Experiment 2
General-Purpose Input/Output (GPIO)

Ahmad Khayyat, Hazem Selmi, Saleh AlSaleh

Version 162, 17 February 2017

Table of Contents

1. Objectives

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

2. Parts List

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

3. Background

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.

3.1.1. 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.

3.1.2. 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`?

3.1.3. 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:

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

Exercise

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

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.

3.2.1. 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 1. 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
```

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.

3.3.1. 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`.

3.3.2. 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 2. 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
```

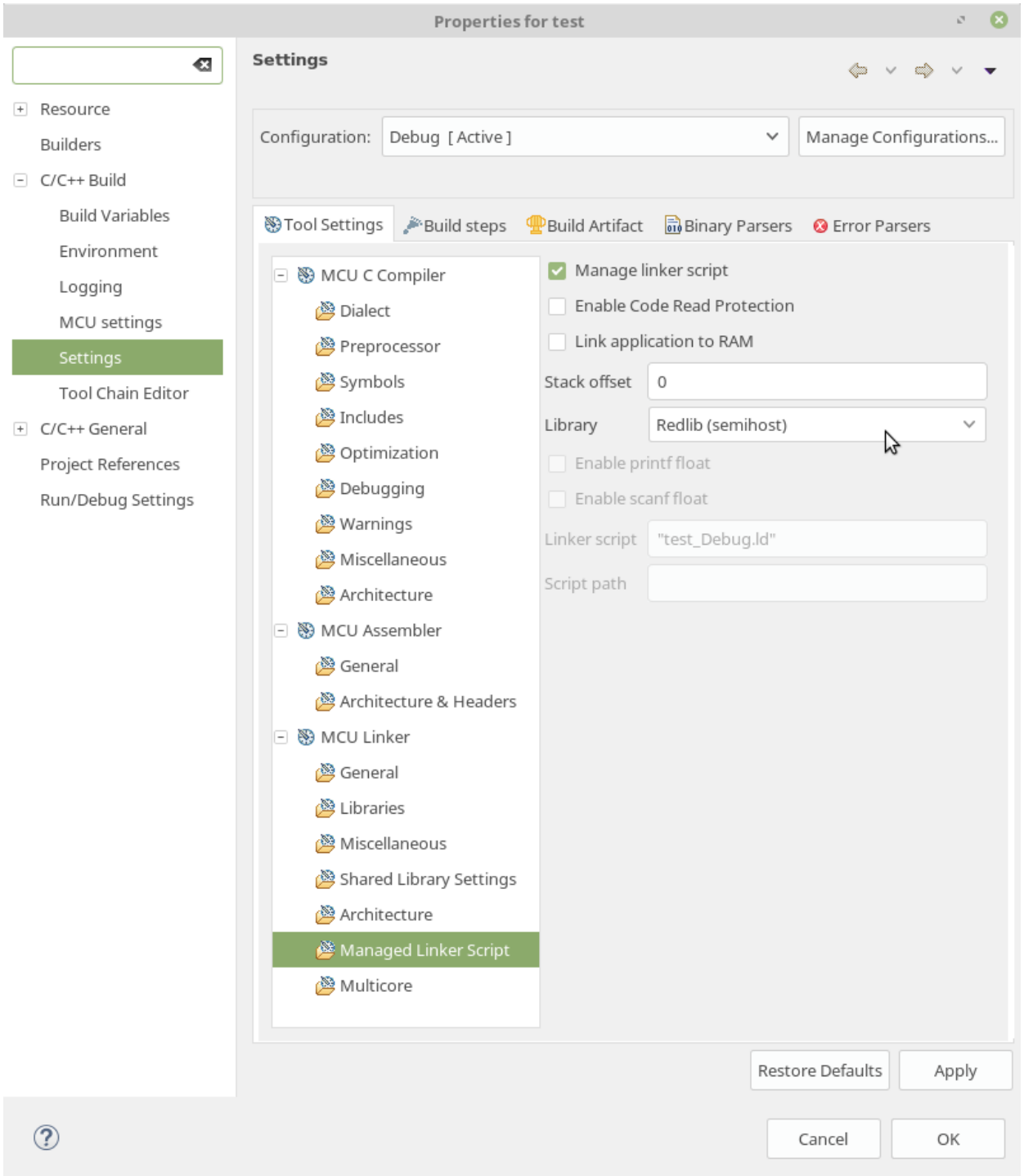

Figure 1. Semihost Debugging Configuration

4. Tasks

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

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.

4.2. Software

4.2.1. 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.

4.2.2. 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.

5. Resources

[lpcxpresso-ide-user-guide]

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

6. Grading Sheet

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