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 [3], 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

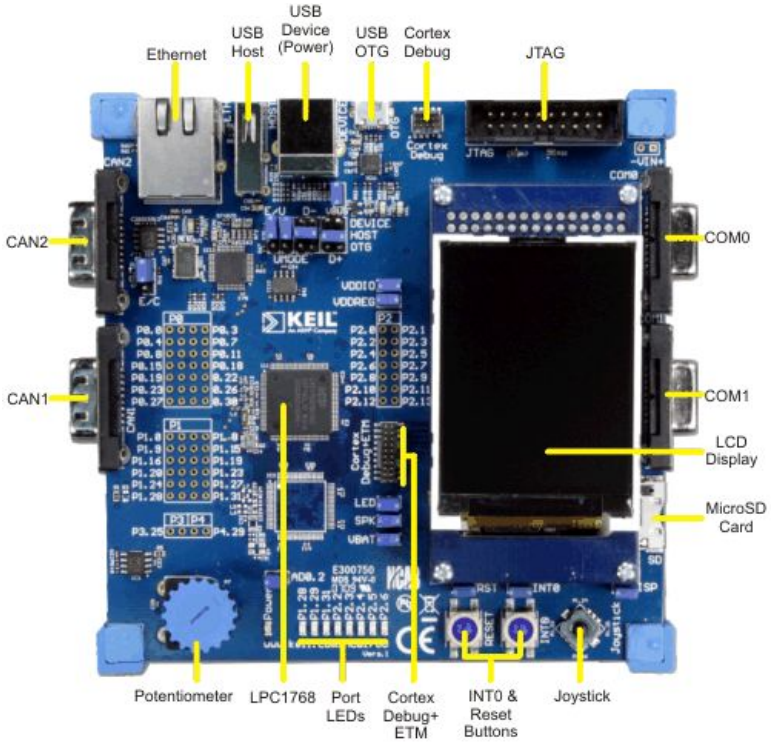


Figure 2.1: MCB1700 Board Components [1]

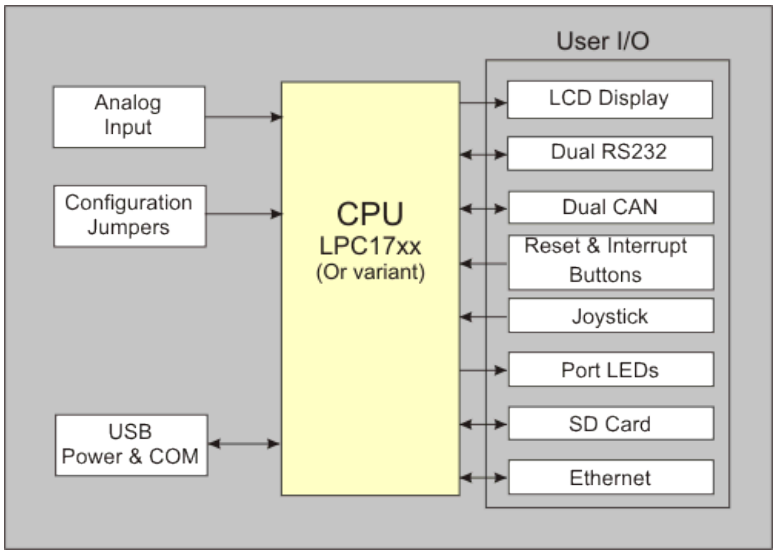


Figure 2.2: MCB1700 Board Block Diagram [1]

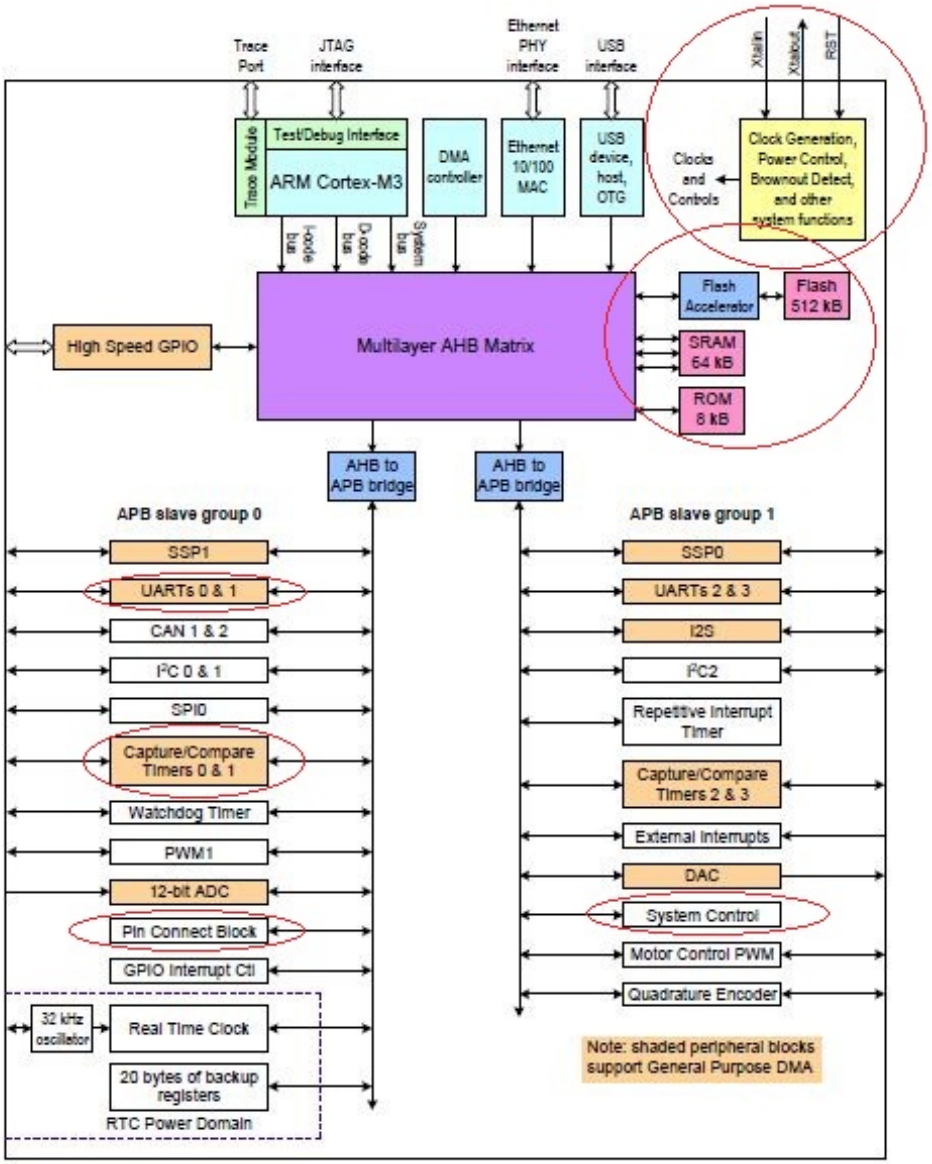


Figure 2.3: LPC1768 Block Diagram

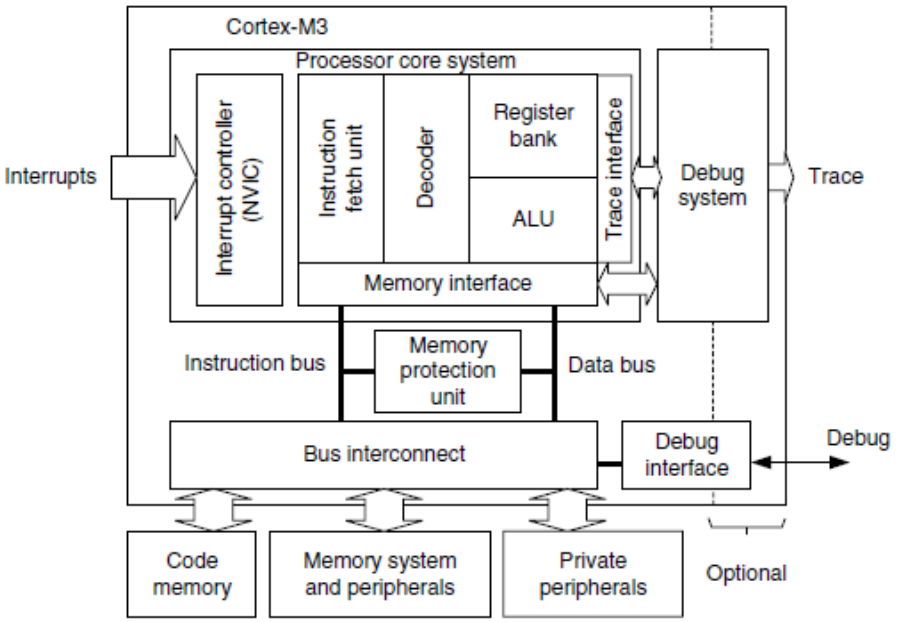


Figure 2.4: Simplified Cortex-M3 Block Diagram[4]

processor [4]. 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.

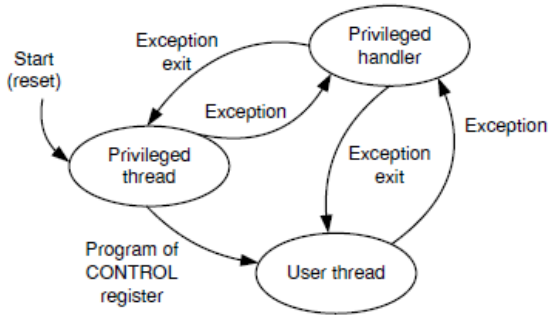


Figure 2.5: Cortex-M3 Operating Mode and Privilege Level[4]

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 [3] 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:

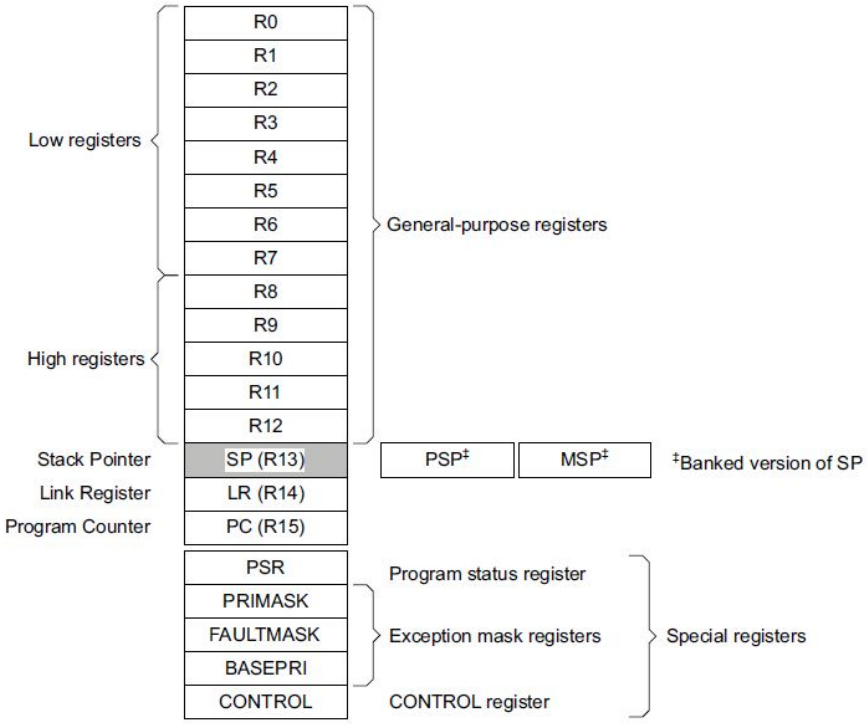


Figure 2.6: Cortex-M3 Registers[3]

- 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 [3] 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.

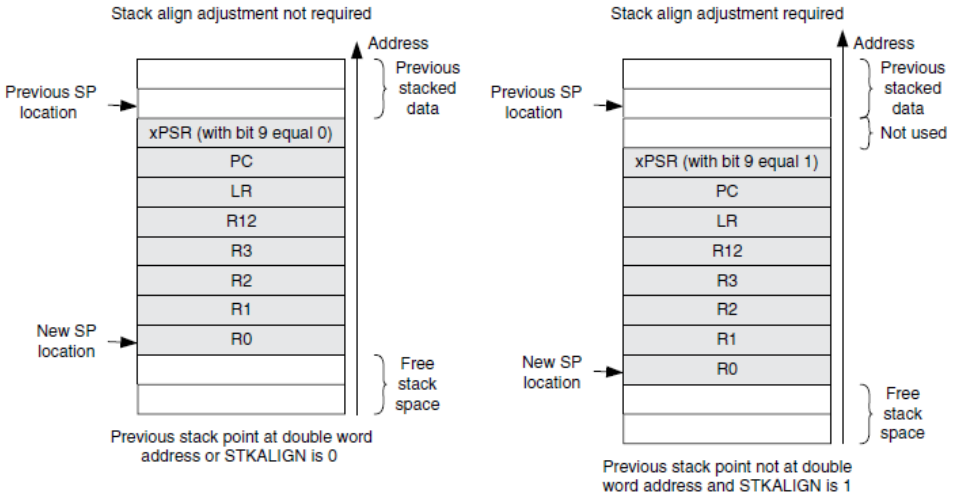


Figure 2.7: Cortex-M3 Exception Stack Frame [4]

- 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 [4]

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.

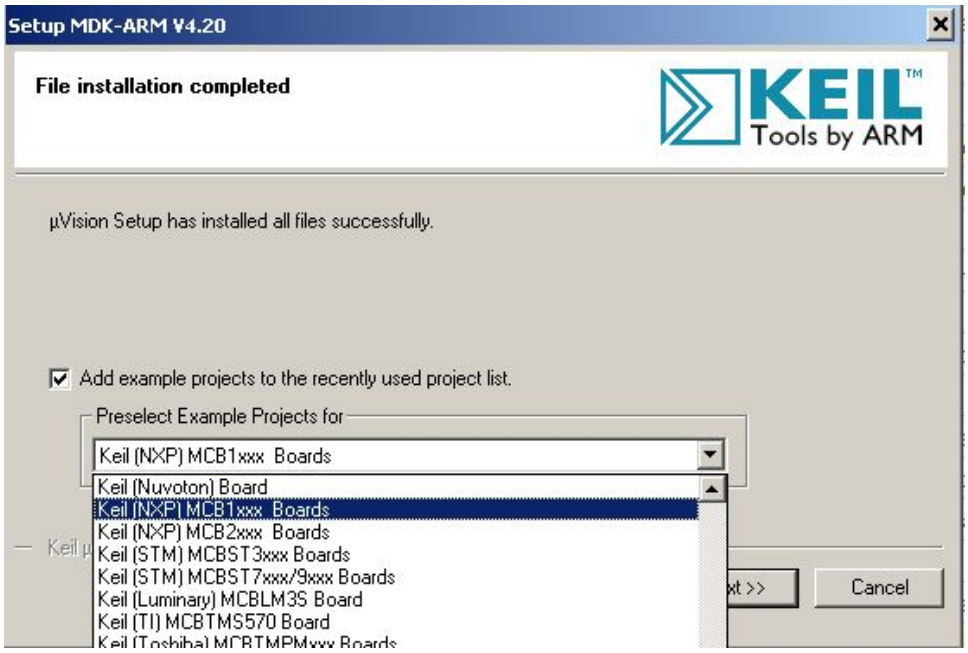


Figure 3.1: MDK-ARM Installation Steps: Choose Example Projects

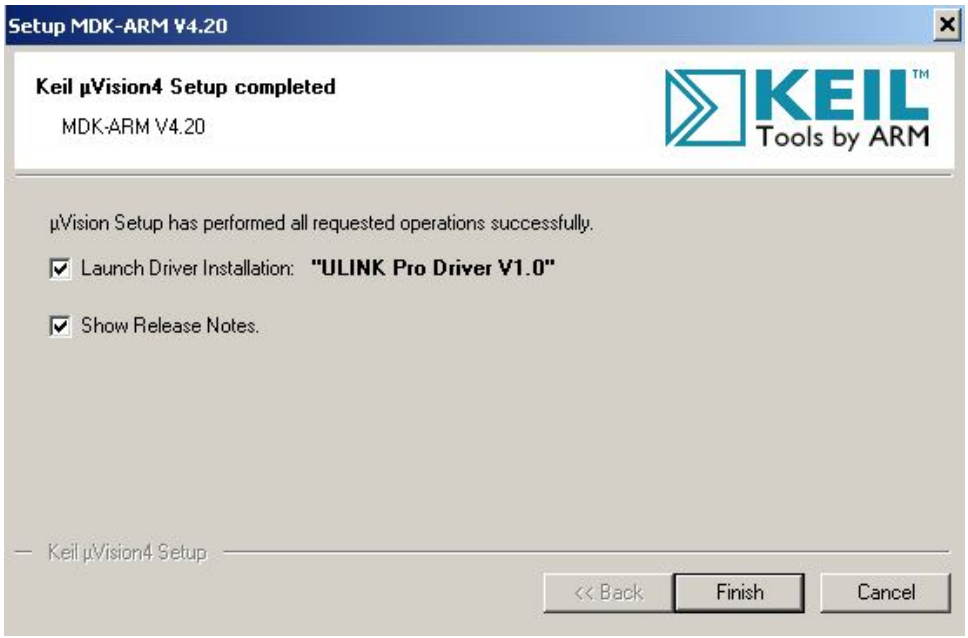


Figure 3.2: MDK-ARM Installation Steps: Finish

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)

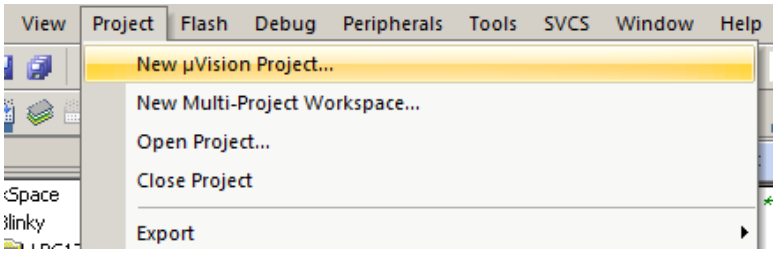
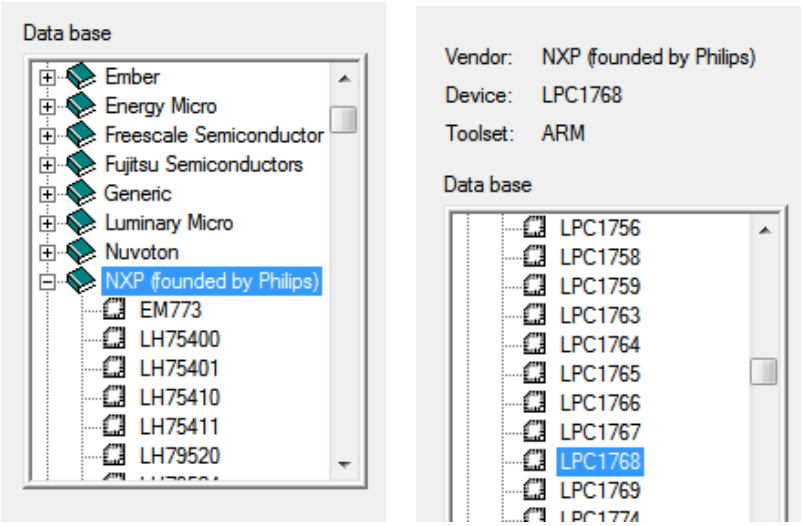


Figure 3.4: Keil IDE: Create a New Project

   - 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).

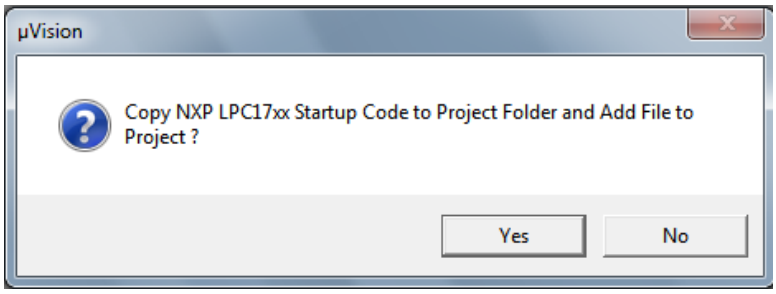


Figure 3.6: Keil IDE: Copy Startup Code

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.

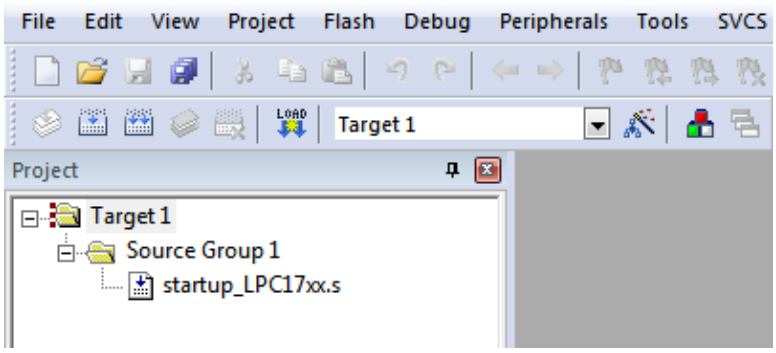


Figure 3.7: Keil IDE: A default new project

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 "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 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".

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

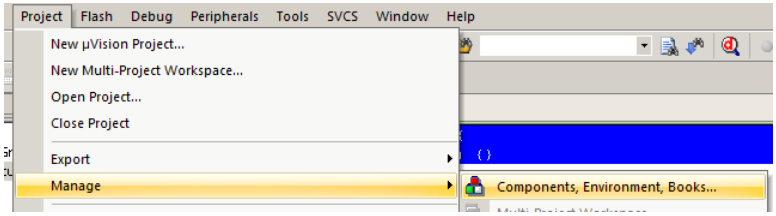


Figure 3.8: Keil IDE: Manage Project Components

groups to the project. Let's add "System Code" and "Source Code" source groups to the project (See Figure 3.9. Your project will now look like Figure 3.10

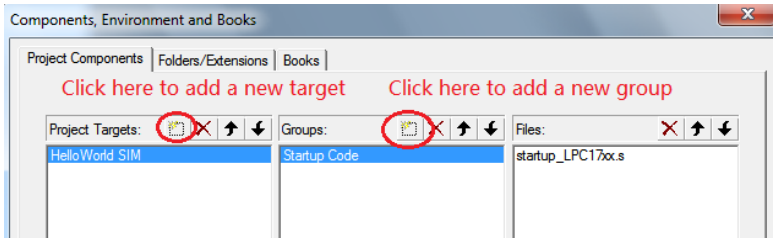


Figure 3.9: Keil IDE: Manage Components Window

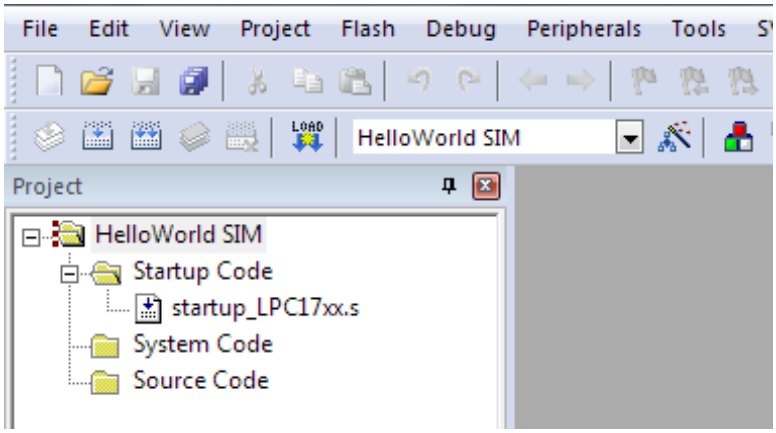


Figure 3.10: Keil IDE: Updated Project Profile

4. Add Source Code to a Source Group
   Now add "system_LPC17xx.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 3.11).

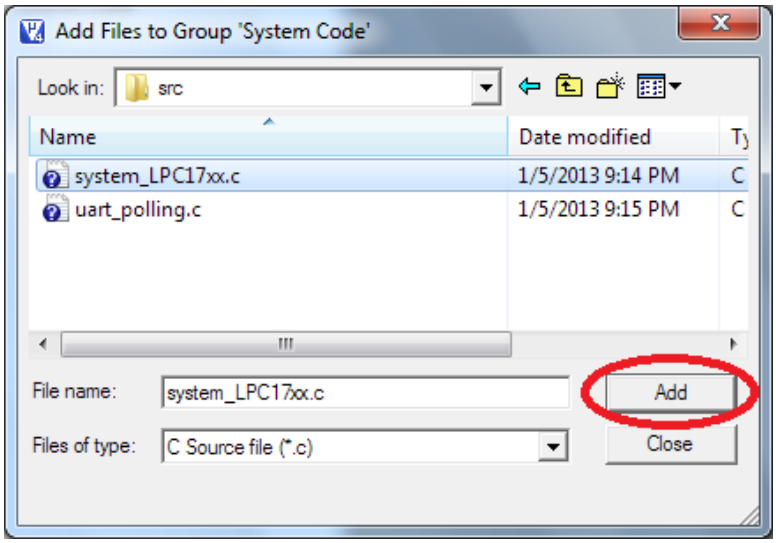


Figure 3.11: Keil IDE: Add Source File to Source Group

Similarly, add "uart_polling.c" to "Source Code" group. Your project will now look like Figure 3.12.

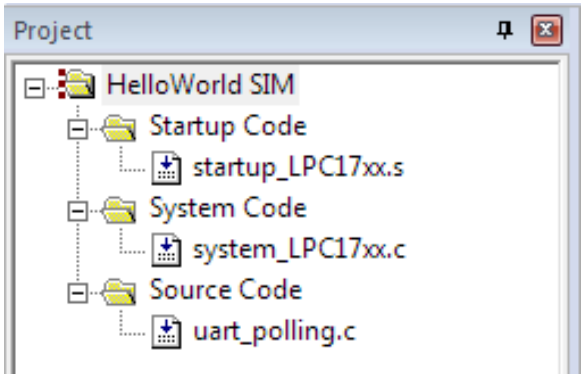


Figure 3.12: Keil IDE: Updated Project Profile

5. Create a new source file
   The project does not have a main function yet. We now create a new file by clicking

the “New” button (See Figure 3.13). Before typing anything to the file, save the file

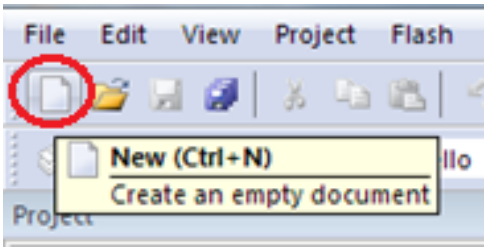


Figure 3.13: Keil IDE: Create New File

and name it “main.c”. Put the following code to the main.c file:

```
#include <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 3.14.

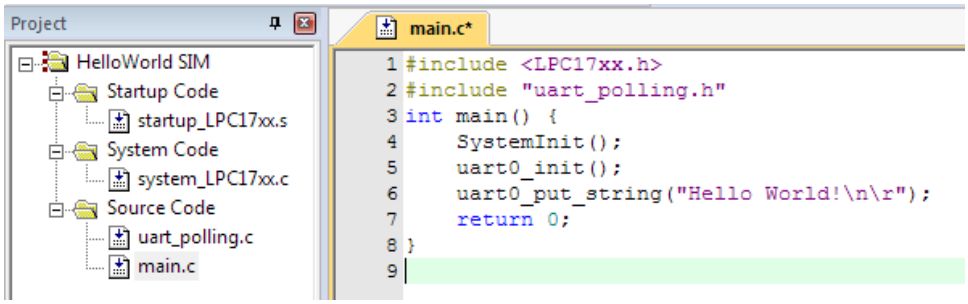


Figure 3.14: Keil IDE: Final Project Setting

3.2.3 Build and Download

To build the target, click the “Build” button (see Figure 3.15). If nothing is wrong, the

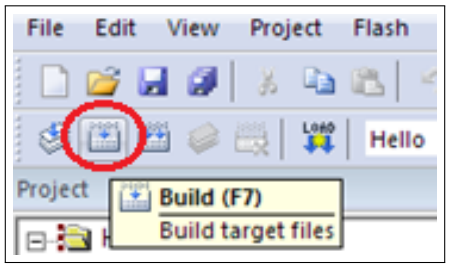


Figure 3.15: Keil IDE: Build Target

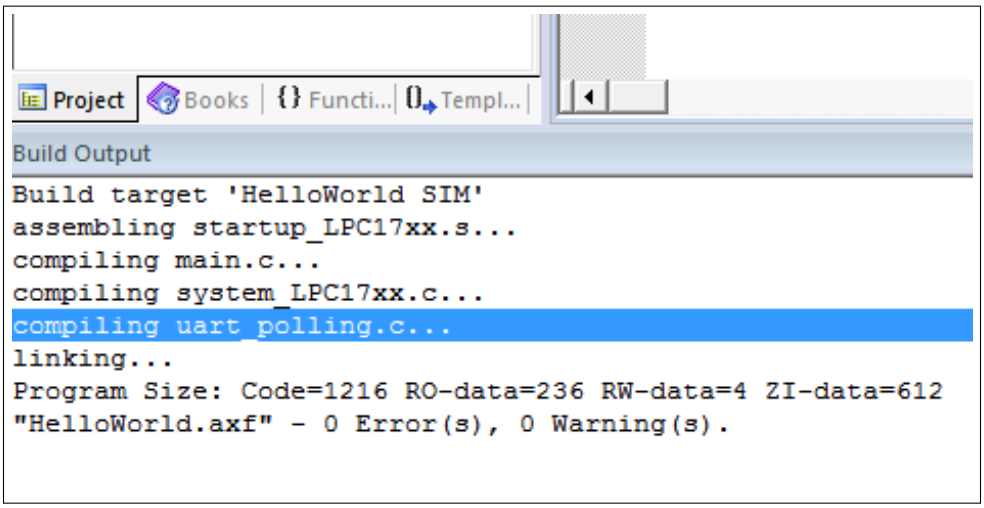


Figure 3.16: Keil IDE: Build Target

build output window at the bottom of the IDE will show a log similar like the one shown in Figure 3.16

To download the code to the board, click the "Load" button (see Figure 3.17). The

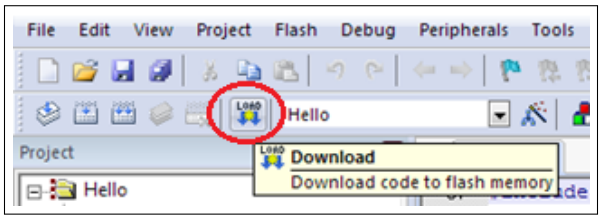


Figure 3.17: Keil IDE: Download Target to Flash

download is through the Ulink-Me.

You will need a terminal emulator such as PuTTY that talks directly to COM ports in order to see output of the serial port. Open up the PuTTY on your PC and choose

COM2. An example PuTTY Serial configuration is shown in Figure 3.18. Press the Reset button on the board and you should see "Hello World!" displayed on PuTTY.

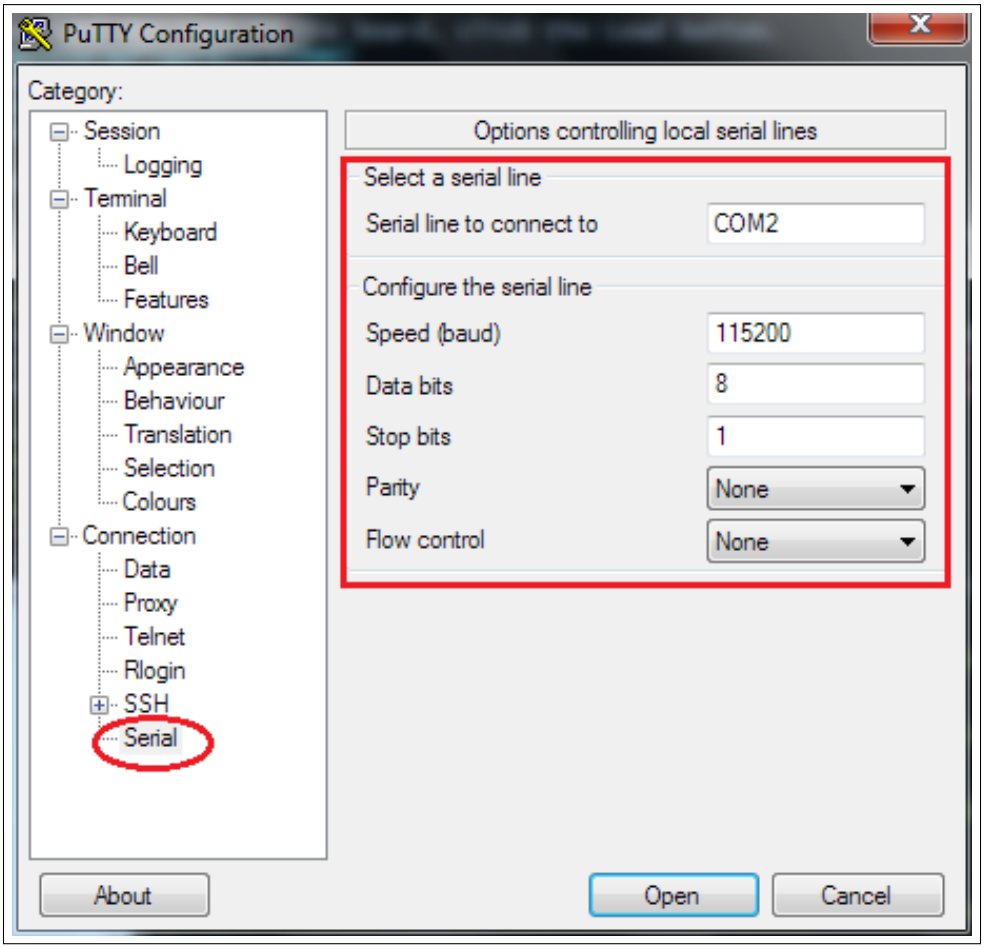


Figure 3.18: Keil IDE: Sample PuTTY Serial Configuration

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 3.19 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

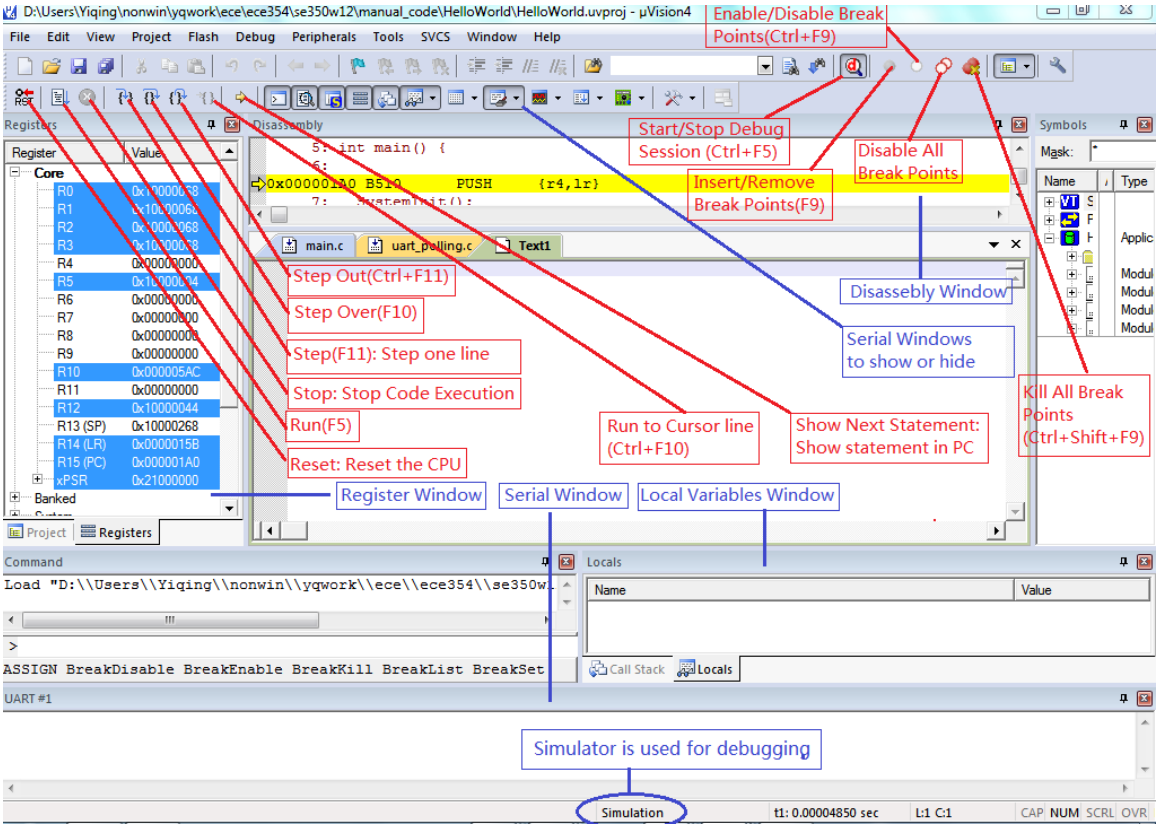


Figure 3.19: Keil IDE: Debugging

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 [3]. In essence if the Assembler Directive NO_CRP is not present, the hardware is initialized to only make the firmware read-only (see Figure 3.20)

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 3.21

```
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 3.20: startup_LPC17xx.s excerpt

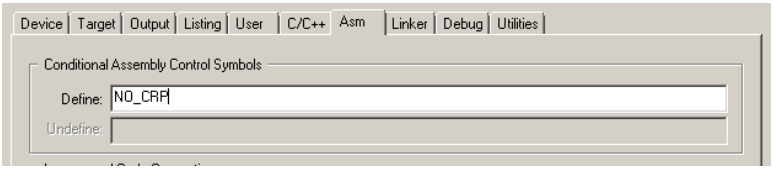


Figure 3.21: Keil IDE: Using Simulator for Debugging

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 3.22 Instead of load the program to the board for execution, you can run the code using the

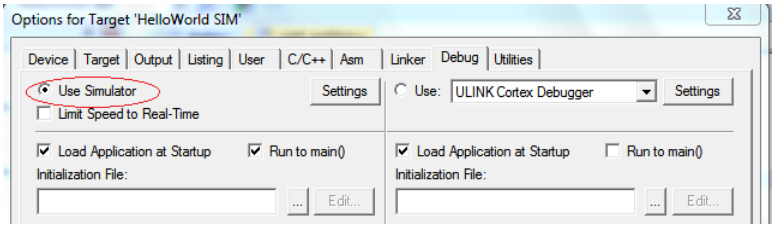


Figure 3.22: Keil IDE: Using Simulator for Debugging

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 3.23.

The default image memory map setting is that the code is executed from the ROM (see Figure3.24(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.

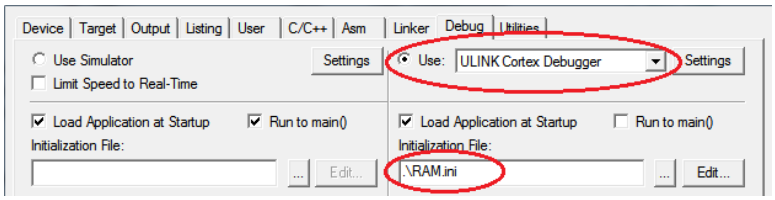
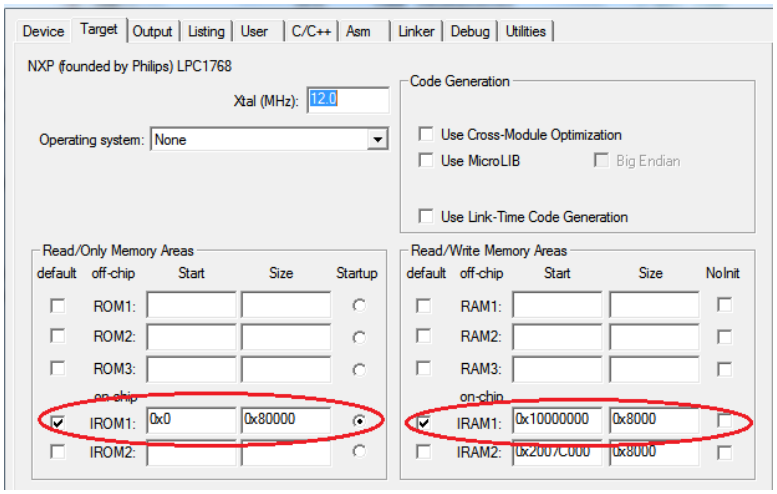


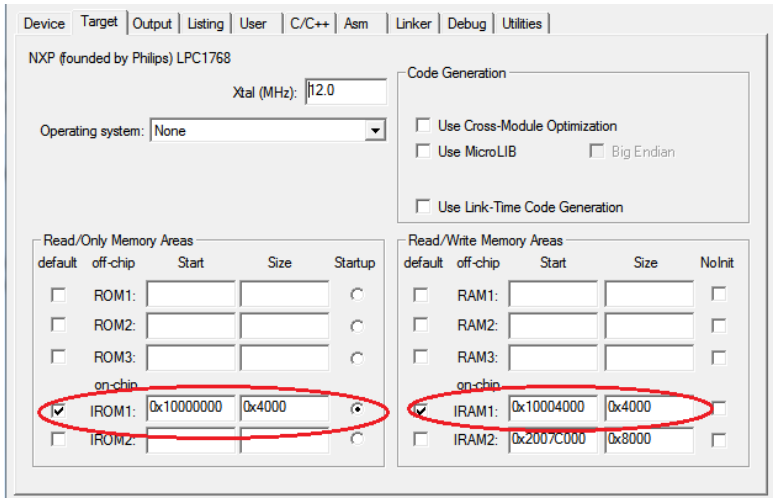
Figure 3.23: Keil IDE: Using ULINK Cortex Debugger

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 3.24(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 3.24: Keil IDE: Configure for In-Memory Execution

Chapter 4

Programming MCB1700

4.1 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 4.1 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 [3].

4.2 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

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 4.1: Assembler instruction examples

C compiler follows the AAPCS to generate the assembly code. Table 4.2 lists registers used by the AAPCS.

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 4.2: Core Registers and AAPCS Usage

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 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.

4.3 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 4.1). This improves software portability and re-usability. It enables software solution suppliers to develop products that can work seamlessly with device libraries from various silicon vendors [2].

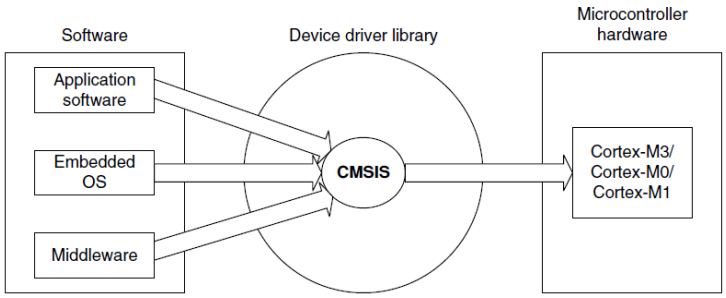


Figure 4.1: Role of CMSIS[4]

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.

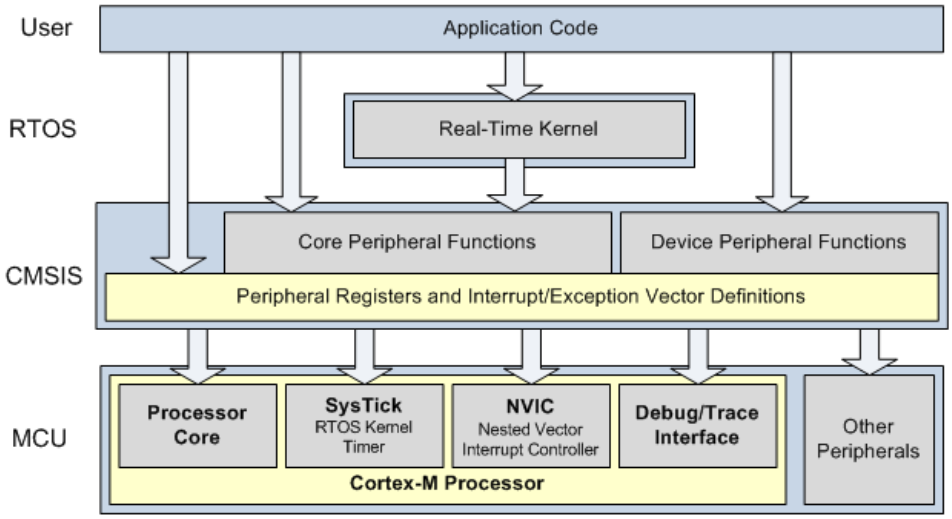


Figure 4.2: CMSIS Organization[2]

4.3.1 CMSIS files

The CMSIS is divided into multiple layers (See Figure 4.2). 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 4.3).

By including the `<device>.h` (e.g., `LPC17xx.h`) file into your code file. The first step to initialize the system can be done by calling the CMSIS function as shown in Listing 4.1.

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

Listing 4.1: 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 4.3.3.

4.3.2 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 4.4). As an

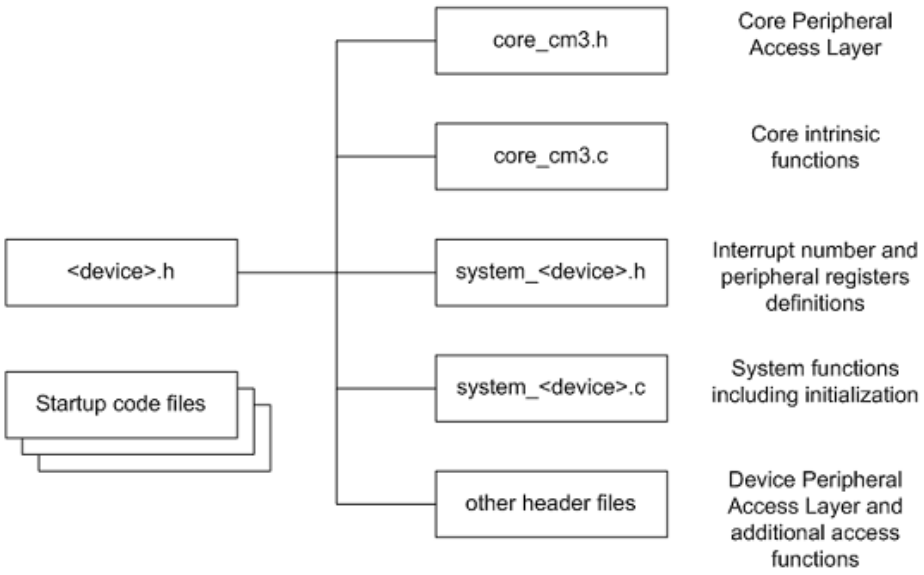


Figure 4.3: CMSIS Organization[2]

	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 4.4: CMSIS NVIC Functions[2]

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
```

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

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

Another way is to use the embedded assembly code:

```
__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,
}
```

4.3.4 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 4.3 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 [3] for the complete list of intrinsic functions.

4.3.5 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
```

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 4.3: CMSIS intrinsic functions defined in `core_cmFunc.h`

4.4 Accessing C Symbols from Assembly

Only embedded assembly is support in Cortex-M3 (i.e. inline assembly is not supported). To write an embedded assembly function, you need to use the `__asm` keyword. For example that the function "`embedded_asm_function`" in Listing 4.2 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

```
typedef struct pcb {
    struct * mp_next;
    uint32_t m_sp;
    //.....
} pcb_t;

pcb_t g_pcb;
uint32_t g_var;

__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.m_sp
    LDR R4, =__cpp(g_var)  ; load R4 with the value of g_var
}
```

Listing 4.2: Example of accessing global variable from assembly

- A C function

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

- A constant expression in the range of $0 - 255$ defined in C.

```
uint8_t const g_flag;

__asm embedded_asm_function(void) {
    ;......
    MOV R4, #__cpp(g_flag) ; load g_flag value to R4
    ;......
}
```

  Note the `MOV` instruction only applies to immediate constant value in the range of $0 - 255$.

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. For example

```
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
}
```

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.

4.5 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 [3] (referred as LPC17xx_UM in the sample code comments). Listings 4.3 and 4.4 give one sample implementation of programming UART0 interrupts.

```
/**
 * @file uart.h
 */

#ifndef _UART_H_
#define _UART_H_

#include <stdint.h> // typedefs

// 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



#define BIT(X)     ( 1<<X )
#define UART_8N1   0x83 // 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 uart0_init()        uart_init(0)

int uart_init(int n_uart);      // initialize the n_uart
void uart_send_string( uint32_t n_uart, uint8_t *p_buffer, uint32_t len );
                                // write a string to the n_uart
#endif // ! _UART_H_
```

Listing 4.3: UART0 IRQ Sample Code uart.h

```
/**
 * @file  uart_irq.c
 * @brief uart interrupt setup and handling functions
 */
#include <LPC17xx.h>
#include "uart.h"

volatile uint8_t g_UART0_TX_empty=1;
volatile uint8_t g_UART0_buffer[BUFSIZE];
volatile uint32_t g_UART0_count = 0;

/**
 * @brief: initialize the n_uart
 * NOTES: only fully supports uart0 so far, but can be easily extended
 * to other uarts. The step number in the comments matches the
 * item number in Section 14.1 on pg 298  of LPC17xx_UM
 */
int uart_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, 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.
        // tables 40 and 42 on pg56 and pg57 in LPC17xx_UM
        // Check the PLL0 configuration to see how XTAL=12.0MHZ gets to
        // CCLK=100MHZ  in system_LPC17xx.c file
```

```
        // 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 for pin settings
        // Done before Steps3-4 for better coding purpose.

        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) {

        // see Table 79 on pg108 in LPC17xx_UM for pin settings
        LPC_PINCON->PINSEL0 |= (2 << 0);   // Pin P2.0 used as TXD1 (Com1)
        LPC_PINCON->PINSEL0 |= (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;// see uart.h file

    // Step 3b: 115200 baud rate @ 25.0 MHZ PCLK
    // seep section 14.4.12.1 pg313-315 in LPC17xx_UM for baud rate
       calculation
    pUart->DLM = 0;           // see table 274, pg302 in LPC17xx_UM
    pUart->DLL = 9;           // see table 273, pg302 in LPC17xx_UM
    pUart->FDR = 0x21;        // FR = 1.507 ~ 1/2, DivAddVal = 1, MulVal = 2
                              // FR = 1.507 = 25MHZ/(16*9*115200)
                              // see table 285 on pg312 in LPC_17xxUM


    // Step 4 FIFO setup
    pUart->FCR = 0x07;        // enable Rx and Tx FIFOs, clear Rx and Tx
       FIFOs
                              // Trigger level 0 (1 char per interrupt)
                              // see table 278 on pg305 in LPC17xx_UM


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

```
    // Step 6 Interrupt setting
    // Step 6a: enable interrupt bits 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
    pUart->LCR &= ~(BIT(7)); // disable the Divisior Latch Access Bit DLAB=0
    pUart->IER = IER_RBR | IER_THRE | IER_RLS;

    // Step 6b: enable the UART interrupt from the system level
    // Use CMSIS call
    NVIC_EnableIRQ(UART0_IRQn);

    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
    uint8_t LSR_Val;        // LSR Value
    uint8_t dummy = dummy;  // dummy variable to clear interrupt upon LSR
       error
    LPC_UART_TypeDef * pUart = (LPC_UART_TypeDef *)LPC_UART0;

    // Reading IIR automatically acknowledges the interrupt
    IIR_IntId = (pUart->IIR) >> 1 ; // skip pending bit in IIR
```

```
    if (IIR_IntId & IIR_RDA) {  // Receive Data Avaialbe
        // Note: read RBR will clear the interrupt
        g_UART0_buffer[g_UART0_count++] = pUart->RBR; // read from the uart
        if ( g_UART0_count == BUFSIZE ) {
            g_UART0_count = 0;  // buffer overflow
        }
    } else if (IIR_IntId & IIR_THRE) {  // THRE Interrupt,
                                        // transmit holding register empty
        LSR_Val = pUart->LSR;
        if(LSR_Val & LSR_THRE) {
            g_UART0_TX_empty = 1;  // UART is ready to transmit
        } else {
            g_UART0_TX_empty = 0;  // UART is not ready to transmit yet
        }

    } else if (IIR_IntId & IIR_RLS) {
        LSR_Val = pUart->LSR;
        if (LSR_Val  & (LSR_OE|LSR_PE|LSR_FE|LSR_RXFE|LSR_BI) ) {
            // There are errors or break interrupt
            // Read LSR will clear the interrupt
            dummy = pUart->RBR; // Dummy read on RX to clear interrupt,
                                // then bail out
            return ;    // error occurs, return
        }
        // If no error on RLS, normal ready, save into the data buffer.
        // Note: read RBR will clear the interrupt
        if (LSR_Val & LSR_RDR) { // Receive Data Ready
            g_UART0_buffer[g_UART0_count++] = pUart->RBR; // read from the
                uart
            if ( g_UART0_count == BUFSIZE ) {
                g_UART0_count = 0;  // buffer overflow
            }
        }
    } else {  // IIR_CTI and reserved combination are not implemented yet
        return;
    }
}

void uart_send_string( uint32_t n_uart, uint8_t *p_buffer, uint32_t len )
{
    LPC_UART_TypeDef *pUart;

    if(n_uart == 0 ) {  // UART0 is implemented
        pUart = (LPC_UART_TypeDef *) LPC_UART0;
    } else {  // other UARTs are not implemented
        return;
    }
```

```
    while ( len != 0 ) {
        // THRE status, contain valid data
        while ( !(g_UART0_TX_empty & 0x01) );
        pUart->THR = *p_buffer;
        g_UART0_TX_empty = 0;  // not empty in the THR until it shifts out
        p_buffer++;
        len--;
    }
    return;
}
```

Listing 4.4: UART0 IRQ Sample Code uart_irq.c

4.6 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 [3]. Listings 4.5 and 4.6 give one sample implementation of programming TIMER0 interrupts. The timer interrupt fires every one millisecond.

```
/**
 * @file timer.h
 */

#ifndef _TIMER_H_
#define _TIMER_H_

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

#endif // ! _TIMER_H_
```

Listing 4.5: Timer0 IRQ Sample Code timer.h

```
/**
 * @file timer.c
 * @brief Timer irq setup and handling routine.
 */

#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, table 46 pg63 in LPC17xx_UM
        // enable TIMER0 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.
        // tables 40 and 42 on pg56 and pg57 in LPC17xx_UM
        // Check the PLL0 configuration to see how XTAL=12.0MHZ
        // gets to CCLK=100MHZ in system_LPC17xx.c file
        // 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 for pin settings

        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
    pTimer->PR = 12499;  // 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

    // Step 4.2: MR setting, see section 21.6.7 on page 496 of LPC17xx_UM
    pTimer->MR0 = 1;
```

```
    // Step 4.3: MCR setting, see table 429 on page 496 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 page 494 of LPC17xx_UM
    pTimer->TCR = 1;

    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 TIMER0_IRQHandler(void)
{
    PRESERVE8
    IMPORT c_TIMER0_IRQHandler
    PUSH{r4-r11, lr}
    BL c_TIMER0_IRQHandler
    POP{r4-r11, pc}
}
/**
 * @brief: c UART0 IRQ Handler
 */
void c_TIMER0_IRQHandler(void)
{
    LPC_TIM0->IR = BIT(0);  // ack interrupt,
                            // see section 21.6.1 on pg 493 of LPC17XX_UM
    g_timer_count++ ;
}
```

Listing 4.6: Timer0 IRQ Sample Code timer.c

Bibliography

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

[2] MDK Primer. http://www.keil.com/support/man/docs/gsac. 31, 32, 33

[3] LPC17xx User Manual, Rev2.0, 2010. 2, 7, 8, 9, 24, 28, 34, 37, 42

[4] J. Yiu. *The Definitive Guide to the ARM Cortex-M3*. Newnes, 2009. 5, 6, 11, 12, 31