloc()` allocates a block of memory from memory pool identified by `mpid` at least `size` bytes 8-byte aligned and returns a pointer to the allocated memory⁶. The allocated memory is not initialized (i.e. not set to zeros). If `size` is 0, then `k_mpool_alloc()` returns NULL.

The input parameter `size` is the number of bytes requested from the allocator. It may be of any size from one byte all the way up to the maximum value of a `size_t` data type. The allocator then returns the starting address of a block of memory in consecutive memory locations that is at least `size` bytes from the memory pool identified by `mpid`. The returned memory address should be 8-byte aligned (i.e. it is a multiple of eight). When the returned block of memory is bigger than the requested size, there will be additional space that is not asked by the caller. This space is the internal fragmentation (and the caller will not be told).

RETURN VALUE

⁶The `k_mpool_create()` needs to be invoked before calling `k_mpool_alloc()`.

The function returns a pointer to the allocated memory, or NULL an error occurred (`errno` is set). If the `size` is zero, NULL is also returned (`errno` is not set).

ERRORS

`EINVAL` The `mpid` is invalid.

`ENOMEM` There is not enough space to support the operation.

SEE ALSO

`k_mpool_create`, `k_mpool_dealloc`

Kernel Memory Deallocation Function

NAME

`k_mpool_dealloc` - Free a block of memory that was allocated from a memory pool

SYNOPSIS

```
#include "k_rtx.h"

int k_mpool_dealloc(mpool_t mpid, void *ptr);
```

DESCRIPTION

The `k_mpool_dealloc()` frees the memory space pointed to by `ptr`, which must have been returned by a previous call to `k_mpool_alloc()`. If `ptr` is NULL, no operation is performed. If the `mpid` is invalid, it returns an error (`errno` is set). If the previous call to `k_mpool_alloc()` did not have `mpid` as the input parameter, it returns an error (`errno` is set). If `k_mpool_dealloc()` has already been called before or there are any other unspecified situations, undefined behaviour occurs.

If the freed memory block is adjacent to other free memory blocks in the memory pool, it is merged with them immediately (i.e. immediate coalescence) according to the allocator algorithm. 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 0 on success, or -1 if error occurred (`errno` is set).

ERRORS

`EINVAL` The `mpid` is invalid.

`EFAULT` The `ptr` points to a memory location outside the memory pool identified by `mpid`.

SEE ALSO

`k_mpool_create`, `k_mpool_alloc`

Utility Function - Dump Free Memory Addresses

NAME

`k_mpool_dump` - Dump addresses and sizes of free memory blocks

SYNOPSIS

```
#include "k_rtx.h"

unsigned int k_mpool_dump(mpool_t mpid);
```

DESCRIPTION

This function outputs all addresses of free memory blocks and their sizes in the memory pool identified by `mpid`, one line for each memory block. The output address is the address of the header of the block. Each line starts with the address of a free block header address, followed by a colon, a space and then the size of the memory block. The size of a memory block in the output includes the header size. The output is ordered in the increasing order of the block size. The order to output blocks of the same size is unspecified (i.e. can be any order). The addresses and sizes are displayed by using hexadecimal number format. The last line in the output summarizes how many free memory block(s) are found in the system (see the EXAMPLE section for details).

RETURN

This function returns the number of free memory blocks in the memory pool identified by `mpid`. If the `mpid` is invalid, it returns 0.

EXAMPLE

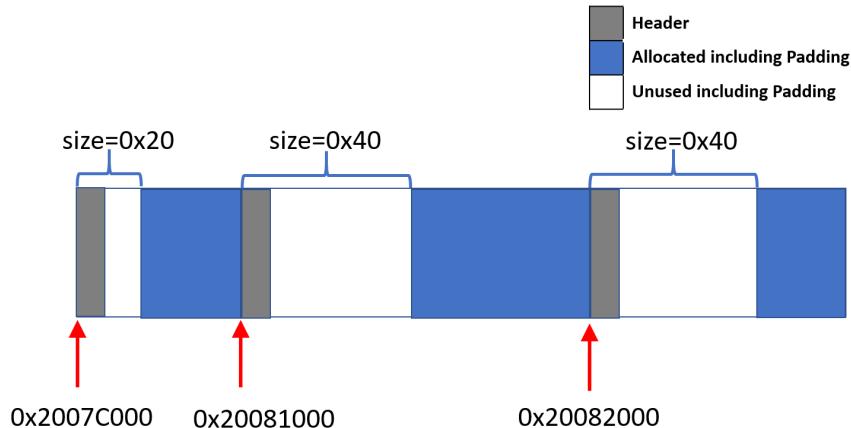


Figure 3.5: A memory map. Figure is not drawn to scale.

Assume there are only three free memory blocks at addresses 0x2007c000, 0x20081000 and 0x20082000 with sizes of 0x20, 0x40 and 0x40 each in the system (see Figure 3.5). There are two accepted outputs. One is as follows.

```
0x2007c100: 0x20
0x20081000: 0x40
0x20082000: 0x40
3 free memory block(s) found
```

The other one is as follows.

```
0x2007c100: 0x20
0x20082000: 0x40
0x20081000: 0x40
3 free memory block(s) found
```

Assume there are no free memory blocks in the system. The program will output the following line:

```
0 free memory block(s) found
```

SEE ALSO

`k_mpool_create`, `k_mpool_alloc`, `k_mpool_dealloc`

3.5 Source Code File Organization and Third-party Testing

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

Don'ts

- Do not change the locations or names of existing files or directories.
- Do not modify any of the header files in the `lab1\include` folder and its sub-folders except for `rtx_ext.h` and `common_ext.h`.
- Do not change the prototypes of existing functions in the `k_mem.h` and `k_init.h` files. You may change the implementation of those functions though.

- Do not include any new header files in the `main.c`.
- Do not include any new header files in the `ae.c` file.

Dos

- You are allowed to add new self-defined functions to `k_mem.[ch]`.
- You are also allowed to create new `.h` and `.c` files⁷.
- The newly created `.h` files are allowed to be included in the files under `kernel` directory.
- Any new files you add to the project can be put into either the `kernel` directory or other directories you will create under the `src` directory.

Note that the `main.c` calls the memory management functions you will implement. The `main.c` starts the third-party testing by calling `ae_init` and `ae_start` functions which the third-party testing software implements. The function prototypes of these two functions do not change. But the implementation of these two functions may change in real testing. Do not delete the lines in the `main.c` where these two functions are invoked. We will write more `ae_mem<n>.c` files with more complicated testing cases than the ones released in the starter code during the third-party testing..

3.5.1 Testing

In order to test your implementation, you need to write at least one test suites in the `ae_mem<suite id>_G<group id>.c` file. Each test suite contains minimum three test cases. To get some ideas, you could look into the sample test cases that are provided with the starter code. Your test cases should be different for the sample test cases. You also need to document the specification of your test cases in the report (see 3.6). There is no hard requirement on the exact testing scenarios. The rule of thumb is that the tests should convince yourself that your implementation is correct. For example, you may want to consider repeatedly allocating and then deallocating memory and make sure no extra memory appears or no memory gets lost. The sum of free memory and allocated memory (including internal fragmentation) should always be constant. Another aspect to consider is the external fragmentation. Allocate and de-allocate memory with different sizes and see how external fragmentation is affected. You will find the utility function `k_mpool_dump()` to be a useful tool.

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):

⁷For example, you may want to put your allocator's data structure functions or some helper functions in new files for better file organization.

```
Gid-TSN: START  
Gid-TSN: some output  
Gid-TSN: some output  
Gid-TSN: x/M tests PASSED  
Gid-TSN: y/M tests FAILED  
Gid-TSN: END
```

In the above example output, the “id” is the Group ID. The “N” is the test suite ID, and “M” is the total number of testing cases. For example, assume that you are in group G99 and you have 3 testing cases in total in test suite 1, if two of the testing cases passed and one of the testing cases failed, the final testing results should be output to the putty terminal as follows:

```
G99-TS1: START  
G99-TS1: some output  
G99-TS1: some output  
G99-TS1: some output  
G99-TS1: 2/3 tests PASSED  
G99-TS1: 1/3 tests FAILED  
G99-TS1: END
```

We have set up the starter code so that the serial window output in the simulator is saved in RTX_App\LOG\SIM\uart.log file. When you submit your project, you need to provide the expected output of your own testing suites and the naming convention of the expected output file is G<group id>-TS<suit id>.log. For example, the expected output of Group 99’s test suit 1 should be named as G99-TS1.log.

3.6 Lab Report

Write the following items in your report and name it lab1_report.pdf.

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

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

- coalescing free regions,
- reusing newly-freed blocks,

- not leaking memory (size of heap should not increase or decrease),
- utilizing memory with minimum overheads, and
- returning correct number of free memory blocks.

3.7 Deliverable

3.7.1 Pre-Lab Deliverables

- Go to https://git.uwaterloo.ca/users/sign_in and sign yourself in.

3.7.2 Post-Lab Deliverables

Create a folder name it “lab1-submission”. Inside this folder put your code, lab report and the expected output as shown in Figure 3.6. The README is optional, though is recommended. It is for briefly describing what is in the submission and any additional notes you have for the teaching team. Zip everything in the lab1-submission folder and save it as lab1-submission.zip. Submit it to the lab1 dropbox.

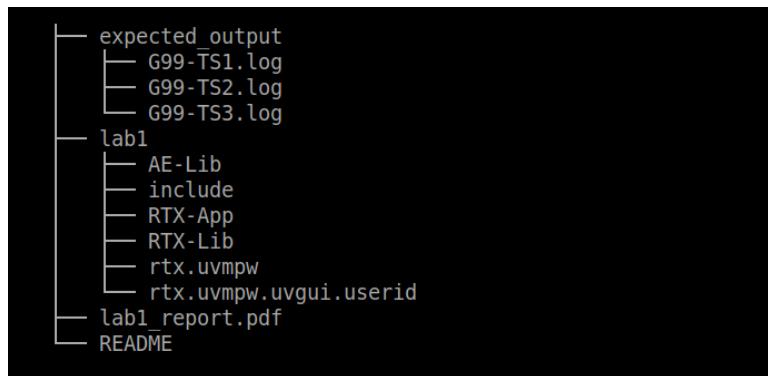


Figure 3.6: Lab1 Submission Directory Layout

3.8 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 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.

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

Table 3.1: Lab1 Marking Rubric

3.9 Errata

1. Page 13, Section [3.3](#), first bullet point, Section [8.9.2](#) changed to [8.9](#).
2. Page 20, Listing [3.2](#) second line of code changed `struct MCB *` to `int *`.
3. Page 20, added a footnote.

Chapter 4

Lab2 Task Management

4.1 Objective

In this lab, you will design and implement a preemptive multitasking kernel for MCB1700, a single Cortex-M3 processor board. The kernel provides user space Application Programming Interface (API) which is a set of system calls defined in `rtx.h`. You will first design and implement memory system calls by using the kernel memory allocator you did in lab1. Then you will design and implement system calls to manage tasks¹ in a multitasking kernel. You will also implement a utility system call to return task information. More specifically, you will learn:

- How to use SVC as a gateway to program a system call,
- How to design and implement a multitasking kernel,
- How to design and implement kernel support for scheduling tasks with different priorities, and
- How to design and implement kernel support for task preemption.

4.2 Starter Files

The starter file is on GitHub at http://github.com/yqh/ece350/tree/master/manual_code/lab2/. It contains a multi-project workspace profile which contains the following individual projects.

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

- AE-Lib/: The testing suite library project which contains some sample tasks written by the lab teaching staff.
- RTX-App/: The RTX kernel uVision application project template for your lab2. It is a rudimentary kernel application project linked with AE-Lib library. It supports context switching between three tasks whose user stacks are statically allocated. You need to change the code so that the user stacks of tasks will be dynamically allocated by using the memory allocator you created in lab1. The main files you will need to work on are `k_task.[ch]`.

4.3 Pre-lab Preparation

- Read “Keil MCB1700 Hardware Environment” in Chapter [10](#)
- Read “SVC Programming: Writing an RTX API Function” in Section [9.5](#).
- Review processor operation mode in [10.2.2](#), processor’s two stacks in [10.2.3](#) and exception return privilege level and stack setting in [10.4.3](#).
- Read C and assembly programming in Section [9.4](#).
- Review the lecture materials on creating thread, context switching between threads, and yielding the processor.
- Execute the RTX-App project in the starter code on the simulator and on the board. Observe the output differences from these two environments.
- Work through the context switching code and understand what they do and how they work.

4.4 Lab Project Part A - Memory System Calls

Project Part A is to practice using SVC as a gate way to create system calls for thread mode tasks to use the memory allocation service provided by the kernel.

4.4.1 Overview

Tasks in the user space and the operating system in kernel space both need memory during runtime. In this lab, we will let the memory pool identified by `MPID_IRAM1` to serve memory allocation requests from the tasks and the memory pool identified by `MPID_IRAM2` to serve memory allocation requests from the kernel itself which include allocating user stack spaces on behalf of the tasks. On ARM Cortex-M3, the processor has two operation modes: thread mode and handler mode. Tasks

operate in thread mode and the operating system kernel operates in handler mode. The kernel directly invokes the kernel memory allocator. A task needs to obtain operating system service such as memory allocation by trapping into the kernel. On Cortex-M3, the SVC instruction causes an exception and switches the processor from thread mode to handler mode.

You will use the SVC mechanism to program the following three memory system calls for thread mode tasks:

```
#include "rtx.h"
void *mem_alloc(size_t size);
int *mem_dealloc(void *ptr);
int mem_dump(void);
```

The `mem_alloc` function is to reserve a free memory block for a task from memory pool identified by `MPID_IRAM1`. The `mem_dealloc` function is to deallocate (i.e. free) a block a user task releases. The `mem_dump` function dumps address and sizes of free memory blocks in the memory pool identified by `MPID_IRAM1` to the debug terminal and returns the number of free memory blocks in the memory pool.

4.4.2 Memory Map

To simplify the implementation, `RAM1_START` defined in `lpc1768_mem.h` is a power of two. This means there will be some free space between the end of the RTX image and the `RAM1_START` left as not managed by the kernel (see Figure 4.1).

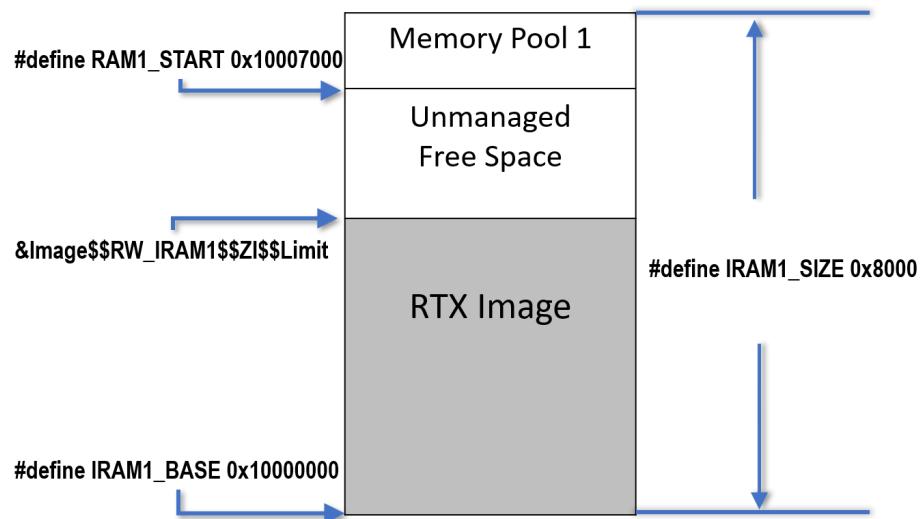


Figure 4.1: Lab2 NXP LPC1768 IRAM1 Memory Execution View. There is a block of free space that is not managed.

This memory set up also puts a limit on the maximum image size in execution

view. Particularly this requires us to update the default target memory map settings for both SIM target and RAM targets so that the image does not use the last 4KiB in the IRAM1.

A simple image generated by arm compiler consists of the following sections and they are:

- Read-only (RO) section which contains Code + RO-data.
- Read-write (RW) section which is RW-data.
- Zero-initialized (ZI) section which is ZI-data.

The build output reports how big each section is (see Figures 4.2 and 4.3). The RO section goes to the Read/Only Memory Areas in the target option memory setting. The RW and ZI sections go to the Read/Write Memory Areas in the target option memory setting. Figures 4.2 and 4.3 show the starter code memory map configuration. Especially for the RAM target, you may need to adjust the Read/Only Memory Areas and Read/Write Memory Areas settings based on how big each section of your image is.

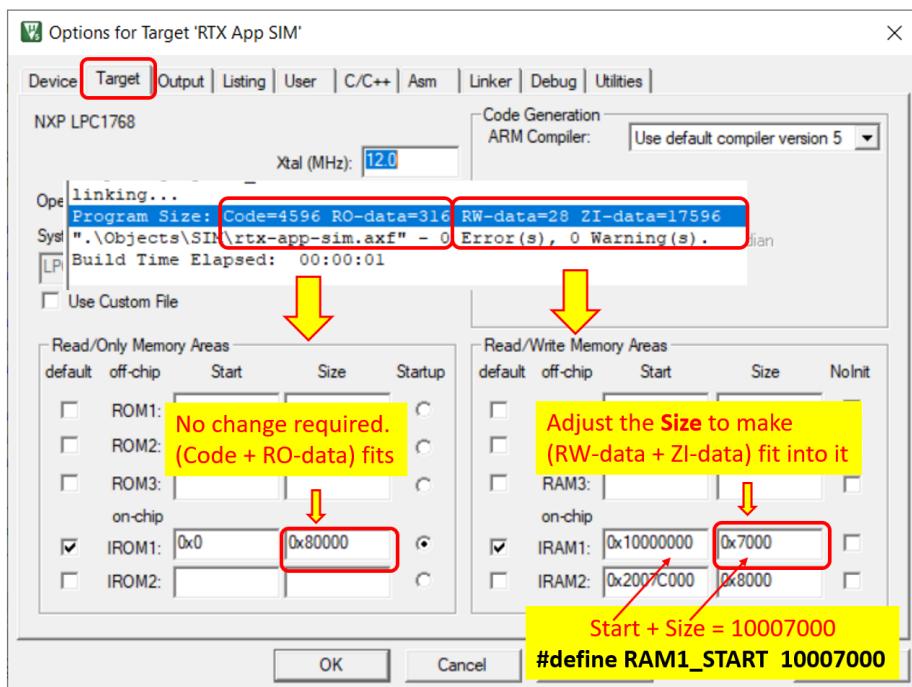


Figure 4.2: Lab2 NXP LPC1768 IRAM1 SIM Target Memory Map Configuration.

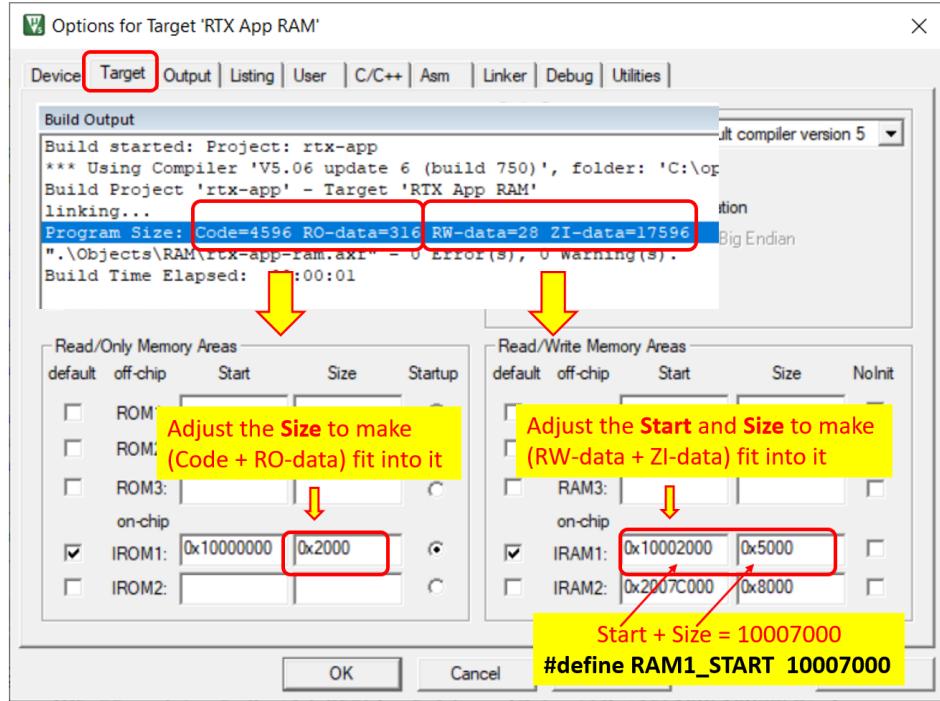


Figure 4.3: Lab2 In-memory Execution RAM Target Memory Map Configuration.

4.4.3 Specifications of Functions

The starter code already implemented the trap table of memory system calls so that they will use the kernel memory allocator to operate on memory pool identified by MP_ID_IRAM1. What you need to do is to write testing tasks to verify the system calls behaviour follows the following specifications. You will re-use your lab1 testing code to do this. The difference is that now you are calling the memory allocator by using system calls in tasks.

Memory Allocation Function

NAME

`mem_alloc` - allocate dynamic memory

SYNOPSIS

```
#include "rtx.h"

void mem_alloc(size_t size);
```

DESCRIPTION

The `mem_alloc()` system call allocates a block of memory at least `size` bytes 8-byte aligned and returns a pointer to the allocated memory ². Internally, the kernel's memory pool identified by `MPID_IRAM1` is used by the kernel memory allocator to serve the request. The allocated memory is not initialized (i.e. not set to zeros). If `size` is 0, then `mem_alloc()` returns `NULL`.

The input parameter `size` is the number of bytes requested from the operating system. It may be of any size from one byte all the way up to the maximum value of a `size_t` data type. The operating system then returns the starting address of a block of memory in consecutive memory locations that is at least `size` bytes from the memory pool identified by `MPID_IRAM1`. The returned memory address should be 8-byte aligned (i.e. it is a multiple of eight). When the returned block of memory is bigger than the requested size, there will be additional space that is not asked by the caller. This space is the internal fragmentation (and the caller will not be told).

RETURN VALUE

The function returns a pointer to the allocated memory, or `NULL` an error occurred (`errno` is set). If the `size` is zero, `NULL` is also returned (`errno` is not set).

ERRORS

`ENOMEM` There is not enough space to support the operation.

SEE ALSO

`mem_dealloc`

Memory Deallocation Function

NAME

`mem_dealloc` - Free dynamic memory

SYNOPSIS

```
#include "rtx.h"

int 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()`. If `ptr` is `NULL`, no operation is performed. If `mem_dealloc(ptr)` has already been

²The `k_mpool_create()` needs to be invoked to initialize the proper memory pool by the kernel before calling `mem_alloc()`.

called before or there are any other unspecified situations, undefined behaviour occurs.

RETURN VALUE

This function returns 0 on success, or -1 if error occurred (`errno` is set).

ERRORS

EFAULT The `ptr` points to a memory location outside the memory pool that the kernel is used for allocating memory for user tasks.

SEE ALSO

`mem_alloc`

Utility Function - Dump Free Memory Addresses

NAME

`mem_dump` - Dump addresses and sizes of free memory blocks available to user tasks

SYNOPSIS

```
#include "rtx.h"  
  
int mem_dump(void);
```

DESCRIPTION

This function outputs all addresses of free memory blocks and their sizes that the system has available for user tasks. The output to the debug terminal is the same as the kernel directly calling `k_mpool_dump(MPID_IRAM1)`.

RETURN

This function returns the number of free memory blocks for user tasks.

SEE ALSO

`k_mpool_dump`, `mem_alloc`, `mem_dealloc`

4.5 Lab Project Part B - Task Management System Calls

Part B is to design and implement a multitasking kernel and a preemptive priority scheduler the kernel uses to schedule tasks.

4.5.1 Overview

Tasks are kernel objects created to multiplex the CPU time. In this lab, all our tasks are operated in thread mode. There are two access levels, one is privileged and the other is unprivileged. Every task has two stacks: one kernel stack that is allocated statically at compile time and one user stack which the kernel allocates on behalf of the task from the memory pool identified by MP_ID_IRAM2. The maximum number of (privileged and unprivileged) tasks that could co-exist in the system is determined at compile time by MAX_TASKS macro in common.h file. You will implement a pre-emptive priority scheduler for the kernel to decide which task should use the CPU time. The kernel provides the following system calls to support multitasking.

```
#include "rtx.h"

int rtx_init(RTX_SYS_INFO *sys_info, TASK_INIT *tasks, int num_tasks);
int tsk_create(task_t *task, void (*task_entry)(void),
               U8 prio, U32 stack_size);
void tsk_exit(void);
int tsk_yield(void);
int tsk_set_prio(task_t task_id, U8 prio);
int tsk_get(task_t task_id, RTX_TASK_INFO *buffer);
```

The rtx_init() system call initializes the kernel with the memory and scheduler initialization parameters and a list of tasks that should be created at kernel startup time. The tsk_create() and tsk_exit() system calls create and terminate tasks at runtime. The tsk_yield() system call yields the processor. The tsk_set_prio() system call adds support for task priority management. The tsk_get() system call returns task information.

The system has a built-in null task (see Section 4.5.7). It comes with the starter code. You will create a number of testing tasks to verify your design and implementation of the above system calls.

4.5.2 Data Structures and Algorithm

You need to design data structures to represent a task and keep track of different tasks in the system. A Task Control Block (TCB) is a structure that represents a task. The provided starter code has put a very simple TCB data structure there. You need to modify and add extra fields to this data structure to complete the lab.

The kernel uses queue data structure to keep track of tasks at the same priority level. For each priority, it has a separate queue. A linked list data structure can be used to implement queue. You can then use an array of queues, where the array index can be easily mapped to a priority level.

You can design and implement generic linked list data structure so that it can

link any type of data items such as memory blocks in the first lab, TCBs in this lab. There are different ways to implement it. One easy way is to put your linked list node structure as the first fields in each data item you want to link and then use C casting to cast the data item to the linked list node structure.

Please note that the performance of the scheduler (see section 4.5.3) is one of the most important aspects of a real-time operating system. In future labs, you will measure and optimize your scheduler. It might be hard to change your code later on. Therefore, you might want to spend some time now thinking about efficient data structures and algorithms to add, remove, and sort ready tasks in the ready queue.

TODO #1

The `rtx_init()` is to set up kernel data structures during RTX initialization time. You need to add your own task management data structures' initialization operations in the starter code.

4.5.3 Scheduler

At the heart of the kernel is the scheduler which makes the decision on which task should use the processor. In this lab, we are going to design and implement a pre-emptive priority scheduler. The scheduler has a number of ready queues with different priorities. Each task is assigned a priority. The scheduler keeps track of ready tasks by putting them on ready queues for their priorities. The scheduler always selects the highest priority ready task. Within the same priority level, the scheduler follows First Come First Serve (FCFS) scheduling policy. That is among same-priority tasks, the earlier a task that is added to the ready queue, the higher its priority. A privileged task can change the priority of other tasks including itself. A unprivileged task can only change the priority of other unprivileged tasks including itself. At any moment, if a higher priority task (compared with the current running task's priority) becomes ready, the scheduler will *preempt* the current running task by putting the current running task into READY state and select the higher priority task to use the processor.

For this lab, we only have three task states and they are READY, RUNNING and DORMANT (see Figure 4.4) defined in the `common.h` file. When a task is just created, it is in READY state. The scheduler selects the highest priority task to run and this brings the selected task to RUNNING state. A RUNNING task can voluntarily give up its turn to use the processor by calling the `tsk_yield()` and becomes READY again. The task is put to the end of its own ready queue. The scheduler will select the highest priority task. The selected task could be the original task that invoked the `tsk_yield()` if it is the highest priority ready task in the system. A task can also change from RUNNING to READY or vice versa due to a running task changes a task's priority or a new task gets created at runtime. When a RUNNING task changes to READY state, if the change is caused by voluntary relinquishes the pro-

cessor by itself (i.e. calling `tsk_yield()`), the task is put to the back of its ready queue; if the change is caused by preemption (i.e. involuntarily loss of the processor usage), the task is added to the front of its ready queue. Preemption due to `tsk_set_prio` and `tsk_create` system calls are involuntary. When a task priority changes and this change causes the kernel to move the task to a new ready queue, the task is added to the end of the new ready queue. The `tsk_exit()` brings a RUNNING task to DORMANT state.

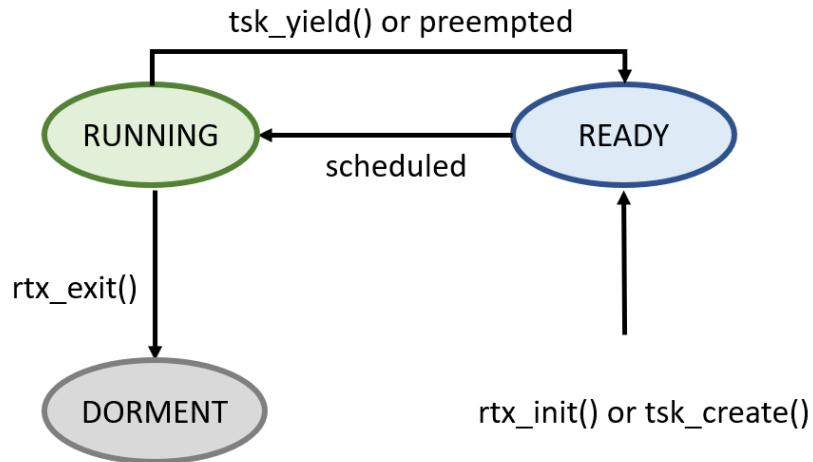


Figure 4.4: Lab2 Three State Transition Diagram

Scheduling Examples

We give some examples to further illustrate the scheduling algorithm.

1. Task A is running, it creates a new task, B.
 - If the priority of B is higher than that of A, then B preempts A and starts running immediately. The preempted task, A, is added to the front of the ready queue. In other words, when A goes back to the ready queue, it maintains its previous position among tasks with the same priority (i.e., A will be sorted as the first task among all tasks with the same priority as A).
 - If the priority of B is not higher than that of A, then B will be added to the back of the ready queue (i.e., it will be sorted as the last task among all tasks with the same priority as B).
2. Task C is running, it changes the priority of a *ready* task, D.
 - If the new priority of D is higher than that of C, then D preempts C and runs immediately (note that D has to be in the ready state to be able to preempt C). The preempted task, C, is added to the front of the ready queue (i.e., it will be sorted as the first task among all tasks with the same priority as C).

- If the new priority of D is the same as the old priority of D, no action is performed. C will continue running and D maintains its original position in its ready queue.
 - If the new priority of D is not the same as its old priority and is not higher than that of C, then D will be moved to the back of its new ready queue (i.e., it will be sorted as the last task among all tasks with the same priority as D).
3. If C changes its own priority, it should continue running unless the new priority is lower than its original priority, in which case C is added to the back of the ready queue associated with C's new priority (i.e., it will be sorted as the last task among all tasks with the same priority).
 4. Task E is running and calls `tsk_yield`.
 - If there are other ready tasks that are with the same priority as E, then E is put to the back of its ready queue and the task at the front of the highest priority ready queue is selected to run.
 - If there are no other ready tasks that are with the same priority as E, then E's ready queue is empty. By putting E to the back of its ready queue, E would be the first task in its ready queue and E will be selected by the scheduler again and hence continues running. For implementation variations, one may decide to skip putting E to the ready queue and then removing it immediately again. One may just let E directly continue running without doing the two queue operations in this case.

TODO #2

The starter code does not implement preemptive priority scheduler. You need to implement this scheduler. You can either create a new scheduler function inside the `k_task.c` file (if you do not like the existing scheduler function prototype) or modify the existing `scheduler()` function in the file. If you decide to create a new scheduler function prototype, then you need to modify the starter code accordingly to invoke your newly created scheduler function.

4.5.4 Context Switching

When a scheduler selects a different task than the current running task to use the processor, the kernel needs to save the CPU context of the current running task and restore the CPU context of the previous task. This is referred to as context switching. Aside from the context saved in the task control block data structure, majority of the CPU register context information is saved on the task's stacks. The key step of performing the context switching is to switch the stacks (both kernel and user) of the two tasks. The starter code has implemented the context switching mechanism.

Cortex-M3 has two banked stack pointers. One is Process Stack Pointer (PSP) and the other is Main Stack Pointer (MSP). When processor is in thread mode, it can use PSP or MSP. We configure the system to use PSP in thread mode in this lab. When processor is in handler mode, MSP is used. This is decided by hardware and there is no other choice. At any moment, only one of the banked stack pointers is active.

In the starter code, a thread mode task uses the PSP, which points to the task's user stack. When system call happens, the hardware creates an exception stack frame which contains xPSR, PC, LR, R12 and R0-R3. These eight registers are saved by the hardware on PSP.

The processor then automatically switches to use MSP upon entering handler mode. The kernel uses the kernel stack for its own computation. When kernel needs to do a task switch, it invokes the `k_tsk_switch` function which saves the CONTROL, PSP, LR and R4-R12 on the kernel stack. The CONTROL needs to be saved so that after we switch to a new task, it will resume to the correct privilege level when exiting from the exception handler. The PSP is saved for efficient implementation of user stack switching. The starter code follows the Arm Architecture Procedure Call Standard (AAPCS) to save R4-R11 and LR, which are callee-saved registers. R12 is saved to maintain stack 8-byte alignment requirement by the AAPCS.

After saving all the registers on the kernel stack, the starter code saves the updated kernel stack pointer in the TCB data structure of the current running task (i.e. the task that is going to be switched out). Then it loads the newly selected task's kernel stack pointer from its TCB and sets the MSP to point to the newly selected task's kernel stack. This stack switch completes the context switching of the two tasks' kernel stacks. Since we save the task's user stack pointer on the task's kernel stack, switching the kernel stack also makes us switching the user stack when we pop off the user stack from the kernel stack upon exiting the exception handler. Once the kernel exits from the exception handler, the newly selected task resumes its execution.

Note in this lab, the SVC exception is configured as the highest interrupt priority, hence it cannot be further interrupted by other external interrupts such as UART or Timer before it finishes processing the exception. The SVC_Handler can still be interrupted by higher priority non-programmable exceptions (i.e. Reset, NMI, Fault exceptions et. al.). To simplify the implementation, we do not require the kernel to handle these exceptions. We accept that these exceptions are catastrophes that will crash the kernel.

TODO #3

In this lab, each task has two separate stacks: a user stack and a kernel stack. The user stack size is decided at runtime and should be dynamically allocated by the kernel on behalf of the task. The kernel stack size is specified at compiled time by `KERN_STACK_SIZE` macro in `common.h` file. The kernel stack is statically allocated at compile time.

The starter code statically allocates fixed-size user stacks (i.e. it ignores the user input of `u_stack_size` in the `TASK_INIT` structure) for three tasks. You need to change the starter code so that the user stacks are allocated dynamically during runtime according to the specified user stack sizes and the system can support `MAX_TASKS` (defined in `common.h`) number of tasks. The kernel should allocate user stacks from the memory pool identified by `MPID_1RAM2`. After you get the user stack allocated by using the dynamic memory allocator you write, then you should remove the statically allocated three user stacks (i.e. the big global array which is 1.5 KiB) from the starter code so that you will have more space for your RTX. Otherwise you will quickly run out of space to compile your project.

You will also need to modify the TCB data structure. This means you also need to modify the context switching related functions accordingly since TCB is the main data structure used for context switching.

You need to study the starter code to understand which registers are saved at which positions on stacks so that you can complete lab.

4.5.5 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 MAX_TASKS          0x10    /* maximum number of tasks          */
#define KERN_STACK_SIZE     0x300   /* fixed kernel stack size in bytes */
#define PROC_STACK_SIZE     0x200   /* minimum user stack size in bytes */
#define TID_NULL            0x0     /* reserved Task ID for the null task */

/* Task Priority. The bigger the number is, the lower the priority is*/
#define HIGH                0x80
#define MEDIUM              0x81
#define LOW                 0x82
#define LOWEST              0x83
#define PRIO_NULL            0xff   /* hidden priority for null task */

/* Main Scheduling Algorithms */
#define DEFAULT      0           /* preemptive priority scheduler */
                           /*      /* FCFS within each priority */

/* task state macro */
#define DORMANT      0           /* terminated task state */
#define READY        1           /* A ready to run task */
#define RUNNING      2           /* Executing
```

An important data structure is the `task_init` (see Listing 4.1). This structure is

used to launch initial tasks during RTX initialization (see `k_rtx_init` function in `k_task.c` file). The deprecated fields are not used in this lab.

```
typedef struct task_init {
    void    (*ptask)();      /**< task entry address */ 
    U32     u_stack_size;   /**< user stack space in bytes */ 
    task_t   tid;           /**< task ID, output parameter, deprecated */ 
    U8      prio;           /**< execution priority */ 
    U8      priv;           /**< = 0 unprivileged, =1 privileged */ 
} TASK_INIT;
```

Listing 4.1: Task Initialization Data Structure

Another important data structure is the `rtx_task_info` (see Listing 4.2). This structure is used to retrieve task information from kernel to and return it to the calling task. The kernel obtain the information by doing computation on the TCB data structure data and some runtime data the kernel have access to. You want to design your TCB data structure in such a way so that you can compute the information encoded in this data structure easily.

```
typedef struct rtx_task_info {
    void    (*ptask)();      /**< task entry address */ 
    U32     k_stack_size;   /**< kernel stack size in bytes */ 
    U32     k_sp;           /**< top of kernel stack */ 
    U32     k_sp_base;      /**< kernel stack base (high addr.) */ 
    U32     u_stack_size;   /**< user stack size in bytes */ 
    U32     u_sp;           /**< top of user stack */ 
    U32     u_sp_base;      /**< user stack base addr. (high addr.) */ 
    task_t   tid;           /**< task id, output param */ 
    U8      prio;           /**< execution priority */ 
    U8      priv;           /**< = 0 unprivileged, =1 privileged */ 
    U8      state;          /**< task state */ 
} RTX_TASK_INFO;
```

Listing 4.2: RTX Task Information Data Structure

4.5.6 Specifications of Functions

The starter code already implemented the trap table of task management system calls. Each system call has its kernel function counterpart. You need to complete the kernel counterparts of the system calls. Specifications of system calls are described below. Their kernel counterparts have the same interface and semantics.

RTX Initialization Function

NAME

`rtx_init` - initialize the kernel

SYNOPSIS

```
#include "rtx.h"

int rtx_init(RTX_SYS_INFO *sys_info, TASK_INIT *tasks, int num_tasks)
;
```

DESCRIPTION

The `rtx_init` system call initializes the kernel. It then creates `num_tasks` tasks to be executed and also a null task, a built-in task. The `sys_info` argument specifies RTX memory allocator and scheduler algorithms.

```
typedef struct rtx_sys_info {
    int mem_algo; /*< memory allocator algorithm */
    int sched;    /*< scheduling algorithm */
} RTX_SYS_INFO;
```

The `mem_algo` field in the structure specifies the memory allocator algorithm. It can only be set to `BUDDY`, which is the binary buddy algorithm. The `sched` field in the structure specifies the scheduling policy. In this lab, it can only be set to `DEFAULT`, which is preemptive priority scheduling policy. The `tasks` argument points to a memory location of an array of `num_tasks` `TASK_INIT` elements. The maximum number of tasks that can co-exist in the kernel including the null task is determined 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 tasks have the same size that is determined by the `KERN_STACK_SIZE` macro defined in `common.h`.

RETURN VALUE

The `rtx_init` function does not return on success. It returns `-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 (`errno` is set).

ERRORS

`EINVAL` The `sys_info` contains invalid parameter value(s).

`ENOMEM` There is not enough memory to support the operation.

SEE ALSO

`k_mpool_create`, `k_mpool_alloc`

Task Creation Function

The kernel has one primitive to create a unprivileged task at runtime. The system call traps into the kernel and sets up the TCB data structure for the new task. You are responsible for designing and implementing the TCB 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. Each task is uniquely identified by a task ID. The task ID is an integer value of $0, 1, 2, \dots, (\text{MAX_TASKS} - 1)$, where `MAX_TASKS` (a macro defined in `common.h`) is the maximum number of tasks (including the null task) the kernel supports. The task ID 0 (`TID_NULL` macro defined in `common.h`) is reserved for the null task (see [4.5.7](#)).

NAME

`tsk_create` - create an unprivileged task

SYNOPSIS

```
#include "rtx.h"

int tsk_create(task_t *task, void (*task_entry) (void),
               U8 prio, U32 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 defined in the `common.h`. The `stack_size` argument specifies the minimum user stack size in bytes. The kernel may decide to allocate more than the specified size, but not less. When `stack_size` is less than `PROC_STACK_SIZE` defined in `common.h`, the kernel will allocate `PROC_STACK_SIZE` for the task. The kernel is responsible for allocating the space from its own kernel memory pool identified by `MPID_IRAM2` for the user 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 (`errno` is set).

ERROR

`ENOMEM` There is not enough memory to support the operation.

`EAGAIN` The system has reached maximum number of tasks.

EINVAL The prio value is not valid.

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_exit()` system call stops and deletes the currently running task. Once a task is terminated, its state becomes DORMANT. Its TCB data structure and its kernel stack still exist in the system. The user stack of the task is deallocated by the kernel. When a task terminates, the kernel needs to make new scheduling decision to select another task to run.

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 voluntarily relinquish the CPU. The task is moved to the end of its 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. The function fails when the RTX encounters some unexpected internal error (`errno` is not set).

Task Priority Function

The RTX scheduling policy is priority based and is preemptive. You need to design and implement a primitive to set the priority of a 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 unprivileged task (including itself). An unprivileged task cannot change the priority of a privileged task. A privileged task may change the priority of any privileged or unprivileged 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. Preemption happens when calling this function results in another ready task having higher priority than the current running task.

RETURN VALUE

The function returns 0 on success and -1 on failure (`errno` is set).

ERROR

EINVAL Invalid task ID or an invalid priority level.

EPERM Operation is not permitted.

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 of a task identified by `task_id` and stores it in the buffer pointed by `buffer` before returning. The buffer is a `rtx_task_info` structure defined in `common.h` (see Section 4.5.5).

RETURN VALUE

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

ERROR

`EINVAL` The `task_id` is invalid.

`EFAULT` The `buffer` is a null pointer.

Note if `task_id` is the calling task's ID, we need to obtain stack pointer information by directly reading it from the stack pointer registers since the stack pointer data in the TCB only get updated when the task is switched out, hence contains outdated stack pointer data when the task is running. You can use the following CMSIS intrinsic functions to read MSP and PSP registers. More CMSIS intrinsic functions are listed in Table 9.3.

```
uint32_t __get_MSP(void);
uint32_t __get_PSP(void);
```

4.5.7 Required Tasks

The Null Task

The kernel has to run a task at any given time. If there are no ready tasks created by user, then kernel will run a unprivileged task named null task, which is created at kernel initialization 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 `TID_NULL` is reserved for the null task. When there is no other tasks that the kernel can execute, 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
```

The starter code has implemented this pseudo-code. 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.

Testing Tasks

In order to test your implementation of the required functions, write an application that uses your kernel primitives. The provided `ae_tasks<TS_id>_G<id>. [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.

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. 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_dump()` 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 in Section [3.5.1](#).

4.6 Source Code File Organization and Third-party Testing

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

Don’ts

- Do not change the locations or names of existing files or directories.
- Do not modify any of the header files in the `lab2\include` folder and its sub-folders except for `rtx_ext.h` and `common_ext.h`.

- Do not change the prototypes of existing functions appeared in the `rtx.h` and `common.h` files. You may change the implementation of those functions though.
- Do not include any new header files in the `main.c`.
- Do not include any new header files in the `ae.c` file.

Dos

- You are allowed to add new self-defined functions to `k_mem[ch]` and `k_task[ch]`.
- You are also allowed to create new `.h` and `.c` files³.
- The newly created `.h` files are allowed to be included in the files under `kernel` directory except for `main.c`.
- Any new files you add to the project can be put into either the `kernel` directory or other directories you will create under the `src` directory.
- For any functions that you are not allowed to change their prototypes, you are allowed to change their implementations.

Note that the `main.c` starts the RTX. The `main.c` starts the third-party testing by calling `ae_init` function and this function will eventually call the `set_ae_tasks` which the third-party testing software implements. The function prototypes of these functions do not change. But the implementation of the function may change in real testing. Do not delete the lines in the `main.c` where `ae_init` function is invoked. We will write more `ae_tasks<n>.c` files with more complicated testing cases than the ones released in the starter code during the third-party testing..

4.7 Report

Write the following items in your report and name it `lab2_report.pdf`.

- Descriptions of the data structures and algorithms used to implement all the functions listed in Section 4.5.6.
- Testing scenario descriptions

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

- creating and terminate tasks,

³For example, you may want to put your task management data structure functions or some helper functions in new files for better file organization.

- scheduling tasks with the same priorities (calling `tsk_yield` and `tsk_set_prio`),
- scheduling tasks with different priorities (calling `tsk_yield` and `tsk_set_prio`), and
- getting correct information on tasks.

4.8 Deliverables

4.8.1 Pre-Lab Deliverables

None.

4.8.2 Post-Lab Deliverables

You should structure your lab2-submission sub-directory in the GitLab repository so that it looks like the one shown in Figure 4.5. We no longer need to RTX-Lib sub directory. You may remove it from your git repository. Keeping it there does not affect anything either.

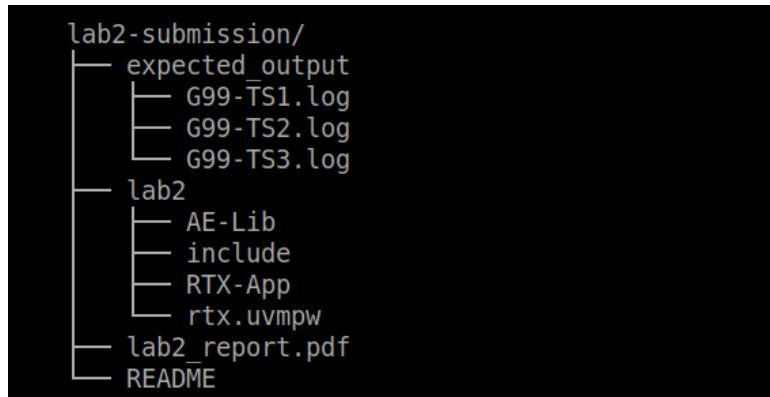


Figure 4.5: Lab2 Submission Directory Layout

The lab project and the report should be committed and pushed to the GitLab group project repository for submission. Put the GitLab commit hash string you would like to use for submission in a file and name it `lab2-submission.txt` for drop box submission. Use the `git log` command to list the commit history and the hash strings (see Figure 4.6).

Commit Hash String



```

commit ae76e4755ac329f3ea1ef79168d05a86da6e85ea
Author: John Smith <email address>
Date:   Sat May 29 10:57:25 2021 -0400

    cleaning up code

commit 1beb5b222dbdb4355807939148e8280796671bb5
Author: John Smith <email address>
Date:   Sat May 29 00:33:35 2021 -0400

```

Figure 4.6: Lab2 Git Commit History

4.9 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 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 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 4.1: Lab2 Marking Rubric

4.10 Errata

1. Page 35, SYNOPSIS section, `unsigned int mpool_dump` corrected to `int mem_dump`.
2. Page 38, Point 3, reworded C's behaviour when it is changing its own priority.

3. Page 43, RETURN VALUE section, the second sentence “It returns 1 on failure” corrected to “It returns -1 on failure”.
4. Page 46, Task Priority Function section, added “EPERM Operation is not permitted.” in the ERROR sub section.
5. Page 46, The DESCRIPTION section, The last sentence is reworded to clarify the preemption situation.

Chapter 5

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.

5.1 Starter Files

The starter file is on GitHub at https://github.com/yqh/ece350/tree/master/manual_code/lab3. It contains three uVision projects, an include directory which contains header files and a multi-project workspace profile for lab3 template code.

- `include/`: It contains header files that are needed for each individual uVision projects in the directory. Compared with lab2 starter code `include` directory contents, `bsp/LPC1768/uart_irq.h` file is a newly added file. The `rtx.h`, `common.h` and `rtx_errno.h` have been updated with lab3 related code.
- `UART0_IRQ/`: This directory contains a stand-alone uVision application to show how to echo a key press. It uses the board UART0 to perform serial communication with the host computer by interrupts. The keyboard is connected to the host computer. The host computer connects to the board COM0 through an USB-to-Serial adapter. The COM0 port connects to the board UART0 (See Figure 5.1). The application captures the keyboard input character transmitted by the host computer. It then uses interrupt to transmit a string from the

UART0 to the host computer where the string contains the keyboard character the board UART0 received.

- AE-Lib/: It contains the testing suite library project which contains some sample tasks written by the lab teaching staff.
- RTX-App/: It contains the RTX kernel uVision application project template for your lab3. It is a rudimentary kernel application project linked with AE-Lib library. It supports both UART0 interrupt and system call triggered context switching between three tasks. A context switch happens when “s” key is pressed on the keyboard or when a task relinquishes the CPU by calling `tsk_yield()`. Note the project does not echo the key input to UART0 terminal. With `DEBUG_0` defined, you will be able to see where in the interrupt handler the key press is captured. Compared with lab2 starter code, `the_k_task.[ch]` have been updated with lab3 template code. The following files are new.
 - `kernel/k_msg.[ch]`
 - `bsp/LPC1768/uart_irq.c`
 - `tasks/cdisp_task.c`
 - `tasks/kcd_task.c`
- `rtx.uvmp`: This is lab 3 multi-project workspace profile which contains the two projects under AE-Lib and RTX-App.

5.2 Pre-lab Preparation

- Refresh your memory on inter-process communications (ECE 252).
- Read Chapter 14 of [4]
- Run the starter projects on the simulator and the real hardware.
- Work through the `UART0_IRQ` code and understand what it does and how it works.

5.3 Lab Project

This lab is to add message-based inter-task communication system calls to the lab2 RTX. You will then design and implement a set of system tasks to enable an interrupt driven RTX console by using message passing.

5.3.1 Overview

The first part of the project is to make the RTX support message-based inter-task communication. A task will be able to request a mailbox to receive messages from other tasks. Sending and receiving messages to and from other tasks will be performed by using system calls. The kernel provides the system calls in Listing 5.1 to support message passing and task information retrieval.

```
#include "rtx.h"

int mbx_create(size_size);
int send_msg(task_t receiver_tid, const void *buf);
int send_msg_nb(task_t receiver_tid, const void *buf);
int recv_msg(void *buf, size_t len);
int recv_msg_nb(void *buf, size_t len);
int tsk_ls(task_t *buf, size_t count);
int mbx_ls(task_t *buf, size_t count);
int mbx_get(task_t tid);
```

Listing 5.1: Lab3 System Calls

The `mbx_create()` system call creates a mailbox for the calling task. The `send_msg()` and `send_msg_nb()` are system calls to send a message to the mailbox of a task. The `recv_msg()` and `recv_msg_nb()` are system calls to receive a message from the mailbox of the calling task. The `send/receive_msg()` are blocking calls and the `send/receive_nb()` are non-blocking calls. The `tsk_ls()` is a utility system call to return task IDs of non-dormant tasks in the system. The `mbx_ls()` is a utility system call to return task IDs of those that have mailboxes. The `mbx_get()` returns a task's mailbox free space size.

The second part is to add UART interrupt services to enable the RTX system console. The microcontroller communicates with the system console using UART0 by interrupts. There are two RS-232 ports : COM0 and COM1 on the board which connects to the UART0 and UART1 respectively on the microcontroller board. Each UART is connected to the microprocessor address, data and control buses on one side, and to the incoming (keyboard) and outgoing (character display) serial data link on the other side (See Figure 5.1).

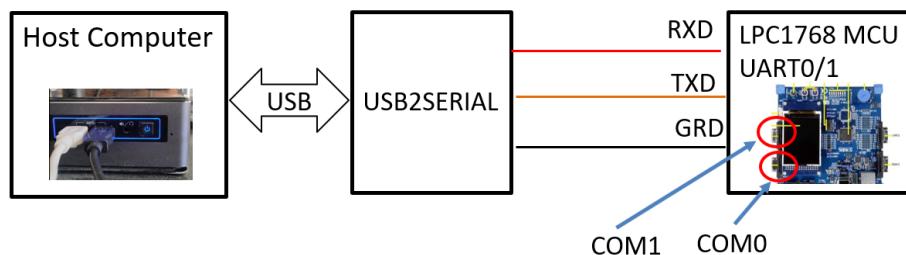


Figure 5.1: UART and Host Computer Connection

An UART consists of two parts: the Receiver and the Transmitter. The receiver

contains a 16-byte hardware FIFO to store the characters received. The transmitter contains a 16-byte hardware FIFO to store the characters to be transmitted. In previous labs, we used UART1 to output characters to a debug terminal by polling. We will still use this for outputting debugging messages and test task output messages in this lab. In addition, we will support system console to accept keyboard input and display characters by using interrupt-driven UART0. The output of a command issued to the system console should be displayed on the system console through UART0 interrupt (i.e. not through the UART1 debug terminal by polling).

The RTX will run two daemon tasks – the Keyboard Command Decoder (KCD) task and the Console Display 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 COM0 will trigger UART0 interrupts. The UART0 handler forwards received characters from the keyboard to the KCD task. It responds to UART0 output requests that are received from the Console Display task by transmitting characters to serial port COM0. Other tasks can also communicate with the Console Display task to output characters to RTX system console.

5.3.2 Data Structures and Algorithm

Mailboxes are buffer-like kernel objects used for data communication and synchronization between two tasks or between an interrupt handler and a task. We need to design data structures to represent a mailbox for a task. The kernel copies the message from the sender's message buffer to the mailbox of the receiver. A mailbox needs memory buffer to store the copied messages.

Aside from the memory buffer, a mailbox has a unique ID to identify itself. The kernel associates this mailbox ID with the calling task when the mailbox is created. A task can request maximum one mailbox from the operating system to receive messages. When the mailbox is created, the task specifies the memory buffer size. The messages are received in the first come first serve order. A ring buffer data structure is recommended for the mailbox memory buffer.

Each mailbox is associated with one receiving task. Multiple tasks can send messages to a mailbox. When the mailbox is empty, a task calling a blocking receive system call will be blocked. When the mailbox is full or does not have enough space for the incoming message, a task calling a blocking send system call will be blocked. Each mailbox has two task-waiting lists. The receiving task-waiting list consists of tasks that are blocked when mailbox is empty. The sending task-waiting list consists of tasks that are blocked when the mailbox is full or does not have enough space for an incoming message.

TODO #1

The `rtx_init()` is to set up kernel data structures during RTX initialization time. You need to add your own mailbox management data structures' initialization operations in the starter code's kernel.

5.3.3 Scheduler

The scheduling policy used in this lab is the same preemptive priority scheduler used in lab2. We have two new states: `BLK_SEND` and `BLK_RECV`. Figure 5.2 shows the state transition diagram with the newly added blocked states.

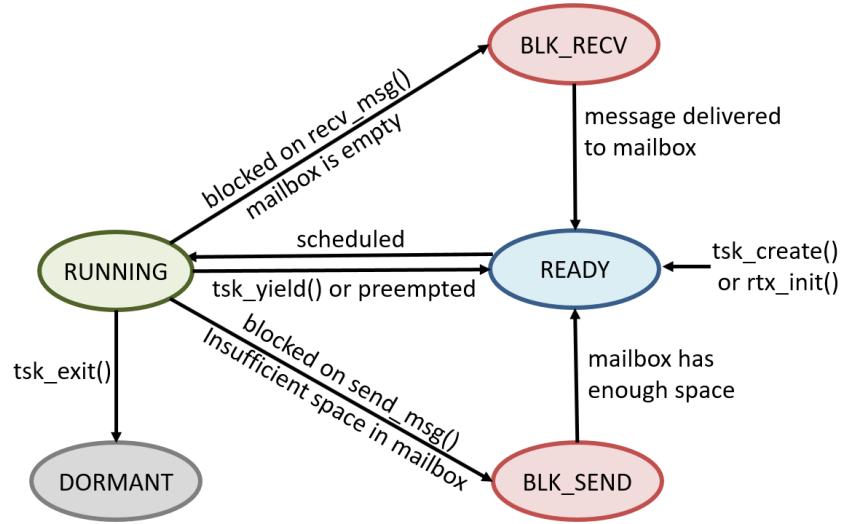


Figure 5.2: Lab3 State Transition Diagram

When a task calls `send_msg` and the receiver's mailbox is full or does not have enough space to receive the incoming message, the kernel changes the state of the calling task to `BLK_SEND` and adds the calling task to the sending task-waiting list of the receiver's mailbox. The scheduler then makes new scheduling decision. When a task calls `recv_msg` and its own mailbox is empty, the kernel changes the state of the calling task to `BLK_RECV` and adds the calling task to the receiving task-waiting list of its own mailbox. The scheduler then makes a new scheduling decision.

When a message is sent to a mailbox and the receiver's task-waiting list is not empty, a task is removed from the front of the waiting list and gets added to the back of the ready queue of its priority and its state changes to `READY`. The scheduler then makes new scheduling decision.

When a mailbox sending task-waiting list is not empty and the mailbox has enough free space to hold the message from the first task in the sending task-waiting list, then the task is removed from the front of the sending task-waiting list of the mailbox and gets added to the back of the ready queue of its priority and its state changes to `READY`. The scheduler then makes new scheduling decision.

If the new scheduling decision preempts the current running task, then this is an involuntary loss of the cpu usage. The preempted task is added to the front of the ready queue of its priority.

TODO #2

The starter code kernel does not implement preemptive priority scheduler. You need to enhance your lab2 preemptive priority scheduler so that it will be able to support the BLK_RECV and BLK_SEND states. You also need to update your `tsk_set_prio` system call so that it can change the priority of a task that is blocked by the kernel.

5.3.4 Context Switching

Similar as in lab2, context switching has been implemented by the starter code by using statically allocated user stacks.

TODO #3

You will need to integrate your lab2 implementation with lab3 starter code's kernel so that the user stacks are dynamically allocated from memory pool identified by `MPID_IRAM2`.

5.3.5 Interrupt Handler and System Tasks

The UART0 interrupt handler is given in the starter code. It shows how to echo a key press and uses interrupts to transmit a string to display on a serial terminal. It also shows how a key press can trigger a context switching. To make the interrupt handler work coherently with the kernel, the kernel provides two built-in system tasks to communicate with the UART0 interrupt handler by using mailboxes. The UART0 interrupt handler associates a reserved task ID of `TID_UART` so that it can send/receive messages to/from two system tasks (see Section 5.3.8) since the message passing uses task ID to identify the sender and receiver. Note the interrupt handler is not a task, this reserved task ID is only for message sender/receiver identification purpose. The function prototypes of the two system tasks and their empty implementations are included in the starter code in the built-in tasks source group of the RTX-App project.

TODO #4

The starter code's kernel does not implement inter-task communications between the UART0 interrupt handler and the two system tasks. The two system tasks are

not implemented either. You will need to add message passing logic to the interrupt handler. The starter code's kernel does not echo the keyboard input to COM0 terminal. Once you have message passing logic implemented for the UART0 to receive display messages from the console display task, you will use code in `UART0_IRQ` project as a reference to add keyboard echo functionality to the kernel. You will complete the two system tasks. The two system tasks are built-in tasks and the kernel needs to create them during the kernel initialization phase without the need for `rtx_init` to pass their information to the system. The sizes of user stacks of these two tasks are free design and implementation choice.

5.3.6 Types, Macros and Message Header Structure

The `common.h` 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.

Listing 5.2 shows lab3 relevant macros that both the tasks and kernel can reference.

```
/* typedef */
typedef signed char          mbx_t;      /* mailbox descriptor type */

/* Task IDs */
#define TID_CON   (MAX_TASKS -2)           /* reserved Task ID for console display task */
#define TID_KCD   (MAX_TASKS -1)           /* reserved Task ID for KCD task */
#define TID_UART  0xFE                  /* reserved TID for UART IRQ handler which is
                                             not a task */
#define TID_UNK   0xFF                  /* reserved TID for unknown tasks */

/* task state macro */
#define BLK_SEND  3                   /* blocked on a full mailbox on send */
#define BLK_RECV  4                   /* blocked on an empty mailbox on receive */

/* Message Types */
#define DEFAULT  0                   /* a general purpose message */
#define KCD_REG  1                   /* a command registration message */
#define KCD_CMD  2                   /* a message that contains a command */
#define DISPLAY  3                   /* a message that contains chars to be displayed
                                             to the RTX console */
#define KEY_IN   4                   /* keyboard input from console */

/* Mailbox Sizes */
#define MSG_HDR_SIZE    sizeof(RTX_MSG_HDR) /* rtx_msg_hdr struct size */
#define MIN_MSG_SIZE    MSG_HDR_SIZE       /* minimum message size in
                                             bytes */
#define KCD_MBX_SIZE    0x200             /* KCD mailbox size */
#define CON_MBX_SIZE    0x80              /* consolde display mailbox size */
#define UART_MBX_SIZE   0x80              /* UART interrupt handler mailbox size */
```

Listing 5.2: Lab3 Macros

A newly added data structure for this lab is the `rtx_msg_hdr` (see Listing 5.3). This structure is used for message header sent/received. The `__packed` qualifier results in no padding inserted to align the structure. The size of this structure therefore is 6 bytes¹.

```
typedef __packed struct rtx_msg_hdr {
    U32     length;      /**< length of the message including
                          the message header size */
    task_t   sender_tid; /*< sending task tid */
    U8      type;        /**< type of the message */
} RTX_MSG_HDR;
```

Listing 5.3: Message Buffer Header Data Structure

5.3.7 Specifications of Functions

The starter code already implemented the trap table of the system calls. Each system call has its kernel function counterpart. You need to complete the kernel counterparts of the system calls. Specifications of system calls are described below. Their kernel counterparts have the same interface and semantics.

Mailbox Creation Function

NAME

`mbx_create` - create a mailbox

SYNOPSIS

```
#include "rtx.h"

mbx_t mbx_create(size_t size);
```

DESCRIPTION

The `mbx_create` 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. The capacity is used for the messages from the tasks². 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. That is a dormant task does not have a mailbox associated with it.

¹Without the `__packed` qualifier, compiler will add two bytes padding to the structure and make the size of the structure to be 8 bytes.

²Any meta-data the kernel needs in order to manage the messages in the mailbox is not counted into the capacity of a mailbox. Kernel should allocates separate memory for the meta-data.

RETURN VALUE

The `mbx_create` function returns a non-negative mailbox ID on success and `-1` on failure (`errno` is set).

ERRORS

- `EEXIST` The calling task already has a mailbox.
- `ENOMEM` There is not enough memory to support the operation.
- `EINVAL` The `size` argument is less than `MIN_MSG_SIZE`.

SEE ALSO

`send_msg`, `send_msg_nb`, `recv_msg`, `recv_msg_nb`, `mbx_ls`

Note that any task can send messages to a mailbox. However, only one task can receive messages from each mailbox.

Send Message Functions

The RTX supports both blocking send and non-blocking send system calls. If the mailbox has enough space to hold the message, both functions deliver the message to the receiver's mailbox and return. A task calling a non-blocking send function will return immediately regardless whether the mailbox has enough space or not and the return code indicates whether the send operation is successful. When a mailbox is full or does not have enough space, a task calling a blocking send function will be blocked by the kernel and the scheduler makes new scheduling decision. When enough free space becomes available in the receiver's mailbox, the kernel will unblock a task that is blocked on `send_msg()` to the mailbox and re-schedule the tasks in the system.

NAME

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

SYNOPSIS

```
#include "rtx.h"

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

DESCRIPTION

The `send_msg` and `send_msg_nb` functions deliver the message specified by `buf` to the mailbox of the task identified by the `receiver_tid`. The message starts with a message header followed by the actual message data (see Figure 5.3).

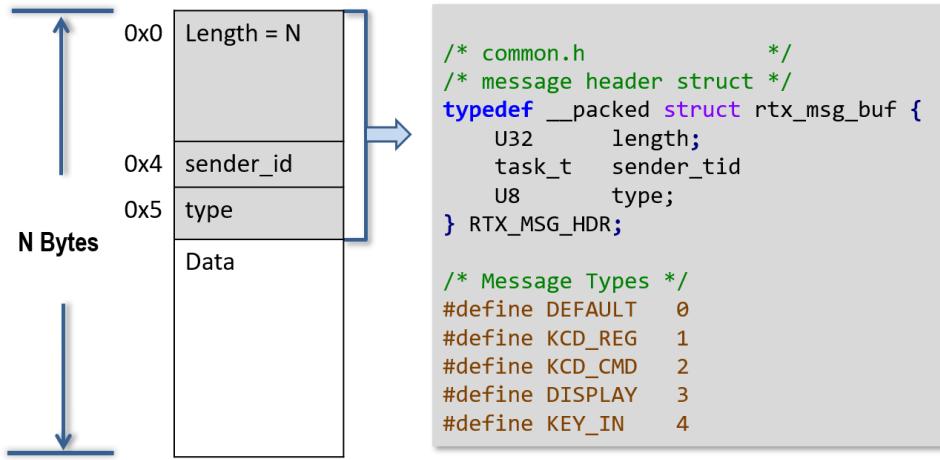


Figure 5.3: Structure of a message buffer

The message header data structure is defined in Listing 5.3. Figure 5.3 shows the relationship between a message buffer and message header. The `length` field in the structure is the size of the message including the message header size. The `sender_tid` field identifies the sender by its task ID. The `type` field is the message type. Tasks can define their own message types for communication provided they are not conflict with the predefined types listed below.

`DEFAULT` A general purpose message.

`KCD_REG` A message to register a command with the KCD task (see Section 5.3.8).

`KCD_CMD` A message that contains a command to be handled by the receiving task (see Section 5.3.8)

`DISPLAY` A message to display data on the console terminal.

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

Both the host computer and the microcontroller board are little-endian systems. Note that kernel copies the actual message data into the receiver task's mailbox.

RETURN VALUE

The `send_msg` function returns `0` on success and `-1` on failure (`errno` is set).

ERROR

<code>ENOENT</code>	The receiver identified by the <code>receiver_tid</code> exists but does not have a mailbox.
<code>EINVAL</code>	The receiver identified by the <code>receiver_tid</code> does not exist. The <code>length</code> field in the <code>buf</code> specifies a size that is less than <code>MIN_MSG_SIZE</code> .
<code>EMSGSIZE</code>	The <code>length</code> field in the <code>buf</code> exceeds the mailbox capacity.
<code>EFAULT</code>	The <code>buf</code> argument is a null pointer.
<code>ENOSPC</code>	The <code>receiver_tid</code> 's mailbox does not have enough free space for the message sent by <code>send_msg_nb()</code> when the <code>length</code> field in the <code>buf</code> is legitimate.

SEE ALSO

`mbx_create`, `recv_msg`, `recv_msg_nb`

Receive Message Functions

NAME

`recv_msg` - blocking receive a message
`recv_msg_nb` - non-blocking receive a message

SYNOPSIS

```
#include "rtx.h"

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

DESCRIPTION

The task calling `recv_msg` or `recv_msg_nb` receives a message from its mailbox if there is any. 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 5.3). 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 of `recv_msg_nb` returns `-1` (`errno` is set). If the mailbox is empty, the calling task of `recv_msg` gets blocked.

RETURN VALUE

The `recv_msg` and `recv_msg_nb` function returns `0` on success and `-1` on failure (`errno` is set).

ERROR

- `EFAULT` The `buf` argument is a null pointer.
- `ENOSPC` The buffer is too small to hold the message .
- `ENOENT` The calling task does not have a mailbox.
- `ENOMSG` There is no message when `recv_msg_nb` is called.

SEE ALSO

`mbx_create`, `send_msg`, `send_msg_nb`

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, size_t count);
```

DESCRIPTION

The `tsk_ls` returns the task IDs of the first `count` tasks (in increasing order of task IDs) in the system that are not in the DORMANT state. 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` (`errno` is set).

ERROR

EFAULT The `buf` argument is a null pointer.

SEE ALSO

`tsk_create`, `tsk_exit`, `tsk_get`, `tsk_gettid`

Mailbox Utility Functions

NAME

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

SYNOPSIS

```
#include "rtx.h"

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

DESCRIPTION

The `mbx_ls` function returns the task IDs of the first `count` tasks (in increasing order of task IDs) in the system that have mailboxes. 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 tasks in the system with mailboxes. On failure, the function returns `-1` (`errno` is set).

ERROR

EFAULT The `buf` argument is a null pointer.

SEE ALSO

`mbx_create`

NAME

`mbx_get` - return free space size of a mailbox

SYNOPSIS

```
#include "rtx.h"

int mbx_get(task_t tid);
```

DESCRIPTION

The `mbx_get` function returns the number of bytes that are not used in the mailbox identified by `tid` argument.

RETURN VALUE

On success, the `mbx_get` function returns the free space of the mailbox in bytes.
On failure, the function returns `-1` (`errno` is set).

ERROR

`ENOENT` There is no mailbox assigned to the task or the interrupt handler identified by `tid`.

SEE ALSO

`mbx_create`, `mbx_ls`

5.3.8 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 character that is the command's identifier. Command identifiers are case sensitive and alphanumeric. 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 reserves `TID_KCD` for KCD's task ID. KCD should request a mailbox when it first starts to run. The KCD's mailbox size is determined by `KCD_MBX_SIZE` macro defined in the `common.h`. 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): 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 5.4 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;
ptr->sender_tid = tsk_gettid();
buf += msg_hdr_size;
*buf = 'W';
send_msg(TID_KCD, (void *)ptr);
```

Listing 5.4: 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). Each task can register multiple command identifiers. Tasks never de-register a command identifier. Tasks can (re-)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 interrupt handler will forward any key input to the mailbox of the KCD using a KEY_IN message. The KCD will forward this message to the console display task's mailbox to be echoed back to the console. KCD also should queue the input keys. Upon receiving "enter" key (i.e. carriage return), KCD dequeues all previous keys to construct a single string. If the string is %LT, then KCD will print the task ID and state of all non-DORMANT tasks on the console. If the string is %LM, then KCD will print the task ID and state of all non-DORMANT tasks with a mailbox on the console. It will also print out the mailbox ID associated with the task, how many tasks are in the sending task-waiting list and how many tasks are in the receiving task-waiting list. 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 % character 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 console display task's mailbox to print "Command not found". 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 console display task to print “Invalid command”.

The relationship among the KCD task, console display task, UART0 interrupt handler and a task that has a command registered is illustrated in Figure 5.4.

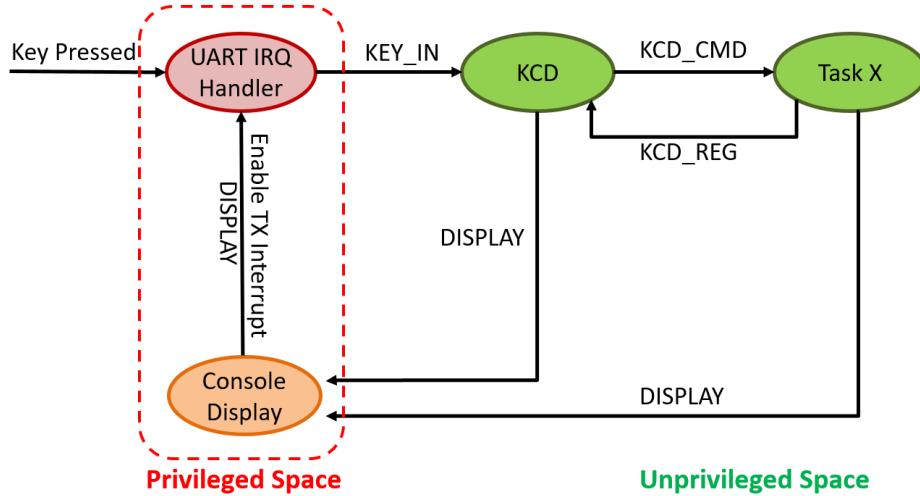


Figure 5.4: Message passing among system console I/O tasks, UART interrupt handler and a command handling task communication.

The Console Display Task

The console display task is a privileged thread-mode task. The kernel reserves the task ID of `TID_CON` for this task. Similar to KCD, the console display task also creates a mailbox when it first runs and then starts listening on its mailbox. The console display task’s mailbox size is determined by `CON_MBX_SIZE` macro defined in the `common.h`. The console display task responds only to one message type (and ignores the rest): a `DISPLAY` message. The message body contains a character string to be displayed. The string may contain control characters (e.g. newline and ANSI escape sequences et. al.). The console display task displays the string to the console terminal. As a privileged kernel task, the console display task can disable the UART0 transmit interrupt when all characters of the string have been displayed. The console display task and the UART0 interrupt handler use mailbox of UART0 irq handler to communicate.

5.3.9 UART0 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_UART` is reserved to indicate the message is from the UART0 interrupt handler. To let UART0 to communicate with the console display task, kernel creates a mailbox for the UART0 interrupt handler to put the strings to be dis-

played in the mailbox. The size of this mailbox is determined by `UART_MBX_SIZE` in the `common.h` file. The `UART0` gets the data from the mailbox and writes the data to `UART0` transmit register. Please note that `UART0` IRQ handler is not a task. It is an interrupt handler that can send messages to KCD and receive messages from the console display task. Note that an interrupt handler cannot be blocked, so it should perform non-blocking send/receive in handler mode by directly invoking kernel counterpart of non-blocking send/receive message functions rather than system calls.

Interrupt handler is not a task. It uses the kernel stack of the interrupted task. After interrupt handler is done, RTX should resume the interrupted task by default unless the state of some other tasks has changed in which case a new scheduling decision has to be made.

5.3.10 User Tasks

Testing Tasks

In order to test your implementation of the required functions, write an application that uses your system calls. The provided `ae_tasks<TS_id>_G<id>. [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.

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 sending and then receiving messages while making sure that no messages are lost. Another testing objective that you may want to consider is preemption. You could create multiple tasks with different priorities and create situations some tasks are blocked due on send or receive to/from the mailbox and later on become unblocked and test preemption. The utility functions `tsk_ls()`, `tsk_get`, `mbx_ls()` and `mbx_get` are useful tools for checking system mailbox and task status information.

Similar to lab1 and lab2, you will use the polled `UART1` terminal to output the testing results, and you will format the testing results as described in in Section [3.5.1](#).

5.4 Source Code File Organization and Third-party Testing

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

Don'ts

- Do not change the locations or names of existing files or directories.
- Do not modify any of the header files in the `lab3/include` folder and its sub-folders except for `rtx_ext.h` and `common_ext.h`.
- Do not change the prototypes of existing functions appeared in the `rtx.h` and `common.h` files. You may change the implementation of those functions though.
- Do not include any new header files in the `main.c`.
- Do not include any new header files in the `ae.c` file.

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 ³.
- The newly created .h files are allowed to be included in the files of RTX-App project.
- Any new files you add to the project can be put into either the `kernel` directory or other directories you will create under the `src` directory.
- For any functions that you are not allowed to change their prototypes, you are allowed to change their implementations.
- For any functions that you are allowed to change their prototypes, you are free to do any modifications to them. You are responsible to modify the rest of the code due to the modifications you have done.

The `main.c` in AE-Lib project starts the third-party testing by calling `ae_init` function and this function will eventually call the `set_ae_tasks` which the third-party testing software implements. The function prototypes of these functions do not change. But the implementation of the function may change in real testing. Do not delete the lines in the `main.c` where `ae_init` function is invoked. We will write more `ae_tasks*.c` files with more complicated testing cases than the ones released in the starter code during the third-party testing..

³For example, you may want to put generic data structure functions or some helper functions in new files for better file organization.

5.4.1 Report

Write the following items in your report and name it `lab3_report.pdf`.

- Descriptions of the data structures and algorithms used to implement all the functions listed in Section 5.3.7.
- Testing scenario descriptions

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

- creating mailboxes,
- blocking and non-blocking send messages,
- blocking and non-blocking receive messages,
- Preemption caused by message passing,
- getting correct information on tasks and mailboxes.

5.5 Deliverables

5.5.1 Pre-Lab Deliverables

None.

5.5.2 Post-Lab Deliverables

You should structure your lab3-submission sub-directory in the GitLab repository so that it looks like the one shown in Figure 5.5. We no longer need to RTX-Lib sub directory. You may remove it from your git repository. Keeping it there does not affect anything either.

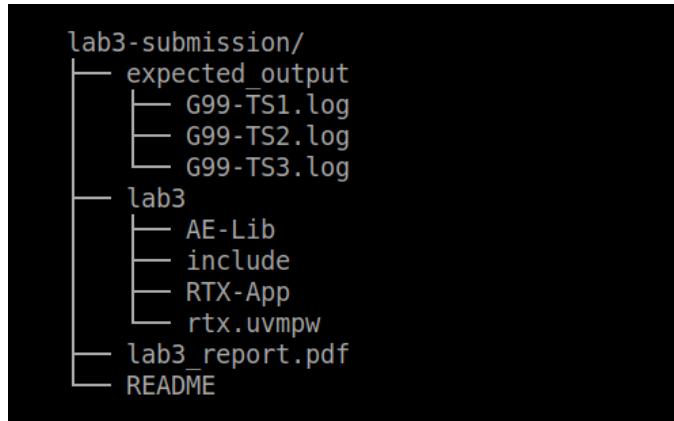


Figure 5.5: Lab3 Submission Directory Layout

The lab project and the report should be committed and pushed to the GitLab group project repository for submission. Put the GitLab commit hash string you would like to use for submission in a file and name it `lab3-submission.txt` for drop box submission. Use the `git log` command to list the commit history and the hash strings (see Figure 4.6).

5.6 Marking Rubric

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

Table 5.1: Lab3 Marking Rubric

The Rubric for marking the submitted source code and report is listed in Table 5.1. The functionality and performance of your implementation will be tested by a third-party test program. 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.

Chapter 6

Lab4

To be released by July 16, 2021.

Part III

Computing Environment and Development Tools Quick Reference Guide

Chapter 7

Windows 10 Remote Desktop

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

The Englаб at <https://englab.uwaterloo.ca/> is the main gateway. Choose ECE → **ece-rtos*** machines. Use remote desktop application on your computer to open up the downloaded file. For Windows 10 platform, when prompt for user name, input Nexus\userid, where the userid is your quest ID. For Linux or MAC platforms, input Nexus in the domain field, input your quest ID in the username field. The password is your Quest password. Then you are connected to one of the lab machines that has the software and hardware installed for this lab.

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

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

Chapter 8

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>.

8.1 Getting Started with 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.

8.2 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.git
```

8.3 Start the Keil uVision5 IDE

The Keil uVision5 IDE shortcut should be accessible from the start menu on school computers. If not, then navigate to C:\Software\Keil_v5\UV4 folder and double click the **UV4.exe** to bring up the IDE (see Figure 8.1).

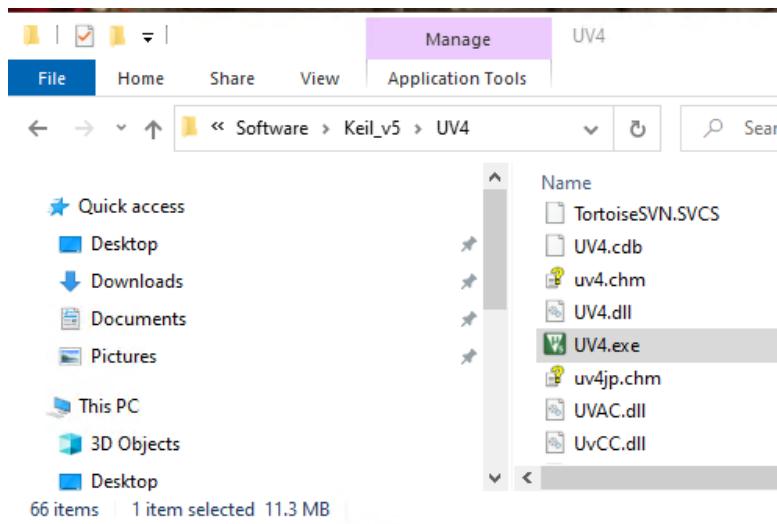


Figure 8.1: Keil IDE: Create a New Project

8.4 Create a New uVision5 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 8.2).

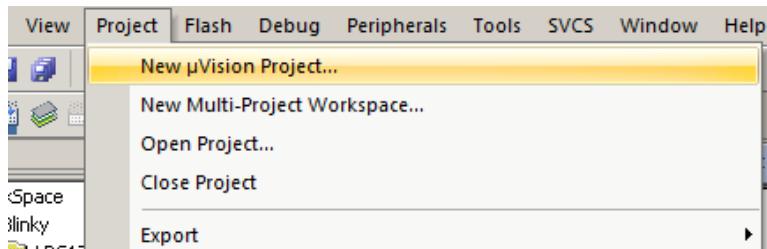


Figure 8.2: Keil IDE: Create a New Project

- Select NXP → LPC1700 Series → LPC176x → LPC1768 (See Figure 8.3).

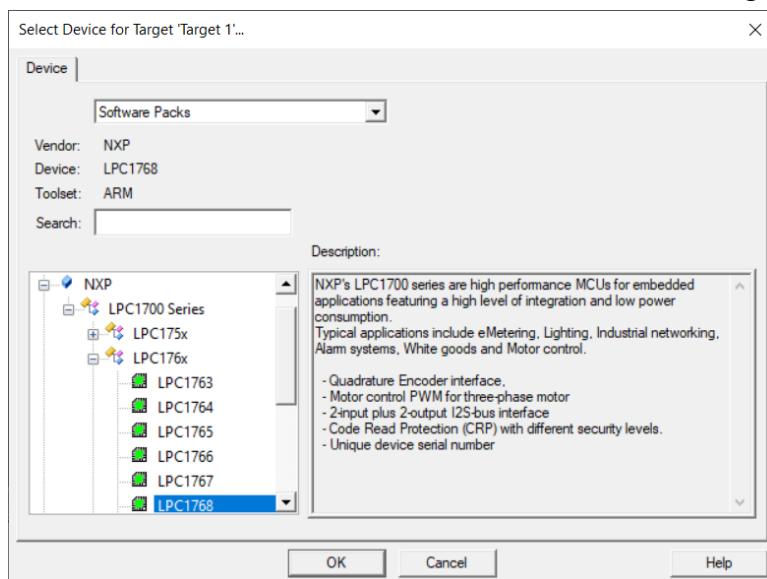


Figure 8.3: Keil IDE: Choose MCU

- Select CMSIS → CORE and Device → Startup (See Figure 8.4).

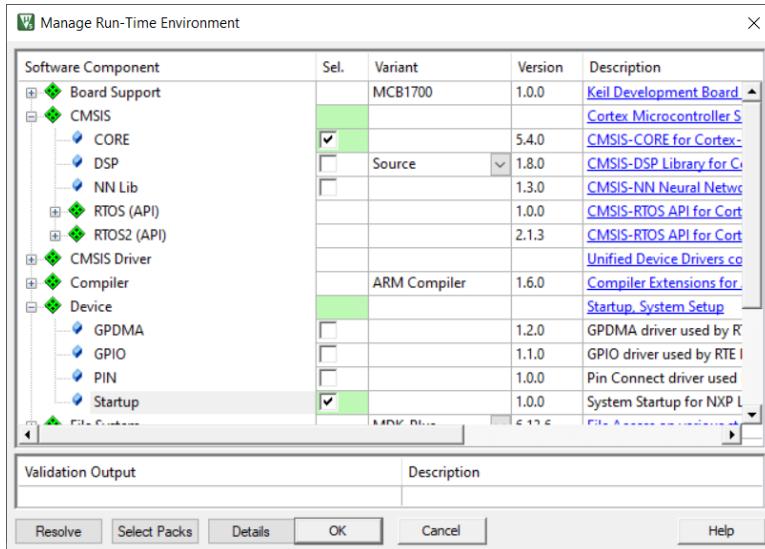


Figure 8.4: Keil IDE: Manage Run-time Environment

8.5 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 8.5.

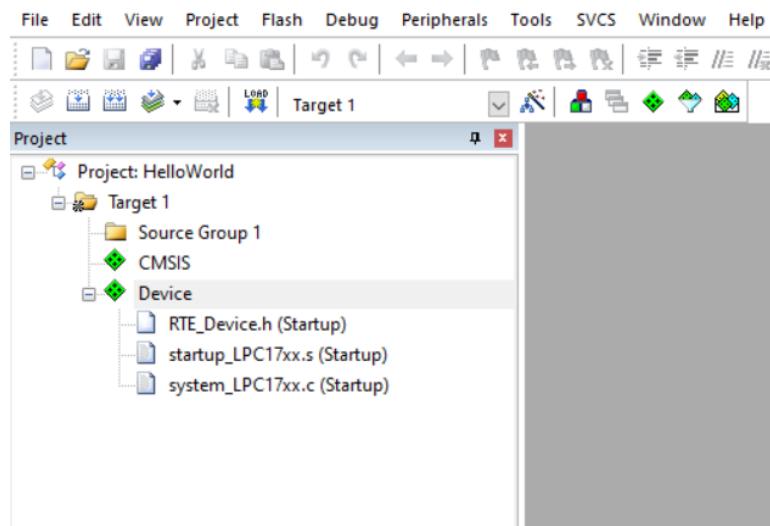


Figure 8.5: Keil IDE: A default new project

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. 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 8.6).

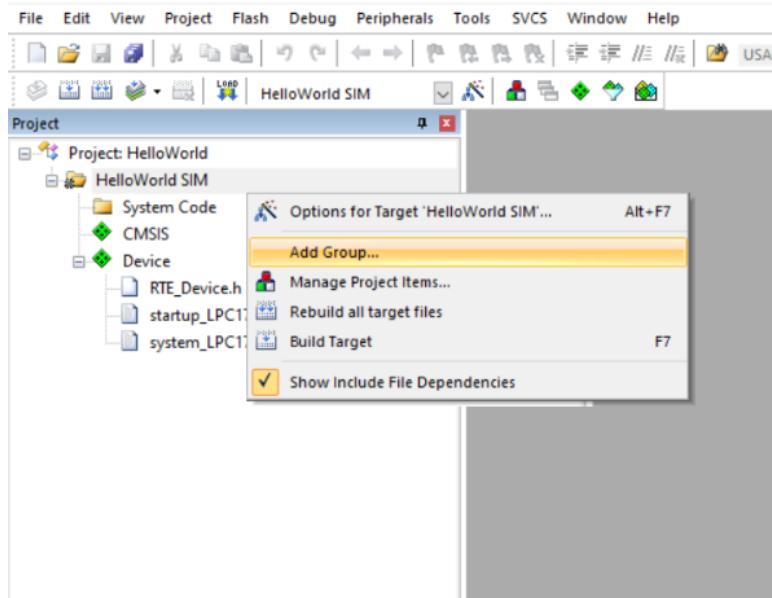


Figure 8.6: Keil IDE: Add Group

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 8.7.

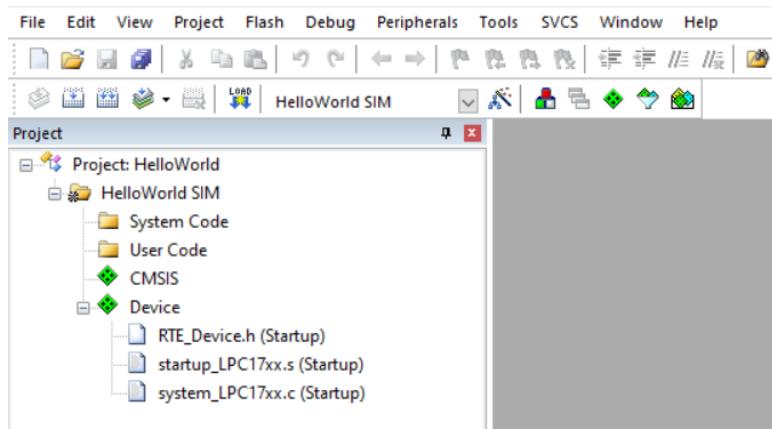


Figure 8.7: Keil IDE: Updated Project Profile

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

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 8.8).

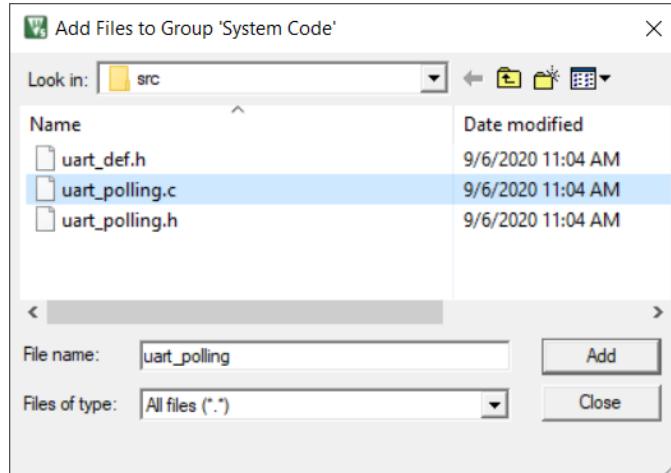


Figure 8.8: Keil IDE: Add Source File to Source Group

Your project will now look like Figure 8.9.

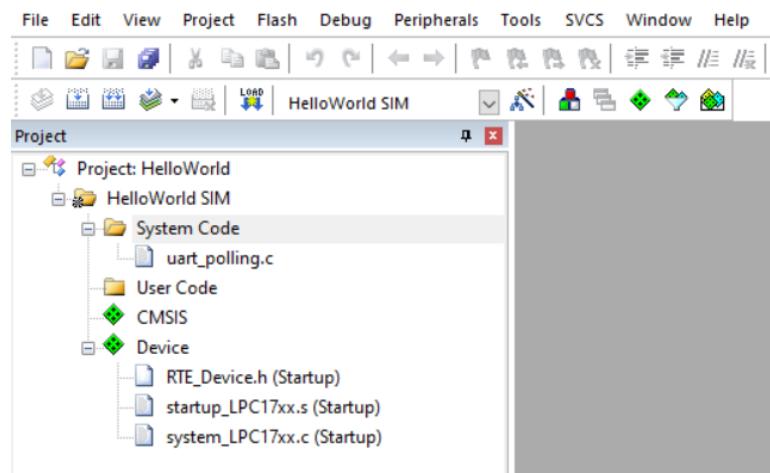


Figure 8.9: Keil IDE: Updated Project Profile

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 8.10).

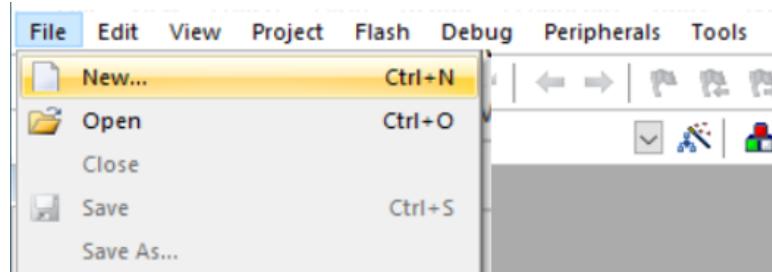


Figure 8.10: Keil IDE: Create New File

Before typing anything to the file, save the file and name it “main.c”.

Add main.c to the “Source Code” group. Type the source code as shown in Figure 8.11. Your final project would look like the screen shot in Figure 8.11.

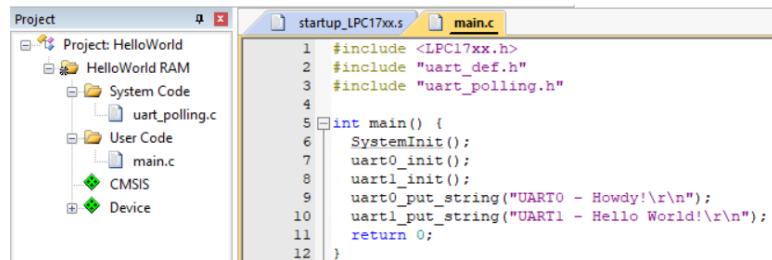


Figure 8.11: Keil IDE: Final Project Setting

8.6 Build the Project Target

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

8.6.1 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 8.12).

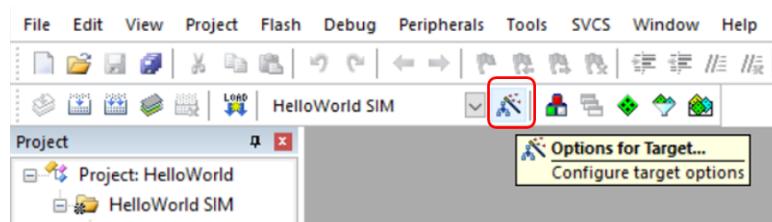


Figure 8.12: Keil IDE: Target Options Configuration

2. Configure the Target tab as shown in Figure 8.13. We want to use the default version 5 arm compiler. We also want remove the IRAM2 from the default setting.

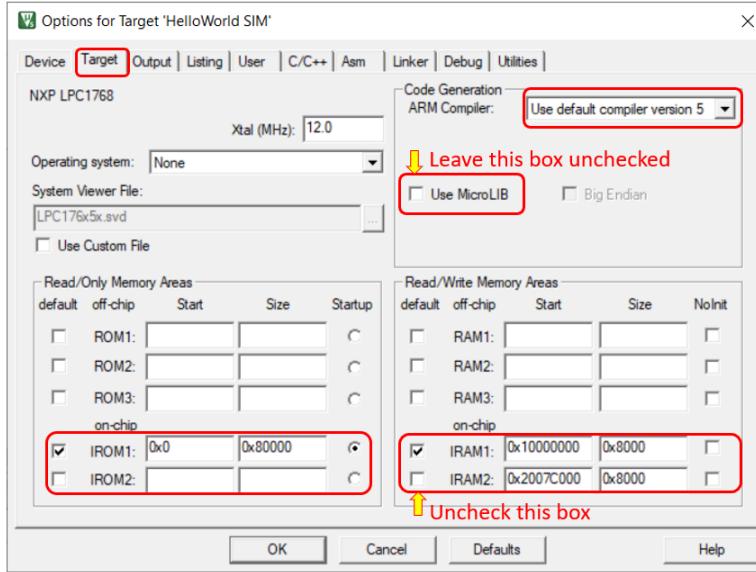


Figure 8.13: Keil IDE: Target Options Target Tab Configuration

3. Configure the C/C++ tab as shown in Figure 8.14. To enable c99, we need to check the C99 Mode box. We also want to keep the default optimization level of zero.

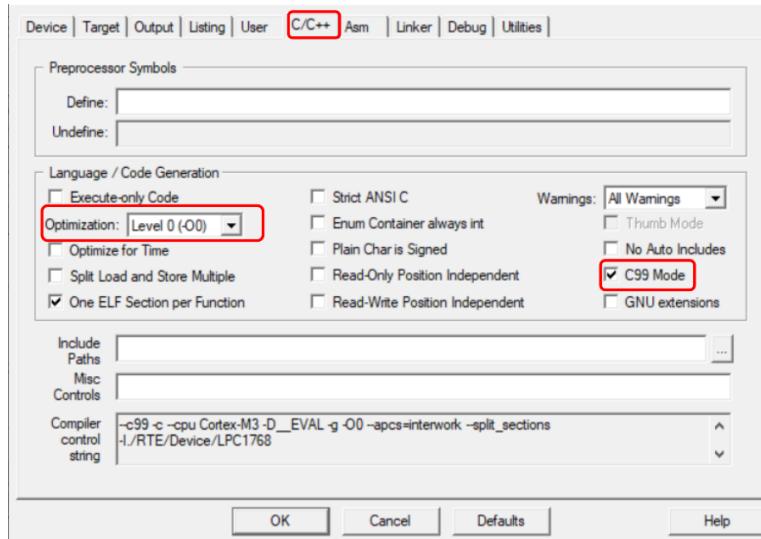


Figure 8.14: Keil IDE: Target Options C/C++ Tab Configuration

4. Configure the Linker tab as shown in Figure 8.15. This is to instruct the linker to use the memory layout from the Target tab setting instead of the default memory layout.

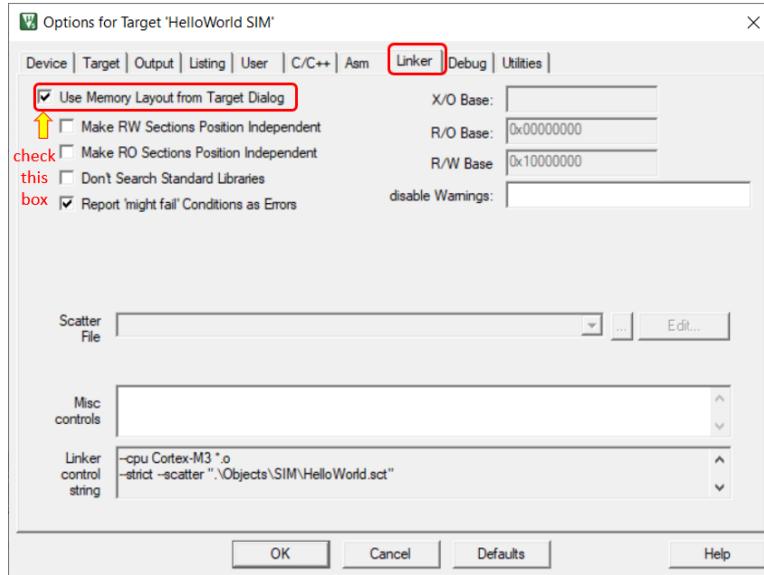


Figure 8.15: Keil IDE: Target Options Linker Tab Configuration

- Configure the Output tab so that the created executable will be put into a sub-folder of .\Objects\SIM. Note, when you press the Select Folder for Objects button (see Figure 8.16), the default directory to put the target is the .\Objects folder. You create SIM sub-folder under the default .\Objects folder and select the SIM sub-folder.



Figure 8.16: Keil IDE: Target Options Output Tab Configuration for SIM Target

- Press the “OK” button to finish the target option configuration.

8.6.2 Build the Target

To build the target, click the “Build” button (see Figure 8.17).

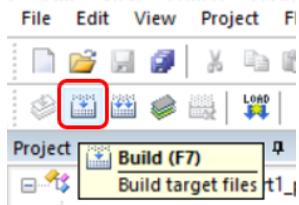


Figure 8.17: Keil IDE: Build Target

If nothing goes wrong, the build output window at the bottom of the IDE will show a log similar like the one shown in Figure 8.18.

```

Build Output
*** Using Compiler 'V5.06 (build 20)', folder: 'C:\Software\Keil_v5\ARM\ARMCC\Bin'
Build target 'HelloWorld SIM'
assembling startup_LPC17xx.s...
compiling system_LPC17xx.c...
compiling main.c...
compiling uart_polling.c...
linking...
Program Size: Code=924 RO-data=220 RW-data=0 ZI-data=608
".\Objects\SIM\HelloWorld.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed: 00:00:02

```

Figure 8.18: Keil IDE: Build Target

8.7 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 8.7.2.

8.7.1 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 8.19. The debug script `SIM.ini` provided in the starter code (see Listing B.1 in Appendix B) is needed to map the second bank of RAM area read and write accessible inside the simulator.

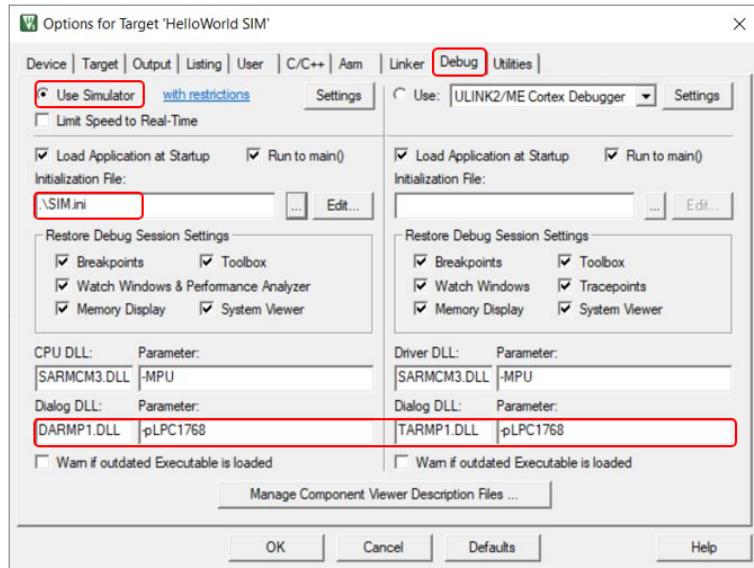


Figure 8.19: Keil IDE: Target Options Debug Tab Configuration

2. Press the “debug” button to bring up the debugger interface (See Figure 8.20).



Figure 8.20: Keil IDE: Debug Button

3. Select UART1 and UART2 (see Figure 8.21) from the serial window drop down list so that they appear in simulator (see Figure 8.22). 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.

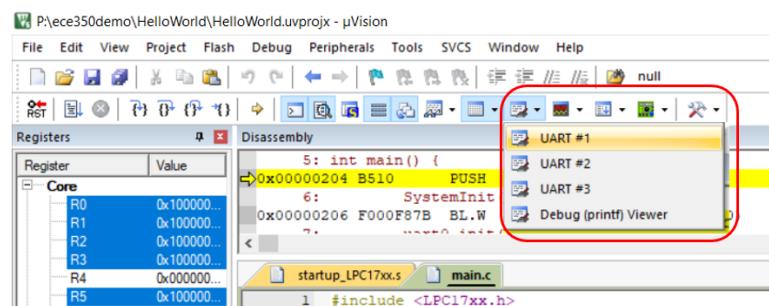


Figure 8.21: Keil IDE: Debugging. Enable Serial Window View.

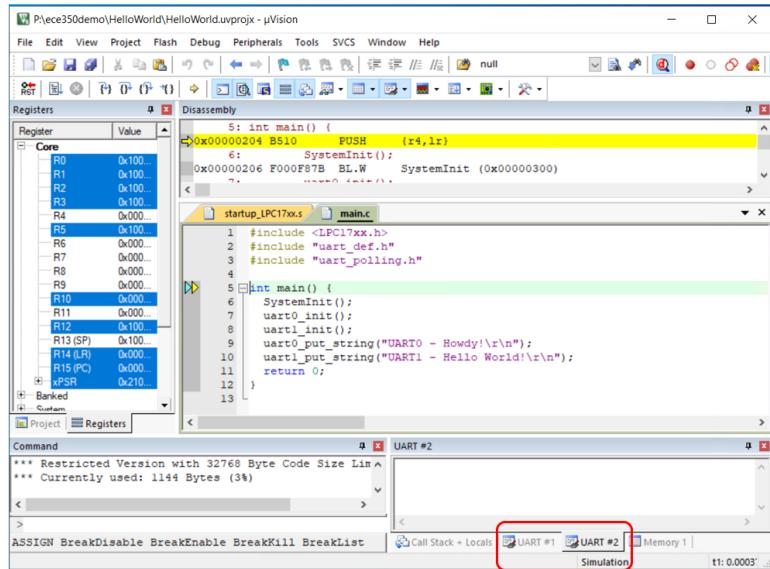


Figure 8.22: Keil IDE: Debugging. Both UART0 and UART1 views are enabled in simulator.

4. Press the “Run” button on the menu to let the program execute (see Figure 8.23). You will see the output of UART0 appearing in UART1 simulator window and the output of UART1 appearing in UART2 simulator window (see Figure 8.24). Note that we moved the UART windows from their default positions in the simulator for better view.

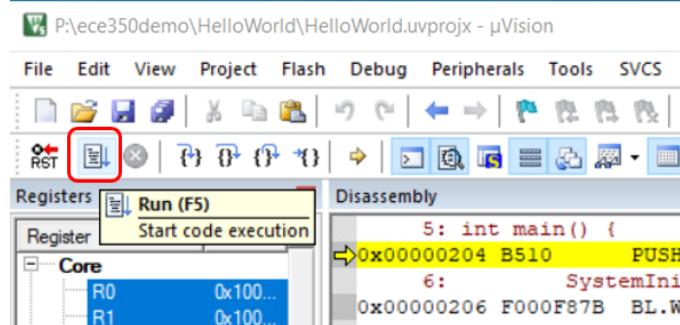


Figure 8.23: Keil IDE: Debugging. The Run Button.

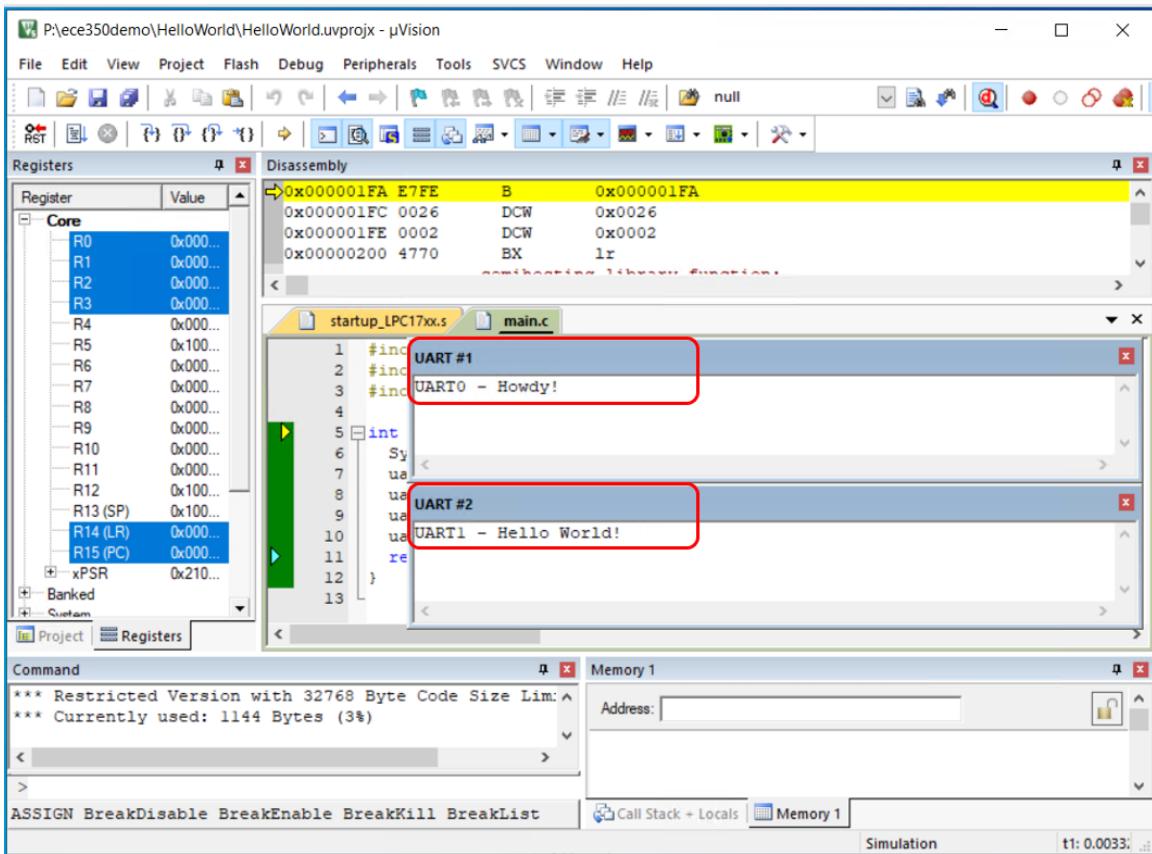


Figure 8.24: Keil IDE: Debugging Output.

- To exit the debugging session, press the “debug” button again (see Figure 8.20).

8.7.2 Debug the Project on the Board by In-Memory Execution

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.

- Press the Managing Project Item button (see Figure 8.25).

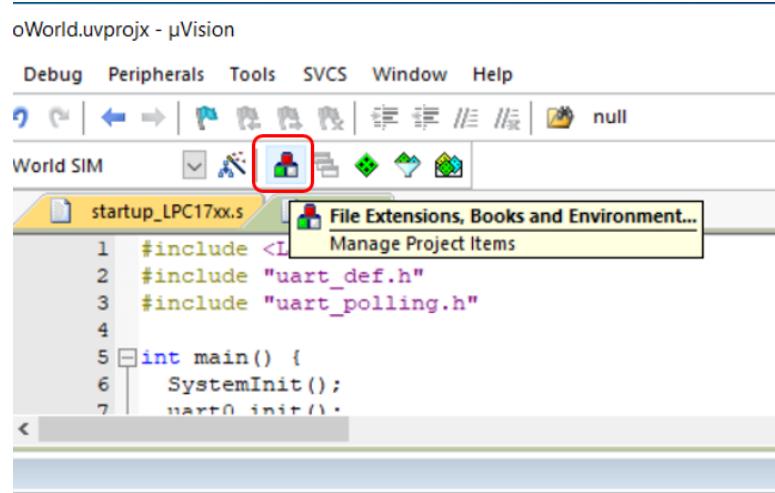


Figure 8.25: Keil IDE: Manage Project Items Button

2. Press the New icon to create a new target and name it "HelloWorld RAM" (see Figure 8.26). The new target duplicates the HelloWorld SIM target configuration.

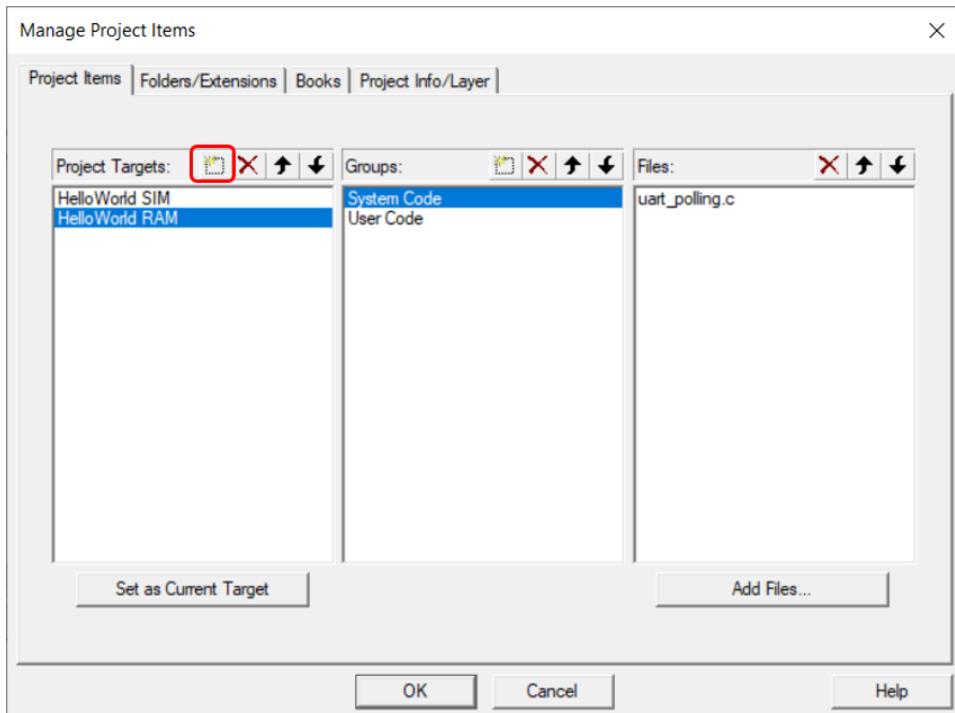


Figure 8.26: Keil IDE: Manage Project Items Window.

3. Switch your target to the newly created RAM target (See Figure 8.27).

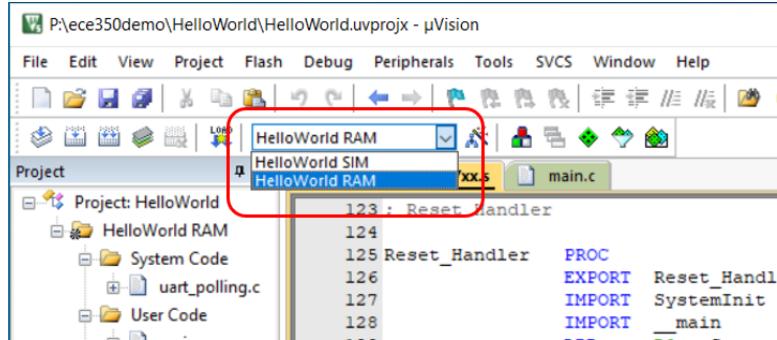


Figure 8.27: Keil IDE: Select HelloWorld RAM Target. Configure in-memory code execution as shown in Figure 8.28.

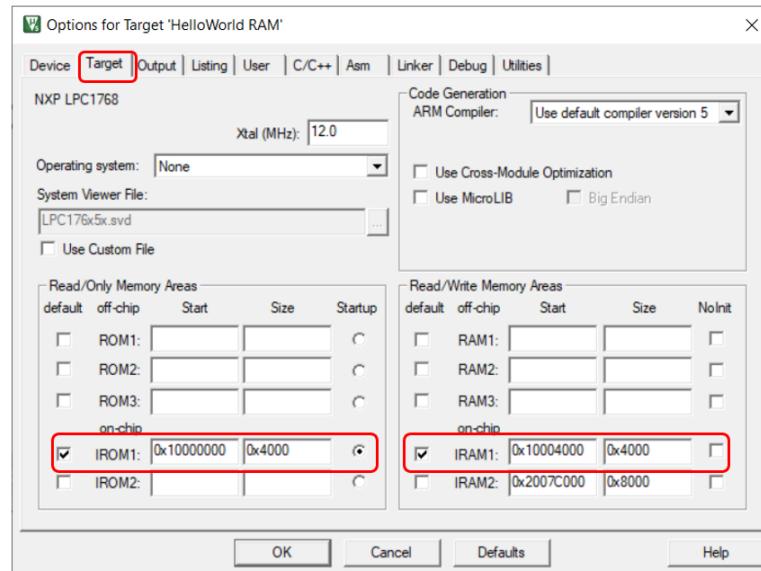


Figure 8.28: Keil IDE: Configure Target Options Target Tab for In-memory Execution.

The default image memory map setting is that the code is executed from the ROM (see Figure 8.13). 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 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 8.28 makes code starting address in RAM.

4. Select the Asm tab and input NO_CRP in the Conditional Assembly Control Symbols section as shown in Figure 8.29.

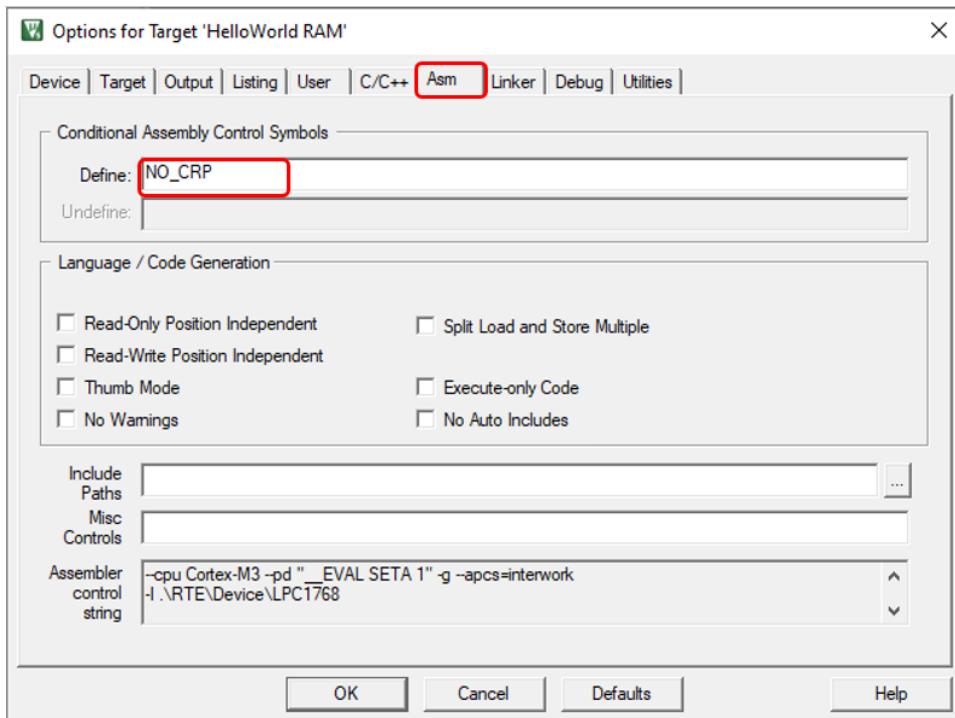


Figure 8.29: Keil IDE: RAM Target Asm Configuration.

5. 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 8.30) as a initialization file. An initialization file RAM.ini (see Listing B.2 in Appendix B) is needed to do the proper setting of SP, PC and vector table offset register.

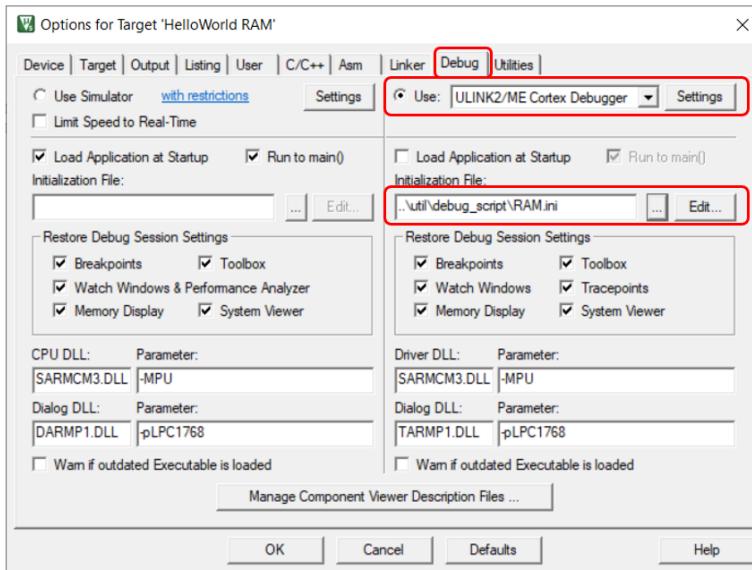


Figure 8.30: Keil IDE: Configure ULINK-ME Hardware Debugger.

6. Press the settings button beside the ULINK2/ME Cortex Debugger (see Figure 8.30) and select the Flash Download tab (see Figure 8.31). Remove the LPC17xx

IAP 512kB Flash algorithm to the Programming Algorithm field if it is there.

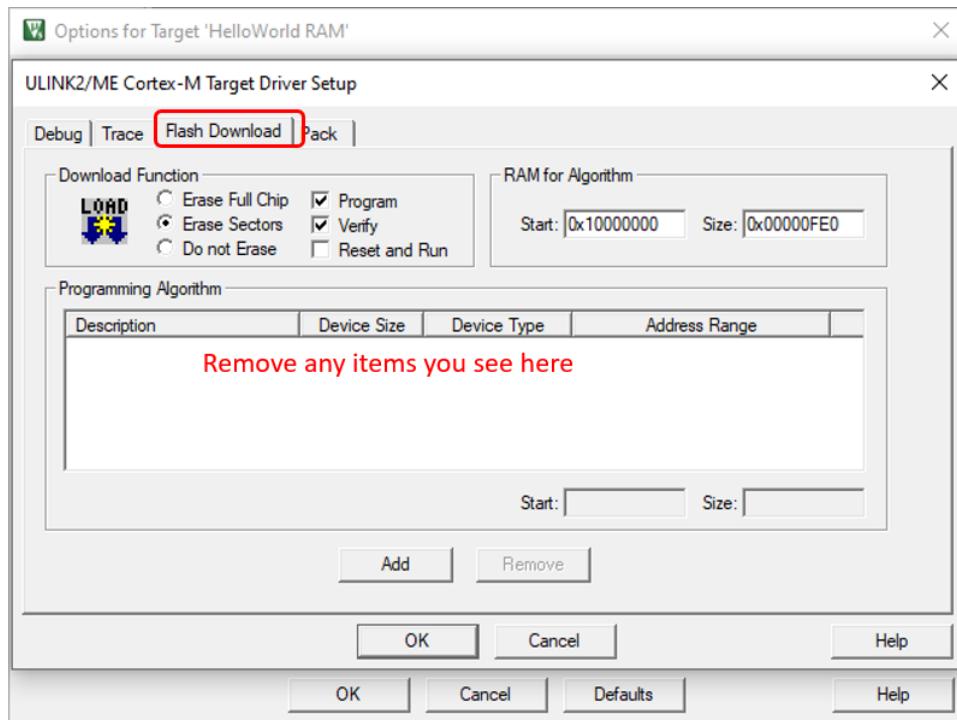


Figure 8.31: Keil IDE: Flash Download Programming Algorithm Configuration.

7. Select the Utilities tab and select the radio button beside “Use External Tool for Flash Programming” (see Figure 8.32).

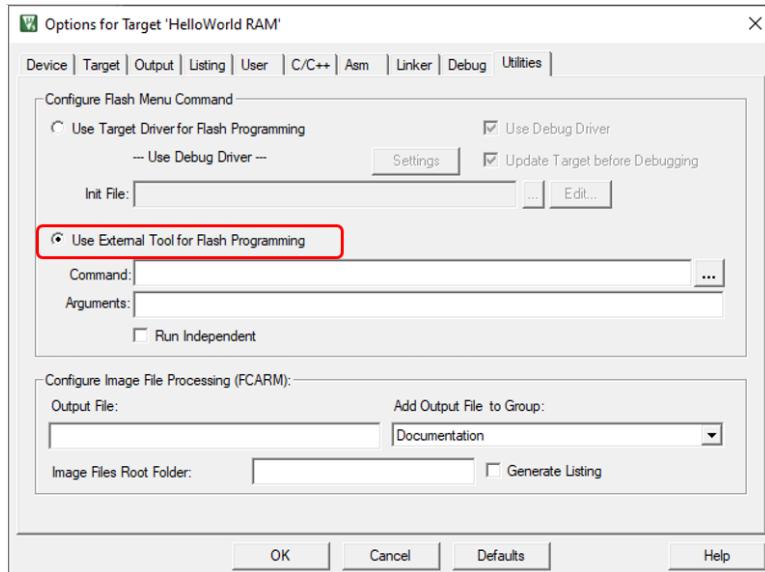


Figure 8.32: Keil IDE: Target Option Utilities Configuration for RAM Target.

8. Configure the Output tab so that the created executable will be put into a sub-folder of .\Objects\RAM (see Figure 8.33).

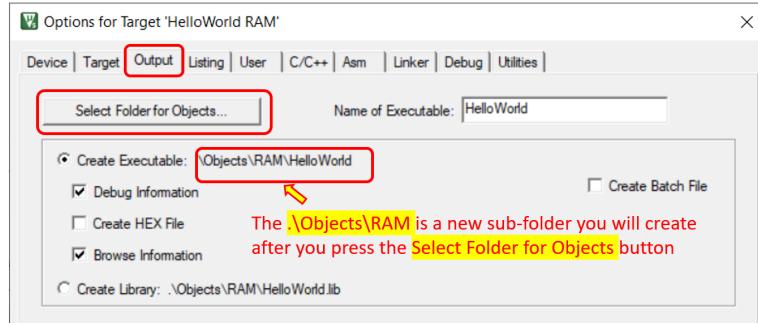


Figure 8.33: Keil IDE: Target Options Output Tab Configuration for RAM Target

9. Press the “OK” button to finish the target option configuration.
10. Build the RAM target by pressing the “Build” button (see Figure 8.17).
11. 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, open up the device manager and expand the Ports (COM & LPT) line (see Figure 8.34).

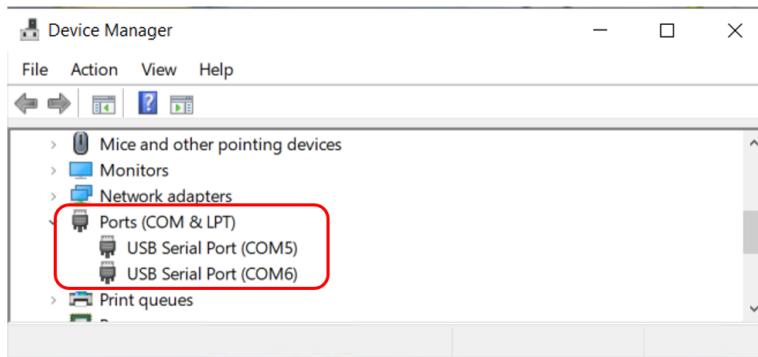


Figure 8.34: Device Manager COM Ports

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 8.35 and 8.36.

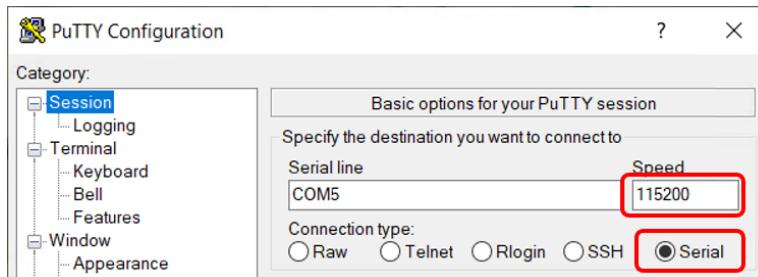


Figure 8.35: PuTTY Session for Serial Port Communication

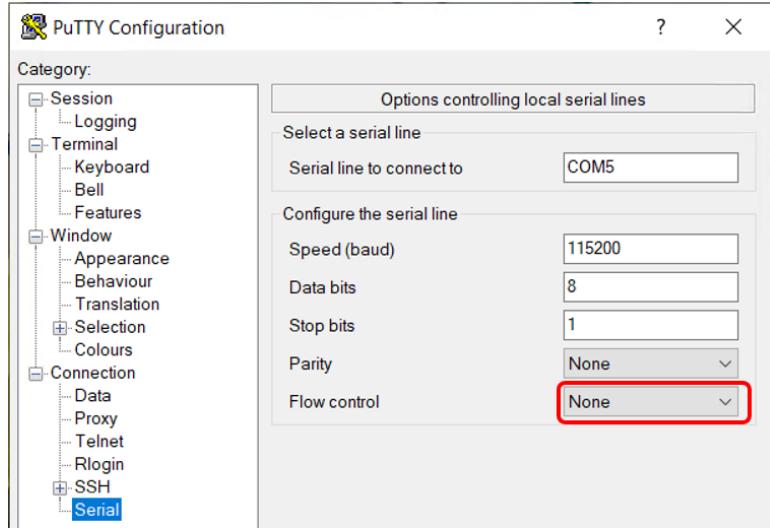


Figure 8.36: PuTTY Serial Port Configuration

12. 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.
13. 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 8.37).



Figure 8.37: PuTTY Output

8.8 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 8.39). 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 8.38). This will execute the code automatically without the need to press the physical reset button on the board. Add the LPC17xx IAP 512kB Flash algorithm to the Programming Algorithm field if it is not already there. Apply all the changes and close the target options configuration window.

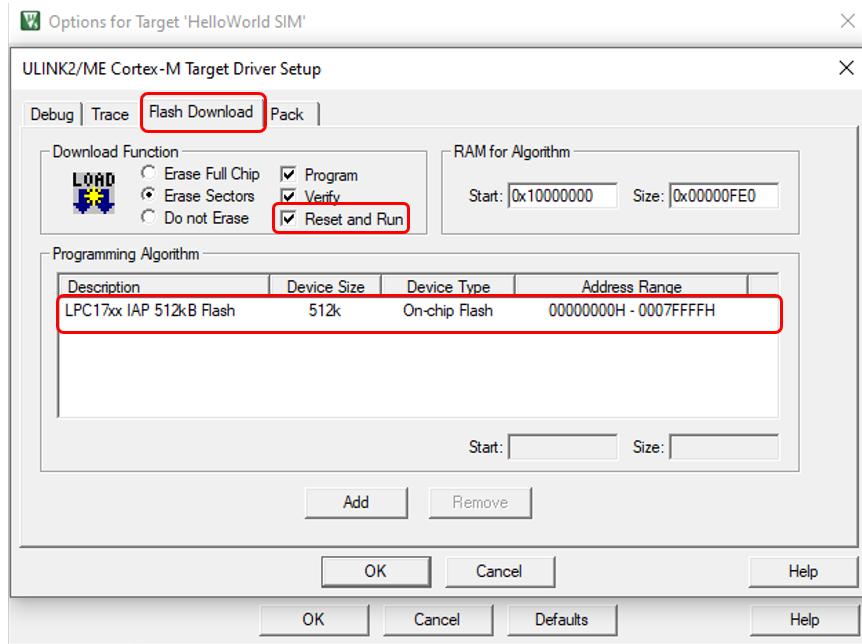


Figure 8.38: Flash Download Reset and Run Setting

To download the code to the on-chip ROM, click the “LOAD” button (see Figure 8.39). The download is through the ULINK-ME. The code automatically runs. You should see the output from PuTTY terminals.

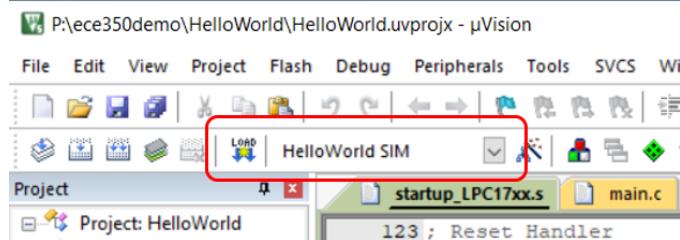


Figure 8.39: Keil IDE: Download Target to Flash

8.9 Create a Library Project

The uart polling code is not as convenient to use as the printf. We will show you how to build a simplified printf library so that printf will use the uart polling code to output to the UART #1 of the board. Note this printf is simplified version of the libc printf. It has small code size, hence has limited functionalities. But it is good enough for us to use to develop the lab project. Note the library does not execute by itself. It needs to be linked with an application project. We will show you how to do this in

Section 8.10.

8.9.1 Preparing Directory Structure

- Create a new folder and name it HelloWorld-Multi.
- Copy the entire HelloWorld application folder you created in previous steps to HelloWorld-Multi.
- Create RTX-Lib sub-folder inside HelloWorld-Multi folder.
 - Create src sub-folder inside the RTX-Lib.
 - Create bsp and libu sub-folders inside RTX-Lib\src folder.
 - Copy uart_polling.c into the bsp sub-folder.
 - Copy printf.c into the libu sub-folder (see Figure 8.40).
- Create include sub-folder inside HelloWorld-Multi folder.
 - Copy printf.h to include folder.
 - Create a bsp sub-folder inside include folder.
 - Create a sub-folder LPC1768 inside include\bsp folder.
 - Copy the following files to the include folder.
 - Copy the uart*.h files to LPC1768 folder.

Your directory structure should look like what is shown in Figure 8.40. Note we omitted the directory layout of the HelloWorld folder.

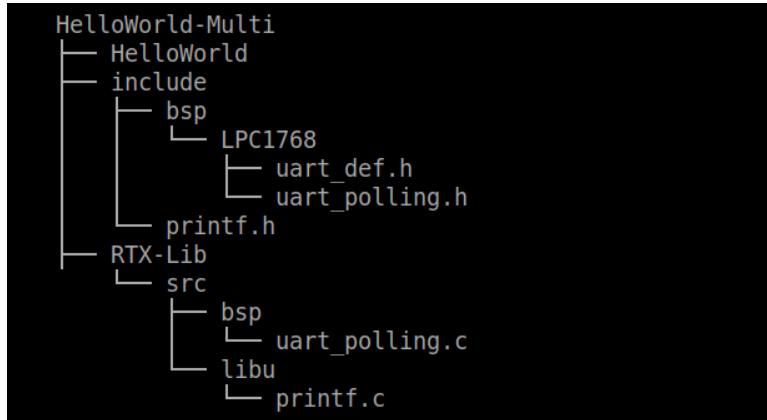


Figure 8.40: Directory Structure of a Multi-Project Workspace. The HelloWorld directory layout is omitted.

8.9.2 Create a New Library uVision Project

To create a new library uVision project, we follow the same steps of creating the new HelloWorld uVision project. Aside from giving the project a different name, the

big difference is that when asked for configuring the run-time environment, do not select anything, directly click the OK button (see Figure 8.41). Here are the steps.

- Click Project → New uVision Project (See Figure 8.2).
- Select NXP → LPC1700 Series → LPC176x → LPC1768 (See Figure 8.3).
- Do not select anything in the “Manage Run-Time Environment” pop-up window. Directly click the OK button. Compared with the HelloWorld application creation, the difference is that we should leave the CORE and Startup checkbox unchecked, which is the default setting (see Figure 8.41).

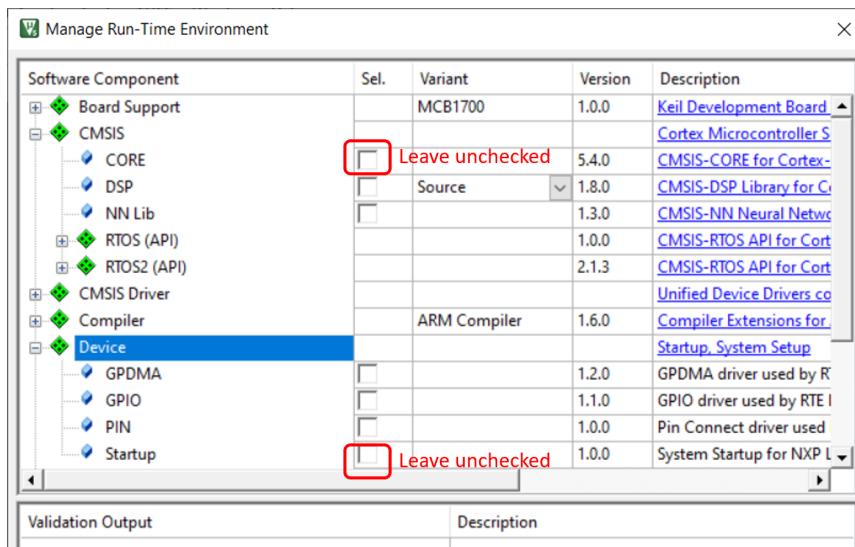


Figure 8.41: Directory Structure of a Multi-Project Workspace. The HelloWorld directory layout is omitted.

- Name the project file as `rtx-lib.uvprojx`.

8.9.3 Managing Library Project Component

You just finished creating another new project. The following steps are similar as creating your first HelloWorld project except that we are adding different source groups and we are adding different files to each source group. You can always refer Section 8.5 if you forget some of the steps.

1. Rename the Target
Rename the default “Target 1” to “RTX-Lib”.
2. Rename the Source Group
Rename the source group to “bsp”. Add a new source group and name it “libu”.

3. Add Source Code to a Source Groups

Add `uart_polling.c` to “bsp” source group. Add `printf.c` to “libu” source group. Your project will now look like Figure 8.42.

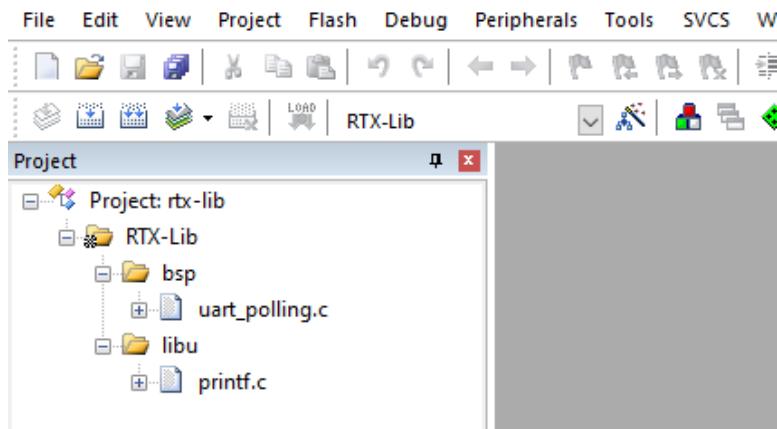


Figure 8.42: Keil IDE: A Library Project Profile

8.9.4 Configure a Library 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 8.12).
2. Configure the Target tab as shown in Figure 8.13. We want to use the default version 5 arm compiler. We also want remove the IRAM2 from the default setting. This is the same as the HelloWorld Application target tab configuration.
3. The most important step is to configure the output to be a library as shown in Figure 8.43

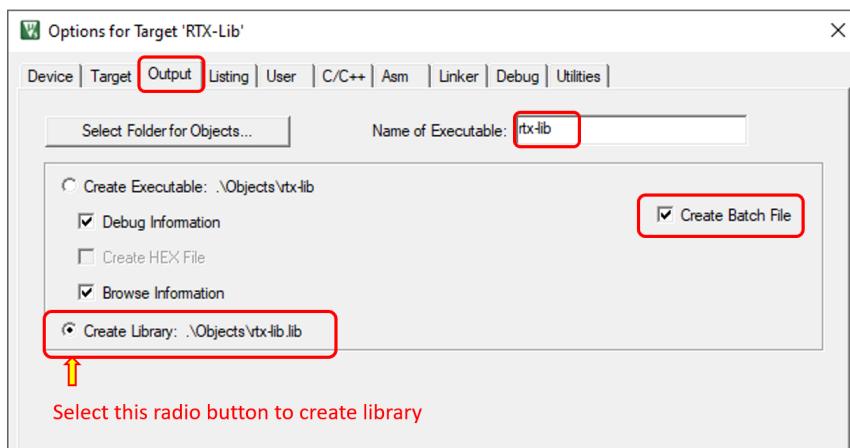


Figure 8.43: Keil IDE: Target Options Output Tab Library Creation Configuration

4. Configure the C/C++ tab as shown in Figure 8.44. In addition to the configuration you did for HellWorld application, you also need to specify the include path so the compiler knows where to find the header files. Note we moved the header files to a separate directory rather than having them in the same directory where the .c files are.

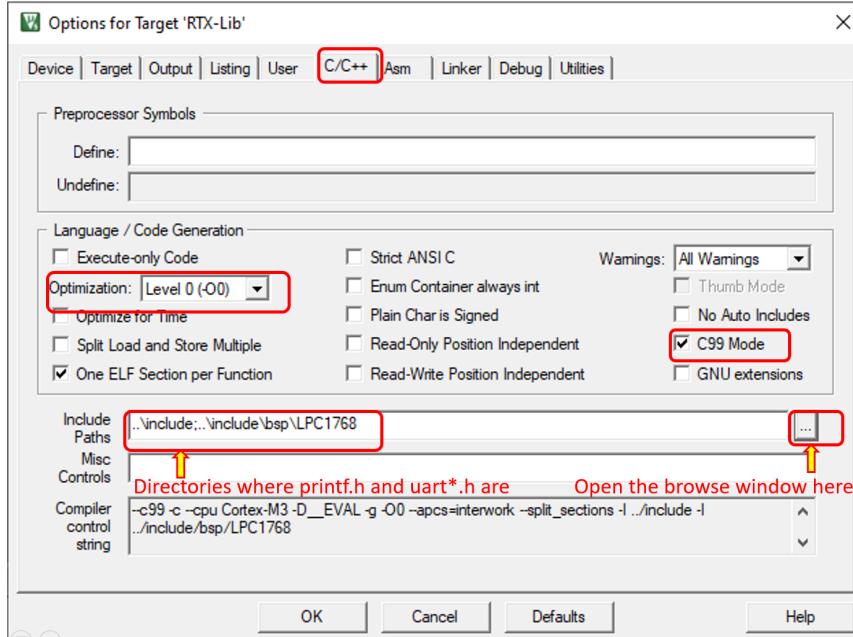


Figure 8.44: Keil IDE: Target Options C/C++ Tab Configuration

5. Configure the Linker tab the same way as you did for the HelloWorld application as shown in Figure 8.15.
6. Leave the rest of tab configurations as default.

8.9.5 Build the Library Target

To build the target, press the “Build” button (see Figure 8.17). If nothing goes wrong, the build output window at the bottom of the IDE will show a log similar like the one shown in Figure 8.45. Note the target is a .lib file and is in the default .\Objects directory. A library project cannot be executed. It needs to be linked with an application and the application generates an executable .axf file.

```
Build Output
Build started: Project: rtx-lib
*** Using Compiler 'V5.06 update 6 (build 750)', folder: 'C:\opt\Keil5\ARM\ARMCC\Bin'
*** Note: Rebuilding project, since 'Options->Output->Create Batch File' is selected.
Rebuild target 'RTX-Lib'
compiling printf.c...
compiling uart_polling.c...
Creating Library...
".\Objects\rtx-lib.lib" - 0 Error(s), 0 Warning(s).
Build Time Elapsed: 00:00:00
```

Figure 8.45: Keil IDE: Build Library Target

8.10 Create an Application that Links with a Library

Let's now open up the copied HelloWorld application and remove the System Code group and its associated files from the project explorer window. Then we add a new Lib Group and add the library file `rtx-lib.lib` in the RTX-Lib/Objects to the Lib group (see Figure 8.46).

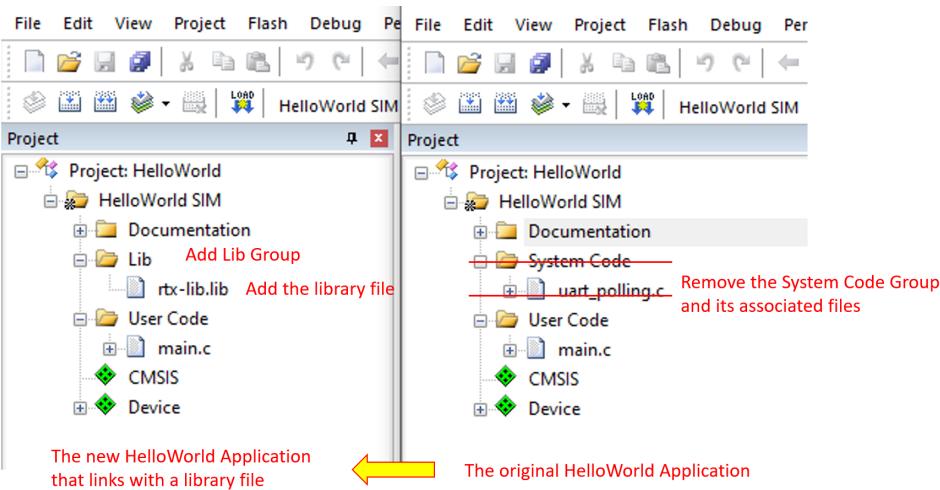


Figure 8.46: Keil IDE: HelloWorld Application that uses a Library

We then navigate to the `HelloWorld-Multi\HelloWorld\src` folder to remove `printf.[ch]` and `uart*. [ch]` files as shown in Figure 8.47.

Type	Name
C File	main
C File	printf
C File	uart_polling
H File	printf
H File	uart_def
H File	uart_polling

Figure 8.47: Keil IDE: Removing source code files from HelloWorld inside the Helloworld-Multi folder

We need to specify the include path in the C/C++ tab of the target option since now all header files are in a separate folder (see Figure 8.44). And we need to do this update for both the SIM and RAM targets. We are now ready to build the application, just press the build button as usual (see Figure 8.17). You will see the application is built and a `.axf` file is generated (see Figure 8.48). You may either use the debugger to run it inside a simulator or on the board as usual.

```

Build Output
Build started: Project: HelloWorld
*** Using Compiler 'V5.06 update 6 (build 750)', folder: 'C:
Build target 'HelloWorld SIM'
linking...
Program Size: Code=1756 RO-data=236 RW-data=8 ZI-data=608
".\Objects\SIM\HelloWorld.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed: 00:00:01

```

Figure 8.48: Keil IDE: Build Output of HelloWorld Application Linked with a Library

8.11 Create a Multi-Project Workspace

We now have two projects and they are related to each other. We want to put them into the same workspace. The uVision IDE can put multiple uVision projects into one workspace so that you can switch between projects easily. To create a uVision multi-project workspace, select Project → New Multi-Project Workspace (see Figure 8.49).

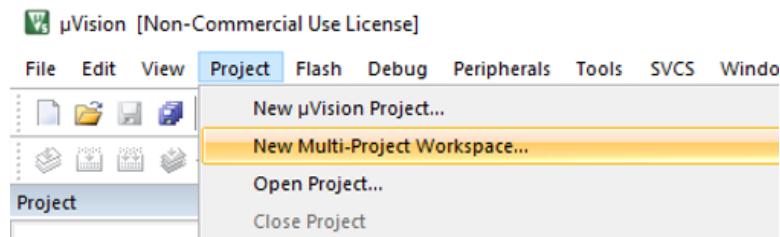


Figure 8.49: Keil IDE: Create a New Multi-Project Workspace Menu Item

You will be asked to save the multi-project profile file. Navigate to the HelloWorld-Multi folder and name the profile as HelloWorld-Multi.uvmpw. Then the “Create New Multi-Project Workspace” window appears for you to select individual projects you would like to add to the workspace (see Figure 8.50).

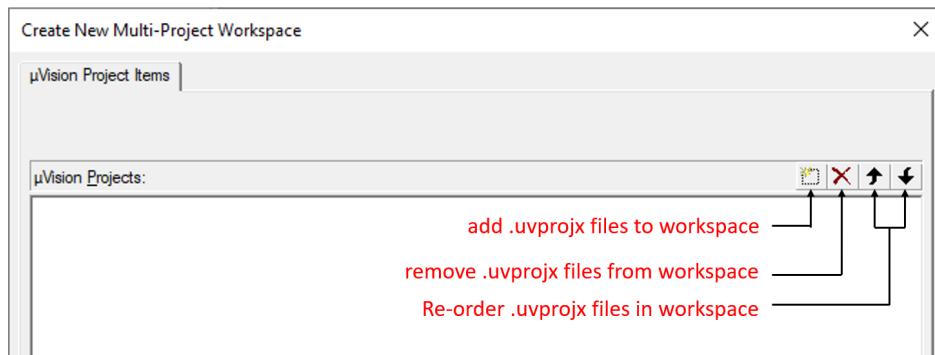


Figure 8.50: Keil IDE: Create a New Multi-Project Workspace Window

Let's first select the library project and then the application project (see Figure 8.51).

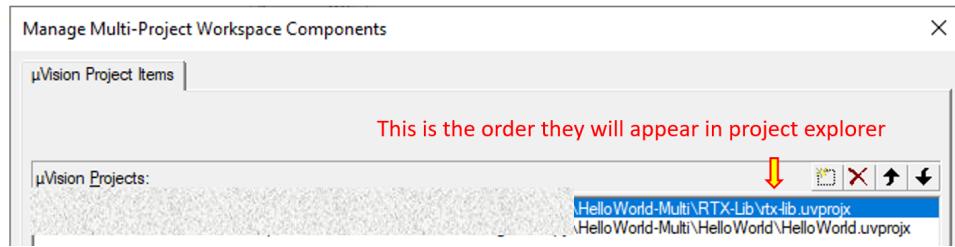


Figure 8.51: Keil IDE: Final New Multi-Project Workspace Window

Note that the order the projects are listed in the window is important. It determines the order of projects appearing in the project explorer window. The more important one is that it also determines the build order in batch build setup (see Section 8.12). After you press OK button, your setup should look like Figure 8.52.

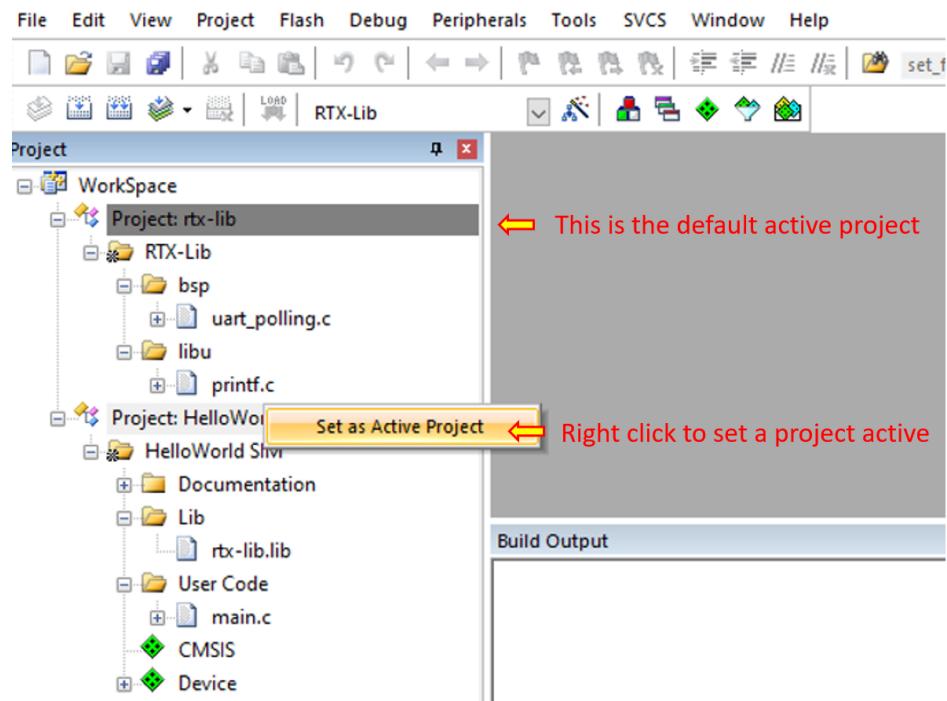


Figure 8.52: Keil IDE: Multi-Project Workspace Explorer

To take the full advantage of the IDE feature such as auto-suggesting function names and struct members, you need to make the project where the source code is associated with as the active project. There can only be one active project at a time. You will notice that when the library project is active, the debug button is greyed out. This is normal because a library is not an executable. The HelloWorld application that links with the library is an application that generates an executable. To make a project as the active project, right click the project to bring up the “Set as Active Project” context menu and select it (See Figure 8.52).

8.12 Batch Build

In the multi-project workspace, pressing the build button (see Figure 8.17) only builds the active project itself. Most of the time, we want to build all projects. This can be done by using the batch build feature of the IDE. To set up the batch build, select Project → Batch Setup (see Figure 8.53).

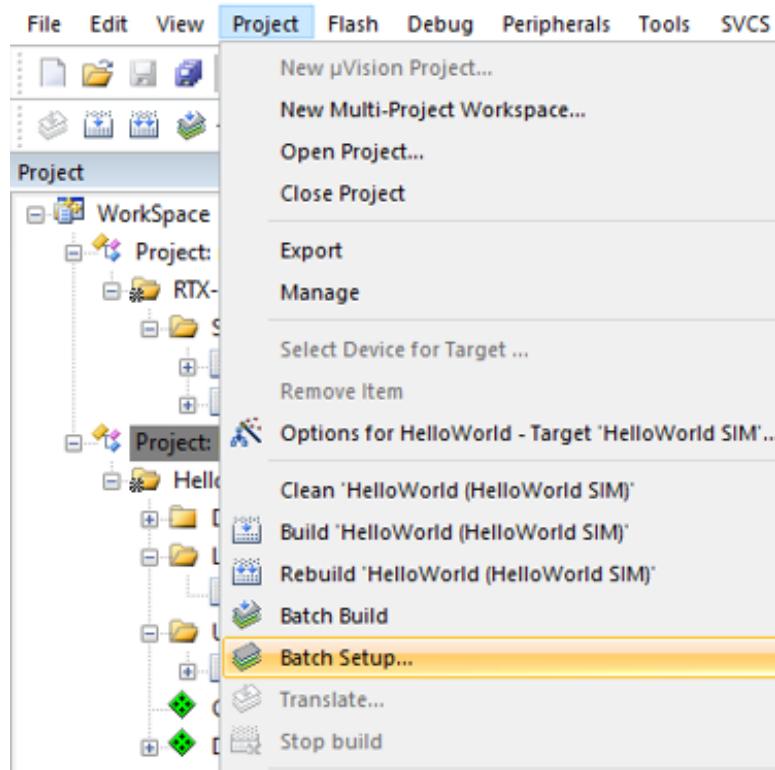


Figure 8.53: Keil IDE: Batch Setup Menu Item

We want to build all targets in the workspace (see Figure 8.54). Note the order of build sequence is important. We would like to first build the library, then the targets that are linked with the library. Reversing the order will make your application targets linked with the previously built library. Most of students will get frustrated when their newly built application code does not reflect the change they just made in the library. Not noticing the build order is culprit, one tends to start to move the Keil IDE down on the secret preferred IDE list. So we would like to bring your attention to this important point. At least this one is not the IDE's fault. When you are puzzled that why the change you made in the library does not appear in the application targets, batch build order is the first thing that you should check.



Figure 8.54: Keil IDE: Batch Setup Window

What if you want to change the build order? You can do so by bringing up the “Manage Multi-Project Workspace” context window to re-order the projects (see Figure 8.55).

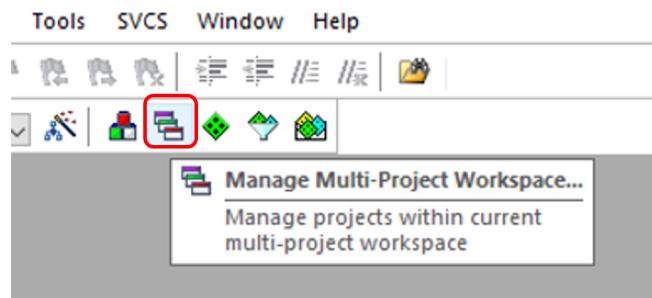


Figure 8.55: Keil IDE: Manage Multi-Project Workspace Button

To batch build the workspace, press the batch build button (see Figure 8.56). You can also select Project → Batch Build from the menu to achieve the same purpose.

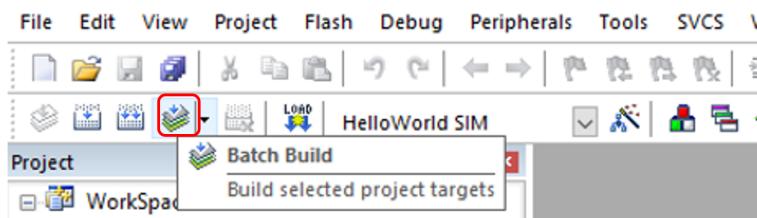


Figure 8.56: Keil IDE: Batch Build Button

You should watch carefully the build output message in the console window. Pay special attention to the last line of build summary (see Figure 8.57). If you have a compilation error of one of the projects, it shows up in the build summary. A commonly seen mistake is that the library failed to be built (due to syntax error), but students did not notice the error message. Then the application is successfully built, but linked with the previously built library. Hence students do not see the changes they made in the library project and once again too quickly move the IDE further down to their preferred IDE list.

```

Build Output

*** Using Compiler 'V5.06 update 6 (build 750)', folder: 'C:\opt\Keil5\ARM\ARMCC\Bin'
*** Note: Rebuilding project, since 'Options->Output->Create Batch File' is selected.
Rebuild Project 'rtx-lib' - Target 'RTX-Lib'
compiling printf.c...
compiling uart_polling.c...
creating Library...
".\Objects\rtx-lib.lib" - 0 Error(s), 0 Warning(s).
Build Time Elapsed: 00:00:03

*** Using Compiler 'V5.06 update 6 (build 750)', folder: 'C:\opt\Keil5\ARM\ARMCC\Bin'
Build Project 'HelloWorld' - Target 'HelloWorld SIM'
linking...
Program Size: Code=1756 RO-data=236 RW-data=8 ZI-data=608
".\Objects\SIM\HelloWorld.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed: 00:00:00

*** Using Compiler 'V5.06 update 6 (build 750)', folder: 'C:\opt\Keil5\ARM\ARMCC\Bin'
Build Project 'HelloWorld' - Target 'HelloWorld RAM'
linking...
Program Size: Code=1736 RO-data=236 RW-data=8 ZI-data=608
".\Objects\RAM\HelloWorld.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed: 00:00:01

Batch-Build summary: 3 succeeded, 0 failed, 0 skipped - Time Elapsed: 00:00:04

```

Figure 8.57: Keil IDE: Batch Build Output

8.13 Using the Library

To use the `printf` in the library we just built, just use the same syntax as the usual `libc printf` function. The `printf` library we have has limitations. It does not support floating point format output. The long format output also does not work properly. However we do not need them. The `%c`, `%d` and `%s` are what we need and they are supported. The `printf` prints to the second serial port by polling. Let's add some code to the `main.c` (see Figure 8.58) to see what it does. Without a memory allocator, we directly write data to the physical memory locations that we know that are free. Note the '`\n`' behaves differently in simulator terminal and the putty terminal (when using the board). The simulator automatically adds a `\r` when '`\n`' presents. The putty terminal does not.

```

37
38 #include <LPC17xx.h>
39 #include "uart_def.h"
40 #include "uart_polling.h"
41 #include "printf.h"
42
43 int main() {
44     SystemInit();
45     uart0_init();
46     uart1_init();
47     uart0_put_string("UART0 - Howdy!\r\n");
48     uart1_put_string("UART1 - Hello World!\r\n");
49     init_printf(NULL, putc);
50
51     char *p = (void *) 0x2007c000;
52     char *ptr = p;
53     for ( int i = 0; i < 26; i++ ) {
54         *ptr++ = 'A' + i;
55     }
56     *ptr = '\0';
57     printf("p = 0x%lx, ptr = 0x%lx\n", p, ptr);
58     for ( int i = 0; i < 3; i++ ) {
59         printf("i = %d: %s\r\n", i, p);
60     }
61     return 0;
62 }

```

Figure 8.58: The main.c code that uses printf

The output of the program by using the simulator is shown in Figure 8.59.

```

UART #2
UART1 - Hello World!
p = 0x2007c000, ptr = 0x2007c004
i = 0: ABCD
i = 1: ABCD
i = 2: ABCD

```

Figure 8.59: Keil IDE: demonstration of printf using simulator

The output of the program by using the simulator is shown in Figure 8.60.

```

COM9 - PuTTY
UART1 - Hello World!
p = 0x2007c000, ptr = 0x2007c004
i = 0: ABCD
i = 1: ABCD
i = 2: ABCD

```

Figure 8.60: Keil IDE: demonstration of printf on board

Now you may go to Chapter 3 to start adding a memory allocator to your RTX-Lib.

8.14 Errata

1. Pages 40 - 41, the order of items 2 and 3 is swapped.
2. Page 42, added new items 5 and 6.
3. Page 51, added new items 9 and 10.

Chapter 9

Programming MCB1700

9.1 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 9.1 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].

9.2 ARM Architecture Procedure Call Standard (AAPCS)

The AAPCS (ARM Architecture Procedure Call Standard) defines how subroutines can be separately written, separately compiled, and separately assembled to work together. The C compiler follows the AAPCS to generate the assembly code. Table 9.2 lists registers used by the AAPCS.

Registers R0-R3 are used to pass parameters to a function and they are not preserved. The compiler does not generate assembler code to preserve the values of

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

Table 9.1: Assembler instruction examples

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

Table 9.2: 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.

9.3 Cortex Microcontroller Software Interface Standard (CMSIS)

The Cortex Microcontroller Software Interface Standard (CMSIS) was developed by ARM. It provides a standardized access interface for embedded software products (see Figure 9.1). This improves software portability and re-usability. It enables soft-

ware solution suppliers to develop products that can work seamlessly with device libraries from various silicon vendors [3].

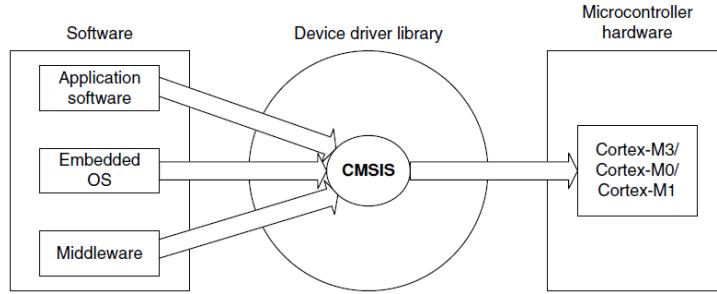


Figure 9.1: Role of CMSIS[6]

The CMSIS uses standardized methods to organize header files that makes it easy to learn new Cortex-M microcontroller products and improve software portability. With the `<device>.h` (e.g. `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.

9.3.1 CMSIS files

The CMSIS is divided into multiple layers (See Figure 9.2). For each device, the MCU vendor provides a device header file `<device>.h` (e.g., `LPC17xx.h`) which pulls in additional header files required by the device driver library and the Core Peripheral Access Layer (see Figure 9.3).

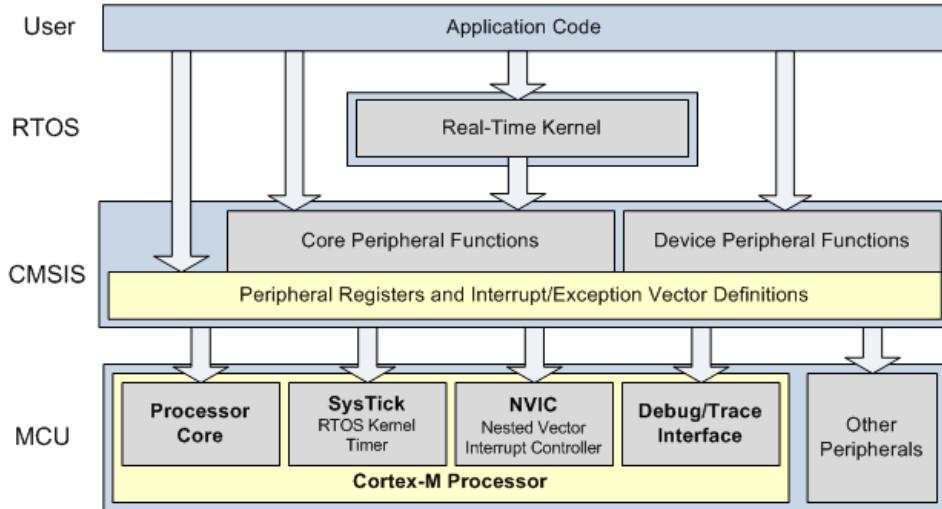


Figure 9.2: CMSIS Organization[3]

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 9.1.

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

Listing 9.1: 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 9.3.3).

9.3.2 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 9.4). 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
```

9.3.3 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`).

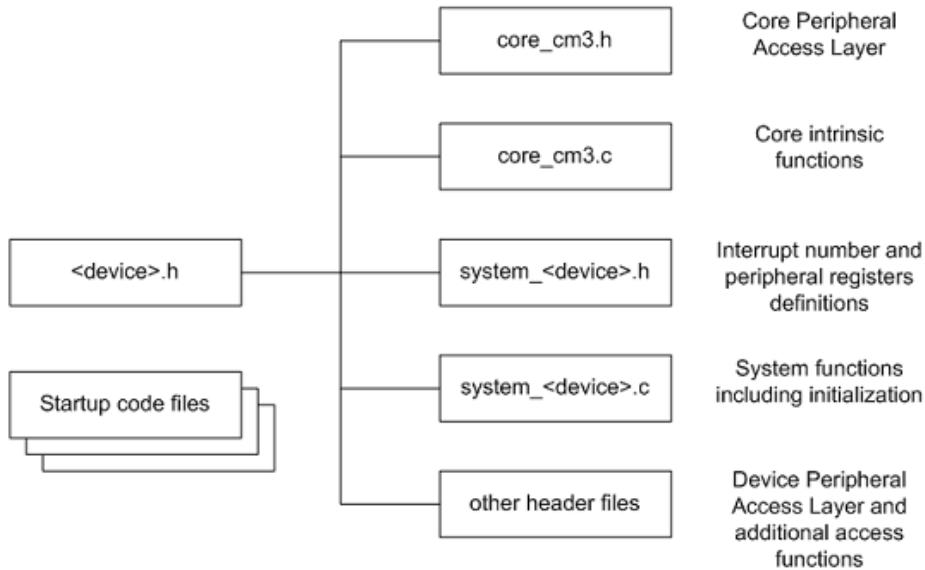


Figure 9.3: CMSIS Organization[3]

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 9.4: CMSIS NVIC Functions[3]

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	uint32_t __get_CONTROL(void)
	Write	void __set_CONTROL(uint32_t value)
MSP	Read	uint32_t __get_MSP(void)
	Write	void __set_MSP(uint32_t value)
PSP	Read	uint32_t __get_PSP(void)
	Write	void __set_PSP(uint32_t value)

Table 9.3: 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,
}

```

9.3.4 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 9.3 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.

9.3.5 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

```

9.4 Accessing C Symbols from Assembly

Both inline and embedded assembly are supported in MDK5. We will mainly using embedded assembly in this lab. To write an embedded assembly function, you need to use the `__asm` keyword. For example the the function “`embedded_asm_function`” in Listing 9.3 is an embedded assembly function. You can only put assembly instructions inside this function.

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

- A global variable defined in C. In Listing 9.2, we have two C global variables `g_pcb` and `g_var`. We can use the `__cpp` to access them as shown in Listing 9.3. 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 9.2: Example of accessing C global variables from assembly. The C code.

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

Listing 9.3: Example of accessing global variable from assembly

- A C function. In Listing 9.4, `a_c_function` is a function written in C. We can invoke this function by using the assembly language.

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

```
    ;.....  
}
```

Listing 9.4: Example of accessing c function from assembly

- A constant expression in the range of 0 – 255 defined in C. In Listing 9.5, `g_flag` is such a constant. We can use `MOV` instruction on it. Note the `MOV` instruction only applies to immediate constant value in the range of 0 – 255.

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

Listing 9.5: Example of accessing constant from assembly

You can also use the `IMPORT` directive to import a C symbol in the embedded assembly function and then start to use the imported symbol just as a regular assembly symbol (see Listing 9.6).

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

Listing 9.6: Example of using `IMPORT` directive to import a C symbol.

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

9.5 SVC Programming: Writing an RTX API Function

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

The basic idea is that when a function in RTX API is called from the user level, this function will trigger an `SVC` instruction. The `SVC_Handler`, which is the CM-SIS standardized exception handler for `SVC` exception will then invoke the kernel function that provides the actual service (see Figure 9.5). Effectively, the RTX API function is a wrapper that invokes `SVC` exception handler and passes corresponding kernel service operation information to the `SVC` handler.

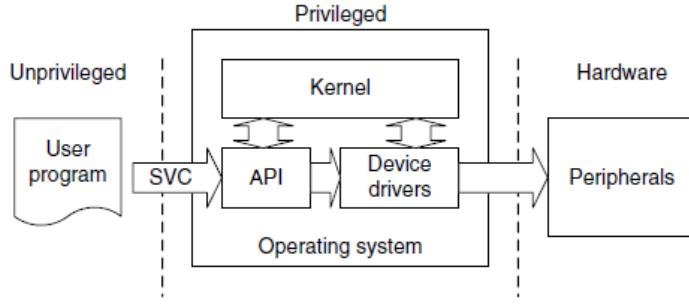


Figure 9.5: SVC as a Gateway for OS Functions [6]

The kernel needs to know which system call maps to which kernel functions call. We can use different SVC number for different system calls if there are sufficient number of SVC numbers the instruction supports. We can use a single SVC number as system call entry to kernel and then use register R12 to encode the information for the kernel to find the corresponding kernel function for a system call.

To generate an SVC instruction, there are two methods. One is to directly program at assembly instruction level. The other one is to use arm compiler-specific keywords to let the compiler generate SVC instructions.

9.5.1 Programming in Assembly Language

We can use the embedded assembly to write SVC instruction directly. Assume we want to create a system call `void *sys_func(size_t size)` and the kernel counter part of this system call is `void *k_sys_func(size_t t)`. One way of implementation is to assign an SVC number, say `0x0` to `sys_func`. Listing 9.7 shows one implementation.

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

Listing 9.7: Code Snippet of `sys_func`

The corresponding kernel function is a C function `k_sys_func`. This function entry point is loaded to register `r12`. Then `SVC 0` causes an SVC exception with immediate number 0. In the SVC exception handler, we can then branch with link and exchange to the address stored in `r12`. Listing 9.8 is an excerpt of `SVC_Handler` written in assembly.

```

__asm void SVC_Handler(void) {
    MRS R0, PSP
    ; code that may use R0-R3 and R12 for computation
    ; 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
    ; code to handle return value of C function omitted
    MVN LR, #:NOT:0xFFFFFFF; set EXC_RETURN, thread mode, PSP
    BX LR
}

```

Listing 9.8: Code Snippet of SVC_Handler

9.5.2 Programming in C with ARM Compiler Keywords

The second method is to ask the compiler to generate the SVC instruction from C code. The ARM compiler provides two keywords. One is `__svc` and the other is `__svc_indirect`.

The `__svc` keyword declares a *SuperVisor Call* (SVC) function taking up to four integer-like arguments and returning up to four results in a `value_in_regs` structure [1]. The immediate value used in the SVC needs to be provided to the compiler. The syntax of it is as follows:

```

__svc(int svc_num) return_type function_name([argument-list]);

```

Listing 9.9: `__svc` compiler keyword syntax

This causes function invocations to be compiled inline as an AAPCS-compliant operation that behaves similarly to a normal call to a function. The `__value_in_regs` qualifier can be used to specify a small structure of up to 16 bytes in registers for return, rather than by the usual structure-passing mechanism defined in the AAPCS.

The following C code creates a system call of `void *sys_call(size_t size)`, the compiler generates the SVC 0 instruction. This is the mechanism we will use in lab.

```

#define SVC_SYS_CALL 0
__svc(0) void *sys_call(size_t size);

```

Listing 9.10: `__svc` compiler keyword example

Another way of making the compiler to generate SVC instructions is to use the `__svc_indirect` keyword which passes an operation code to the SVC handler in

`r12` [1]. 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]);
```

Listing 9.11: `__svc_indirect` compiler keyword syntax

The SVC handler must make use of the `r12` value to select the required operation. Using the example system call `void *sys_call(size_t size)` again, the following code is relevant to the implementation of the function so that it will use SVC 0 to enter the SVC handler with the corresponding kernel function address loaded in R12.

```
#define __SVC_0 __svc_indirect(0)
extern void *k_sys_call(size_t size);
#define sys_call(size) __sys_call((U32)k_sys_call, size);
extern void *_sys_call(U32 p_func, size_t size) __SVC_0;
```

Listing 9.12: `__svc_indirect` compiler keyword example

The compiler generates two assembly instructions

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

The `SVC_handler` in Listing 9.8 then can be used to handle the SVC 0 exception.

9.6 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 9.13, 9.14 and 9.15 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_CTL 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 9.13: 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 9.14: 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"
#ifndef 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 reference  

//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) within 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 Divisionor 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 Available
        /* 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 {
#endif 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 */
#endif DEBUG_0
    uart1_put_string("Should not get here!\n\r");
#endif // DEBUG_0
    return;
}
}

```

Listing 9.15: UART0 IRQ Sample Code uart_irq.c

Listings 9.16 and 9.17 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 9.16: 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 Division 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 9.17: UART0 IRQ Sample Code `uart_polling.c`

9.7 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 9.18 and 9.19 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 9.18: 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 matches the value in the TC,
generate an interrupt.
Reset on MR0: Reset TC if MR0 matches it.
*/
pTimer->MCR = BIT(0) | BIT(1);

g_timer_count = 0;

```

```

/* Step 4.4: CMSIS 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 interrupt, see section 21.6.1 on pg 493 of LPC17XX_UM */
    LPC_TIM0->IR = BIT(0);

    g_timer_count++ ;
}

```

Listing 9.19: Timer0 IRQ Sample Code timer.c

Chapter 10

Keil MCB1700 Hardware Environment

10.1 MCB1700 Board Overview

The Keil MCB1700 board is populated with NXP *LPC1768* Microcontroller. Figure 10.1 shows the important interface and hardware components of the MCB1700 board.

Figure 10.2 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 10.3 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.

10.2 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 10.4 is the simplified block diagram of the Cortex-M3 processor [6]. 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

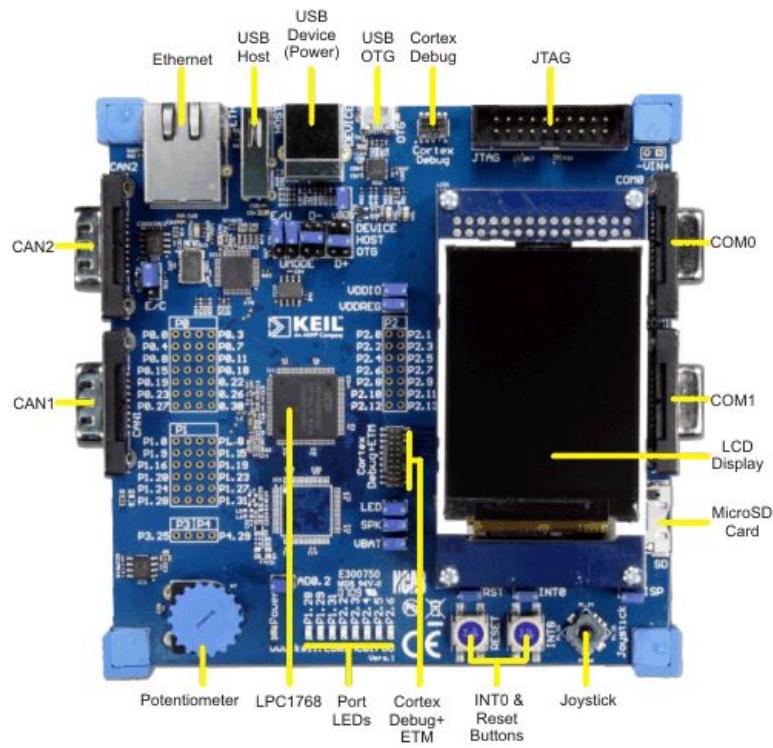


Figure 10.1: MCB1700 Board Components [2]

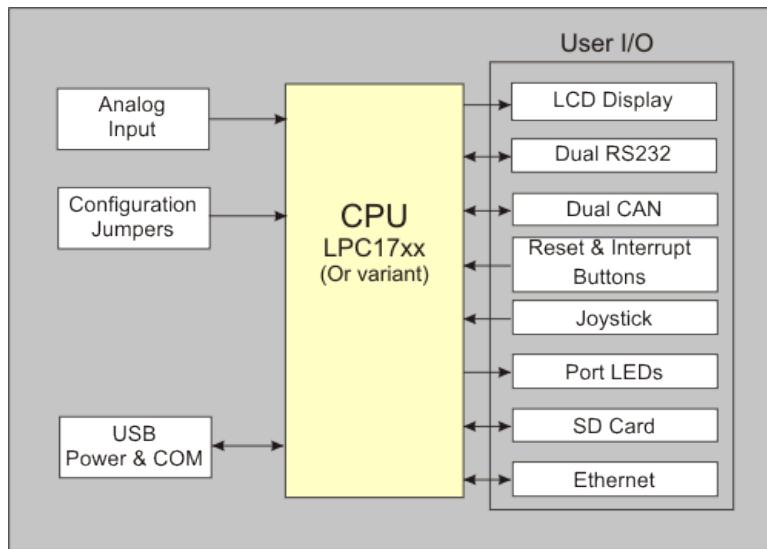


Figure 10.2: MCB1700 Board Block Diagram [2]

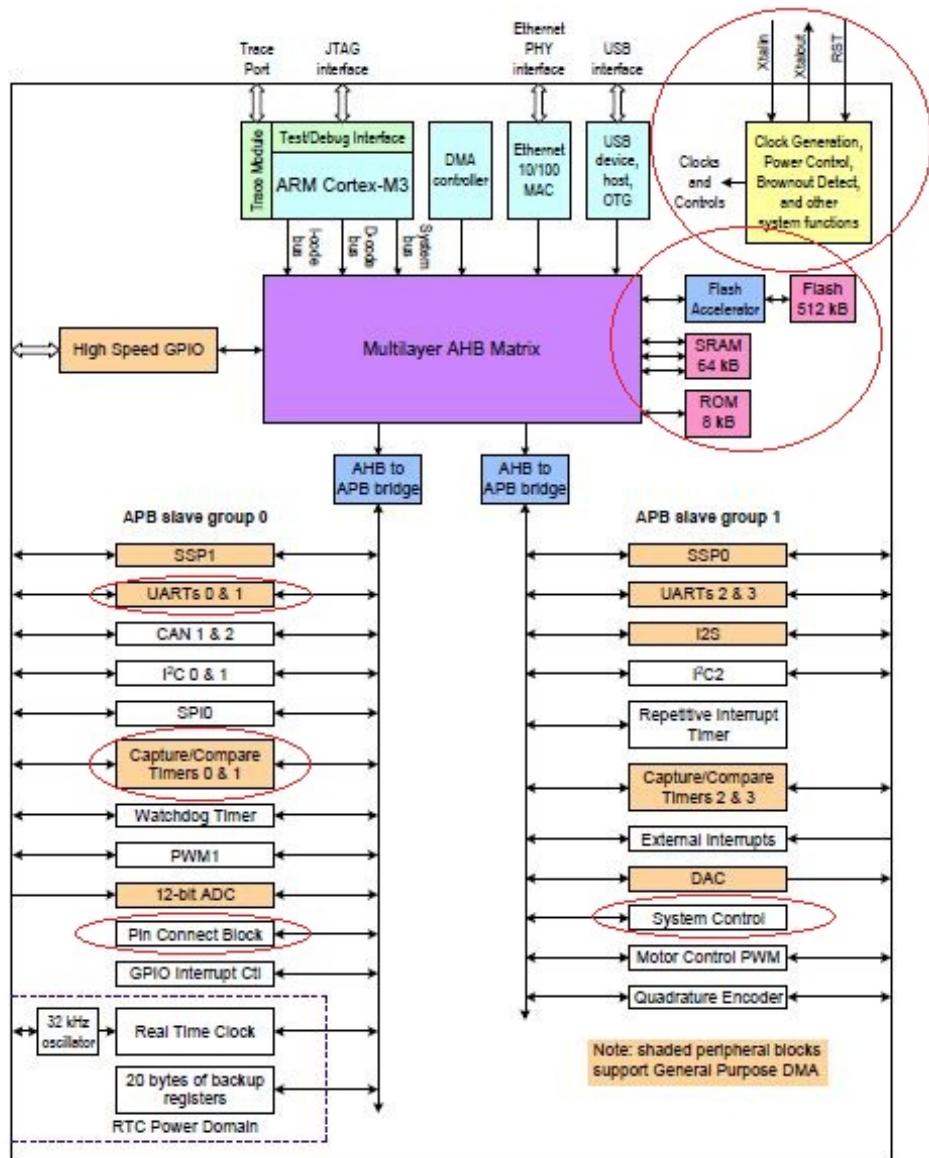


Figure 10.3: LPC1768 Block Diagram. The circled blocks are the ones that we will use in the lab project.

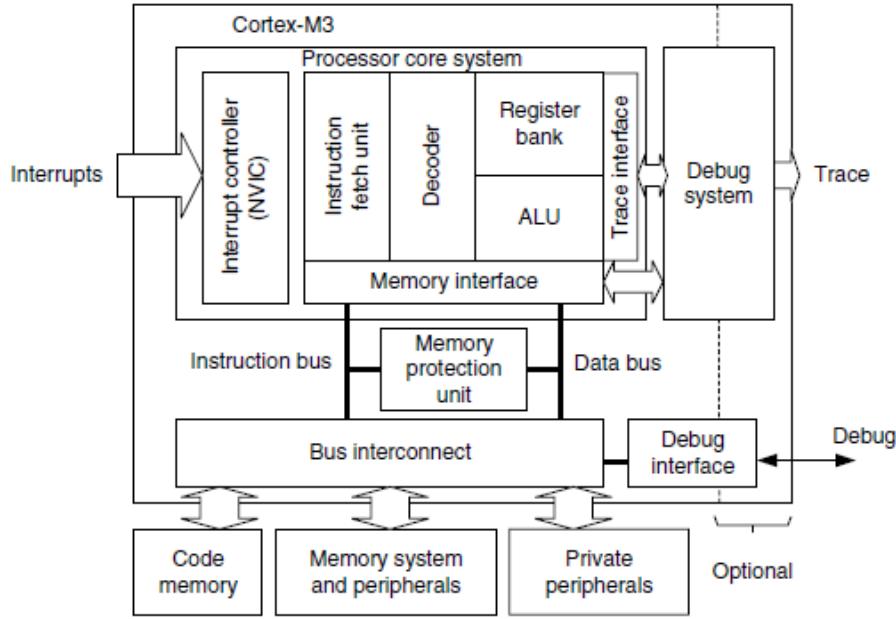


Figure 10.4: Simplified Cortex-M3 Block Diagram[6]

debugging components which provides debugging features such as breakpoints and watchpoints.

10.2.1 Registers

The processor core registers are shown in Figure 10.5. 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.

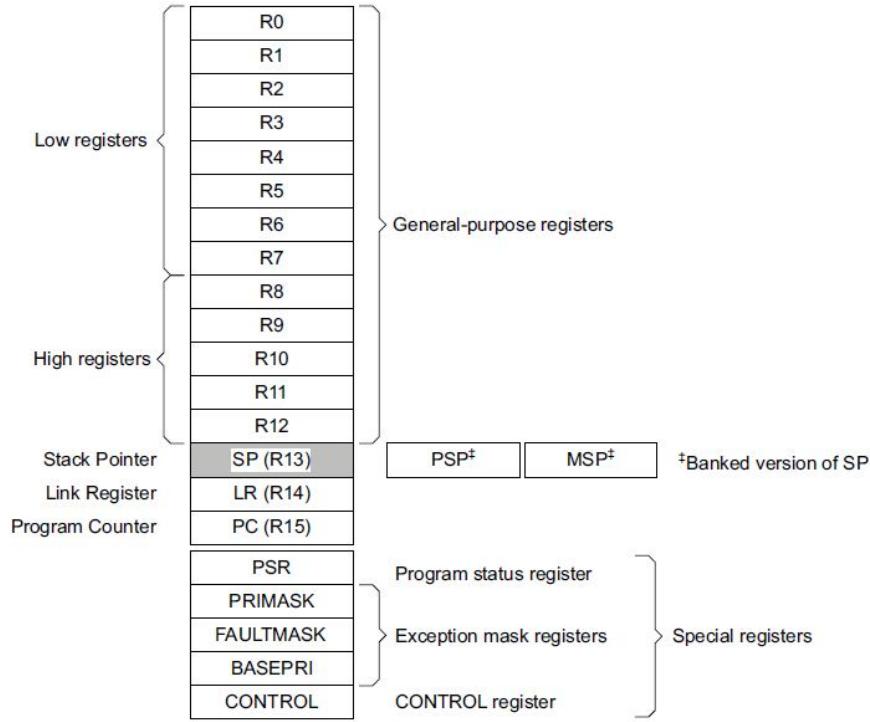


Figure 10.5: Cortex-M3 Registers[4]

- R15(PC) is the program counter. It can be written to control the program flow.
- 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.

10.2.2 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 10.6 illustrate the mode and privilege level of the processor.

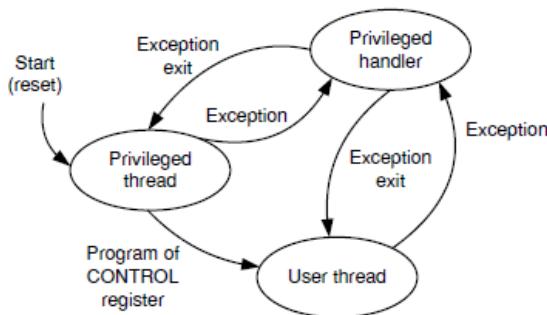


Figure 10.6: Cortex-M3 Operating Mode and Privilege Level[6]

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 initialization phase and also context switching between privileged processes and unprivileged processes.

10.2.3 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 10.1 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 10.1: Summary of processor mode, execution privilege level, and stack use options

10.3 Memory Map

The Cortex-M3 processor has a single fixed 4GB address space. Table 10.2 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 10.2: 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.

10.4 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.

10.4.1 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 10.3 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.

10.4.2 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 10.3: LPC1768 Exception and Interrupt Table

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

Figure 10.7 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:

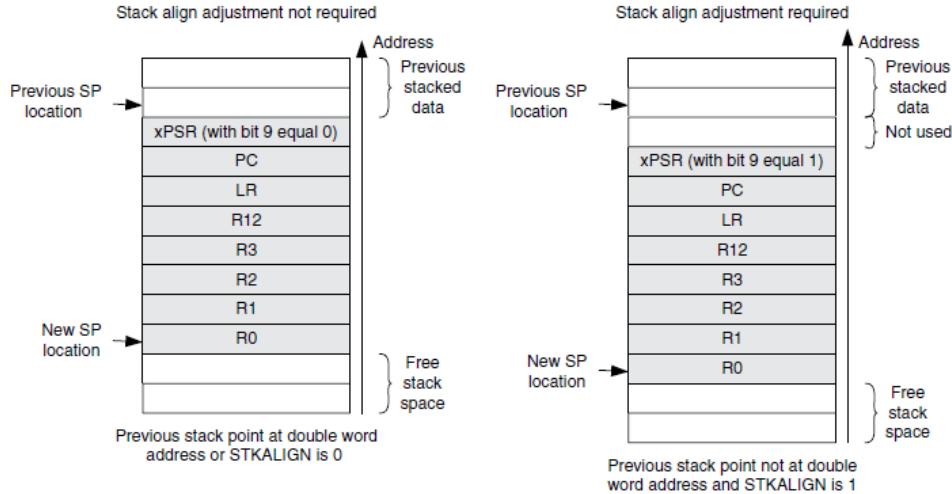


Figure 10.7: Cortex-M3 Exception Stack Frame [6]

- SP: The SP (MSP or PSP) will be updated to the new location during stack-ing. Stacking from the privileged/unprivileged thread to the first level of the exception handler uses the MSP/PSP. During the execution of excep-tion 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 com-pletes 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.

10.4.3 EXC_RETURN Value

EXC_RETURN is the value loaded into the LR on exception entry. The exception mech-anism relies on this value to detect when the processor has completed an exception handler. The EXC_RETURN bits [31 : 4] is always set to 0xFFFFFFFF by the processor. When this value is loaded into the PC, it indicates to the processor that the excep-tion is complete and the processor initiates the exception return sequence. Table 10.4 describes the EXC_RETURN bit fields. Table 10.5 lists Cortex-M3 allowed EXC_RETURN values.

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

Table 10.4: EXC_RETURN bit fields [6]

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

Table 10.5: EXC_RETURN Values on Cortex-M3

10.4.4 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.

10.5 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.

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 Debugger Initialization Files

The SIM.ini file in the starter code can be found in Listing B.1. The simulator by default does not detect the second bank of RAM (i.e. IRAM2 in the Target page of the Target option window), which starts 0x2007C000 and ends at 0x20083FFF. We use the MAP command to specify the memory access rights of this range of memory.

```
MAP 0x2007C000, 0x20083FFF READ WRITE // set up IRAM2 memory access
```

Listing B.1: The SIM.ini file

The RAM.ini file in the starter code can be found in Listing B.2. 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) {
    SP = _DWORD(0x10000000); // Setup Stack Pointer
    PC = _DWORD(0x10000004); // Setup Program Counter
    XPSR = 0x01000000; // Set Thumb bit
    _DWORD(0xE000ED08, 0x10000000); // Setup Vector Table Offset Register
    _DWORD(0x400FC0C4, _DWORD(0x400FC0C4) | 1<<12); // Enable ADC Power
    _DWORD(0x40034034, 0x00000F00); // Setup ADC Trim
}
LOAD %L INCREMENTAL // Download

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

Listing B.2: The RAM.ini file

Bibliography

- [1] Arm compiler v5.06 for uvision armcc user guide.
<https://www.keil.com/support/man/docs/armcc/default.htm>.
- [2] MCB1700 User's Guide. <http://www.keil.com/support/man/docs/mcb1700>.
- [3] MDK Primer. <http://www.keil.com/support/man/docs/gsac>.
- [4] LPC17xx User Manual, Rev2.0, 2010.
- [5] Donald E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley, third edition, 1997.
- [6] J. Yiu. *The Definitive Guide to the ARM Cortex-M3*. Newnes, 2009.