# 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, May 18, 2021

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

# Contents

# List of Tables

# List of Figures

# Preface

## Who Should Read This Lab Manual?

This lab manual is written for students who will design and implement a small Real-Time Executive (RTX) for Keil MCB1700 board populated with an NXP LPC1768 microcontroller.

## What is in This Lab Manual?

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

Part I describes the lab administration policies.

Part II is the project description. We break the project into the following 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 manufacture, for shipping the boards in a timely manner.
- Imperas Software for providing us one evaluation license to experiment with their software tools during the lab development.

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

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

# Part I

# Lab Administration

# Lab Administration Policy

## Group Lab Policy

- **Group Size.** All labs are done in groups of *four*. A group size of less than four is not recommended. There is no reduction in project deliverables regardless the size of the project group. The Learn system (http://learn.uwaterloo.ca) is used to 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 your 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.*

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

## Project Grading Policy

- **Project Grading Procedure.** The project is graded by automated testing framework. For each deliverable, we publish a small set of testing cases. We require students to pass these testing cases before they submit. If you are not able to pass these testing cases, then 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 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*

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

*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/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 image [1]. 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.

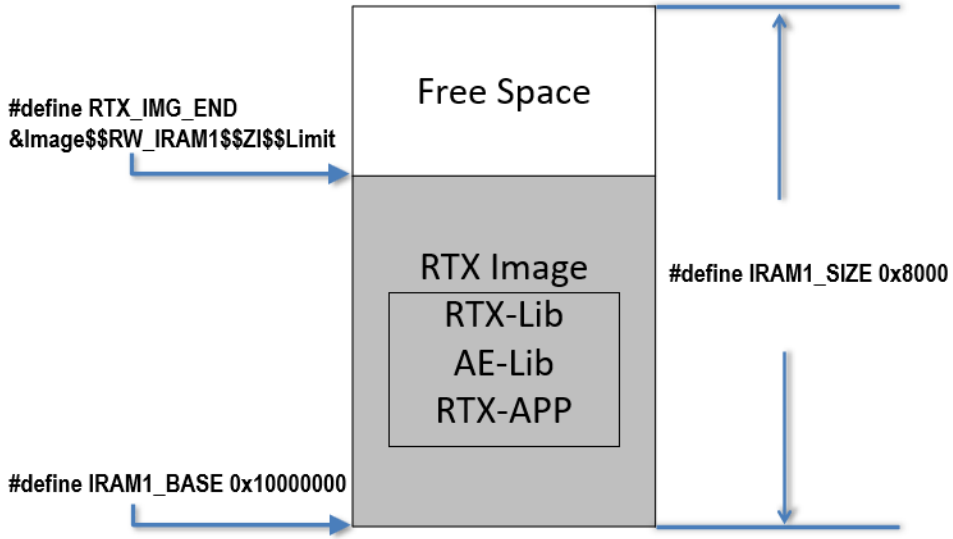


Figure 3.2: NXP LPC1768 IRAM1 Memory Execution View. The RTX Image has two libraries (RTX-Lib and AE-Lib) built into it together with the application (RTX-App) that uses the libraries.

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

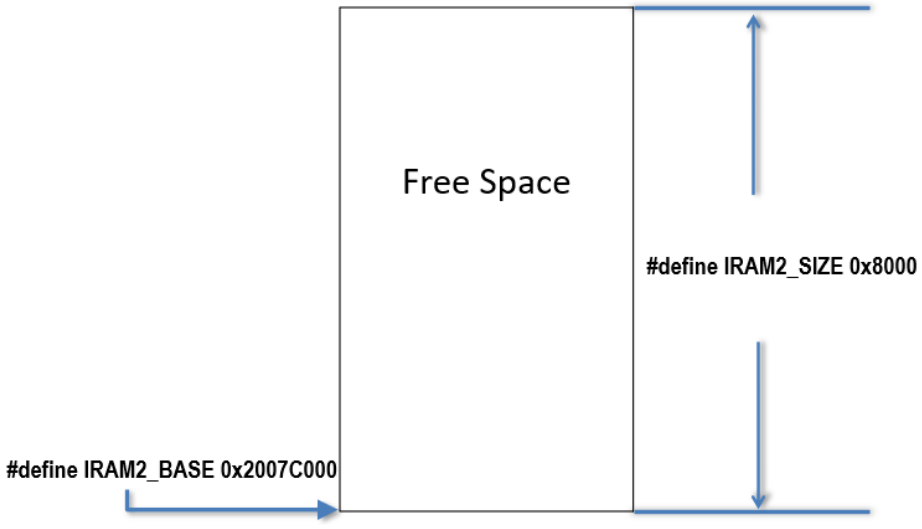


Figure 3.3: NXP LPC1768 IRAM2 Memory Execution View.

[1] We 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 programs [3]:

- `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.

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

[3] The 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 list [4]. Padding might be needed for reasons such as alignment requirement and allocator's strategy for coalescing et. al..

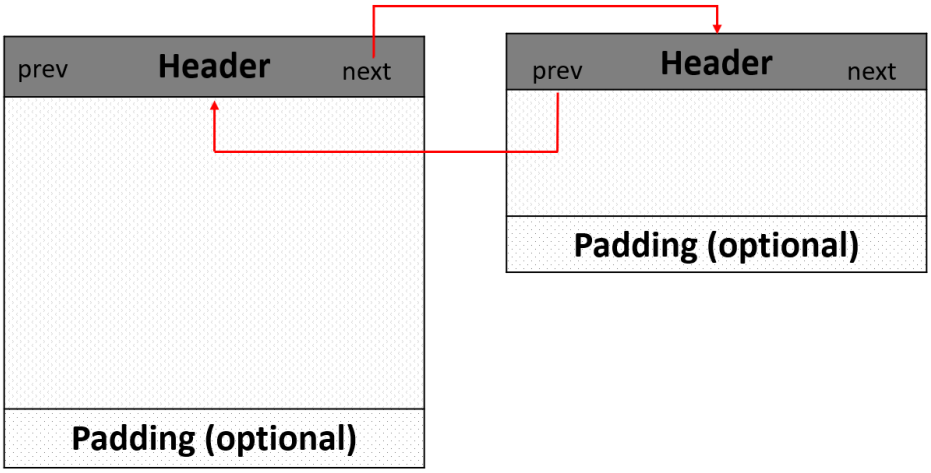


Figure 3.4: A Free Block Format

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.

[4] There 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 $i = 1, 2, ..., m - k$. 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 every 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 are 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 an 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

[5] BUDDY is the only supported algorithm in this lab.

serves the allocation and deallocation requests, the memory will be partitioned into allocated and free member 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 memory [6]. 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

[6] The `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(ptr)` 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

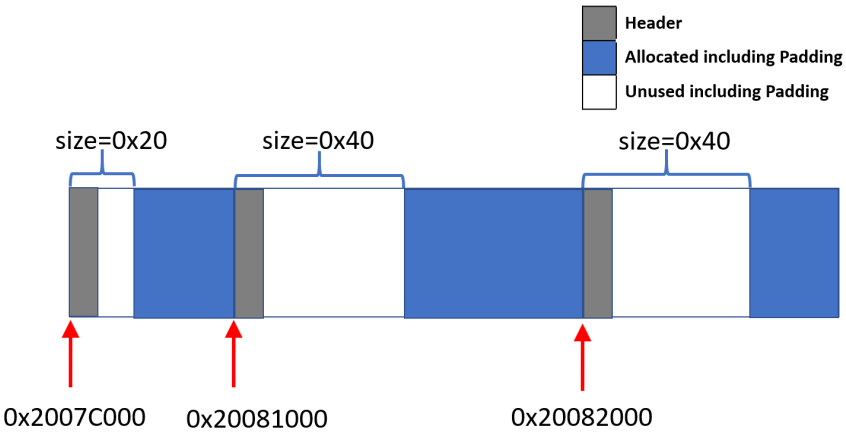


Figure 3.5: A memory map.Figure is not drawn to scale.

Assume there are only three free memory blocks at addresses 0x2007c000, 0x20081000 and 0x20088200 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 member 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` files [7].
- 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 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):

[7] For 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/N tests PASSED
Gid-TSN: y/N tests FAILED
Gid-TSN: END
```

In the above example output, the "id" is the Group ID. The "N" is the test suite ID. 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/N tests PASSED
G99-TS1: 1/N 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 `p1_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.

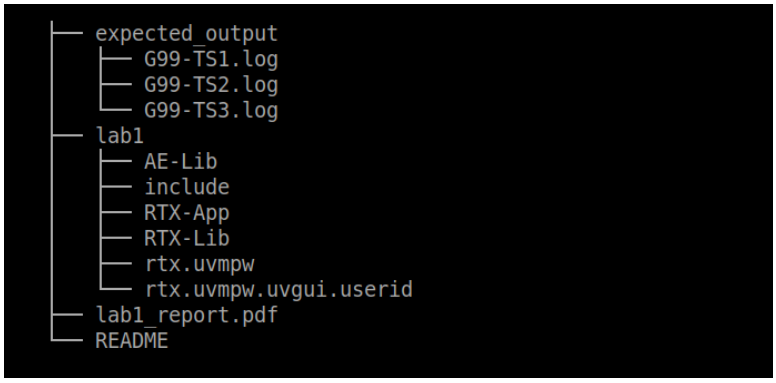


Figure 3.6: Lab1 Submission Directory Layout

## 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<br>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

To be released by May 28, 2021.

# Chapter 5

# Lab3

To be released by Jun 25, 2021.

# 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 Englab at https://englab.uwaterloo.ca/ is the main gateway. Choose **ECE** $\rightarrow$ **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).

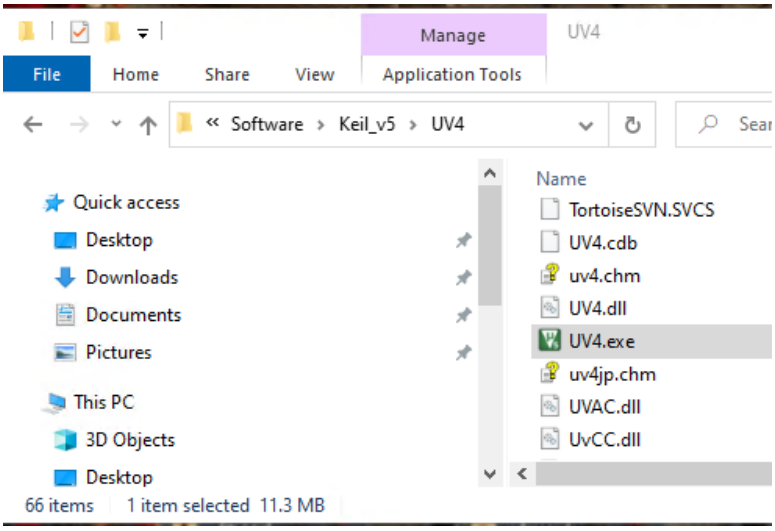


Figure 8.1: Keil IDE: Create a New Project

## 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).

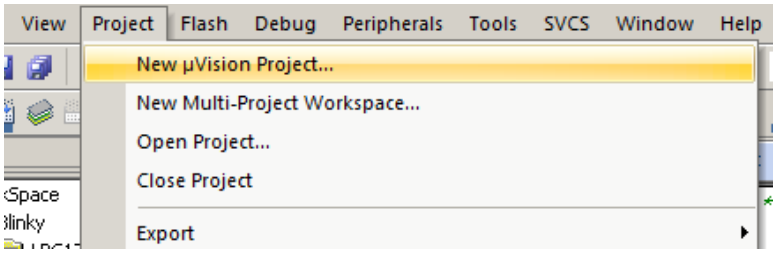


Figure 8.2: Keil IDE: Create a New Project

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

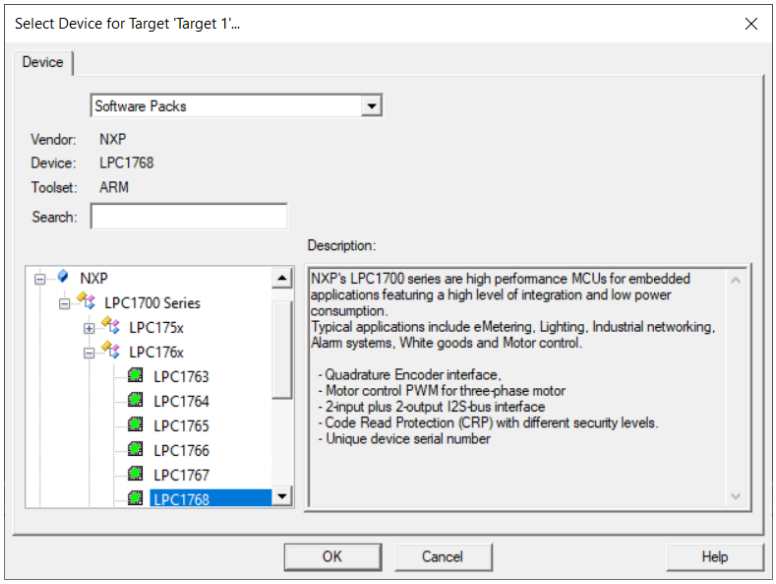


Figure 8.3: Keil IDE: Choose MCU

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

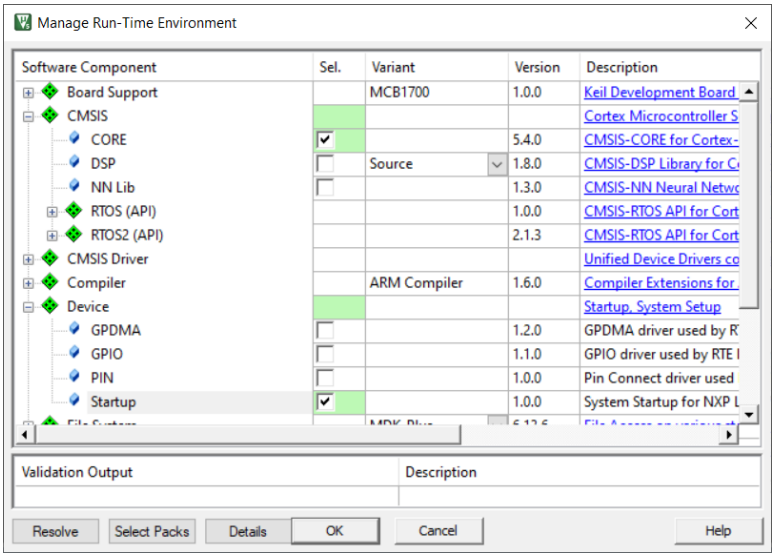


Figure 8.4: Keil IDE: Manage Run-time Environment

## 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.

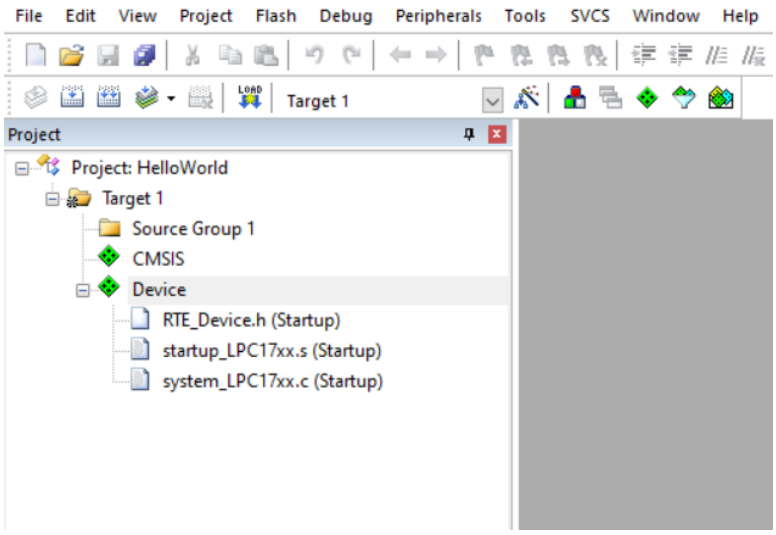


Figure 8.5: Keil IDE: A default new project

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).

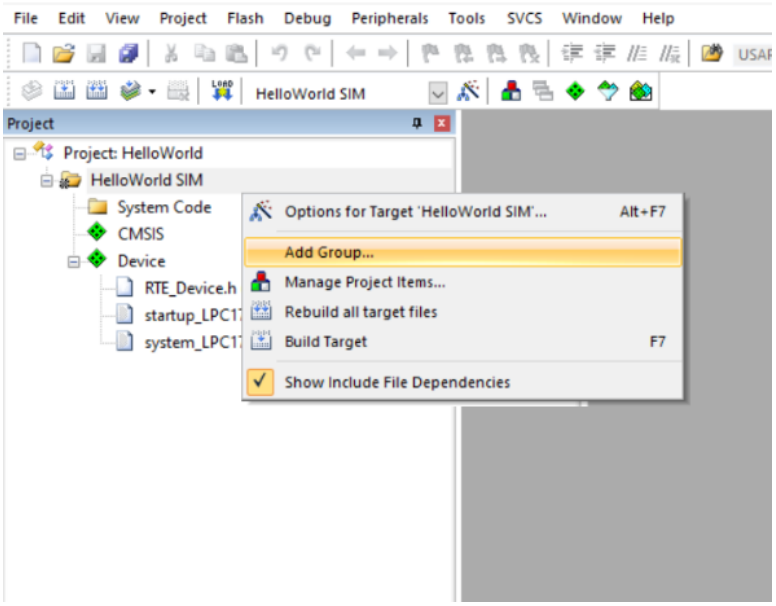


Figure 8.6: Keil IDE: Add Group

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.

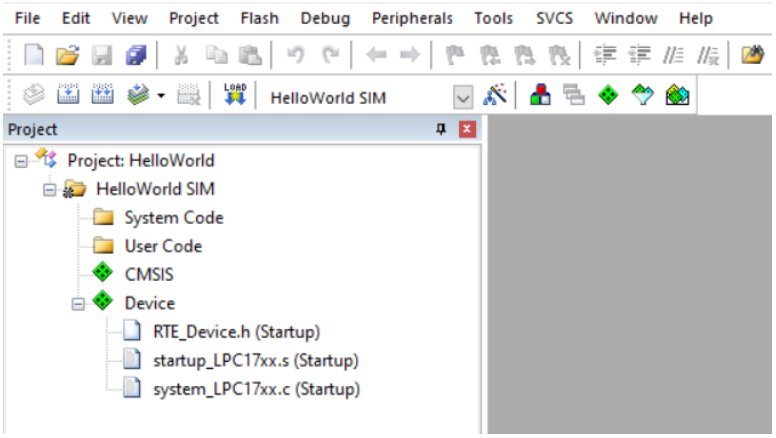


Figure 8.7: Keil IDE: Updated Project Profile

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

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).

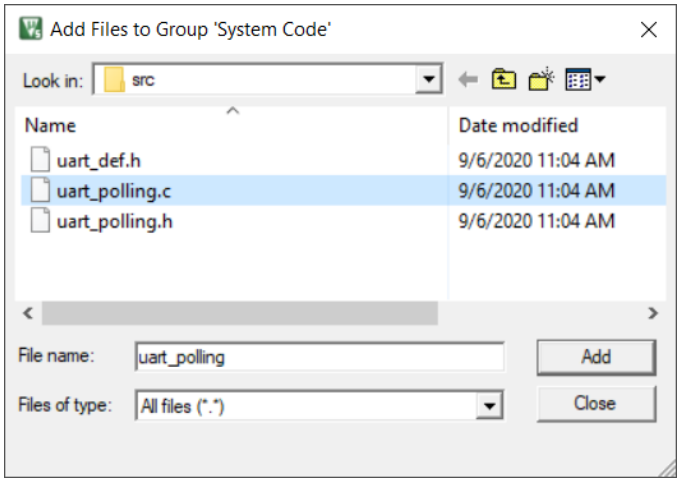


Figure 8.8: Keil IDE: Add Source File to Source Group

Your project will now look like Figure 8.9.

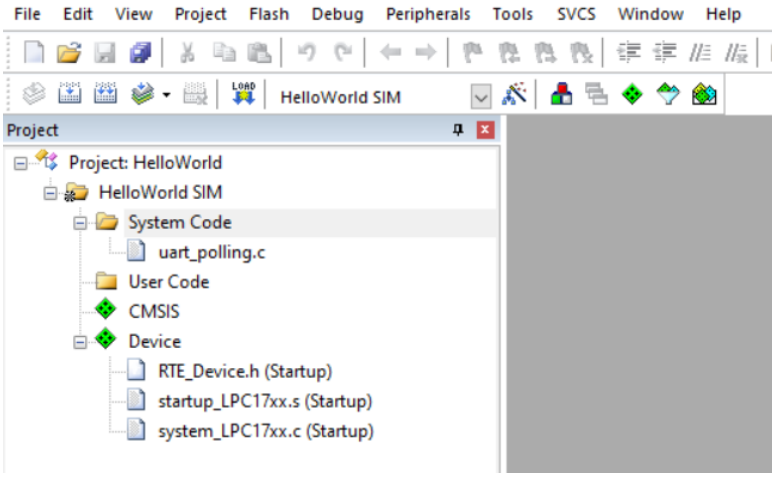


Figure 8.9: Keil IDE: Updated Project Profile

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).

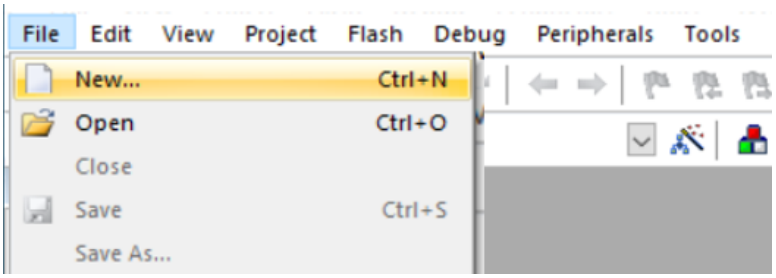


Figure 8.10: Keil IDE: Create New File

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.

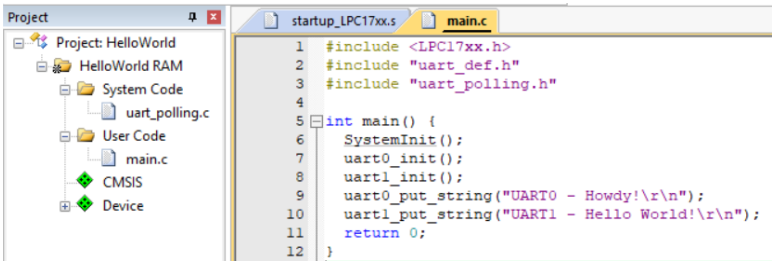


Figure 8.11: Keil IDE: Final Project Setting

## 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).

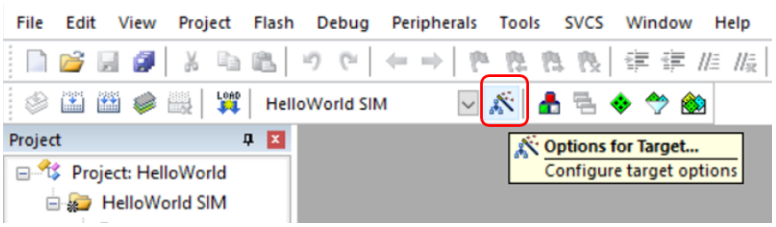


Figure 8.12: Keil IDE: Target Options Configuration

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.

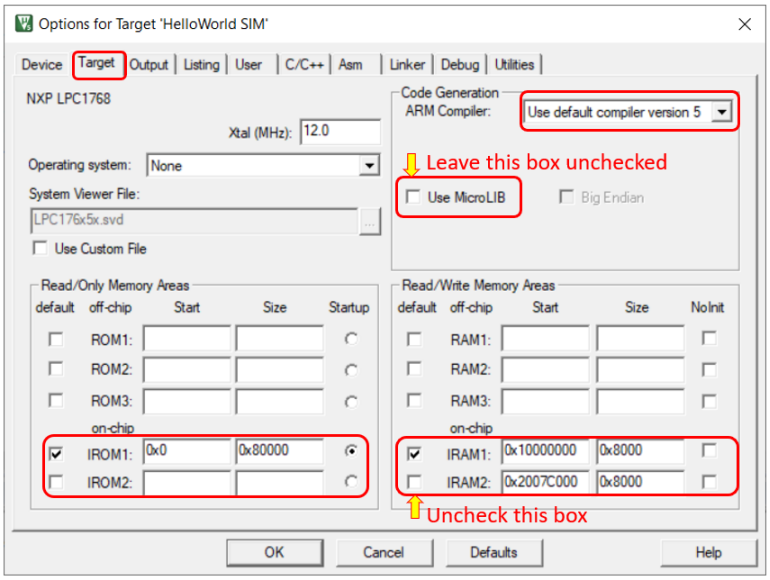


Figure 8.13: Keil IDE: Target Options Target Tab Configuration

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.

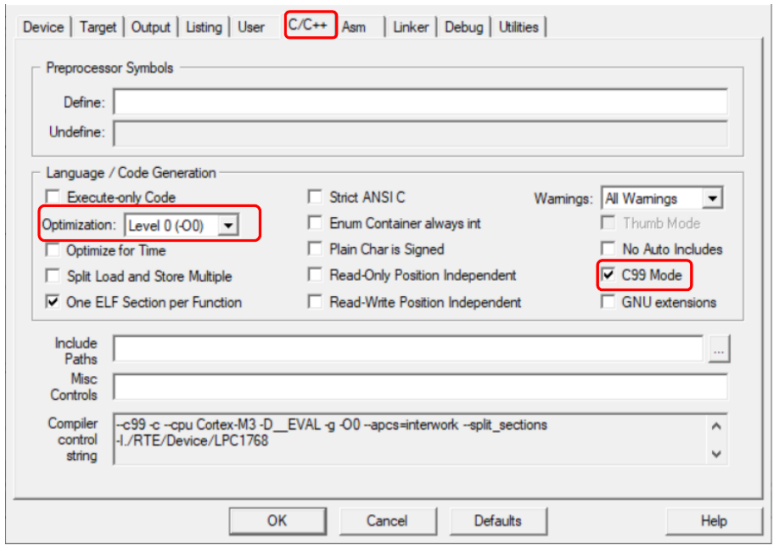


Figure 8.14: Keil IDE: Target Options C/C++ Tab Configuration

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.

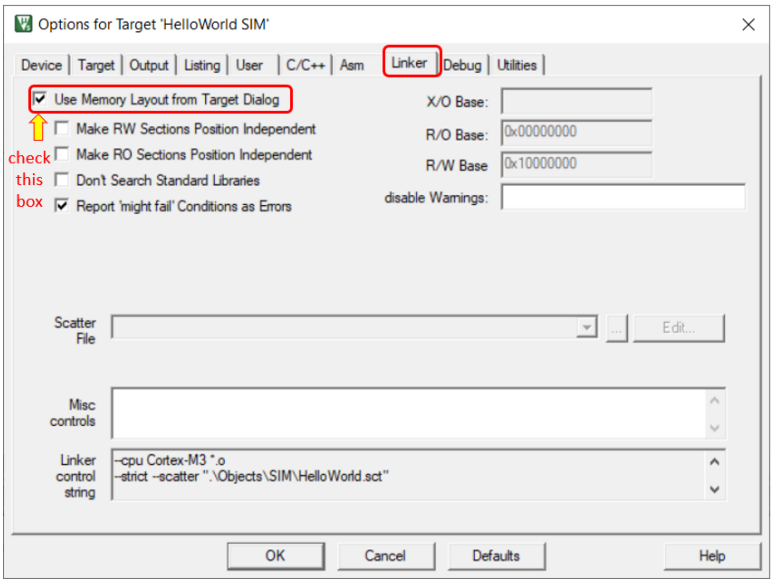


Figure 8.15: Keil IDE: Target Options Linker Tab Configuration

### 8.6.2 Build the Target

To build the target, click the "Build" button (see Figure 8.16).

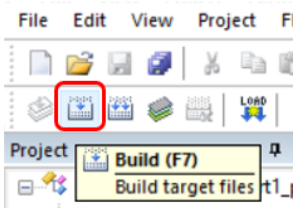


Figure 8.16: Keil IDE: Build Target

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 8.17.

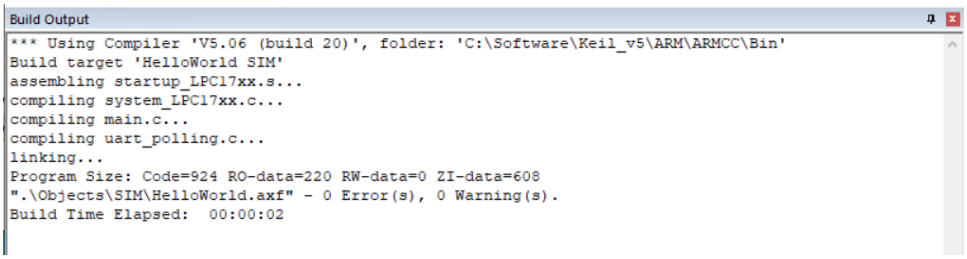


Figure 8.17: Keil IDE: Build Target

## 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 8.18. 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.

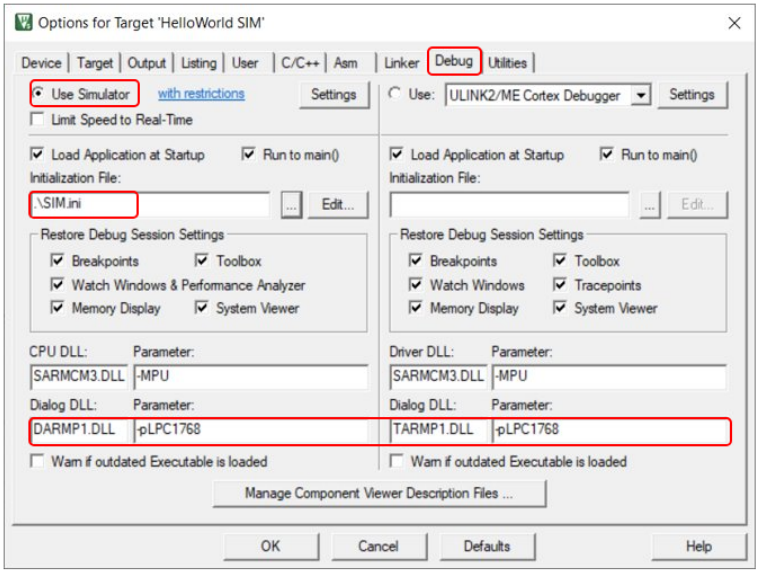


Figure 8.18: Keil IDE: Target Options Debug Tab Configuration

2. Press the "debug" button to bring up the debugger interface (See Figure 8.19).

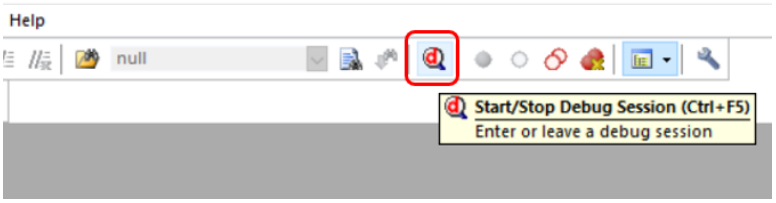


Figure 8.19: Keil IDE: Debug Button

3. Select UART1 and UART2 (see Figure 8.20) from the serial window drop down list so that they appear in simulator (see Figure 8.21). 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.

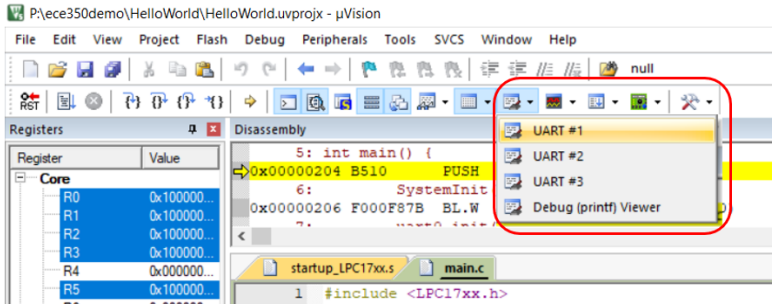


Figure 8.20: Keil IDE: Debugging. Enable Serial Window View.

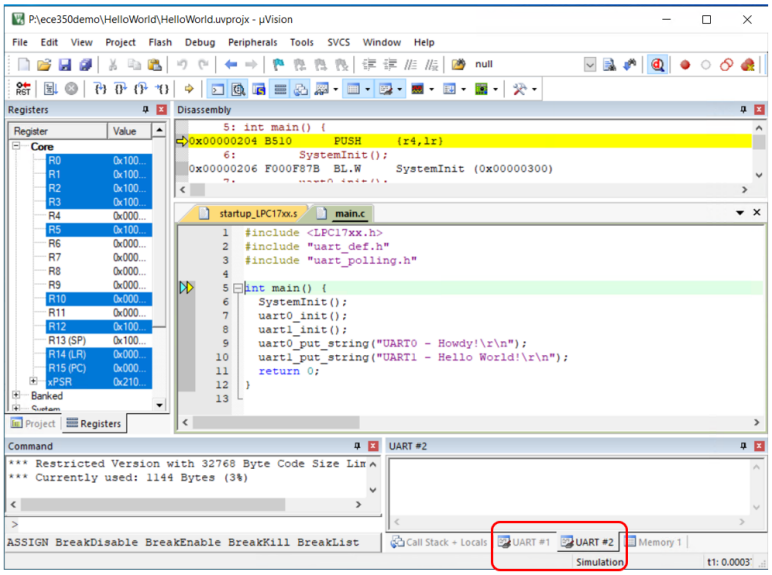


Figure 8.21: Keil IDE: Debugging. Both UART0 and UART1 views are enabled in simulator.

4. Press the "Run" button on the menu to let the program execute (see Figure 8.22). You will see the output of UART0 appearing in UART1 simulator window and the output of UART1 appearing in UART2 simulator window (see

Figure 8.23). Note that we moved the UART windows from their default positions in the simulator for better view.

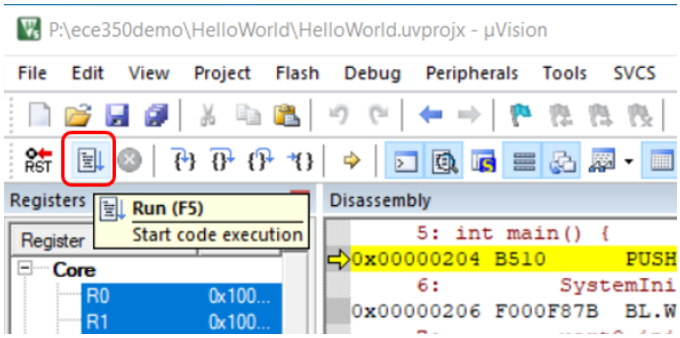


Figure 8.22: Keil IDE: Debugging. The Run Button.

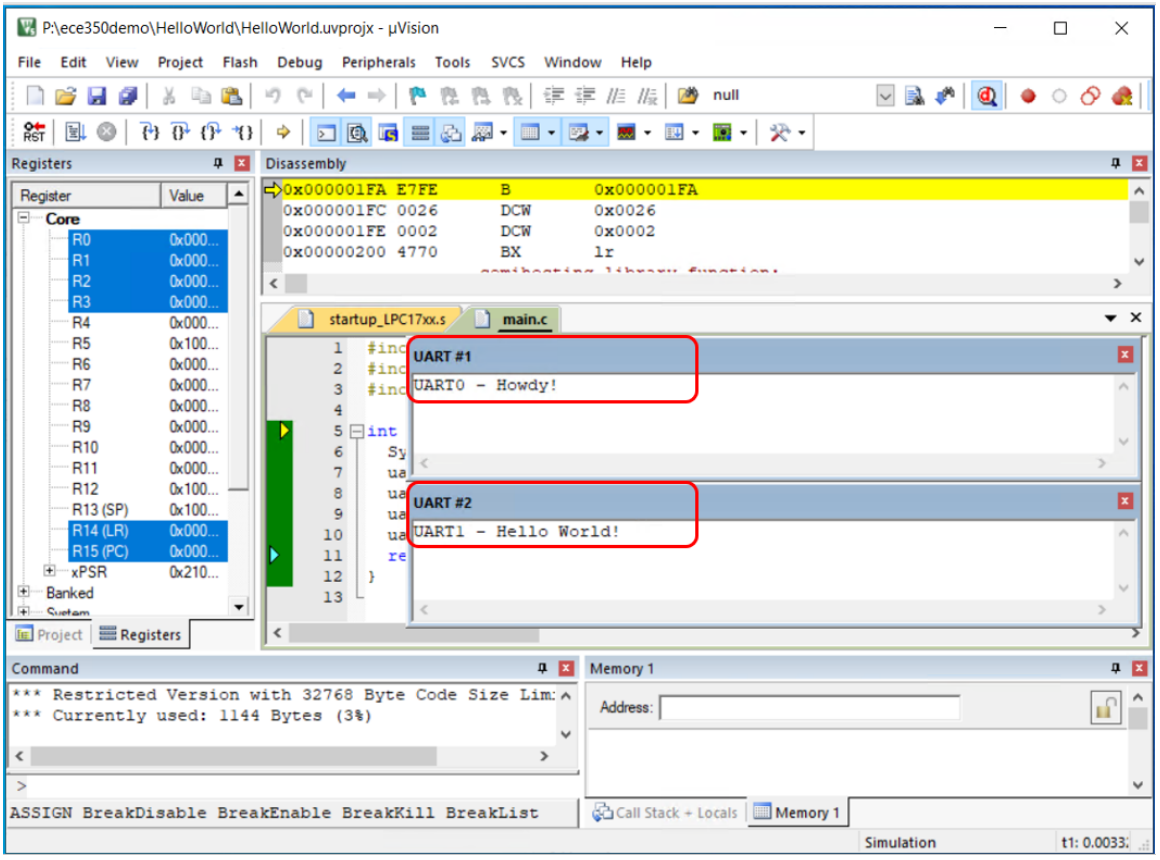


Figure 8.23: Keil IDE: Debugging Output.

5. To exit the debugging session, press the "debug" button again (see Figure 8.19).

### 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.

1. Press the Managing Project Item button (see Figure 8.24).

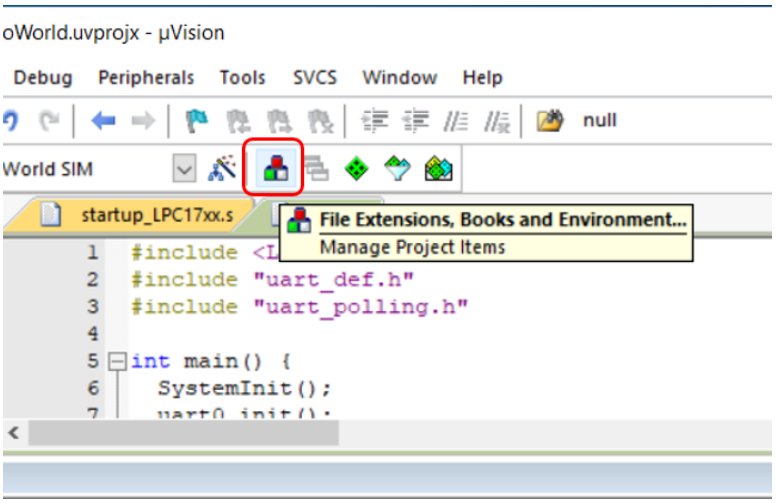


Figure 8.24: Keil IDE: Manage Project Items Button

2. Press the New icon to create a new target and name it "HelloWorld RAM"(see Figure 8.25). The new target duplicates the HelloWorld SIM target configuration.

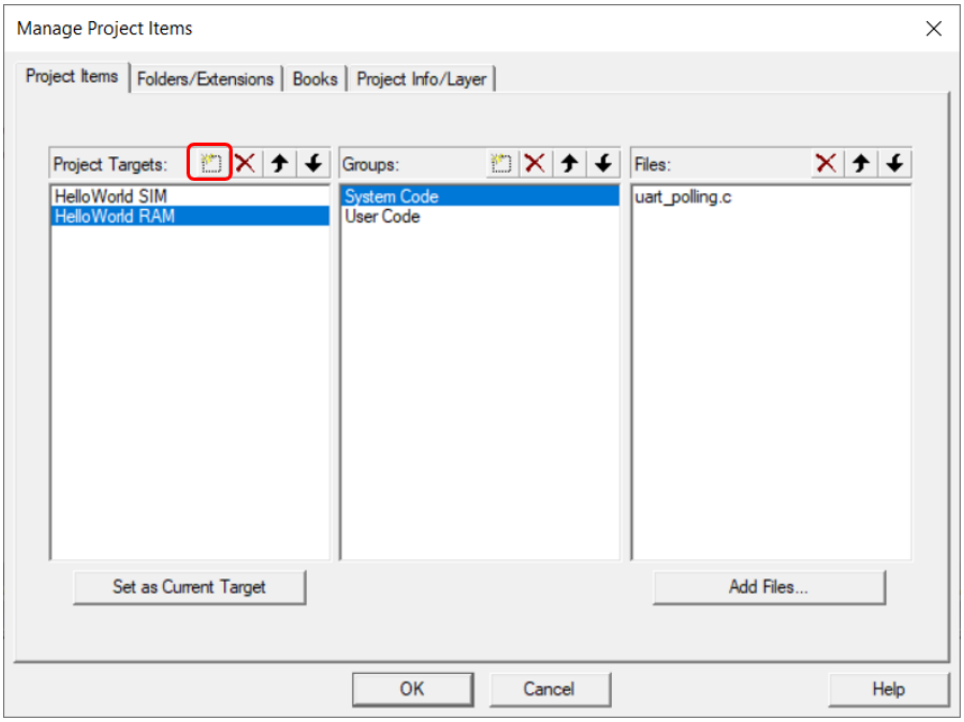


Figure 8.25: Keil IDE: Manage Project Items Window.

3. Switch your target to the newly created RAM target (See Figure 8.26).

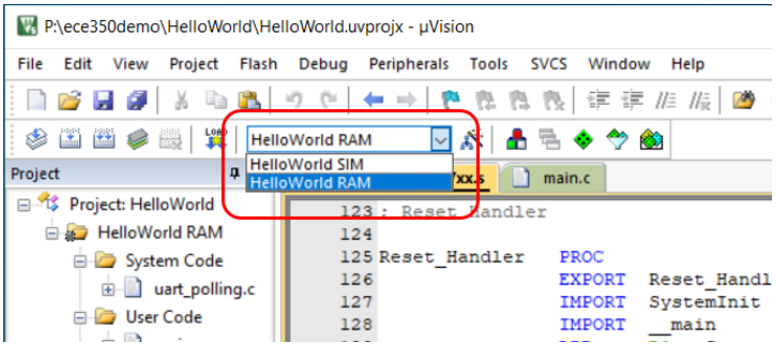


Figure 8.26: Keil IDE: Select HelloWorld RAM Target.

Configure in-memory code execution as shown in Figure 8.27.

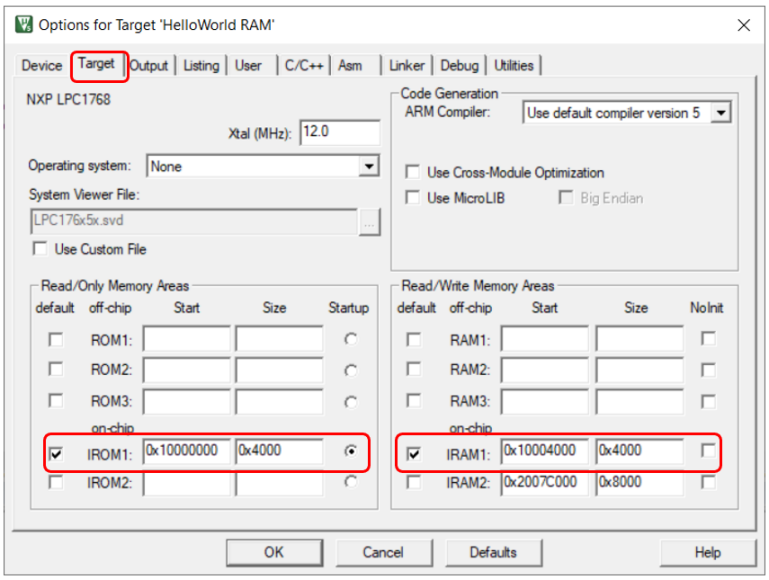


Figure 8.27: Keil IDE: Configure Target Options Target Tab for In-memory Execution.

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 8.27 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 8.28.

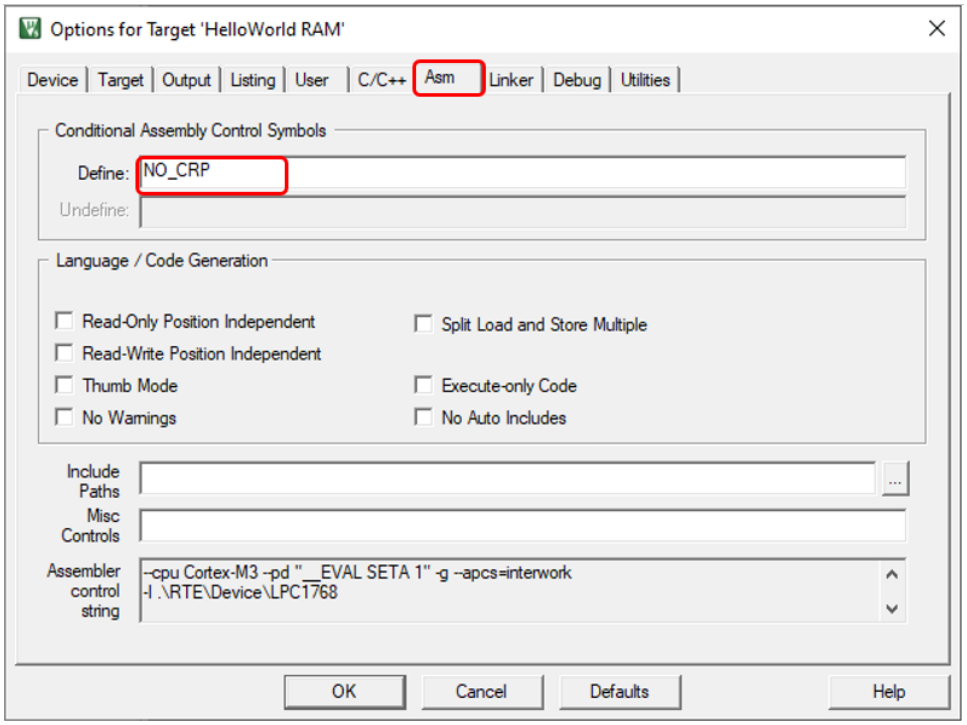


Figure 8.28: Keil IDE: RAM Target Asm Configuration.

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 8.29) 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.

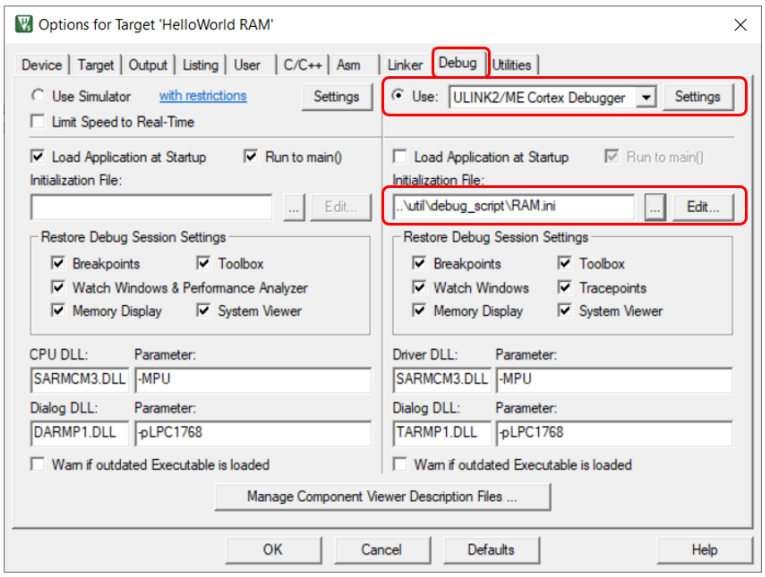


Figure 8.29: Keil IDE: Configure ULINK-ME Hardware Debugger.

6. Press the settings button beside the ULINK2/ME Cortex Debugger (see Figure 8.29) and select the Flash Download tab (see Figure 8.30). Remove the LPC17xx

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

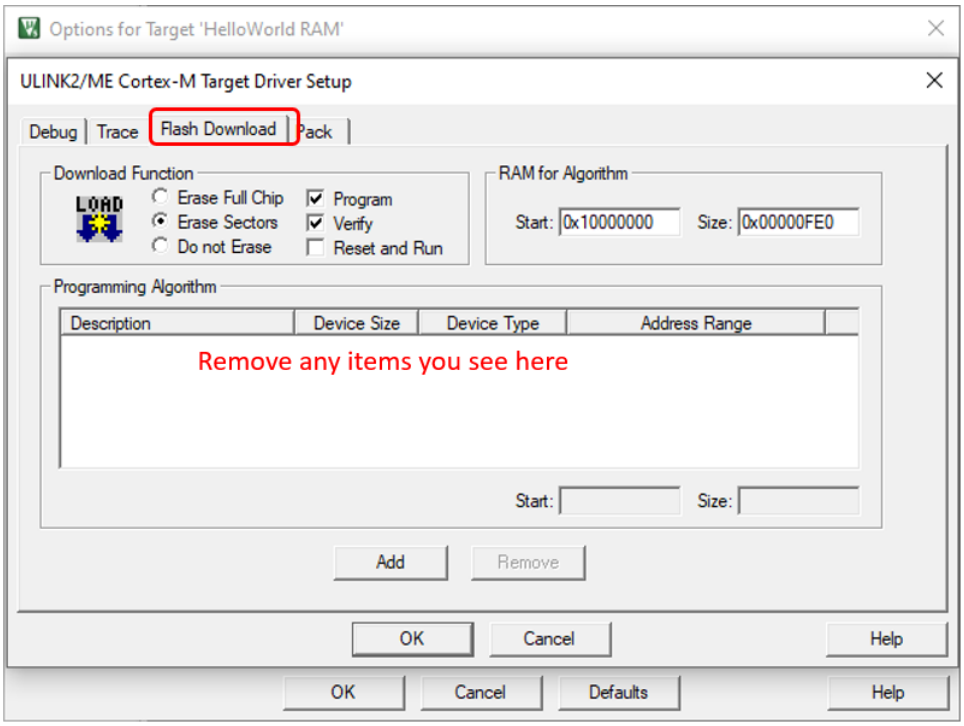


Figure 8.30: Keil IDE: Flash Download Programming Algorithm Configuration.

7. Select the Utilities tab and select the radio button beside "Use External Tool for Flash Programming" (see Figure 8.31).

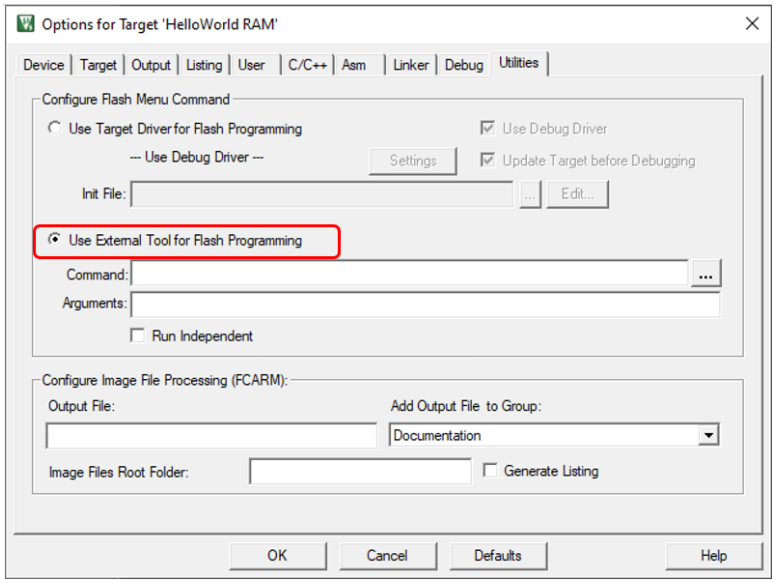


Figure 8.31: Keil IDE: Target Option Utilities Configuration for RAM Target.

8. 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 8.32).

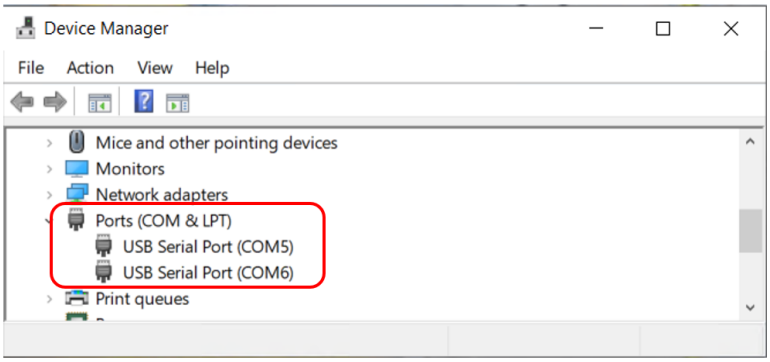


Figure 8.32: Device Manger COM Ports

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 8.33 and 8.34.

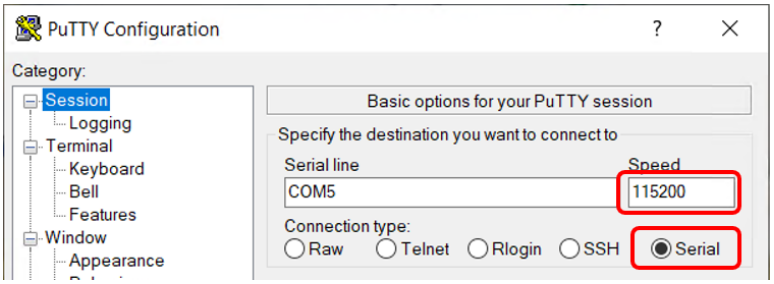


Figure 8.33: PuTTY Session for Serial Port Communication

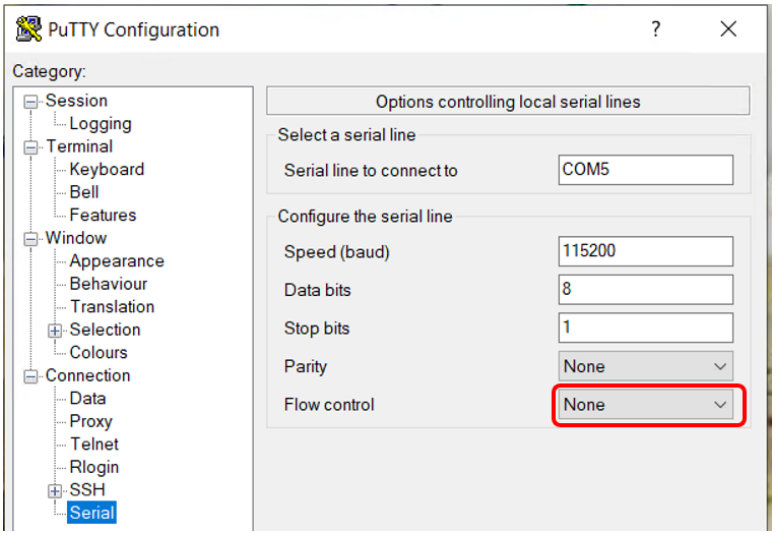


Figure 8.34: PuTTY Serial Port Configuration

9. 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.

10. 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 8.35).


Figure 8.35: 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 8.37). 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 8.36). 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.

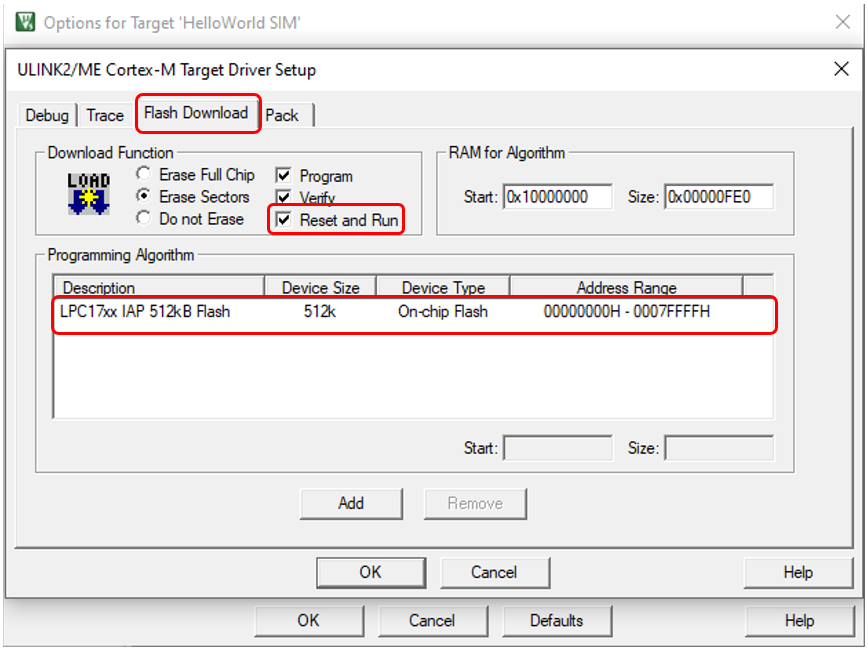


Figure 8.36: Flash Download Reset and Run Setting

To download the code to the on-chip ROM, click the "LOAD" button (see Figure 8.37). The download is through the ULINK-ME. The code automatically runs. You should see the output from PuTTY terminals.

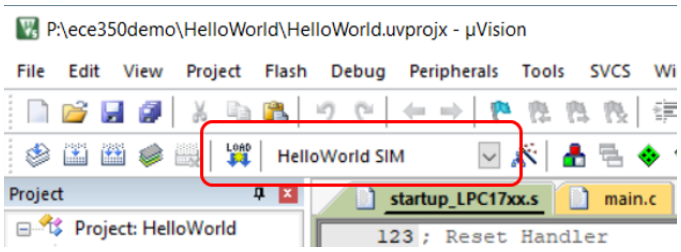


Figure 8.37: Keil IDE: Download Target to Flash

## 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 `Helloword-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 8.38).
- 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 8.38. Note we omitted the directory layout of the `HelloWorld` folder.

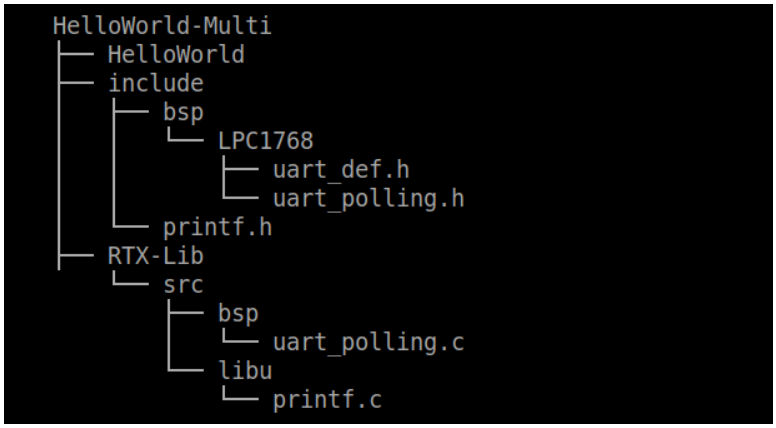


Figure 8.38: Directory Structure of a Multi-Project Workspace. The HelloWorld directory layout is omitted.

### 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 Firgure 8.39). 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 8.39).

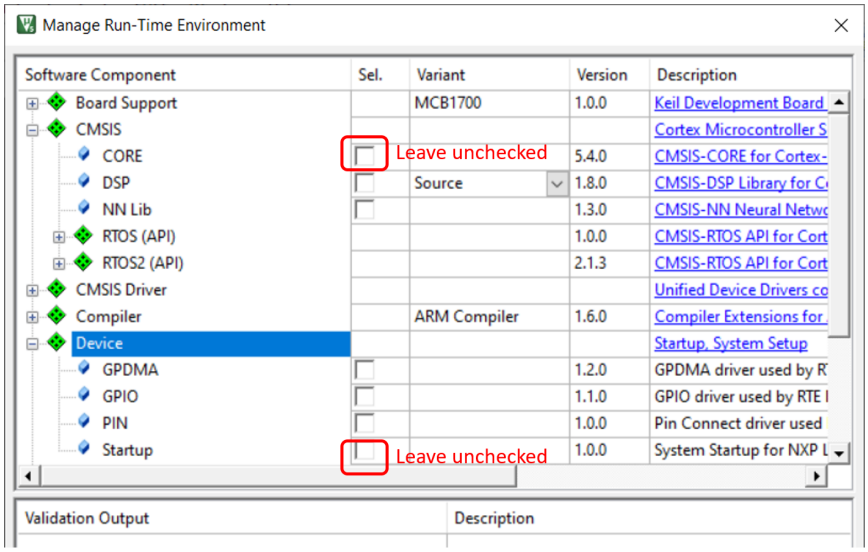


Figure 8.39: Directory Structure of a Multi-Project Workspace. The HelloWorld directory layout is omitted.

- 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 8.40.

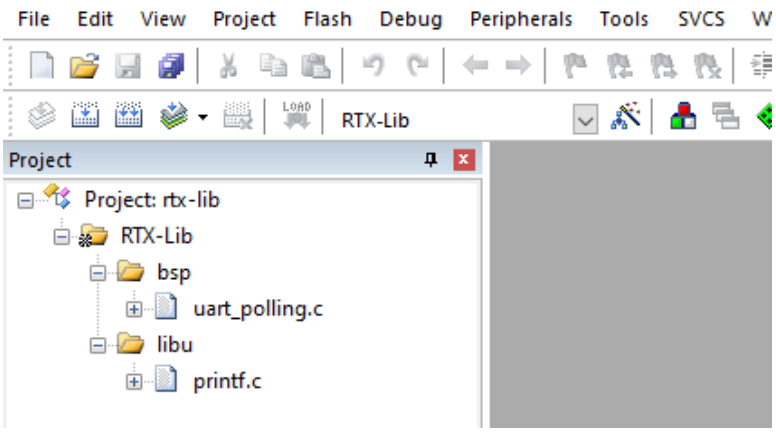


Figure 8.40: Keil IDE: A Library Project Profile

### 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 8.41

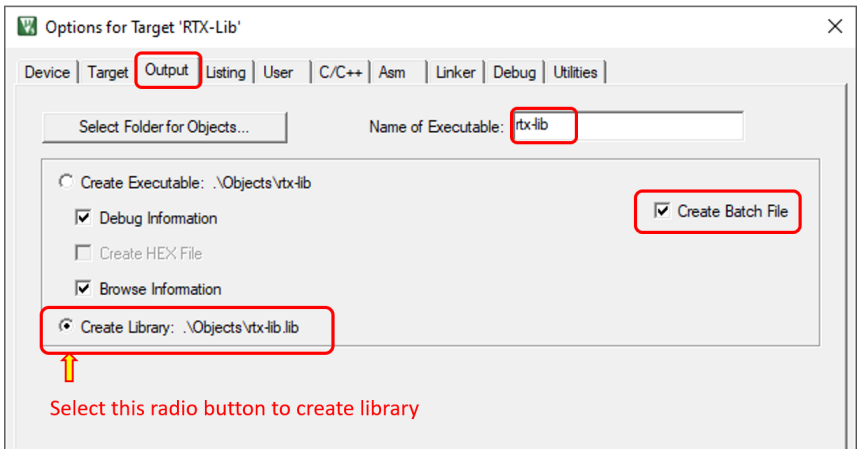


Figure 8.41: Keil IDE: Target Options Output Tab Library Creation Configuration

4. Configure the C/C++ tab as shown in Figure 8.42. 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.

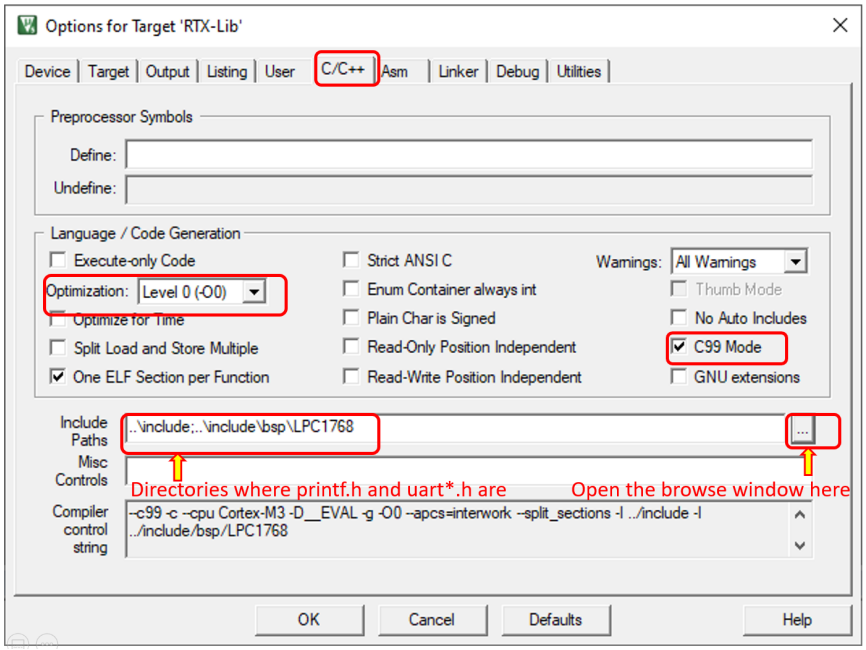


Figure 8.42: Keil IDE: Target Options C/C++ Tab Configuration

5. Configure the Linker tab the same way as you we 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 8.16). 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 8.43. 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.

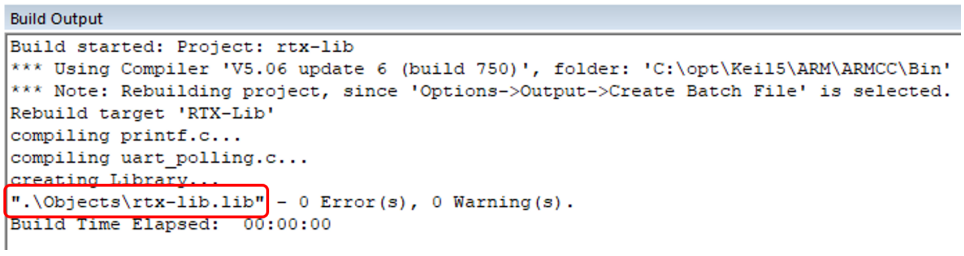


Figure 8.43: Keil IDE: Build Library Target

## 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 8.44).

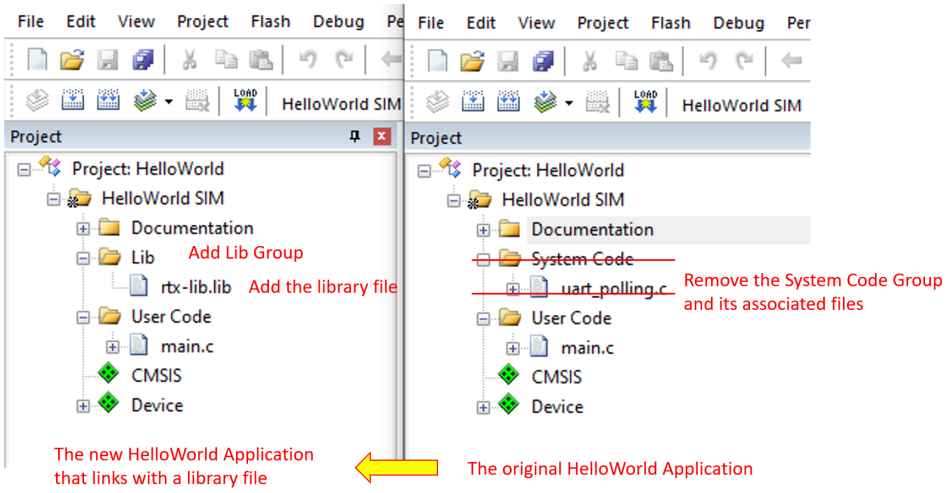


Figure 8.44: Keil IDE: HelloWorld Application that uses a Library

We then navigate to the `HelloWorld-Multi\HelloWorld\src` folder to remove `printf.[ch]` and `uart*.[ch]` files as shown in Figure 8.45.

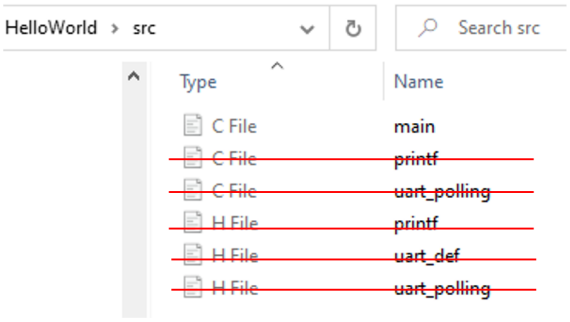


Figure 8.45: Keil IDE: Removing source code files from HelloWorld inside the Helloworld-Multi folder

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 8.42). 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 8.16). You will see the application is built and a `.axf` file is generated (see Figure 8.46). You may either use the debugger to run it inside a simulator or on the board as usual.

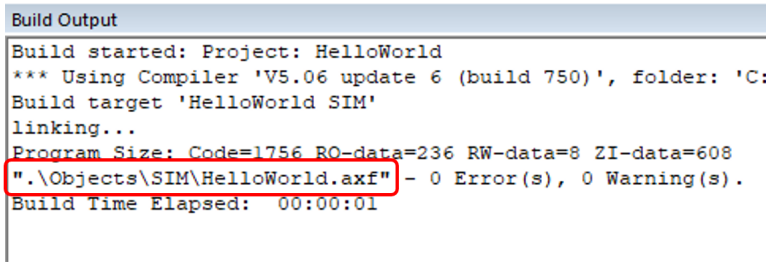


Figure 8.46: Keil IDE: Build Output of HelloWorld Application Linked with a Library

## 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 8.47).

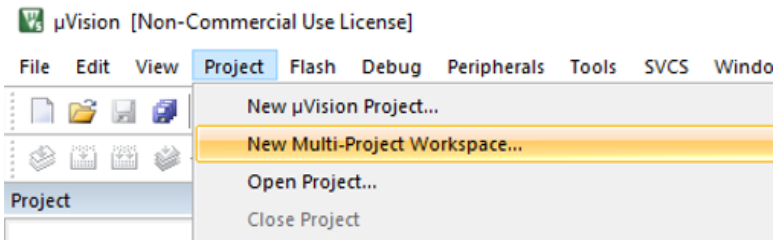


Figure 8.47: Keil IDE: Create a New Multi-Project Workspace Menu Item

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 8.48).

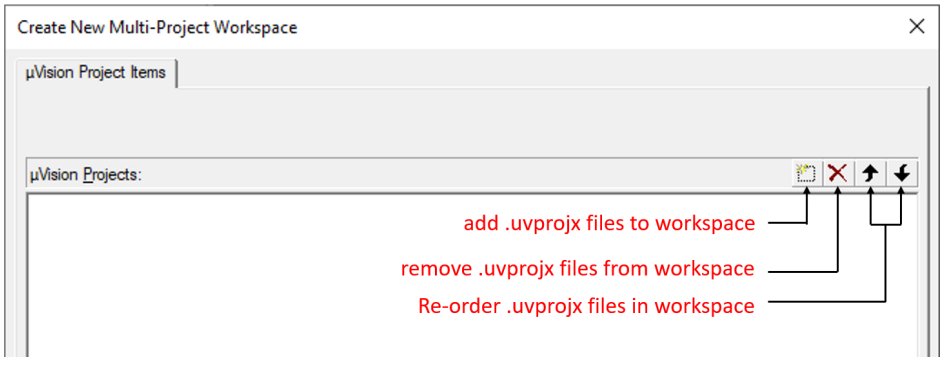


Figure 8.48: Keil IDE: Create a New Multi-Project Workspace Window

Let's first select the library project and then the application project (see Figure 8.49).

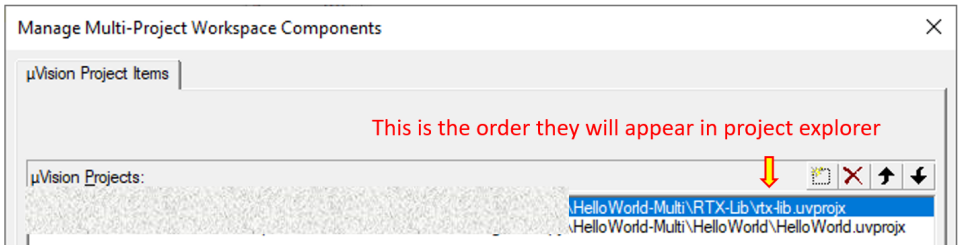


Figure 8.49: Keil IDE: Final New Multi-Project Workspace Window

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 8.50.

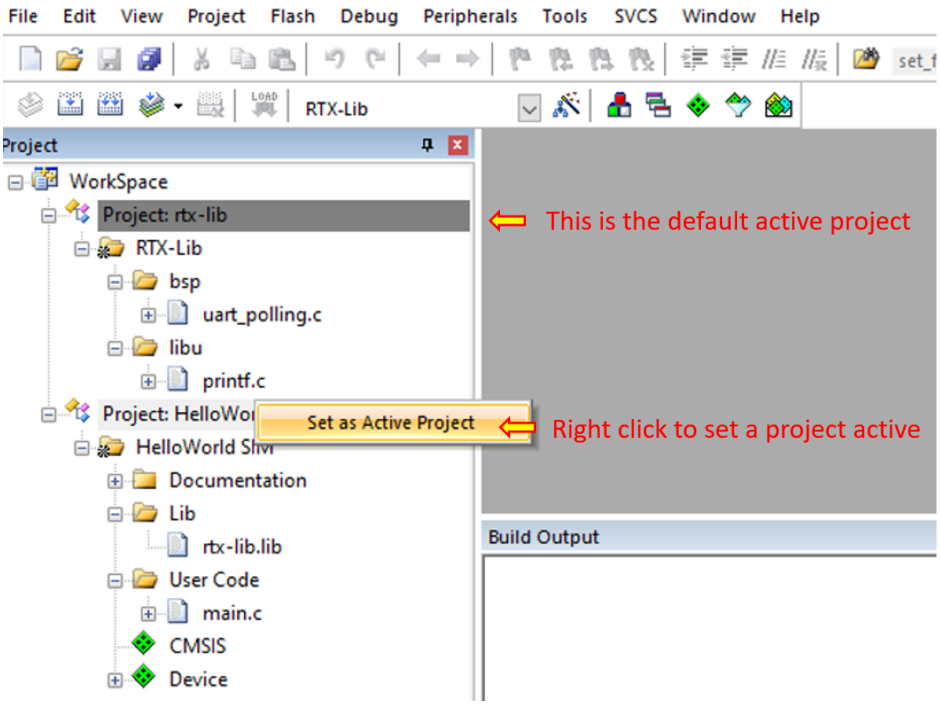


Figure 8.50: Keil IDE: Multi-Project Workspace Explorer

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 8.50).

## 8.12 Batch Build

In the multi-project workspace, pressing the build button (see Figure 8.16 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 8.51).

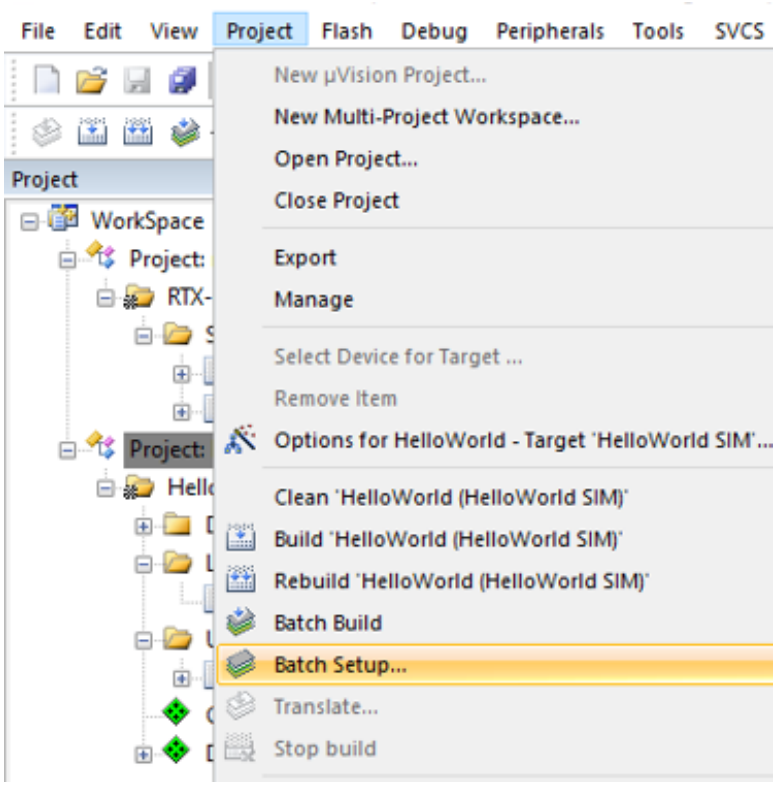


Figure 8.51: Keil IDE: Batch Setup Menu Item

We want to build all targets in the workspace (see Figure 8.52). 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 secrete 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.

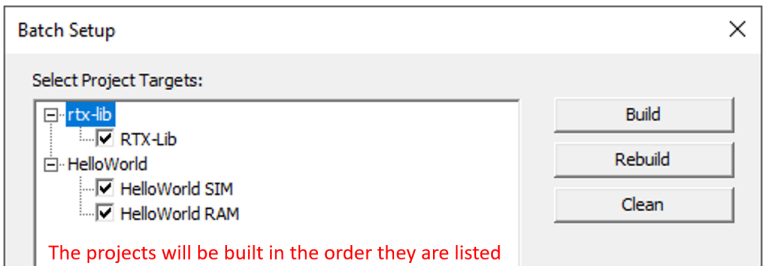


Figure 8.52: Keil IDE: Batch Setup Window

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 8.53).

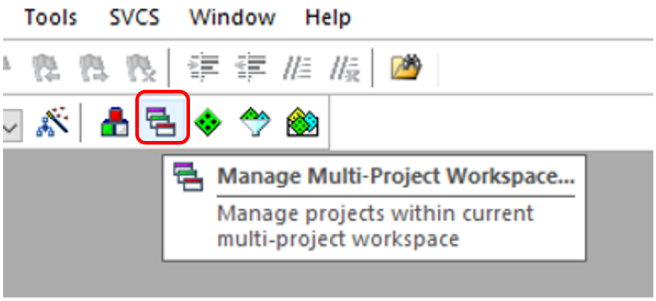


Figure 8.53: Keil IDE: Manage Multi-Project Workspace Button

To batch build the workspace, press the batch build button (see Figure 8.54). You can also select Project → Batch Build from the menu to achieve the same purpose.

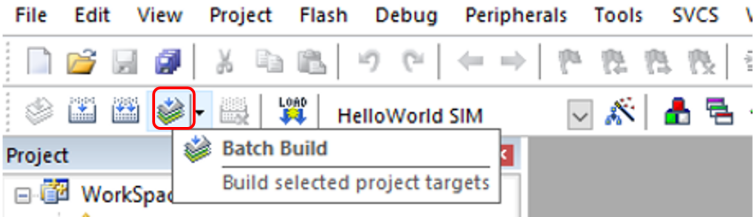


Figure 8.54: Keil IDE: Batch Build Button

You should watch carefully the build output message in the console window. Pay special attention to the last line of build summary (see Figure 8.55). 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.

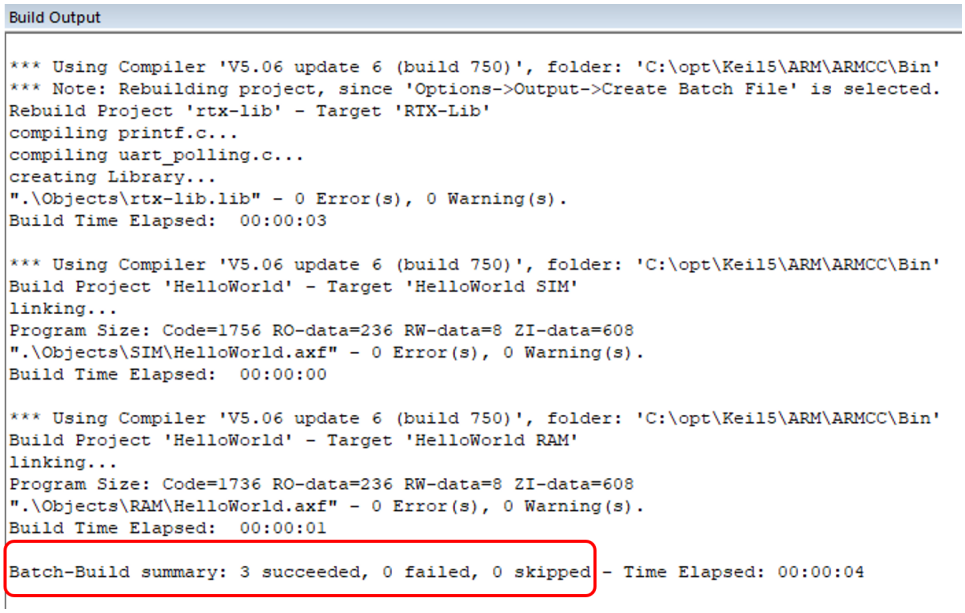


Figure 8.55: Keil IDE: Batch Build Output

## 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 8.56) 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.

main.c

```
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%x, ptr = 0x%x\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  }
```

No \r

\r presents

Figure 8.56: The main.c code that uses printf

The output of the program by using the simulator is shown in Figure 8.57.

UART #2

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

Figure 8.57: Keil IDE: demonstration of printf using simulator

The output of the program by using the simulator is shown in Figure 8.58.

COM9 - PuTTY

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

Figure 8.58: 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 78 - 79, item 2 and item 3 orders are swapped.

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

Table 9.1: Assembler instruction examples

| Register | Synonym | Special | Role in the procedure call standard |
|---|---|---|---|
| r15 | | PC | The Program Counter. |
| r14 | | LR | The Link Register. |
| r13 | | SP | The Stack Pointer (full descending stack). |
| r12 | | IP | The Intra-Procedure-call scratch register. |
| r11 | v8 | | Variable-register 8. |
| r10 | v7 | | Variable-register 7. |
| r9 | | v6<br>SB<br>TR | Platform register.<br>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 [2].

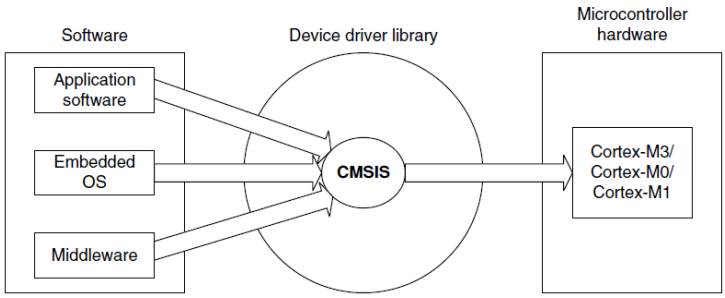


Figure 9.1: Role of CMSIS[6]

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).

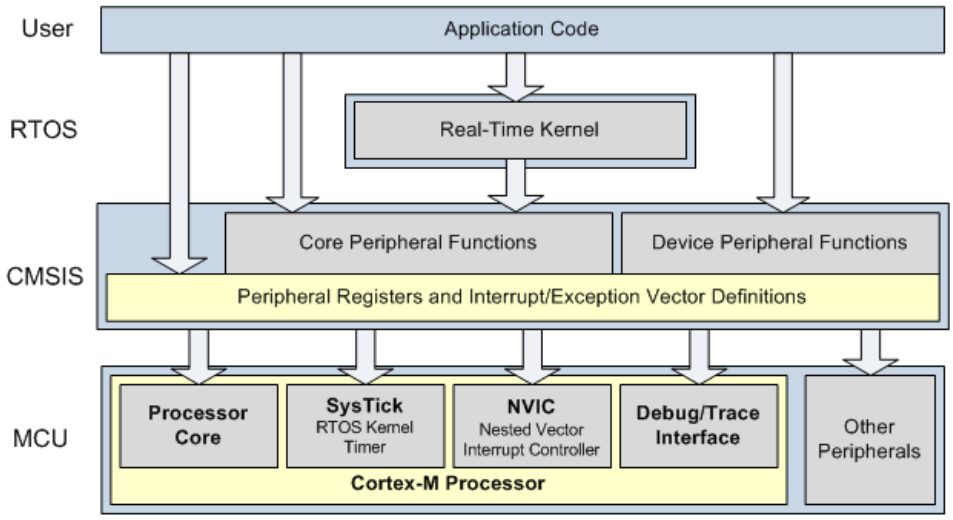


Figure 9.2: CMSIS Organization[2]

By including the `<device>.h` (e.g., `LPC17xx.h`) file into your code file. The first step to initialize the system can be done by calling the CMSIS function as shown in Listing 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`).

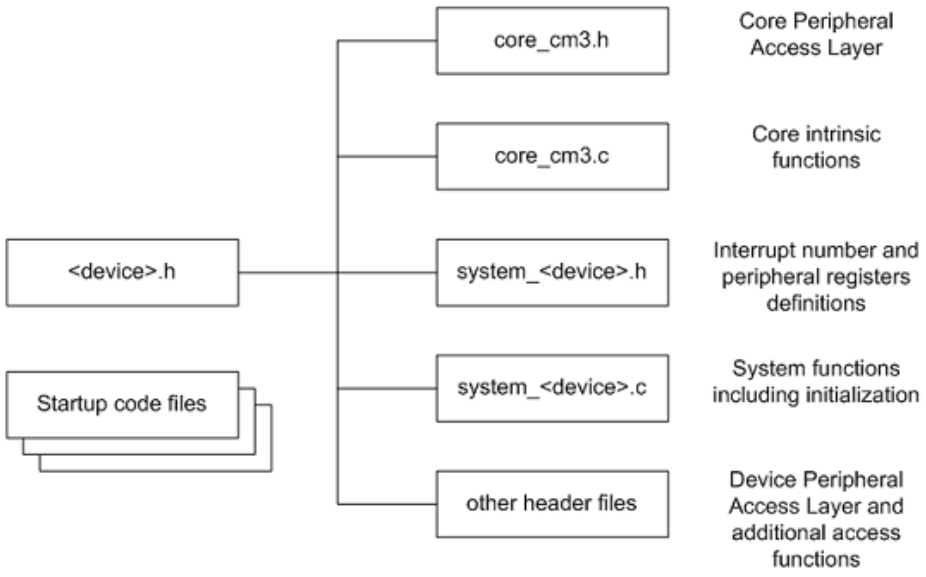


Figure 9.3: CMSIS Organization[2]

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

Figure 9.4: CMSIS NVIC Functions[2]

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

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

Another way is to use the embedded assembly code:

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

Table 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

Only embedded assembly is support in Cortex-M3. To write an embedded assembly function, you need to use the __asm keyword. For example the the function "embedded_asm_function" in Listing 9.3 is an embedded assembly function. You can only put assembly instructions inside this function. Note that inline assembly is not supported in Cortex-M3.

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

- A global variable defined in C. In Listing 9.2, we have two C global variables g_pcb and g_var. We can use the __cpp to access them as shown in Listing 9.3. 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 CMSIS 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.

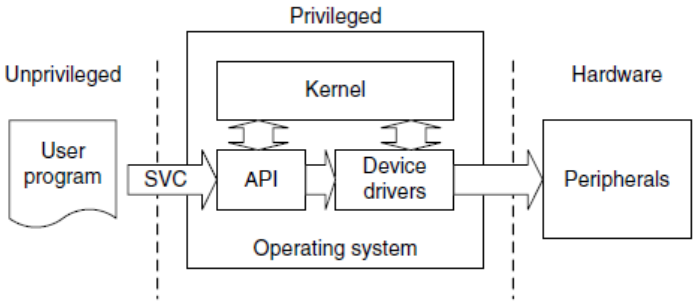


Figure 9.5: SVC as a Gateway for OS Functions [6]

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

The direct method is to program at assembly instruction level. We can use the embedded assembly mechanism and write `SVC` assembly instruction inside the embedded assembly function. One implementation of `void *mem_alloc(size_t size)` is shown in Listing 9.7.

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

Listing 9.7: Code Snippet of mem_alloc

The corresponding kernel function is the C function `k_mem_alloc`. This function entry point is loaded to register `r12`. Then `SVC 0` causes an SVC exception with immediate number `0`. In the SVC exception handler, we can then branch with link and exchange to the address stored in `r12`. Listing 9.8 is an excerpt of the HAL.c from the starter code.

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

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

   ; code to save cpu registers omitted

   BLX R12 ; R12 contains the kernel function entry point

   ;Code to restore registers omitted

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

Listing 9.8: Code Snippet of SVC_Handler

The indirect method is to ask the compiler to generate the SVC instruction from C code. The ARM compiler provides an intrinsic keyword named __svc_indirect which passes an operation code to the SVC handler in r12[3]. This keyword is a function qualifier. The two inputs we need to provide to the compiler are

- svc_num, the immediate value used in the SVC instruction and
- op_num, the value passed in r12 to the handler to determine the function to perform. The following is the syntax of an indirect SVC.

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

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

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

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

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

The compiler generates two assembly instructions

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

The SVC_handler in Listing 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 9.9, 9.10 and 9.11 give one sample implementation of programming UART0 interrupts.

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

#ifndef UART_DEF_H_
#define UART_DEF_H_

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

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

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

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


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

/*
  8 bits, no Parity, 1 Stop bit

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

#ifndef NULL
#define NULL 0
```

```
#endif


#endif /* !UART_DEF_H_ */
```

Listing 9.9: UART0 IRQ Sample Code uart_def.h

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

#ifndef UART_IRQ_H_
#define UART_IRQ_H_

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

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

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

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

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


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

/*
  8 bits, no Parity, 1 Stop bit

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

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


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

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

#endif /* ! UART_IRQ_H_ */
```

Listing 9.10: UART0 IRQ Sample Code uart.h

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

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


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

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

  LPC_UART_TypeDef *pUart;

  if ( n_uart ==0 ) {
```

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

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

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

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

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

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

  pUart = (LPC_UART_TypeDef *) LPC_UART0;

} else if ( n_uart == 1) {

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

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

  pUart = (LPC_UART_TypeDef *) LPC_UART1;

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

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

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

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

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



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

pUart->FCR = 0x07;

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

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

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

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

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


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

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

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

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

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

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

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

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

Listing 9.11: UART0 IRQ Sample Code uart_irq.c

Listings 9.12 and 9.13 give one sample implementation of programming UART0 by polling.

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

#ifndef UART_POLLING_H_
#define UART_POLLING_H_

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

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

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

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

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

#endif /* ! UART_POLLING_H_ */
```

Listing 9.12: UART0 IRQ Sample Code uart_polling.h

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

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


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

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

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

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

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

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

   pUart = (LPC_UART_TypeDef *) LPC_UART0;

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

   pUart = (LPC_UART_TypeDef *) LPC_UART1;

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

  /* Step 3: Transmission Configuration */

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

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

  return 0;
}


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

int uart_get_char(int n_uart)
{
  LPC_UART_TypeDef *pUart;

  if (n_uart == 0) {
```

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

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

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

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

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

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

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

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

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

Listing 9.13: 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 9.14 and 9.15 give one sample implementation of programming TIMER0 interrupts. The timer interrupt fires every one millisecond.

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

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

#endif /* ! _TIMER_H_ */
```

Listing 9.14: Timer0 IRQ Sample Code timer.h

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  g_timer_count = 0;

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

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

  return 0;
}

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

  g_timer_count++ ;
}
```

Listing 9.15: 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

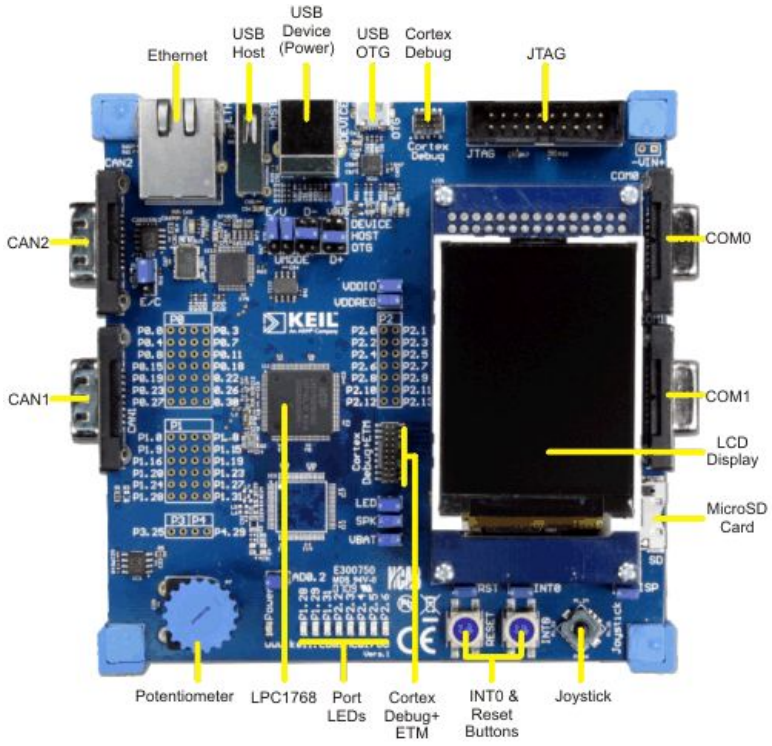


Figure 10.1: MCB1700 Board Components [1]

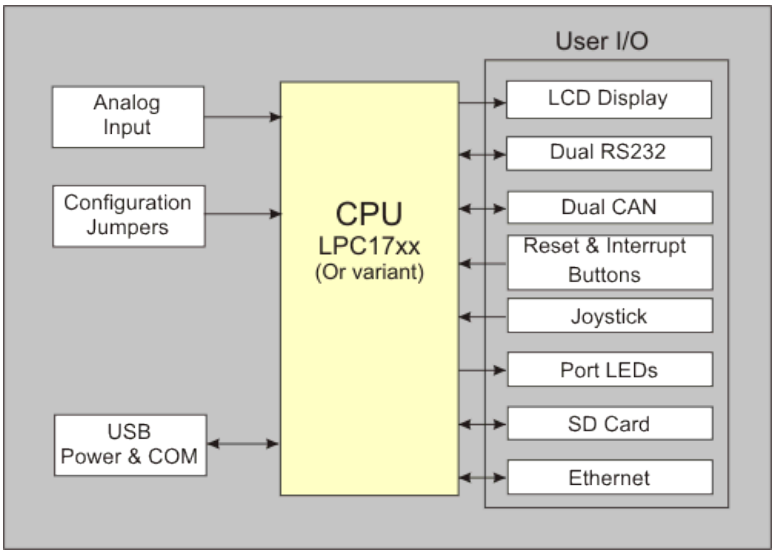


Figure 10.2: MCB1700 Board Block Diagram [1]

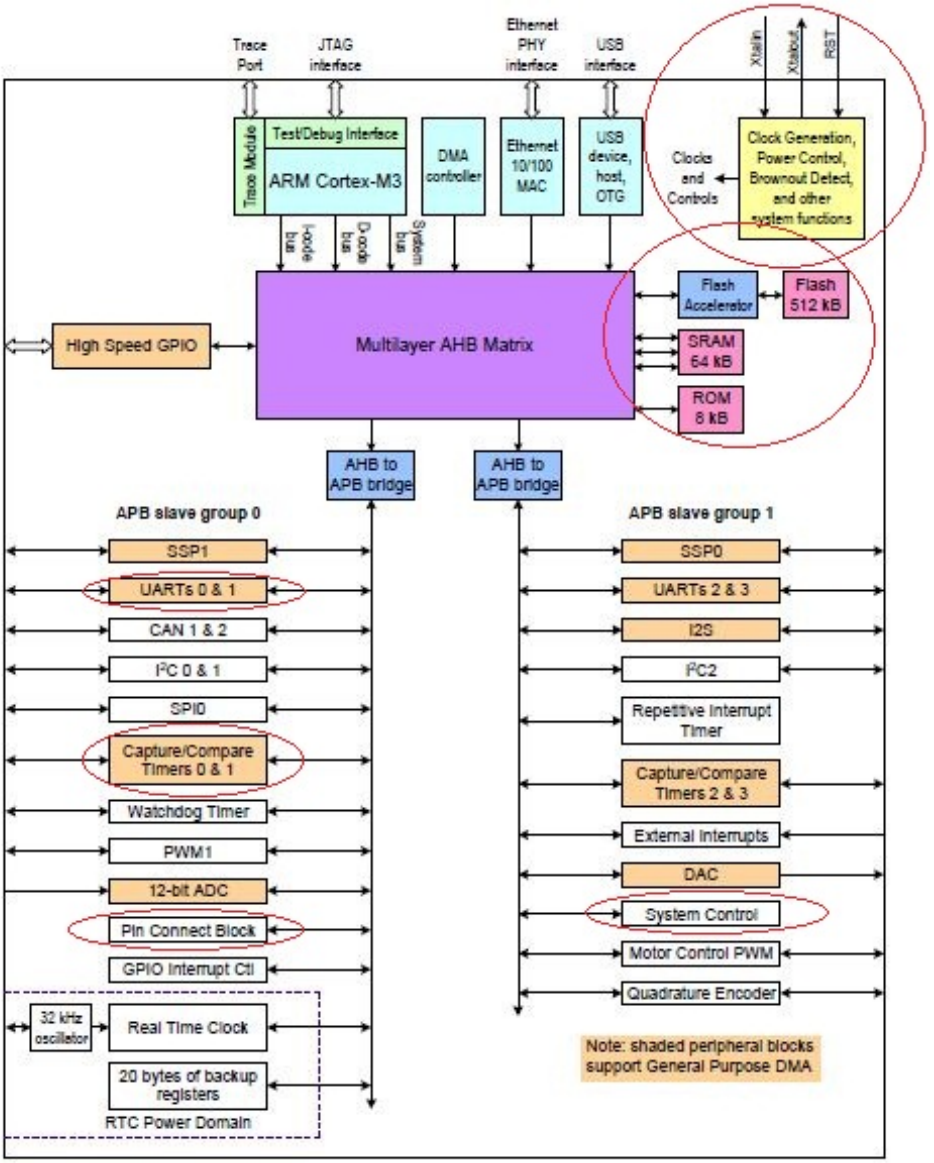


Figure 10.3: LPC1768 Block Diagram. The circled blocks are the ones that we will use in the lab project.

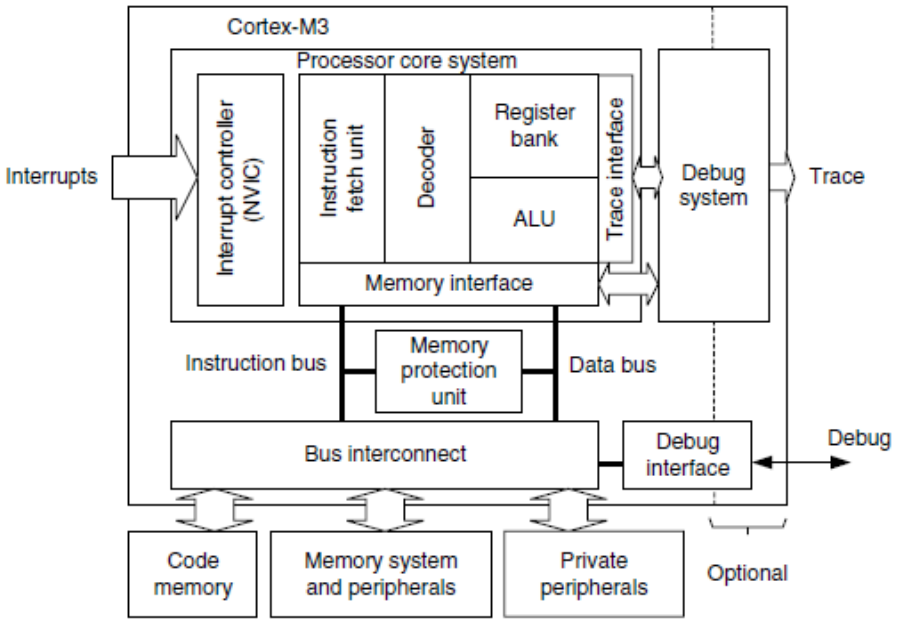


Figure 10.4: Simplified Cortex-M3 Block Diagram[6]

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.

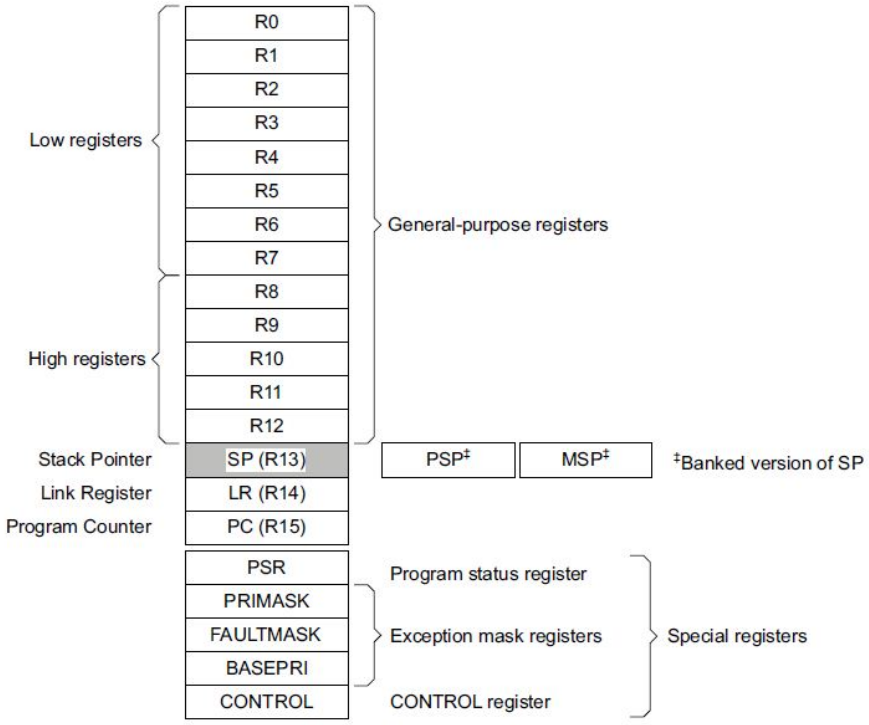


Figure 10.5: Cortex-M3 Registers[4]

- 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.

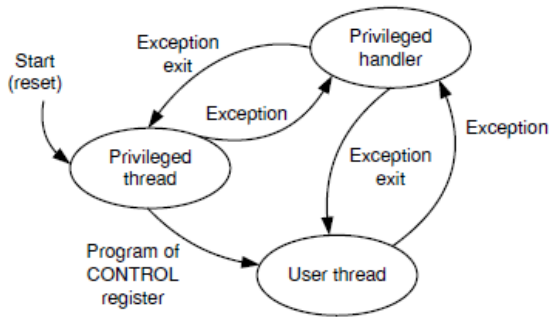


Figure 10.6: Cortex-M3 Operating Mode and Privilege Level[6]

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 intialization 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:

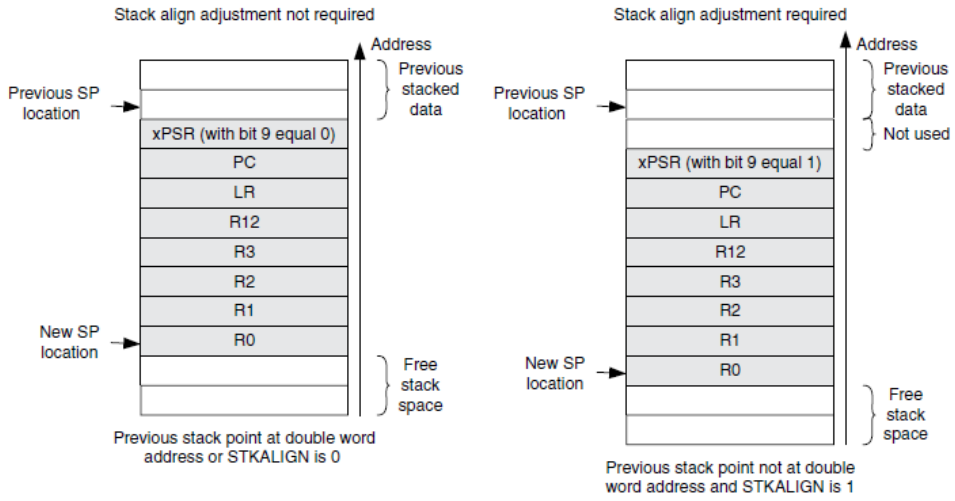


Figure 10.7: Cortex-M3 Exception Stack Frame [6]

- SP: The SP (MSP or PSP) will be updated to the new location during stacking. Stacking from the privileged/unprivileged thread to the first level of the exception handler uses the MSP/PSP. During the execution of exception handler routine, the MSP will be used when stack is accessed.
- PSR: The IPSR will be updated to the new exception number
- PC: The PC will change to the vector handler when the vector fetch completes and starts fetching instructions from the exception vector.
- LR: The LR will be updated to a special value called `EXC_RETURN`. This indicates which stack pointer corresponds to the stack frame and what operation mode the processor was in before the exception entry occurred.
- Other NVIC registers: a number of other NVIC registers will be updated .For example the pending status of exception will be cleared and the active bit of the exception will be set.

### 10.4.3 `EXC_RETURN` Value

`EXC_RETURN` is the value loaded into the LR on exception entry. The exception mechanism relies on this value to detect when the processor has completed an exception handler. The `EXC_RETURN` bits $[31 : 4]$ is always set to `0xFFFFFFF` by the processor. When this value is loaded into the PC, it indicates to the processor that the exception is complete and the processor initiates the exception return sequence. Table 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 | 0xFFFFFFF | 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 |
| 0xFFFFFFFD | 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 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 = _RDWORD(0x10000000); // Setup Stack Pointer
  PC = _RDWORD(0x10000004); // Setup Program Counter
  XPSR = 0x01000000; // Set Thumb bit
  _WDWORD(0xE000ED08, 0x10000000); // Setup Vector Table Offset Register
  _WDWORD(0x400FC0C4, _RDWORD(0x400FC0C4) | 1<<12); // Enable ADC Power
  _WDWORD(0x40034034, 0x00000F00); // Setup ADC Trim
}
LOAD %L INCREMENTAL // Download

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

Listing B.2: The RAM.ini file

# Bibliography


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

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

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

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

[5] 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.