Lab 5: Interrupts

ECE 3720

Preview

The MC will count from 0 to 15 as in Lab 1, but with the addition of an interrupt that will turn on all the LEDs for a few seconds. Afterwards, the counting will resume where it left off. The interrupt will be triggered by a debounced button.

Interrupts

- When an interrupt is triggered,
  1. MC halts main work and saves state
  2. MC finds and executes interrupt service routine (ISR)
  3. MC restores main state and continues operation
- Possible interrupt sources include external pins, timers, and other peripherals.
- Interrupts remove the need to repeatedly check values while waiting for event.
  - Constantly checking like that is called “polling”, and it’s what we did in Lab 2.
- Once an interrupt is set up, it is handled automatically by the microcontroller
  - You will use registers to tell the MC what triggers the interrupt, and write an ISR to tell the MC what to do when it occurs.
  - You will *not* call the interrupt function at any point.
  - You will *not* check the value of a pin.

> PIC32MX1XX/2XX devices generate interrupt requests in response to interrupt events from peripheral modules. The interrupt control module exists externally to the CPU logic and prioritizes the interrupt events before presenting them to the CPU.

PIC32 datasheet, pg. 87

Interrupt Registers

The following are the registers of interest for this lab

- **IECx** (datasheet pg. 92)
  - Used to enable or disable particular interrupts
- **INTCON** (pg. 90)
  - We will use this to set edge polarity of external interrupts
  - If set to rising edge, the interrupt will trigger when the pin transitions to HIGH
- **IFSx** (pg. 92)
  - Interrupt flag status register
  - Every interrupt has a "flag" that goes HIGH when the interrupt is triggered
  - The flag must be cleared before the interrupt can trigger again
- **IPCx** (pg. 93)
  - Interrupt priority control register
  - 1 is highest priority, 7 is lowest
  - A priority of 0 effectively disables the interrupt
  - Sub-priority does not matter for our purposes

Interrupt Bit Locations

- **Table 7-1 (on pg. 88 of the datasheet) provides the necessary information for using each interrupt.**
  - This is an important resource. You will want to refer to it again in future labs.
- Note that the number in the register name may not match the number in the peripheral name.
- The *vector number* identifies the interrupt source, and will be used when writing your ISR.

TABLE 7-1: INTERRUPT IRQ, VECTOR AND BIT LOCATION

Interrupt Source(1)	IRQ #	Vector #	Interrupt Bit Location				Persistent Interrupt
			Flag	Enable	Priority	Sub-priority	
Highest Natural Order Priority							
CT – Core Timer Interrupt	0	0	IFS0<0>	IEC0<0>	IPC0<4:2>	IPC0<1:0>	No
CS0 – Core Software Interrupt 0	1	1	IFS0<1>	IEC0<1>	IPC0<12:10>	IPC0<9:8>	No
CS1 – Core Software Interrupt 1	2	2	IFS0<2>	IEC0<2>	IPC0<20:18>	IPC0<17:16>	No
INT0 – External Interrupt	3	3	IFS0<3>	IEC0<3>	IPC0<28:26>	IPC0<25:24>	No

PIC32 datasheet, pg. 88

ISR (Interrupt Service Routine)

- Tells the MC what to do when an interrupt occurs
- Use the following macro to identify the interrupt source:

  __ISR(vector, ipl)
  - Note that there are two underscores
  - The first argument is the vector number for the interrupt source
  - We can omit the second argument, which sets a priority level
- Write your ISR function, with _ISR() between the return type and function name.
  - The interrupt takes no arguments and returns no data, so both are *void.*
  - **Make sure to clear the correct interrupt's flag at the end of the function.**

```
void __ISR(/* vector */) exampleFunction(void)
{
    // code to execute
    // clear flag in IFSX register
}
```

The function name can be anything

Setting up the interrupt

- To globally enable interrupts, you must include the following in your code:

```
#include <sys/attribs.h>      // at the top of the program

INTCONbits.MVEC = 1;          // in main
__builtin_enable_interrupts(); // in main
```

- In this lab we will use External Interrupt 0 (INT0).
  - One of 5 external interrupts on the PIC32
  - INT0 is the only one hard-mapped to a particular pin (find it on the pinout diagram)
- Use the registers in slide 4 to set the priority and polarity of INT0, then enable it.
  - It's also good practice to clear the interrupt's flag when setting it up.
- Then write your ISR as shown on the previous slide.
  - Make sure you don't put it inside your main function.

```
1  #include <xc.h>
2  #include <sys/attribs.h>
3
4  main(){
5      INTCONbits.MVEC = 1;
6      __builtin_enable_interrupts();
```

INT0 is mapped to this pin on our version of the PIC32

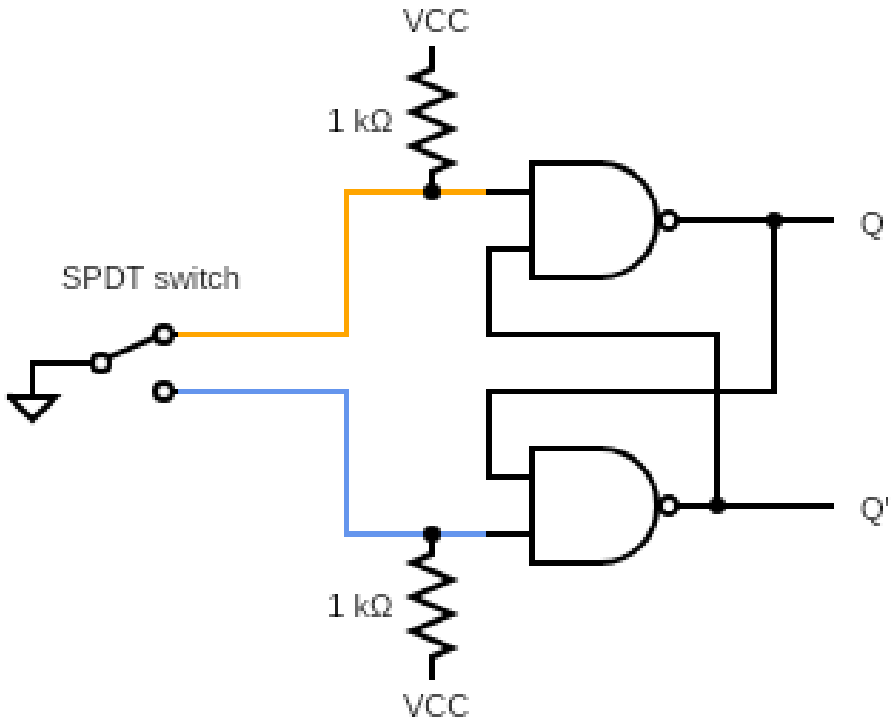
TABLE 1-1: PINOUT I/O DESCRIPTIONS (CONTINUED)

Pin Name	Pin Number(1)				Pin Type	Buffer Type
	28-pin QFN	28-pin SSOP/ SPDIP/ SOIC	36-pin VTLA	44-pin QFN/ TQFP/ VTLA		
INT0	13	16	17	43	I	ST
INT1	PPS	PPS	PPS	PPS	I	ST
INT2	PPS	PPS	PPS	PPS	I	ST
INT3	PPS	PPS	PPS	PPS	I	ST
INT4	PPS	PPS	PPS	PPS	I	ST

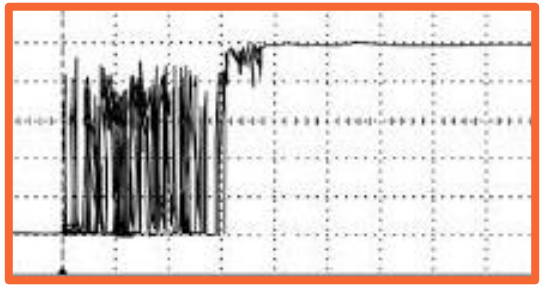
PIC32 datasheet, pg. 21

Debouncing

- Switches have a tendency to bounce rapidly between HIGH and LOW when first toggled.
  - This can be addressed with both hardware and software
- We will implement the debouncer seen at right using the SN74LS00 and a SPDT (single pole, double throw) switch.
  - This is an example of an SR (set-reset) latch
  - Q' = !Q
  - When the switch transitions between throws, the pull-up resistors and NAND gates ensure the output remains unchanged it settles.
- Connect one of the outputs to INT0
  - You can wire both outputs to LEDs first to check their behavior



Switch output bouncing between LOW and HIGH



Lab Goals

- Recreate Lab 1, but with an interrupt that causes all the lights to turn on and stay on for a few seconds.
- The interrupt should be triggered by the debounced output of the SPDT switch.
- When the interrupt ends, the MC should resume counting where it left off.

Simple Diagram