

University of Waterloo
Electrical and Computer Engineering Department

Digital Computers
ECE-222 Lab manual

Winter 2023

January 10, 2023

Developed by: Rasoul Keshavarzi, Hiren D. Patel
Assisted by: Roger Sanderson, Eric Praetzel, Gordon B. Agnew

Lab Instructor: Julius Olajos

This manual is for the exclusive use of students registered in the course.

Reproduction or use for any other purpose is prohibited.

Contents

| | |
|--|----|
| General Information..... | 5 |
| Lab schedule..... | 5 |
| Lab groups | 5 |
| Lab marking | 5 |
| Due dates and on-time delivery | 6 |
| Lab-0: Introduction to the ARM platforms in the lab..... | 7 |
| Objective | 7 |
| What you do | 7 |
| Pre-lab | 7 |
| Introduction to hardware and software..... | 7 |
| Hardware..... | 7 |
| Software | 8 |
| In-lab procedure..... | 8 |
| Running assembly language code on the MCU | 9 |
| Using the simulator | 10 |
| Debugging assembly language code | 11 |
| Lab report..... | 12 |
| The assembly language code..... | 14 |
| Lab-1: Flashing LED..... | 16 |
| Objective | 16 |
| Background..... | 16 |
| Pre-lab | 17 |
| In-lab procedure..... | 17 |
| Using Flash versus RAM memory | 18 |
| Switching from Flash memory to RAM..... | 19 |
| Coding Goals..... | 22 |
| Lab report | 22 |
| The assembly language code..... | 23 |
| Lab-1 Submission form | 24 |
| Lab-2: Subroutines and parameter passing..... | 25 |
| Objective | 25 |

| | |
|---|----|
| What you do | 25 |
| Pre-lab | 25 |
| In-lab procedure | 25 |
| Lookup Tables | 27 |
| Lab report | 28 |
| The Morse Code | 29 |
| Lab-2 Submission form | 30 |
| Lab-3: Input/Output interfacing | 31 |
| Objective | 31 |
| What you do | 31 |
| Background | 31 |
| Pre-lab | 32 |
| In-lab procedure | 32 |
| Optional Improvements | 33 |
| Lab report | 33 |
| Extra Information | 34 |
| Lab-3 Submission form | 35 |
| Lab-4: Interrupt handling | 36 |
| Objective | 36 |
| Pre-lab | 36 |
| What you do | 36 |
| In-lab procedure | 36 |
| Lab report | 37 |
| Lab-4 Submission form | 38 |
| Appendix A: The LPC1768 microprocessor | 40 |
| Appendix B: Abbreviated Instruction set summary | 43 |
| Appendix C: Memory map | 47 |
| Appendix D: Input / Output ports | 49 |
| Appendix E: Exception and Interrupts | 55 |
| Appendix F: Schematic diagram [4] | 56 |
| References: | 62 |
| Appendix G: Hand Assembly | 63 |

| | |
|--|----|
| University Expectations and Policies | 65 |
| Academic Integrity | 65 |
| Grievance..... | 65 |
| Discipline | 65 |
| Appeals | 65 |
| Note for Students with Disabilities..... | 65 |

General Information

This section contains general information about the ECE-222 lab. All lab contents and resource are posted on the University of Waterloo on-line learning system called LEARN. It is a password protected environment and can be accessed here: learn.uwaterloo.ca

In the Fall 2012 term we shifted from the Freescale ColdFire® to the ARM® CPU for ECE-222. The Coldfire CPU was exceptionally good for teaching the basics but many devices today use an ARM based CPU. Please report typos, errors, or other challenges in this manual the Lab Instructor. We appreciate your feedback and cooperation.

We also support the Texas Instruments Tiva-C microcontrollers. At \$13 they are a cheap development platform with superior power and instruction performance to Arduino. We currently use the Keil MDK development software but it runs under Windows only. TI Code Composer Studio and Energia work under Linux, Mac or Windows but are not supported for ECE 222. There are many other ARM development kits available – however support for assembly language programming is often lacking.

Lab schedule

The three hours lab sessions scheduled for this course are listed here: Please see the Lab Schedule posted on LEARN

Lab groups

All labs are to be done in groups of two students. Groups of three, or more, students are prohibited. If a lab session contains an odd number of students, every effort will be made to pair-up the single student with another student from other sessions, if students' schedules allow.

It is expected that both members will put equal effort into the lab tasks during the term. Unequal participation or other conflicts in a group should be brought to the lab instructor's attention at the earliest possible time.

Lab marking

The course marking scheme is stated in the Course Outline.

There are three main components related to each lab session. The lab manual for each experiment will tell you what you will need to submit for that component.

- **Prelab.** It is designed to get you started with the task. Once you accomplish what is asked in this section, you will be ready to start coding in assembly language.
- **Lab session/Demo.** You will present your work to a lab staff to be marked for that section. Some questions will be posed to students regarding the contents, procedures, debugging, and techniques used to get the code working correctly.
- **Lab report.** You will submit a report which often contains your assembly language code, and a TA will mark your report.

Different labs carry different marks allocated to them. Marking sheets are in lab manual.

Warning: Failure to complete ALL labs may result in an Incomplete mark for the course. This means each student is expected to attend all lab sessions.

Due dates and on-time delivery

Lab reports and lab demonstration sessions will carry some marks associated with each experiment. They should be treated like examination sessions. Students should not miss them without a legitimate reason, otherwise they will lose some marks.

If you have an interview scheduled during a lab demonstration session, or if you have to miss a lab demonstration session because of another legitimate reason, please inform the Lab Instructor to avoid being recorded as 'Absent'. They will try to assign you to another session for that particular lab.

Details about deadlines and penalties are included in the Course Outline.

Electronic lab report submission is done through LEARN (learn.uwaterloo.ca).

Lab-0: Introduction to the ARM platforms in the lab

Objective

We will familiarize ourselves with the basics of the ARM boards used in the ECE-222 lab. Here is a short list of what we will do in this session:

- Introduction to ARM board
- Introduction to the Keil μ Vision4 software
 - o How to create or open a project
 - o How to build, or assemble, a target
 - o How to download object code into memory on the target board
 - o How to debug code
 - o How to use the simulator

What you do

In this lab you will load, assemble, download, and run some short programs. Each program performs a specific task. For example, one program loads some values into some registers and then adds them up. You will confirm the result by checking the contents of the registers in debug mode.

Pre-lab

N/A

Introduction to hardware and software

In order to get students familiarized with the tools used in the ECE-222 lab, let us take a closer look at the hardware and software used in the lab. More details can be found in Appendix A.

Hardware

Figure 1.1 shows the MCB1700 board. The board employs a LPC1768, a micro-controller unit (MCU) made by NXP (affiliated with Philips). There are several input/output peripheral devices available on the board.

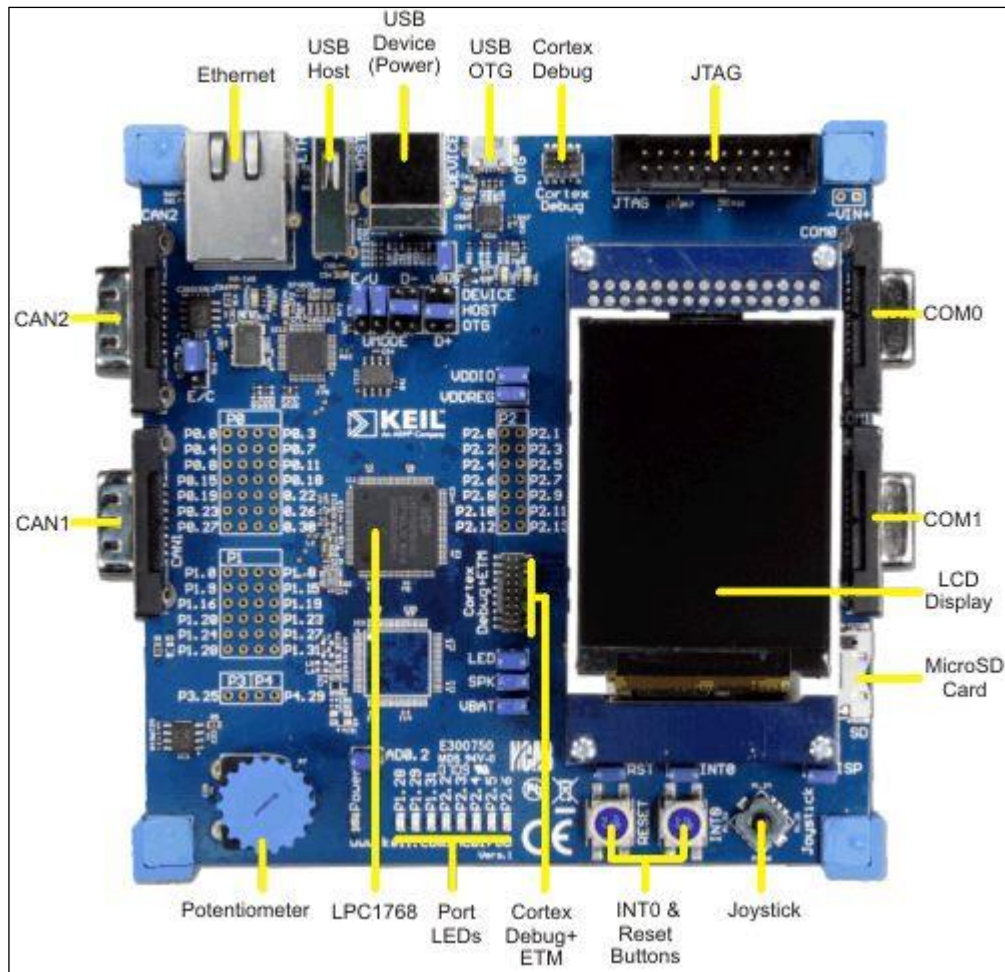


Figure 1.1 – The MCB1700 board [2]

The heart of the board is the LPC1768 MCU (microcontroller unit), which contains a CPU, on-chip flash memory, RAM (Random Access Memory) and some peripheral blocks.

Software

The software toolchain used to program the MCB1700 board is μ Vision[®] developed by the [Keil](#), the manufacturer of the board.

The Keil μ Vision[®] toolchain has been designed for high-level programming languages such as C++. The manufacturer clearly states that it is not meant to work for pure assembly language! Regardless, the board can be used to develop assembly language programs.

In-lab procedure

First, we will build and run code on the MCU. Then we will review how to debug the code.

Running assembly language code on the MCU

Follow the following steps in order to get yourself familiarized with the μ Vision4 toolchain.

- 1 Run the software by clicking on **Start/All Programs/Keil μ Vision4**
- 2 Click on the **Project** tab, and choose **New μ Vision Project**
- 3 Select or create a subdirectory on N: drive (like N:/ECE_222/Lab_0), then assign a name to your project (ie Lab0 ... it can be different then the folder name), then click on **Save.. DO NOT MAKE A DIRECTORY, FILE OR PROJECT NAME WITH A SPACE IN IT!** A space will prevent simulation from working properly.
- 4 To select a CPU, double click on **NXP (founded by Philips)** and select **LPC1768**. Click **OK**
- 5 Click **NO** when prompted to copy '**startup_LPC17xx.s to Project Folder**' as it is for C programmers rather than assembly programmers.
- 6 This step is done outside of the μ Vision4 software. Copy the provided **startup_LPC17xx.s** file and the sample program **Lab0_program.s** from Learn to the folder used for this lab - like N:/ECE_222/Lab_0. **DO NOT USE My Documents!**
- 7 Switch back to the μ Vision4 screen, which should now resemble Figure 1.2.
- 8 Right click on the **Source Group 1** under **Target 1**. Select **Add Files to Group 'Source Group 1' ...**. Select **All Files** from '**Files of type**' drop-down menu, which will list all files in the folder. Select **startup_LPC17xx.s**, click **Add**, then select the file **Lab0_program.s**, click **Add** then click **Close**.
- 9 Next, click on **Target 1** so that it is highlighted (if it is not already) and then click again on it, waiting a couple of seconds to edit the text. Type in **LPC1768_FLASH** to rename the target. The name is appropriate because our program will be written to the Flash memory of the device as opposed to the RAM.
- 10 Next, double-click on Target 1 (or click once on the + symbol) to show the files under the target. Your screen should now match Figure 1.3
- 11 Now you are ready to assemble your code. This is called 'Build target' in the μ Vision software. Click on **Project** tab and then on **Build Target** or hit **F7**. The target, or binary code, written to the LPC1768, should assemble with no errors and only one warning.

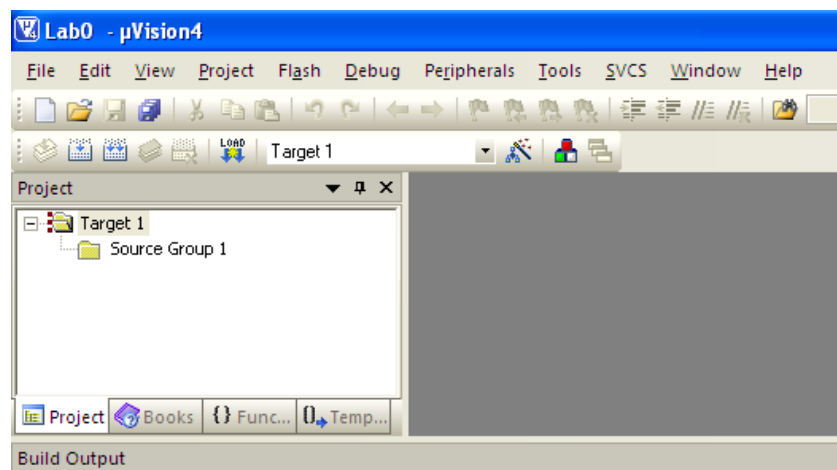


Figure 1.2 – The μ Vision4 environment [5]

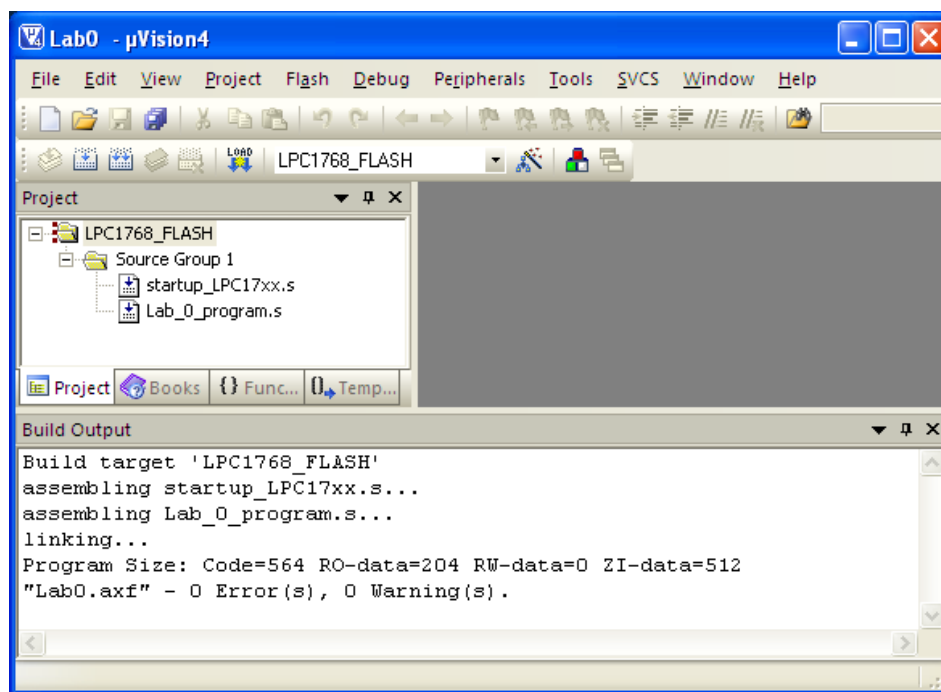


Figure 1.3 – Building the target [5]

- 12 The next step is to download the program into the LPC1768. Click on the **Flash** drop-down menu and select **Download**. To run the code, press the **Reset** button on the board.
 - a. If, when downloading the program to the device, you receive a **SWD Communication Failure** message push the reset button on the board and try again. Lastly be sure both USB cables are connected to the PC. Also, you can reset the USB Hub.
 - b. To eliminate the need to press the Reset button after every download, right click on your target, **LPC1768_FLASH**, and select **Options for Target 'LPC1768_FLASH'** and then select the **Utilities** tab. **Use Target Driver for Flash Programing** should be selected. Click on **Settings** and click on the **Flash Download** tab, and then select the checkbox for **Reset and Run** under the **Download Function** and then click **OK**.

Using the simulator

The μ Vision software comes with a powerful Simulator and it can be used to test code when you do not have access to an ARM board. In ECE 222 we only debug via the ARM board. Here is how to switch between debugging on a physical board and the simulator:

- 1 Make sure that you are not in the Debug mode. If in Debug mode, simply exit from it by clicking on the Debug button.

- 2 Right-click on the **LPC1768_FLASH** and choose the **Options for Target 'LPC1768_FLASH'** and then click on the **Debug** tab. You should see Figure 1.4
- 3 You have the option to choose between the Simulator or the MCB1700 board. If you click on **Use Simulator** on the left pane, then you are no longer using the actual board. But if you choose **ULINK2/ME CORTEX DEBUGGER** on the right pane, you will need the MCB1700 board connected to the computer you are working on. Click on the ULINK2/ME Cortex Debugger radio button on the right side for the Debugger.

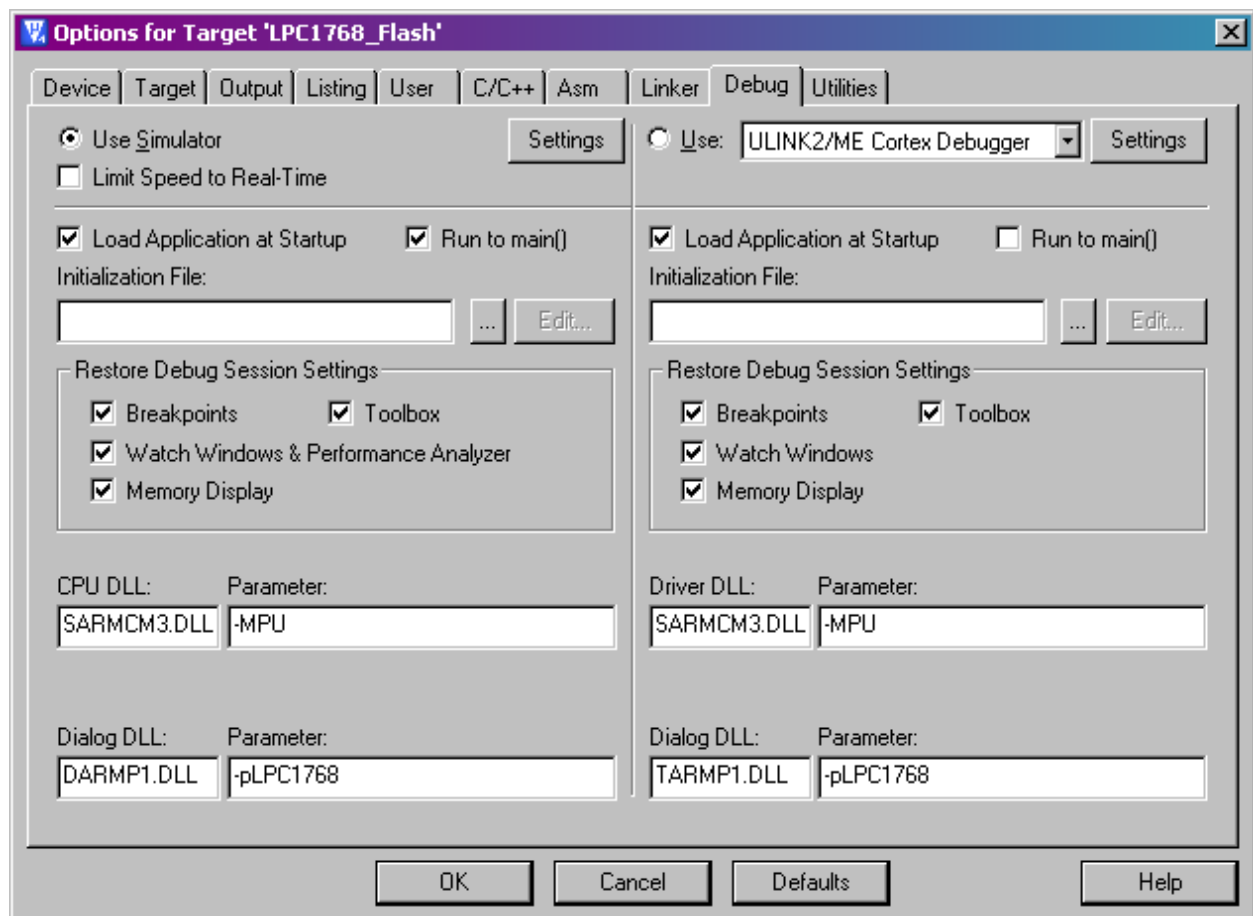


Figure 1.4 – Simulator versus MCB1700 board [5]

Debugging assembly language code

As you may have noticed, there is no visual difference on the board when the code is running. So, how can we make sure that the code is generating the correct results? This is done by running the code step-by-step and checking the content of registers.

This is called Debug mode. It is a very powerful and useful mode when you want to find a bug in your code. Your code must generate no errors when assembled before you activate the Debug mode.

The Debug mode can be used both with the Simulator or the board itself. When debugging using the board, every instruction will be executed on the MCB1700, and the results are communicated over the 'ULINK2/ME Cortex Debugger'. Be sure that your workstation is physically connected to the board via the USB cables, otherwise communication will not be possible.

If the Simulator is chosen, then the board is not used at all during the debug mode.

Follow these instructions in order to step through (debug) your code:

- 1 Make sure you are using the board and not the simulator for the following steps. (see section **Using the Simulator**)
- 2 Choose **Start/Stop Debug Session** from the **Debug** drop-down menu.
- 3 Click **OK** when presented with the message about being in "Evaluation Mode." Your screen should now resemble Figure 1.5
- 4 Make note of the following important buttons in the graphical user interface (GUI):



From left to right: **Reset, Run, Stop, Step, Step Over, Step Out, Run to Cursor Line, Show Next Statement, Command Window, Disassembly Window, Symbol Window, Registers Window, Call Stack Window, Watch Windows, Memory Windows, Serial Windows, Analysis Windows, Trace Windows, System Viewer Windows, Toolbox, Debug Restore Views**

- 5 Click on the **Reset** button on the GUI. The arrow should point to the line **LDR R0, =__MAIN ONLY** if using the Simulator – but not when using the hardware debugger with flash memory.
- 6 Set a breakpoint on the first line of code "**MOV R0, #0x5678**" by selecting the line and hitting **F9** or by right clicking on the line. The red circle on the left indicates that a breakpoint is set. Then click on the **Run** button, or use **F5**, to run the program to the breakpoint.
- 7 Click on the **Step** button or **F11**. The yellow arrow moves down by one line. This means that the first line of code was run and you are now about to run the next line.
- 8 Click on the **Step** button, or push **F11** button on keyboard, several times until you reach the last line of code "**loop B loop**" before the **END**. In each step look at the register values to make sure that the program is working properly

Lab report

Although there is no deliverable assigned to this lab, attendance is mandatory. You must complete Lab 0 before starting Lab 1.

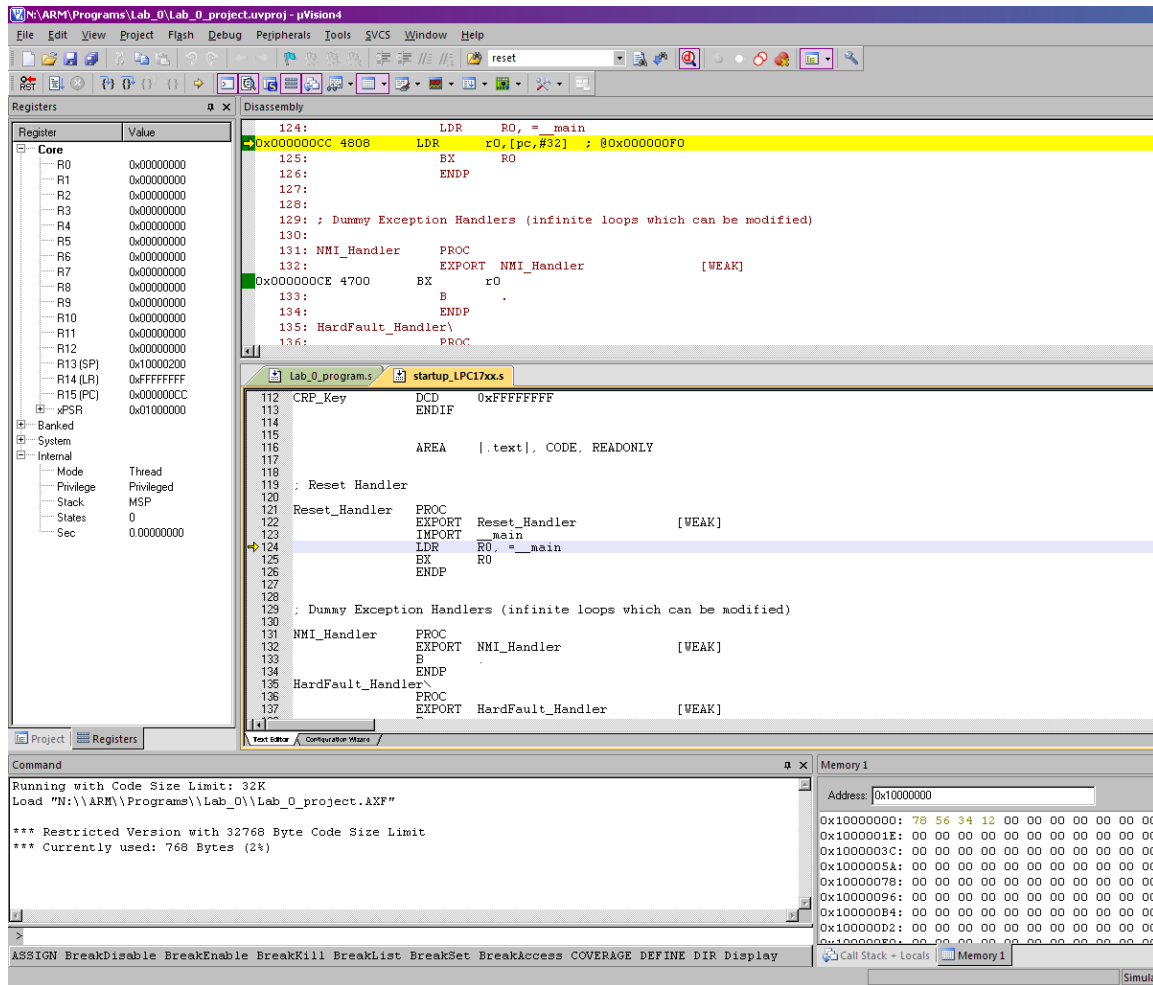


Figure 1.5 – Debug session in µVision software [5]

The assembly language code

```
;*-----
;* Name:  Lab_0_program.s
;* Purpose: Teaching students how to work with the uVision software
;* Author:  Rasoul Keshavarzi
;*-----

                THUMB                ; Thumb instruction set
                AREA      My_code, CODE, READONLY
                EXPORT    __MAIN
                ENTRY
__MAIN          ; This name must not be changed. It matches with the startup_LPC17xx.s file.

; Store 0x1234,5678 into memory address 0x1000,0000 in order to see how the little-endian
; approach writes data into memory
                MOV        R0, #0x5678 ; Load the lower half of R0 and clear the upper half
                MOVT       R0, #0x1234 ; Load the upper half of R0
                MOV        R1, #0x0    ; Load the lower half of R1 with zeros
                MOVT       R1, #0x1000 ; 0x1000,0000 is now stored in R1
                STR        R0, [R1]    ; Store register R0 in the address pointed to by R1
(0x1000,0000)

; Look at memory address 0x1000,0000 after this runs
; Importing values to registers
                MOV        R0, #0x123  ; Loading 123 into R0
                MOV        R1, #0x456  ; Loading 456 into R1
                MOV        R2, #0x789  ; Loading 789 into R2
                MOV        R3, #0xABC  ; Loading ABC into R3
                MOV        R4, #0xDEF  ; Loading DEF into R4
                MOV        R5, #0x0    ; Loading R5 with zeros

; Swapping the values in R0 and R1 (R5 is used as temporary buffer)
                MOV        R5, R0      ; R5 <-- R0 (content of R0 is stored in R5)
                MOV        R0, R1      ; R0 <-- R1 (content of R1 is stored in R0)
                MOV        R1, R5      ; R1 <-- R5 (content of R5 is stored in R1)

; Adding five values together R5 <-- R0+R1+R2+R3+R4
                ADD        R5, R0, R1  ; R5 <-- R0 + R1
                ADD        R5, R2      ; R5 <-- R5 + R2
                ADD        R5, R3      ; R5 <-- R5 + R3
                ADD        R5, R4      ; R5 <-- R5 + R4

LOOP           B          LOOP        ; Branch back to this line – an infinite loop

                END
```

This Page Left Intentionally Blank

Lab-1: Flashing LED

Objective

The objective of this lab is to complete, assemble and download a simple assembly language program. Here is a short list of what you will do in this session:

- Write some THUMB assembly language instructions
- Use different memory addressing modes
- Test and debug the code on the Keil board
- The on-board RAM is used instead of Flash memory

You will flash an LED (Light Emitting Diode) at an approximate 1 Hz frequency.

Background

The LPC1768 belongs to the Cortex-M3 family of microprocessors which uses the THUMB instruction set. Thumb is a subset of the ARM instruction set.

Conditional instructions are possible via the PSR (Program Status Register). The register can be viewed by expanding xPSR in the Register Window of Keil MDK. There is only one bit, Z, which will be used in this course. If the result of an operation (memory read, test, compare, math, logic), which sets the status bits, is zero this bit will be set to 1. Appendix B details which status bits can be set by which instruction. The bits, https://en.wikipedia.org/wiki/Status_register in brief are:

Z – Zero – was the result zero

N – Negative – the highest bit which may indicate the sign of the number

C – Carry – was a carry (overflow) generated by an operation

V – Overflow (only used for signed math)

Code can be conditionally executed by using an instruction which updates the Z flag (ie compare to 0, ADDS, MOVS) and then branching (see Appendix B) based upon the result of the test. BNE and BEQ are the only branches you will use. BNE will branch if the Z flag is 0 – if Z indicates that the instruction updating the Z flag was Not Equal to zero. BEQ will branch if Z is set or if the instruction set the Z flag and indicates that Equal to zero.

In order to flash an LED, one needs to know how the LPC1768 microprocessor is connected to the LEDs – the pin configuration and interfacing. A lot of the hardware interfacing details are covered in Lab-3. The details to accomplish this lab are:

- Writing 0xB0000000 into memory address 0x2009C020 turns “off” the three LEDs on port 1 (pins P1.28, P1.29, and P1.31)
- Writing 0x0000007C into memory address 0x2009C040 turns “off” the five LEDs on port 2 (pins P2.2 to P2.6)
- Toggling bit 28 of the address 0x2009C020 will cause the corresponding LED (P1.28) to alternate between “on” and “off”. The memory address is 32 bits wide (bit 31 down to bit 0). You should switch between 0xB0000000 and 0xA0000000 to flash the LED on pin P1.28.

Pre-lab

Before the lab session, look at the THUMB instruction set in Appendix B. The LPC1768 is a Cortex-M3 ARM CPU using the THUMB instruction set.

In order to see a flashing LED, implement a delay between the LED “on” and “off” states. Think about implementing a delay in assembly language.

Hint: Increment or decrement a register in a loop until it reaches a certain value.

There is no deliverable as pre-lab for this lab.

In-lab procedure

Complete the given code that is given at the end of Lab-1 manual. Feel free to change any part of the code if you wish so.

Please note that the following line in the given program is for the short flowchart where the LED is toggled as opposed to turning on and off.

```
STR        R3, [R4, R2]        ; write data to toggle bit 28 of port 1
```

Try to make a connection between the given code and the flowcharts, and then complete the given code. All you need to do is adding four or five lines of code that constructs the main loop that causes the LED to flash.

- Create a new folder (like N:\ECE222\Lab1) and project as was done in Lab-0
- Start by turning off all eight LEDs (three on port 1 and five on port 2)

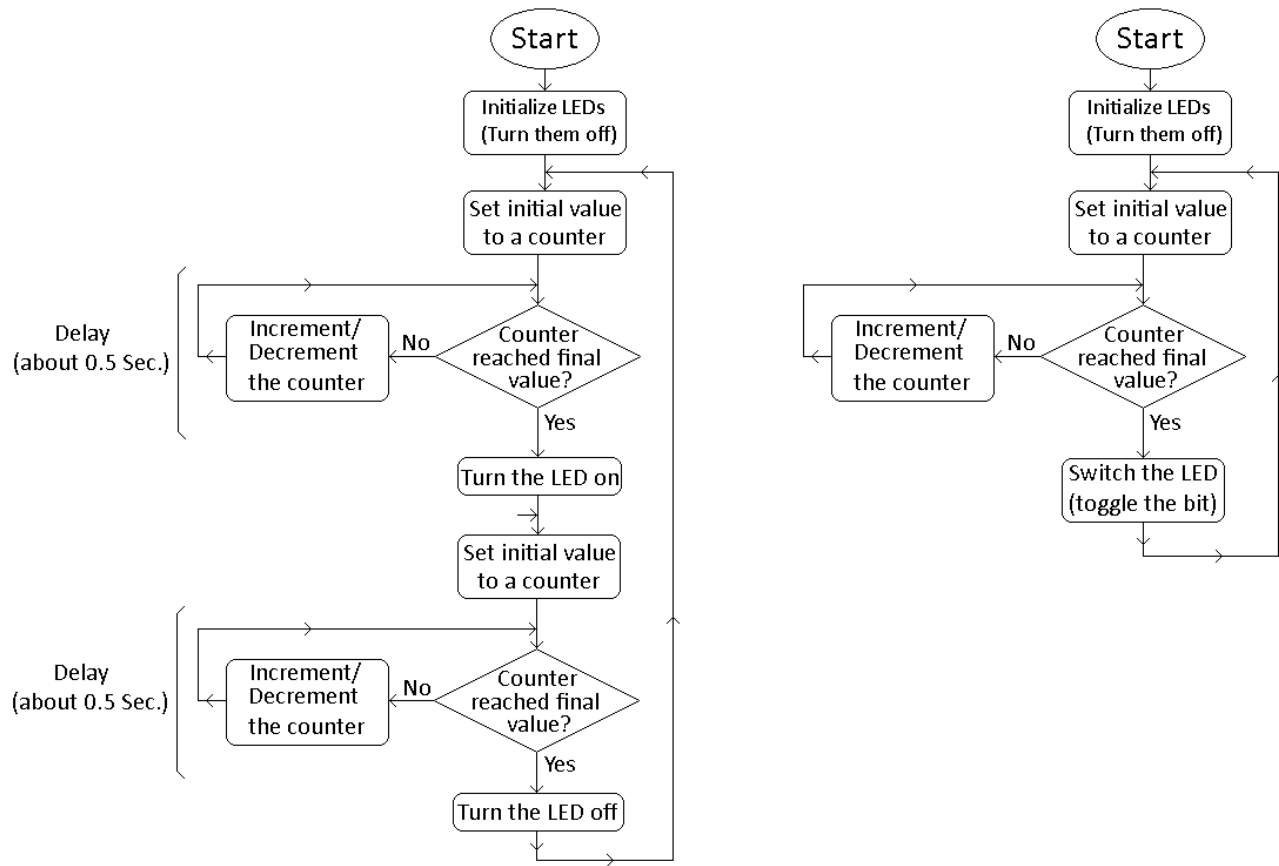


Figure 1.1 – Flowcharts for flashing LED

- Then implement the flashing LED code using an infinite loop which toggles bit 28 of the address 0x2009C020. Figure 1.1 shows the two different approaches. The shorter flowchart leads to smaller code size and is more efficient. Your code, when demonstrated for marking, must be using the short flowchart. **But it is strongly recommended that you implement the longer flowchart as the first step.** Once you get the longer flowchart working, changing your code to implement the short flowchart will be an easy step.
- Don't forget to insert a 500ms delay in the loop; otherwise the LED blinks too fast to see.
- Assemble the code, download it to the board, and debug it if necessary.
- **You must use onboard RAM as opposed to flash memory.**

Using Flash versus RAM memory

Flash memory is non-volatile, meaning that it retains its contents without power. SRAM (Static Random Access Memory) is volatile and the contents written to a RAM are lost as soon as power to the RAM is lost.

Flash memory has a limitation in the number of times that the contents can be overwritten. After a few thousands 'Write' operations the Flash memory will fail.

In Lab-0 you used the Flash memory every time you downloaded your code to the microprocessor. In order to preserve the lifetime of our boards, we will work with RAM instead of Flash memory from now on.

Please note that working with flash memory for code development is not an option. All students MUST use RAM for code development and we will check this during the demonstrations!

Switching from Flash memory to RAM

For using the Simulator, you don't need to change anything. Just continue to work the same way you worked for Lab-0. In other words, you will work with the LPC1768_FLASH target whenever you are using the Simulator.

But when it comes to working with the hardware (not the Simulator), perform the following steps to switch to RAM memory:

- When in development, not Debug, mode click on the **Project** menu, then to **Manage**, then to **Components, Environments, Books ...** submenu.
- Under the **Project Components** tab, click on **New (Insert)** icon for **Project Targets**. Figure 1.2 shows a captured screen.
- Give a meaningful name (like LPC1768_RAM) to the new Target and click on **OK**.

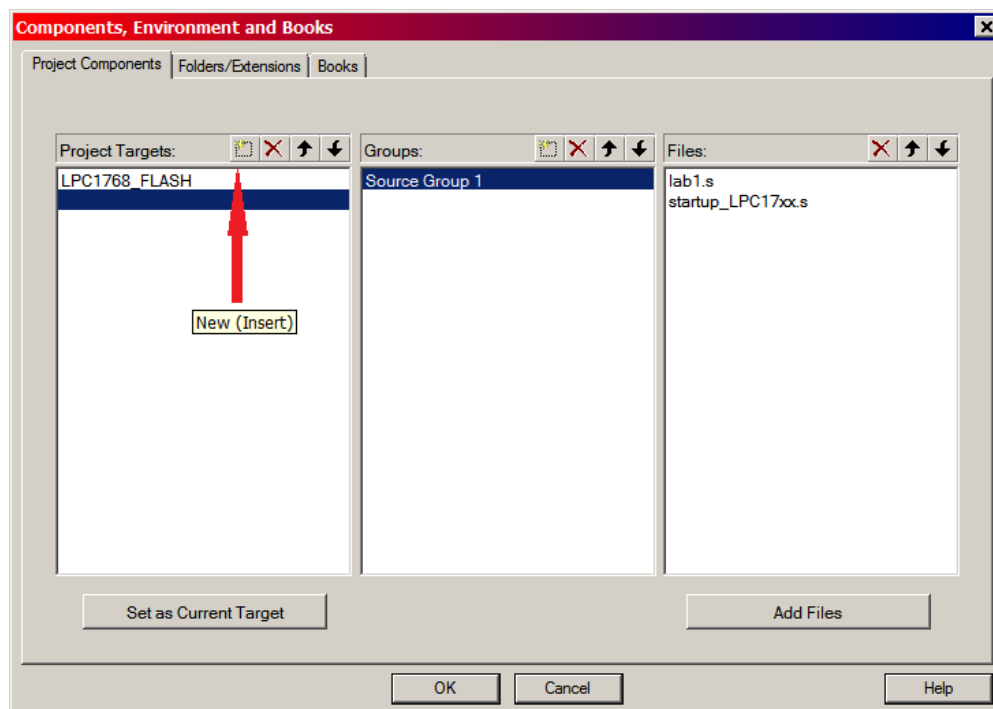


Figure 1.2 – Creating new targets

A warning message about not being able to write the file C:\Software\Keil\tools.ini is normal.

The new target has now been created. The following steps show some parameter adjustments for the new Target.

- Select the **LPC1768_RAM** target by clicking on the pull-down menu located roughly under the **Debug** menu button.

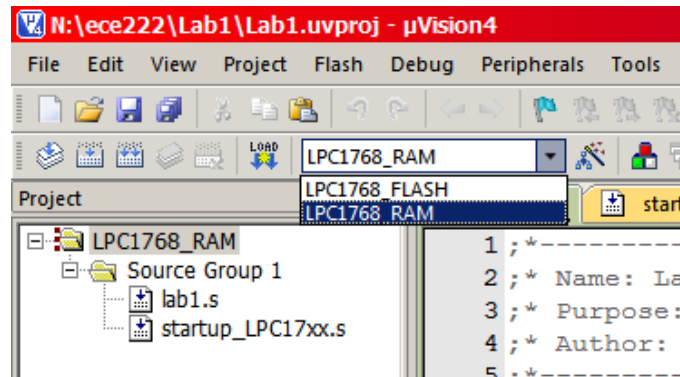



Figure 1.3 – Choosing different Targets

- Go to **Options for Target LPC1768_RAM** (by right click on the Target)
- Under the **Target** tab, adjust the IROM1 and IRAM1 addresses according to figure 1.4.
- Under the **Debug** tab, add the file name Dbg_RAM.ini to the field **Initialization File** (Make sure you have chosen the **Ulink2/ME Cortex Debugger**, not the Simulator). Figure 1.5 shows a screen capture.
- Copy the file Dbg_RAM.ini from LEARN into your Lab-1 subdirectory.

Running a program in RAM

You already know how to download and run a program into Flash memory. It is done by clicking on

the  button, or by clicking on **Flash** menu and then on **Download** button. **This method will NOT work for the RAM target!**

You must go to the Debug mode in order to get the program loaded into RAM. In this case, once you run the program, it is run from RAM. You can confirm it by looking at the addresses.

RAM in LPC1768 starts at the address 0x1000 0000 and flash memory starts at 0x0000 0000.

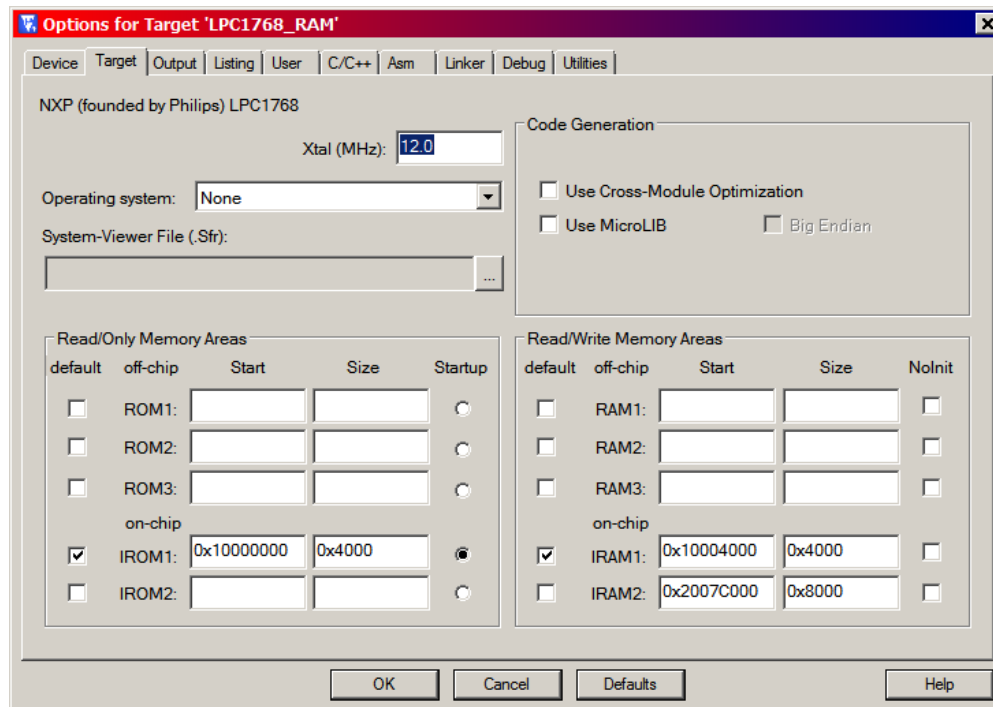


Figure 1.4 – Address settings for LPC1768_RAM target

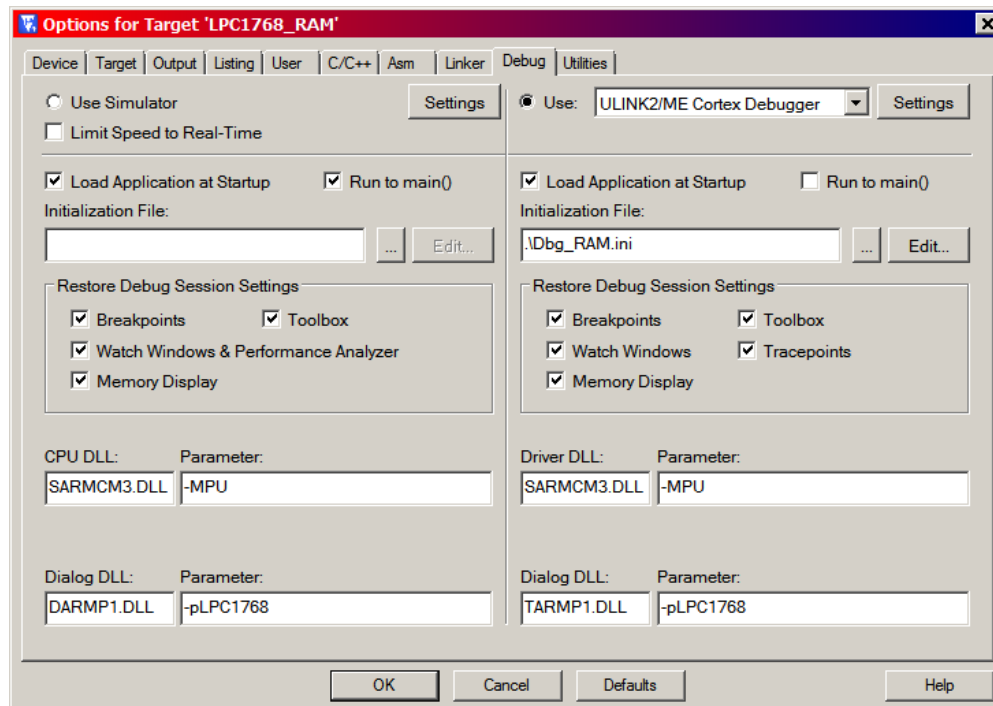


Figure 1.5 – Initialization file for LPC1768_RAM target

Coding Goals

The goal is to get the LED flashing at a frequency close to 1 Hz (on for 500ms and off for 500ms). An accurate period of 1Hz is NOT the primary goal of this lab; time it or count the number of flashes in 60 seconds to scale the number to get better than 10% accuracy.

All code should be well commented. The number of lines of code added to the given assembly language program does NOT affect your mark. ALL documentation should be within the program.

Lab report

Hand assemble, using Appendix G, the instruction below (if you want a challenge select another supported instruction) using the tables in Appendix G. Note that only 8-bit MOV instructions are supported. NOTE: Appendix G is for an OBSOLETE ARM processor and the instructions are not compatible with the ARM processor we are using.

ADD R4, R4, R2.

The assembly instruction should appear as comments at the end of your code.

Submit your commented, well-written code, and a picture of the marked Submission form, to the Lab-1 drop-box LEARN. To understand the deliverables look at the Lab1 Submission form.

The assembly language code

```

;-----
; Name:      Lab_1_program.s
; Purpose:   This code flashes one LED at approximately 1 Hz frequency
; Author:    Rasoul Keshavarzi
;-----

                THUMB                ; Declare THUMB instruction set
                AREA      My_code, CODE, READONLY ;
                EXPORT    __MAIN      ; Label __MAIN is used externally q
                ENTRY

__MAIN
; The following operations can be done in simpler methods. They are done in this
; way to practice different memory addressing methods.
; MOV #0xWXYZ writes the value 0xWXYZ into the lower word (16 bits) and clears the upper word
; MOVT #0xPQRS writes the value 0xPQRS into the upper word
; BNE and BEQ can be used to branch on the last operation being Not Equal or Equal to zero

                MOV        R2, #0xC000      ; Move 0xC000 into R2
                MOV        R4, #0x0         ; Initialize R4 register to 0 to build address
                MOVT       R4, #0x2009      ; Assign 0x2009 to higher part of R4
                ADD        R4, R4, R2       ; Add 0xC000 to R4 to get 0x2009C000

                MOV        R3, #0x0000007C  ; Move initial value for port P2 into R3
                STR        R3, [R4, #0x40]   ; Turns off five LEDs on port 2

                MOV        R3, #0xB0000000  ; Move initial value for port P1 into R3
                STR        R3, [R4, #0x20]   ; Turns off three LEDs on Port 1

                MOV        R2, #0x20        ; Put Port 1 offset into register R2 for later

                MOV        R0, #0xFFFF     ; Initialize R0 for countdown

Loop
                SUBS       R0, #1           ; Decrement R0 and set the N,Z,C status bits
                ;
                ;      Approximately five lines of code are missing here.
                ;      Add the instructions to get the LED flashing.
                ;
                STR        R3, [R4, R2]     ; write R3 to port 1, YOU NEED to toggle bit 28 first

                B          Loop            ; Keep the LED flashing forever

                END

```

Lab-1 Submission form

| | | | | |
|-------------------------------------|------------------------------|------------------------------|------------------------------|------------|
| Class: 001 <input type="checkbox"/> | 201 <input type="checkbox"/> | 202 <input type="checkbox"/> | 203 <input type="checkbox"/> | Demo date: |
| 002 <input type="checkbox"/> | 204 <input type="checkbox"/> | 205 <input type="checkbox"/> | 206 <input type="checkbox"/> | |

Submission Statement: We (I) are (am) submitting this report for grading in ECE 222. We (I) certify that this report (including any code, descriptions, flowcharts, etc., that are part of the submission) were written by us (me) and have received no prior academic credit at this university or any other institution. **The penalty for copying or plagiarism will be a grade of zero (0).**

| Member 1 | Member 2 |
|-------------------------------|-------------------------------|
| Name: | Name: |
| UW-ID (NOT student #) | UW-ID (NOT student #) |
| Signature: | Signature: |

Note: Reports submitted without a signed submission statement will receive a grade of zero (0).

| | | Weight | Grade | Comment |
|------------------------|-------------------------------------|------------|-------|---------|
| Part-I | Pre-lab | 0 | -- | |
| Part-II Lab-demo | Lab completion (short flowchart) | 35 | | |
| | Questions | 35 | | |
| Part-III Lab report | Hand Assembly | 10 | | |
| | Code quality | 10 | | |
| | Code comments | 10 | | |
| Total | | 100 | | |

Lab-2: Subroutines and parameter passing

Objective

In structured programming, big tasks are broken into small routines. A short program is written for each routine. The main program calls these short subroutines.

In most cases when a subroutine is called, some information and parameters must be communicated between the main program and the subroutine. This is called parameter passing.

In this lab, you will use subroutines and parameter passing by implementing a Morse code system.

What you do

In this lab you will turn one LED into a Morse code transmitter. You will cause one LED to blink in Morse code for a five character word. The LED must be turned on and off with specified time delays until all characters are communicated.

Pre-lab

Think about implementing Lab-1 code using subroutines. Write a subroutine called LED_OFF that turns LED P1.28 off, and another subroutine called LED_ON that turns the LED on. Write a third subroutine called DELAY that takes one input parameter (register R0) and waits for $R0 * 500\text{ms}$ (roughly) before returning.

There is no deliverable as Pre-lab for this experiment.

In-lab procedure

Use the code you developed in lab-1 as a start point for this lab. A template code for lab-2 is available on LEARN. Start by initializing the on-board LEDs to off. Then additional functionalities are added to the code as shown in the flowchart depicted in figure 2.1. This is described in the following steps:

- Turn all LEDs off (go to lab-1 for memory addresses and contents to write)
- Put the initials of the two lab partners together to create a four character word (**Capital letters** only). Add a fifth character of your choice (capital) which is different from the four previous ones. Set the five characters in your program at InputLUT label.
- Write a subroutine called LED_OFF that turns the LED connected to pin P.1.28 “off”

- Write a subroutine called LED_ON that turns the LED connected to pin P1.28 “on”
- Write a subroutine called DELAY that causes an $R0 * 500\text{ms}$ delay before returning to main program. R0 is passed to subroutine from the main program.
- Write a subroutine called CHAR2MORSE that converts an ASCII code into a Morse pattern. You will use registers for parameter passing between subroutines and the main program.

To deal with each of the characters in the InputLUT string:

- Fetch a character by reading the string memory (from the first to last). It is in ASCII format as shown in table 2.1
- Subtract 0x41 from the ASCII value to get the index for the Morse LUT (look up table)
- Read the Morse pattern from the Morse LUT (using the index)
- Blink the LED for the Morse pattern. This step can be broken into the following sub-steps:
 - a) Move the Morse code pattern for a character in register R0
 - b) Shift left the pattern in R0 until a 1 falls into the C status bit
 - c) If C is ‘1’, or set, then call the LED_ON subroutine
 - d) If C is ‘0’, or clear, then call the LED_OFF subroutine
 - e) Call the DELAY subroutine with a delay of 1
 - f) If R0 is 0 the pattern has been all displayed
 - g) Otherwise go to step b
- Insert long delay (equivalent to three dots) before fetching the next character
- Repeat the above steps for all characters. If the whole string has been processed, then insert another four DELAY intervals and start from the beginning again as a new word

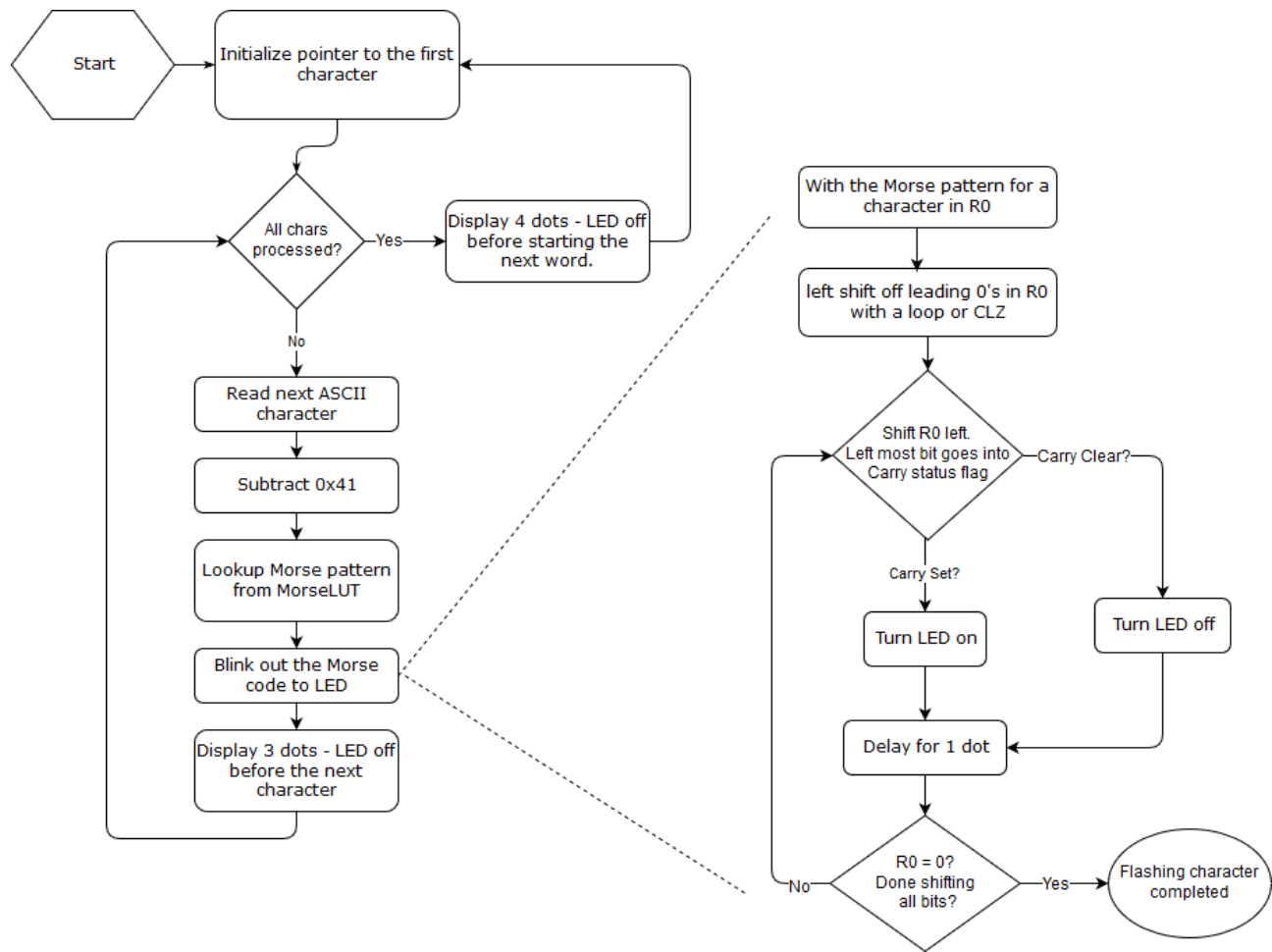


Figure 2.1 – Flowchart for the Morse code transmission using an LED

Hint: The shift operations require an S on the end in order to set the status bits: LSRS, LSLs. Don't forget that logical operations can be conditional upon the C status flag using CC (Carry Clear) or CS (Carry Set). Conditional math and logic instructions (ORREQ, ANDCS, ...) allow one to often write code without conditional branches and this will result in code that is faster and smaller.

Lookup Tables

Lookup tables are used to provide data to a program. You should be careful about how you read the data and index into it. Data can be 8-bit (byte), 16-bit (half-word) or 32-bit (word). Data has been stored using DCB (Define Constant Byte) or DCW (Define Constant Word – where word means 16-bits).

Here is some simple code to read a character:

```

LDR R0, =InputLUT ; R0 points to the start of the string
; LDRB zero-extends (fills the top of) R2 with 0's!
LDRB R2, [R0, R1] ; Read a character and put it into R2
; R1 is an offset

```

Here is some code to read from an array of 16-bit data:

```

LDR R3, =Morse_LUT
; LDRH zero extends so the the top of R4 will be 0's
LDRH R4, [R3, R5] ; Reading the Morse pattern
; R5 is an offset

```

Lots more code here

```

InputLUT
DCB "BIRDS", 0 ; This is a five character word
; to be sent on the LED using Morse

```

```

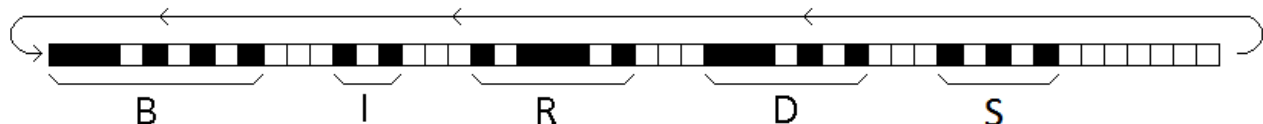
Morse_LUT
DCW 0x17, 0x1D5, 0x75D, 0x75 ; A, B, C, D
DCW 0x1, 0x15D, 0x1DD, 0x55 ; E, F, G, H
DCW ... ; Use the template code on
; LEARN for complete list

```

ALIGN

In order to work with 16-bit data the LUT for Morse code has to be type DCW, you have to increment by 2 (bytes) to move thru the data and to read the table you need to use LDRH. For Bytes use DCB and LDRB.

Example: Suppose the lab partner's initials plus a fifth letter compose the word BIRDS. Then the program should extract the letters (B I R D S) and create a Morse code pattern like this:



Please note that all five letters are considered to be one word.

Lab report

Submit your commented, well-written code, and a picture of the marked Submission form, to the Lab-2 drop-box LEARN. Examine the Lab-2 Submission form to see what you will need to deliver.

The Morse Code

Table 2.1 shows corresponding Morse codes for the English language alphabet.

Note the “Morse code” is left justified and the “Binary Morse code value” is right justified.

Table 2.1 – The Morse code

| Letter | ASCII value | Morse code | Morse code value | | |
|--------|-------------|---|---------------------|---------|---------|
| | | | Binary | Decimal | Hex |
| A | 0x41 | | 0000,0000,0001,0111 | 23 | 0x17 |
| B | 0x42 | | 0000,0001,1101,0101 | 469 | 0x1D5 |
| C | 0x43 | | 0000,0111,0101,1101 | 1885 | 0x75D |
| D | 0x44 | | 0000,0000,0111,0101 | 117 | 0x75 |
| E | 0x45 | | 0000,0000,0000,0001 | 1 | 0x 1 |
| F | 0x46 | | 0000,0001,0101,1101 | 349 | 0x 15D |
| G | 0x47 | | 0000,0001,1101,1101 | 477 | 0x 1DD |
| H | 0x48 | | 0000,0000,0101,0101 | 85 | 0x 55 |
| I | 0x49 | | 0000,0000,0000,0101 | 5 | 0x 5 |
| J | 0x4A | | 0001,0111,0111,0111 | 6007 | 0x 1777 |
| K | 0x4B | | 0000,0001,1101,0111 | 471 | 0x 1D7 |
| L | 0x4C | | 0000,0001,0111,0101 | 373 | 0x 175 |
| M | 0x4D | | 0000,0000,0111,0111 | 119 | 0x 77 |
| N | 0x4E | | 0000,0000,0001,1101 | 29 | 0x 1D |
| O | 0x4F | | 0000,0111,0111,0111 | 1911 | 0x 777 |
| P | 0x50 | | 0000,0101,1101,1101 | 1501 | 0x 5DD |
| Q | 0x51 | | 0001,1101,1101,0111 | 7639 | 0x 1DD7 |
| R | 0x52 | | 0000,0000,0101,1101 | 93 | 0x 5D |
| S | 0x53 | | 0000,0000,0001,0101 | 21 | 0x 15 |
| T | 0x54 | | 0000,0000,0000,0111 | 7 | 0x 7 |
| U | 0x55 | | 0000,0000,0101,0111 | 87 | 0x 57 |
| V | 0x56 | | 0000,0001,0101,0111 | 343 | 0x 157 |
| W | 0x57 | | 0000,0001,0111,0111 | 375 | 0x 177 |
| X | 0x58 | | 0000,0111,0101,0111 | 1879 | 0x 757 |
| Y | 0x59 | | 0001,1101,0111,0111 | 7543 | 0x 1D77 |
| Z | 0x5A | | 0000,0111,0111,0101 | 1909 | 0x 775 |
| | | Notes: - A dash is equal to three dots - The space between parts of the same letter is equal to one dot - The space between two letters is equal to three dots - The space between two words is equal to seven dots - LED on for one dot - LED off (same length as one dot) | | | |

Lab-2 Submission form

| | | | |
|------------------------------|------------------------------|------------------------------|------------|
| 201 <input type="checkbox"/> | 202 <input type="checkbox"/> | 203 <input type="checkbox"/> | Demo date: |
| 204 <input type="checkbox"/> | 205 <input type="checkbox"/> | 206 <input type="checkbox"/> | |

Submission Statement: We (I) are (am) submitting this report for grading in ECE 222. We (I) certify that this report (including any code, descriptions, flowcharts, etc., that are part of the submission) were written by us (me) and have received no prior academic credit at this university or any other institution. **The penalty for copying or plagiarism will be a grade of zero (0).**

| Member 1 | Member 2 |
|-------------------------------|-------------------------------|
| Name: | Name: |
| UW-ID (NOT student #) | UW-ID (NOT student #) |
| Signature: | Signature: |

Note: Reports submitted without a signed submission statement will receive a grade of zero (0).

| | | Weight | Grade | Comment |
|--|----------------|------------|-------|---------|
| Part-I | Pre-lab | 0 | | |
| Part-II Lab-demo | Lab completion | 40 | | |
| | Questions | 40 | | |
| Part-III Lab report | Code quality | 10 | | |
| | Code comments | 10 | | |
| Penalty for using flash memory for code development | | -20 | | |
| Total | | 100 | | |

Marking TA:

Lab-3: Input/Output interfacing

Objective

The objective of this lab is to learn how to use peripherals (LEDs, switch) connected to a microprocessor. The ARM CPU is connected to the outside world using Ports and in this lab you will setup, and use, Input and Output ports.

What you do

In this lab you will measure how fast a user responds (reflex-meter) to an event accurate to a 10th of a millisecond. Initially all LEDs are off and after a random amount of time (between 2 to 10 seconds), one LED turns on and then the user presses the push button.

Between the two events of 'Turning the LED on' and 'Pressing the push button', a 32 bit counter is incremented every 10th of a millisecond in a loop. The final value of this 32 bit number will be sent to the 8 LEDs in separate bytes with a 2 second delay between them.

A delay of 2 to 10 seconds +/-5% is required. The delay routine will be modified to delay $R0 * 100\mu s$ and so R0 must be between 20,000 and 100,000. To get a 100us delay loop just scale the value used in Lab 1. A **pseudorandom** number in R11 will be between 1 and 65,535 and this is used to provide the delay value between 20,000 and 100,000. At least two ways to do this are:

- 1) Keep generating random numbers until one fits the range required
- 2) Take a certain number of bits (between 8 and 16) of R11 and scale it to fit

Background

During the code development, you will need to read through some parts of chapters 7, 8, and 9 of the **LPC17xx User manual** [1] (literature number UM10360). Part of the information you will need to gather from that reference is given in Appendix D of this lab manual. Appendix F also shows the full blown schematic diagram of the MCB1700 board.

Note that in the previous labs the LEDs have not been turned on and off in the most efficient manner! The addresses 0x2009c020 and 0x2009c040 are the FIO1DIR and FIO2DIR registers – they configure a pin(s) on a port as being an input or output! When a 0, the default value, is written into a bit in the FIOxDIR register, then that pin is setup to be an input. Each input pin includes a pull-up resistor.

Writing a 1 to a FIOxDIR bit sets a pin as an output. The default output value for a pin is 0 and so when a pin is set to output, it goes to a logic low unless a different value is written to the FIOxPIN register.

The addresses for the FIOxDIR registers can be found in Table 102 GPIO Register Map of the LPC17xx User Manual.

If you do not change how you drive the LEDs – then you must invert ONLY the values written out to the LEDs as they're active low. DO NOT INVERT ALL BITS being written to the ports! Only invert the bits being driven to the LEDs – all other bits MUST BE ZERO – OR ELSE you'll have a hell of a debugging time in Lab #4!

Pre-lab

There are no deliverable for this part. It is for your practice only.

Determine what to write in which memory-address to turn any of the 8 LEDs on or off. Refer to Appendix D or the **LPC17xx User manual** [1] for more information. By doing this step, you should understand the code that was given to you in Lab-1 to turn the P1.28 LED on and off.

The INTO push button, pin P2.10, is thankfully, by default, configured as a GPIO port with a pull-up resistor. So it does not have to be reconfigured with PINSEL3 and PINSEL4.

In-lab procedure

Please note that you will have to demonstrate **two parts** in Lab-3:

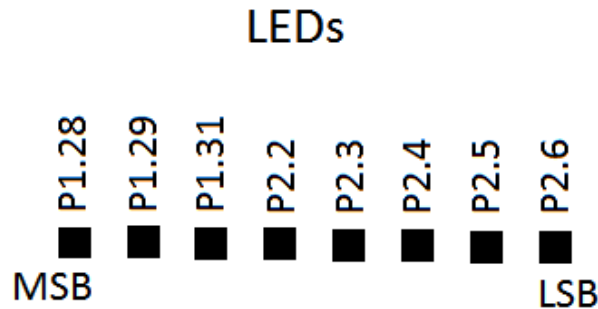
- A simple counter subroutine that increments from 0x00 to 0xFF, wraps to 0, and continues counting. This will prove that the bits are displayed in the correct order on the LEDs.
- The reflex-meter.

Here are the suggested steps to implement this program:

1. Modify your assembly language code to implement a 0.1 millisecond delay routine.
2. Create a simple counter to generate numbers from 0 to 255 (0xff) and write these to the 8 LEDs to verify the LED decoding functionality. Use a 100 ms delay between numbers.
3. To implement the reflex-meter project:
 - a) Turn off all 8 LEDs
 - b) Call the provided **pseudorandom** number subroutine to generate a 16 bit number, then scale it and add an offset, to result in a 2 to 10 +/- 5% second delay.
 - c) Call the delay function for that amount of time (in 0.1 millisecond increments)
 - d) Turn one LED on (P1.29) and start incrementing a register value once every 0.1

millisecond

- e) Monitor the status of the INT0 push button using polling
- f) Once the button is pressed, stop incrementing (exit from the loop)
- g) Send the first 8 bits (least significant part) of the register to the LEDs
- h) Wait for 2 seconds
- i) Send the next 8 bits and wait again.
- j) Do the above steps two more times until all 32 bits are shown on the LEDs.
- k) Wait 5 seconds and go back to step (h).



Optional Improvements

Here are some optional ways to improve the program if you have the time and interest:

- 1- Go to a website like <http://www.humanbenchmark.com/tests/reactiontime/> and measure your average reaction time for comparison to your ARM program.
- 2- Merge the two programs into one by starting with the simple counter subroutine and when the button is pressed the program changes to the reflex-meter.
- 3- To make the pseudo-random generator more random keep calling it every 100uS, while in the counting subroutine. The variable time delay, while waiting for the user to press the button to exit the counter, ensures a random delay in the game.
- 4- To enable replaying the game; if the button is pressed during the 5 second delay - restart the reflex-meter.
- 5- If the 8 highest bits of the time delay are 0x0 simply perform a 3 second delay (while optionally checking for a key press to restart the game) and then redisplay the time delay again.

Lab report

Submit your commented, well-written code for the simple counter **AND** reflex-meter, and a picture of the marked Submission form, to the Lab-3 drop-box LEARN.

Answer these questions and put them as comments at the end of your program:

- 1- If a 32-bit register is counting user reaction time in 0.1 milliseconds increments, what is the maximum amount of time which can be stored in 8 bits, 16-bits, 24-bits and 32-bits?
- 2- Considering typical human reaction time, which size would be the best for this task (8, 16, 24, or 32 bits)?

Extra Information

Random numbers with Fibonacci linear feedback shift registers at WikiPedia:

http://en.wikipedia.org/wiki/Linear_feedback_shift_register

Lab-3 Submission form

| | | | |
|------------------------------|------------------------------|------------------------------|------------|
| 201 <input type="checkbox"/> | 202 <input type="checkbox"/> | 203 <input type="checkbox"/> | Demo date: |
| 204 <input type="checkbox"/> | 205 <input type="checkbox"/> | 206 <input type="checkbox"/> | |

Submission Statement: We (I) are (am) submitting this report for grading in ECE 222. We (I) certify that this report (including any code, descriptions, flowcharts, etc., that are part of the submission) were written by us (me) and have received no prior academic credit at this university or any other institution. **The penalty for copying or plagiarism will be a grade of zero (0).**

| Member 1 | Member 2 |
|-------------------------------|-------------------------------|
| Name: | Name: |
| UW-ID (NOT student #) | UW-ID (NOT student #) |
| Signature: | Signature: |

Note: Reports submitted without a signed submission statement will receive a grade of zero (0).

| | | Weight | Grade | Comment |
|--------------------------------|---|--------|-------|---------|
| Part-I | Pre-lab | 0 | | |
| Part-II Lab-demo | Simple counter | 15 | | |
| | Reflex-meter | 25 | | |
| | Questions | 40 | | |
| Part-III Lab report | Code quality | 6 | | |
| | Code comments | 6 | | |
| | Prove time delay meets 2 to 10 sec +/- 5% spec | 6 | | |
| | Two questions | 2 | | |
| Penalty for using flash memory | | -20 | | |
| Total | | 100 | | |

Lab-4: Interrupt handling

Objective

The objective of this lab is to learn about interrupts. You will enable an interrupt source in the LPC1768 microprocessor, and write an interrupt service routine (ISR) that is triggered when the INT0 button is pressed. The ISR returns to the main program after handling the interrupt.

Pre-lab

There is no deliverable for this part. It is for your education only.

Read the contents of Appendix E of this lab manual. Review the information presented in the section '9.5.6 – GPIO interrupt registers' of the **LPC17xx User manual** [1].

What you do

Reuse your Lab-3 code. The random number generator output will be used to generate a number which gives a time delay of 5.0 to 25.0 seconds with a resolution of 0.1s. I.e. An integer between 50 and 250.

Once the program is started R6 is set to 0. Then all eight LEDs flash on and off at a rate between 1 and 10 Hz while the random number routine is called continuously until R6 is non zero. An INT0 button press causes the interrupt routine to run. It creates a random integer, between 50 and 250, generated from the R11 random number, and stores it in R6. The main program then displays this (without a decimal so that 5.6 seconds displays as 56 in binary) on the 8 LEDs. The program delays one second and then the count in R6 is decremented by the equivalent of 1 second (10) and the new count (time left) displayed. This continues. When the count would go to zero, or less, all decrementing stops and R6 is set to 0 and the program starts flashing all LEDs again and so the process repeats. **Do NOT USE BLO, BHS, BGT, BGE, BLT or BLE as they are for signed number comparisons only and they will not work as expected!**

In-lab procedure

The following steps are suggested for handling the INT0 button as a source of interrupt:

- The INT0 button or P2.10 pin is, by configured correctly as a GPIO input pin.
- Enable the EINT3 channel (External INTerrupt 3) which is shared with GPIO interrupts with ISER0 (Interrupt Set Enable 0 – Table 52 of the **LPC17xx User manual** [1])
- Enable the GPIO interrupt on pin P2.10 for falling edge with IO2IntEnF (IO 2 INTerrupt Enable Falling) using Table 117 of the **LPC17xx User manual** [1]. P2.10 is high when the INT0 button is not pressed. This is why sensitivity to falling edge (see Figure 4.1) is necessary. If you configure it so that it is sensitive to the rising edge on P2.10 then the

program will react to the releasing of the push button.

- Complete the External Interrupt 3 (EINT3) interrupt service routine EINT3_IRQHandler. There are two parts to this routine:
 - o Generate a new random number from R11 that is between 50 and 250 and store it in R6
 - o Clear the cause of interrupt (falling edge on P2.10 pin) using IO2IntClr (IO 2 INTerrupt CLearR) detailed in Table 123 of the **LPC17xx User manual** [1]. Do NOT disable the interrupt input. Just clear the latch that has recorded an interrupt on the falling-edge for the GPIO pin. This is often missed.

Hint: Putting a breakpoint in the ISR for EINT3_IRQHandler will reveal if the ISR is called or not, and whether it is run only once or more often. This is very helpful when debugging.



Figure 4.1 – Falling/Rising edges when the INT0 push-button is pressed/released

The following steps are suggested for implementing the procedure. Add each step and test the code.

- Revert to the Lab 3 code which used a counter to exercise all 8 LED outputs
- Modify your delay subroutine to have a resolution of 100ms (ie delay R0 * 0.1s)
- Generate a random number, in R6, by using the random number in R11. The 16-bit output of the random number generator in R11 processed to generate an integer between 50 and 250 - a delay between 5.0 and 25.0 seconds.
- Count the random number, in R6, down, using 1 second delays, until the count would be ≤ 0 and hold the display at zero on the LEDs
- Add code to configure a falling edge INT0 interrupt and enable it. Your interrupt service routine (ISR) will be called when the button is pressed.
- Write code so that the ISR generates a new random number from R11 and stores it in R6

Yes there are issues with this procedure. After a new random number is generated it can take up to one second to display it and the new random number may be decremented by one second nearly immediately after being generated. Fixing these problems is optional.

Lab report

Submit your commented, well-written code for Lab 4, and a picture of the marked Submission Form, to the Lab-4 drop-box on LEARN.

Lab-4 Submission form

| | | | | |
|-------------------------------------|------------------------------|------------------------------|------------------------------|------------|
| Class: 001 <input type="checkbox"/> | 201 <input type="checkbox"/> | 202 <input type="checkbox"/> | 203 <input type="checkbox"/> | Demo date: |
| 002 <input type="checkbox"/> | 204 <input type="checkbox"/> | 205 <input type="checkbox"/> | 206 <input type="checkbox"/> | |

Submission Statement: We (I) are (am) submitting this report for grading in ECE 222. We (I) certify that this report (including any code, descriptions, flowcharts, etc., that are part of the submission) were written by us (me) and have received no prior academic credit at this university or any other institution. **The penalty for copying or plagiarism will be a grade of zero (0).**

| Member 1 | Member 2 |
|-------------------------------|-------------------------------|
| Name: | Name: |
| UW-ID (NOT student #) | UW-ID (NOT student #) |
| Signature: | Signature: |

Note: Reports submitted without a signed submission statement will receive a grade of zero (0).

| | | Weight | Grade | Comment |
|---|----------------|--------|-------------|---------|
| Part-I | Pre-lab | 0 | | |
| Part-II Lab-demo | Lab completion | 40 | | |
| | Questions | 40 | | |
| Part-III Lab report | Code quality | 10 | | |
| | Code comments | 10 | | |
| Penalty for using flash memory for code development (LPC 1768 only) | | -20 | | |
| | | | | |
| Total | | 100 | <div></div> | |

Appendix A: The LPC1768 microprocessor

Figure A.1 shows block diagram of the LPC1768. Detailed information can be found in chapter 1 of the document **LPC17xx User manual** [1].

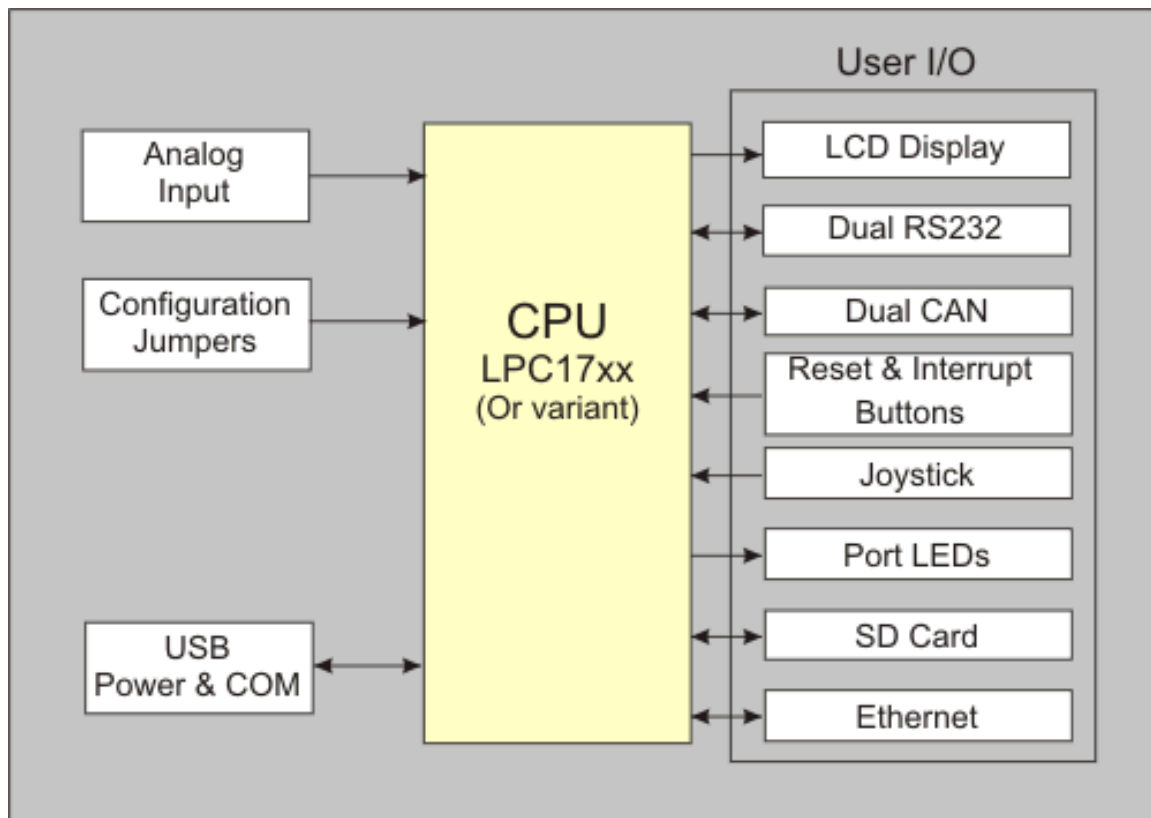


Figure A.1 – Block diagram of LPC1768 [1]

Figure A.2 shows a simplified block diagram of the LPC1768 microprocessor.

As you can see there is no memory block in the above figure. This is because all volatile (RAM) and non-volatile (Flash) memories are on-chip.

Figure A.3 shows some details of the CPU and buses.

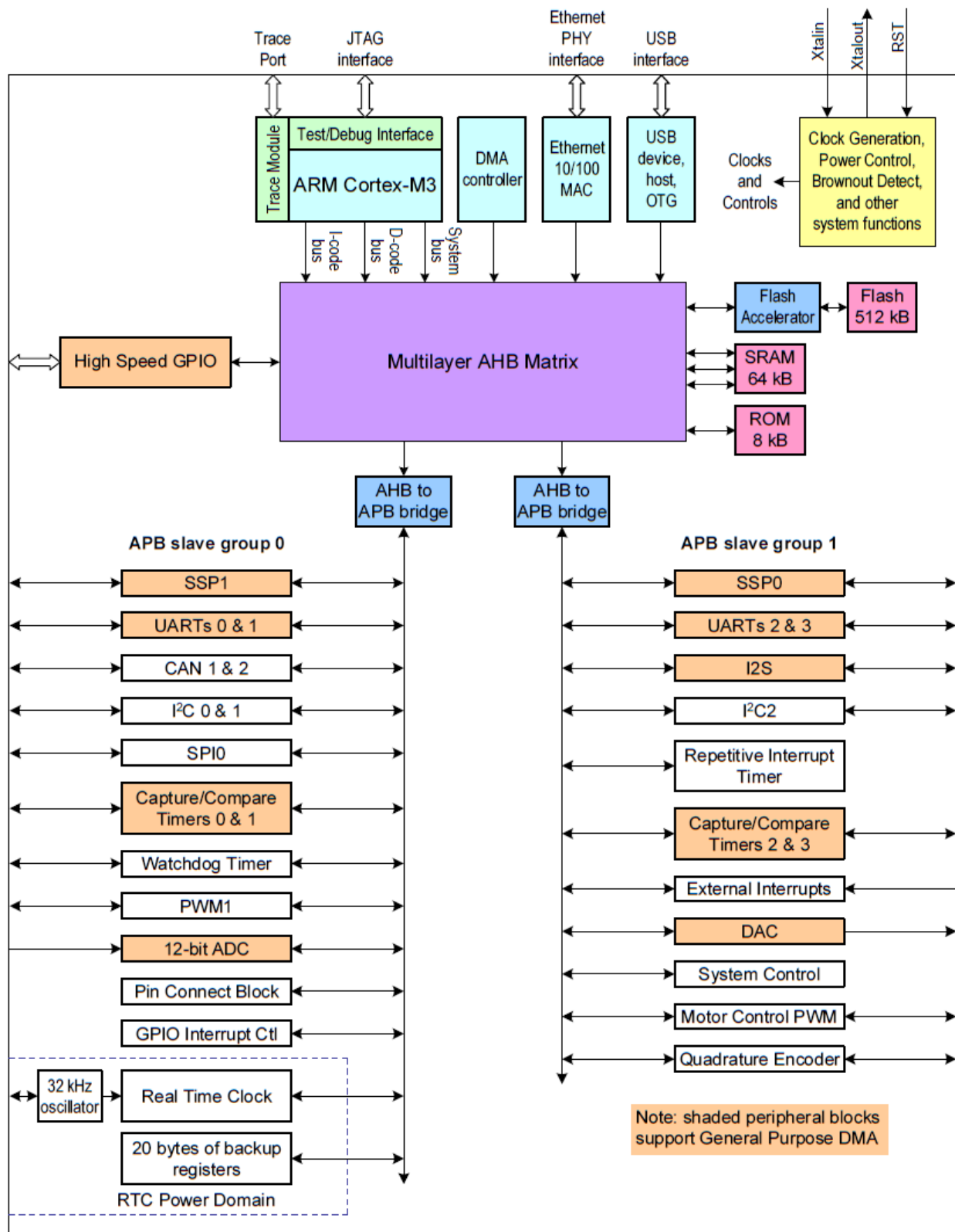


Figure A.2 – Simplified block diagram of LPC1768 [1]

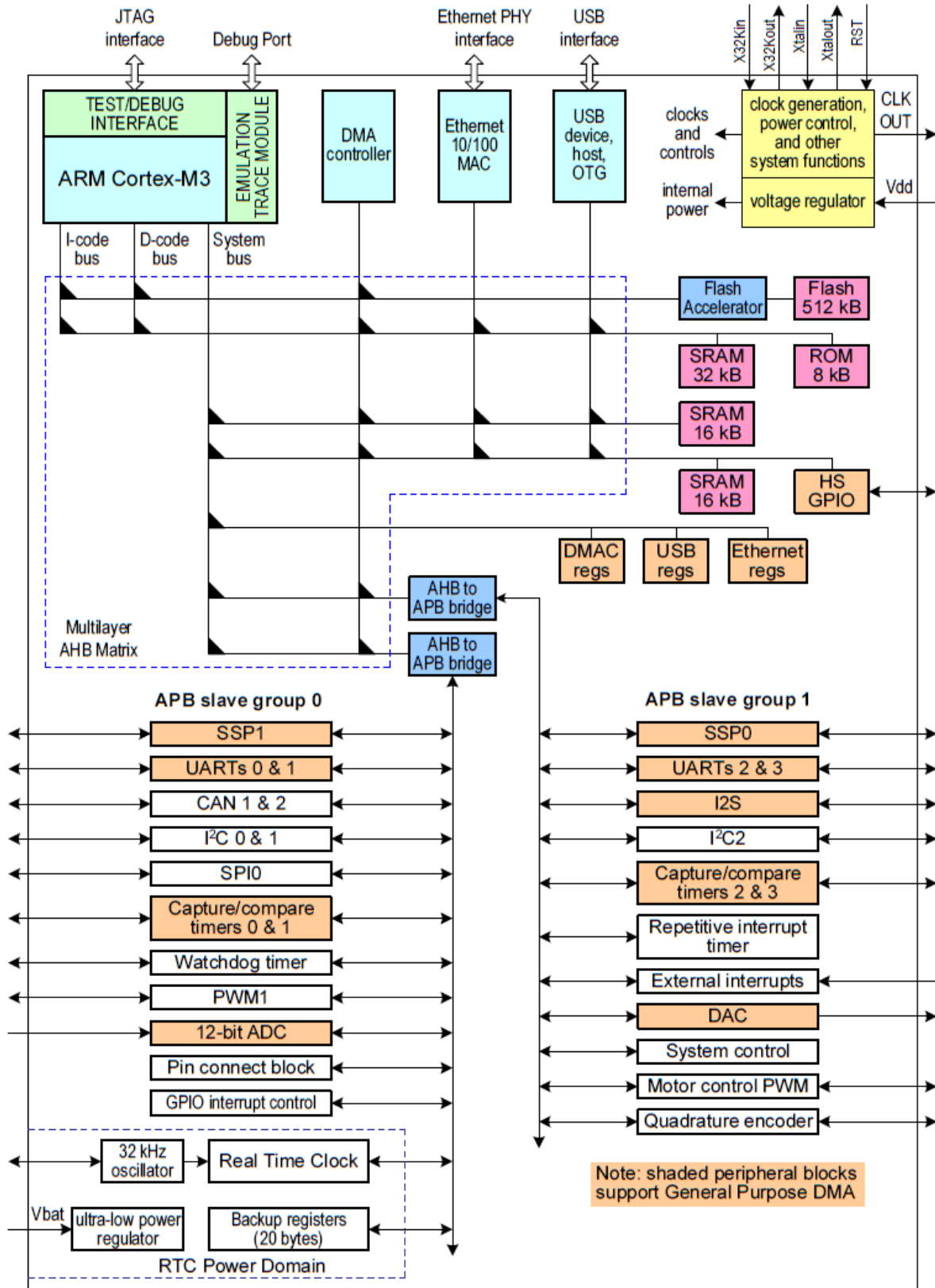


Figure A.3 – LPC1768 block diagram, CPU, and buses [1]

Appendix B: Abbreviated Instruction set summary

The processor implements a version of the Thumb instruction set. Table B.1 lists some supported instructions [7].

Conditional branching is done with the conditional branch instructions. An “L” can be added to the branches (BLNE, BLHI, ...) to save the return address in the Link Register. Some conditional branches are:

| Flag | Flag Set | Flag Clear |
|------|-------------------------------|--------------------------------------|
| Z | BEQ – Equal to Zero | BNE – Not Equal to Zero |
| C | BCS – Carry Set | BCC – Carry Clear |
| N | BMI – Minus – result negative | BPL – Plus – result positive or zero |
| C,Z | BLS – Lower or Same | BHI – Higher |

Avoid BLO, BHS, BVC, BVS, BGT, BGE, BLT, BLE as they are for signed number comparisons ONLY.

Note: In Table B.1:

- angle brackets, <>, enclose alternative forms of the operand
- an “S” suffix sets condition codes (N,V,Z) for math and logic operations
- braces, {}, enclose optional operands or condition
- the Operands column is not exhaustive
- Op2 is a flexible second operand that can be either a register or a constant
- many instructions can use an optional condition code suffix such as S (set condition codes), H (half-word size), B (byte size), T (register top half).

For more information on the instructions and operands, see the instruction descriptions.

Table B.1. Cortex-M3 instructions [7]

| Mnemonic | Operands | Brief description | Flags | Also See |
|----------------------------|---------------------------------------|---|---------|--------------------------------|
| ADD{S} ADC{S} ADD{W} | {Rd,} Rn, Op2 {Rd,} Rn, #imm12 | Add Add with Carry Add with Immediate | N,Z,C,V | ADD, ADC, SUB, SBC, and RSB |
| AND{S} | {Rd,} Rn, Op2 | Logical AND | N,Z,C | AND, ORR, EOR, BIC, and ORN |
| ASR{S} | Rd, Rm, <Rs #n> | Arithmetic Shift Right | N,Z,C | ASR, LSL, LSR, ROR, and RRX |

| Mnemonic | Operands | Brief description | Flags | Also See |
|-----------------------------------|------------------------|--|---------|---------------------------------------|
| B BL | label | Branch Always Branch with Link | - | <i>B, BL, BX, and BLX</i> |
| BFC | Rd, #lsb, #width | Bit Field Clear | - | <i>BFC and BFI</i> |
| BFI | Rd, Rn, #lsb, #width | Bit Field Insert | - | <i>BFC and BFI</i> |
| BIC{S} | {Rd,} Rn, Op2 | Bit Clear | N,Z,C | <i>AND, ORR, EOR, BIC, and ORN</i> |
| BLX BX | Rm | Branch indirect with Link Branch indirect | - | <i>B, BL, B_rX, and BLX</i> |
| CBNZ CBZ | Rn, label | Compare and Branch FORWARD if Non Zero if Zero | - | <i>CBZ and CBNZ</i> |
| CLZ | Rd, Rm | Count Leading Zeros | - | <i>CLZ</i> |
| CMP CMN | Rn, Op2 | Compare, Rn – Op2 Compare Negative | N,Z,C,V | <i>CMP and CMN</i> |
| EOR{S} | {Rd,} Rn, Op2 | Exclusive OR | N,Z,C | <i>AND, ORR, EOR, BIC, and ORN</i> |
| LDM | Rn{!}, reglist | Load Multiple registers, increment after | - | <i>LDM and STM</i> |
| LDMDB, LDMEA | Rn{!}, reglist | Load Multiple registers, decrement before | - | <i>LDM_r and STM</i> |
| LDMFD, LDMIA | Rn{!}, reglist | Load Multiple registers, increment after | - | <i>LDM and STM</i> |
| LDR LDRB, LDRBT LDRH, LDRHT | Rt, [Rn, #offset] | Load Register with word (32-bit) Load Register with byte Load Register with Halfword | - | <i>Memory access instructions</i> |
| LDRD | Rt, Rt2, [Rn, #offset] | Load Register with two bytes | - | <i>LDR and STR, immediate offset</i> |
| LSL{S} LSR{S} | Rd, Rm, <Rs #n> | Logical Shift Left Logical Shift Right | N,Z,C | <i>ASR, LSL, LSR, ROR, and RRX</i> |
| MOV{S} MVN{S} | Rd, Op2 | Move Move NOT | N,Z,C | <i>MOV and MVN</i> |

| Mnemonic | Operands | Brief description | Flags | Also See |
|------------------------------|-----------------------------------|---|--------------|--------------------------------------|
| MOVT | Rd, #imm16 | Move Top | - | <i>MOVT</i> |
| MOV{W} | Rd, #imm16 | Move 16-bit constant | N,Z,C | <i>MOV and MVN</i> |
| MUL{S} | {Rd,} Rn, Rm | Multiply, 32-bit result | N,Z | <i>MUL, MLA, and MLS</i> |
| NOP | - | No Operation | - | <i>NOP</i> |
| ORR{S} ORN{S} | {Rd,} Rn, Op2 | Logical OR (Rn OR Op2) Logical OR NOT (Rn OR NOT Op2) | N,Z,C | <i>AND, ORR, EOR, BIC, and ORN</i> |
| POP PUSH | reglist | Pop registers from stack Push registers onto stack | - | <i>PUSH and POP</i> |
| RBIT REV | Rd, Rn | Reverse Bits Reverse byte order in a word | - | <i>REV, REV16, REVSH, and RBIT</i> |
| ROR{S} | Rd, Rm, <Rs #n> | Rotate Right | N,Z,C | <i>ASR, LSL, LSR, ROR, and RRX</i> |
| SBFX | Rd, Rn, #lsb, #width | Signed Bit Field Extract | - | <i>SBFX and UBFX</i> |
| SDIV | {Rd,} Rn, Rm | Signed Divide | - | <i>SDIV and UDIV</i> |
| STM | Rn{!}, reglist | Store Multiple registers, increment after | - | <i>LDM and STM</i> |
| STMDB, STMEA | Rn{!}, reglist | Store Multiple registers, decrement before | - | <i>LDM and STM</i> |
| STMFD, STMIA | Rn{!}, reglist | Store Multiple registers, increment after | - | <i>LDM and STM</i> |
| STR{T} STRB{T} STRH{T} | Rt, [Rn, #offset] | Store Register word Store Register byte Store Register Halfword | - | <i>Memory access instructions</i> |
| STRD | Rt, Rt2, [Rn, #offset] | Store Register two words | - | <i>LDR and STR, immediate offset</i> |
| SUB{S} SBC{S} SUB{W} | {Rd,} Rn, Op2 {Rd,} Rn, #imm12 | Subtract Subtract with Carry Subtract with Immediate | N,Z,C,V | <i>ADD, ADC, SUB, SBC, and RSB</i> |
| SXTB SXTH | {Rd,} Rm {,ROR #n} | Sign extend a byte Sign extend a halfword | - | <i>SXT and UXT</i> |

| Mnemonic | Operands | Brief description | Flags | Also See |
|-----------------|----------------------|--|--------------|---------------------------------------|
| TEQ TST | Rn, Op2 | Test Equivalence (Rn AND Op2) Test (Rn EOR Op2) | N,Z,C | <i>TST and TEQ</i> |
| UBFX | Rd, Rn, #lsb, #width | Unsigned Bit Field Extract | - | <i>SBFX and UBFX</i> |
| UDIV | {Rd,} Rn, Rm | Unsigned Divide | - | <i>SDIV and UDIV</i> |
| UMULL | RdLo, RdHi, Rn, Rm | Unsigned Multiply (32 x 32), 64-bit result | - | <i>UMULL, UMLAL, SMULL, and SMLAL</i> |
| UXTB UXTH | {Rd,} Rm {,ROR #n} | Zero extend a Byte Zero extend a Halfword | - | <i>SXT and UXT</i> |

Appendix C: Memory map

Table C.1 shows a rough memory map. Detailed information can be extracted from chapter 2 of the document **LPC17xx User manual**.

Table C.1 Memory usage for Coretx-M3 and LPC1768 Microprocessor

| Address range | General use | Address range details for our boards | Description |
|---------------------------|--|--------------------------------------|--|
| 0x0000 0000 – 0x1FFF FFFF | On-chip non-volatile memory | 0x0000 0000 – 0x0007 FFFF | Flash memory |
| | On-chip SRAM | 0x1000 0000 – 0x1000 7FFF | 32 kB user program memory |
| | Boot ROM | 0x1FFF 0000 – 0x1FFF 1FFF | 8 kB boot ROM with flash services |
| 0x2000 0000 – 0x3FFF FFFF | On-chip SRAM (typically used for peripheral data) | 0x2007 C000 – 0x2007 FFFF | 16 kB AHB SRAM – Bank 0 |
| | | 0x2008 0000 – 0x2008 3FFF | 16 kB AHB SRAM – Bank 1 |
| | GPIO | 0x2009 C000 – 0x2009 FFFF | 16 kB |
| 0x4000 0000 – 0x5FFF FFFF | APB Peripherals | 0x4000 0000 – 0x4007 FFFF | APB0 Peripherals |
| | | 0x4008 0000 – 0x400F FFFF | APB1 Peripherals |
| | AHB Peripherals | 0x5000 0000 – 0x501F FFFF | DMA, Ethernet, USB |
| 0xE000 0000 – 0xE00F FFFF | Cortex-M3 Private Peripheral Bus | 0xE000 0000 – 0xE00F FFFF | Cortex-M3 functions including NVIC and System Tick Timer |

Figure C.1 shows a detailed memory map for the LPC1768 MCU.

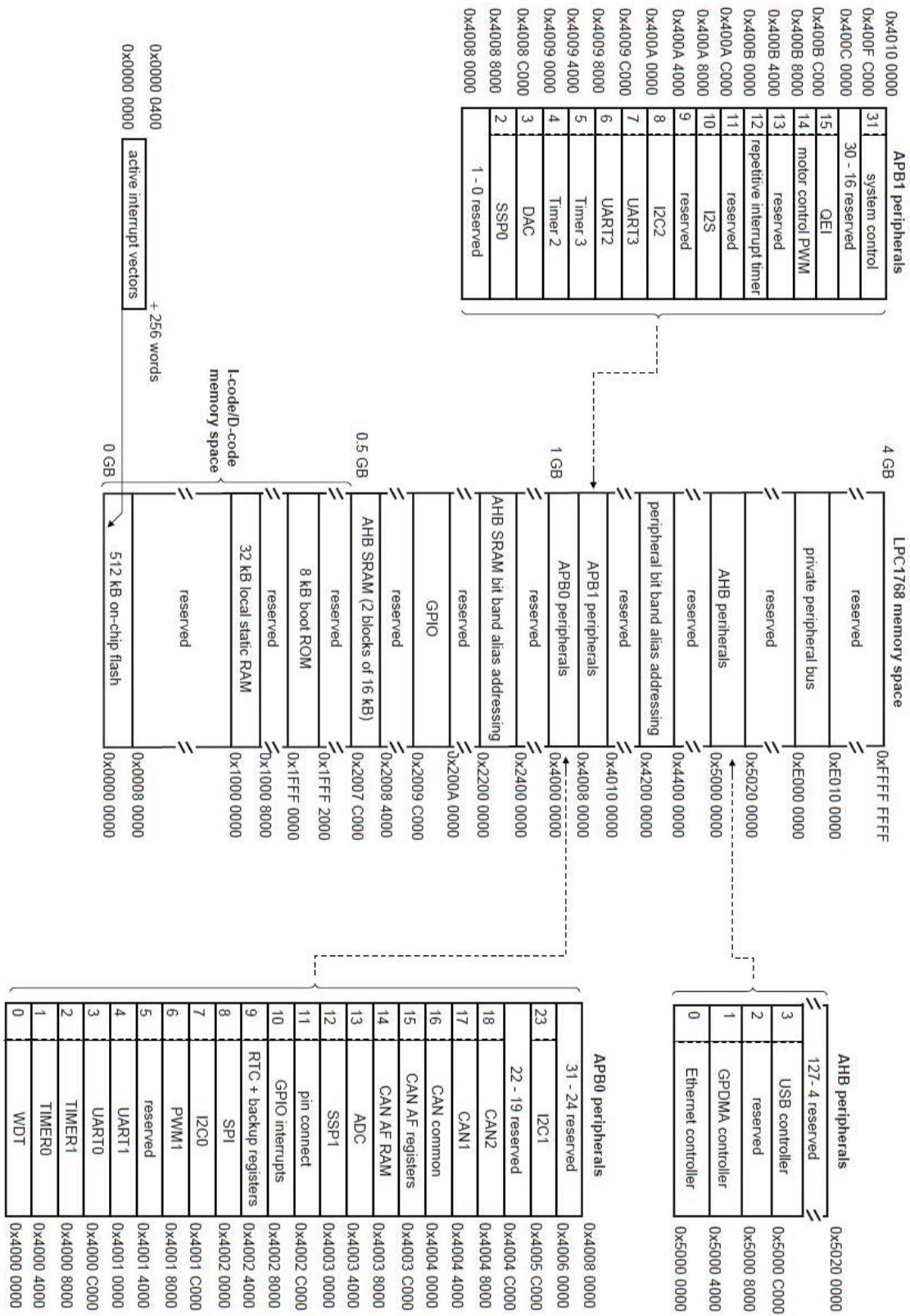


Figure C.1 – Detailed memory map for LPC1768 microprocessor

Appendix D: Input / Output ports

Detailed information on this topic can be found in chapters 7, 8, and 9 of the document **LPC17xx User manual** [1].

The LPC1768 microprocessor on the MCB1700 board has 100 pins, as shown in figure D.1. Figure D.2 shows the functionality of selected pins which we are interested in.

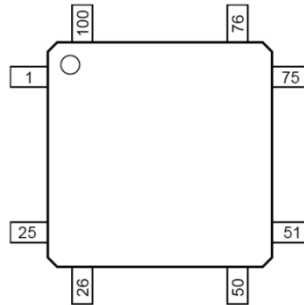


Figure D.1 – The LPC1768 microprocessor in LQFP100 pin configuration [1]

As you can see in figure 2, many pins can have more than one functionality. For example, pin number 73 named as P2.2 can take four different roles as indicated by P2.2/PWM1.3/CTS1/TRACEDATA3.

Actually many pins of LPC1768 can have up to four different functionalities. For each pin there is a two bits field in a registers named PINSELx that determines the functionality of the pin. Table D.1 shows how to choose the bits pin functionality. For more information refer to Appendix A or tables 80 to 87 of the **LPC17xx User manual**.

In this lab, we would like to define the pins we are going to work with as GPIO (general purpose input output). In most cases the GPIO is default functionality of the pin. This means, in order to define a pin as GPIO, you need to do nothing!

| IC1A | | | | | |
|-------|----|------------------------------|--------------------------------|----|-------|
| P0.0 | 46 | P0.0/RD1/TXD3/SDA1 | P1.0/ENET_TXD0 | 95 | P1.0 |
| P0.1 | 47 | P0.1/TD1/RXD3/SCL1 | P1.1/ENET_TXD1 | 94 | P1.1 |
| P0.2 | 98 | P0.2/TXD0/AD0.7 | P1.4/ENET_TX_EN | 93 | P1.4 |
| P0.3 | 99 | P0.3/RXD0/AD0.6 | P1.8/ENET_CRS | 92 | P1.8 |
| P0.4 | 81 | P0.4/I2SRX_CLK/RD2/CAP2.0 | P1.9/ENET_RXD0 | 91 | P1.9 |
| P0.5 | 80 | P0.5/I2SRX_WS/TD2/CAP2.1 | P1.10/ENET_RXD1 | 90 | P1.10 |
| P0.6 | 79 | P0.6/I2SRX_SDA/SSEL1/MAT2.0 | P1.14/ENET_RX_ER | 89 | P1.14 |
| P0.7 | 78 | P0.7/I2STX_CLK/SCK1/MAT2.1 | P1.15/ENET_REF_CLK | 88 | P1.15 |
| | | | P1.16/ENET_MDC | 87 | P1.16 |
| | | | P1.17/ENET_MDIO | 86 | P1.17 |
| P0.8 | 77 | P0.8/I2STX_WS/MISO1/MAT2.2 | | | |
| P0.9 | 76 | P0.9/I2STX_SDA/MOSI1/MAT2.3 | | | |
| P0.10 | 48 | P0.10/TXD2/SDA2/MAT3.0 | P1.18/USB_UP_LED/PWM1.1/CAP1.0 | 32 | P1.18 |
| P0.11 | 49 | P0.11/RXD2/SCL2/MAT3.1 | P1.19/MC0A/nUSB_PPWR/CAP1.1 | 33 | P1.19 |
| P0.15 | 62 | P0.15/TXD1/SCK0/SCK | P1.20/MCFB0/PWM1.2/SCK0 | 34 | P1.20 |
| | | | P1.21/MCABORT/PWM1.3/SSEL0 | 35 | P1.21 |
| P0.16 | 63 | P0.16/RXD1/SSEL0/SSEL | P1.22/MC0B/USB_PWRD/MAT1.0 | 36 | P1.22 |
| P0.17 | 61 | P0.17/CTS1/MISO0/MISO | P1.23/MCFB1/PWM1.4/MISO0 | 37 | P1.23 |
| P0.18 | 60 | P0.18/DCD1/MOSI0/MOSI | P1.24/MCFB2/PWM1.5/MOSI0 | 38 | P1.24 |
| P0.19 | 59 | P0.19/DSR1/SDA1 | P1.25/MC1A/MAT1.1 | 39 | P1.25 |
| P0.20 | 58 | P0.20/DTR1/SCL1 | P1.26/MC1B/PWM1.6/CAP0.0 | 40 | P1.26 |
| P0.21 | 57 | P0.21/RI1/RD1 | P1.27/CLKOUT/nUSB_OVRCR/CAP0.1 | 43 | P1.27 |
| P0.22 | 56 | P0.22/RTS1/TD1 | P1.28/MC2A1.0/MAT0.0 | 44 | P1.28 |
| P0.23 | 9 | P0.23/AD0.0/I2SRX_CLK/CAP3.0 | P1.29/MC2B/PCAP1.1/MAT0.1 | 45 | P1.29 |
| | | | P1.30/VBUS/AD0.4 | 21 | P1.30 |
| P0.24 | 8 | P0.24/AD0.1/I2SRX_WS/CAP3.1 | P1.31/SCK1/AD0.5 | 20 | P1.31 |
| P0.25 | 7 | P0.25/AD0.2/I2SRX_SDA/TXD3 | | | |
| P0.26 | 6 | P0.26/AD0.3/AOUT/RXD3 | | | |
| P0.27 | 25 | P0.27/SDA0/USB_SDA | P2.0/PWM1.1/TXD1 | 75 | P2.0 |
| P0.28 | 24 | P0.28/SCL0/USB_SCL | P2.1/PWM1.2/RXD1 | 74 | P2.1 |
| P0.29 | 29 | P0.29/USB_D+ | P2.2/PWM1.3/CTS1/TRACEDATA3 | 73 | P2.2 |
| P0.30 | 30 | P0.30/USB_D- | P2.3/PWM1.4/DCD1/TRACEDATA2 | 70 | P2.3 |
| | | | P2.4/PWM1.5/DSR1/TRACEDATA1 | 69 | P2.4 |
| | | | P2.5/PWM1.6/DTR1/TRACEDATA0 | 68 | P2.5 |
| | | | P2.6/PCAP1.0/RI1/TRACECLK | 67 | P2.6 |
| P3.25 | 27 | P3.25/MAT0.0/PWM1.2 | | | |
| P3.26 | 26 | P3.26/STCLK/MAT0.1/PWM1.3 | | | |
| | | | P2.7/RD2/RTS1 | 66 | P2.7 |
| | | | P2.8/TD2/TXD2 | 65 | P2.8 |
| P4.28 | 82 | P4.28/RX_MCLK/MAT2.0/TXD3 | P2.9/USB_CONNECT/RXD2 | 64 | P2.9 |
| P4.29 | 85 | P4.29/TX_MCLK/MAT2.1/RXD3 | P2.10/nEINT0/NMI | 53 | P2.10 |
| | | | P2.11/nEINT1/I2STX_CLK | 52 | P2.11 |
| | | | P2.12/nEINT2/I2STX_WS | 51 | P2.12 |
| | | | P2.13/nEINT3/I2STX_SDA | 50 | P2.13 |

Joystick centre

LED

INT0

LPC1768

Figure D.2 – LED connections to NXP LPC1768 [4]

In total, there is ten PINSEL (pin select) registers. Two of them, PINSEL3 and PINSEL4, are used for pins connected to LEDs. The PINSEL3 (address 0x4002 C00C) controls P1[31:16] pins and the PINSEL4 (0x4002 C010) controls P2[15:0] pins. Table D.2 shows pin function select for PINSEL3 (partial), and table D.3 shows pin function select for PINSEL4 (partial). Pins connected to LEDs have **bold purple** font. You should also find the pins connected to INTO and Joystick centre push buttons.

Table D.1 – Pin function select register bits

| PINSEL0 to PINSEL9 values | Function | Value after reset |
|---------------------------|---|-------------------|
| 00 | Primary (default) function, typically GPIO port | 00 |
| 01 | First alternate function | |
| 10 | Second alternate function | |
| 11 | Third alternate function | |

Table D.2- Pin function select register 3

| PINSEL3 | Pin name | Function when 00 | Function when 01 | Function when 10 | Function when 11 | Reset value |
|--------------|--------------|-----------------------|------------------|------------------|------------------|-------------|
| ⋮ | | | | | | |
| 19:18 | P1.25 | GPIO Port 1.25 | MCOA1 | Reserved | MAT1.1 | 00 |
| 21:20 | P1.26 | GPIO Port 1.26 | MCOB1 | PWM1.6 | CAP0.0 | 00 |
| 23:22 | P1.27 | GPIO Port 1.27 | CLKOUT | USB_OVRCR | CAP0.1 | 00 |
| 25:24 | P1.28 | GPIO Port 1.28 | MCOA2 | PCAP1.0 | MAT0.0 | 00 |
| 27:26 | P1.29 | GPIO Port 1.29 | MCOB2 | PCAP1.1 | MAT0.1 | 00 |
| 29:28 | P1.30 | GPIO Port 1.30 | Reserved | V _{BUS} | AD0.4 | 00 |
| 31:30 | P1.31 | GPIO Port 1.31 | Reserved | SCK1 | AD0.5 | 00 |

Table D.3- Pin function select register 4

| PINSEL4 | Pin name | Function when 00 | Function when 01 | Function when 10 | Function when 11 | Reset value |
|---------|----------|---------------------|---------------------|---------------------|---------------------|----------------|
| 1:0 | P2.0 | GPIO Port 2.0 | PWM1.1 | TXD1 | Reserved | 00 |
| 3:2 | P2.1 | GPIO Port 2.1 | PWM1.2 | RXD1 | Reserved | 00 |
| 5:4 | P2.2 | GPIO Port 2.2 | PWM1.3 | CTS1 | Reserved | 00 |
| 7:6 | P2.3 | GPIO Port 2.3 | PWM1.4 | DCD1 | Reserved | 00 |
| 9:8 | P2.4 | GPIO Port 2.4 | PWM1.5 | DSR1 | Reserved | 00 |
| 11:10 | P2.5 | GPIO Port 2.5 | PWM1.6 | DTR1 | Reserved | 00 |
| 13:12 | P2.6 | GPIO Port 2.6 | PCAP1.6 | RI1 | Reserved | 00 |
| 15:14 | P2.7 | GPIO Port 2.7 | RD2 | RTS1 | Reserved | 00 |
| 17:16 | P2.8 | GPIO Port 2.8 | TD2 | TXD2 | ENET_MDC | 00 |
| ⋮ | | | | | | |

Once a pin is defined as GPIO, then the next step is to determine whether it is an input pin or an output pin. Figure D.3 shows general structure for a GPIO pin.

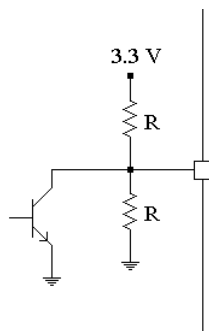


Figure D.3 – General structure of a GPIO pin

Each GPIO pin can be either input or output. It can also have a pull-up resistor, pull-down resistor, or no resistor at all.

For output configuration, the pull-up resistor can generate a high (or 1) state, and the open-collector transistor can generate a low (or 0) state when the transistor is turned on.

For input configuration, the open-collector transistor is off.

Again, there is a two bit field in register named PINMODEx that determines GPIO pin configuration. Table D.4 shows how it is done.

Table D.4 – Pin mode configuration (pull-up/pull-down resistor configuration)

| PINSEL0 to PINSEL9 values | Function | Value after reset |
|---------------------------|---------------------------------------|-------------------|
| 00 | pin has a pull-up resistor enabled | 00 |
| 01 | pin has repeater mode enabled | |
| 10 | pin has neither pull-up nor pull-down | |
| 11 | Pin has a pull-down resistor enabled | |

For more detailed information, refer to Appendix A or tables 87-93 of the **LPC17xx User manual** [1].

Schematic diagram

Now you know how ports of the LPC1768 are set. However you need to know how the LPC1768 is connected to peripherals on the MCB1700 board. A full schematic diagram of the board is given in Appendix F.

Figure D.4 shows the connections for the LEDs. It is clear that in order to be able to turn the LEDs on and off, the IC9 (74LVC244T) must be enabled. Jumper LED on the board permanently activates this buffer IC. The eight inputs of A1~A8 to IC9 are coming directly from the LPC1768. Three inputs are from port number 1, and five inputs are from port number 2.

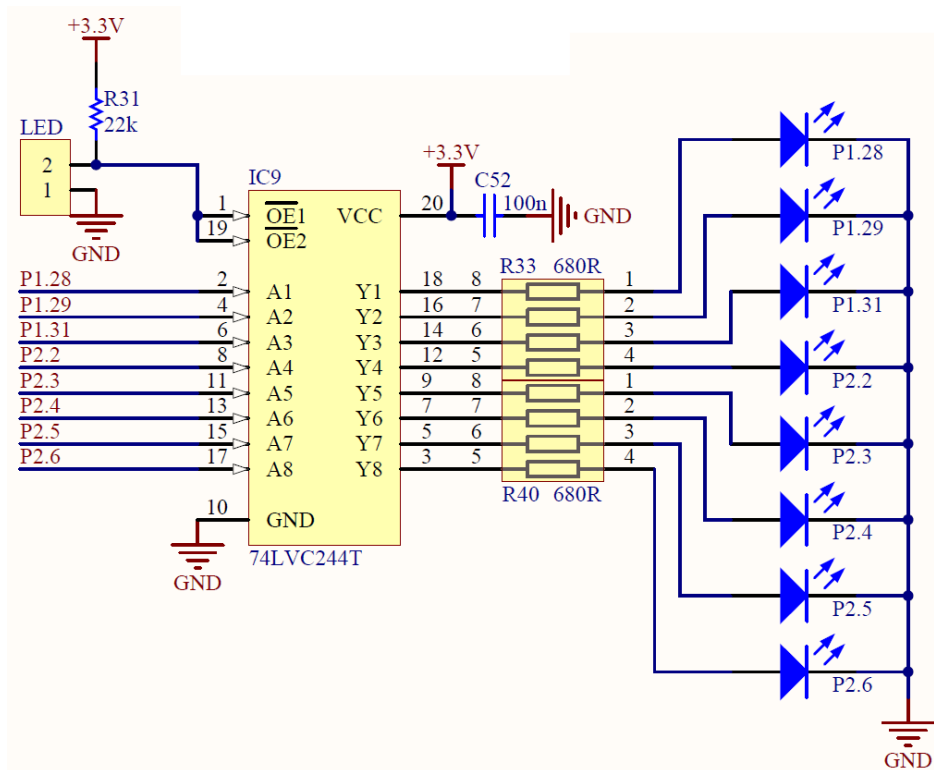


Figure D.4 – Hardware connection for on-board green LEDs [4]

In order to turn one LED on (like P1.28), a '1' must be written to the pin P1.28. similarly, writing a '0' to P1.28 pin will turn the LED off. The IC 74LVC244T is just a buffer meaning that output pins will be the same as corresponding input pins.

Figure 5 shows the schematic diagram for INTO push buttons, and figure 6 shows the Joystick schematic diagram.

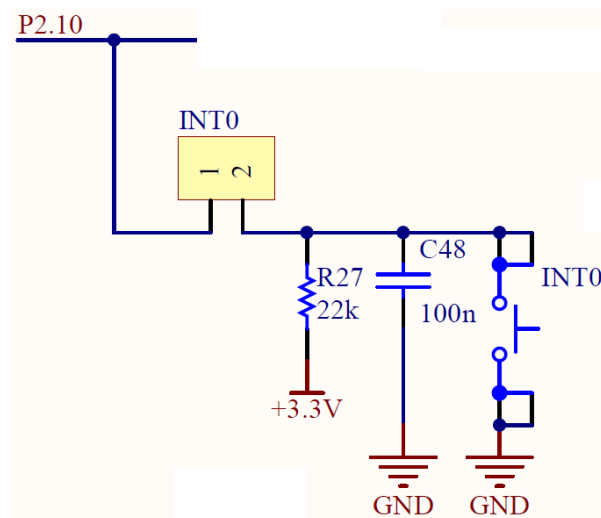


Figure D.5 – Schematic diagram for the INTO push button [4]

Appendix E: Exception and Interrupts

The LPC1768 microprocessor can have many sources of interrupt. Selected GPIO pins can also be set to generate interrupts.

The Nested Vectored Interrupt Controller (NVIC) is an integral part of the ARM Cortex-M3. In the LPC17xx microprocessors, the NVIC supports 35 vectored interrupts. The default location for the vectors is address 0x0.

The push button INT0 is connected to pin P2.10 of the LPC1768 microprocessor. This pin can be a source of an external interrupt to the MCU. Table E.1 shows different functionalities that can be assigned to P2.10 pin.

For our purpose, of generating an interrupt, there is no big difference between the three types of interrupt sources. For simplicity you may use the default function of GPIO.

Table E.1 – Pin functions for P2.10

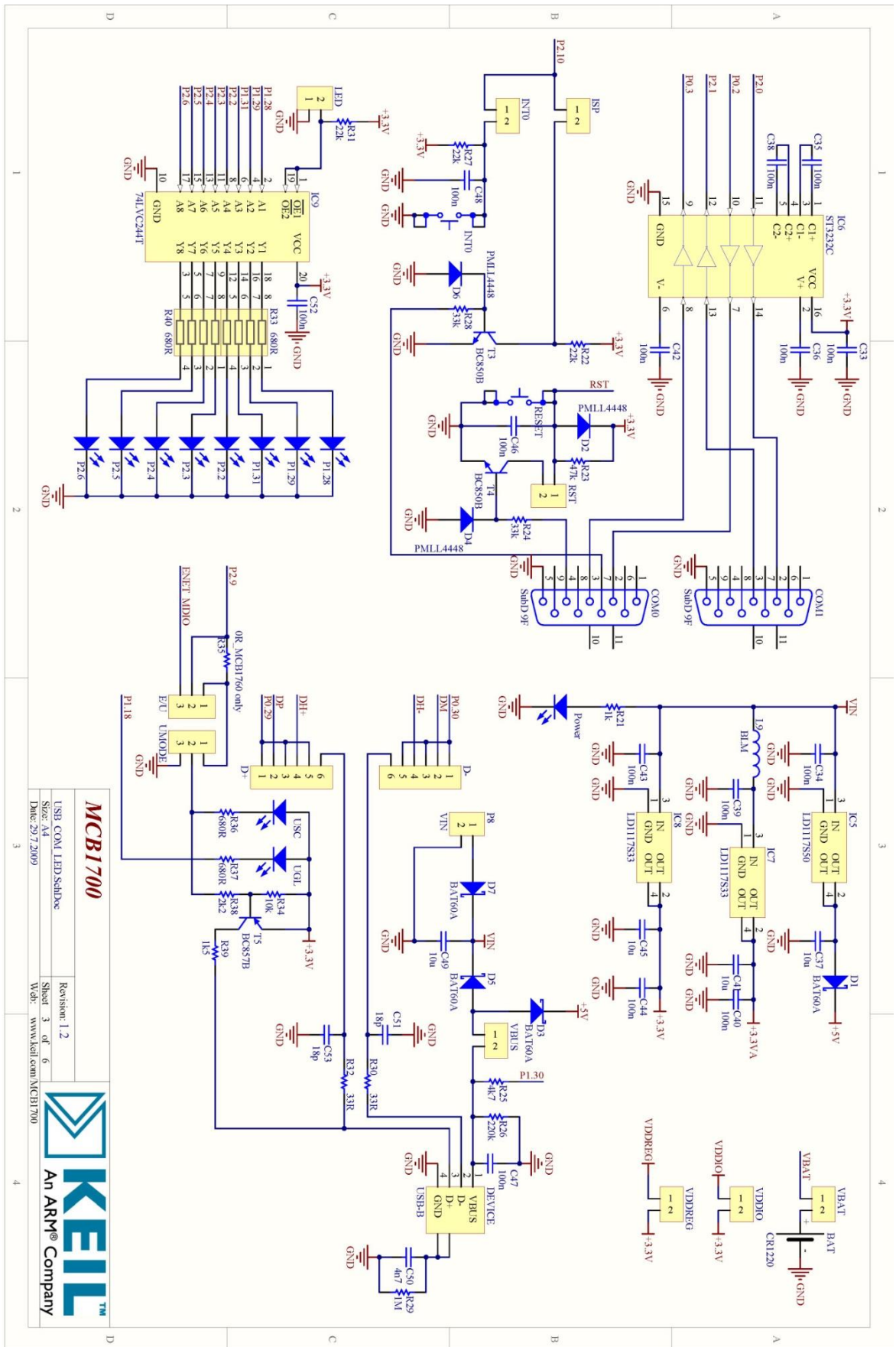
| Bits 21:20 of PINSEL4 | Function | Value after reset |
|------------------------------|--------------------------|-------------------|
| 00 | GPIO P2.10 pin (default) | 00 |
| 01 | $\overline{EINT0}$ | |
| 10 | NMI | |
| 11 | Reserved | |

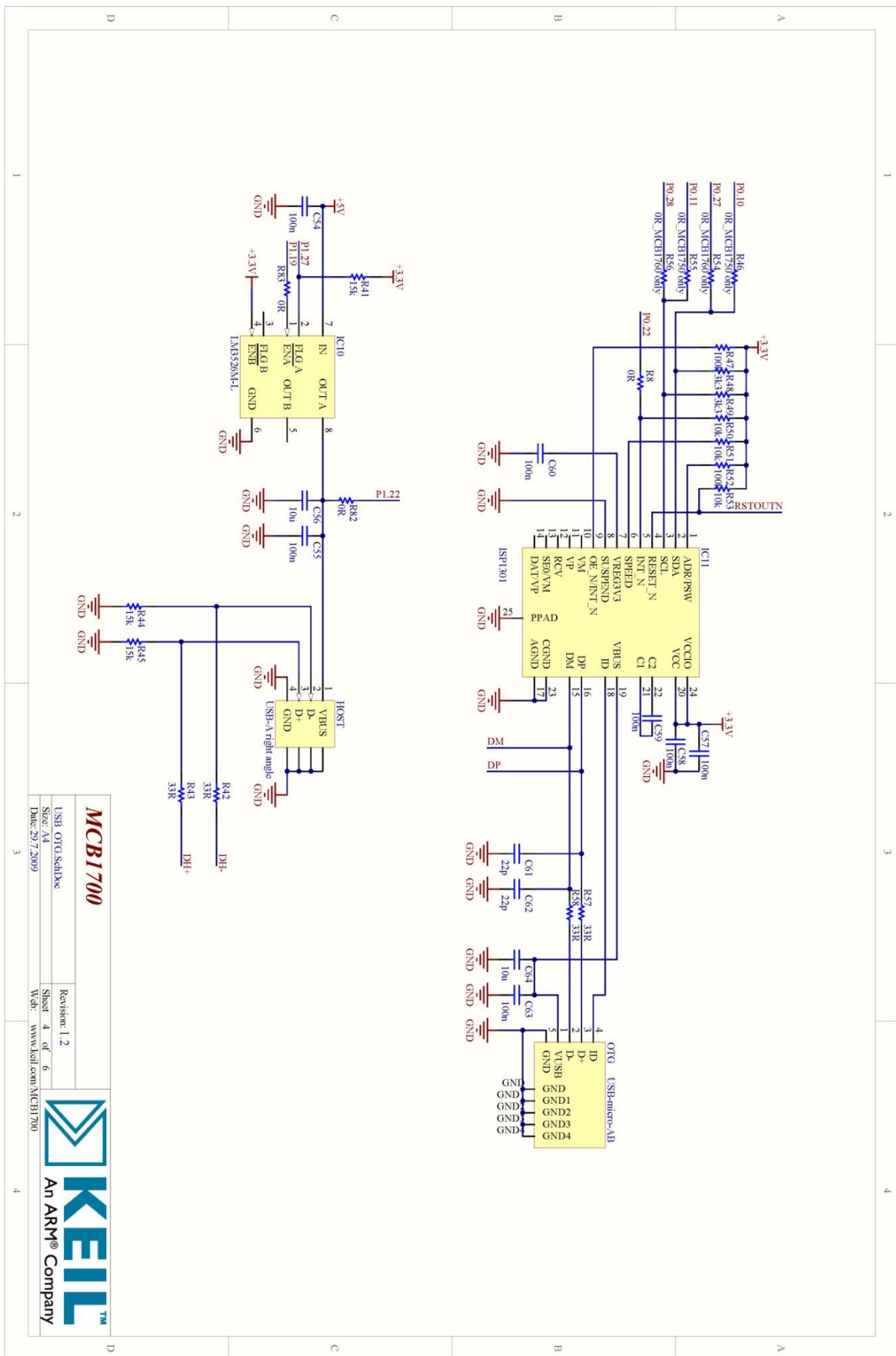
For P2.10 configured as GPIO, you should enable this source of interrupt as described in section 9.5.6 of the document **LPC17xx User manual** [1]. You can set the P2.10 pin to be sensitive to either the rising edge or the falling edge.

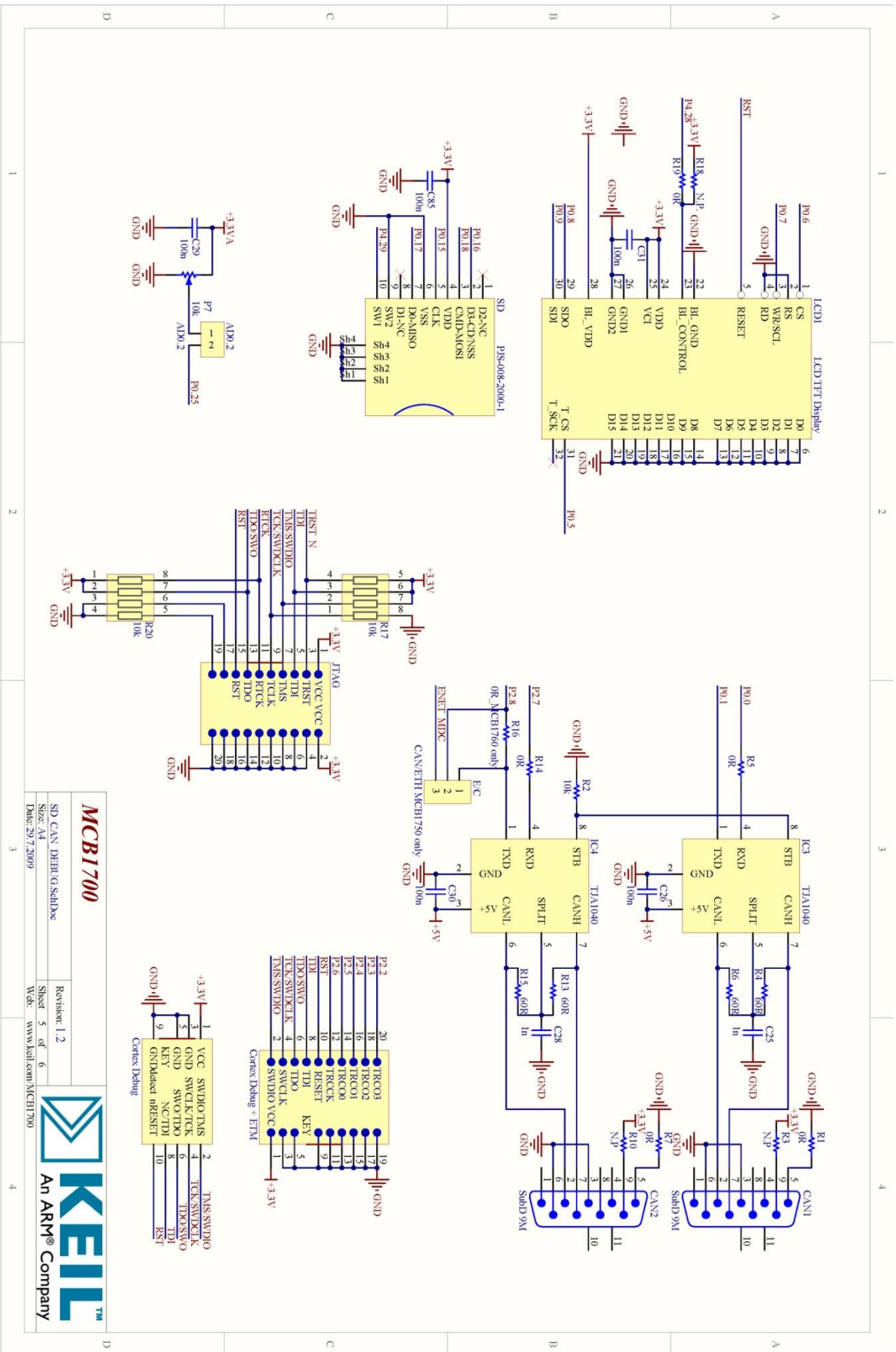
Information on clearing the interrupt pending pin can be found in table 123 in section 9.5.6.11 – page 139/840 of [1].

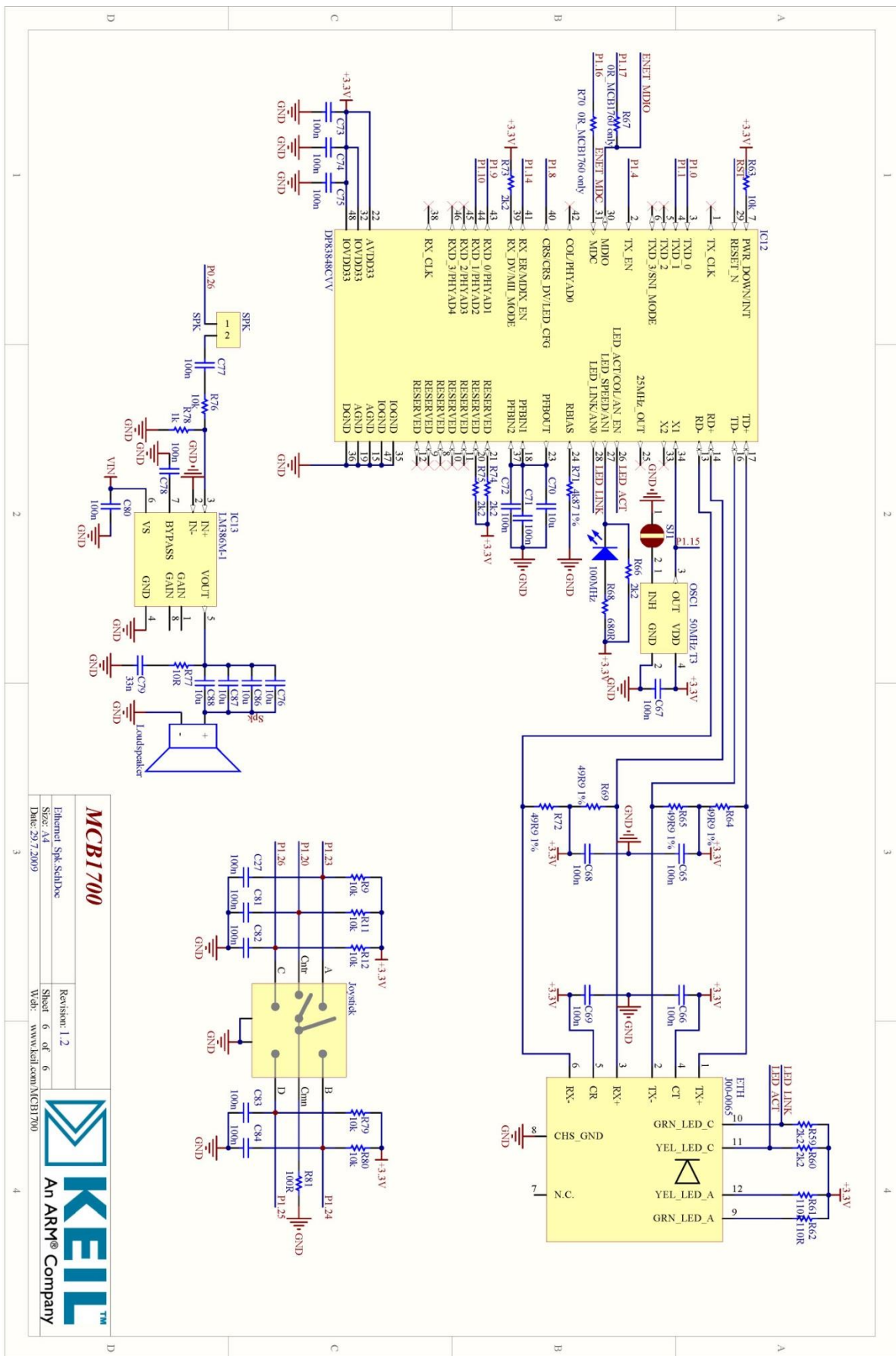
[illegible]











References:

- [1] **LPC17xx User manual**, Literature number UM10360, Rev.4.1, December 19, 2016, Published by NXP®. Can be accessed online: http://www.nxp.com/documents/user_manual/UM10360.pdf (Accessed on January 5, 2023) [You need to create an account to access this document.]
- [2] **Getting Started, Creating Applications with μ Vision®4**, Published by Keil®. Can be accessed online: <http://www.keil.com/product/brochures/uv4.pdf> (Accessed on January 5, 2023)
- [3] **ECE-455 Lab manual**, Bernie Roehl, E&CE Department, University of Waterloo. Can be accessed online: https://ece.uwaterloo.ca/~ece455/lab_manual.pdf (Accessed on January 5, 2023)
- [4] **Schematic Diagram for the MCB1700 board**, Published by Keil®. Can be accessed online: <http://www.keil.com/mcb1700/mcb1700-schematics.pdf> (Accessed on January 5, 2023)
- [5] **Snap-shots** taken from the Keil μ Vision4 software.
- [6] **“Morse Code”**, From Wikipedia, the free encyclopedia. Can be accessed online: https://en.wikipedia.org/wiki/Morse_code (Accessed on January 5, 2023)
- [7] **Cortex™-M3 Devices**, Generic User Guide, Published by ARM. Can be accessed online: http://infocenter.arm.com/help/topic/com.arm.doc.dui0552a/DUI0552A_cortex_m3_dgug.pdf (Accessed on January 5, 2023)

Appendix G: Hand Assembly

ARM7500 (1995 vintage) instructions are 32-bits long and are documented here:

http://infocenter.arm.com/help/topic/com.arm.doc.ddi0050c/DDI0050C_7500_ds.pdf This is the information needed to hand assemble an arithmetic instruction. Note that this is a partial table of what can be hand assembled. For instance the MOV instruction also allows a 16-bit value to be transferred into a register but that is not covered here.

| Bit Position | 31-28 | 27-26 | 25 | 24-21 | 20 | 19-16 | 15-12 | 11-0 |
|--------------|-----------|-------|----|---------|----|-------|-------|-----------|
| Bit Content | Condition | 0 0 | RI | OP code | S | Rn | Rd | Operand 2 |

Here is an abbreviated table of the possible conditions:

| Condition Bits | Condition suffix | Name | Condition Tested |
|----------------|------------------|----------------------------------|------------------|
| 0000 | EQ | Equal to zero | Z = 1 |
| 0001 | NE | Not equal to 0 | Z = 0 |
| 0010 | CS | Carry Set | C = 1 |
| 0011 | CC | Carry Clear | C = 0 |
| 0100 | MI | Minus (negative) | N = 1 |
| 0101 | PL | Plus (positive or zero) | N = 0 |
| 1110 | AL | Always [default] | |
| 1111 | NV | Never [no-operation instruction] | |

Bit 25 “RI” sets the twelve “Operand 2” bits to Register or Immediate value. Assume that shift and rotate operations are to the left. 0xF1 can be rotated upto 16 times to give 0x00f10000

| Bit 25 “RI” | Second Operand | First field | Second field |
|-------------|-----------------|---------------------------|----------------------------|
| 0 | Register | Bits 11-4 (Shift amount) | Bits 3-0 (Register #) |
| 1 | Immediate Value | Bits 11-8 (Rotate amount) | Bits 7-0 (Immediate value) |

| Mnemonic | Op code | | Mnemonic | Op code | | Mnemonic | Op code |
|----------|---------|--|----------|---------|--|----------|---------|
| ADD | 0100 | | EOR | 0001 | | ORR | 1100 |
| ADC | 0101 | | CMP | 1010 | | AND | 0000 |
| SUB | 0010 | | TEQ | 1001 | | MOV | 1101 |
| SBC | 0110 | | TST | 1000 | | MVN | 1111 |

The S bit is 1 if the condition flags are to be set, 0 otherwise.

Rn is the operand register. These bits go to a multiplexer to choose one of the 16 registers.

Rd is the destination register. These 4-bits drive a multiplexer to choose the destination register.

The second operand is either a register or immediate value depending upon bit 25 "RI". An unsigned 8-bit immediate value can be rotated or a register can be shifted.

Hand assembling ORREQ R3, R2, R5 gives:

- ORR R2 and R5 and store the result in R3 but ONLY if the last operation to update the Z bit set it to 1 - that is to say that the result of the operation updating the status bits was EQUAL to zero

Going thru the tables to get the bits:

Condition is 0000 for EQ

I (now called RI to make it more readable) is "0" because the 2nd operand is a register and not an immediate value.

The Op code is 1100 for ORR.

The S bit is 0 because the status bits are not being set by the ORR operation (that would be ORRS).

Rn is the operand for R2

Rd is the destination or R3

R5 is the second operand and it is not being shifted.

This gives us the bit pattern (broken down into the sections given in the Bit Position table):

0000 00 0 1100 0 0010 0011 00000000 0101

| | | | | | |
|-------|-------|----|----|-------|---------------------|
| Cond. | Opcod | Rn | Rd | shift | R5 (second operand) |
|-------|-------|----|----|-------|---------------------|

Slice into 4 bit pieces and convert to hex gives:

0x01823005

University Expectations and Policies

The following statements represent university expectations and policies with respect to academic integrity, discipline, grievance, student appeals, and academic accommodations. If you would like more clarification, please contact your course instructor directly.

Academic Integrity

In order to maintain a culture of academic integrity, members of the University of Waterloo community are expected to promote honesty, trust, fairness, respect and responsibility. [Check uwaterloo.ca/academic-integrity for more information.]

Grievance

A student who believes that a decision affecting some aspect of his/her university life has been unfair or unreasonable may have grounds for initiating a grievance. Read Policy 70, Student Petitions and Grievances, Section 4, uwaterloo.ca/secretariat/policies-procedures-guidelines/policy-70. When in doubt please be certain to contact the department's administrative assistant who will provide further assistance.

Discipline

A student is expected to know what constitutes academic integrity [check uwaterloo.ca/academic-integrity] to avoid committing an academic offence, and to take responsibility for his/her actions. A student who is unsure whether an action constitutes an offence, or who needs help in learning how to avoid offences (e.g., plagiarism, cheating) or about "rules" for group work/collaboration should seek guidance from the course instructor, academic advisor, or the undergraduate Associate Dean. For information on categories of offences and types of penalties, students should refer to Policy 71, Student Discipline, uwaterloo.ca/secretariat/policies-procedures-guidelines/policy-71. For typical penalties check Guidelines for the Assessment of Penalties, uwaterloo.ca/secretariat/policies-procedures-guidelines/guidelines/guidelines-assessment-penalties.

Appeals

A decision made or penalty imposed under Policy 70 (Student Petitions and Grievances) (other than a petition) or Policy 71 (Student Discipline) may be appealed if there is a ground. A student who believes he/she has a ground for an appeal should refer to Policy 72 (Student Appeals) uwaterloo.ca/secretariat/policies-procedures-guidelines/policy-72.

Note for Students with Disabilities

Access Ability Services, located in Needles Hall, Room 1132, collaborates with all academic departments to arrange appropriate accommodations for students with disabilities without compromising the academic integrity of the curriculum. If you require academic accommodations to lessen the impact of your disability, please register with Access Ability Services at the beginning of each academic term.