Embedded Systems Lab Manual

Ahmad Khayyat, Hazem Selmi, Mohannad Mostafa, Saleh AlSaleh, Mansour Alharthi

Version 162, 24 April 2017

Table of Contents

1. Experiment 1: Development Platform

Ahmad Khayyat; Hazem Selmi; Saleh AlSaleh 162, 13 February 2017

1.1. Objectives

- Get familiar with the development platform:

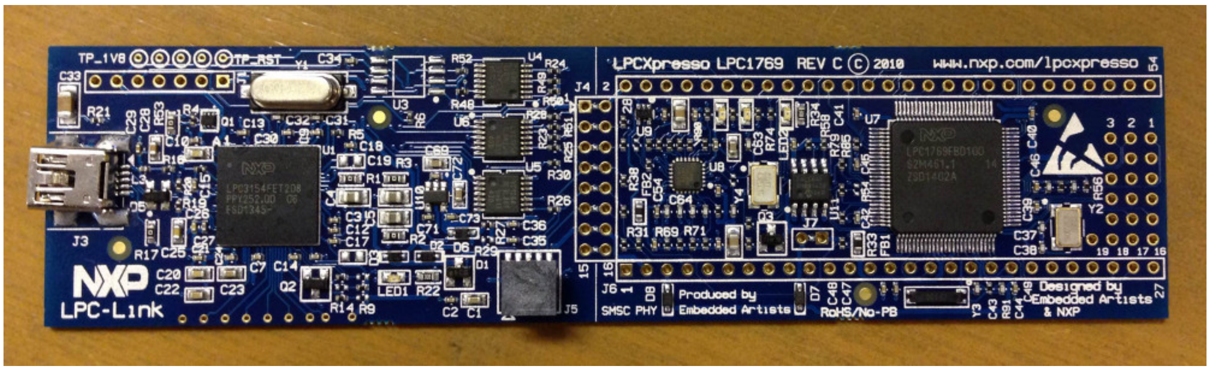
  Hardware
  : microcontroller, development board, peripherals

  Software
  : IDE, compiler, debugging, programming

- General-purpose input/output (GPIO): digital output
- Introduce CMSIS: Cortex Microcontroller Software Interface Standard

1.2. Parts List

- LPC1769 LPCXpresso board



- USB A-Type to Mini-B cable

1.3. Background

1.3.1. Microcontroller

LPC1769 is a microcontroller manufactured by NXP. LPC1769 is only one member of a big family of microcontolles. NXP was founded by Philips as Philips Semiconductors, and renamed NXP in 2006. A major difference between a microcontroller and a microprocessor is that the former has some additional built-in devices, such as memory, I/O peripherals, and timers.

The LPC1769 microcontroller is an ARM 32-bit Cortex-M3 microcontroller. Some of its features include: CPU clock up to 120MHz, 512kB on-chip Flash ROM, 64kB RAM, Ethernet 10/100 MAC, USB 2.0 full-speed Device controller and Host controller, four UARTs, general purpose I/O pins, 12-bit ADC, 10-bit DAC, four 32-bit timers, Real Time Clock, System Tick Timer.

The product data sheet for the LPC1769 microcontroller [lpc1769-data-sheet] and the more detailed user munual [lpc1769-manual] are essential resources for any developer.

1.3.2. Development Board

The LPC1769 microcontroller chip is used to build the *LPC1769 LPCXpresso board*, which we will be using in this LAB.

The LPCXpresso board consists of two parts:

1. LPCXpresso target board, which hosts the LPC1769 microcontroller
2. LPC-Link: a debug probe for debugging and the target microcontroller.

1.3.3. LPCXpresso IDE

The LPCXpresso IDE is an Eclipse-based software development environment for NXP’s LPC microcontrollers.

The LPCXpresso IDE uses the GNU toolchain (compiler and linker), and offers the choice of two C libraries:

- Redlib (default): a proprietary ISO C90 standard C library, with some C99 extensions. Often results in smaller binary size.
- Newlib: an open source complete C99 and C++ library.

The LPCXpresso IDE is available in two editions:

- Free edition: requires free activation (details below). Supports code sizes up to 256 kB after activation.
- Pro edition: per-seat licensing fees. Supports unlimited code size.

Installation

To install your copy of the LPCXpresso IDE:

1. Using a web browser, navigate to the LPC Microcontroller Utilities page, then follow the LPCXpresso IDE link.
2. Use the *Download* button to go to the download page, then use the download link that matches your operating system.

   You may need to create an account to access the download page.

3. Run the downloaded installer.

Activation

To activate your copy of the LPCXpresso IDE:

1. Using a web browser, log into the activation web page:
   https://lpcxpresso-activation.nxp.com/registerapp.html

   Register a new account if you do not have one.

2. Follow the instructions on that page to find and submit your LPCXpresso IDE serial number, and to obtain and use your activation code.

For more information on installing and using the LPCXpresso IDE, see [lpcxpresso-ide-user-guide].

1.3.4. Input/Output Ports

The LPC1769 microcontroller has five input/output (I/O) ports. Each port is 32 bits. However, not all of them are available for the developer. For example, pins 12, 13, 14, and 31 of port 0 are not available, leaving only 28 pins of port 0 available for the user.

Each of the I/O pins can be referred to using the port number and the pin number. For example, `P0.17` or `P0[17]` is pin 17 in port 0, and `P1.22` or `P1[22]` is pin 22 in port 1.

Most of the I/O pins have multiple functions. For example: `P0.10` can perform one of these jobs:

P0[10]
: General purpose digital input/output pin.

TXD2
: Transmitter output for UART2.

SDA2
: I2C2 data input/output.

MAT3[0]
: Match output for Timer 3, channel 0.

Don’t worry if you don’t understand these functions, you will learn about them throughout the course.

A command is needed to choose which function is to be used in a specific pin. The only exception is the first function (GPIO) because it is the default function.

Relying on a default value may be acceptable in simple programs. However, a good programming style when you have many functions and interrupts is not to assume any default value as they may have been changed somewhere in your program. Instead, you should explicitly specify any desired values.

Pin Layout

The pin layout of the LPC1769 LPCXpresso board is shown in the LPC1769 pin layout figure below (source: [lpc1769-schematic], page 7).

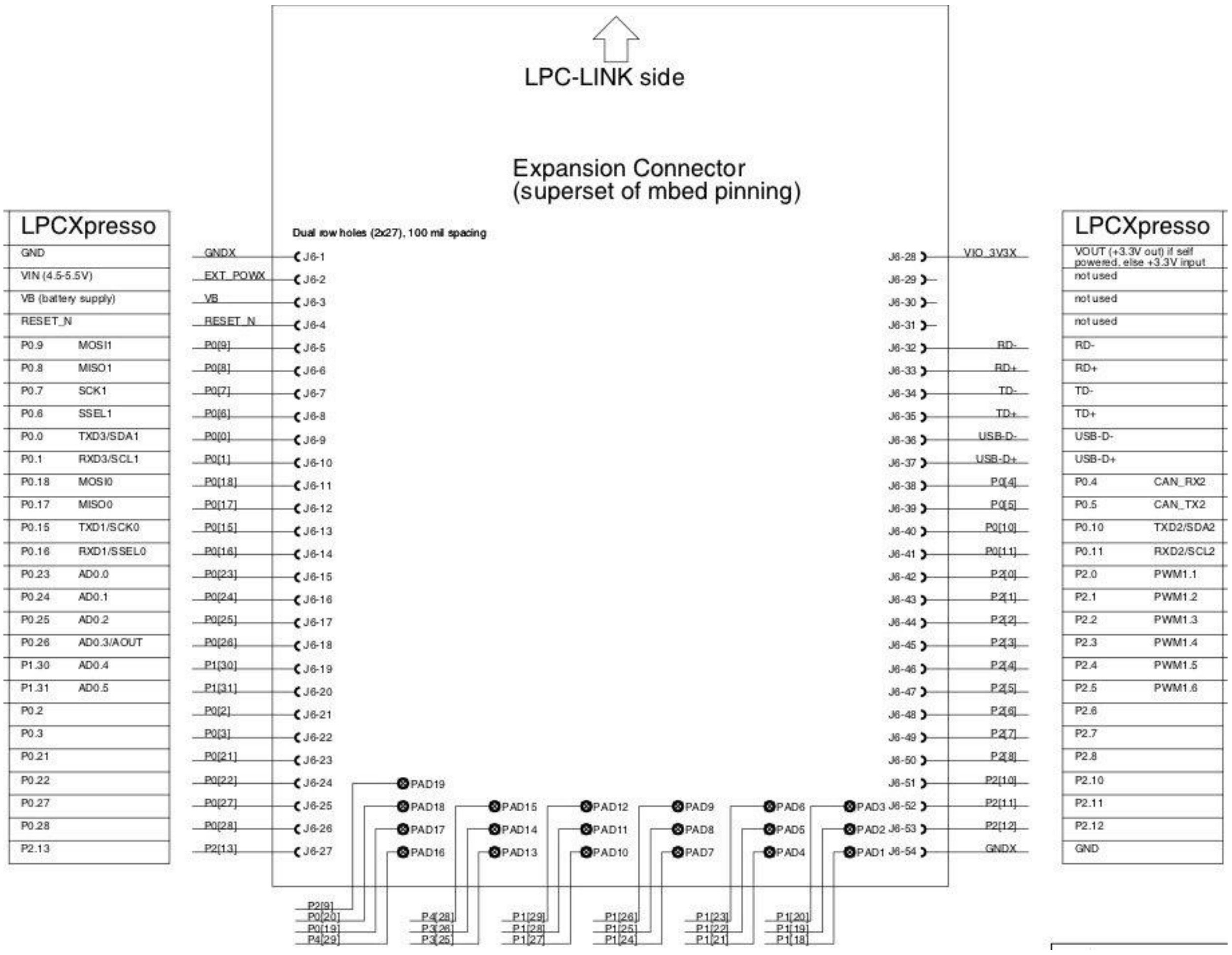


Figure 1. LPC1769 pin layout

Memory-Mapped I/O

ARM uses memory-mapped I/O. When using memory-mapped I/O, the same address space is shared by memory and I/O devices. Some addresses represent memory locations, while others represent registers in I/O devices. No separate I/O instructions are needed in a CPU that uses memory-mapped I/O. Instead, we can use any instruction that can reference memory to move values to or from memory-mapped device registers.

General-Purpose Input/Output (GPIO)

GPIO is available in most I/O pins. A GPIO pin is a pin that can be used for digital input or digital output. You need to choose the direction of the pin (whether it is used for input or output). In the first example of this experiment, we will set the direction to be output. To use a digital output pin, you need to be able to to *set* the output to HIGH (1), and to *clear* it to LOW (0).

In summery, we need to learn about 3 registers for our first experiment:

1. The register that controls the *direction* of GPIO pins
2. How to *set* a pin to HIGH.
3. How to *clear* a pin to LOW.

Accessing Registers

Each I/O register has an address. For example:

1. The address of the register that controls the direction of port 0 pins is: `0x2009c000`.
2. The address of the register that *sets* port 0 pins to HIGH is: `0x2009c018`.

3. The address of the register that *clears* port 0 pins to LOW is: `0x2009c01c`.

To access a register more easily, you can give it a name. One way to give a register a name in the C programming language is to use *pointers*, *pointer dereferencing*, and the `define` directive.

For more details about these features (and more) of the *C programming language*, it is strongly recommended to consult the document Data Structures in C.

Example 1. Giving Registers Names

Here are three examples showing how to assign names to registers:

```
#define DIR_P0  (*((volatile unsigned long *) 0x2009c000))
#define SET_P0  (*((volatile unsigned long *) 0x2009c018))
#define CLR_P0  (*((volatile unsigned long *) 0x2009c01c))
```

Example 2. Setting Pin Direction

To set the direction for pins 1, 2, 3 and 4 of port 0 as output, while setting the direction of the remaining pins as input:

```
DIR_P0 = 0x0000001E;
// OR
DIR_P0 = 30;

// Make sure that you understand that these statements are equivalent!
```

The first task is to blink an LED using the above three registers!

In the first experiment, you can avoid making any external connections by using the on board LED, which is connected to `P0.22`.

1.3.5. CMSIS

> The *Cortex Microcontroller Software Interface Standard* (CMSIS) is a vendor-independent hardware abstraction layer for the Cortex-M processor series [...]. The CMSIS enables consistent device support and simple software interfaces to the processor and the peripherals, simplifying software re-use [...].
>
> — ARM Ltd., CMSIS: Introduction

The CMSIS components are: CMSIS-CORE, CMSIS-Driver, CMSIS-DSP, CMSIS-RTOS API, CMSIS-Pack, CMSIS-SVD, CMSIS-DAP, CMSIS-DAP [cmsis-intro].

The most relevant component to us is CMSIS-CORE [cmsis-core].

> CMSIS-CORE implements the basic run-time system for a Cortex-M device and gives the user access to the processor core and the device peripherals. In detail it defines:
>
> - **Hardware Abstraction Layer (HAL)** for Cortex-M processor registers with standardized definitions for the SysTick, NVIC, System Control Block registers, MPU registers, FPU registers, and core access functions.
> - **System exception names** to interface to system exceptions without having compatibility issues.
> - **Methods to organize header files** that makes it easy to learn new Cortex-M microcontroller products and improve software portability. This includes naming conventions for device-specific interrupts.
> - **Methods for system initialization** to be used by each MCU vendor. For example, the standardized `SystemInit()` function is essential for configuring the clock system of the device.
> - **Intrinsic functions** used to generate CPU instructions that are not supported by standard C functions.
> - A variable to determine the **system clock frequency** which simplifies the setup the SysTick timer.
>
> — ARM Ltd., CMSIS-CORE: Overview

CMSIS provides abstraction at the chip level only. Other libraries provide more extensive APIs for additional peripherals and board features, but are usually less generic and more vendor-specific [lpcx-cmsis].

1.3.6. Accessing Registers Using CMSIS

When using CMSIS, you don't need to know register addresses, which implies that you don't need to use the `#define` directive to name the registers. Instead, you use the `#include` directive to include the `lpc17xx.h` header file, which contains all the register address definitions for the LPC17xx family of microcontrollers. When you use the LPCXpresso IDE to create a CMSIS project, the IDE generates a basic source file which already includes this header file.

In the `lpc17xx.h` header file, the names are not given using the `#define` directive only. They are given using `#define` (for the base address) then using structures to group similar (and adjecent) registers.

Structures and Pointers

The CMSIS header file, `lpc17xx.h`, organizes the registers into logical groups based on their functions, using C structures. First, a structure is defined by listing its fields. Then, a pointer is defined for each needed instance of that structure, pointing to the starting address of the instance, as documented in the microcontroller manual.

For example, the names of the pointers to the structure instances for the five GPIO ports are:

`LPC_GPIO0`
    for port 0

`LPC_GPIO1`
    for port 1

`LPC_GPIO2`
: for port 2

`LPC_GPIO3`
: for port 3

`LPC_GPIO4`
: for port 4

These pointers are defined in the `lpc17xx.h` file as follows:

```
#define LPC_GPIO0  ((LPC_GPIO_TypeDef *)  LPC_GPIO0_BASE)
#define LPC_GPIO1  ((LPC_GPIO_TypeDef *)  LPC_GPIO1_BASE)
#define LPC_GPIO2  ((LPC_GPIO_TypeDef *)  LPC_GPIO2_BASE)
#define LPC_GPIO3  ((LPC_GPIO_TypeDef *)  LPC_GPIO3_BASE)
#define LPC_GPIO4  ((LPC_GPIO_TypeDef *)  LPC_GPIO4_BASE)
```

where `LPC_GPIO_TypeDef` is the name of the structure, which is defined earlier in the file to describe the registers related to GPIO ports, and `LPC_GPIO0_BASE` through `LPC_GPIO4_BASE` are fixed addresses, also defined earlier in the header file, at which the registers for each port start. Other structures are also defined for registers related to functions other than GPIO.

Fields are Registers

For each instance of a structure, such as `LPC_GPIO0`, you can access a register by accessing the corresponding field in that structure instance. For example, the three registers used in Experiment 1 are defined in the aforementioned `LPC_GPIO_TypeDef` structure as the following fields:

1. `FIODIR`
2. `FIOSET`
3. `FIOCLR`

Each of these registers is accessible within the structure instance of each port.

Therefore, when using CMSIS, you need to know two names to access a register:

1. The name of the pointer to the structure instance.
2. The name of the field within the structure, corresponding to the desired register.

Example 3. Setting Pin Directions and Values

- To set the direction of pins 3,4, 5, and 6 in port 2 as output (and set the remaining pins as input):

```
LPC_GPIO2->FIODIR = 0x00000078;
```

- To set pins 3 and 7 in port 1 while clearing the rest of the pins, use:

```
LPC_GPIO1->FIOSET = 0x00000088;
```

Again, to learn more about structures and pointers in the *C programming language*, refer to the Data Structures in C document.

1.3.7. LEDs

- What is an LED?
- How does an LED work?
- What is the maximum voltage that an LED can tolerate?
- If the output voltage is higher than the LED maximum voltage, what should you do?

An LED should be connected to an output GPIO pin.

GPIO, Revisited

The GPIO mode is available in all I/O pins. A GPIO pin is one that can be used as a digital input or digital output. Obviously, you need to choose the direction of the pin to determine whether it is going to be used as input or output. In this experiment, we will choose the direction to make the required pin work as GPO (General-Purpose Output). In this case (GPO), you need a command to set this output pin to HIGH (1), and a command to Clear it to LOW (0).

A `0` in a *SET* or a *CLR* register has no effect on the port pins!

A basic way to add *delay* is to use a `for` loop, e.g.: `for(i=0;i<500000,i++);`. You will learn about more sophisticated and accurate ways in later experiments.

1.4. Tasks

1.4.1. Create a Non-CMSIS Project

1. Click *Quickstart Panel* > *New project…*.
2. Choose *LPC13 / LPC15 / LPC17 / LPC18* > *LPC175x_6x* > *C Project*.
3. Choose a project name, e.g. `blinky`.
4. In the *Target selection* dialog, choose *LPC1700* > *LPC1769*.
5. In the *CMSIS Library Project Selection* dialog, set *CMSIS Core library to link project to* to `None`.
6. In the *CMSIS DSP Library Project Selection* dialog, set *CMSIS DSP Library to link project to* to `None`.
7. Uncheck *Enable linker support for CRP*, then click *Finish*.
8. Open the main source file named after the project, and write your `main` function.

1.4.2. Blink an LED without CMSIS

1. Figure out which pin is connected to the LED.

   Refer to the LPC1769 board documentation.

2. Give the required registers some friendly names using the `#define` directive.
3. In an infinite loop inside the `main` function:
   a. Set the pin to act as output by setting the correct bit in the direction register to `1`.
   b. Set the output pin to `1`.
   c. Clear the output pin (set to `0`).
   d. Insert a delay loop after both set and clear, to be able to see the LED blink.
4. Which value of the pin turns the LED on, and which value turns it off? and why?

1.4.3. Import the CMSIS Libraries

1. Click *Quickstart Panel* > *Import project(s)*
2. In the *Project archive (zip)* dialog, click *Browse* next to the *Archive* field, and choose:

```
C:\nxp\LPCXpresso_<version>\lpcxpresso\Examples\NXP\LPC1000\LPC17xx\LPC17xx_LatestCMSIS_Librarie
s.zip
```

3. Keep both projects selected: `CMSIS_CORE_LPC17xx` and `CMSIS_DSPLIB_CM3`, and click *Finish*.

1.4.4. Create a CMSIS Project

To create a project that uses CMSIS, follow the same instructions for creating a non-CMSIS project up to the *CMSIS Library Project Selection* dialog. Instead of `None`, select `CMSIS_CORE_LPC17xx`.

1.4.5. Blink an LED Using CMSIS

Using a CMSIS project, rewrite your LED blinking program to use CMSIS facilities.

1.4.6. Debug Your Project

1. Click *Quickstart Panel* > *Build 'cmsis_blinky' [Debug]* to build the project.
2. Connect the LPC1769 board to the PC using the USB cable.
3. Click *Quickstart Panel* > *Debug 'cmsis_blinky' [Debug]* to debug the project interactively on the target board.

> *Running the Debugger*
>
> You can run the debugger using any of the following three ways:
>
> 1. In the *Quickstart Panel* at the lower left corner, click *Debug '<project-name>' [Debug]*.
> 2. In the main menu, choose *Run* > *Debug As* > *C/C++ (NXP Semiconductors) MCU Application*.
> 3. In the toolbar, click on the debug button ▾.

4. Once the debugger starts, it will pause execution at the first statement in the progrm. Resume execution by hitting the `F8` key, or using the resume button in the toolbar ▶.

1.5. Resources

[]

NXP Semiconductors. *LPC1769/68/67/66/65/64/63 — Product data sheet*. Rev. 9.5. 24 June 2014. http://www.nxp.com/documents/data_sheet/LPC1769_68_67_66_65_64_63.pdf

[]

NXP Semiconductors. *UM10360 — LPC176x/5x User Manual*. Rev. 4.1. 19 December 2016. http://www.nxp.com/documents/user_manual/UM10360.pdf

[]

NXP Semiconductors. *LPCXpresso IDE User Guide*. Rev. 8.1. 31 May 2016. http://www.nxp.com/assets/documents/data/en/user-guides/LPCXpresso_IDE_User_Guide.pdf

[]

Embedded Artists AB. *Board Schematics for current LPC1769 board — LPCXpresso LPC1769 rev B*. 11 February 2011. http://www.embeddedartists.com/sites/default/files/docs/schematics/LPCXpressoLPC1769revB.pdf

[]

Data Structures in C. http://www.ccse.kfupm.edu.sa/~akhayyat/files/coe306-162/lab-manual/data-structures-in-c.html

1.6. Grading Sheet

Task	Points
Blink an LED without CMSIS	4
Blink an LED using CMSIS	4
Debug your project	2

2. Experiment 2: General-Purpose Input/Output (GPIO)

Ahmad Khayyat; Hazem Selmi; Saleh AlSaleh 162, 17 February 2017

2.1. Objectives

- Using GPIO pins as input and output
- Interfacing with external LEDs, switches, and push-buttons
- Bit manipulation in C

2.2. Parts List

- LPC1769 LPCXpresso board
- USB A-Type to Mini-B cable
- Breadboard
- LEDs
- 330-Ohm Resistors
- Jumper wires

2.3. Background

2.3.1. Bit Manipulation in C

The core of embedded system programming is setting (or clearing) specific bits in different registers inside the microcontroller. This highlights the importance of *bit manipulation* as a programming skill.

Most modern architectures are byte-addressable: the smallest unit of data is the byte. Nonetheless, it is possible to operate on individual bits by clever use of bitwise operators.

Bitwise Operators

Bitwise operators apply to each bit of their operands.

Operator	Function	Examples

`&`	Bitwise AND	`0011 & 0101 = 0001`
		`   3 &    5 =    1`

`|`	Bitwise OR	`0011 | 0101 = 0111`
		`   3 |    5 =    7`

`^`	Bitwise XOR	`0011 ^ 0101 = 0110`
		`   3 ^    5 =    6`

`~`	Bitwise NOT	`~00110101 = 11001010`
`<<`	Shift left	`3 << 2 = 12`
`>>`	Shift right	`8 >> 2 = 2`

In C, numbers can be written in decimal, octal, or hexadecimal, but not in binary, e.g. `16 = 020 = 0x10`.

Right-shifting in C is implementation-specific. Often, *logical shifting* is applied to unsigned values, whereas *arithmetic shifting* is applied to signed values.

Masking

A simple assignment to a 32-bit register or memory location will overwrite all 32 bits. However, manipulating specific bits implies that the remaining bits in the register remain intact. An essential technique to achieve that is bit masking.

A mask is a value that can be used in a binary, i.e. two-operand, bitwise operation to change specific bits of some other value. Masking relies on the following rules of Boolean Algebra:

- ANDing a bit with a `0` results in a `0`. ANDing a bit with a `1` results in the same bit.
- ORing a bit with a `0` results in the same bit. ORing a bit with a `1` results in a `1`.
- XORing a bit with a `0` results in the same value. XORing a bit with a `1` inverts the bit.

Exercises

1. What mask and bitwise operation are required to set the third least significant bit (bit 2) to `1` without affecting the other bits in a 32-bit variable `x`?
2. What mask and bitwise operation are required to reset bit 10 of a 16-bit variable `y` to `0`?
3. What mask and bitwise operation are required to toggle bit 20 of a 32-bit variable `z`?

Creating Masks by Shifting

If you have worked out the exercises above, you would have noticed that spelling out masks can be tedious, verbose, and error-prone. One trick that makes it easier to create masks is to use the shift operations. For example, to create a mask whose bit 10 is `1` and whose other bits are `0`, you can use the following C statement:

```c
mask = 1 << 10;
```

Exercise

Repeat the three exercises above by using shift operations to create the masks.

2.3.2. Digital Input

A GPIO pins can be configured to act as a general-purpose input pin by setting the corresponding bit in the `FIODIR` register to `0`. A digital input pin is *digital* because it is *driven* by an external digital device that has only two states (HIGH or LOW); and it is *input* because its state is *read* by the microcontroller. That implies that some external device/circuit is needed to generate that digital input value (HIGH or LOW).

Examples for simple digital input devices include switches and push-buttons.

A common mistake is to forget about or misuse the `FIODIR` register.

The `FIOPIN` Register

In addition to the three registers used in Experiment 1 (`FIODIR`, `FIOSET`, and `FIOCLR`), there are a few additional GPIO-related registers. The one that is particularly essential for reading from a digital input peripheral is `FIOPIN`.

This register is a R/W register that stores the current state of a port's pins. In other words, you can write to `FIOPIN` to set and clear pins of a specific port. You can also read the state of port pins. `FIOPEN` is essential for the read operation, but since it is a R/W register, it can also be used with output pins. For instance, you can redo Experiment 1 using `FIOPIN` only instead of `FIOSET` and `FIOCLR`.

There is an `FIOPIN` register for each one of the five I/O ports, and it can be accessed in the same way `FIOSET` and `FIOCLR` are. For example, port 1's `FIOPIN` register can be accessed using:

```
LPC_GPIO1->FIOPIN
```

Example 4. Using the `FIOPIN` Register

To set the third bit of port 0, i.e. `P0[2]`:

```
LPC_GPIO0->FIODIR |= (1 << 2);  // configure the pin for output
LPC_GPIO0->FIOPIN |= (1 << 2);  // set the the pin value to high or one
LPC_GPIO0->FIOPIN &= ~(1 << 2); // clear the pin value, set its value to be low or zero
```

2.3.3. Debugging

The LPCXpresso IDE along with LPC-Link hardware provide the ability to step through the code by executing one statement or instruction at a time. This helps find which line in the code causes some errors or invalid values.

To step through the program statements or instructions, run it by pressing `F6` instead of `F8`. This will execute the code one statement at a time.

You can add a breakpoint to a statement and the debugger will stop at that statement only.

You can learn more about LPCXpresso's debugging support by referring to the [lpcxpresso-ide-user-guide].

In particular, you may want to check out section *3.4.2 Controlling Execution*, which lists all the possible ways to step through your code, such as stepping into and stepping over functions.

Inspect Variable Values at Runtime

After uploading a program to the microcontroller, start debugging it by stepping through the statements or by adding a breakpoint. Now, you can get the value of any variable simply by hovering the mouse over the variable in the code. A window will be shown containing details about the variables such as type and value.

This can also be used for registers. For example, you can use it to find the value of `LPC_GPIO0 → FIOPIN`.

Print Variable Values

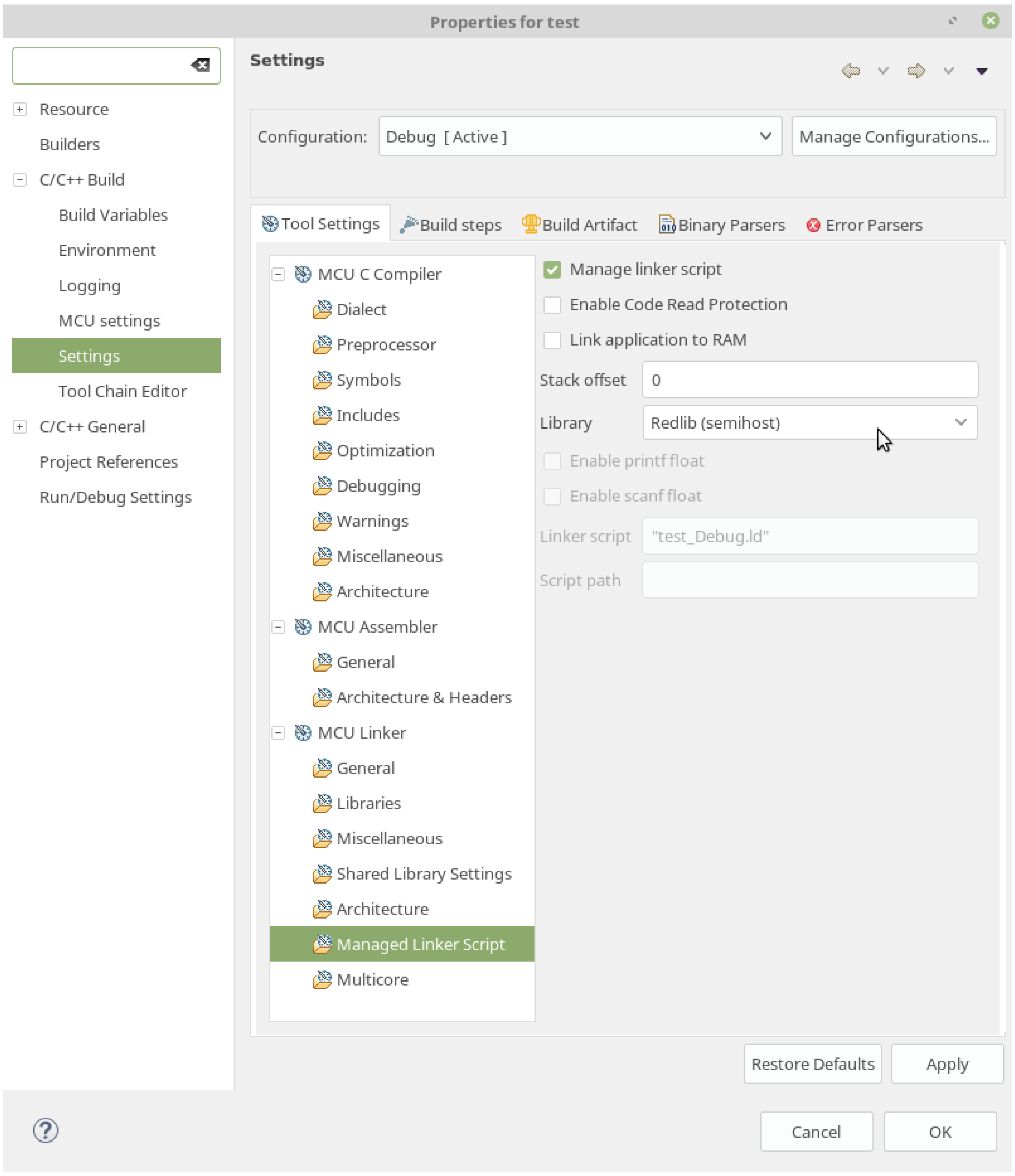
To be able to use the `printf` function to print variables to the console:

1. Right Click on the project's name and then click on *Properties*.
2. Expand *C/C++ Build* and select *Settings*.
3. Under *MCU Linker*, click on *Managed Linker Script*.
4. Change the *Library* used from *Redlib (none)* to *Redlib (semihost)*, as shown in the Semihost Debugging Configuration figure.

This is a limited implementation of the `printf` function that does not recognize all format specifiers, but is sufficient for most debugging needs.

Example 5. Using the `printf` Function

```
printf("push-button value: %d\n", value);
// Given that value is an integer (int) containing the state of a push button
```


Properties for test

Settings

Configuration: Debug [Active]

Manage linker script: ✓
Enable Code Read Protection: ☐
Link application to RAM: ☐
Stack offset: 0
Library: Redlib (semihost)
Enable printf float: ☐
Enable scanf float: ☐
Linker script: "test_Debug.ld"
Script path:

Figure 2. Semihost Debugging Configuration

2.4. Tasks

In this experiments, you will control LED lights by software.

2.4.1. Hardware

1. Find out which I/O Pins you *can* use for controlling LEDs, and choose specific ones.
2. Connect four LEDs using a proper current-limiting resistor.
3. Connect two switches using a proper current-limiting resistor.

2.4.2. Software

Blink an LED using `FIOSET` and `FIOCLR`

Pick an LED and blink it using *only one* `FIOPIN` *statement* and *one delay loop!*

> Use bitwise exclusive-OR.

Implement LED Scrolling

1. Write a program that makes it look like the light is scrolling through 4 LEDs that are connected externally. The scroll effect can be achieved by turning LEDs ON and OFF sequentially.
2. Use two switches to control the scrolling. For example, you can use one switch to turn the scrolling ON and OFF, and the second switch to reverse the scroll direction.

2.5. Resources

[]

NXP Semiconductors. *LPCXpresso IDE User Guide*. Rev. 8.1. 31 May 2016. http://www.nxp.com/assets/documents/data/en/user-guides/LPCXpresso_IDE_User_Guide.pdf

2.6. Grading Sheet

Task	Points
Using FIOPIN with XOR	4
Using FIOPIN with input pins	4
Discussion	2

3. Experiment 3: Interrupts

Hazem Selmi; Ahmad Khayyat 162, 24 February 2017

3.1. Objectives

- Understand interrupts in LPC1769
- Introduce the `PINSELx` registers
- Using external interrupts

3.2. Parts List

- LPC1769 LPCXpresso board
- USB A-Type to Mini-B cable
- Breadboard
- LEDs
- Push-buttons or switches
- 330-Ohm Resistors
- Jumper wires

3.3. Background

Interrupts are essential for embedded systems. The information in this experiment will be used in all future experiments in this lab.

3.3.1. Interrupts in LPC1769

Interrupts allow for suspending the currently executing code, and having the CPU switch to execute a routine associated with the received interrupt request.

In LPC1769, there are 35 hardware interrupts. Each interrupt is identified by a number called `IRQn`, or *Interrupt ID* as called in the LPC1769 manual. Below are some examples of hardware interrupts and their IDs:

Table 1. Example Interrupt IDs

Interrupt ID	Interrupt Source
1	Timer 0
2	Timer 1
5	UART 0
6	UART 1
13	SPI
18	EINT0
19	EINT1
20	EINT2
21	EINT3
33	USB Activity Interrupt

In CMSIS, interrupts are numbered from 0 to 34.

In addition to these 35 interrupts, there are 8 exceptions with negative numbers. Exceptions are not discussed in this experiment.

Exercise

Find the interrupt number definitions of the LPC1769 in the `lpc17xx.h` header file.

Setting up Interrupts in LPC1769

There are four main steps to correctly setup any interrupt in LPC1769:

1. Configure the required peripheral to generate interrupt requests. For example, to be able to use a timer interrupt, a timer must be activated and configured to generate interrupt requests.
2. Enable the interrupt in the *Nested Vectored Interrupt Controller (NVIC)*. See Enabling the Interrupt in the NVIC for more information about the NVIC.
3. Write an *interrupt service routine (ISR)*: the routine that needs to be executed when an interrupt request is received.

4. Clear the interrupt request at the end of the ISR to allow future requests of the same interrupt.

> Some peripherals that are capable of generating interrupts can also be used without interrupts. There are valid uses for either approach. That's why it is required to configure devices to generate interrupt requests when that behavior is desired (step 1 in the list above).

Upon setting up an interrupt as described above, the LPC1769 microcontroller will be responsible for:

- Detecting the interrupt request generated by a peripheral. This request will be generated by the peripheral that has been enabled by software.
- Jumping to the interrupt service routine associated with that request.

3.3.2. Configuring Microcontroller Interrupts

This section elaborates the four steps listed in the previous section (Setting up Interrupts in LPC1769) to configure the LPC1769 microcontroller to handle interrupt requests generated by a given device.

The programmer must perform four main steps:

1. Enable a peripheral to generate a hardware interrupt request (not to be discussed here because it is peripheral dependent)
2. Enable the required interrupt in the NVIC
3. Write an interrupt service routine (ISR)
4. Clear the interrupt request at the end of the ISR

Enabling the Interrupt in the NVIC

The Nested Vectored Interrupt Controller (NVIC) offers very fast interrupt handling and provides the vector table [keil-nvic].

In addition, the NVIC:

- Saves and restores automatically a set of the CPU registers (R0-R3, R12, PC, PSR, and LR).
- Does a quick entry to the next pending interrupt without a complete pop/push sequence.
- Provides many other advanced features.

NVIC Functions

The CMSIS core module defines a set of interrupt helper functions. For example, to enable interrupts for a given interrupt ID, you can use the function:

```
NVIC_EnableIRQ(IRQn);  // IRQn is the interrupt ID
```

For example, for `UART1`, `IRQn` is 6 (see Example Interrupt IDs Table). You can use this number or use the given name in `lpc17xx.h`: `UART1_IRQn`.

The ISR

Whenever an interrupt request is generated, the CPU will jump to the corresponding ISR. When using CMSIS, the ISR is a C function that has the following prototype format:

```
void __peripheral___IRQHandler();
```

Example 6. An ISR for the TIMER2 Device

```
void TIMER2_IRQHandler() {
    // Your code goes here
    // Clear the interrupt request at the end of the ISR
}
```

Clearing the Interrupt Request

As indicated in the format of the ISR above, the last statement in any ISR should be to clear the request that has just been served. This is required to allow future requests of the same interrupt.

This step is peripheral-dependent and is usually done by clearing a bit in one of the peripheral registers.

Other Interrupt-Related Operations

Interrupts will not function at all without the above four steps. There are other issues, however, that are not essential in simple applications, but can be very useful and even essential in some applications, especially when you have multiple interrupts. We will discuss two such issues here:

1. Interrupt status
2. Interrupt priority

Interrupt Status

Sometimes, you need to check the status of a specific interrupt. For example, is it pending, active or disabled.

When using CMSIS, the status of interrupts can be checked by calling one of the following functions, depending on the application:

- `uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)`
    - If the interrupt status is *not pending*, the function returns `0`.
    - If the interrupt status is *pending*, the function returns `1`.
- `uint32_t NVIC_GetActive(IRQn_Type IRQn)`
    - If the interrupt status is *not active*, the function returns `0`.
    - If the interrupt status is *active*, the function returns `1`.

Interrupt Priority

When using CMSIS, you can set interrupt priorities by calling the function:

`void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)`

- The fist argument is the interrupt ID.
- The second argument represents the priority, where 0 is the highest priority and 31 is the lowest priority.

To assign a different priority for each interrupt, you need to call this function for every interrupt you are using.

3.3.3. External Interrupts

To practice interrupts, we will concentrate on external interrupts only in this experiment, since other hardware interrupts require understanding the functions with which they are associated, which we did not cover yet.

One type of interrupts that is easy to experiment with is external interrupts. They are the interrupts that are generated by a device outside the microcontroller. An external interrupt should be connected to one of the I/O port pins.

In external interrupts, an interrupt request is generated by a pulse at a pin that has been enabled to accept external interrupt requests. A simple way to implement that is to use a push-button to generate that request.

The difference between such implementation and what you did in Experiment 2 is that in Experiment 2, we used *polling*, where the CPU is always busy reading the pin in order to detect a change that would trigger some action. When using interrupts, however, the CPU is available to execute other code. When the push-button is pressed, the CPU stops whatever it is doing and jumps to the routine associated with that interrupt request.

There are four external interrupt channels available to the developer, called `EINT0`, `EINT1`, `EINT2` and `EINT3`. In older ARM versions, a pin's function must be set for the pin to act as an external interrupt. This is done using the `PINSELx` register (see The `PINSELx` Registers). However, one of the new features of the newer Cortex family is accepting external interrupts from some GPIO pins! Any GPIO pin used for external interrupts will be using external interrupt channel 3 (`EINT3`).

You can use GPIO pins from ports 0 and 2 only for external interrupts. You have about 40 different pins to choose from. Compare that, for example, to ARM7 where only 7 pins are available for external interrupts.

External interrupts can be enabled on two sets of pins:

1. Four dedicated pins (P2.10, P2.11, P2.12 and P2.13) that act as `EINT0`, `EINT1`, `EINT2`, and `EINT3`, respectively.
2. Any GPIO pin in port 0 and port 2.

In the two following sections, we will discuss and practice:

- GPIO external interrupts
- Non-GPIO external interrupts

GPIO external interrupts

As discussed in Setting up Interrupts in LPC1769, you always need to enable the NVIC and write an ISR.

For GPIO external interrupts, that leaves two more steps:

1. Activating GPIO external interrupts
2. clearing a GPIO interrupt request at the end of the ISR

Activating GPIO External Interrupts

To activate external interrupts on a GPIO pin, you only need to configure whether the pin is to generate an interrupt request on the rising edge or on the falling edge.

You can set external interrupts to be generated on the *rising edge* on a GPIO pin by setting the `IO0IntEnR` and `IO2IntEnR` registers, depending on the port to which the pin belongs. These names refer to 32-bit registers. Setting a bit to `1` enables rising-edge interrupts at the corresponding pin.

To generate interrupts on the *falling edge*, you can use the `IO0IntEnF` and `IO2IntEnF` registers instead.

In `LPC17xx.h`, the structure that deals with GPIO external interrupts is `LPC_GPIOINT`, which includes a few fields that control the GPIO pins when acting as an external interrupt.

Example 7. Enable Rising-Edge Interrupts on Pin 0 of Port 2 Only

```
LPC_GPIOINT->IO2IntEnR = 1;
```

Clearing External GPIO Interrupt Requests

To clear the interrupts of a port pin, set the corresponding bit to `1` in register `IO0IntClr` or `IO2IntClr`, depending on the port. Both registers are fields of the `LPC_GPIOINT` structure.

Other issues related to GPIO interrupts

Level and Edge Sensitivity (For Non-GPIO Interrupts)

You may have noted that, to enable GPIO interrupts, you have to select whether they are triggered by the rising or falling edge of the pulse at the pin.

Interrupt Status for GPIO External Interrupts

You can check for pending GPIO interrupts by reading the appropriate status register. There are four status registers for ports 0 and 2 that indicate whether an interrupt is pending, and whether it is triggered by a rising edge or a falling edge. They are `IO0IntStatR`, `IO2IntStatR`, `IO0IntStatF`, and `IO2IntStatF`.

For Example, if bit 9 of `IO2IntStatR` is `1`, then P2.09 has a pending rising-edge interrupt request.

This is particularly important when you have multiple interrupts sharing the same interrupt channel (EINT3 in our case). Any one of them can result in executing the same ISR. Now, If you want to perform different actions for each interrupt, you need to identify the source interrupt in order to perform the corresponding action. You can do that by checking the status registers in your ISR.

Non-GPIO external interrupts

As is the case with GPIO external interrupts, the two basic steps for using non-GPIO external interrupts are:

1. Activating Non-GPIO external interrupts
2. Clearing a Non-GPIO interrupt request at the end of the ISR

Read Setting up Interrupts in LPC1769 again!

Activating Non-GPIO External Interrupts

External interrupt requests can be generated using any of the 4 dedicated external interrupt pins, named `EINT1`, `EINT2`, `EINT3`, and `EINT4`:

EINT0
    P2.10

EINT1
    P2.11

EINT2
    P2.12

EINT3

P2.13

Review

If an external interrupt is requested through one of the above pins, the CPU will jump to the corresponding ISR, which is a C function identified by its name as follows:

Name	Pin	Handler (ISR)
`EINT0`	P2.10	`void EINT0_IRQHandler()`
`EINT1`	P2.11	`void EINT1_IRQHandler()`
`EINT2`	P2.12	`void EINT2_IRQHandler()`
`EINT3`	P2.13	`void EINT3_IRQHandler()`

To configure one of the above pins to act as `EINTx`, you must set the appropriate `PINSELx` register bits (see The `PINSELx` Registers).

Exercise

Which `PINSELx` register(s) and bits do you need to set to have access to each of `EINT0`, `EINT1`, `EINT2`, and `EINT3`?

Refer to the LPC1769 User Manual.

Setting the function of a pin to external interrupt (`EINTx`) *enables* the corresponding interrupt.

From the above, you can see that external interrupt no. 3, `EINT3`, can be activated in two ways:

1. Using `PINSEL4` to activate `EINT3` at P2.13; or
2. Using a GPIO pin from port 0 or port 2.

In other words, the `EINT3` channel is shared with GPIO interrupts.

Clearing External Non-GPIO Interrupt Requests

When a Non-GPIO external interrupt request is received, an interrupt flag is set in the `EXTINT` register, which would assert the request to the NVIC. The four least significant bits in the `EXTINT` register indicate the pending external interrupts. For example, when `EINT0` is enabled and requested, bit 0 in `EXTINT` will be set to `1` by the CPU.

Active bits in the `EXTINT` register *must be cleared in the ISR*. Otherwise, future similar interrupt requests will not be recognized. To clear a bit in the `EXTINT` register, set it to `1`.

The `EXTINT` register is also a field in the `LPC_SC` structure.

For example, to clear all external interrupts:

```
LPC_SC->EXTINT |= 0xF;
```

Other issues related to Non-PIO interrupts

Level and Edge Sensitivity (For Non-GPIO Interrupts)

The default for non-GPIO interrupts is that LOW triggers an interrupt request. However you can change that using the `EXTPOLAR` and `EXTMODE` registers, which control the non-GPIO external interrupt behavior.

Although these registers are 32-bit, only the least significant 4 bits are used. Bit `0` controls `EINT0`, bit `1` controls `EINT1`, bit `2` controls `EINT2`, and bit `3` controls `EINT3`.

`EXTMODE` selects whether external interrupts are *level*-sensitive (`0`) or *edge*-sensitive (`1`).

`EXTPOLAR`, the External Interrupt Polarity Register, controls which level or edge on each pin will cause an interrupt (depending on `EXTMODE`):

1. If `EXTMODE` is set to level sensitivity, setting a bit in `EXTPOLAR` to a `0` specifies that the corresponding external interrupt is *LOW-active* (triggered by the `0` level), and setting it to a `1` makes it *HIGH-active* (triggered by the `1` level).
2. If `EXTMODE` is set to edge sensitivity, setting a bit in `EXTPOLAR` to a `0` specifies that the corresponding external interrupt is *falling-edge sensitive*, and setting it to a `1` makes it *rising-edge sensitive*.

Both `EXTMODE` and `EXTPOLAR` registers are fields of the `LPC_SC` structure (`SC` for System Control).

3.3.4. The `PINSELx` Registers

This section is not specific to interrupts. It is about configuring the function of a pin in a port. One possible functions is *external interrupt*.

The Configuring Interrupts section above covered three of the four required steps to fully setup interrupts. The remaining step is to configure the hardware that is responsible for generating the interrupt request. This step is largely dependent on the hardware that is going to generate the request, but is always required.

Configuring the hardware involves a common step regardless of the hardware being configured. That common step is configuring the functions of the relevant pins.

Each pin can be configured to perform one of four functions. Therefore, the function of each pin is controlled by two bits, as follows:

00

Primary (default) function, (GPIO)

01

First alternate function

10

Second alternate function

11

Third alternate function

As such, to configure the functions of the five 32-bit ports, ten function selection registers are required. They are named `PINSEL0`, `PINSEL1`, `PINSEL2`, …, `PINSEL9`. `PINSEL0` controls the functions of the lower half of port 0 (P0.0 to P0.15), `PINSEL1` controls the functions of pins P0.16 to P0.31, `PINSEL2` controls the functions of pins P1.0 to P1.15, and so on.

For example, the two least significant bits in `PINSEL0` control the function of pin P0.0 as follows:

00
  GPIO

01
  `RD1`: CAN1 receiver input

10
  `TXD3`: Transmitter output for UART3

11
  `SDA1`: I2C1 data input/output

(See Table 73 in the LPC1769 User Manual.)

All `PINSELx` registers are fields in the `LPC_PINCON` structure.

So, to configure P0.0 to function as `TXD3` instead of GPIO:

```
LPC_PINCON -> PINSEL0 = 0x00000002;   // Assignments like this are not the best way,
                                      // unless you want to set the remaining pins to GPIO
```

To avoid affecting other pins, You may want to use bitwise operations to set and/or clear the required bits in `PINSELx`.

Using `00` for any pin sets its function to GPIO. The reset value for `PINSELx` registers is `0x00000000`. That is why the default function for all I/O pins after a reset is GPIO.

You may want to refer back to this section whenever you want a pin to have a function other than GPIO.

3.4. Tasks

3.4.1. One External Interrupt

1. Use a push-button to generate an external interrupt using a GPIO pin. Do something interesting in the ISR!
2. Use a push-button to generate an external interrupt using a non-GPIO pin, i.e. a pin explicitly configured for external interrupts. Use the same ISR from task 1.

3.4.2. Two External Interrupts

3. Use two external interrupts, where each interrupt triggers a different task. For example, each interrupt could blink an LED 10 times at a different rate.

   The faster rate interrupt should have a higher priority; if it is activated while the slow rate interrupt is being serviced, the slow rate interrupt handler will be paused to service the fast rate interrupt and then come back to the *pending* slow interrupt.

All tasks must be completed during the lab session.

3.5. Resources

[]

ARM Ltd. 'Nested Vectored Interrupt Controller'. 2013.
http://www.keil.com/support/man/docs/gsac/gsac_nvic.htm

[]

NXP Semiconductors. 'UM10360 LPC176x/5x User manual'. Rev. 3.1. 2 April 2014.
http://www.nxp.com/documents/user_manual/UM10360.pdf

3.6. Grading Sheet

Task	Points
Task 1: External interrupt using a GPIO pin	2
Task 2: External interrupt using a non-GPIO pin	3
Task 3: Two external interrupts with priorities	3
Discussion	2

4. Experiment 4: Hardware Timers

Hazem Selmi; Ahmad Khayyat; Mansour Alharthi 162, 10 March 2017

4.1. Objectives

- Using hardware timers
- Using the LPC1769 manual to figure out how to use a given register
- Identifying how to access a given register by referring to the `LPC17xx.h` file

4.2. Parts List

- LPC1769 LPCXpresso board
- USB A-Type to Mini-B cable
- Breadboard
- LEDs
- 330-Ohm Resistors
- Jumper wires

4.3. Background

There are four hardware timers in LPC1769: Timer 0, Timer 1, Timer 2, and Timer 3. They are identical and can work independently with different settings.

Throughout this document, *timer* or `TIMERx` refer to one of these LPC1769 timers. Every one of the timer-related registers discussed henceforth applies to all of these four timers, and cannot be used without specifying the targeted timer.

4.3.1. Timer Basic Operation: Timer Counter

The basic function of any timer is to have a counter running. In LPC1769, this counter is called *Timer Counter* (`TC`).

In this section, we will learn how to enable `TC` to start ticking, and will find out how fast it can run.

Controlling the Counting Speed

Peripherals in LPC1769 are fed with an input clock called the peripheral clock (`PCLK`). By default, the *Timer Counter* (`TC`) register is incremented every `PCLK` cycle.

There are two ways to change that:

1. Divide `PCLK` by a factor other than the default. This will change the input clock frequency. Since this method is applicable for all peripherals we will discuss it in a separate section at the end of this document.
2. Using an intermediate counter called the *prescale counter*.

The *prescale counter* is *always* incremented every `PCLK` pulse. This continues till the *prescale counter* = the *prescale register*. When that happens, two events take place in the next `PCLK` pulse:

- Increment `TC` by 1
- Reset the *prescale counter* and continue counting

If the *prescale register* is not set to any value (the default is 0), `TC` to be incremented every `PCLK`.

Although it was claimed ealier that `TC` is by default incremented every `PCLK`, you now know that this is only true when the *prescale register* = 0.

Example 8. `TC` and the Prescale Register

If you set the *prescale register* to 5, `TC` will be incremented every 6 `PCLK` pulses.

Enabling the Counter

To start using a timer, you first must enable counting! In LPC1769, the *Timer Control Register* (`TCR`) is the register that allows you to do that.

As should be clear from previous experiments, you interact with peripherals through registers. In the case of timers, to enable a counter and have it start counting, you need to write to the `TCR` register. To do that, you must:

1. Identify the relevant bit(s) of the register. For that, refer to table 428 in chapter 21 of the LPC1769 manual.
2. Find out how to access this register using CMSIS. One easy way to do that is to check the `lpc17xx.h` file to find the structure containing the `TCR` field.

Exercise

- How can you enable Timer 0 counter?
- How can you enable Timer 3 counter?

You can write a single line of code that would enable the counter, and then use `printf()` to see whether `TC` is counting.

4.3.2. Timer Counter (`TC`) is Ticking; Now What?

There are two main ways to use a ticking timer:

1. Load a *match register* (`MR`) with some value and then wait till `TC`=`MR` to trigger some action.
2. Capture the time in a *capture register* (`CR`), i.e. set `CR`= `TC`, whenever an event takes place on a given pin. The event is simply any change of the pin state (HIGH → LOW or LOW → HIGH, i.e. a positive edge or a negative edge).

In this section, we will discuss these two options.

Timing Using a Match Register

For each LPC1769 timer, there are four match registers: `MR0`, `MR1`, `MR2`, and `MR3`.

Timer Registers

Hereafter, `MR` or `MRx` refers to one of `MR0`, `MR1`, `MR2`, and `MR3` *match registers* of a specific LPC1769 timer.

When the value of `TC` reaches the value in the *match register* (`MR`), an action is triggered. Therefore, setting `MR` specifies the timer's period. The action triggered every time `TC` reaches `MR0` can be set using the *Match Control Register* (`MCR`) to one (or more) of the following:

1. Generate an interrupt
2. Reset `TC`
3. Stop `TC`

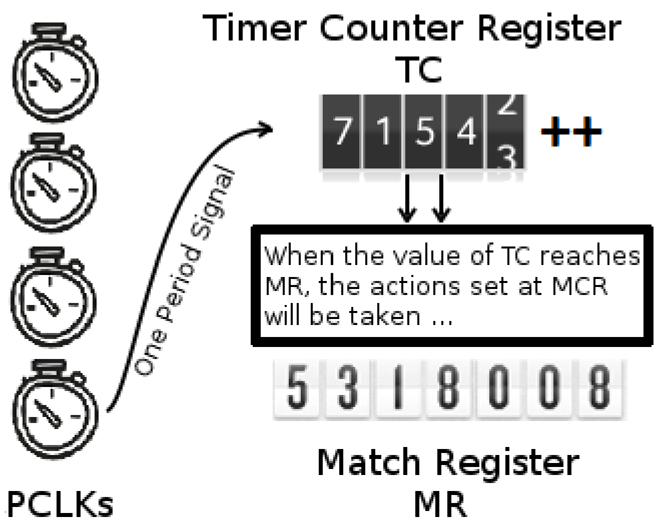


Figure 3. Timers in LPC1769

You can enable or disable the above actions by setting or clearing the three least significant bits of the `MCR` register.

Table 2. Setting Timer Actions Using the `MCR` Register

MCR bit	Bit value = 1	Bit value = 0
0	Enable timer interrupt	Disable timer interrupt
1	Reset TC	Disable this feature
2	Stop TC	Disable this feature

External Match Action

You can also trigger a different action when TC=MRx, which is, to *set*, *reset*, or *toggle* a specific bit. This bit can be pinned out to an external output pin, hence the name: *External Match* bit (EMx).

For each timer, there are 4 EM bits, namely EM0, EM1, EM2, and EM3. Each EMx bit can be controlled when TC equals the corresponding MRx. These four EM bits belong to the EMR register. In other words, for each MRx, the external match *control bits* and the *controllable* bit are all part of the same EMR register.

Study the EMR register tables (432 and 433) in chapter 21 of the LPC1769 manual to understand the following examples.

Example 9. External Match Actions

- Assigning 0 to bit 6, and 1 to bit 7 in EMR will force bit 1 in EMR to be HIGH when TC = MR1.
- Assigning 1 to both bits 10 and 11 in EMR will toggle bit 3 in EMR when TC = MR3.

Theoretically, any EM bit can be pinned out to a pin that is named MATx.y, where x is the timer number and y is the match register number.

Example 10. Pinned Out External Match Actions

- When using MR3 with Timer 2, the EM3 bit of the EMR register of Timer 2 can be pinned out to MAT2.3.
- When using MR1 with Timer 0, the EM1 bit of the EMR register of Timer 0 can be pinned out to MAT0.1.

Practically, however, only MATx.0 and MATx.1 are available in LPC1769 for Timer 0, Timer 1, and Timer 3, whereas Timer 2 can use all four MAT2.y pins.

You need to change a pin's function to use it as MATx.y. Refer to the PINSEL section in experiment 3 and chapter 8 of the LPC1769 manual for more details.

Exercise

One of the tasks in this experiment is about external match actions. To be able to complete that task, you need to find a suitable MATx.y pin.

So, refer to chapter 8 of the LPC1769 manual and list all the MATx.y pins and find out which of them is physically available and accessible on your LPCXpresso board.

Capturing an Event (Event Timers)

Instead of using a *match register*, you can capture the time in a *capture register* (CR) when a pin's state changes. In other words, you can take a snapshot of the timer value when an input signal changes.

This happens by loading the TC value into a CR (CR ⇐ TC) when an input pin has a positive edge and/or a negative edge.

For each timer, there are two *capture registers*: `CR0` and `CR1`. A pin that can be used with a `CR` is named `CAPx.y`, where `x` is the timer number and `y` is *capture register* number.

Example 11. Capture Registers

- By using `CAP1.0`, you will be loading `TC` into `CR0` of Timer 1.
- By using `CAP0.1`, you will be loading `TC` into `CR1` of Timer 0.

To enable this feature, you need to use the `CCR` register. In addition to capturing the time, you can use the `CCR` register to enable generating an interrupt when the state of `CAPx.y` changes.

Study the `CCR` register table (431) in chapter 21 of the LPC1769 manual to understand the following examples.

Example 12. Using the `CCR` Register

Assign 15 (1111 in binary) to the `CCR` register of Timer 0 will:

- Load `TC` to `CR0` on both the positive and negative edges of `CAP0.0`
- Generate a Timer 0 interrupt request
- Load `TC` to `CR1` only on the positive edges of `CAP0.0`, without generating interrupt requests.

You need to change a pin's function to use it as `CAPx.y`. Refer to the `PINSEL` section in experiment 3 and chapter 8 of the LPC1769 manual for more details.

Exercise

One of the tasks in this experiment is about capturing event times. To be able to complete that task, you need to find a suitable `CAPx.y` pin.

So, refer to chapter 8 of the LPC1769 manual and list all the `CAPx.y` pins and find out which of them is physically available and accessible on your LPCXpresso board.

4.3.3. Important Notes

- If you choose to enable the timer interrupt, remember to enable the the NVIC and to clear the interrupt bit in the ISR. To clear the `MR0` interrupt flag, set the least significant bit in the *Interrupt Register* (`IR`).
- A common misconception is to assume that register `MR0` can be used with timer 0 only, register `MR1` with timer 1 only, and so on. Each timer has its own 4 match registers.
- As usual, all the registers in this experiment are fields of some structures. Refer to the `LPC17xx.h` header file to find the required name and field to access the required register.

Exercise

In this exercise, we will use a hardware timer and timer interrupts to blink an LED.

```
// "x" is a placeholder. Replace x with an appropriate value.

int main(void) {

    // Try to find out the IRQ number. Why is this step important?
    NVIC_EnableIRQ(x);
    // Answer:

    // What does register TCR do?
    LPC_TIMx->TCR |= x;
    // Answer:

    // What does register MRx do?
    LPC_TIMx->MRx = x;
    // Answer:

    // What does register MCR do?
    LPC_TIMx->MCR = x;
    // Answer:

    LPC_GPIOx->FIODIR = 1 << x ;

    // Can we remove this while loop? Why?
    while(1);
    // Answer:

    return 0 ;
}


// When will the following function be executed? Who is going to call it?
// Answer:

void TIMERx_IRQHandler() {

    LPC_GPIOx->FIOPIN ?? (1 << x);
    // Replace "??" with the appropriate operator

    // What does register IR do?
    LPC_TIMx->IR |= (1 << x);

}
```

4.3.4. Peripheral Clock (`PCLK`)

Timers, among other devices, rely on *peripheral clocks* (`PCLK`), which in turn are derived from the *core clock* (`CCLK`).

There are four possible frequency configurations for the peripheral clock (`PCLK`), which are set using a pair of bits.

Table 3. Peripheral Clock (`PCLK`) Frequency Configurations

Bit Values	Frequency Configuration
`01`	`PCLK` = `CCLK`
`10`	`PCLK` = `CCLK` / 2
`00`	`PCLK` = `CCLK` / 4
`11`	`PCLK` = `CCLK` / 8

These pairs of bits belong to the `PCLKSEL0` and `PCLKSEL1` registers, which control the `PCLK` frequency for all peripherals.

The `PCLKSEL0` and `PCLKSEL1` Register Fields figure illustrates some of the fields of the `PCLKSEL0` and `PCLKSEL1` registers. Every two bits control the `PCLK` frequency for a specific peripheral.

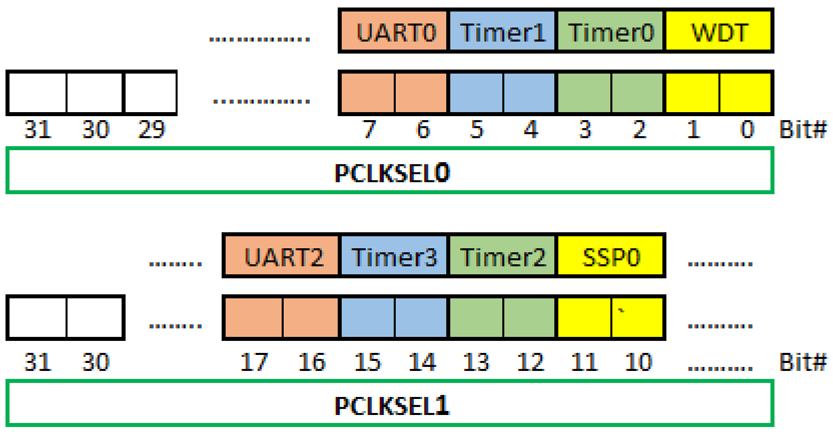


Figure 4. `PCLKSEL0` and `PCLKSEL1` Register Fields

Question

Can you ignore this step? What would happen if we skip it?

For the full list of peripherals and their corresponding two bits in `PCLKSEL0` or `PCLKSEL1`, you can refer to Chapter 4 (section 4.7.3) in the LPC1769 manual.

This section is not specific to timers. It is about configuring the frequency of `PCLK`, which is required for timers.

You may want to refer back to this section whenever you want to use a peripheral that requires `PCLK`.

4.3.5. Power Up

All microcontroller peripherals must be powered up before they can be used. This was not a concern in earlier experiments because we were using peripherals that are powered up by default.

Powering peripherals up and down is controlled through the *Power Control for Peripherals Register* (`PCONP`).

By referring to table 46 in Chapter 4 of the LPC1769 manual, you can see that the reset value (default value) is `1` for some peripherals, meaning that they are powered on by default, whereas it is `0` (OFF by default) for others.

Example 13. Powering peripherals on

```
LPC_SC -> PCONP |= (1 << xx);
// where xx is the bit number in PCONP that controls the
// power (ON/OFF) for a specific peripheral.
```

Timer 0 and Timer 1 are powered up by default. However, if you use Timer 2 or Timer 3, your experiment will not work without powering up the timer in your program.

To save power, you can turn the power OFF for any unused peripherals that are ON by default.

4.4. Tasks

1. Complete the LED blinking exercise above. Note that a for loop is not needed to implement the delay.
2. Blink an LED *without* using timer interrupts.
3. Connect an output pin to two capture pins, say `CAP2.0` and `CAP2.1`. Enable one of them to capture the time with the rising edge and the other one with falling edge.

   Now, set the output pin high then clear it immediately. Calculate the difference between `CR0` and `CR2` and use `printf()` to display this difference.

   Can you explain the result?

   Try using `FIOPIN` instead of `FIOSET` and `FIOCLR` to control the output pin.

   Try using direct assignment or bitwise OR for masking the remaining bits.

   Can you explain the different results?

Use external match actions for task 2.

4.5. Resources

[]

NXP Semiconductors. 'UM10360 LPC176x/5x User manual'. Rev. 3.1. 2 April 2014. http://www.nxp.com/documents/user_manual/UM10360.pdf

4.6. Grading Sheet

Task	Points
Use hardware timers with Interrupts	4
Use External match pins MATx.y	4
Use the CAPx.y pins with capture registers	2

5. Experiment 5: Analog Input and Output

Hazem Selmi; Ahmad Khayyat 162, 19 March 2017

5.1. Objectives

- Using the *Analog-to-Digital Converter (ADC)* to read analog input
- Using the *Digital-to-Analog Converter (DAC)* to write analog output

5.2. Parts List

- LPC1769 LPCXpresso board
- USB A-Type to Mini-B cable
- Breadboard
- Light sensor and/or potentiometer
- Seven-segment display or set of LEDs
- 330-Ohm Resistors
- Jumper wires

5.3. Background

Many microcontrollers have pins that can be used for *analog input*. Because the microcontroller processes digital data only, analog input must be converted to digital data. An analog-to-digital converter (ADC) is an I/O circuit often integrated into microcontrollers to allow directly connecting external analog devices, such as sensors. The ADC would convert the sensor voltage into a digital value by transforming it into a binary code with a specific number of bits.

Although not critical to conducting this experiment, it would be useful to review the three steps involved in analog-to-digital conversion: sampling, quantization, and bit encoding (COE 241).

5.3.1. Using LPC1769 Peripherals

The LPC1769 includes an integrated ADC peripheral device. In general, using any peripheral device involves three main issues:

1. Powering up the peripheral
2. Configuring the peripheral clock
3. Configuring pin functions

Power Up

All microcontroller peripherals must be powered up before they can be used. This was not a concern in earlier experiments because we were using peripherals that are powered up by default.

Powering peripherals up and down is controlled through the *Power Control for Peripherals Register* (`PCONP`).

By referring to table 46 in Chapter 4 of the LPC1769 manual, you can see that the reset value (default value) is 1 for some peripherals, meaning that they are powered on by default, whereas it is 0 (OFF by default) for others.

For example, in the timer experiment, if you use a timer other than timer 0 or timer 1, your experiment wouldn’t work without powering up the timer in your program.

The A/D converter (ADC) power is controlled by bit 12 of the `PCONP` register, which is 0 by default. *You must set that bit to power up your ADC*.

To save power, you can turn the power OFF for any unused peripherals that are ON by default.

Peripheral Clock

Most of the microcontroller peripherals, including timers and the ADC, require setting a peripheral clock (`PCLK`) to drive the peripheral.

You have seen in Experiment 7 (Hardware Timers) that you can configure a device's `PCLK` using the `PCLKSEL0` and `PCLKSEL1` registers.

Exercise

Refer to Chapter 4 in the LPC1769 manual to find out the two bits needed to configure the `PCLK` frequency for the ADC.

Exercise

What would happen if you skip this step?

Pin Functions

Many microcontroller pins can be configured to perform one of many functions. From Experiment 3 (Interrupts), recall that the `PINSELx` registers are used to configure a pin's function. To use the ADC, you must set the function of an appropriate pin to be analog input (`AD0.x` in the manual).

Exercise

Refer to Chapter 8 of the LPC176x manual to determine:

1. which `PINSELx` register should be modified
2. which bits of the register should be modified
3. what value should the bits be set to

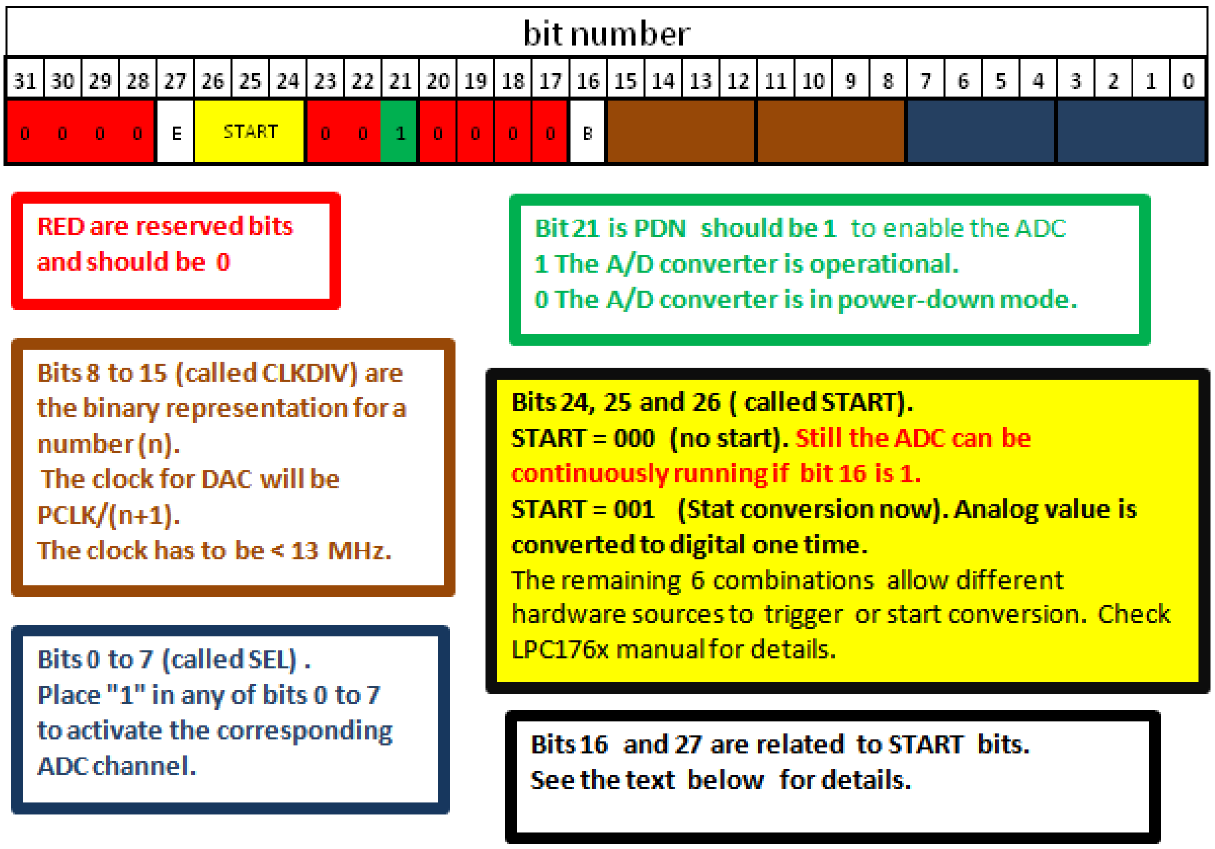
You should connect a device that generates an analog voltage signal to the selected pin. Examples of such devices are light sensors (LDR) and potentiometers.

It is professional to correctly address the above three issues for every peripheral you plan to use, regardless of the defaults.

5.3.2. ADC Configuration

The main setup register for the ADC is the *A/D Control Register* (`AD0CR`). The `AD0CR` Register Fields figure illustrates the fields of the `AD0CR` register.

There is only *one* ADC in the LPC1769 microcontroller. In the `LPC17xx.h` header file, the control register is referred to as `ADCR`; while in the chip manual it is called `AD0CR`. The reason for that is that some other chips have multiple ADCs, named: `AD0CR`, `AD1CR`, etc.



RED are reserved bits and should be 0

Bit 21 is PDN should be 1 to enable the ADC
1 The A/D converter is operational.
0 The A/D converter is in power-down mode.

Bits 8 to 15 (called CLKDIV) are the binary representation for a number (n).
The clock for DAC will be PCLK/(n+1).
The clock has to be < 13 MHz.

Bits 24, 25 and 26 (called START).
START = 000 (no start). Still the ADC can be continuously running if bit 16 is 1.
START = 001 (Stat conversion now). Analog value is converted to digital one time.
The remaining 6 combinations allow different hardware sources to trigger or start conversion. Check LPC176x manual for details.

Bits 0 to 7 (called SEL) .
Place "1" in any of bits 0 to 7 to activate the corresponding ADC channel.

Bits 16 and 27 are related to START bits.
See the text below for details.

Figure 5. A/D Control Register (`AD0CR`) Fields

The following table explains the function of the `B` (*Burst*) and `E` (*Edge*) bits of the `AD0CR` register.

Bit 27 (`E`) works only if `B` = 0 and `START` ≥ 2. When 2 ≤ `START` ≤ 7, the conversion starts when the state of a specific pin is changed. The `E` bit decides whether the ADC is triggered on the positive edge or the negative edge of that pin specified by `START`.

Bit	Label	Value	Effect
16	`B`	0	The `START` bits control when the ADC starts the conversion
		1	The ADC is continuously running (`START` should be `000`)
27	`E`	0	Start conversion on a falling edge
		1	Start conversion on a rising edge

START vs. BURST

Using `START` will perform the conversion only once.

If you want the analog value to be repeatedly converted, you have two options:

1. Set the `B` bit (Burst) of the `AD0CR` register to 1; or
2. Set the `START` bits to `001` repeatedly, i.e. in a loop. The analog value is read every time such a statement is executed.

Using ADC Interrupt

In simple ADC applications, you don't need interrupts. You can simply read the digitized value from the proper register whenever needed and take some action. However, in some applications, such as real time applications, you may need to interrupt the CPU to take an action *only when* the conversion is completed. To do that, you can use the `ADGINTEN` register.

See Table 534 in Chapter 29 of the LPC1769 manual for details.

5.3.3. Reading Digital Values

There are 8 ADC channels, each corresponding to an analog pin. The digitized value corresponding to an input analog voltage is stored in 12 bits in one of the *A/D Data Registers*: `ADDR0` to `ADDR7`, where each register corresponds to an analog pin.

The `ADDR` Register Fields figure illustrates the fields of the `ADDRx` registers.

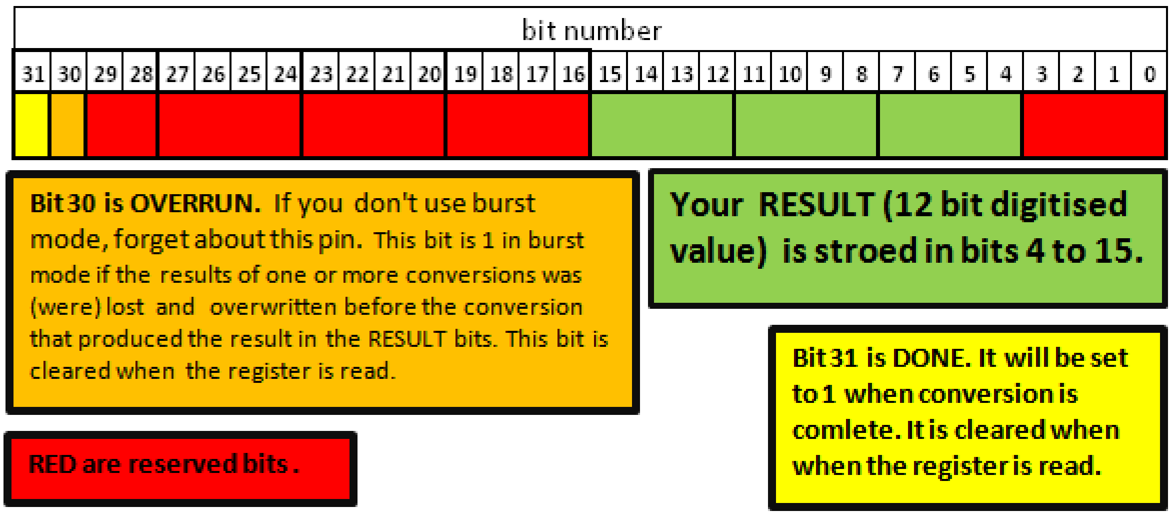

bit number
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Bit 30 is OVERRUN. If you don't use burst mode, forget about this pin. This bit is 1 in burst mode if the results of one or more conversions was (were) lost and overwritten before the conversion that produced the result in the RESULT bits. This bit is cleared when the register is read.
Your RESULT (12 bit digitised value) is stroed in bits 4 to 15.
Bit 31 is DONE. It will be set to 1 when conversion is comlete. It is cleared when when the register is read.
RED are reserved bits.

Figure 6. A/D Data Register (`ADDR`) Fields

Using proper shifting and bitwise operations, you should be able to get the proper value representing the analog voltage.

The `DONE` and `OVERRUN` bits are less important (may not be needed) in `BURST` mode. However in `START` mode, you may need to check them to avoid reading an old or unintended value.

Example 14. Using the `DONE` Bit

To wait until the conversion of the ADC channel 3 is over, you may use:

```
while ((LPC_ADC->ADDR3 & (1 << 31)) == 0);     // Check the DONE bit for ADC channel #3
```

The 12-bit digital value generated by the ADC ranges from 0 to 4095. The way to process this value depends on your application.

You may want to divide this range to a number of sub-ranges, and assign different actions for each sub-range. In this case, you can use an if-else block.

In many applications, however, you will want to map this range to a another range using a mathematical formula. For example, if you are reading from an analog temperature sensor, you would want to map the 0-to-4095 range to the range of temperatures supported by the sensor, as specified in the sensor’s data sheet. In most cases, a linear relationship is sufficient.

5.3.4. Analog Output

To write analog values to an analog output device, use the LPC1769's digital-to-analog converter (DAC) as follows:

1. Use `PINSELx` to configure P0.26 to function as analog output (`AOUT`).
2. Use the *D/A Converter Register* (`DACR`) to set the digital value to be converted to analog.

   Refer to Chapter 30 in the LPC1769 manual for details.

5.4. Tasks

1. Use the ADC in LPC1769 to read an analog input device, such as the LDR (light sensor) or the potentiometer.

   The output can be any thing you want. The seven-segment display is a good option. You can simply display the analog level. If you use one seven-segment display, you have 10 different levels (0 to 9).

   It is recommended to use a formula to map the readings to sensible values, instead of using an if-else block.

2. Use the DAC in LPC1769 to output analog values to an analog device.

5.5. Resources

[]

NXP Semiconductors. 'UM10360 LPC176x/5x User manual'. Rev. 3.1. 2 April 2014. http://www.nxp.com/documents/user_manual/UM10360.pdf

5.6. Grading Sheet

Task	Points
Analog Input	5
Analog Output	2
Discussion	3

6. Experiment 6: Pulse-Width Modulation

Hazem Selmi; Ahmad Khayyat 162, 9 April 2017

6.1. Objectives

- Understanding and using *pulse-width modulation (PWM)*.

6.2. Parts List

- LPC1769 LPCXpresso board
- USB A-Type to Mini-B cable
- Breadboard

- RGB-LED or buzzer
- Jumper wires
- Servo motor

6.3. Background

6.3.1. Pulse-Width Modulation (PWM)

A pulse-width modulated (PWM) signal is a periodic square wave signal. The difference between a PWM signal and a clock signals is the flexibility of its *duty cycle*.

A periodic square wave is high for some part of its period, and low for the rest of the period. Its *duty cycle* is the percentage of the period for which the signal is high. Usually, a clock wave has a duty cycle of 50%. In a PWM signal, the duty cycle is controllable. The name is derived from the idea that the *width* of the high *pulse* is *modulated* according to some value.

6.3.2. PWM Applications

PWM has many useful applications in embedded systems. The main two categories are:

1. When a microcontroller does not have a DAC circuit, PWM can be used to *modulate* different analog values.
2. Some devices are built to be used with PWM. The most famous example is servo motors.

   Servo motors usually require a 50-Hz square wave (period of 20 ms). The duration of the high pulse determines the motor's angle. Usually, the full swing of the servo corresponds to a high interval of 1 to 2 ms, whereas a high interval of 1.5 ms corresponds to the neutral servo position [1: https://circuitdigest.com/article/servo-motor-basics].

6.3.3. Generating PWM with LPC1769

The LPC1769 features a pulse-width modulator peripheral. The generic steps discussed in Experiment 5 for setting up a peripheral device apply here:

1. Power: the PWM circuit is powered on by default.
2. Peripheral Clock (PCLK): recall that the default division factor is 4.
3. Pin functions: a PWM pin must be configured for PWM use.

Additionally, generating a PWM signal in particular requires:

1. Setting the period of the PWM signal using the `MR0` register.
2. Specifying the duty cycle using an `MRx` register, which would control the `PWM1.x` output.
3. The PWM circuit should be enabled to generate a PWM signal, otherwise it will act as a standard timer (or counter).
4. The corresponding `PWM1.x` output should be enabled.

1. There is only one PWM circuit, called `PWM1`. That does not imply that there is a `PWM0` or `PWM2`.
2. There are six PWM channels, referred to as `PWM1.1` to `PWM1.6`.
3. You have the option of more than one pin to pin out any of the channels.

If you care about the accuracy of your PWM output voltage levels, you need to disable the pull-up resistor to avoid affecting the PWM voltage. That can be done using the `LPC_PINCON→PINMODEx` register.

In many applications this is not required.

Exercise

Refer to chapter 8 of the LPC176x manual to determine:

1. Which pins are you going to use for PWM?
2. Which `PINSELx` register should you use?
3. Which `PINSELx` bits should you set?
4. To what value should you set those `PINSELx` bits?
5. How to disable the pull-up resistor?

`MR0` and `MRx`

To fully specify a PWM signal, you need to specify:

1. Its period (or, equivalently, its frequency)
2. Its duty cycle

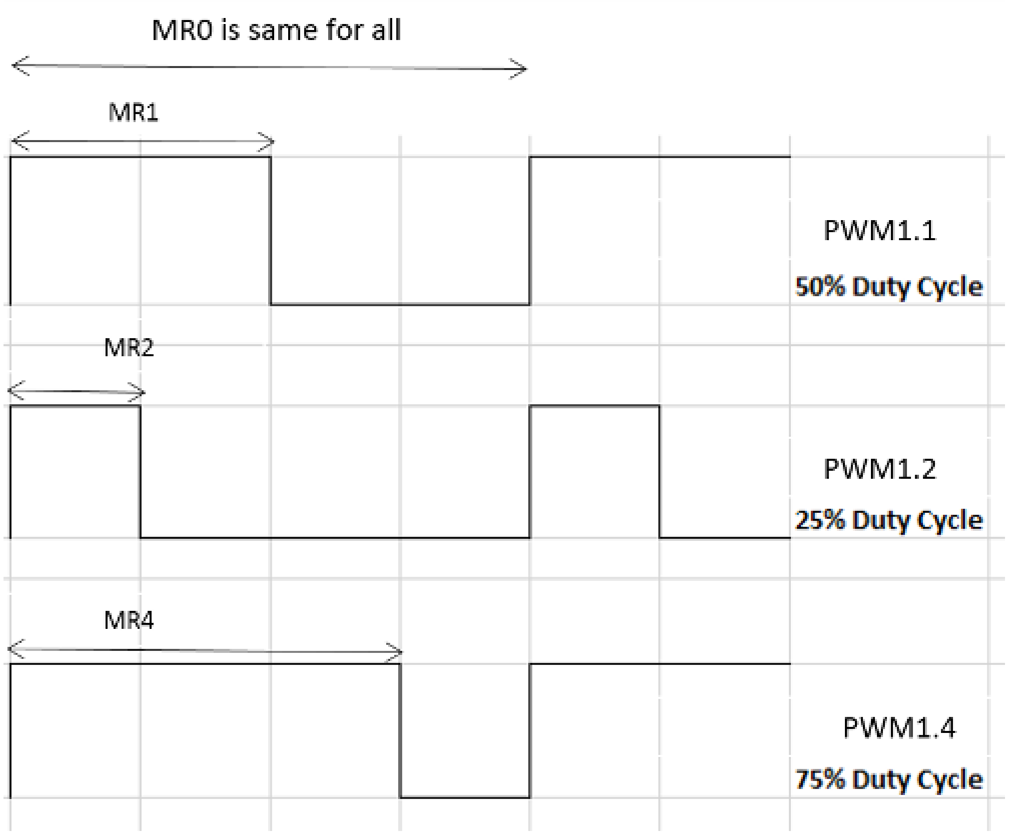
The value of the `MR0` register (aka `PWM1MR0`) determines the period, while any of the `MR1` to `MR6` registers determine the duty cycle for the corresponding `PWM1.1` to `PWM1.6` outputs, as illustrated in the following example.

Example 15. Period and Duty Cycles

If `MR0` is set to 80, then:

Register	Value	Duty Cycle	PWM Channel
`MR1`	40	50%	1 (`PWM1.1`)
`MR2`	20	25%	2 (`PWM1.2`)
`MR4`	20	75%	4 (`PWM1.4`)
`MR5`	72	90%	5 (`PWM1.5`)

The figure below shows the different PWM outputs for the same `MR0`.



Single Edge Controlled PWM

In the example above, the periodic signal on all channels will go high at the beginning of the period, and each channel will be reset when matching the number in the corresponding `MR1` to `MR6` register.

This PWM configuration is called *single edge controlled PWM*.

In summary:

1. Control the period duration of the PWM signal by setting the `MR0` register.
2. Use the appropriate `MRx` register to control the duty cycle of `PWM1.x`, where `x` is a number between 0 and 6.

Example 16. A PWM Period of 1 Second

```
LPC_PWM1->MRx = 1000000; // PWM period is 1 second.
```

To have different PWM channels be set and reset at different times, some PWM channels can be configured as *double edge controlled PWM* signals.

Double Edge Controlled PWM

In double edge controlled, you can control when to set or reset the pulse within the period, and whether to set or reset first.

The `MR0` register still controls the duration of the full period.

Example 17. Double Edge Controlled PWM

PWM channel 2 (`PWM1.2`) is set by `MR1` and reset by `MR2`.

So, setting `MR0` = 100, `MR1` = 50, and `MR2` = 75 will result in a signal that is low at the beginning of the period, becomes high in the middle of the period, and goes back to low in the middle of the second half of the period.

In contrast, setting `MR0` = 100, `MR1` = 75, and `MR2` = 50 will result in a signal that is high at the beginning of the period, becomes low in the middle of the period, and goes back to high in the middle of the second half of the period.

PWM channels can be configured to be *single edge controlled* or *double edge controlled* using the `PWMSELn` bits of the *PWM Control Register* (`PWM1PCR` or `LPC_PWM1→PCR`).

For details, see Table 444 and Table 452 in the LPC176x manual.

PWM vs. Timers

From a hardware point of view, PWM is based on the standard timer block, and inherits all of its features [lpc1769-manual].

Let us review the relation between the timer counter, the prescale register, and the prescale counter. `TC` is a 32-bit register that is incremented every `PR` + 1 cycles of `PCLK`, where `PR` is the *Prescale Register* (`PWM1PR` or `LPC_PWM1→PR` in CMSIS).

Recall that you can use the default value of the `PR` register (0) to simply increment `TC` every `PCLK` pulse.

IF `PR` is set to a non-zero value, `TC`'s frequency would be given by:

$$\texttt{TC}\text{ frequency in Hz} = \frac{\text{System clock}}{\text{PCLKdivisor} \times (\text{PR} + 1)}$$

where *PCLK divisor* is 1, 2, 4, or 8, depending on the setting of the `PCLKSELx` register (default is 4).

For *system clock*, you can use the `SystemCoreClock` variable, which is set by CMSIS to the CPU clock speed.

Example 18. Setting the Prescale Register

To set the prescale register such that TC is incremented every 1 µs (frequency of 1,000,000 Hz):

```
LPC_PWM1->PR = SystemCoreClock / (4 * 1000000) - 1;
```

If MR0 is set to 100, every 100 pulses of the *PWM Timer Counter* register (PWM1TC, or TC for short), a new PWM period starts. That happens even if TC is not reset. This is an important operational difference between pure timers and a PWM signals. The other crucial difference is the control of the duty cycle, which is at the heart of the the PWM concept.

Summary of Important PWM Control Registers

- LPC_PWM1→LER is used to latch the new MRx values. You must use it every time you change any of the MRx values.
- LPC_PWM1→PCR is used to enable PWM1 with single or double edge operation. If ignored, PWM will act as a counter.
- LPC_PWM1→TCR is used to enable, disable, or reset counting in the TC register. You should use it at least once to enable counting.
- LPC_PWM1→MCR is similar to the timers' MCR registers. It can be used to generate interrupts or reset TC when matches occur if needed.

6.4. Tasks

1. Basic operation: Write a program that generates a PWM signal, and use it on an external device.
2. Control a servo motor: Rotate a servo motor 90 degrees to the right, move it back to the neutral position, then rotate it 90 degrees to the left.
3. Show different colors on an RGB LED using at least two PWM signals

6.5. Resources

[]

NXP Semiconductors. 'UM10360 LPC176x/5x User manual'. Rev. 3.1. 2 April 2014. http://www.nxp.com/documents/user_manual/UM10360.pdf

6.6. Grading Sheet

Task	Points
Basic operation	3
Servo Control	7
Bonus: RGB	+2

7. Experiment 7: Serial Communication

Hazem Selmi; Mohannad Mostafa; Ahmad Khayyat 162, 16 April 2017

7.1. Objectives

- Introduction to serial communication protocols
- Using the *Serial Peripheral Interface* (SPI) protocol

7.2. Parts List

- LPC1769 LPCXpresso board
- USB A-Type to Mini-B cable
- Breadboard
- Jumper wires
- Pololu LSM303D 3D compass and accelerometer module

Figure 7. Pololu LSM303D 3D Compass and Accelerometer Module

7.3. Background

In this experiment, you will use one of the serial communication interfaces of the LPC1769 microcontroller, specifically the SPI interface (through the SSP controller), to interact with a digital accelerometer.

7.3.1. SPI Communication Using the LPC1769 Microcontroller

Serial vs. Parallel Communication

Serial communication is the process of sending data one bit at a time, sequentially. In contrast, parallel communication involves sending multiple bits at the same time, as illustrated in the Parallel vs. Serial Communication figure below.

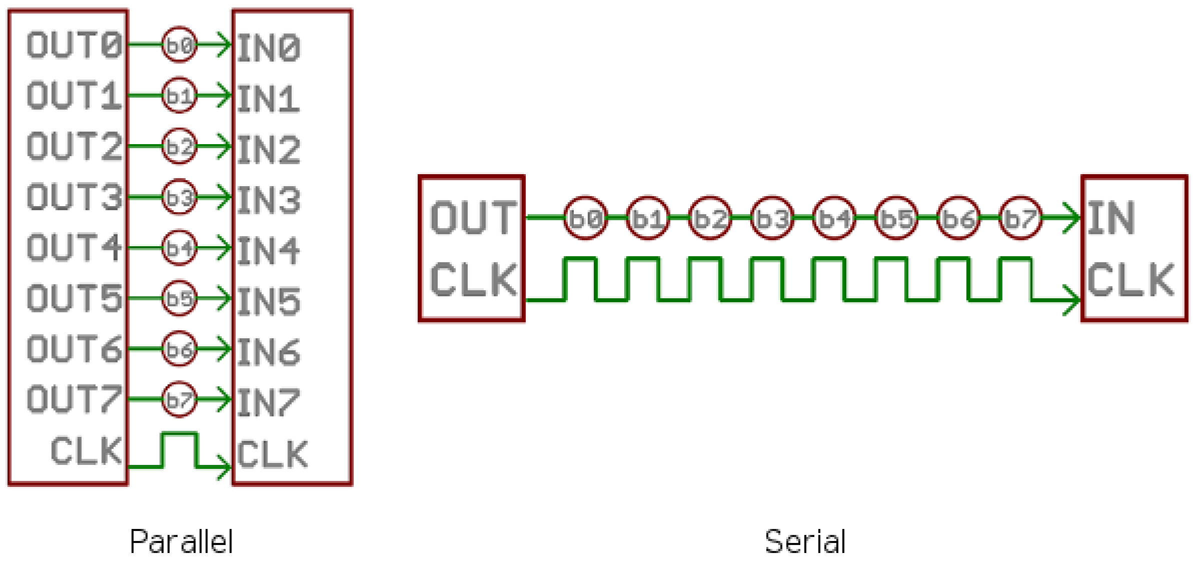


Figure 8. Parallel vs. Serial Communication

Some of the main differences between serial and parallel communication are:

- A parallel link requires more wires, occupying more space and resulting in higher cost.
- To keep all wires in a parallel link synchronized, the link rate is limited. In contrast, serial links can sustain much higher clock rates.
- Parallel links are more susceptible to crosstalk interference.
- Parallel communication between ICs require more pins, increasing the IC cost.
- Parallel communication is easier to implement because it does not require data serialization and deserialization.

Serial communication is becoming more common for transmitting data between a computer and a peripheral device or even another computer, as improved signal integrity and transmission speeds in newer serial technologies have begun to outweigh the parallel bus's advantages.

Serial Communication Protocols

Serial communication standards include USB, FireWire, Serial ATA (SATA), PCI Express (PCIe), and Ethernet. Serial protocols commonly used in embedded systems include UART, I^{2}C, and SPI.

Serial communication protocols can be synchronous or asynchronous. An asynchronous protocol sends a *start signal* prior to each code word, and a *stop signal* after each code word. UART is an asynchronous serial protocol supported by UART interfaces.

A synchronous serial protocol sends a *clock* signal on a dedicated wire. Additional wire(s) are required for data. I^{2}C and SPI are synchronous serial protocols.

LPC1769 Serial Interfaces

The LPC1769 microcontroller provides the following serial interfaces (LPC1769 Manual):

- Two *Synchronous Serial Port* (SSP) controllers, SSP0 and SSP1, with multi-protocol capabilities. They can operate as SPI, 4-wire TI SSI, or Microwire bus controllers.
- A *Serial Peripheral Interface* (SPI) controller. SSP0 is intended to be used as an alternative for the SPI interface. SPI is included as a legacy peripheral.
- Three enhanced *Inter-Integrated Circuit* (I^{2}C) bus interfaces, one supporting the full I^{2}C specification, and two with standard port pins. I^{2}C is pronounced I-squared-C.

- Four UARTs.
- A two-channel CAN controller.
- Ethernet MAC with RMII interface and dedicated DMA controller.
- USB 2.0 full-speed controller that can be configured for either device, host, or OTG operation with an on-chip PHY for device and host functions and a dedicated DMA controller.

In this experiment, we will use the SSP interface configured for the SPI protocol.

Serial Peripheral Interface (SPI)

SPI is a four-wire, full-duplex, master-slave bus that was created by Motorola. There can be only a single master. Multiple slaves are allowed with individual *slave select* (`SS` or `SSEL`) lines. The four wires are:

1. `SCLK`: Serial Clock (output from master)
2. `MOSI`: Master Output, Slave Input (output from master)
3. `MISO`: Master Input, Slave Output (output from slave)
4. `SSEL`: Slave Select (active low, output from master) — one per slave

The microcontroller is usually the master. It uses the `MOSI` pin to send data, and the `MISO` pin to read data. The `SCLK` pin dictates the transmission rate; a bit is sent/received every clock pulse. A simple timing diagram for writing data is shown below.

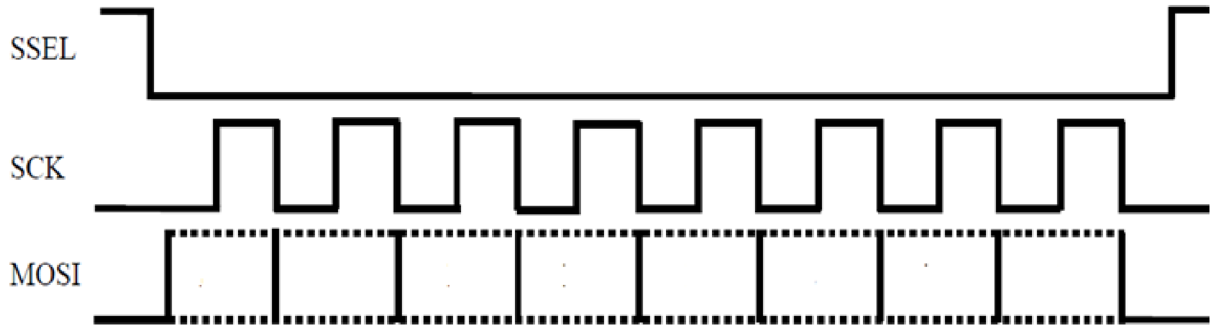


Figure 9. Timing diagram for writing data on a SPI bus

The *slave select* (`SSEL`) signal is used to select the slave in a data transfer. `SSEL` is active low: it must be low before the transaction begins, and must stay low for the duration of the transaction.

To connect multiple slaves, you need a dedicated `SSEL` for each slave. All slaves can share the remaining wires.

Even though the `SSEL` signal is a part of the SPI protocol, it is not uncommon to leave its control to the software instead of the SPI/SSP controller. The LPC176x manual states that "This signal is not directly driven by the master. It could be driven by a simple general purpose I/O under software control." In the LPCXpresso Base Board, `SSEL` is connected to GPIO P2.2. It should be driven low (by software) prior to placing data in the *Data Register* (`DR`), and then switched back to high.

Using SSP/SPI in LPC1769

The section describes how to use the SSP interface of the LPC1769 microcontroller as an SPI interface by listing the involved registers and their functions.

Data Register (`DR`)

The data to be sent serially must be loaded into the SSP *Data Register* (`LPC_SSP1→DR`). The serial transfer rate is controlled by the SSP clock as described below.

The LPC_SSP1→DR register has both a transmitter FIFO and a receiver FIFO.

To transmit the value stored in x, you can use:

```
LPC_SSP1->DR = x;
```

Similarly, to receive a new value and store in x, you can use:

```
x = LPC_SSP1->DR;
```

Every time you send data by writing to the LPC_SSP1→DR register, some data are also received in that same register. Make sure you read that (dummy) data to clear the receiver buffer.

Also, to be able to receive something from a slave, you need to trigger the two way communication by putting dummy data in the DR.

SSP Control Registers

There are two control registers for the SSP1 interface (see LPC17xx.h):

1. SSP1CR0: can be accessed as LPC_SSP1→CR0
2. SSP1CR1: can be accessed as LPC_SSP1→CR1

The CR0 register has 5 fields:

1. Data size (bits 0-3): the number of bits transferred in each frame.
2. Frame Format (bits 4-5): the serial protocol to be used.

   00
   : SPI

   01
   : TI

   10
   : Microwire

   11
   : Not supported

3. Clock Out Polarity (bit 6): should be 0 in our application.
4. Clock Out Phase (bit 7): should be 0 in our application.
5. Serial Clock Rate (SCR) (bits 8-15): used with the *Clock Prescale Register* (CPSR) to control the SSP clock. This is crucial when the SSP peripheral requires a specific value or range of frequencies.

The CR1 register has 4 fields, the most crucial of which is bit 1: *SSP enable*.

In addition to CR0 and CR1, there is the SSP *Clock Prescale Register* (CPSR). The CSPR register contains a single field, CPSDVSR, in bits 0-7. Its remaining bits are reserved (unused).

The SSP clock frequency is calculated using the formula:

$$\text{SSP frequency} = \frac{PCLK}{CPSDVSR \times (SCR + 1)}$$

The SSP's CPSR register must be properly initialized. Otherwise, the SSP controller will not be able to transmit data correctly.

For details, see Tables 371, 372, and 375 in the LPC176x manual.

Exercise

What values of CPSDVSR and SCR will result in the highest SSP frequency?

Exercise

If the frequency of PCLK is 25 MHz, what would be the shortest possible amount of time to generate eight SCLK pulses?

7.3.2. Using the LSM303D Accelerometer

The LSM303D chip is a system-in-package featuring two devices: a 3D digital linear acceleration sensor, and a 3D digital magnetic sensor. It includes both I^2C and SPI interfaces. It also can be configured to generate an interrupt signal for free fall, motion detection, and magnetic field detection. The magnetic and accelerometer parts can be enabled or put into power-down mode separately.

In this experiment, we will focus on the digital accelerometer. Nonetheless, the digital magnetic sensor, or compass, can be used by following similar procedures, as documented in the chip datasheet [lsm303d-manual].

To be able to conveniently use the LSM303D chip, we will be using the Pololu carrier module/board [lsm303d-pololu].

Accelerometers

An accelerometer is an electromechanical device that will measure acceleration forces. These forces may be static, like the constant force of gravity pulling at your feet, or they could be dynamic, caused by moving or vibrating the accelerometer.

An accelerometer can help your project understand its surroundings better. Is it driving uphill? Is it going to fall over when it takes another step? Is it flying horizontally? A good programmer can write code to answer all of these questions using the data provided by an accelerometer. An accelerometer can even help analyze problems in a car engine using vibration testing.

In the computing world, IBM and Apple have been using accelerometers in their laptops to protect hard drives from damage. If you accidentally drop the laptop, the accelerometer detects the sudden freefall, and switches the hard drive off so the heads don't crash on the platters. In a similar fashion, high-g accelerometers are the industry standard way of detecting car crashes and deploying airbags at just the right time. [accelerometers]

The LSM303D SPI Interface

The LSM303D chip provides an SPI interface with the device acting as a slave on the SPI bus. It allows writing and reading the registers of the device. The serial interface interacts with the outside world through

4 wires: `CS`, `SPC`, `SDI` and `SDO`.

Check the LSM303D datasheet. Read the ``*SPI bus Interfaces*" section to find out how to read from and write to the registers of LSM303D.

Using the LSM303D Accelerometer

The accelerometer measures acceleration along the three dimensions, and makes them available in the following registers:

`OUT_X_L_A` *(28h)*, `OUT_X_H_A` *(29h)*

X-axis acceleration data. The value is expressed in 16 bits as 2's complement.

`OUT_Y_L_A` *(2Ah)*, `OUT_X_H_A` *(2Bh)*

Y-axis acceleration data. The value is expressed in 16 bits as 2's complement.

`OUT_X_L_A` *(2Ch)*, `OUT_X_H_A` *(2Dh)*

Z-axis acceleration data. The value is expressed in 16 bits as 2's complement.

A simple program that shows how to read data from the accelerometer is available in the AN3192 Application note document, page 10.

The Directions of the Three Accelerometer Readings figure shows the directions corresponding to positive values along each of the three axes, relative to the chip.

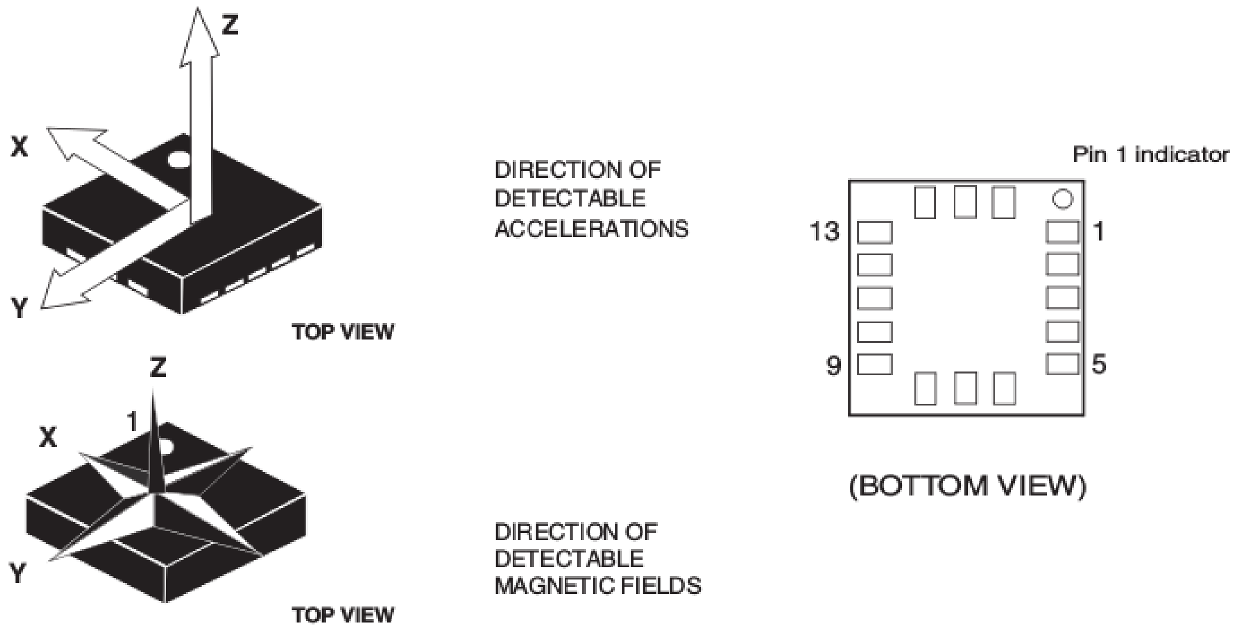

Z
X
Y
TOP VIEW
DIRECTION OF DETECTABLE ACCELERATIONS
Pin 1 indicator
13
1
9
5
(BOTTOM VIEW)
Z
X
1
Y
TOP VIEW
DIRECTION OF DETECTABLE MAGNETIC FIELDS

Figure 10. Directions of the Three Accelerometer Readings

You must configure the `CTRL1` register in order to read the accelerometer data.

Reading data from the accelerometer device is completed in 16 clock pulses. Thus, in order to read the data correctly from the registers, you have 2 options: send multiple 8-bit data, or send 16-bit data. The description is as follows:

1. Send the first 8 bits, which include the read/write bit and the address bits of the register that you want to read. As a result of generating the clock pulses required to send this byte, you will receive dummy data. Then, send another 8 bits of dummy data just to generate the required clock pulses to receive the requested 8-bit data.
2. Send 16-bit data, where the first 8 bits include the read/write bit and the register address, and the next 8 bits contains the data to write, in case of a write command, or dummy data if you are reading.

7.4. Tasks

1. Use the LPC1769's SSP/SPI interface to read the accelerometer data from the LSM303D device.
2. Write a simple application to indicate different stationary positions. For example, indicate whether the device is tilted to the right or to the left, tilted forward or backward, and whether it's facing upward or downward. Use some output device to reflect this data in real-time. The following table summarizes the readings corresponding to each of the six stationary positions.

Stationary Position	**Ax**	**Ay**	**Az**
Z down	0	0	-
Z up	0	0	+
Y down	0	-	0
Y up	0	+	0
X down	-	0	0
X up	+	0	0

7.5. Resources

[]

Embedded Artists AB. 'LPCXpresso Base Board Rev B User's Guide'. 2013-01-25. http://www.embeddedartists.com/sites/default/files/support/xpr/base/LPCXpresso_BaseBoard_rev_B_Users_Guide.pdf

[]

NXP Semiconductors. 'UM10360 LPC176x/5x User manual'. Rev. 3.1. 2 April 2014. http://www.nxp.com/documents/user_manual/UM10360.pdf

[]

Dimension Engineering Inc. 'A Beginner's Guide to Accelerometers'. Retrieved: 2015-11-7. http://www.dimensionengineering.com/info/accelerometers

[]

STMicroelectronics. 'LSM303D: Ultra compact high performance e-Compass 3D accelerometer and 3D magnetometer module — Datasheet — preliminary data'. Doc ID 023312 Rev 1. June 2012. https://www.pololu.com/file/0J703/LSM303D.pdf

[]

Pololu Corporation. 'LSM303D 3D Compass and Accelerometer Carrier with Voltage Regulator'. Retrieved: 2015-11-7.
https://www.pololu.com/product/2127

[]

STMicroelectronics. 'AN3192 Application note: Using LSM303DLH for a tilt compensated electronic compass'. Doc ID 17353 Rev 1. August 2010.
https://www.pololu.com/file/0J434/LSM303DLH-compass-app-note.pdf

7.6. Grading Sheet

Task	Points
Operate a seven-segment display using the SSP/SPI interface	7
Discussion	3

8. Experiment 8: Microcontroller on an FPGA

Mohannad Mostafa; Ahmad Khayyat; Hazem Selmi 162, 23 April 2017

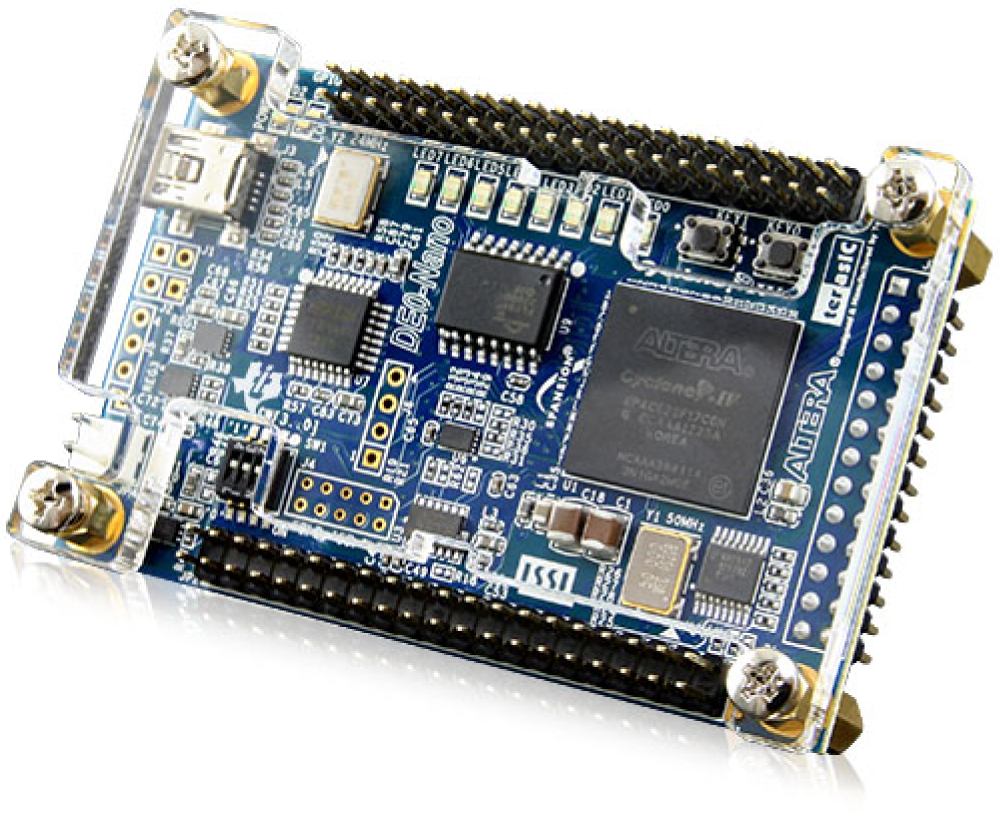
8.1. Objectives

In this experiment, you will learn about the hardware of a microcontroller by:

- Building a microcontroller system on an FPGA by integrating Altera’s Nios II soft processor and a few peripherals
- Writing software for the FPGA-based microcontroller system

8.2. Parts List

- Altera DE0-Nano FPGA board



- USB A-Type to Mini-B cable

8.3. Background

This experiment involves two main tasks:

1. Building the hardware for a microcontroller system using an FPGA device.

   We will use Altera’s *DE0-Nano* FPGA development board, which incorporates a small FPGA device and a number of peripherals. To create the microcontroller system, we will configure the FPGA device to implement a soft processor and a number of required components for the processor to function properly.

   To configure the FPGA, we will use Altera’s *Quartus II* design software. To build the soft processor system, we will use Altera’s *Qsys* system integration software to implement a system around Altera’s Nios II soft processor core.

2. Developing software to be executed on your microcontroller system.

   We will use Altera’s Eclipse-based *Nios II Embedded Design Suite (EDS)* software development environment to build software for the Nios II-based hardware system.

Altera Software

All software tools that you need to conduct this experiment are either installed on the lab machines or will be provided to you. However, you can also obtain and install all the required software for this experiment freely from Altera's website. You will want to download and install *Quartus II Web Edition software*, which includes *Qsys*, the *Nios II EDS*, and *Altera IP Library*.

For more details on available Altera software, software licensing, download options, and hardware and software requirements, consult the Altera Software Installation and Licensing document.

8.3.1. The DE0-Nano FPGA Board

The DE0-Nano is a low-cost, low-power, portable, compact board (49 mm x 75 mm) aimed at developing embedded soft processor systems using the Nios II processor.

Board Features

- Three-axis accelerometer with 13-bit resolution
- Eight-channel, 12-bit resolution analog-to-digital (A/D) converter
- Expansion headers: two 40-pin headers and one 26-pin header
- Two-pin external power header
- 32-MB SDRAM
- 2-Kb EEPROM
- Eight green LEDs
- Four dual in-line package (DIP) switches
- Two push-button switches

8.3.2. The Nios II Processor

Nios II is the name of Altera's proprietary soft processor architecture.

Nios II is a RISC machine. A soft processor is a processor that can be implemented on reconfigurable logic, e.g. an FPGA.

Xilinx also has a soft processor architecture, named MicroBlaze.

Nios II Processor Features

- Full 32-bit instruction set, data path, and address space
- 32 general-purpose registers
- 32 interrupt sources
- External interrupt controller interface for more interrupt sources
- Optional floating-point instructions for single-precision floating-point operations
- Access to a variety of on-chip peripherals, and interfaces to off-chip memories and peripherals
- Hardware-assisted debug module enabling processor start, stop, step, and trace under control of the Nios II software development tools
- Optional memory management unit (MMU) to support operating systems that require MMUs
- Software development environment based on the GNU C/C++ tool chain and the Nios II Software Build Tools (SBT) for Eclipse

The Nios II Processor Reference Handbook states that:

> A Nios II processor system is equivalent to a microcontroller or "computer on a chip" that includes a processor and a combination of peripherals and memory on a single chip. A Nios II processor system consists of a Nios II processor core, a set of on-chip peripherals, on-chip memory, and interfaces to off-chip memory, all implemented on a single Altera device. Like a microcontroller family, all Nios II processor systems use a consistent instruction set and programming model.
>
> — Nios II Processor Reference Handbook

For more information on the Nios II processor, consult its extensive documentation.

8.3.3. Design Flow

Unlike previous experiments, we need to create the hardware of the microcontroller system before we can program it.

In order to create a Nios II soft processor system on the Altera DE0-Nano FPGA board, and write software for it, you are going to use the following software tools:

System Builder
: used to generate a preconfigured Quartus II project for the DE0-Nano FPGA development board.

Quartus II
: used to compile all design files, including those generated by Qsys, into an FPGA configuration file, known as an *SRAM Object File* (`.sof`), which can be downloaded into the FPGA device to implement the designed system.

> *Altera vs. Xilinx Tools*
>
> Quartus II is the design software used to develop hardware for Altera FPGAs. DE0-nano is a development board that contains an Altera FPGA chip.
>
> In contrast, for Xilinx FPGAs, ISE design suite is the design software used to develop hardware for Xilinx FPGAs, and Spartan, for example, is a board that contains a Xilinx FPGA chip.

Qsys
: used to specify the Nios II processor core(s), memory, and other components your system requires. Qsys automatically generates the interconnect logic to integrate the components in the hardware system. Qsys is integrated with Quartus II software. You can start it from *Tools* menu in Quartus II.

Nios II EDS
: the Nios II Embedded Design Suite includes Nios II Software Build Tools (SBT) for Eclipse, which is an eclipse installation preconfigured to use a set of plugins to support developing software for the Nios II processor. To create a new Nios II C/C++ application project, the Nios II SBT for Eclipse uses information from the files generated by Qsys to learn about the target hardware.

Here is a summary of the general flow steps; the details will come later:

1. Use the *System Builder* utility to generate a Quartus II project preconfigured for the DE0-Nano board. This step is specific to the DE0-Nano board.
2. Use *Qsys* to generate the hardware description of your processor system. In addition to the HDL files,

Qsys generates an `.sopcinfo` file that describes the system.

3. Use *Quartus II* to compile the hardware description generated by Qsys into an FPGA configuration file (`.sof`), and to download the configuration file into the FPGA to implement the system's hardware.
4. Use *Nios II SBT for Eclipse* to write the software that is executed by the Nios II CPU. Nios II SBT for Eclipse learns about the hardware from the Qsys-generated `.sopcinfo` file, and is thus able to compile your software for the Nios II generated hardware.

8.3.4. Creating a Quartus II Project

The Quartus II project will eventually contain all the information required to generate and implement the hardware of our system.

The DE0-Nano kit ships with a convenient software utility called *System Builder*, which creates preconfigured Quartus II projects for the DE0-Nano board. For example, it automatically configures the project to target the specific FPGA device in the DE0-Nano, and configures the pin locations for the selected peripherals.

Run the DE0-Nano's System Builder utility, and choose the following configuration options:

- CLOCK
- LED x 8
- EEPROM, 2KB
- SDRAM, 32MB

Then, press *Generate* to create a Quartus II project. After that, open the generated project in Quartus II by opening the `.qpf` file. In the next section, we will use Qsys from within this project.

❗ Avoid using directories with spaces in their names for your Quartus II or Nios II EDS projects.

ℹ Since the purpose of this experiment is to understand the makeup of a microcontroller system, it is suggested to create a minimal system by only including the few peripherals listed above. But you are welcome to include any of the other available peripherals.

For example, the DE0-Nano FPGA board has a built-in accelerometer. You are free to try to use it if you manage to complete the listed tasks in this experiment!

8.3.5. Building the Processor System Using Qsys

Qsys allows you to put together the hardware components that make up your microcontroller system, and to create all the required connections, including the system bus.

We would like to build a Nios II system that includes the following hardware components:

- Nios II/s core with 2 KB of instruction cache
- 20 KB of on-chip memory
- Timer
- JTAG UART
- Eight output-only parallel I/O (PIO) pins
- System ID component

For more information about these and other components, refer to the Embedded Peripherals IP User Guide.

To build this system, run Qsys from the *Tools* menu in Quartus II, and follow the instructions in the Nios II Hardware Development Tutorial, page 1-11 (*Define the System in Qsys* section).

Qsys Errors

While you are adding the components, connecting them, and configuring them, there will be error messages disappearing gradually till you correctly complete your design. Theses error messages can be useful in reminding you of any missed step.

Qsys Notes

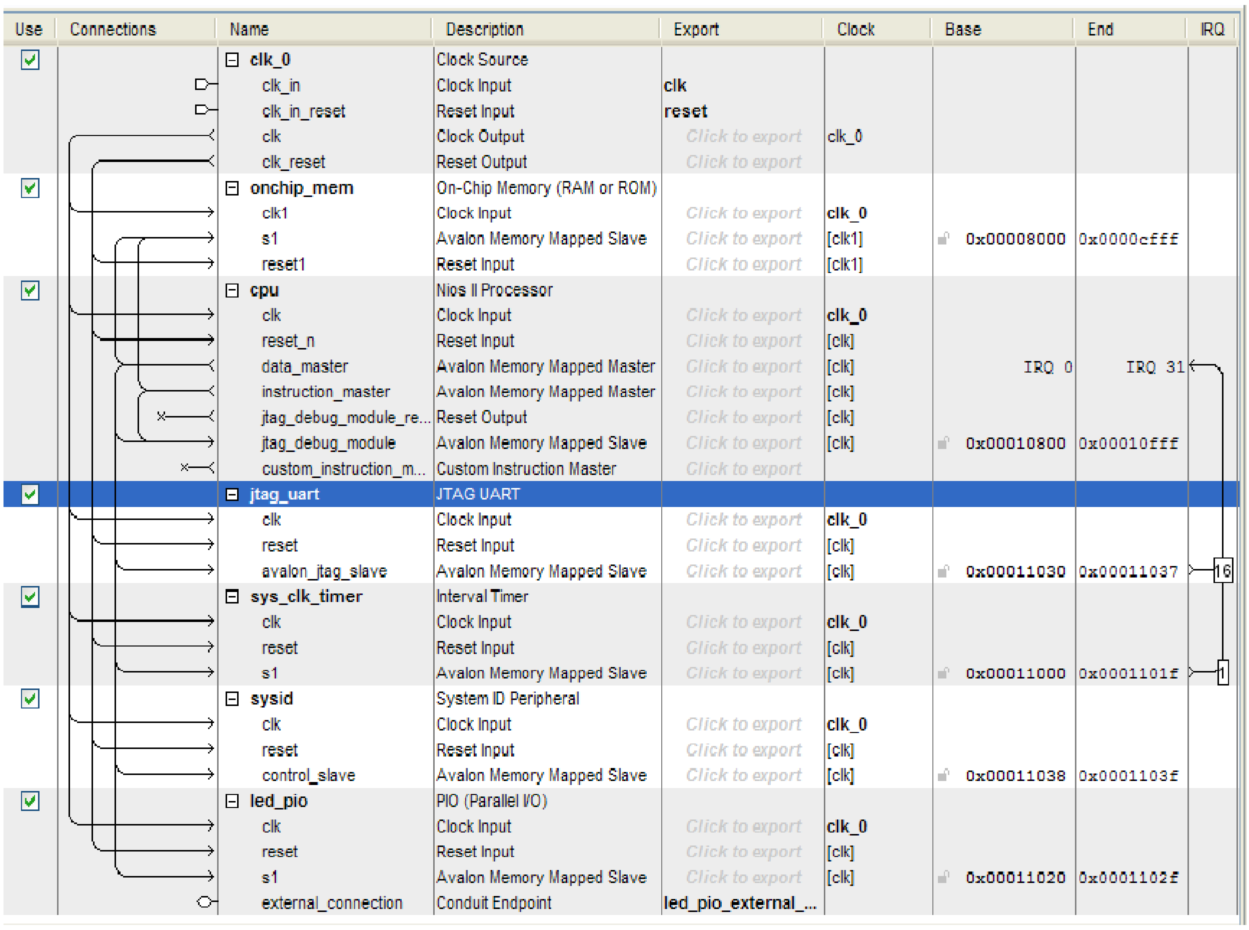
1. Edit the export column for the three components: `clk_in`, `clk_in_reset`, and `external connection`.
2. Edit the name of the memory component to use a simple short name.
3. After adding the CPU core, use the name of your memory component as *Reset Vector memory* and *exception Vector memory*.
4. Edit the IRQ numbers in the IRQ column to be 16 for the *JTAG UART* component, and 1 for the *timer*.
5. Edit the *PIO* component name to a simple short name that you can remember. You will need it later.
6. Generate the base addresses automatically by choosing *Assign Base Addresses* from the *Tools* menu in Qsys.
7. Finally, use the *generate* button in Qsys to generate the project files and save them in a known directory.

Qsys Components

By following the Nios II Hardware Development Tutorial, you may have some questions about some of the components. Here are some answers for such anticipated questions:

1. Although the tutorial asks you to set the cashe size, we don't care about the cache in this experiment. We can use a CPU with no cache. It makes no difference for the purposes of this experiment.
2. JTAG UART is used by the development environment to communicate with the microcontroller that we are creating on the FPGA, especially to download and debug software. JTAG is a standard created for this purpose specifically.
3. System ID is similarly used by the development tools to identify the target hardware and determine which software to download to which hardware. You can set the target system ID value in the IDE to match the value you may set in the System ID hardware component, in case you use a non-default value.
4. Usually, IRQ numbers need to be configured in device drivers or system software. The values set for the JTAG UART and the timer components are the numbers configured in the base system software generated by the Nios II IDE.

Your completed Qsys system should look like this:



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
✓		clk_0	Clock Source					
		clk_in	Clock Input	clk				
		clk_in_reset	Reset Input	reset				
		clk	Clock Output	Click to export	clk_0			
		clk_reset	Reset Output	Click to export				
✓		onchip_mem	On-Chip Memory (RAM or ROM)					
		clk1	Clock Input	Click to export	clk_0			
		s1	Avalon Memory Mapped Slave	Click to export	[clk1]	0x00008000	0x0000cfff	
		reset1	Reset Input	Click to export	[clk1]			
✓		cpu	Nios II Processor					
		clk	Clock Input	Click to export	clk_0			
		reset_n	Reset Input	Click to export	[clk]			
		data_master	Avalon Memory Mapped Master	Click to export	[clk]	IRQ 0	IRQ 31	
		instruction_master	Avalon Memory Mapped Master	Click to export	[clk]			
		jtag_debug_module_re...	Reset Output	Click to export	[clk]			
		jtag_debug_module	Avalon Memory Mapped Slave	Click to export	[clk]	0x00010800	0x00010fff	
		custom_instruction_m...	Custom Instruction Master	Click to export				
✓		jtag_uart	JTAG UART					
		clk	Clock Input	Click to export	clk_0			
		reset	Reset Input	Click to export	[clk]			
		avalon_jtag_slave	Avalon Memory Mapped Slave	Click to export	[clk]	0x00011030	0x00011037	16
✓		sys_clk_timer	Interval Timer					
		clk	Clock Input	Click to export	clk_0			
		reset	Reset Input	Click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Click to export	[clk]	0x00011000	0x0001101f	1
✓		sysid	System ID Peripheral					
		clk	Clock Input	Click to export	clk_0			
		reset	Reset Input	Click to export	[clk]			
		control_slave	Avalon Memory Mapped Slave	Click to export	[clk]	0x00011038	0x0001103f	
✓		led_pio	PIO (Parallel I/O)					
		clk	Clock Input	Click to export	clk_0			
		reset	Reset Input	Click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Click to export	[clk]	0x00011020	0x0001102f	
		external_connection	Conduit Endpoint	led_pio_external_...				

Figure 11. Qsys complete system

8.3.6. Integrate the Qsys System into the Quartus II Project

To integrate the Qsys system with the Quartus II project, here is a summary of what we need to do:

1. Add the Qsys system to the Quartus II project.
2. Instantiate the Qsys system
3. Connect the ports of the Qsys sytem to signals in the Quartus II project.

For Quartus II to recognize the Qsys system, the Qsys system, represented by its Quartus II IP file (`.qip`), must be added to the Quartus II project as follows:

1. Make sure the project generated by *System Builder* is open in Quartus II.
2. From the Quartus II menu, select *Project > Add/remove Files in project*
3. Click the browse button (…) next to the *File name* field
4. Select the file `<qsys_project_directory>/synthesis/<qsys_project_name>.qip`
5. Click *Add* to include `.qip` file in the project, then click *OK* to close the dialog box

To instantiate the Qsys-generated Nios II system, and to connect each port of the Qsys system instance to the appropriate signal in the top-level module of the Quartus II project, use the following Verilog instantiation code in the top-level module of your Quartus II project, which is typically named `<quartus_project>.v`, where `<quartus_project>` is the name of your Quartus II project.

Verilog Code to Instantiate the Qsys-Generated System

```
<qsys_project> u0(
    .clk_clk (CLOCK_50),
    .reset_reset_n (1'b1),
    .led_pio_external_export (LED)
);
```

About the Qsys-system-instantiation Verilog Code

In the Verilog code above, replace `<qsys_project>` with the name of your Qsys project.

The code creates an instance, named `u0`, of the Qsys system, and maps, i.e. connects, the ports of the Qsys system (the names following the periods) to signals declared in the module in which this code resides (the names between parentheses). `1'b1` is a single-bit constant value of `1`.

The exported port names of the Qsys system are derived from the Qsys system definition.

8.3.7. Compile and Download the Hardware Design

The Quartus II hardware compiler consists of a set of modules that perform different compilation steps. The modules are *Analysis & Synthesis*, the *Fitter*, the *Assembler*, and the *TimeQuest Timing Analyzer*. To obtain the downloadable `.sof` FPGA configuration file, we need to run the assembler. Running the assembler will trigger all other required modules.

After compiling the Quartus II Project, connect the DE0-Nano board to your PC in order to download the hardware design.

To download the FPGA configuration data file (`.sof`) to a the FPGA device, you use Altera's USB-Blaster download cable, which interfaces a USB port on your host computer to the Altera FPGA.

The USB-Blaster cable requires a driver for the host computer to recognize it. For details on using the USB-Blaster and installing its driver, refer to the USB-Blaster Download Cable User Guide.

The driver has already been installed on the lab PCs.

To download your hardware design to the FPGA:

1. Run the programmer from the *Tools* menu in Quartus II
2. Click the *Hardware Setup* button and choose *USB-Blaster* if it is not selected
3. Click the *Start* button to start downloading the `.sof` file to the FPGA chip on your board.

Don't close the *OpenCore Plus Status* dialog when it appears.

For more details on downloading your design to the FPGA, refer to the *Download the Hardware Design to the Target FPGA* section of the Nios II Hardware Development Tutorial (page 1-31).

8.3.8. Software Development Using Nios II SBT

Now, you have a Nios II hardware system running on the Altera FPGA board. To make use of this system, we need to write some software to be executed on it.

To be able to that, you need a toolchain (compiler, assembler, debugger) that can compile code for the Nios II CPU. We will use Altera's *Nios II SBT for Eclipse*, which is already installed on lab machines.

You can open and then edit some Nios II example programs as follows: . Select *File* > *New* > *Nios II Application and BSP from Template* . In the wizard, browse to your Qsys project directory, and open the SOPC Information File (`.sopcinfo`) of your design. . Choose the program that you would like to run. . Name your software project. . Click *Finish*.

We will first start with a simple program to explore the software development process. We will use the *Hello World Small* template program by following the instructions on the Nios II Hardware Development Tutorial, page 1-32 (*Develop Software Using the Nios II SBT for Eclipse* section), only use the *Hello World Small* template instead of the *Count Binary* template.

The difference between the *Hello World Small* template and the *Hello World* template is that the former is configured to generate an optimized-for-space program that would fit in the small on-chip memory that was created in Qsys.

You can use the *Hello World* template instead of the *Hello World Small* template, but you would then need to adjust the properties of the BSP project in order to minimize the memory footprint of the software, as described on page 1-34 of the Nios II Hardware Development Tutorial.

To make the program slightly more interesting, replace your code with the one on page 1-9 of the My First Nios II Software Tutorial.

In the function call `IOWR_ALTERA_AVALON_PIO_DATA`, replace the first argument with your system's base address of the PIO peripheral. Look the address up in your `system.h` file.

To understand how this program works, read the *Why The LED Blinks* section on page 1-10.

8.4. Tasks

8.4.1. Build and Download the Hardware Design

1. Using the *System Builder* program, create a Quartus II project for the DE-Nano board. Configure your project to use the board's CLOCK, LEDs, EEPROM, and SDRAM.
2. Build a Nios II system using Qsys.
3. Instantiate your Nios II system in the Quartus II project.
4. Compile and download the hardware design to the DE0-Nano board.

8.4.2. Build and Download the Software

1. Create a software project for your Nios II system using Nios II SBT for Eclipse. Use the Hello World template.
2. Run Hello World application on your Nios II system on the DE0-Nano board.
3. Create and run another application that blinks an LED on the DE0-Nano.
4. Create a third program that blinks all eight LEDs on the DE0-Nano sequentially.

8.4.3. Discussion

- What peripherals are readily available for inclusion in this microcontroller system? (list three)
- What peripherals would you add to your microcontroller systems?

- What is the address of your PIO peripheral, which is driving the LEDs?
- How can you change it?

8.5. Resources

[]

Altera Corporation. 'Altera Software Installation and Licensing'. MNL-1065. 2014.12.15.
https://www.altera.com/en_US/pdfs/literature/manual/quartus_install.pdf

[]

Altera Corporation. 'Documentation: Nios II Processor'.
https://www.altera.com/products/processors/support.html

[]

Altera Corporation. 'Nios II Processor Reference Handbook'. NII5V1-13.1. February 2014.
https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/nios2/n2cpu_nii5v1.pdf

[]

Altera Corporation. 'Embedded Peripherals IP User Guide'. UG-01085. 2014.24.07.
https://www.altera.com/en_US/pdfs/literature/ug/ug_embedded_ip.pdf

[]

Altera Corporation. 'Nios II Hardware Development Tutorial'. TU-N2HWDV-4.0. May 2011.
https://www.altera.com/en_US/pdfs/literature/tt/tt_nios2_hardware_tutorial.pdf
Newer revision (Quartus II 14.0+): 'Nios II Gen2 Hardware Development Tutorial'. AN-717. 2014.09.22
https://www.altera.com/en_US/pdfs/literature/an/an717.pdf

[]

Altera Corporation. 'USB-Blaster Download Cable User Guide'. UG-USB81204-2.5. April 2009.
https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_usb_blstr.pdf

[]

Altera Corporation. 'My First Nios II Software Tutorial'. TU-01003-2.1. December 2012.
https://www.altera.com/en_US/pdfs/literature/tt/tt_my_first_nios_sw.pdf

8.6. Grading Sheet

Task	Points
Build the processor system using Qsys	2
Instantiate the processor system in a Quartus project	2
Run the *Hello World* program	2
Run an LED-blinking program	2
Discussion	2

9. Programming Assignment: Seven-Segment Display and C Libraries

Hazem Selmi; Ahmad Khayyat 162, 8 March 2017

9.1. Objectives

- Using seven-segment displays
- Organizing your code into libraries

9.2. Parts List

- LPC1769 LPCXpresso board
- USB A-Type to Mini-B cable
- Breadboard
- Seven-segment display
- 330-Ohm Resistors
- Jumper wires

9.3. Background

9.3.1. Seven-Segment Displays

Seven-segment displays can be useful in many types of projects as an output device.

The part number of the seven-segment display included in your kit is LTD-4608JR.

You should be able to use the data sheet to figure out how to display a number on the seven-segment display.

This seven-segment display part can show two digits; it is actually two seven-segment displays in one part, with common seven pins connected to them both. Simply setting the seven pins would show the same digit on both displays.

To show a different digit on each display simultaneously, we need to employ time multiplexing. Each display has its own *common* pin, which enables and disables that display. Setting the seven pins affects enabled displays only. Alternatively enabling each display while disabling the other allows you to set each display to show a different digit. However, leaving a display disabled will clear any previously shown digit. Therefore, time multiplexing is required, whereby the two displays are continuously enabled alternatively.

9.3.2. Building a C Library

A C library consists of one or more `c`-files and `h`-files that can be used as part of a project to make your code modular and more efficient.

The `c` file

The `c` file can have one or more, usually related, C functions. To make your code readable and easily reusable, it is crucial to carefully comment your code and wisely choose the function and parameter names.

In LPCXpresso, you can add new C source files to your project by right-clicking the `src` folder and adding a new *source file*.

The `h` file

To call functions defined in a library, your source file needs to tell the compiler what the function signatures, or prototypes, are. Instead of having to write the function prototypes every time you use a library, libraries provide a file containing those prototypes, without the function implementations. Such a file has the `.h` extension, and is called a *header file*.

The c files contain the full functions that the library defines. In contrast, the h files contain only the function prototypes, without their implementations. This allows you to use libraries without necessarily having access to their full sources, just their function prototypes and binary implementation. Moreover, header files may contain other macros, or preprocessor directives, such as define and include statements.

To use a library, you only need to include its header file(s) in your program. Including a file is equivalent to inserting the contents of the included file in your program file.

As with any source file, it is good practice to comment your library implementation (c files) and headers (h files).

To avoid including multiple copies of the same header file, which would result in function name collision, it is recommended to use the following template for your header files.

```
#ifndef LIBRARY_NAME_H
#define LIBRARY_NAME_H

/* Header file statements */

#endif /* LIBRARY_NAME_H */
```

Exercise

Study the above template and explain how it ensures that a header file that uses this template will always be included at most once.

The above template is automatically generated for you by LPCXpresso when you choose the *default C-header template* when adding a new header file.

9.3.3. A Seven-Segment Display Library

We want to write a C library that would help us use the seven-segment display, by abstracting out the details of decoding each digit into the actual bit pattern that will set the display to show the required digit. The ultimate objective is to set the pins that are connected to the display to show the digit. However, for the library to be reusable, it would be wise not to tie it to specific pins, and leave the mapping of the bits to the actual I/O pins outside the library.

Therefore, the library would include one public function that acts as a BCD-to-seven-segment decoder. Think of the function as a BCD-to-seven-segment decoder chip. If you use such a chip, you still need to map each of its outputs to the seven-segment display pins.

The library function will generate the seven bits. Then, in your main program, or perhaps in a separate function in your main program file, you assign each of these bits to one of the seven-segment display pins. This way, you can use the library regardless of where the seven-segment display is connected.

Modular design, which is achieved by appropriate abstraction, improves the reusability of your code, which makes you more productive.

In this case, the library function abstracts the decoding of a BCD digit to seven-segment bits, and does nothing else.

It is also useful to write another library for reading and writing to GPIO pins. The prototypes for the functions in such a library may look something like the following.

```
void set_gpio_pin(int port, int pin, int value); // write to an output pin
int  get_gpio_pin(int port, int pin);            // read an input pin
```

In this case you need to call the write function seven times (once per bit).

Consider using the newer 8-bit `uint8_t` type instead of the typically 32-bit `int` type when you can guarantee that 8 bits are enough to represent the values. `uint8_t` is a part of the C99 standard, among other similarly named types, which are guaranteed to have the same exact width across all platforms, unlike the plain `int` type.

`uint8_t` is defined in `inttypes.h` and `stdint.h`.

9.4. Tasks

9.4.1. Display a Hard-wired Number on the Seven-Segment Display

1. Use the seven-segment display along with VDD and GND pins on the microcontroller to display a number. You do not need to program the microcontroller. Display the same number on both displays.

   This task will help you confirm your understanding of how the seven-segment display works.

9.4.2. Display a Number on the Seven-Segment Display by Software

2. Write a program that displays different numbers on the seven-segment display. For example, a program that counts from 0 to 9. Display the same number on both displays.

9.4.3. Write a Seven-Segment Display Library

3. Write a reusable library consisting of a `c` file and a corresponding header (`h`) file for decoding a BCD digit to seven-segment display bits. The library should not be tied to any specific I/O pins; it just implements a seven-segment display decoder.
4. Write a library for easily accessing GPIO pins (reading and writing).
5. Write a program to test your seven-segment display decoder library and your GPIO library.

9.4.4. Display Two Different Digits Simultaneously

6. Write a program that implements time multiplexing to show a different digit on each display.