SE350 Keil MCB1700 Lab Manual

by

Yiqing Huang
Thomas Reidemeister

Electrical and Computer Engineering Department
University of Waterloo

Waterloo, Ontario, Canada, 2012

Acknowledgments

We would like to thank Professor Sebastian Fischmeister who made the Keil Boards and MDK-ARM donations possible. We would also like to thank Professor Jim Barby who provided timely departmental resource towards the development of the course project, without which this project will not be possible in winter 2012. Mr. Roger Sanderson oversees the ECE lab and provides us with all necessary experiment tools and resources, which we are grateful. We appreciate that Mr. Bernie Roehl has shared his valuable Keil board experiences with us. Our gratitude also goes out to Mr. Eric Praetzel who sets up the RTOS lab and also maintains the Keil software on Nexus machines; Miss Laura Winger who managed to customize the boards so that we have the neat plastic cover to protect our hardware. Mr. Bob Boy from ARM always answers our questions in a detailed and timely manner. Thank everyone who has helped.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This document is intended to be a reference guide for the MCB1700 boards used in the SE350 course project RTX. This document is organized as follows:

- Hardware Environment
- Software Development Tools
- Programming MCB1700

Use of the lab after hours is a privilege, not a right. Loss, damage, improper use of equipment or indication of food or beverage consumption in the lab will lead to cancellation of after-hour access.

Chapter 2

Hardware Environment

2.1 MCB1700 Board Overview

The Keil MCB1700 board is populated with *NXP LPC1768* Microcontroller. Figure 2.1 shows the important interface and hardware components of the MCB1700 board.

Figure 2.2 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 100MHz 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 2.3 is the simplified LPC1768 block diagram [2], where the components to be used in your RTX project are circled with red. Note that this manual will only discuss the components are relevant to the RTX course project. The LPC17xx User Manual is the complete reference for LPC1768 MCU.

2.2 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 2.4 is the simplified block diagram of the Cortex-M3

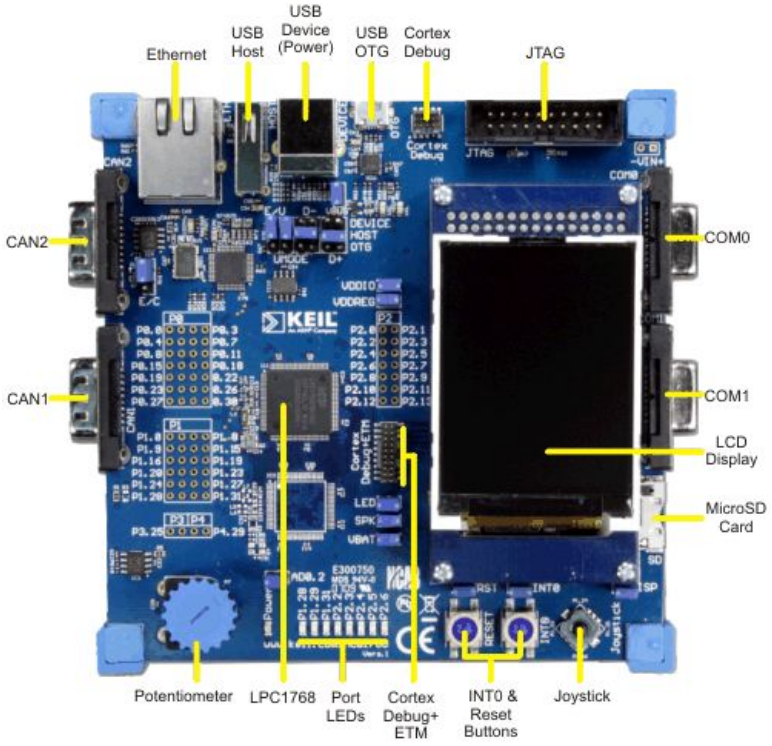


Figure 2.1: MCB1700 Board Components [1]

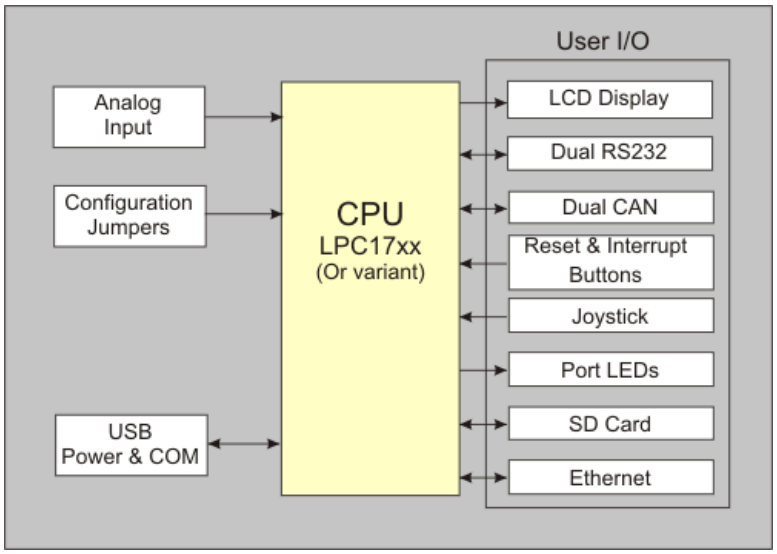


Figure 2.2: MCB1700 Board Block Diagram [1]

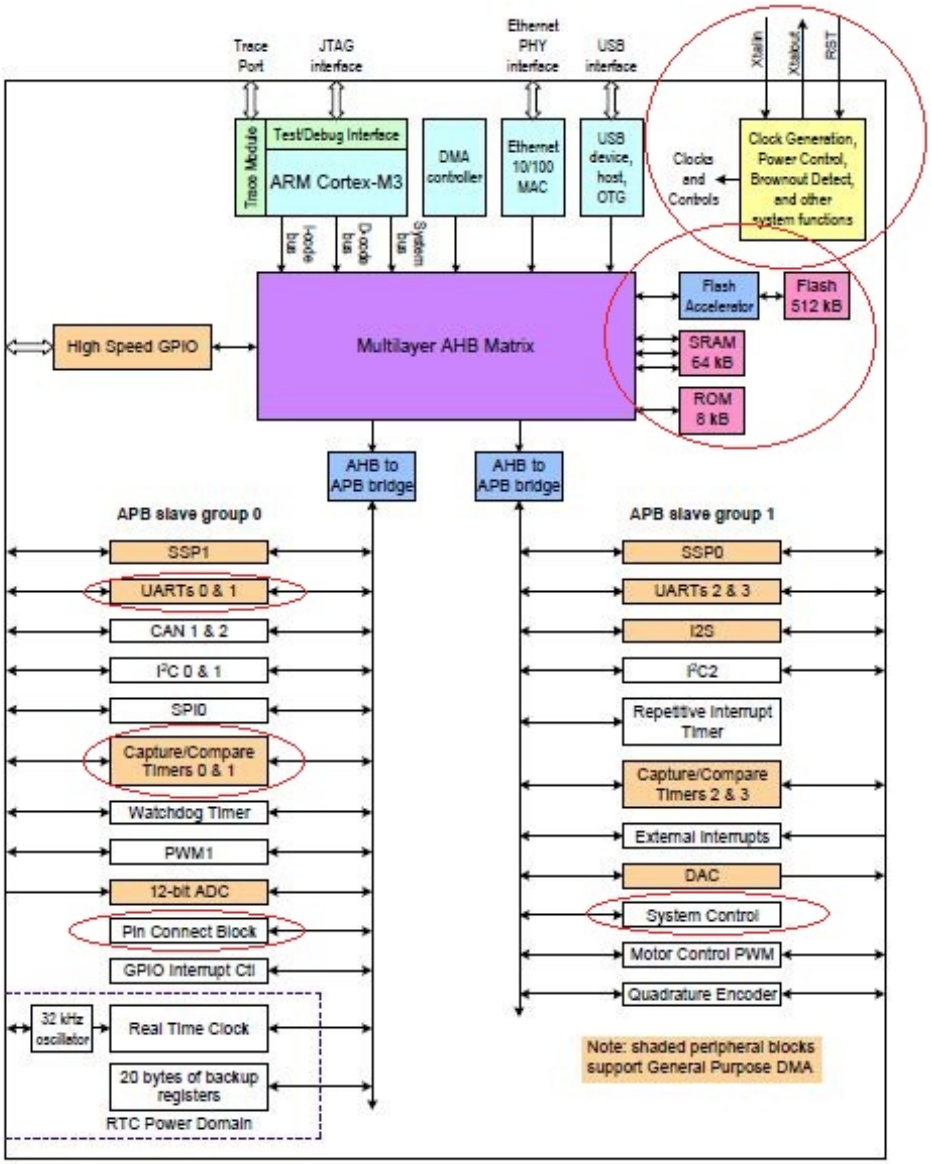


Figure 2.3: LPC1768 Block Diagram

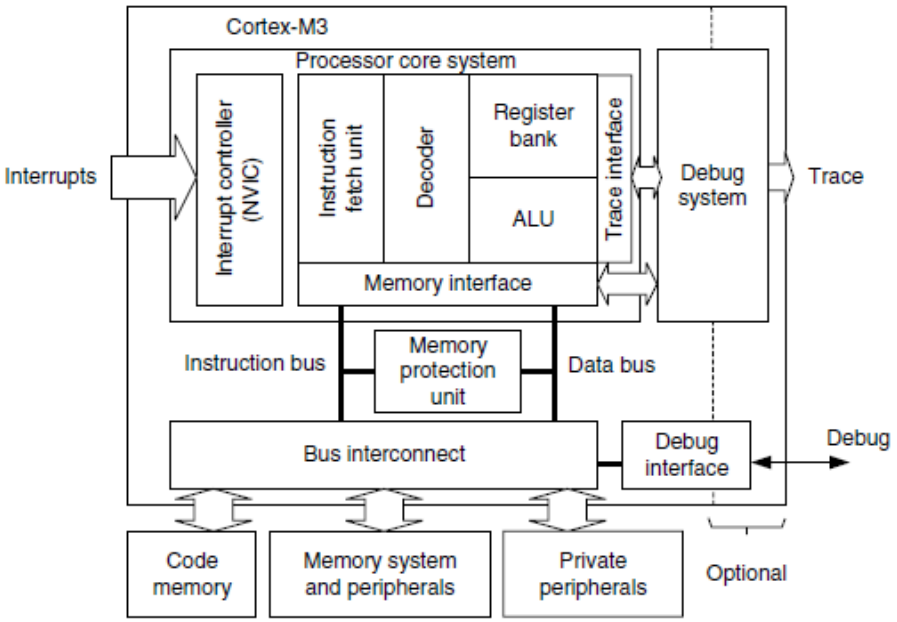


Figure 2.4: Simplified Cortex-M3 Block Diagram[3]

processor [3]. The processor has private peripherals which are system control block, system timer, NVIC (Nested Vectored Interrupt Controller) and MPU (Memory Protection Unit). The MPU programming is not required in the course project. The processor includes a number of internal debugging components which provides debugging features such as breakpoints and watchpoints.

2.2.1 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. Figure 2.5 illustrate the mode and privilege level of the processor.

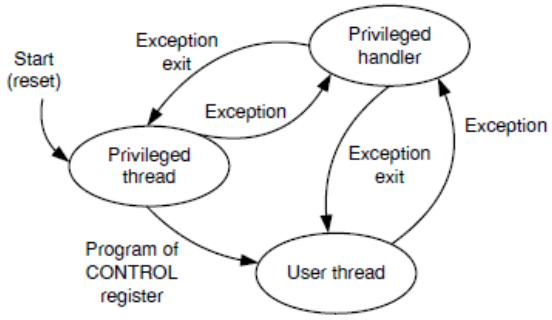


Figure 2.5: Cortex-M3 Operating Mode and Privilege Level[3]

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. You may think the `SVC` is similar as the `TRAP` for the ColdFire processor. Another way to change between Privileged Thread mode and Unprivileged thread mode is to modify the `EXC_RETURN` value in the LR (R14) when returning from an exception. You probably want to use this mechanism for context switching.

2.2.2 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 2.1 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
		Unprivileged	1	1	Process Stack
Handler	Exception handlers	Privileged	-	0	Main Stack

Table 2.1: Summary of processor mode, execution privilege level, and stack use options

2.2.3 Registers

The processor core registers are shown in Figure 2.6. For detailed description of each register, Chapter 34 in [2] is the complete reference.

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

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

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

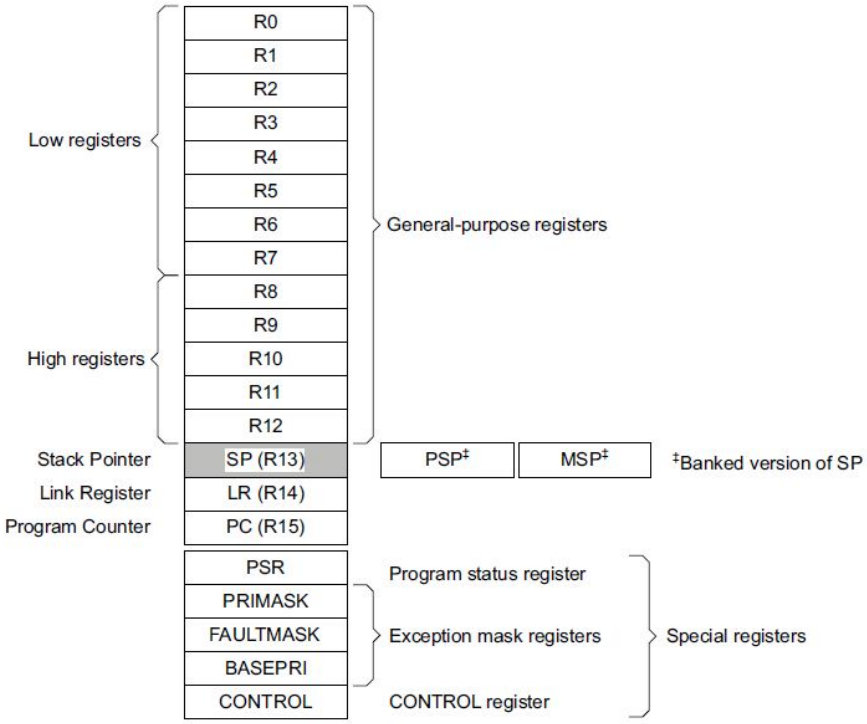


Figure 2.6: Cortex-M3 Registers[2]

- 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.

2.3 Memory Map

The Cortex-M3 processor has a single fixed 4GB address space. Table 2.2 shows the how this space is used on the LPC1768.

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.

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 2.2: LPC1768 Memory Map

2.4 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.

2.4.1 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 2.3 shows system exceptions and part of interrupt sources you may need for your RTOS project. See Table 50 and Table 639 in [2] 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.

2.4.2 Exception Entry

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

- the processor is in Thread mode

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 2.3: LPC1768 Exception and Interrupt Table

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

Figure 2.7 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.

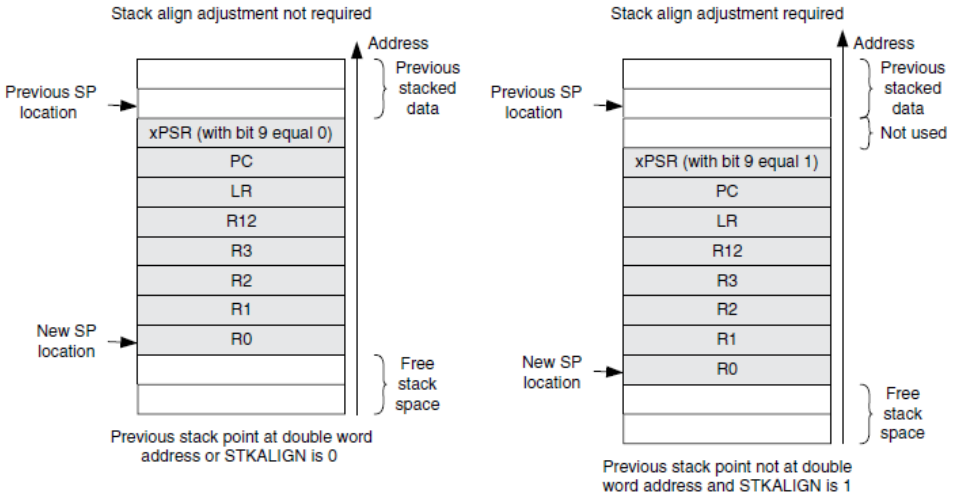


Figure 2.7: Cortex-M3 Exception Stack Frame [3]

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

- Register Updates
  After the stacking and vector fetch are completed, the exception vector will start to execute. On entry of the exception handler, the following registers will be updated as follows:

  – SP: The SP (MSP or PSP) will be updated to the new location during 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.

2.4.3 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 2.4 describes the EXC_RETURN bit fields. Table 2.5 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 2.4: EXC_RETURN bit fields [3]

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 2.5: EXC_RETURN Values on Cortex-M3

2.4.4 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.

2.5 Data Types

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

Chapter 3

Software Development Tools

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

- μ`Vision4` IDE that combines the project manager, source code editor and program debugger into one environment.
- The ARM compiler, assembler, linker and utilities.
- ULINK USB-JTAG Adapter. This allows you to debug the embedded programs running on the board.

The MDK-Lite is the evaluation version and does not require a license. However it has a code size limit of 32KB, which is adequate for your RTOS project though. The licensed MDK-Standard does not have code size limit and is installed on a couple of PCs in E2-2363.

3.1 Install MDK-ARM on your own computer

There is only a windows port for the Keil MDK-ARM for now. You can download the latest version of MDK-ARM from the following Keil website:

`http://www.keil.com/download/product/`

You need to fill out a short form. We have put MDK-ARM V4.23 direct download link inside the Learn (`http://learn.uwaterloo.ca`) to save your time of filling out the form.

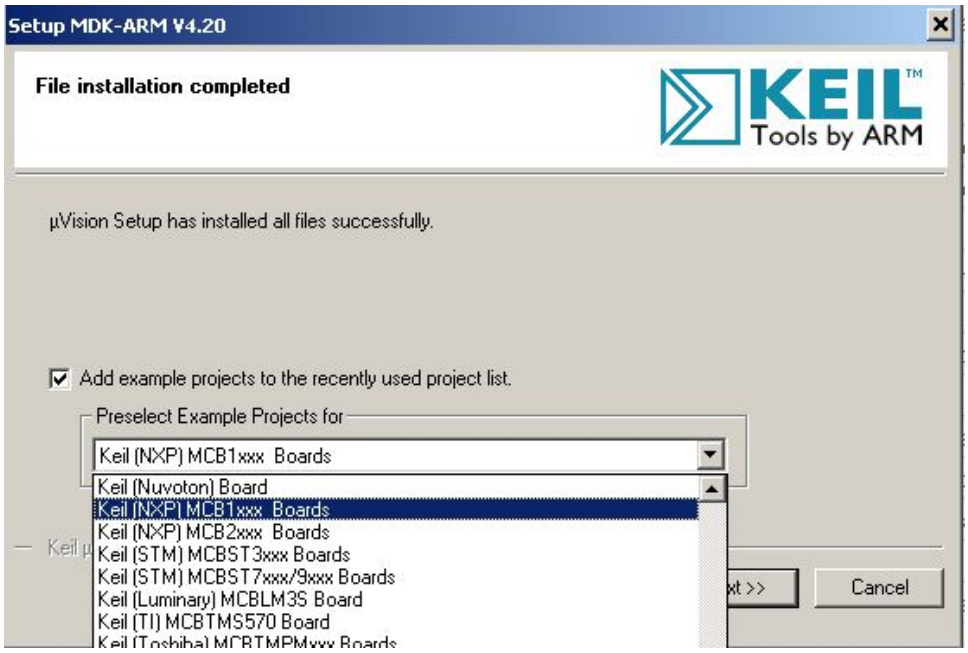


Figure 3.1: MDK-ARM Installation Steps: Choose Example Projects

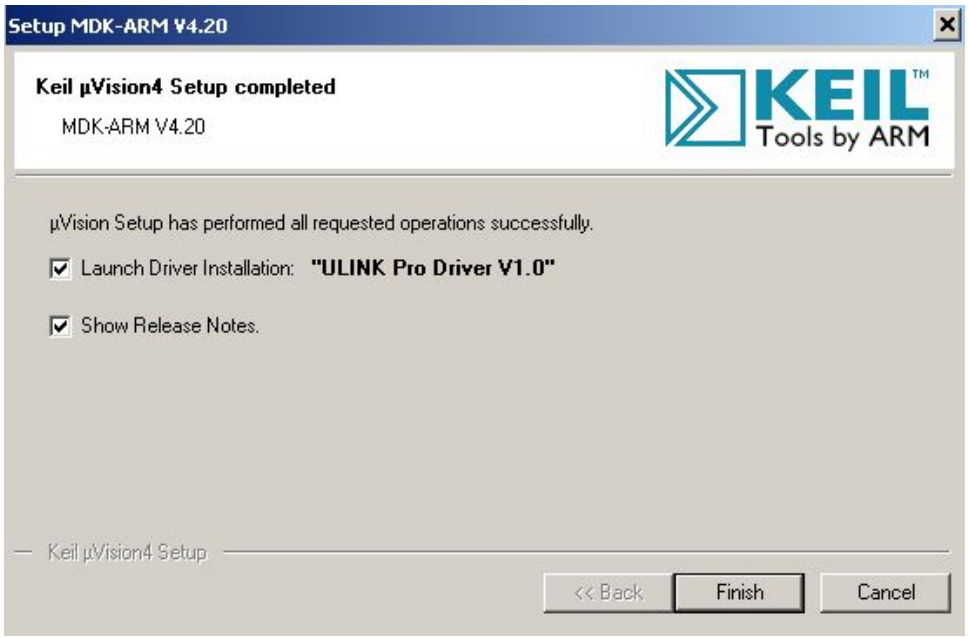


Figure 3.2: MDK-ARM Installation Steps: Finish

Figure 3.3: MDK-ARM Installation Steps: ULINK Pro Driver

During the process of the installation of the MDK-ARM, you will be asked to add example code. Choose Keil(NXP) MCB1xxx Boards example projects (see Figure 3.1. At the last step of MDK-ARM installation, be sure that the launch the "ULINK Pro Driver V1.0" driver installation check box is checked (see Figure 3.2. Once you click "Finish" button, the ULINK Pro Driver installation starts. Click "Install" button to install the driver (see Figure 3.3).

3.2 Creating an Application in μVision4 IDE

To get started with the Keil IDE, the MDK-ARM Primer

`http://www.keil.com/support/man/docs/gsac/`

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 that is connected to the lab PC. Note the HelloWorld example uses polling rather than interrupt.

3.2.1 Create a New Project

1. Create a folder named "HelloWorld" on your computer.
2. Copy the following files to "HelloWorld" folder:

- manual_code\UART_polling\src\UART_polling.h
- manual_code\UART_polling\src\UART_polling.c
- manual_code\Startup\system_LPC17xx.c

3. Create a new μ`Vision` project by click
   - Project → New μ`Vision` Project (See Figure 3.4)

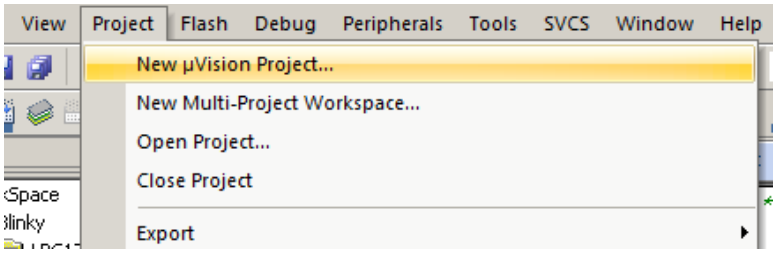
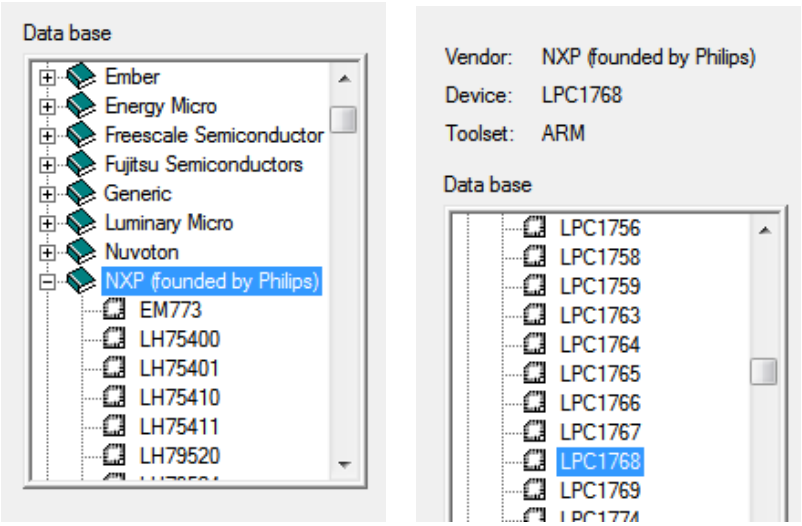


Figure 3.4: Keil IDE: Create a New Project

   - Choose NXP(Founded by Philips) → LPC1768 (See Figure 3.5(a) and Figure 3.5(b))



(a) Choose NXP (b) Choose LPC1768

Figure 3.5: Keil IDE: Choose MCU

   - Answer "Yes" to copy the startup code (See Figure 3.6).

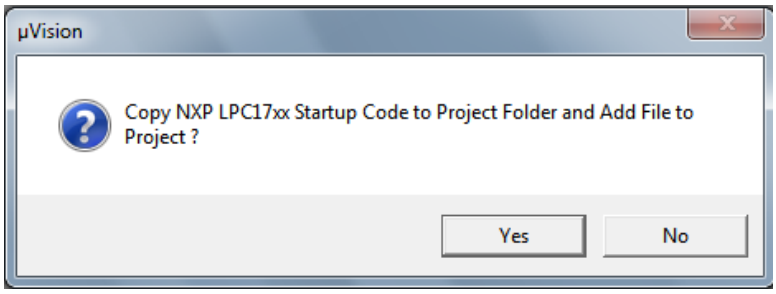


Figure 3.6: Keil IDE: Copy Startup Code

3.2.2 Managing Project Components

You just finished creating a new project. One the left side of the IDE is the Project window and expand all objects, you will see the default project setup as shown in Figure 3.7.

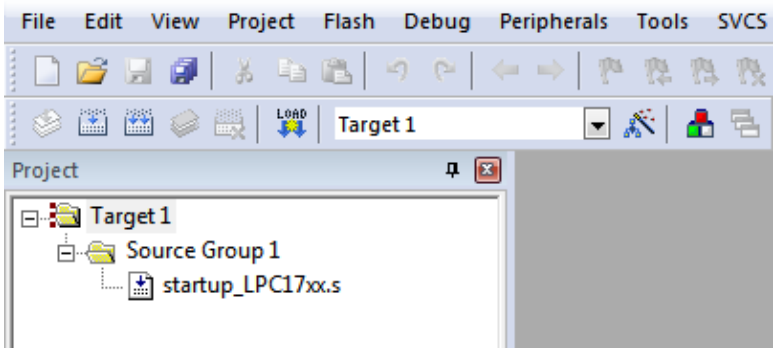


Figure 3.7: Keil IDE: A default new project

1. Rename the Target
   The "Target 1" is the default name of the project build target and you can rename it by clicking the target name to highlight it and then click the highlighted name to input a new target name, say "Hello World 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 the startup code "startup_LPC17xx.s" is put under this source group. You can rename the source group by clicking the source group name to highlight it and then click again to input a new name, say "Startup Code".

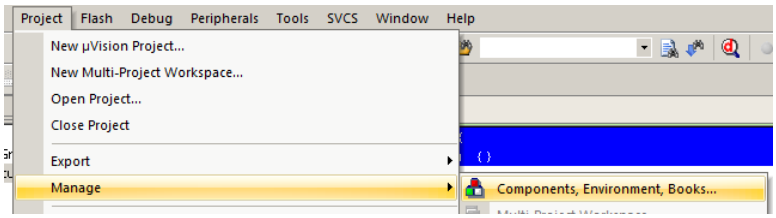


Figure 3.8: Keil IDE: Manage Project Components

3. Add a New Source Group You can add new source groups to your project. Click "Project → Manage" → "Components, Environment, Books..." (See Figure 3.8 You can now add new source group to the project. Let's add "Source Code" source group to the project (See Figure 3.9. We add a new source group and name it "Source Code". Your project will now look like Figure 3.10

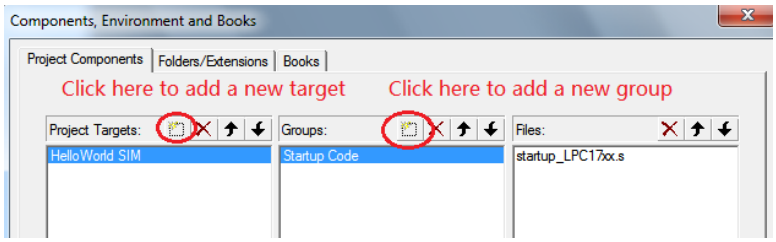


Figure 3.9: Keil IDE: Manage Components Window

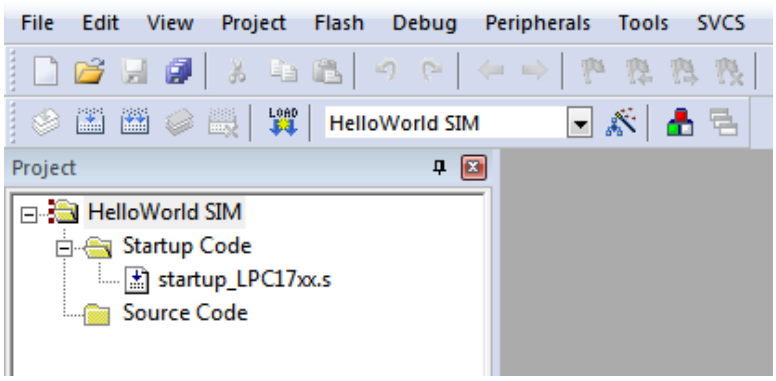


Figure 3.10: Keil IDE: Updated Project Profile

4. Add Source Code to a Source Group Now add "UART_polling.c" and "system_LPC17xx.c" to "Source Code" group by double clicking "Source Code" and choose these two files from the file window. Double click 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.

5. Create a new source file The project does not have a main function yet. We now create a new file by clicking "File"→"New". Before typing anything to the file, save the file and name it "main.c". Put the following code to the main.c file:

```
#include <LPC17xx.h>
#include <system_LPC17xx.h>
#include "uart_polling.h"
int main() {
    SystemInit();
    uart0_init();
    uart0_put_string("Hello World!\n\r");
    return 0;
}
```

Then add main.c to the "Source Code" group. Your final project would look like the screen shot in Figure 3.11.

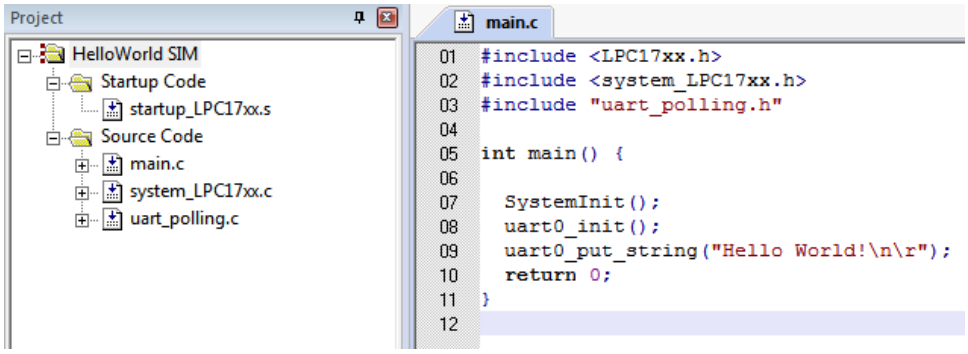


Figure 3.11: Keil IDE: Final Project Setting

3.2.3 Build and Download

To build the target, click Project→Build target. If nothing is wrong, the Build Output window at the bottom of the IDE will show a log similar like the one shown in Figure 3.12 To download the code to the board, click the Load button. The download is through the Ulink-Me.

Open up the HyperTerminal on your PC and choose COM2. Press the Reset button on the board and you should see "Hello World!" displayed on the HyperTerminal.

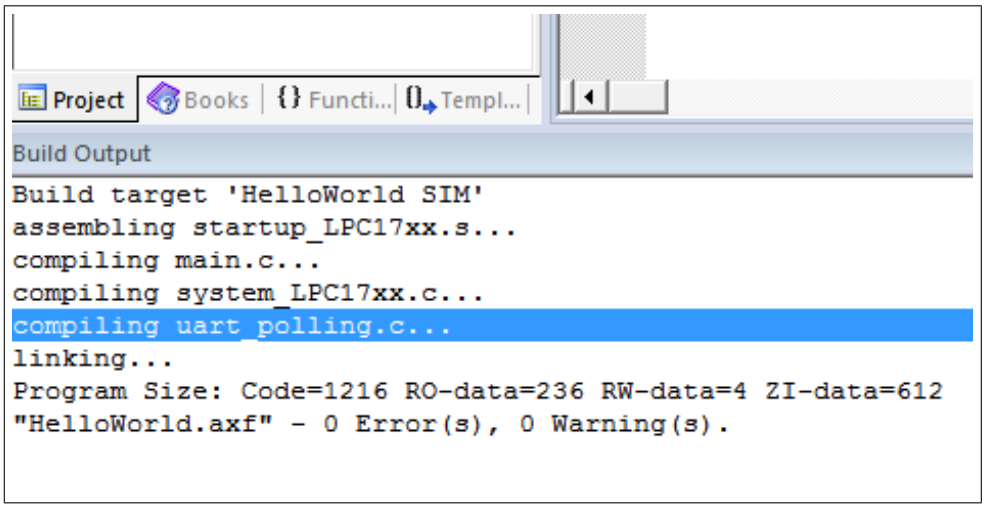


Figure 3.12: Keil IDE: Build Target

3.3 Debugging

You can use either the simulator within the IDE or the ULINK Cortex Debugger to debug your program. To start a debug session, click Debug→Start/Stop Debug Session from the IDE menu bar or press Ctrl+F5. Figure 3.13 shows the a typical debug session interface.

As any other GUI debugger, the IDE allows you to set up break points and step through your source code. It also shows the registers, which is very helpful for debugging low level code. Click View, Debug and Peripherals from the IDE menu bar and explore the functionality of the debugger.

3.3.1 Disabling CRP

In order to avoid stealing firmware , the LPC1768 provides Code Read Protection (CRP) that allows fine-grain control about which areas of the memory can be read. A detailed description is found in Section 32.6 of [2]. In essence if the Assembler Directive NO_CRP is not present, the hardware is initialized to only make the firmware read-only (see Figure 3.14)

Since it is advisable to change values on the fly when debugging, the CRP should be disabled during prototyping. Open up the target option window and click the Asm tab. Put “NO_CRP” as shown in Figure 3.15

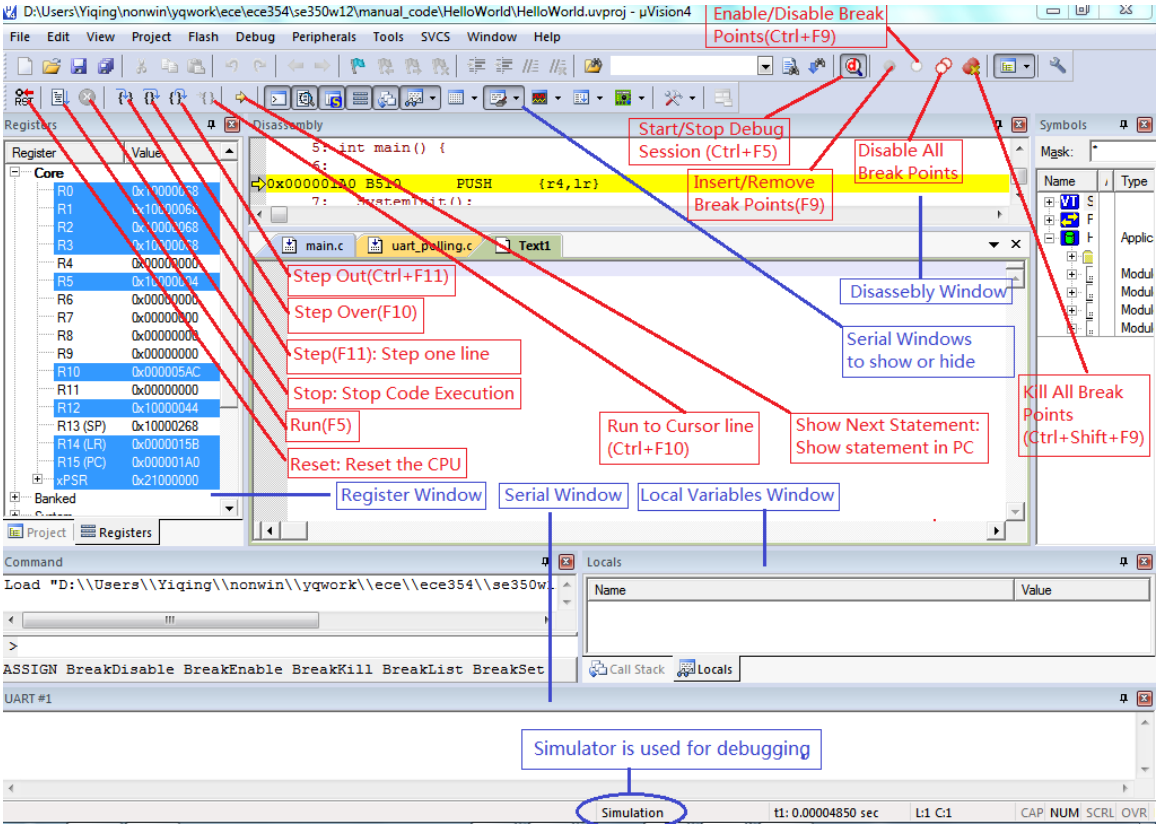


Figure 3.13: Keil IDE: Debugging

```
110                     IF      :LNOT::DEF:NO_CRP
111                     AREA    |.ARM.__at_0x02FC|, CODE, READONLY
112 CRP_Key             DCD     0xFFFFFFFF
113                     ENDIF
114
115
116                     AREA    |.text|, CODE, READONLY
```

Figure 3.14: startup_LPC17xx.s excerpt

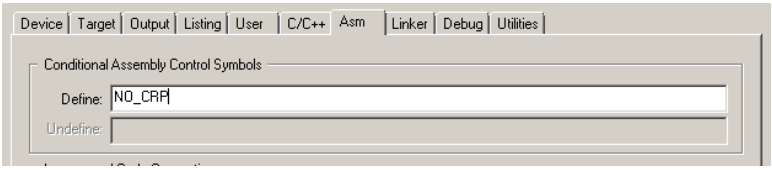


Figure 3.15: Keil IDE: Using Simulator for Debugging

3.3.2 Simulation

Most of the development normally is done under the simulation mode. The default setting of the project uses the simulator to debug as shown in the target option (see Figure 3.16 Instead of load the program to the board for execution, you can run the code using the

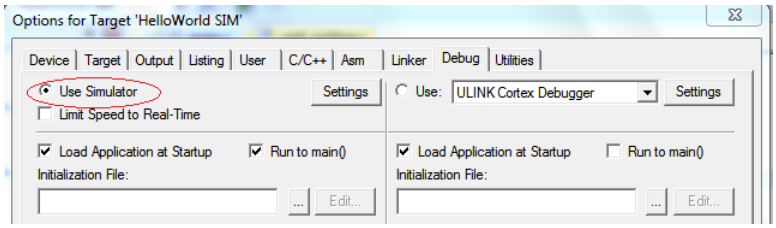


Figure 3.16: Keil IDE: Using Simulator for Debugging

debugger under simulation mode.

3.3.3 Configure In-Memory Execution Using ULINK Cortex Debugger

When you debug hardware related problems, you most likely will find the ULINK Cortex Debugger is helpful. You need to configure the debugger as shown in Figure 3.17.

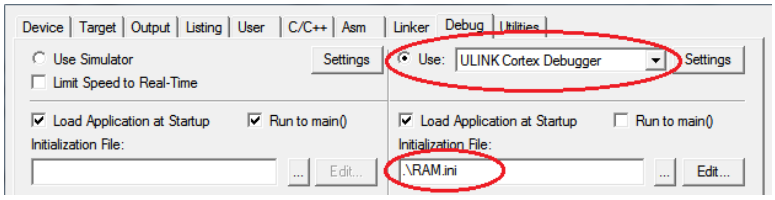
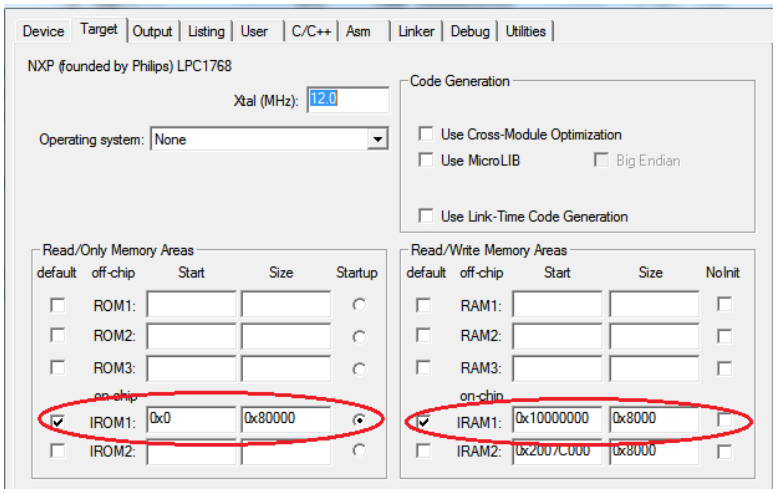


Figure 3.17: Keil IDE: Using ULINK Cortex Debugger

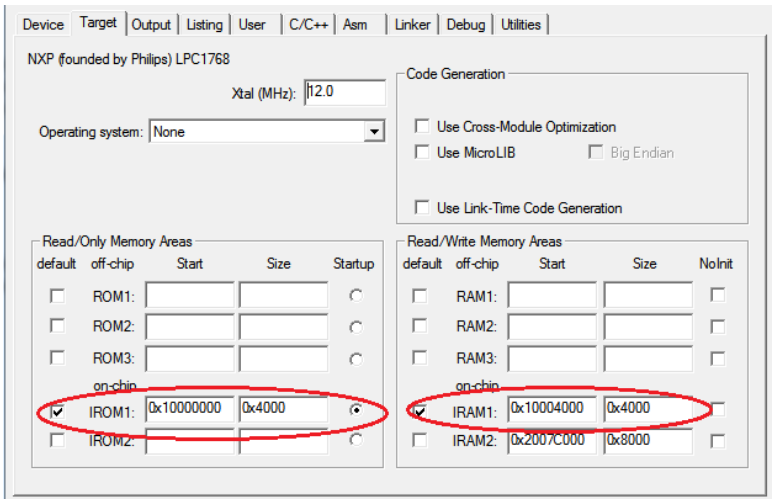
The default image memory map setting is that the code is executed from the ROM (see Figure3.18(a). Since the ROM portion of the code needs to be flashed in order to be executed on the board, this incurs wear-and-tear on the on-chip flash of the LPC1768. Since most attempts to write a functioning RTX 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. We can create a RAM target where the code starting address is in RAM (see Figure 3.18(b). An

initialization file RAM.ini is needed to do the proper setting of SP, PC and vector table offset register.



(a) Default Memory Setting



(b) In-Memory Execution Setting

Figure 3.18: Keil IDE: Configure for In-Memory Execution

Bibliography

[1] MCB1700 User's Guide. http://www.keil.com/support/man/docs/mcb1700. 3

[2] LPC17xx User Manual, Rev2.0, 2010. 2, 7, 8, 9, 21

[3] J. Yiu. *The Definitive Guide to the ARM Cortex-M3*. Newnes, 2009. 5, 6, 11, 12