Experiment 1: Development Platform

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
142	1 February 2015		A

Contents

1 Objectives

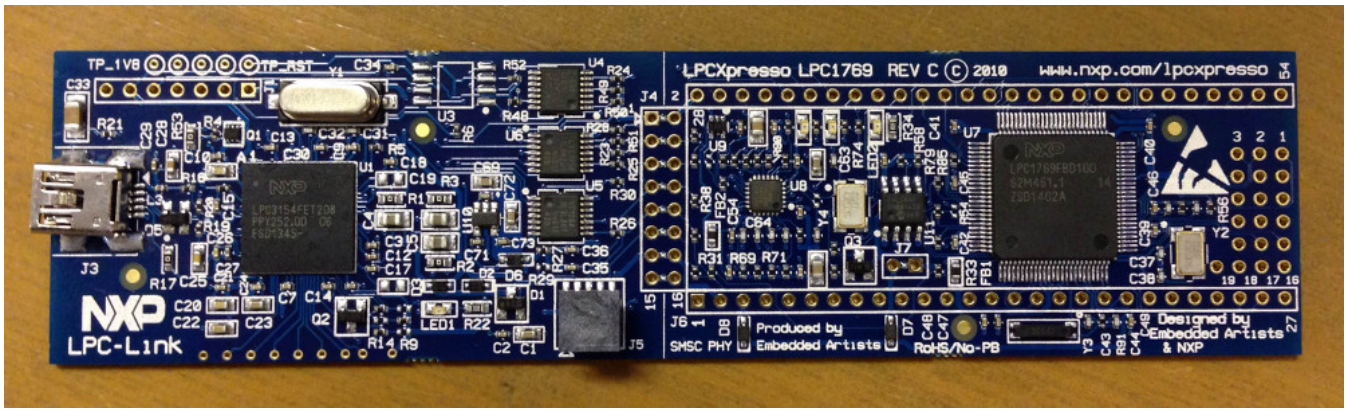
- Get familiar with the development platform:

Hardware	microcontroller, development board, peripherals
Software	IDE, compiler, debugging, programming

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

2 Parts List

- LPC1769 LPCXpresso board



- USB A-Type to Mini-B cable

3 Background

3.1 Microcontroller

LPC1769 is a microcontroller manufactured by NXP. 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] is an essential resource for any developer.

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.

For an overview of the LPCXpresso platform (hardware and software), see [lpcx-platform].

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.

3.3.1 Installation

To install your copy of the LPCXpresso IDE:

1. Using a web browser, navigate to the download page:
   http://www.lpcware.com/lpcxpresso/download
2. Under *Product downloads*, follow the link that matches your operating system. LPCXpresso supports Windows, Linux, and Mac OS X.

   Note
   You may also want to get a copy of the *LPCXpresso 7 User Guide* listed at the top of the page.

3. Click on the first download link in the download page for your operating system. The link text looks like:

   ```
   Download the current release (7.x.y) of LPCXpresso 7 for ...
   ```

4. Run the downloaded installer.

3.3.2 Activation

To activate your copy of the LPCXpresso IDE:

1. Using a web browser, log into the activation web page:
   http://www.lpcware.com/lpcxpresso/activate
   Register a new account if you do not have one.
2. In the LPCXpresso IDE, select Help > Product Activation > Create Serial number and Activate.
3. Copy the shown serial number, and paste it into the activation web page from step 1.
4. You will receive an *LPCXpresso Key Activation* email, which includes the *Product Activation Key*.
5. In the LPCXpresso IDE, select *Help* > *Enter Activation Code*, and copy the *Product Activation Key* received in the email into this form.

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

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 available for the user.

Each of the I/O pins is 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[21] is pin 22 in port1. In some cases, the same pin is called by its actual pin number on the chip; For example, P0.17 is called pin-61 and P1.22 is called pin-35.

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.

Note
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.

Note
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.

3.4.1 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.

3.4.2 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.

3.4.3 Naming 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`.

One way to give these registers some names is as follows:

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

3.5 Data Structures in C

In the C programming language, the `struct` keyword is used to define a complex data type as a group of variables. The resulting data type can then be used to declare variables, each of which would contain all of the listed variables in the structure definition.

Tip
A C structure variable references a contiguous block of physical memory.

Defining a Structure and Declaring a Variable

```
struct point {
    int x;
    int y;
};

struct point p;
```

In the example above, `p` is an instance of the `struct point` structure.

Structure Aliases Using typedef It is possible to use a shorter name to identify the structure type using the `typedef` keyword. The following example results in a variable `p` that is identical to the `p` variable declared in the previous example.

```
struct point {
    int x;
    int y;
};

typedef struct point Point;

Point p;
```

Note
C is case-sensitive. In the example above, `point` is different from `Point`.

Combining typedef and struct The `struct` and `typedef` statements can be combined into a single statement.

```
typedef struct point {
    int x;
    int y;
} Point;
```

In fact, when combined, the name immediately following the `struct` keyword, also known as the *structure tag*, can be removed.

```
typedef struct {
    int x;
    int y;
} Point;
```

Accessing Fields In the examples above, the variable `p` is of type `Point`, and thus contains two integer fields, `x` and `y`. To access the fields, the dot operator is used (`.`).

```
p.x = 5;
int z = p.y;
```

Pointers to Structures It is common to refer to structure variables by their address, or pointer, instead of the variable itself. This is especially useful when passing a structure instance as an argument to a function to avoid copying possibly-large variables during the function call.

Note
C passes arguments by value, not by reference. Pointers can be used to pass arguments by reference.

Pointers to structure variables are also useful for declaring another structure instance as a field within the structure.

Pointers to structures are used like any other pointers. The `&` operator retrieves the address of a variable, which can be stored in a pointer variable. The `*` operator is used to declare a pointer variable, and to dereference a pointer in order to access the variable it points to.

```
Point p;                /* an instance variable */
Point *pointer = &p;    /* a pointer to the same instance */
p.x = 5;
(*pointer).x = 5;       /* Identical to the previous statement */
```

Because it is very common to refer to structures using pointers instead of structure variables, a special operator, the arrow (`->`), is available to access a field of a structure using its pointer.

```
p.x = 6;
pointer->x = 6;         /* Identical to the previous statement */
```

Unions A union in C is a data type that stores different data types in the same memory location. There are two main uses of unions:

1. Storing mutually-exclusive data. If you never need to store both variables `a` and `b` at the same time, you can define them using a union so that they use the same memory space. This also applies if you want to declare a generic variable that can have multiple types.
2. Accessing the same data in different ways, or as different data types. For example:

```
union {
    uint32_t x;
    struct {
        uint16_t xL;
        uint16_t xH;
    };
};
```

Here, `x` refers to a 32-bit integer, whereas `xL` and `xH` refer to the low and high 16 bits of that 32-bit integer, respectively. Changing the value of `x` would also change the values of `xL` and `xH`, depending on which bits have changed.

3.6 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].

3.7 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.

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.

Important
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.

EXAMPLES

- To control the direction of pins in port 2, use the following register:

```
LPC_GPIO2->FIODIR
```

- To set pins in port 1, use the following register:

```
LPC_GPIO1->FIOSET
```

3.8 Connecting External 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).

4 Tasks

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.

4.2 Blink an LED without CMSIS

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

   Tip
   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?

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_Libraries.zip
```

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

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`.

4.5 Blink an LED Using CMSIS

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

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  .

5 Resources

[lpc1769]
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

[lpcx-platform]
NXP Semiconductors. *Getting started with NXP LPCXpresso (User guide)*. Rev. 12. 17 April 2013.
http://www.nxp.com/documents/other/LPCXpresso_Getting_Started_User_Guide.pdf

[lpcx-ide]
NXP Semiconductors. *LPCXpresso v7 User Guide*. Rev. 7.3. 30 June 2014.
http://www.lpcware.com/system/files/LPCXpresso_User_Guide_0.pdf

[cmsis-intro]
ARM Ltd. *CMSIS Introduction - Cortex Microcontroller Software Interface Standard*. Version 4.1. 13 June 2014.
http://www.keil.com/pack/doc/CMSIS/General/html/index.html

[cmsis-core]
ARM Ltd. *CMSIS-CORE Overview - CMSIS-CORE support for Cortex-M processor-based devices*. Version 3.30. 13 June 2014
http://www.keil.com/pack/doc/CMSIS/Core/html/index.html

[lpcx-cmsis]
LPCXpresso Support. *CMSIS support in LPCXpresso*. 7 May 2014.
http://www.lpcware.com/content/faq/lpcxpresso/cmsis-support

6 Grading Sheet

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