Experiment 3: Interrupts

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
142	10 February 2015		H

Contents

1 Objectives

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

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 Background

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

Note
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.

There are four main steps to correctly setup any interrupt:

1. Configure the required hardware 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 Section 3.2.1 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.

Note
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).

When the above steps are done correctly, Cortex-M3 CPU job will be:

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. In case of external interrupts, an interrupt request is generated by a pulse at a pin that has been enabled to accept external interrupt requests.
- Jumping to the interrupt service routine associated with that request.

3.2 Configuring Interrupts

This section describes the steps required by the programmer to configure the LPC1769 microcontroller to handle interrupt requests generated by a given device.

The programmer must perform three main steps:

1. Enable the required interrupt in the NVIC
2. Write an interrupt service routine (ISR)
3. Clear the interrupt request at the end of the ISR

Note
You may want to refer back to this section whenever you use interrupts with any peripheral in future experiments.

3.2.1 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.

Interrupt Numbers In CMSIS, interrupts are enumerated from 0 to 34.

Note
In addition to these 35 interrupts, there 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.

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`.

3.2.2 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 format:

```
void TIMER2_IRQHandler() { // ISR for the TIMER2 device
    // Your code goes here
    // Clear the interrupt request at the end of the ISR
}
```

3.2.3 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.

3.3 Other Interrupt-Related Operations

Interrupts will not function at all without the above three 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

3.3.1 Interrupt Status

Sometimes, you need to check the status of a specific interrupt. For example, If you have multiple interrupts sharing the same interrupt channel, either 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.

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`.

3.3.2 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.

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

3.4 The `PINSELx` Registers

Note
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.)

Tip
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
```

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

Note
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.

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

3.5 External Interrupts

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 is detected by one of the I/O port pins.

External interrupts can be generated by a push-button or a switch connected to a pin that can generate external interrupt requests. The difference between such implementation and what you did in Experiment 2 should be clear. 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 (discussed in the previous section). 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.

Tip
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.

3.5.1 Activating External Interrupts

Activating External Interrupts on GPIO Pins 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.

For example, to enable rising-edge interrupts on pin 0 of port 2 only:

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

Activating External Interrupts on Non-GPIO Pins 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.

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

Note
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.

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. For non-GPIO external interrupts, however, such configurations must be performed separately.

The `EXTPOLAR` and `EXTMODE` registers control the non-GPIO external interrupt behaviour.

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).

3.5.2 Clearing External Interrupt Requests

For External Interrupts on GPIO Pins 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.

For External Interrupts on Non-GPIO Pins 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;
```

3.5.3 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.

You do not need to worry about the status when you have only one external interrupt for EINT3. If you have multiple external interrupts, however, you can check the status registers in your ISR to determine the source of the request and, in turn, how to handle it.

All GPIO interrupt status registers are fields of the LPC_GPIOINT structure.

4 Tasks

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.

4.2 Two External Interrupts

1. 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.

Warning
All tasks must be completed during the lab session.

5 Resources

[keil-nvic]
ARM Ltd. *Nested Vectored Interrupt Controller*. 2013.
http://www.keil.com/support/man/docs/gsac/gsac_nvic.htm

[lpc1769-manual]
NXP Semiconductors. *UM10360 LPC176x/5x User manual*. Rev. 3.1. 2 April 2014.
http://www.nxp.com/documents/user_manual/UM10360.pdf

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