SE 350 Laboratory Project Manual
A Real-time Executive for Keil MCB1700

by

Yiqing Huang
Thomas Reidemeister

Electrical and Computer Engineering Department
University of Waterloo

Waterloo, Ontario, Canada, January 14, 2022

Contents

List of Tables

List of Figures

Preface

The University of Waterloo Software Engineering (SE) SE350 course laboratory project is to design a small real-time executive (RTX) and implement it on a Keil MCB1700 board populated with an NXP LPC1768 microcontroller.

The main purpose of this document is a quick reference guide of the relevant hardware environment and software development tools of the Keil MCB1700 board for completing the laboratory project. To make the manual self-contained, we also include the project description [1] to further guide students.

There are three parts of the document.

- Part I Lab Project Administration Policy
- Part II RTX Project Description
- Part III Design and Implementation Notes
- Part IV Computing Environment and Keil MCB1700 Development Quick Reference Guide

Acknowledgments

We would like to sincerely thank Professor Paul Dasiewicz who originally designed the RTX course project and provided us with detailed notes and sample code. We also own many thanks to our students who did this course project in the past and provided constructive feedback. Professor Sebastian Fischmeister made the Keil Boards and MDK-ARM donations possible. Professor Jim Barby provided timely departmental resource towards the development of the course project, without which this project will not be possible to start. Roger Sanderson provided us with all necessary experiment tools and resources, which we are grateful for. We appreciate that Bernie Roehl has shared his valuable Keil board experiences with us. Our gratitude also goes out to Eric Praetzel who sets up the RTOS lab and also maintains the Keil software on Nexus machines; Laura Winger who managed to customize the boards so that we have the neat plastic cover to protect our hardware. Bob Boy from ARM

[1] The original project description was written by Professor Paul Dasiewicz. The project description included in this manual is a modified version of the original one.

always answers our questions in a detailed and timely manner. Rollen S. D'Souza shared his recent lab teaching experiences with us and this helped us to improve the manual. Thank everyone who has helped.

Part I

Lab Project Policy

Lab Project Administration Policy

Project Group Policy

- **Group Size.** The project is done in groups of four. Five is not allowed and groups of less than four members is not recommended. There is no reduction in project deliverables regardless the size of the project group. Everyone in the group normally gets the same mark. LEARN (https://learn.uwaterloo.ca) is used to signup for groups. *The project group signup is due by 08:30 am EST on Tuesday in the second week of the academic term*. Group sign-up weighs `3%` of the final lab grade (see Table 0.1).
- **Group Split-up.** If you notice workload imbalance, try to solve it as soon as possible within your group or split-up the group as the last resort. Group split-up is only allowed once. Each member in the old group will lose the `3%` signup points. We highly recommend everyone to stay with their 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 becomes intellectual property of each individual in the old group.
- **Group Split-up Deadline.** To split from your group for a particular project deliverable, you need to notify the lab instructor in writing and sign the group slip-up form in the appendix at least one week before the particular project deliverable is due.
- **Collaboration Policy** Explaining concepts to someone in another group, discussing algorithms/testing strategies with other groups, helping someone from another group to debug their code, and searching online for generic algorithms (e.g., hash table) are allowed. Sharing code and test cases with another group, open-sourcing code (e.g., hosting code publicly on GitHub) even after this term, copying/reading other groups' code and test cases, and copying/reading online code and test cases from prior years are not allowed. Any suspected plagiarism or infractions of this honor code will be reported to the appropriate Associate Dean.

Project Submission Policy.

- **Project Deliverables.** The lab project is divided into three deliverables. We will create GitLab repositories for all groups. The submission is through GitLab by tagging the commit for submission with a special tag name (see Table 0.1).
- **Late Submissions.** There are *three grace days*[2] that can be used for project deliverables late submissions without incurring any penalty. *Submission is not*

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

Deliverable	Weight	Due Date	Git Tag Name
P0 Group Sign-up	3%	Jan 11 08:30	
P1 Memory and Task Management	33%	Feb 01 08:30	p1-submit
P2 Message Passing and and Timing Service	32%	Mar 01 08:30	p2-submit
P3 Console I/O and Stress Testing	32%	Mar 29 08:30	p3-submit

Table 0.1: Project Deliverable Weight and Deadlines. All times are in (UTC - 05:00) Estern Standard Time (US & Canada).

accepted if it is more than three days late. Please be advised that to simplify the book-keeping, late submission is counted in the unit of day rather than hour or minute and is rounded up to the nearest day. An hour late submission is one day late, so does a fifteen hour late submission. Unless notified otherwise, we always take the latest submission from the Learn dropbox.

Project Grading Policy.

- **Project Grading Procedure.** The project is graded by demonstration. We will evaluate your submitted commit (i.e. it could be different from your latest commit) during the demo. You will use your own testing cases to demonstrate you have completed the project requirements. We will also provide you our testing cases so that you can see how your code runs with a third party testing suite. We publish a small set of testing cases for each deliverable. We require students to demo that they pass these testing cases as well as some unseen cases. Please be advised that these public testing cases are by no means comprehensive and their main purpose is to get you started with writing you own testing cases by using the automated testing framework we have. Writing your own testing cases to thoroughly test your code is part of the project requirements. Having unseen test cases is a common practice and we usually do not release the unseen cases after the demo.

- **Project Demo** All group members need to present themselves in the demo. The demo time is not allowed to be exceeded. You will get zero points for the part you run out of time to demo. Note the demo session is not a debugging session and the evaluator will not help you to debug your code.

- **Hardware vs. Simulator.** Demos will be evaluated on the board connected to a Microsoft Windows 10 lab machine. Lab machines are accessible through ENGLab remote desktop session when connected to the campus virtual private network (VPN) if it is an online offering. A **15%** penalty will be applied to a deliverable that is only able to function on the simulator but not on the actual hardware.

- **Project Re-grading.** Lab grades are usually finalized at the end of the demo and released by the end of the lab demo week. Re-grading requests need to be directly submitted through LEARN re-grading request dropboxes within two calendar days after the lab grade is released. You will need to write a detailed appeal document to support your request. Acceptable regrading request reasons are:
  - There is a data entry error of your lab grades on LEARN.
  - You find a grading mistake. For example a bug in the unseen test code used in the demo.
  - You find the grading is unfair.

  Name the regrading request document as `G<gid>-P<pid>-regrade.pdf`, where `gid` is your group id and `pid` is the project id of 1, 2 or 3. For example, `G99-P2-regrade.pdf` is the regrading request file submitted by Group 99 to appeal P2 grade. Please do not email us your regrading request. Dropbox submissions will help everyone to keep good track of all regrading requests. Your entire project will be re-evaluated and the chance that the new lab grade may be lower than your original lab grade exists.

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 Solution Internet Policy

Publishing your solutions of lab projects including the source code and design documentation 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 encourage students to use the Piazza (https://piazza.com/uwaterloo.ca/winter2022/se350) to ask the teaching team questions instead of sending individual emails to lab teaching staff. For a question related to a particular deliverable, our target response time is one to two business day(s) before the deadline of the particular deliverable [3]. *After the deadline, there is no guarantee on the response time.*
- **Office Hours.** Lab office hours can be booked online. The booking website is posted in Learn. You must use your school email account when booking the appointment. Otherwise you may not receive the appointment confirmation email.

Important Note

Teaching staff are not permitted to give students direct solution to lab project. Guidance and hints will be provided to help students to find solutions by themselves. This applies to debugging as well. Note that debugging is a non-trivial part of the work of the lab project. Teaching staff will not debug student's program for the student to complete the lab. Teaching staff will be able to demonstrate how to use the debugger and provide case specific debugging tips to help students better debug their programs.

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

Part II

RTX Project Description

Chapter 1

Introduction

1.1 Overview

In this project, you will design a small real-time executive (RTX) and implement it on a Keil MCB1700 board populated with an NXP LPC1768 microcontroller . The executive will provide a basic multiprogramming environment, with five priority levels, preemption, simple memory management, message-based inter-process communication, a basic timing service, system console I/O and debugging support.

Such an RTX is suitable for embedded computers which operate in real time. A cooperative, non-malicious software environment is assumed. The design of the RTX should allow its placement in ROM. Applications and non-kernel RTX processes must execute in the unprivileged level of LPC1768. The RTX kernel will execute in the privileged level [1]. It has 32K of RAM for use by the RTX and application processes. It contains 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 summary of the RTX requirements are listed as follows:

1. **Scheduling Strategy**
   Four user priority levels plus an additional "hidden" priority level for the Null process, preemption, no time slicing, FIFO (First In, First Out) discipline at each priority level.

[1] We do not require application processes to use Process Stack Pointer (PSP). You may use the Main Stack Pointer (MSP) both for your kernel and non-kernel code. However non-trivial implementations that are not required such as using PSP for user processes and using memory protection unit to safe guard kernel sensitive data will be rewarded with bonus marks.

2. **RTX Primitives and Services**
   Refer to the Chapter 2 (Description of RTX Primitives and Services).

3. **RTX Footprint and Processor Loading**
   A reasonably *lean* implementation is expected. No standard C library function call is allowed in the kernel code.

4. **Error Detection and Recovery**
   At minimum, the RTX kernel must detect one type of error: an attempt to `send_message` to or `set_process_priority` of a non-existent process ID. The primitive will return an error code (a non-zero integer value). No error recovery is required. It may be assumed that the application processes can deal with this situation.

Chapter 2

Description of RTX Primitives and Services

This chapter lists the RTX primitive and services. You must implement theses as described and may not modify the prototypes in any way. The primitives listed below will always return a value, either a pointer or an `int` return code. In the latter case, the return code value of `0` indicates a success; non-zero value indicates a failure where applicable.

2.1 The Application Programming Interface

The RTX user-space Application Programming Interface (API) is in `rtx.h`. The `rtx.h` includes `rtx_ext.h` and `commoh.h` files. The `rtx_ext.h` is for students to put their self-defined RTX user API that is not required in this document. Note this is optional and you can leave this file empty if you do not implement not-required API. The `common.h` file contains macro definitions and data structures that both the kernel and the user-space code can include. The `common.h` includes `common_ext.h` file, which is for students to put self-defined macro definitions and data structures to be included by the kernel and the user-space code. Note that one can leave this file empty if no such extra macro definitions or data structures are needed. One *should not modify the contents of rtx.h and common.h*. A third-party testing software will include these two files. The `rtx_ext.h` and `common_ext.h` files can be freely modified.

2.2 Memory Management

The RTX supports a simple memory management scheme. The memory is divided into blocks of fixed size, The size of each memory block is `MEM_BLK_SIZE` bytes and the number of these blocks is `MEM_NUM_BLKS`, where both macros are defined in the

`common.h` file. The blocks can be used by the requesting processes for storing local variables or as envelopes for messages sent to other processes (see Section 2.5). A block which is no longer needed must be returned to the RTX.

Two primitives are to be provided.

```
void *request_memory_block();
```

The `request_memory_block` primitive allocates a free memory block and returns a pointer to the memory block to the calling process. If no free memory block is available, the calling process is blocked until a free memory block becomes available. If several processes are waiting for a memory block and a block becomes available, the highest priority waiting process will get it. Within the same priority level, a first-come-first-served policy applies.

```
int release_memory_block(void *memory_block);
```

The `release_memory_block` primitive returns the memory block to the RTX. If there are processes waiting for a block, the block is given to the highest priority process, which is then unblocked. The caller of this primitive never blocks, but could be preempted (see Section 6.1.2). Thus, it may affect the currently executing process.

2.3 Processor Management

One primitive is to be provided.

```
int release_processor();
```

The `release_processor` primitive enables the process to relinquish the processor (the calling process voluntarily releases the processor). The calling process remains ready to execute and is put at the end of the ready queue of the same priority. The RTX will make new scheduling decision to determine the next process to run. Another process may possibly be selected for execution.

2.4 Process Priority

Process priorities have an integer priority value `(0, 1, 2, 3, 4)` where `0` is the highest priority level. Two primitives are to be provided to set and get the process priority.

```
int set_process_priority(int process_id, int priority);
```

The `set_process_priority` primitive sets the priority of the process with `process_id` to the value given in priority. A process may change priority of any process (including itself) except for i-processes (see Section 3.2). The priority of the null process may not be changed from level 4 and it is the only process that can be assigned to level 4 (see Section 3.1.1). The caller of this primitive never blocks, but could be preempted. If the `priority` is higher than the priority of the current running process, and the process identified by `process_id` is ready to run, then the process identified by the `process_id` preempts the current running task. Otherwise, the current running process continues its execution. When the specified `process_id` does not exist or the `priority` is invalid, the function returns `-1`.

```
int get_process_priority(int process_id);
```

This `get_process_priority` primitive returns the priority of the process identified by the `process_id` parameter. For an invalid `process_id`, the primitive returns `-1`.

2.5 Interprocess Communication

The RTX will support a message-based Interprocess Communication (IPC) discussed in lectures. Messages are carried in envelopes (memory blocks, see below) with a header which is less than `64` bytes. Two IPC primitives will be implemented.

```
int send_message(int process_id, void *message_envelope);
```

The `send_message` primitive delivers to the destination process identified by `process_id` a message carried in a message envelope. The `message_envelope` argument is a pointer to the following general form structure:

```
struct msgbuf {
#ifdef K_MSG_ENV
   void *mp_next; /* ptr to the next message */
   int m_send_pid; /* sender pid */
   int m_recv_pid; /* receiver pid */
   int m_kdata[5]; /* extra 20B kernel data place holder */
#endif
   int mtype;      /* user defined message type */
   char mtext[1]; /* body of the message */
};
```

The fields that are enclosed in the `#ifdef K_MSG_ENV` and `#endif` form a data structure that kernel uses to manage the message passing. If the `K_MSG_ENV` is defined, then these fields will be exposed to user space. The user processes however should not access these fields even they are exposed to the user space. The kernel are responsible for modifying these fields. If the `K_MSG_ENV` is not defined, then the afore mentioned kernel fields will be hidden from the user space and the kernel is responsible to find space to store these fields.

User processes are permitted to read and write the `mtype` and `mtext` fields. The `mtype` field takes a user defined message type. And the following macro defines the value of this field:

`DEFAULT`

A general purpose message.

`KCD_REG`

A message to register a command with the Keyboard Command Decoder Process (see Sectoin 3.1.2)

`KCD_CMD`

A message that contains a command to be handled by the receiving process (see Section 3.1.2) .

`CRT_DISPLAY`

A message to display the message body to the RTX console.

`KEY_IN`

A message that contains an input key (may contain multiple characters if it is a control key) from keyboard.

These macros are defined in `common.h`, which is included by `rtx.h` file as follows.

```
#define DEFAULT 0
#define KCD_REG 1
#define KCD_CMD 2
#define CRT_DISPLAY 3
#define KEY_IN 4
```

You are free to add more user defined message type macros in the `common_ext.h` file. Please note that both the host computer and the board are little-endian systems.

The `mtext` field is an array (or other structure) whose size is limited to the size of one memory block less the total size of all other fields in the `msgbuf` structure. The primitive changes the state of destination process to ready-to-execute if appropriate. The sending process is preempted if the receiving process was blocked waiting for a message and has higher priority, otherwise the sender continues executing.

```
void *receive_message(int *sender_id);
```

The `receive_message` is a blocking receive. If there is a message waiting, a pointer to the message envelope containing it will be returned to the caller. If there is no such message, the calling process blocks and another process is selected for execution. The sender of the message is identified through `sender_id`, unless it is NULL. Note the `sender_id` is an output parameter and is not meant to filter which message to receive.

2.6 Timing Services

Unprivileged level processes obtain the timing service from RTX by the following primitive [1].

```
int delayed_send(int process_id, void *message_envelope, int delay);
```

[1] Unprivileged processes should not read kernel timer data directly. You are free to add a primitive to return the kernel clock ticks to unprivileged processes should you find the delayed_send primitive is not sufficient to provide the timing service you need. Self-defined user API can be added to the `rtx_ext.h` file

The calling process does not block. The message (in the memory block pointed to by the second parameter) will be sent to the destination process (`process_id`) after the expiration of the delay (timeout, given in *millisecond* units).

Chapter 3

Required Processes

This chapter describes the processes which you must implement for the project. The initial priority of processes is in the range of `{0, 1, 2, 3, 4}` and is left as a free choice if it is not specified explicitly. The priority level 4 is a *hidden* priority level that is reserved for the Null process (see Section 3.1.1. Note the I-processes (see Section 3.2) are not scheduled based on the priority. They are triggered by interrupts hence has no initial priority.

3.1 System Processes

System processes are those processes needed by the system to perform basic services (scheduling and I/O). You will need to make your design choice to determine which system processes require privileged level and which system processes may operate at unprivileged level.

3.1.1 The Null Process

This process runs as the lowest priority process (level 4) in the RTX. The Null process is the only process assigned to level `4`. Level `4` is basically a"hidden" priority level reserved for the Null process. This preserves the four levels of user priorities (levels `0`, `1`, `2` and `3`). Process id `0` is reserved for the null process. Initially, the following pseudo code can be used to design your null process:

```
loop forever
      release the processor
end loop
```

Once you have preemption working, then the "`release the processor`" line could be removed from the infinite loop.

3.1.2 System Console I/O Processes

The system console is used for communication with the RTX and application processes. It consists of two devices: keyboard and CRT display. These two devices communicate serially with the microcomputer; using the receive and transmit lines of one of the two RS-232 ports.

The RTX will include two system processes, the Keyboard Command Decoder (KCD) process and the CRT Display process. These processes work in cooperation with the UART interrupt handler i-process.

The Keyboard Command Decoder (KCD) Process

A keyboard command starts with the prompt character `%`, followed by a single letter command identifier and possibly additional command data. For example, `%WS 12:45:00` is a command that the wall clock process handles to start the wall clock display and setting the current time to `12:45:00` (where the command format is `%WS hh:mm:ss`). The `W` is a single command identifier and the `S 12:45:00` following `W` are additional command data.

The keyboard command decoder (KCD) process is an unprivileged user-space process. It responds to two types of messages: command registration (`KCD_REG`) and console keyboard input (`KEY_IN`). The former contains the command identifier and the process id of the process to which such commands are to be delivered when entered on the console keyboard. The processing of messages received depends on their type:

- Command Registration
  The command identifier is associated with the process id of the registrant. To register a command with KCD, a process will send a message with type of `KCD_REG` and the body of the message starts with `%` followed by the command identifier (for example `W` for wall clock related commands) and a null character to terminate the message (See Q3 in Section 6.3 for a more detailed example). KCD forwards any input command with a registered identifier to the corresponding registered process. Processes can register an already registered command identifier. KCD always forwards a command to the latest process that has registered the command's identifier.

- Keyboard Input
  The UART i-process (see Section 3.2) forwards any key input to the KCD using the `KEY_IN` message. The KCD forwards this message to the CRT Display process to be echoed back to the console. KCD also queues the input keys inside its internal buffer. Upon receiving the "enter" key, KCD construct a single string using the queued characters inside the buffer. If the string starts with `%` followed by a registered command, then the KCD will forward the string to the corresponding registered process using a `KCD_CMD` message. The receiving

process is responsible for handling the command. The message body contains the command string (including the % character and the "enter" key).

If the command identifier is not registered, then KCD ignores the string and sends a message to the CRT display process to display "Command not found". If the string does not start with % or the length of the string exceeds the capacity of the message envelope, then KCD will ignore the string and send a message to CRT display process to display "Invalid command".

The relationship among the KCD process, CRT display process, UART I-Process and a process that has a command registered is illustrated in 3.1.

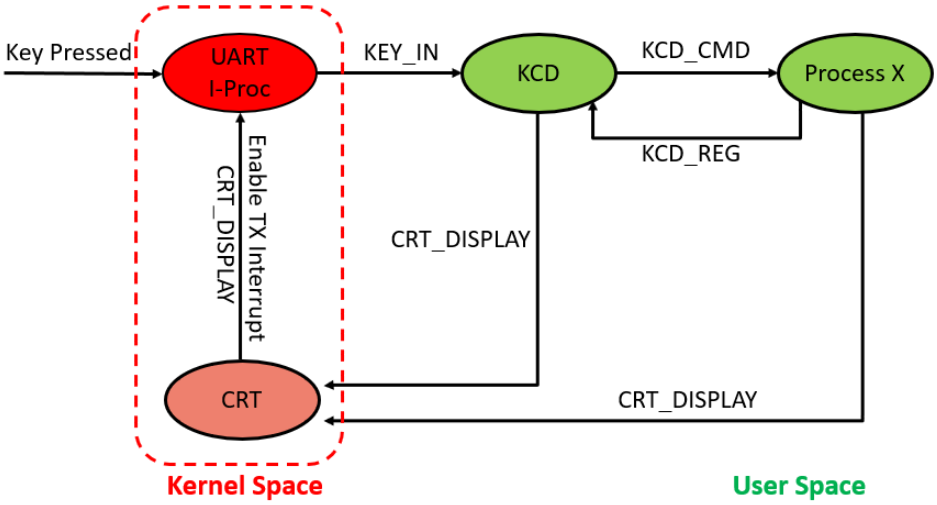

Figure 3.1: System console I/O processes, uart i-process and a command handling process communication relationship.

The CRT Display Process

The CRT display process is a privileged process. It responds to only one message type: a CRT display request. The message type is `CRT_DISPLAY`. The message body contains the character string to be displayed to the RTX console. The string may contain control characters (e.g. newline and ANSI escape sequences et. al.). In printing to the console display, the process must use the UART i-process. The CRT display process forwards the received display request to the UART i-process by re-using the message envelope it received. It than enables the UART transmission interrupt. Any message received but not forwarded to the UART i-process is freed using the `release_memory_block` primitive.

3.2 Interrupt Processes (I-Processes)

Two interrupt handling processes are required:

3.2.1 The Timer I-Process

The timer i-process is executed each time a hardware timer interrupt occurs. The timer i-process should handle the delivery of delayed send messages after the required time has expired.

3.2.2 The UART I-Process

The UART i-process uses interrupts for both the transmission and receiving of characters from the serial port. No polling or busy waiting strategies may be implemented. The UART i-process forwards characters (or commands) received to the KCD by sending it the `KEY_IN` message, and also responds to messages received from the CRT display process (i.e. `CRT_DISPLAY` type message) to transmit characters to the serial port (see Figure 3.1). The UART i-process turns of the transmit interrupt once it finishes transmitting the characters to the host machine serial port. The receiving interrupt is always on.

Three Hot Keys

The UART i-process is used to provide debugging services which will be used during the demonstration. Upon receiving a specific character (a hot key) as input, the UART i-process will print debugging information to the RTX system debug terminal, which is a polling terminal. The three required hot keys are:

- `!` hot key displays the processes currently on the ready queue(s) and their priority.
- `@` hot key displays the processes currently on the blocked on memory queue(s) and their priorities.
- `#` hot key displays the processes currently on the blocked on receive queue(s) and their priorities.

It is up to you whether you want to echo back the hot key input either to the debug terminal or the RTX console terminal. Not echoing it back to any terminal is an accepted solution.

As well, you are free to implement other hot keys to help in debugging. For example, the `$` can be a hot key which lists the processes, their priorities, their states. The

`&` can be yet another hot key which prints out the number of memory blocks available. Like all other debug prints, the hot key implementation should be wrapped in

```
#ifdef _DEBUG_HOTKEYS
...
#endif
```

preprocessor statements and should be turned off during automated testing. If the automated test processes fail, you may be asked to turn the hot keys on again in determining why the test processes are failing. Another hot key `^` may be used to display recent interprocess message passing. A (circular) log buffer keeps track of the 10 most recent `send_message` and `receive_message` invocations made by the processes; upon receiving a specific hot key, these most recent `10` sent and `10` received messages are printed to the debug con-sole. The number `10` is used only as an example. The information printed could contain information such as:

- Sender process id
- Destination process id
- Message type
- First 16 bytes of the message text
- The time stamp of the transaction (using the RTX clock)

3.3 User Processes

These processes operate at *unprivileged level* and will be used to demonstrate the operation of your system.

3.3.1 User-Level Test Process

Write up to six user-space test processes to test your own OS. These test processes should run at unprivileged level and do not assume any kernel level data structures. These test processes only call the RTX user-space APIs (see Section 2.1). The test processes should provide at least two and at most six test cases and finish testing within three minutes. The process id `1`, `2`, `3`, `4`, `5`, and `6` are reserved for these processes.

Since the test processes have no knowledge of your detailed internal design, they only invoke the functions specified by the RTX user-space API. The test processes can use the timer that is not used by the RTX for timing testing.

We require the testing results to comply with the following format and you output the results to the debug UART terminal by polling (i.e. UART1):

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

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

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

3.3.2 24 Hour Wall Clock Display Process

This process registers itself with the Keyboard Command Decoder process as the handler for the `%W` command.

- The `%WR` command will reset the current wall clock time to `00:00:00`, starts the clock running and causes display of the current wall clock time on the console CRT. The display will be updated every second.
- The `%WS hh:mm:ss` command sets the current wall clock time to `hh:mm:ss`, starts the clock running and causes display of the current wall clock time on the console CRT. The display will be updated every sec-ond.
- The `%WT` command will cause the wall clock display to be terminated.

3.3.3 Set Priority Command Process

This process registers itself with the Keyboard Command Decoder process as the handler for the `%C` command. The `%C` command has two parameters:
`%C process_id new_priority`

The `process_id` and `new_priority` are integers. This command changes the priority of the specified process, `process_id`, to `new_priority`. The change in priority level is immediate. It could also affect the target processs position on a ready queue or a blocked resource queue. The parameters must be verified to ensure a valid `process_id` and priority level are given. A `%C` command with illegal parameters will be ignored with an error message of "Invalid command input" printed on the console.

3.3.4 Stress Test Processes

An important category of software tests are the stress tests. These tests seek to verify the behaviour of the system under heavy stress scenarios. One such scenario is depletion (or near depletion) of system resources. For the demonstration of this project, you will implement three processes whose behaviour is described below. The stress scenario being tested is depletion of memory blocks.

Process A:

```
p <- request_memory_block
register with Command Decoder as handler of %Z commands
loop forever
  p <- receive a message
  if the message(p) contains the %Z command then
    release_memory_block(p)
    exit the loop
  else
    release_memory_block(p)
  endif
endloop
num = 0
loop forever
  p <- request memory block to be used as a message envelope
  set message_type field of p to count_report
  set message_text field of p to num
  send the message(p) to process B
  num = num + 1
  release_processor()
endloop
/* note that Process A does not de-allocate any received envelopes in the
     second loop */
```

Process B:

```
loop forever
  receive a message
  send the message to process C
endloop
```

Process C:

```
perform any needed initialization and create a local message queue
loop forever
  if (local message queue is empty) then
    p <- receive a message
  else
    p <- dequeue the first message from the local message queue
  endif
  if msg_type of p == count_report then
    if message_text field of p is evenly divisible by 20 then
      send "Process C" to CRT display using msg envelope p
      hibernate for 10 sec
    endif
  endif
  deallocate message envelope p
  release_processor()
endloop
```

The line "`hibernate for 10 sec`" is further expanded as:

```
q <- request_memory_block()
use q to delayed_send to itself with 10 sec delay and msg_type=wakeup10
loop forever
  p <- receive a message //block and let other processes execute
  if (message_type of p == wakeup10) then
    exit this loop
  else
    put message (p) on the local message queue for later processing
  endif
endloop
```

Notes:

- Process C has a local message queue (distinct from the incoming message queue maintained by the RTX) onto which it enqueues (in FIFO order) messages which arrive while it hibernates. It processes these mes-sages later.
- For your own testing, set the priority levels for processes A, B and C to values which are most likely to cause memory block depletion in the RTX. During project demo, you may be asked to re-initialize your RTX with TA/instructor specified priorities for A, B, and C and vary the total number of message envelopes available.

3.4 Process ID Assignment

To facilitate the project evaluation, we enforce the process ID assignment rule listed in Table 3.1.

process ID	Process	Process ID	Process
0	Null	8	B
1	Test 1	9	C
2	Test 2	10	Set process priority process
3	Test 3	11	Wall clock display
4	Test 4	12	KCD
5	Test 5	13	CRT
6	Test 6	14	Timer i-process
7	A	15	UART i-process

Table 3.1: Required RTX Processes

Chapter 4

RTX Initialization

To make the RTX more generally applicable, the RTX will be configured at initialization through the `rtx_init` system call. The initialization contains three parts:

1. Memory configuration: memory block size, number of memory blocks created;
2. System process initialization: system process control blocks and stacks created;
3. Application process initialization: user process control blocks and stacks created.

Chapter 3 (Required Processes) lists the processes to be created. Each entry contains the following data:

- process id,
- priority,
- stack size,
- start address, and
- for system processes, whether the process is an i-process.

All initializations must take place after the RTX execution starts.

Chapter 5

Deliverables and Demonstration

5.1 Deliverable

The project has three deliverables, which will be evaluated through third-party automated testing grading software. The deliverables are as follows:

5.1.1 RTX P1

This is the source code of a tiny kernel which provides memory management, processor management and process priority services. You need to implement

- APIs listed in Sections 2.2, 2.3 and 2.4;
- processes in Sections 3.1.1 and 3.3.1 [1]; and
- the corresponding part in Chapter 4.

5.1.2 RTX P2

This is the source code of an RTX that builds on RTX P1. It added interprocess communications by message passing service and timing service.

- add APIs listed in Sections 2.5 and 2.6;
- finish implementing the timer i-process as described in Section 3.2.1.
- enhance user test processes in Section 3.3.1 so that they test this version of the kernel; and
- implement the corresponding part in Chapter 4.

[1] Note you need to write the six user testing processes to demonstrate that your implementation meets the requirements.

5.1.3 RTX P3

This is the final RTX source code to implement the specifications in Chapters 2, 3, and 4 based on RTX P1 and RTX P2 implementations done previously. The main work is to provide console I/O services and perform stress testing of your final RTX. To be more specific, you will implement the following items:

- the rest of system processes (KCD and CRT display processes) as described in Section 3.1
- the UART i-process as described in Section 3.2.2
- the wall clock process as specified in 3.3.2
- the set process priority processes as specified in 3.3.3
- the three stress testing processes as specified in 3.3.4
- the user test processes in Section 3.3.1 so that they test the final version of the kernel
- the corresponding part in Chapter 4.

5.2 Demonstration

The three milestones of the project are evaluated by demonstration.

5.2.1 Demo Session Form Submission

Before the first demo session, everyone in the group needs to sign the demo session form (available in LEARN Lab → Forms) and submit it to the demo form Dropbox on Learn. The agreement established in this form applies to all demo sessions.

5.2.2 The Demo Policy

- The project is evaluated during the assigned demo time slot. It is recommended that you arrive the lab about 15 minutes earlier to set up your demo environment. The evaluation will end when the reserved demo time slot ends regardless whether you have finished demoing/answering all the items on the evaluation form or not. For the items that you are not able to finish during the reserved demo time slot, you will receive zero mark.
- During the demo of your project, your original submitted RTX implementation archive file will be retrieved and the demo will use those files in the archive. No substitutions are allowed. It is student's responsibility to verify the submitted

files are the correct version. We highly recommend to download your submission right after you submit it to confirm it is the correct version you want to submit.

- The demo is not some dry run to do additional debugging under ”live” conditions. If minor bugs are discovered during the demo, depending on the complexity, you might be allowed to fix the bug, recompile and download and continue the demo with some penalty points deducted. Under no circumstances will file replacements be allowed during the demo. “fixes” are basically limited to minor manual editing of a source file. If you want to fix minor bugs, the fix needs to be finished during the reserved demo time slot.
- You are only allowed to demo once. If you decide to go for the bonus demo [2], you are not allowed to do a regular demo again even you are unable to complete the bonus project demo.
- ALL project group members are required to be presented during the demonstrations.

5.2.3 The Demo Procedure

P1 and P2 Demo Procedures

- **Basic Functionality Demo**
  You will need to demonstrate you have successfully completed the required APIs by using your own six test processes.
- **Source Code Spot Check**
  An evaluator will ask each group member implementation questions.

P3 Demo Procedure

- **Basic Functionality Demo**
  You will demonstrate the basic functionality of the RTX through commands line input. The evaluator will
  - start, observe and stop wall clock display by using various `%W` commands;
  - check the hot keys output; and
  - set priority of processes by using `%C` commands.

  test of wall clock display
- **Stress Test**
  Your RTX will go through a stress testing demo by using processes A, B and C. The evaluator will

[2] Bonus demo may not be available for some offerings

  - reinitialize RTX with `N` (`N=32`) envelopes, one combination of processes A,B and C priorities;
  - start test process activity by `%Z` command;
  - observe system operation (wall clock display as indicator), may also display trace buffer (is your have it implemented);

  The above will be repeated with two other combinations of processes A,B and C priorities.

- **Contribution Check**
  Each group member will be asked what he/she has contributed to the RTX (i.e. P1, P2 and P3) implementation.

5.3 Third-party Testing and Source Code File Organization

We will provide third party testing object code to replace the six testing processes written by students during the demo as part of the RTX functionality testing. You will need to maintain the file organization of the project skeleton in the starter code. There are dos and don'ts that you need to follow.

Don'ts

- Do not move any file from the `src` directory to any other directories;
- Do not change the file names under the src directory;
- Do not make any changes of the contents of the `rtx.h` and `common.h` files;
- Do not change the existing function prototype in the given `k_*.[ch]` files; and
- Do not modify the `ae.h` files.

Dos

- You are allowed to add new self-defined functions to `k_*.[ch]`;
- You are also allowed to create new `.h` and `.c` files; [3].
- The newly created `.h` file is allowed to be included in the `k_*.c` file;

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

- Any new files you add to the project can be put into either the `src` directory or other directories you will create.
- You are allowed to make modifications of existing function contents in `k_*.c` as long as you keep the existing function prototype unchanged.
- You are allowed to make modifications of existing function contents in `ae.c` as long as you keep the existing function prototype unchanged.

Note that the `main_svc.c` calls third-party testing by calling `ae_init` function which the third-party testing software implements. The function prototypes of this function does not change. But the implementation of this function may change in real testing. Do not delete the lines in the `main_svc.c` where this function is invoked. During the third-party testing, the files with prefix of `ae_` will be replaced by more complicated testing cases than the ones published on GitHub.

Part III

Design and Implementation Notes

Chapter 6

Frequently Asked Questions

We list frequently asked questions with answers in this chapter.

6.1 Processor Management

This section contains the frequently asked questions and answers of the following topics:

- context switching,
- scheduling and preemption .

6.1.1 Context Switching

Q1: In tutorial slides, it showed our processes as having 5 states: Ready, Blocked on Resource, Waiting for Message, Executing, and Interrupted. Will these five states be sufficient for our lab, or should we, for example, have a"New" state (as in the context switching sample code on GitHub)?

A1: The NEW state says a process which has never been run before is now ready to run. Tutorial slides are at a higher abstraction level. If you feel adding a new state will help you to manage the processor, then feel free to add more states. The NEW state on GitHub can be considered as a special READY state in tutorial slides.

Q2: If a process completes itself without calling `release_processor()`, should it be taken off the ready queue automatically? Should the next process be dispatched?

A2: The project requires processes never terminate (i.e. they never exit). You can keeping calling the `release_processor()` after all useful actions are completed to prevent the process from terminating.

Q3: What does the `__set_MSP()` function do, and how is it able to update the program counter/link register to switch to a new process?

A3: This function sets the MSP to the value supplied. Each process has its own stack, by switching between stacks, we achieve the purpose of context switching.

Q4: How does saving/loading process contexts work in assembly code? Is there any part of the sample code which would help us understand this?

A4: You will need to use assembly code to access C data structures. Section 9.4 in the SE350 lab manual provides some code excerpts on this topic. The starter code https://github.com/yqh/SE350/blob/master/manual_code/Context_Switching/src/k_process.c shows how to context switch between two processes. Please be aware of the assumptions made at the beginning of the code comments block.

Q5: Where to put the PCB priority queues?

A5: The PCB queue structure memory can either be allocated at compile time (i.e. inside the image of your RTX) or at run time (i.e. when you do memory initialization in RTX.).

Q6: What is the initial value of PC for each process?

A6: From coding point of view, each process starts in a function. The entry point of this function (i.e. the address of this function) is the initial PC value of the process. Lines 73-78 of https://github.com/yqh/SE350/blob/master/manual_code/Context_Switching/src/ae.c#L73-78 shows an example of setting initial PC values of two user processes.

6.1.2 Scheduling and Preemption

Q1: What is preemption?

A1: Preemption in this project means when a process whose priority is higher than the priority of the current running process becomes ready, then the current process should be taken out of the processor and let the higher priority ready process to run. In other words, the preemption will guarantee a higher priority ready process will run before a lower priority ready process. It implies that if process P has higher priority than process Q, then at any moment, the system should not allow that P is in READY state and process Q is in RUN state.

Here is a more detailed example: assume we have two processes P and Q. priority(P) > priority(Q). P is blocked on some sort of resource. Q is currently running. Preemption means if an interrupt (software or hardware) happens and it results in a process (say, P) with higher priority than the current running process (say, Q) changing state to READY, then when the interrupt finishes, P will run instead of Q. We say P preempts Q.

Q2: In the project description, when it says a process *may* get preempted, does this imply preemption is not mandatory?

A2: Preemption is mandatory for the entire project and for very single deliverable. However a preemptive API call may not necessarily cause a preemption to happen because it depends on what the values you pass into the API input parameters. For example, `set_process_priority` is a preemptive call. It takes two parameters, a PID and a priority. Calling this function may or may not cause preemption depending on the values of the input parameters of this function call and priorities and states of other processes in the system. Assume we only have two processes P1 and P2. Both are at MEDIUM priority. P1 is currently in RUN state and P2 is in READY state. Example 1: P1 calls set process priority API to lower down itself to LOW priority. This case, P1 will be preempted by P2. P2 will start to RUN after the API call. Example 2: P1 calls set process priority API to increase itself to HIGH priority. This case, P1 will NOT be pre-empted by P2. P1 will continue in RUN state after the API call.

Q3: Which deliverable requires preemption to be implemented?

A3: Preemption is required for all deliverables (i.e. P1, P2 and P3).

Q4: Since we aren't using timers/interrupts in P1, I was a little unclear what sort of preemption was needed. One method we discussed having each system call call the scheduler, instead of possible returning back to the user code. Can you provide some more details on this though?

A4: Suppose a process calls `set_process_priority` to lower down its own priority, then it might be preempted due to this function call. We do not have hardware interrupts in P1, however we do have software interrupt `SVC`. There are other scenarios a preemption could happen as well in P1.

Q5: What is the initial priority of each process?

A5: Except for the null process, which should be assigned a hidden priority of 4 and the I-processes where the user level priority concept does not apply, all other processes take priority levels from the range of `{0, 1, 2, 3}`. You decide each process's initial priority when demoing your kernel with your own test processes. You want to make your initial priority settings reasonable. For example certain priority combinations will put the system into a deadlock. Unless you want to demonstrate the deadlock as a test case, you don't select this type of initial priority settings. We will ask you to set priority of some processes to certain levels during the demo.

Q6: In the lab tutorial slides, four priority queues are show. Is it OK to implement the four priorities by just using one queue?

A6: The detailed implementation is left as a free choice. So the short answer is Yes. However you may want to think about performance difference between one queue approach and the four queues approach.

6.2 Memory Management

This section contains frequently asked questions regarding memory management.

Q1: Where do we allocate memory for the blocked resource queue?

A1: Similar as the PCB queues, you could either allocate the queue in compile time (i.e. inside the image of your RTX) or at run time (i.e. when you do memory initialization in RTX). Keep in mind this only applies to the memory queue structure itself, not the nodes (i.e. the memory blocks) that the memory queue holds. The nodes are outside the RTX image.

Q2: Do we even need the block resource queue? Can't we use the array of `PCBs` and `PROC_INIT` to get blocked and priority?

A2: You are free to implement the RTX the way you want as long as the final project satisfies the required behavior of the APIs and the processes. You have the freedom to make data structure and algorithm choices, though different choices imply different performance of your RTX. Assume that you do not have a blocked on resource queue, when a certain resource becomes available, you will need to traverse the entire pcb table to find the process that is waiting for the resource and try to unblock it, this probably will be OK if you only have small number of pcbs, but will show performance problem when the number of pcbs grow.

Q3: In our `request_memory_block` function for the kernel, how do we know which process is requesting memory?

A3: Normally the kernel will have a global variable that points to the current running process. The Context Switching sample project names this variable `g_current_process`. So you know it is this process that is requesting the memory.

Q4: If we want to get the end of RAM, instead of hardcoding `0x10008000`, can we use `Image$$RW_IRAM1$$RW$$Limit`? Or some other symbol?

A4: The end of RAM can only be hard-coded since it is tied to the specific hardware. The `Image$$RW_IRAM1$$RW$$Limit symbol` is put at the end of the Image and it should always be less than `0x10008000`.

Q5: Is each stack element 8 bytes? Or is it 4 bytes?

A5: Each stack element is 4 bytes.

Q6: Why does the stack pointer need to be 8-byte aligned on ARM?

A6: The ARM Architecture Procedure Call Standard (AAPCS) requires stack to be double-word aligned at a public interface. The AAPCS is posted in Learn under manufacture manual section for your reference.

Q7: I'm looking at `k_memory.c`. What are these 8-byte alignment shenanigans?

```
U32 *gp_stack;
/* code to initialize gp_stack skipped */
if ((U32)gp_stack & 0x04) { /* 8 bytes alignment */
   --gp_stack;
}
```

A7: The code basically checks whether the `gp_stack` is aligned by four bytes (i.e. bit AND with binary 100), but not aligned by eight bytes. If yes, then decrement it by four bytes to make it eight bytes aligned. The `gp_stack` is a U32 *, the compiler will give it an address of four bytes aligned. So it is either eight bytes aligned or not eight bytes aligned, but always four bytes aligned. Because it is U32 *, pointer arithmetic `gp_stack` decrements it by four bytes and we get eight byte alignment.

Q8: How do we assign a memory block to a PCB?

A8: When a process (say process P) invokes `release_memory_block` primitive, if there are processes that are blocked on memory, then the OS should allocate this memory block to the highest waiting-for-memory process that waited longest (say process Q) instead of putting the memory block back to the pool. This results in changing the state of Q to ready and move Q from the blocked queue to the ready queue. However this may not necessarily mean P will be switched out and Q will start to run. After you put Q into the ready queue, the schedule will make the scheduling decision and select the next to run process. It could be P, it could be Q and it even could be some other ready processes that is in the same priority level as P. One way is to have some variable defined in PCB to remember this piece of memory block. Other approaches exist.

Q9: How big is each memory block and how many memory blocks are there?

A9: The minimum size of each memory block is 128 bytes (see Section 2.2). During your development, you decide how many memory blocks the RTX can manage. At minimum, your RTX should be able to manage 32 memory blocks (see `common.h` for compile-time macros).

6.3 Interprocess Communication (IPC)

Q1: What is a message envelope?

A1: The project requires a message-based IPC scheme. Messages are carried in shared memory blocks. A process writes a message into a shared memory block, sends a pointer to the memory block to another process and receiving

process reads the message from the memory block. A shared memory block used for message passing is a "message envelope.

Q2: What is the format of the message envelope?

A2: The message envelope contains two data structures. One is for the kernel management purpose. The kernel is supposed to write to this data structure. For example it may contain some linked list pointers, sender PID, receiver PID and other kernel data. This data structure can be either put in the kernel space or user space. The compilation macro `K_MSG_ENV` controls whether the OS wants to expose this data structure to the user space or not. The second data structure is exposed to user space. It contains the message type and the actual message body. The `struct msgbuf` data structure is defined in Section 2.5. See Q1 in Section 6.6 on how to use the Keil IDE to define compilation macros.

Q3: Where does the message envelope come from and how does user process use it to send a message?

A3: The user process calls `request_memory_block` primitive to request a memory block to be used as a message envelope and the user will start to fill the `struct msgbuf` non-kernel data structure defined in Section 2.5. For example, the following user process (wall clock process) code will send a command registration message to process KCD (Keyboard Command Decoder).

```
struct msgbuf *p_msg_env = (struct msgbuf *) request_memory_block();
p_msg_env->mtype = KCD_REG;
p_msg_env->mtext[0] = '%';
p_msg_env->mtext[1] = 'W';
p_msg_env->mtext[2] = '\0';
send_message(PID_KCD, (void *)p_msg_env);
```

Q4: The lab manual has the following structure:

```
struct msgbuf {
#ifdef K_MSG_ENV
   void *mp_next; /* ptr to the next message */
   int m_send_pid; /* sender pid */
   int m_recv_pid; /* receiver pid */
   int m_kdata[5]; /* extra 20B kernel data place holder */
#endif
   int mtype;     /* user defined message type */
   char mtext[1]; /* body of the message */
};
```

The slides have something like this

```
msg t (an envelope) {
   next message
   Sender PID
   Destination PID
   Message Type
   Message Data
};
```

which one is correct?

A4: The slides give you one possible message envelope data structure as an example to illustrate the idea of a message envelop. The example in slides exposes kernel data structure to user space. The manual uses the compilation macro `K_MSG_ENV` to expose the next message, Sender PID and Destination PID fields to the user process view. If you want to hide these fields in the kernel, then do not define the compilation macro and instead find some kernel memory space to save these data. The manual specification makes it possible to hide more information from the view of a user process by using a compilation macro of `K_MSG_ENV`. See Q1 in Section 6.6 on how to use the Keil IDE to define compilation macros.

6.4 Processes

6.4.1 The Six User Test Processes

Q1: Do we need to write testing cases to test the six test processes?

A1: The purpose of the user test processes is to implement test cases at user level to test the OS APIs your kernel provides. You do not really test the test processes.

Q2: Does each user test process implement one test case/scenario?

A2: No. Some test scenarios may require more than one test process. One example is to test a process to be blocked on memory when the system is out of memory. This scenario cannot be tested with just one process.

6.4.2 System Processes

Q1: Is system process a kernel process or user process?

A1: A system process can run as a user process or a kernel process depending what the system process is doing and what resources it needs.

Q2: Where should we put the null process (which file)?

A2: Where to put the process in a file is up to you. You have the sole freedom to organize the code in this project. To facilitate the automated third-party user level testing, you should not put the null process in the same file where you define the six user test processes.

Q3: Are system processes such as KCD and CRT scheduled the same as user processes?

A3: Yes.

Q4: Does the sample interrupt-driven UART code include any mechanism for calling our keyboard/CRT routines on interrupt? Or do we need to write that ourselves?

A4: The provided sample UART IRQ code contains both receive and transmit interrupts handling logic. You can borrow the relevant part of the code to write your own CRT, KCD and I-process code.

6.4.3 The I-processes

Q1: Why does an I-process need a pcb?

A1: Because the RTX uses message passing to request interrupt related services and the I-process needs at least a process ID to facilitate this.

Q2: How to invoke an I-process?

A2: The I-process is triggered (or scheduled) by interrupts. When an interrupt fires, it goes to the specific IRQ service routine and this service routine will need to save the context of the current running process (say P) and then makes the code that implement the I-process (most likely a function) to run. Once the I-process finishes, the scheduler may pick another process to run (say Q, P and Q may not necessarily be the same) and the context of Q needs to be restored properly.

There are still context switching involved, but due to the special two state property of an I-process, some students may choose not to do a full fledged context switching as a normal process would do to run the I-process. The bottom line is that an I-process does not get interrupted and the process before the I-process and after the I-process need to have context saved and restored properly.

Q3: Do we need to save the context of the I-process and restore it later?

A3: An i-process only has two states: `RUN` or `WAITING_FOR_INTERRUPT`. It does not get interrupted, so there is no need to save or restore i-process's context. But there is a need to save the context of the process that gets interrupted by the I-process and restore the context of the process that the system decides to run after the I-process finishes.

Q4: To get the interrupt processes to act immediately, I am considering giving them a process priority of -1 so that the scheduler gives interrupt processes the highest priority. This would allow a smooth integration of these processes into the current OS. One of the functions that we must have is `get_process_priority(int processID)` **returns -1 on error, and just above that, it says process' can only have priority of 0 - 4. Is this function only expected to work for user processes, thus our solution is viable? Or are we not allowed to use the priority of -1 as specified by the 0-4 range for some reason?**

A4: The project specification requires when a user process calls `get_process_priority`, the function returns -1 on error. If you want to have "-1" as a valid process priority for special processes such as i-process, then you have to hide this fact from regular user process. The i-process is *scheduled* by the interrupt, so technically it by-passes the regular process scheduler. However some students in the past did implement the i-process invocation by giving it the highest priority in the system. As long as your implementation follows the specification, we accepted it.

Q5: When an I-process use message passing to send data to the KCD and CRT process, that means they have to request a memory block. What happens if there is no memory in the system? An I-process cannot be memory blocked so should we just disregard that particular interrupt?

A5: In the project description, it mentioned to adapt the behaviour of blocking primitives when they are called by an I-process. You should never let an I-process be blocked.

6.5 Interrupts

Q1: Do we need to use atomic(on) and atomic(off) for the RTX P1 deliverable. If yes can you explain how atomic works?

A1: In sense of this project, the atomic(on) turns off all interrupts and atomic(off) turns on all interrupts. For P1, we assume there is no interrupts, so no need for atomicity for P1. But they are needed in P2 and P3.

Q2: Why do we need to call the scheduler at the end of an interrupt handling process? Wouldn't simply going back to the previously running process work?

A2: The reason is that an interrupt may cause a preemption. For example, if you have a higher priority process blocked on receiving a message that is sent by `delayed_send()` primitive. A timer interrupt may cause the message to be delivered to this higher priority process. Hence when the interrupt finishes, you should not return to the previous process which has lower priority.

Q3: What should the correct behaviour be when a new interrupt happens while handling a previous interrupt?

A3: In this project, nested interrupt is not required. You could just disable all interrupts when you are inside the interrupt handler.

Q4: How to enable and disable all interrupts?

A4: This can be done either in the assembly code or the C code. Please see table 9.3.

Q5: Is the TX interrupt activated by setting the TX FIFO rest bit specified in Table 278 on p305 of LPC17xx user's manual[4]?

A5: It is the THRE Interrupt Enable bit in Table 275 on page 302.

6.6 Keil IDE

Q1: Is there a way we can have two configurations, one for debug and one for release? This way we can have a DEBUG macro be defined only in the debug configuration.I noticed you use `DEBUG_0`. Is this some special macro?

A1: The conditional compilation symbol `DEBUG_0` is defined in target options under C/C++ tab (See Figure 6.1). This is a symbol we define for debugging purpose which makes the code to print out more debugging messages. You could define your own symbol to control compilation in a similar way. If you have multiple preprocessor symbols, use space(s) to separate them.

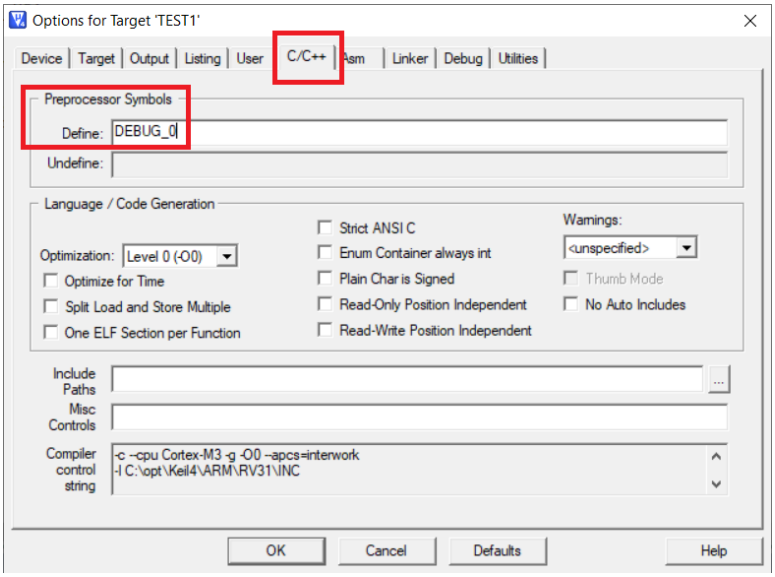

Figure 6.1: Keil IDE: preprocessor definition

Q2: Is it possible to trigger UART interrupts via the keyboard while debugging with the simulator?

A2: Yes, just type in the `UART#1` window in simulator (assuming you are programming UART0). If you are looking at the sample UART IRQ code on GitHub, you need to work on the SIM target in order to see IRQs under simulator. Not the RAM target.

Part IV

Computing Environment and Keil MCB1700 Development 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 SE 350 lab starter github is at https://github.com/yqh/SE350. Let's first make a clone of this repository by using the following command:

```
git clone https://github.com/yqh/SE350.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).

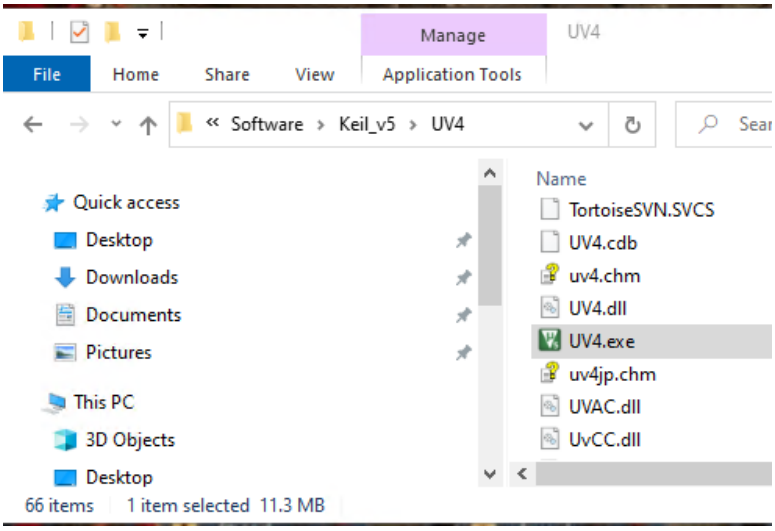

Figure 8.1: Keil IDE: Create a New Project

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).

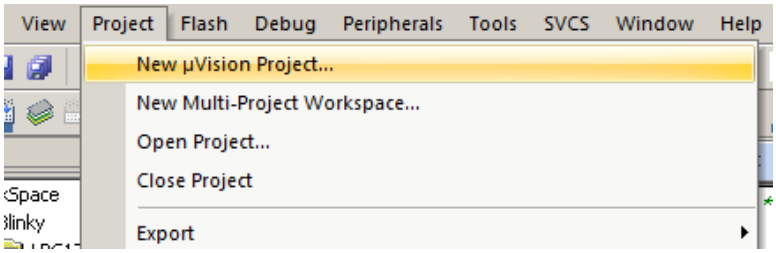

Figure 8.2: Keil IDE: Create a New Project

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

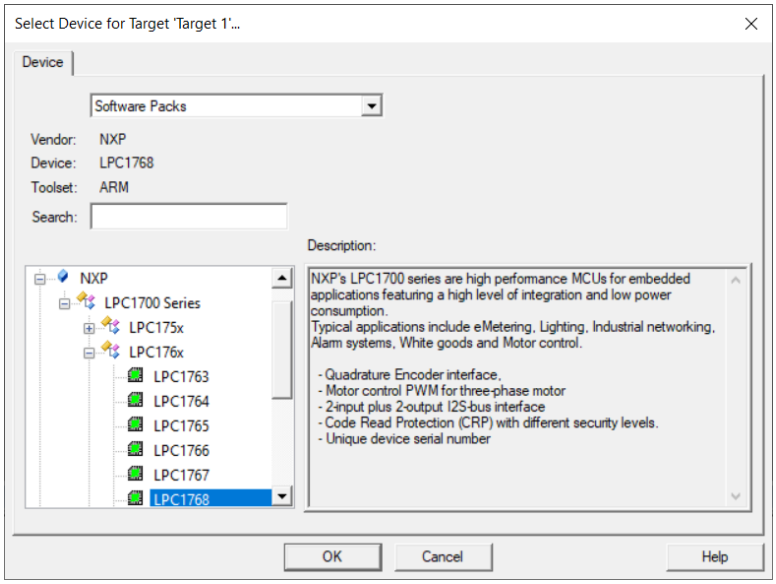

Figure 8.3: Keil IDE: Choose MCU

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

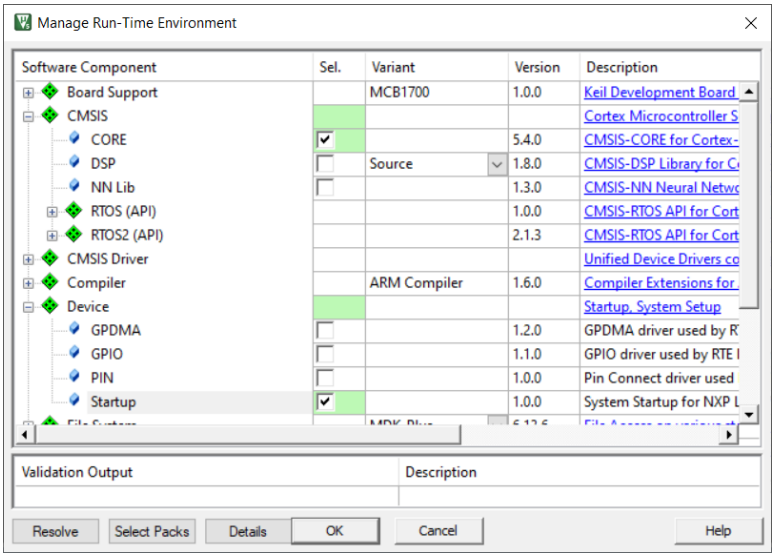

Figure 8.4: Keil IDE: Manage Run-time Environment

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.

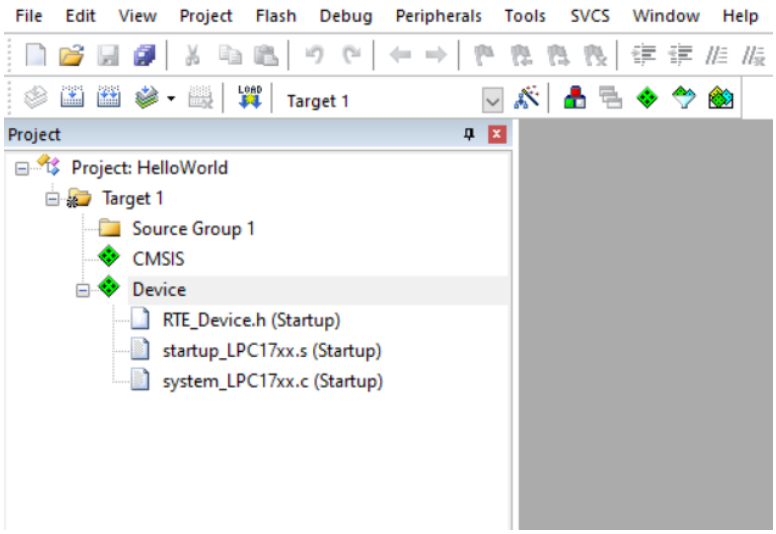

Figure 8.5: Keil IDE: A default new project

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).

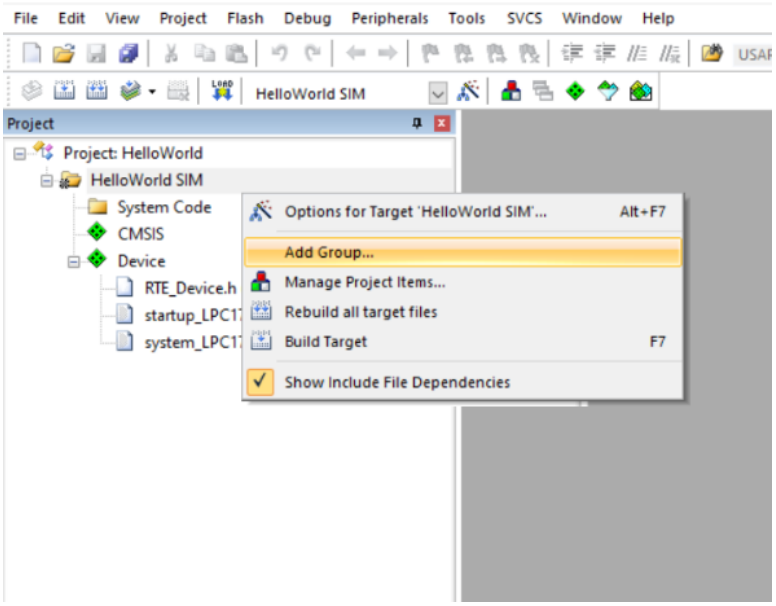

Figure 8.6: Keil IDE: Add Group

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.

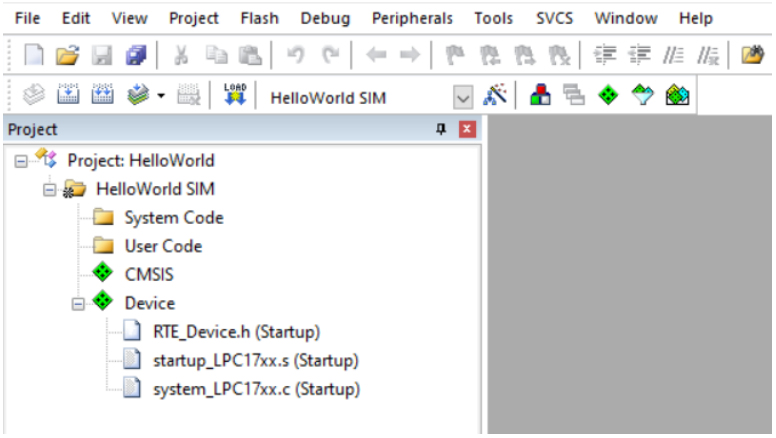

Figure 8.7: Keil IDE: Updated Project Profile

[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).

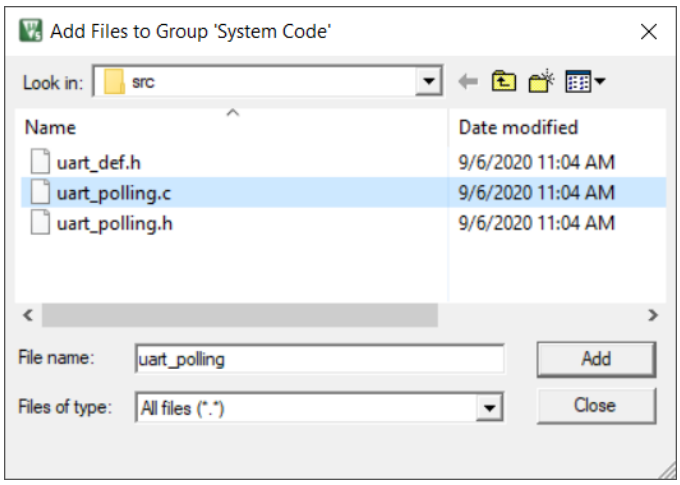

Figure 8.8: Keil IDE: Add Source File to Source Group

Your project will now look like Figure 8.9.

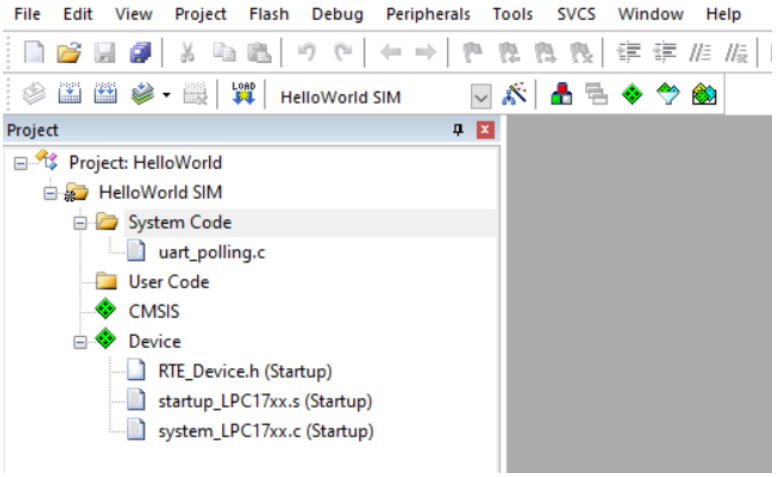

Figure 8.9: Keil IDE: Updated Project Profile

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).

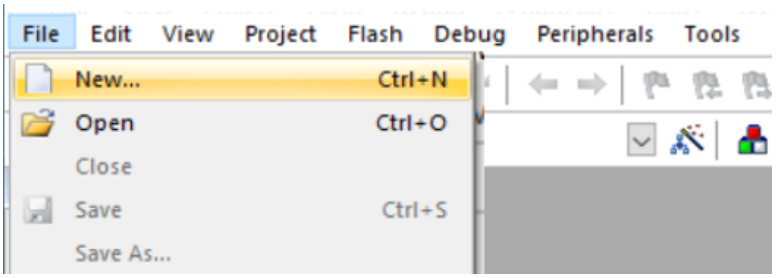

Figure 8.10: Keil IDE: Create New File

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.

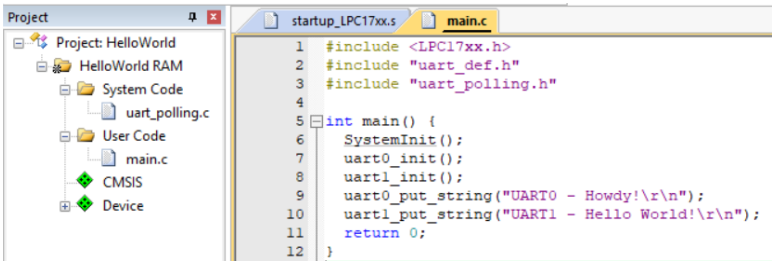

Figure 8.11: Keil IDE: Final Project Setting

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).

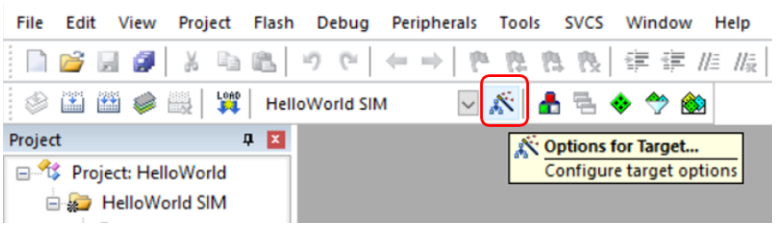

Figure 8.12: Keil IDE: Target Options Configuration

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.

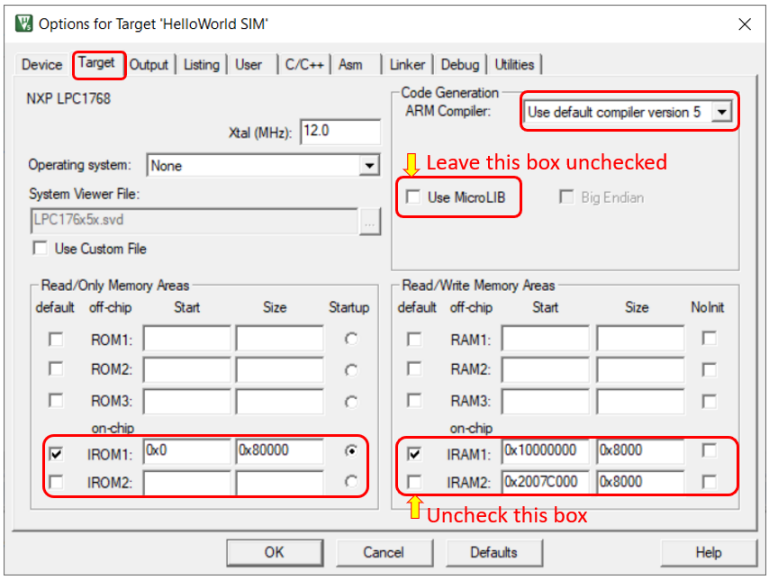

Figure 8.13: Keil IDE: Target Options Target Tab Configuration

3. Configure the C/C++ tab as shown in Figure 8.14. We do not need c99 for this example. So we leave it unchecked.

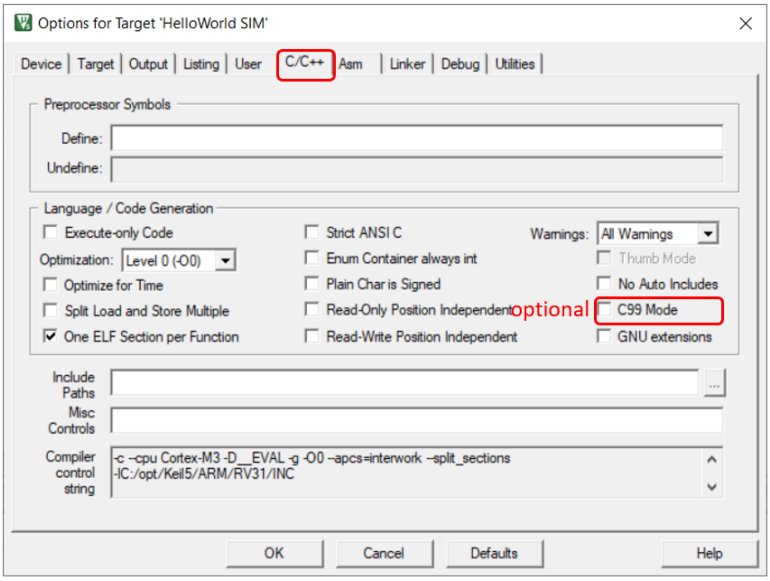

Figure 8.14: Keil IDE: Target Options C/C++ Tab Configuration

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.

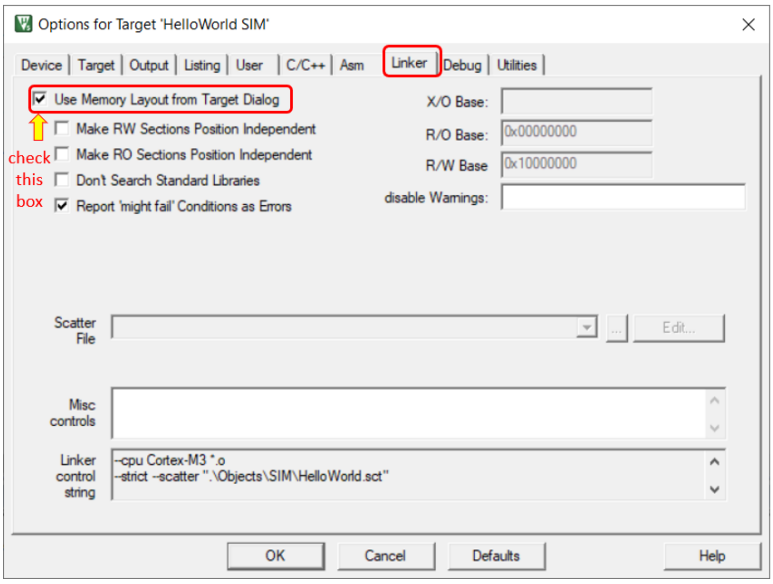

Figure 8.15: Keil IDE: Target Options Linker Tab Configuration

8.6.2 Build the Target

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

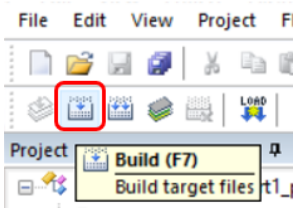

Figure 8.16: Keil IDE: Build Target

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.

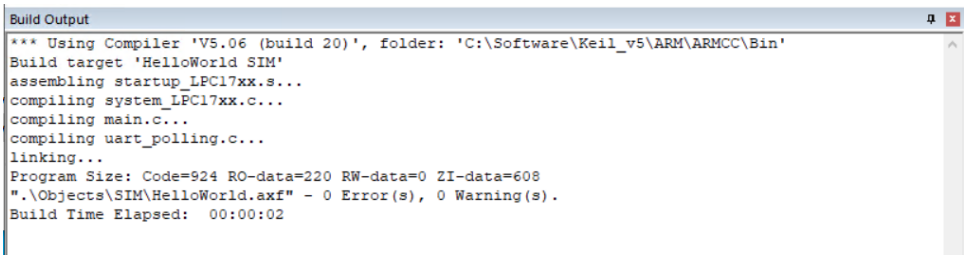

Figure 8.17: Keil IDE: Build Target

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.2 in Appendix B) is needed to map the second bank of RAM area read and write accessible inside the simulator.

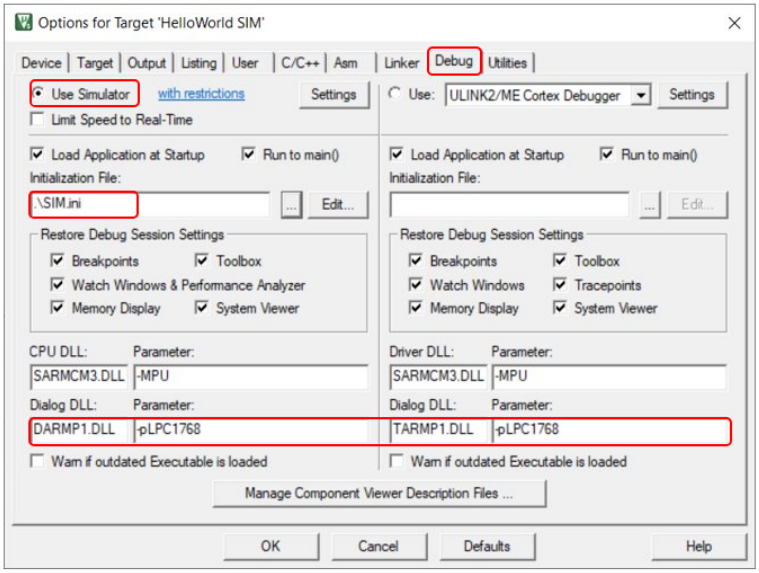

Figure 8.18: Keil IDE: Target Options Debug Tab Configuration

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

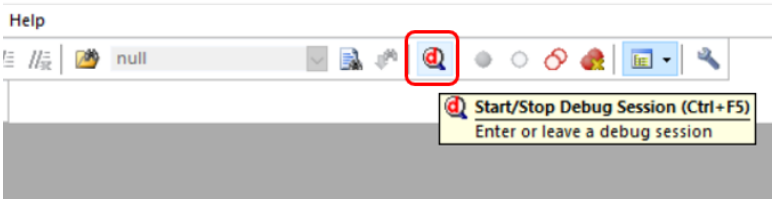

Figure 8.19: Keil IDE: Debug Button

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.

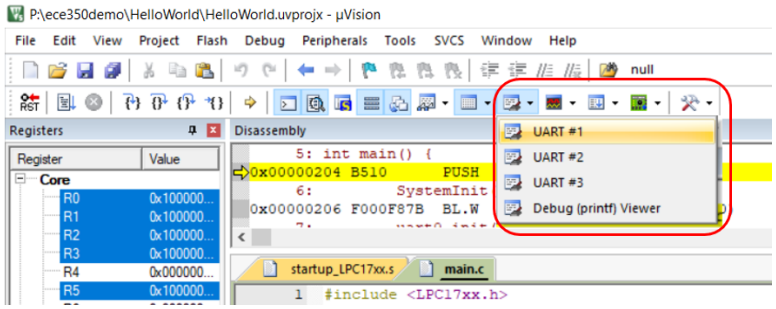

Figure 8.20: Keil IDE: Debugging. Enable Serial Window View.

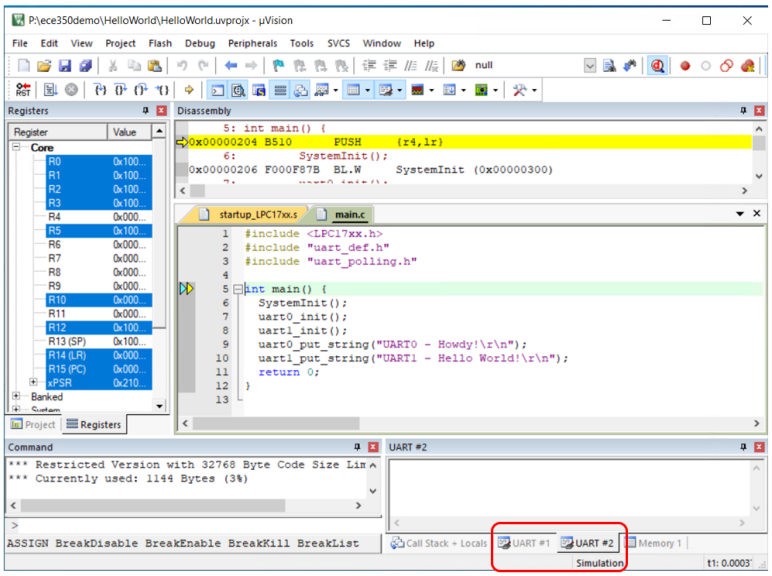

Figure 8.21: Keil IDE: Debugging. Both UART0 and UART1 views are enabled in simulator.

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.

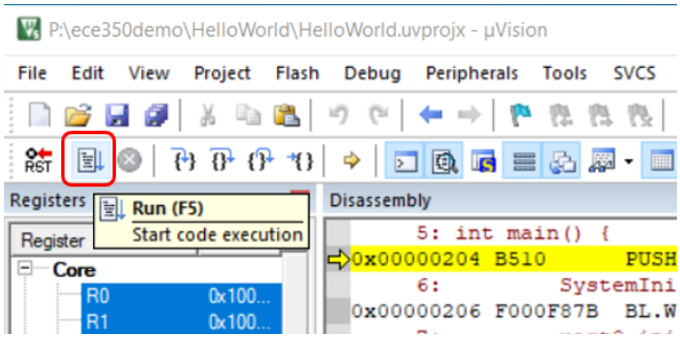

Figure 8.22: Keil IDE: Debugging. The Run Button.

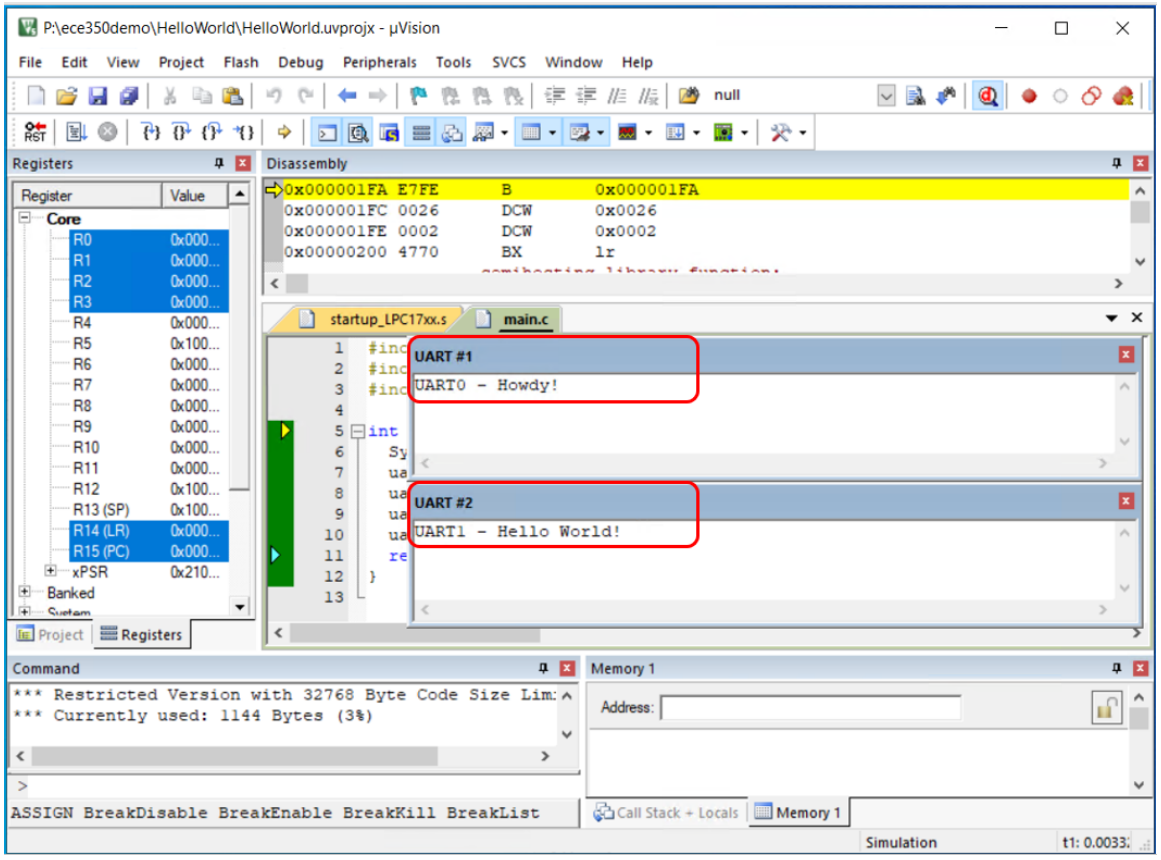

Figure 8.23: Keil IDE: Debugging Output.

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).

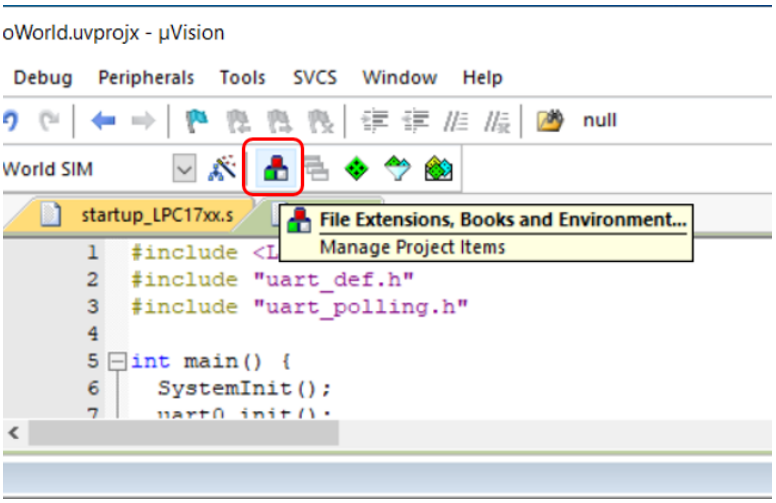

Figure 8.24: Keil IDE: Manage Project Items Button

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.

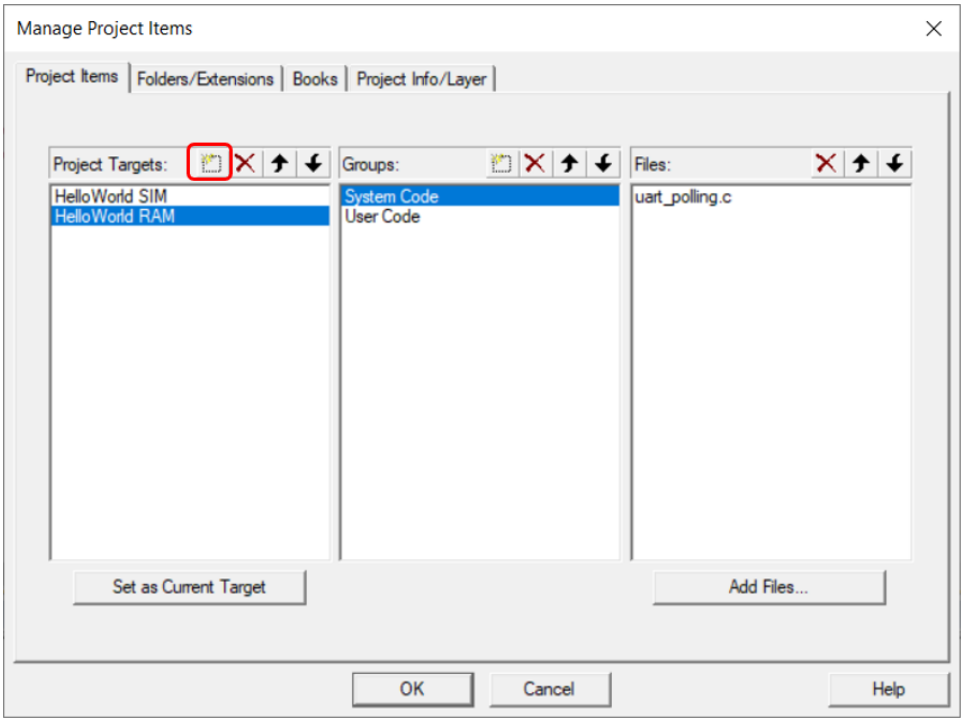

Figure 8.25: Keil IDE: Manage Project Items Window.

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

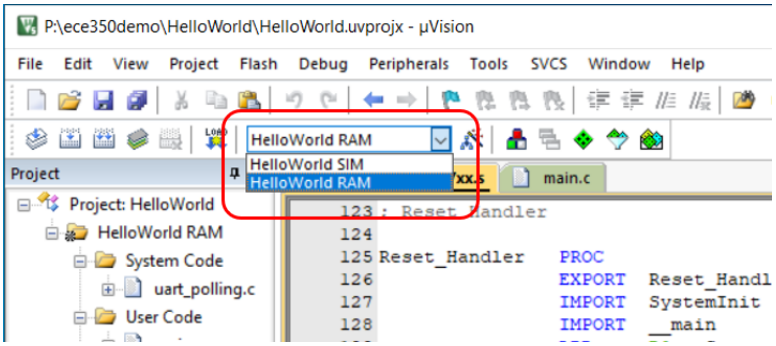

Figure 8.26: Keil IDE: Select HelloWorld RAM Target.

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

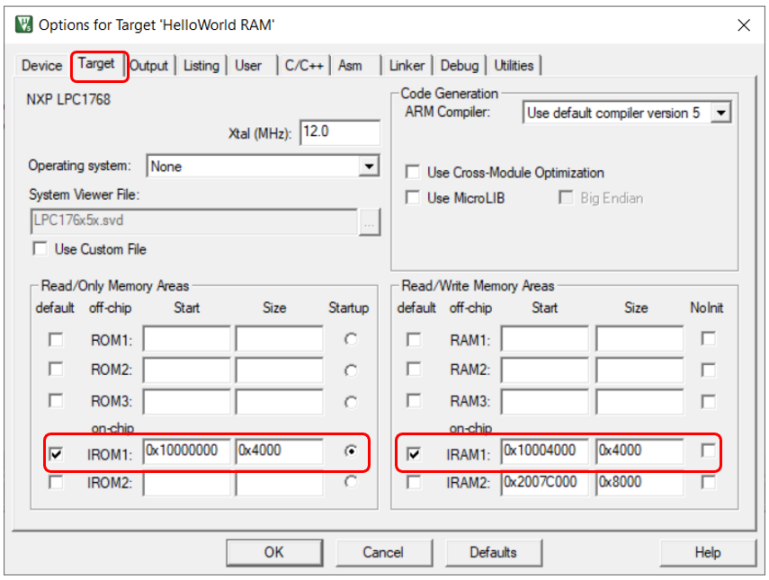

Figure 8.27: Keil IDE: Configure Target Options Target Tab for In-memory Execution.

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.

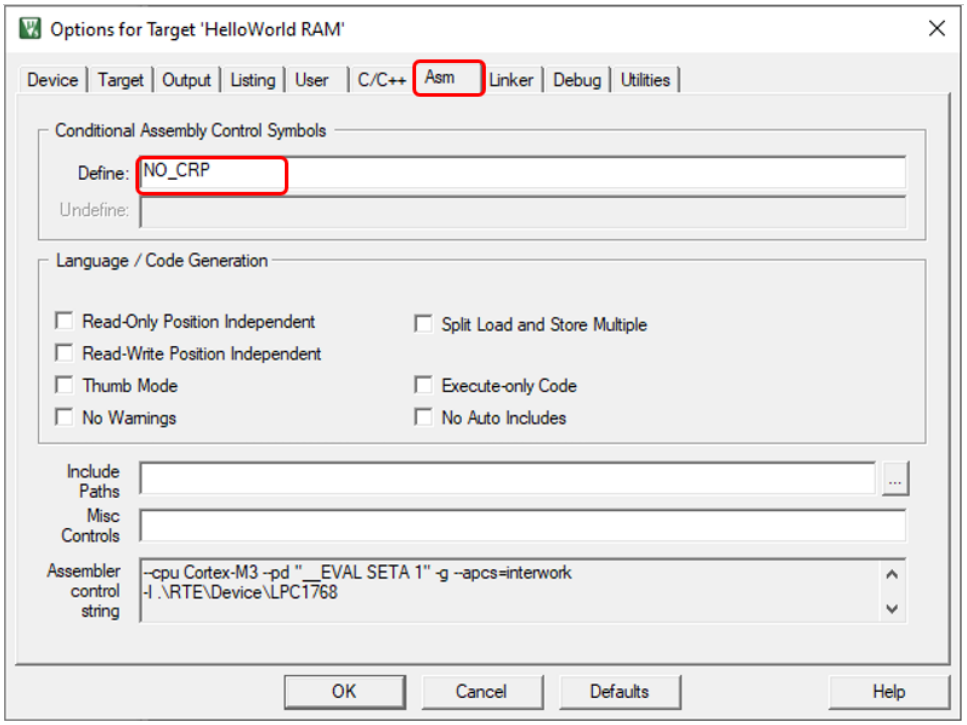

Figure 8.28: Keil IDE: RAM Target Asm Configuration.

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.1 in Appendix B) is needed to do the proper setting of SP, PC and vector table offset register.

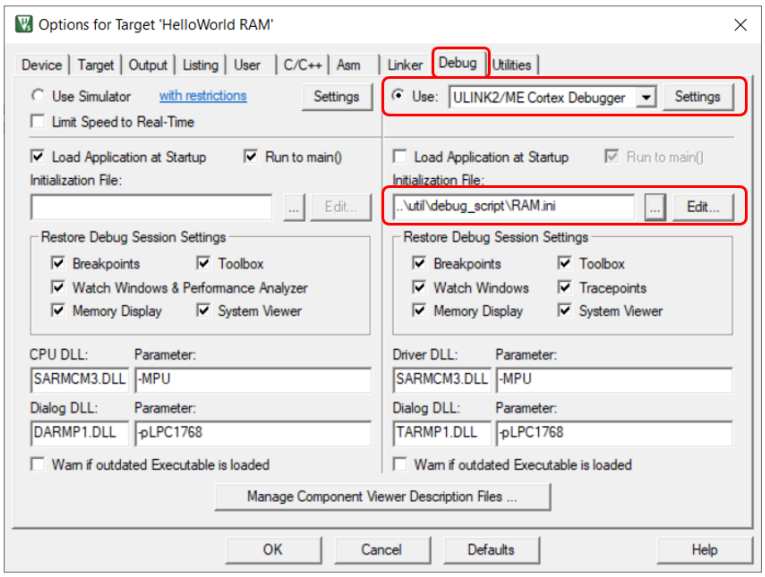

Figure 8.29: Keil IDE: Configure ULINK-ME Hardware Debugger.

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.

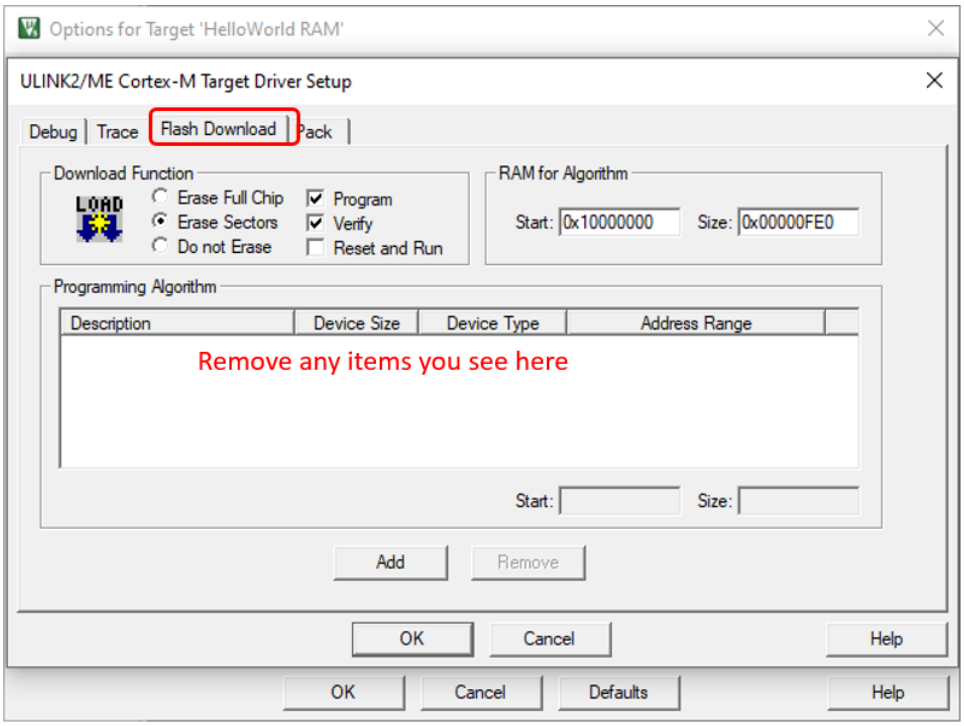

Figure 8.30: Keil IDE: Flash Download Programming Algorithm Configuration.

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

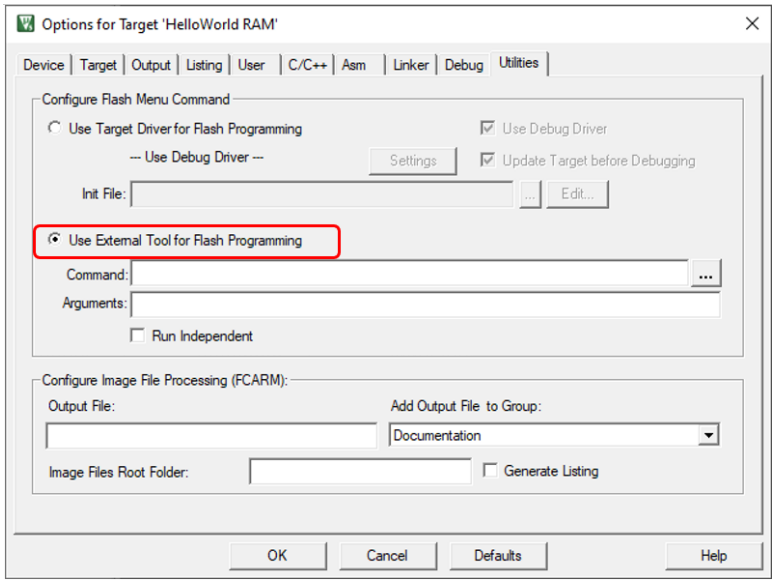

Figure 8.31: Keil IDE: Target Option Utilities Configuration for RAM Target.

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).

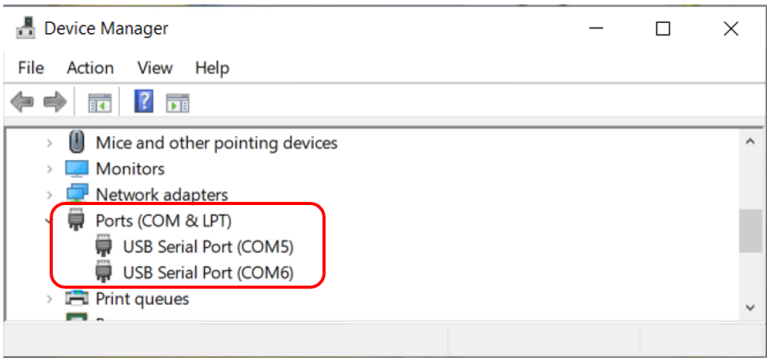

Figure 8.32: Device Manger COM Ports

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.

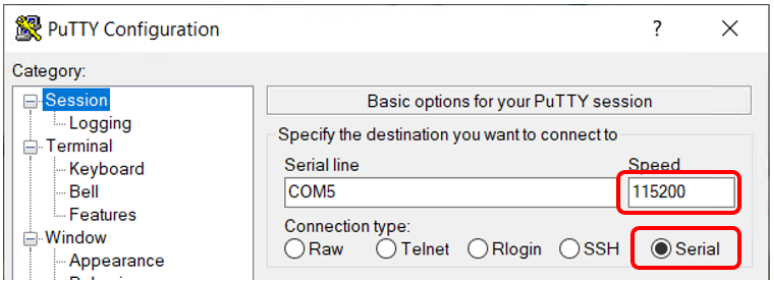

Figure 8.33: PuTTY Session for Serial Port Communication

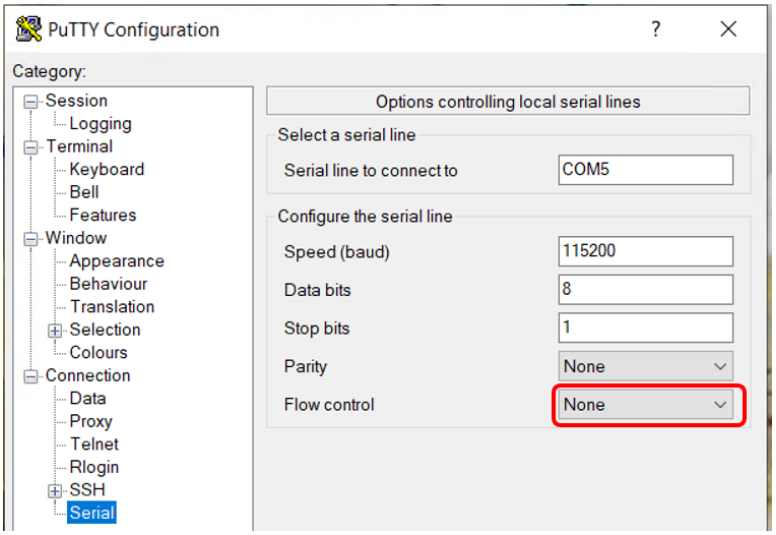

Figure 8.34: PuTTY Serial Port Configuration

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.

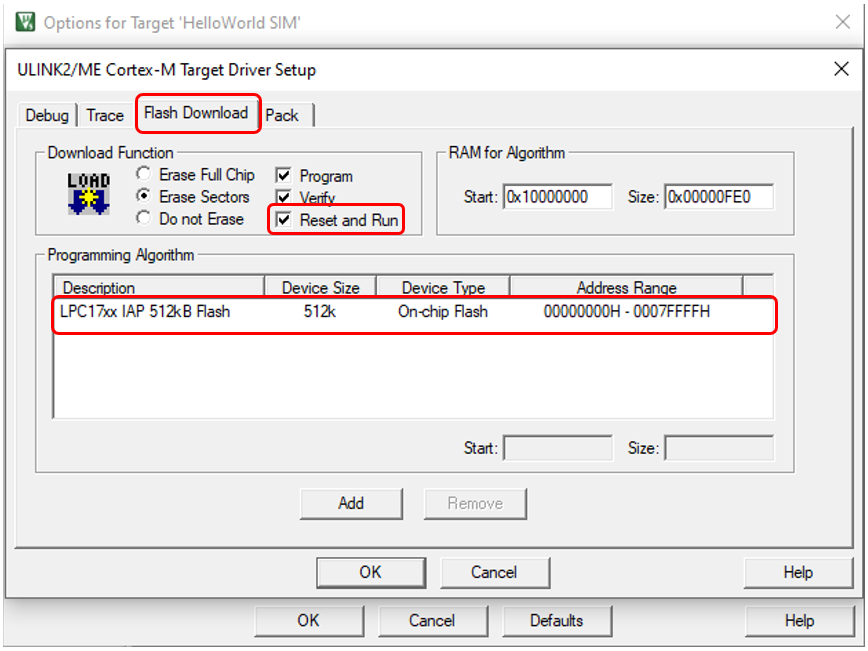

Figure 8.36: Flash Download Reset and Run Setting

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.

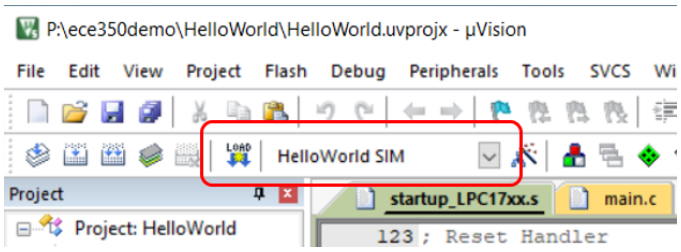

Figure 8.37: Keil IDE: Download Target to Flash

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	Platform register.
		SB	The meaning of this register is defined by platform standard.
		TR	
r8	v5		Variable-register 5.
r7	v4		Variable-register 4.
r6	v3		Variable-register 3.
r5	v2		Variable-register 2.
r4	v1		Variable-register 1.
r3	a4		argument / scratch register 4
r2	a3		argument / scratch register 3
r1	a2		argument / result / scratch register 2
r0	a1		argument / result / scratch register 1

Table 9.2: Core Registers and AAPCS Usage

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].

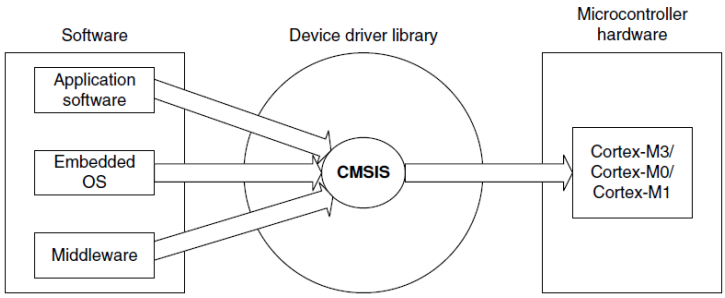

Figure 9.1: Role of CMSIS[5]

The CMSIS uses standardized methods to organize header files that makes it easy to learn new Cortex-M microcontroller products and improve software portability. With the `<device>.h` (e.g. `LPC17xx.h`) and system startup code files (e.g., `startup_LPC17xx.s`), your program has a common way to access

- **Cortex-M processor core registers** with standardized definitions for NVIC, SysTick, MPU registers, System Control Block registers , and their core access functions (see `core_cm*.[ch]` files).
- **system exceptions** with standardized exception number and handler names to allow RTOS and middleware components to utilize system exceptions without having compatibility issues.
- **intrinsic functions with standardized name** to produce instructions that cannot be generated by IEC/ISO C.
- **system initialization** by common methods for each MCU. Fore example, the standardized `SystemInit()` function to configure clock.
- **system clock frequency** with standardized variable named as `SystemFrequency` defined in the device driver.
- **vendor peripherals** with standardized C structure.

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).

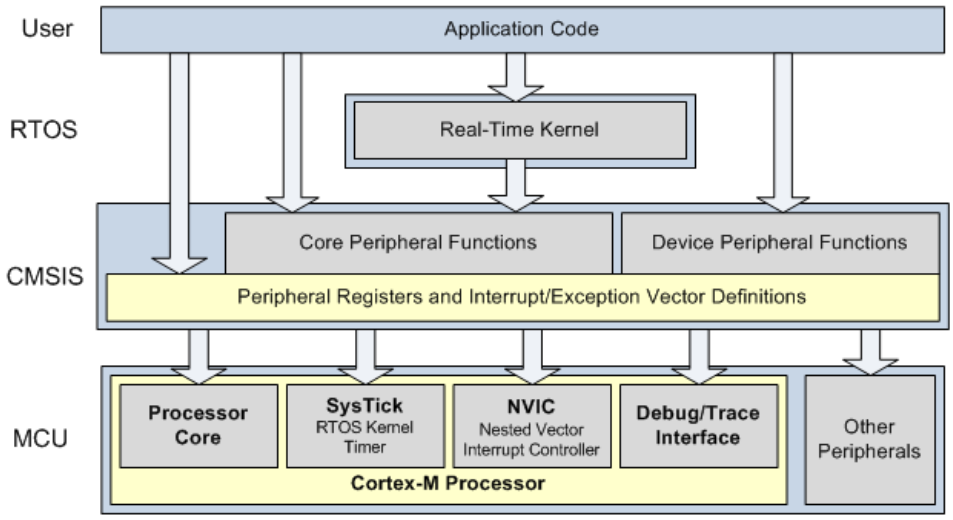

Figure 9.2: CMSIS Organization[2]

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`).

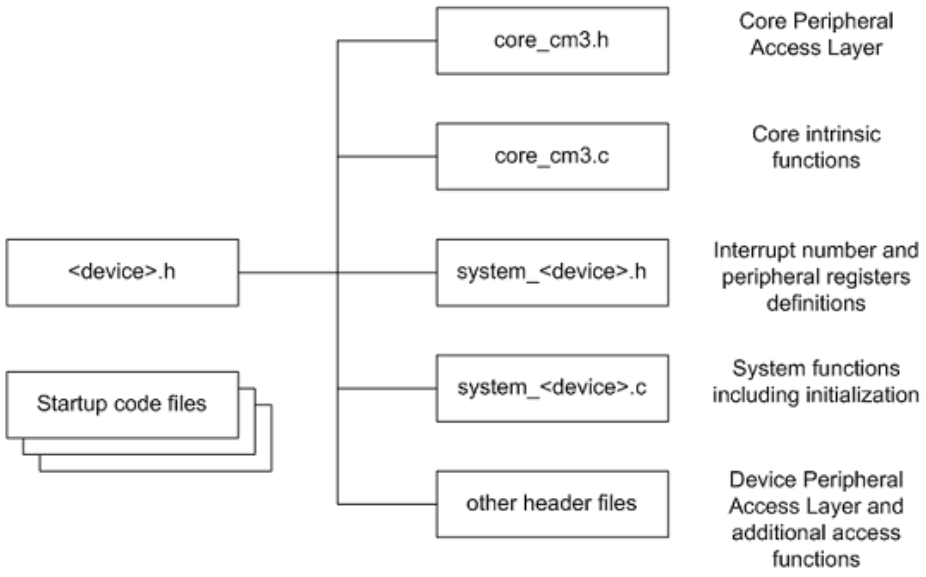

Figure 9.3: CMSIS Organization[2]

	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
}
```

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`

Another way is to use the embedded assembly code:

```
__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.

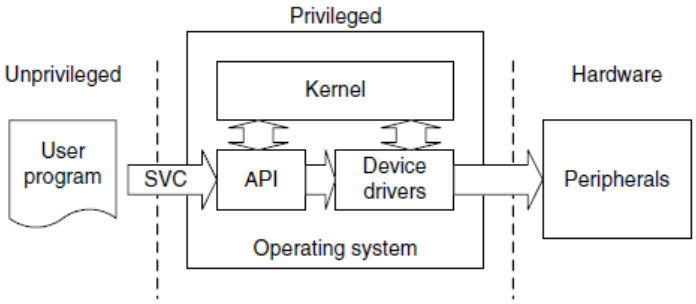

Figure 9.5: SVC as a Gateway for OS Functions [5]

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

The direct method is to program at assembly instruction level. We can use the embedded assembly mechanism and write `SVC` assembly instruction inside the embedded assembly function. An example implementation of is shown in Listing 9.7 shows an example implementation of `int release_memory_block(void *memory_block)`.

```
__asm int release_memory_block(void *memory_block) {
   LDR R12,=__cpp(k_release_memory_block)
   ; privileged mode code handling omitted
   SVC 0
   BX  LR
   ALIGN
}
```

Listing 9.7: Code Snippet of release_memory_block

The corresponding kernel function is the C function `k_release_memory_block`. This function entry point is loaded to `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 `SVC_handler` in `HAL.c` from the starter code (https://github.com/yqh/SE350/blob/master/manual_code/SVC/src/HAL.c).

```
 __asm void SVC_Handler(void) {

   ; extract SVC number, if it 0, then extract R12 from the exception
      stack frame
   ; save cpu registers

   BLX R12 ; R12 contains the kernel function entry point

   ; store C kernel function return value in R0 to R0 on the exception
      stack frame omitted
   ; restore cpu registers
}
```

Listing 9.8: Code Snippet of SVC_Handler

The indirect method is to ask the compiler to generate the SVC instruction from C code. The ARM compiler provides an intrinsic keyword named __svc_indirect which passes an operation code to the SVC handler in r12[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 release_memory_block is a user function with the following signature:

```
#include <rtx.h>
int release_memory_block(void *memory_block);
```

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

```
#define __SVC_0 __svc_indirect(0)
extern int k_release_memory_block(void *);
#define release_memory_block(p_mem_blk) _release_memory_block((U32)
   k_release_memory_block, p_mem_blk)
extern int _release_memory_block(U32 p_func, void *p_mem_blk) __SVC_0;
```

The compiler generates two assembly instructions

```
   LDR.W r12, [pc, #offset]; Load k_release_memory_block in 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 [5]. The processor has private peripherals which are system control block, system timer, NVIC (Nested Vectored Interrupt Controller) and MPU (Memory Protection Unit). The processor includes a number of internal

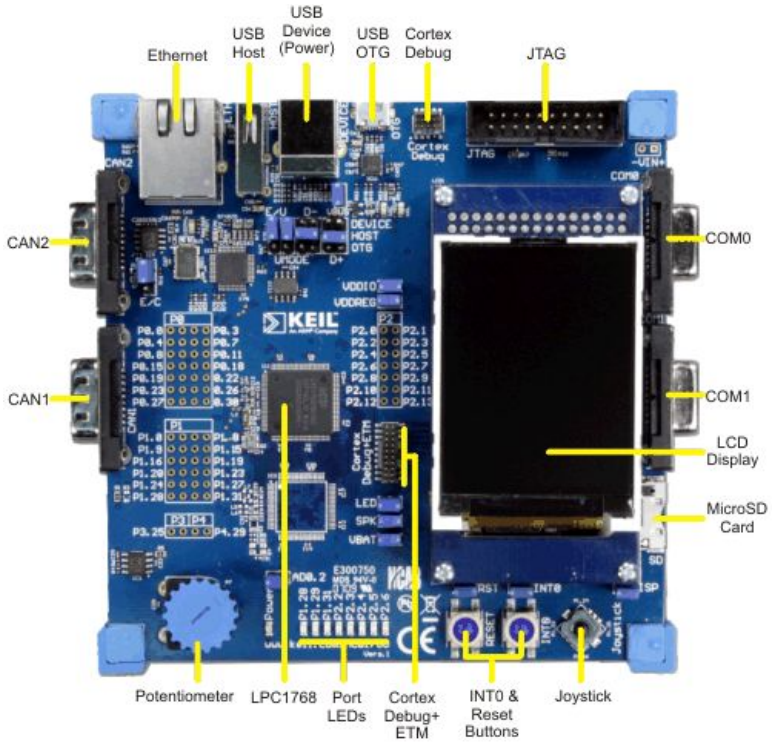

Figure 10.1: MCB1700 Board Components [1]

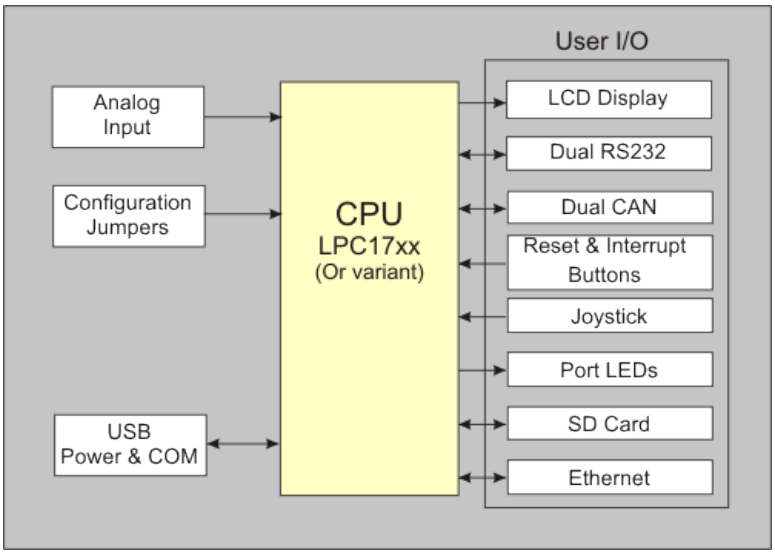

Figure 10.2: MCB1700 Board Block Diagram [1]

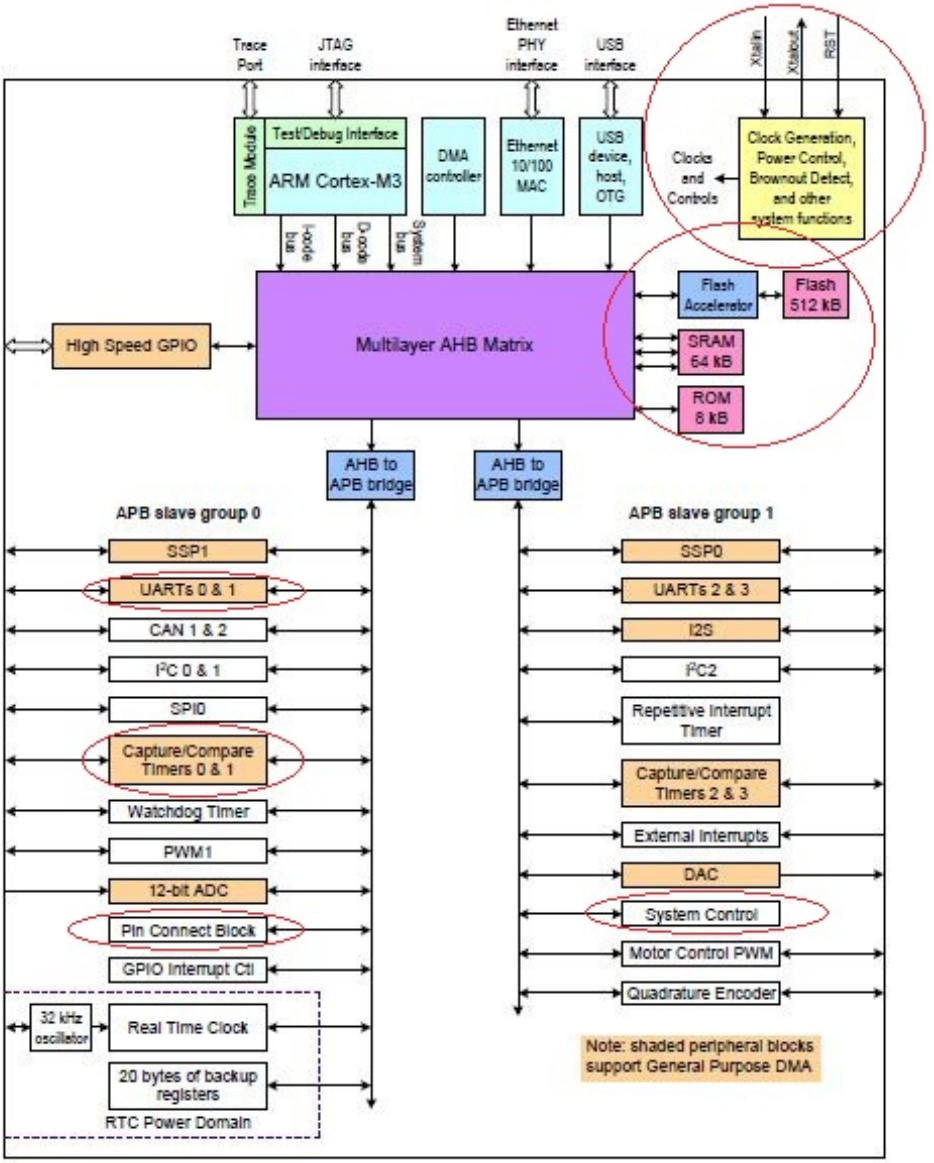

Figure 10.3: LPC1768 Block Diagram. The circled blocks are the ones that we will use in the lab project.

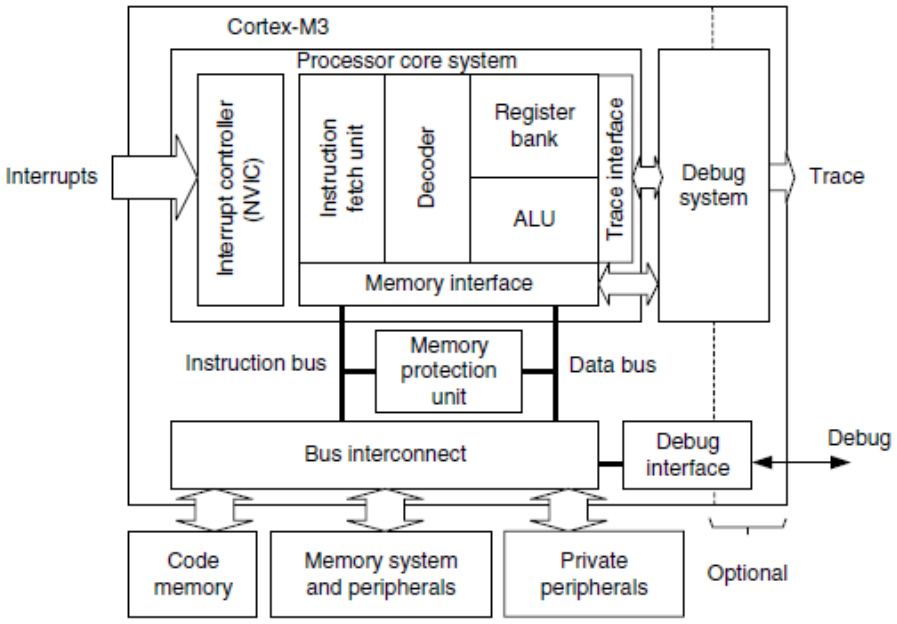

Figure 10.4: Simplified Cortex-M3 Block Diagram[5]

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.

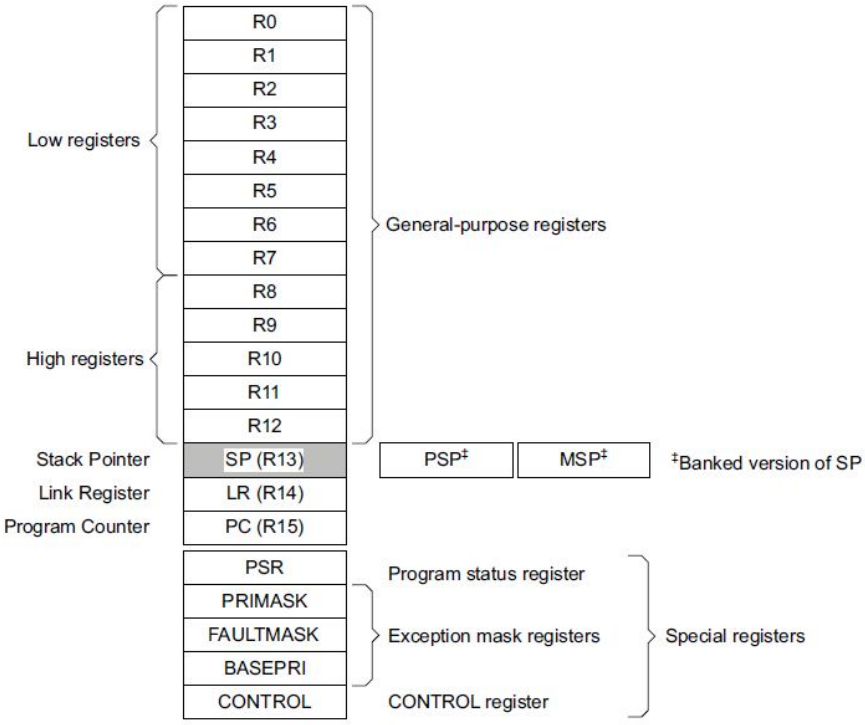

Figure 10.5: Cortex-M3 Registers[4]

- 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.

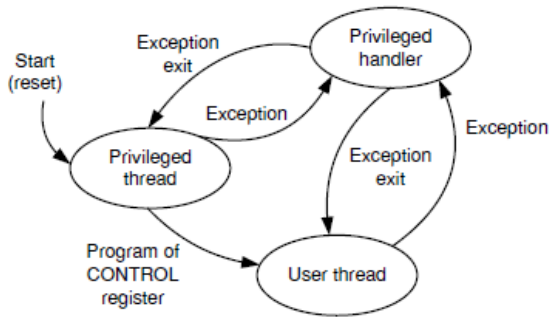

Figure 10.6: Cortex-M3 Operating Mode and Privilege Level[5]

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

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:

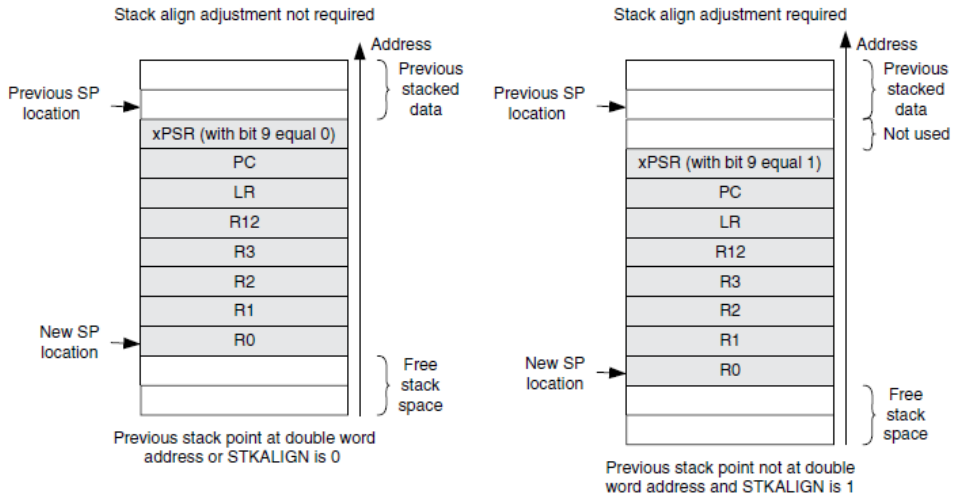

Figure 10.7: Cortex-M3 Exception Stack Frame [5]

- 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 [5]

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

Table 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.

SE350 Request to Leave a Project Group Form

Name:	
Quest ID:	
Student ID:	
Lab Assignment ID	
Group ID:	
Names of Other Group Members:	

Provide the reason for leaving the project group here:

Signature _____ Date _____

Appendix B

The Debugger Initialization Files

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

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

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

Listing B.1: The RAM.ini file

The `SIM.ini` file in the starter code can be found in Listing B.2. 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. We use `SLOG` command to log the UART output to a file.

```
MAP 0x2007C000, 0x20083FFF READ WRITE
SLOG > uart1.log
```

Listing B.2: The SIM.ini file

Bibliography

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

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

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

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

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