Lesson 8
Memory Banking and I/O

Overview

Introduction

Up until now, all of our programs have lived within the simulator. In this lesson, we examine how the PIC can sense its environment, and how it can influence the circuit where it lives.

In this section

Following is a list of topics in this section:

PIC I/O

Introduction	Every PIC has some compliment of I/O devices. In most models, the bulk of the pins are taken up with I/O. In fact, the main driving force for having PICs in packages with lots of pins, is to provide more places to connect.
Digital Input and Output	All PICs have some number of pins that can be used for simple digital I/O. When used as an input, these pins are at a logical high (1) when they sense a logical high voltage[1], and logical low (0), when they sense a logic low voltage. The pins are directly associated with bits in a particular file register location; when the pin goes high, the corresponding bit becomes a logic '1'. These same pins can be configured as outputs. When the program stores a 1 in the bit corresponding to an output pin, the pin asserts a high logic voltage level (approximately 5 volts). When the bit is set low, the pin asserts a low voltage (about zero volts).
Counter I/O	Most PICs have some pins which can be associated with a counter. These pins have an associated register which retains a count of the number of times a pin has toggled between high and low. Sometimes these counters have prescalers or postscalers associated with them which allow the range of the counter to be extended. Some of these counters are 8 bits, some 16 bits.
Other I/O	The various model PICs offer quite a number of different types of I/O. These may include: Comparator: The designer provides a reference voltage and the pin responds to whether its voltage is above or below the reference Analog: The pin has a register which permits the program to read the voltage on the pin PWM: The pin toggles at a duty cycle determined by a register which can be set by the programmer Synchronous Serial Port: Data is clocked into a shift register by an external clock USART: Synchronous or asynchronous data can be clocked into or out of a shift register.
PIC16F84A I/O	The PIC16F84A has 13 I/O pins. All of these pins may be configured as inputs or outputs. One of the pins may be optionally configured as a counter input. There is an 8 bit prescaler on the chip which can be associated with the counter.

[1] Throughout this course we will assume that logic high is 5 volts, and logic low is zero. In fact, that's not strictly true. The PIC can use a fairly wide supply voltage and this influences what is meant by high and low. Further, some pins can be configured to behave a little differently to give more flexibility in design. If you have a situation where the details really matter, look at the DC Characteristics tables in the datasheet, but for most purposes, assuming a 5 volt supply, "high" is anything above 4 volts, and "low" is anything below 1 volt.

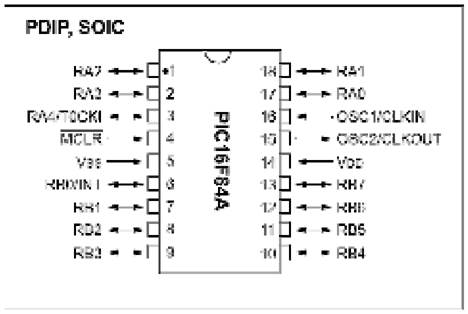
Using the Inputs

Introduction

On the PIC16F84A all of the I/O pins are configured as inputs on power up. In this section we will look at how to use those inputs.

I/O Pins

In your datasheet you will find the following picture:



Pins 17, 18, 1, 2, and 3 comprise PORTA. Pin 17 (RA0) is connected to bit 0 of PORTA. Pin 18 to bit 1 and so forth (PORTA has only 5 bits implemented in the PIC16F84A). Similarly, pins 6 through 13 are connected to bits 0 through 7 of PORTB. If any of these pins is raised to a logical high voltage, the corresponding pin of PORTA or PORTB will read as a '1'.

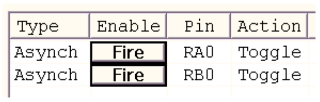
Pin 3 (RA4/T0CKI) and pin 6 (RB0/INT) have additional features that we won't use for now, however they behave just like the other I/O pins unless you specifically configure them to be different.

Setting up a project

Set up a new project, Lesson8a, with a very minimal program. Include the normal header information, and our typical ending `loop goto loop`, but nothing else. Assemble the program and select MPLAB SIM as the debugger.

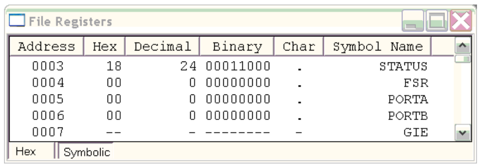
Select View->File Registers and Debugger->Stimulus. The file register window we've seen before but the Stimulus window is new. This allows us to change the state of the PIC pins as seen by the simulator.

In the stimulus window, pin stimulus tab, click on 'Add Row' and select RA0 from the dropdown under 'Pin'. (You may need to widen the Pin column a bit to see the values in the dropdown). Select 'Toggle' for the action. Do the same thing for RB0:



Type	Enable	Pin	Action
Asynch	Fire	RA0	Toggle
Asynch	Fire	RB0	Toggle

Arrange your windows so you can see both the stimulus window and PORTA and PORTB in the file register window:



```
Address  Hex  Decimal  Binary    Char  Symbol Name
 0003    18       24   00011000   .         STATUS
 0004    00        0   00000000   .            FSR
 0005    00        0   00000000   .          PORTA
 0006    00        0   00000000   .          PORTB
 0007    --        -   --------   -            GIE
```

Continued on next page

Using the Inputs, Continued

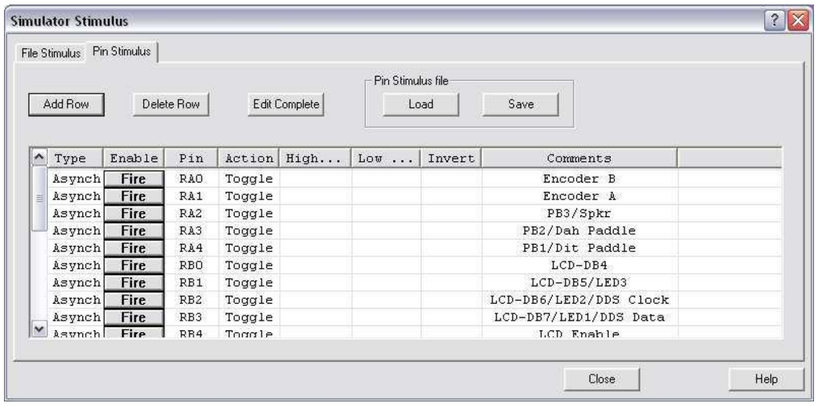
Running the program

If you click on ‘Step Into’ a few time, not much happens. This would be expected; after all, the program is simply looping from `loop` to `loop`.

Now click on the ‘Fire’ button next to RA0 and then click step. PORTA has changed from H’00’ to H’01’. Bit 0 of PORTA tracks the state of PIC pin RA0 (pin 17). If you click ‘Fire’ again, pin 17 will go low and PORTA will return to 0. Similarly for PORTB. The ‘Toggle’ mode of the pin stimulus changes the state of the corresponding pin each time we click the fire button.

This is probably the most useful mode of the stimulus window. The other stimulus modes allow us to base the pin states on the processor clock. Rarely are we dealing with signals that are predictable, and testing with synchronous stimuli can often lead us into problems because real world signals are rarely so well behaved.

Sometimes we want to set up a number of different stimuli to test. If we are testing something fairly complex, we may want to test it over and over. We can add comments and save our stimulus window settings in a file by clicking the Save button.



Type	Enable	Pin	Action	High...	Low ...	Invert	Comments
Asynch	Fire	RA0	Toggle				Encoder B
Asynch	Fire	RA1	Toggle				Encoder A
Asynch	Fire	RA2	Toggle				PB3/Spkr
Asynch	Fire	RA3	Toggle				PB2/Dah Paddle
Asynch	Fire	RA4	Toggle				PB1/Dit Paddle
Asynch	Fire	RB0	Toggle				LCD-DB4
Asynch	Fire	RB1	Toggle				LCD-DB5/LED3
Asynch	Fire	RB2	Toggle				LCD-DB6/LED2/DDS Clock
Asynch	Fire	RB3	Toggle				LCD-DB7/LED1/DDS Data
Asynch	Fire	RB4	Toggle				LCD Enable

Reading the inputs

Within a program, we normally would assign a symbol to a bit number rather than using the bit number directly. Suppose, for example, we wanted to read pushbutton 1. Since this is connected to RA4, we might do something like:

```
PB1          equ          D’4’
```

We would then test the state of that bit with something like:

```
             btfsc      PORTA,PB1
```

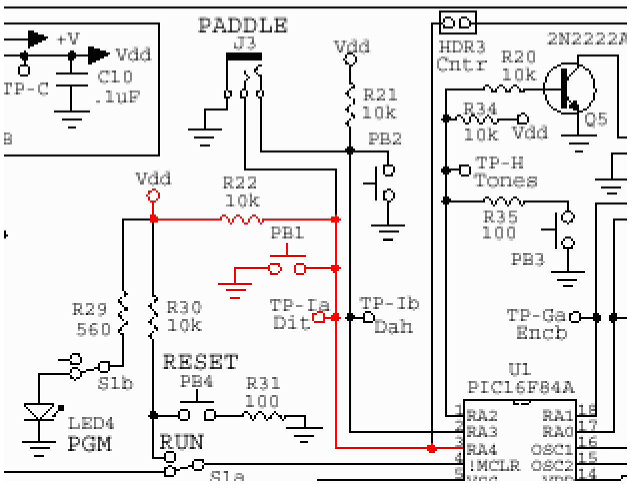
One of the problems we encounter, however, is that the PIC, running at 4 MHz, is pretty fast. When the PIC looks at mechanical things like buttons and switches it may see hundreds of closures each time the switch is activated. This is called “contact bounce”. Thus, we need to read external things a bunch of times to be sure the contacts have stopped bouncing.

Continued on next page

Using the Inputs, Continued

Reading the inputs (continued)

Let's expand our program to count the number of times we have pressed button 1. Before we do this, though, look at the PIC-EL schematic and notice how PB1 is connected:



R22 keeps RA4 at V_{dd} (+5 volts) until PB1 is pressed, in which case it is pulled to ground. Therefore, bit 4 of PORTA will normally be high, and it will go low (0 volts) when PB1 is pressed.

Our code might look something like:

```
TestPB1d
                btfsc       PORTA,PB1       ; PB1 down?
                goto        TestPB1d        ; No, wait for press
TestPB1u
                btfss       PORTA,PB1       ; PB1 up?
                goto        TestPB1u        ; No, wait for release

                incf        Count,F         ; Add button press
                goto        TestPB1d        ; and go test again
```

(Obviously, we had to allocate space for the Count variable, as well as defining PB1 to be equal to 4.)

If we were to run this in animate mode, we could click on our pin 3 stimulus and see the program first loop through the first loop, then the second, each time we toggled pin 3.

Of course, we could simply read an entire port with something like:

```
     movf       PORTA,W
```

and as we will see later in the course, on more complex applications this is often what we want to do. But in many cases we simply want to test a single bit of the port, and the bit test instructions work quite well for that purpose.

Memory Banking

Introduction

Up until now, we have pretended that the file register is pretty well behaved. Perhaps you have noticed some quirky behavior. In this section, we are going to dig a little deeper into what is going on with the file register.

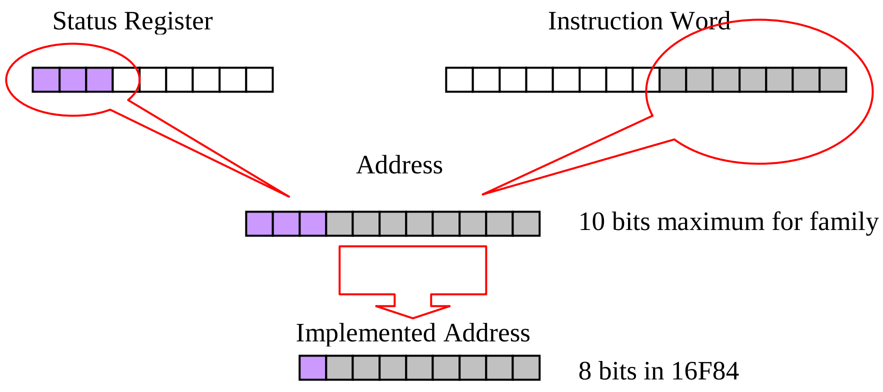
Generating File Register Addresses

You may have noticed that if you expand out the file register view to see addresses beyond H'7f', it appears as if the first 128 locations are duplicated.

You may also have noticed, if you were very inquisitive, that instructions that address the file register may only take addresses from 0 to 127, yet five of the registers described in p16f84a.inc have values above 127. In the datasheet, there is a picture (Fig. 2-2) showing 128 file register locations, and another 128 to the right of those with addresses from H'80' through H'ff'. But if the instructions can only access addresses from H'00' through H'7f' what good are they? Clearly, there is some sort of disconnect here.

What is happening is this. The PIC16 family of processors has a 10-bit address bus for the file register. The low seven bits come from the instruction, and the high three bits come from the status register. In the case of the PIC16F84A, only eight bits are actually implemented, so the left most two bits of the status register are always zero. This gives the program a view of memory that is broken into two memory *banks*. In some other PIC processors there may be more banks, depending on the amount of file register memory supported.

Further, in the PIC16F84A (and in most other PIC processors), most of the special function registers are duplicated in all banks. Also in the PIC16F84A, but in very few of the other PIC processors, all of the RAM locations are duplicated in all the banks.



Thus, by adjusting bit 5 of the status register (known as RP0) we can address 256 memory locations, even though the instruction word only allows for 128.

Continued on next page

Memory Banking, Continued

Programming for Banks

Notice that there are only five special function registers in bank 1 that do not exist in bank 0:

- OPTION_REG
- TRISA
- TRISB
- EECON1
- EECON2

There are similarly five registers that exist in bank 0 but not in bank 1:

- TMR0
- PORTA
- PORTB
- EEDATA
- EEADR

In order to switch between being able to access the second group and being able to access the first group, we are going to have to adjust RP0.

Programmatically, this is fairly simple. Typically, we want to use bank 0, so if we need to adjust something in bank 1, we set RP0, do our business, and then clear RP0:

```
        bsf      STATUS,RP0
        movlw    B’00001111’
        movwf    TRISA
        bcf      STATUS,RP0
```

This is typical of code we might have in the initialization section of our program. Later in this lesson, we will talk specifically about the TRISA and TRISB registers. The four ‘EE’ registers have to do with the electrically erasable PROM, which we won’t use until later in the course. Similarly, the OPTION_REG is used infrequently.

The above code works fine, but it is very specific to the PIC16F84. If this program were to be used on a different PIC, this section would have to be rewritten. However, there is an assembler directive we can use instead, which does this in a way that is a little more independent of the particular PIC. The `banksel` directive takes the name of a register as an argument and causes the assembler to generate the necessary bank selection code. On the 16F84A, the code generated will be identical to what we showed above, but if we move the code to a different PIC, it might be different.

```
        banksel  TRISA        ; Select bank containing TRISA
        movlw    B’00001111’  ; Will turn make bit 4 output,
        movwf    TRISA        ; bits 0-3 inputs
        banksel  PORTA        ; Back to bank containing PORTA
```

We simply name the register we wish to access in the `banksel` directive, and the assembler remembers what has to happen to the status register to get to the appropriate bank.

Continued on next page

Memory Banking, Continued

A test program

Let’s make a Lesson8b project, and begin by putting in our normal starting and ending statements. Let’s make room for two variables, `Bank0` and `Bank1`, and let’s see what happens when we read `PORTB` from each bank:

```
movf        PORTB,W     ; Read PORTB
movwf       Bank0       ; Save the value
banksel     TRISB       ; Switch to Bank 1
movf        PORTB,W     ; Read PORTB again
movwf       Bank1       ; Save the value again
banksel     PORTB       ; Back to Bank 0
```

(From here on, we will leave the example code without coloring for those folks who are using monochrome printers. The colors print very light on some printers)

When we run this program, we discover that we get a 0 stored in `Bank0`, and a H’ff’ stored in `Bank1`. What is happening, of course, is that the second `movf PORTB,W` isn’t reading `PORTB` at all, but rather `TRISB`, which has the same lower seven bits in its address, but has bit 7 set to true. This could be very confusing for someone reading the code later, so we should always use `TRISB` when we mean `TRISB`.

However, if we do this, we get an annoying warning about referencing a register that is not in bank 0. We can use the `errorlevel` directive to suppress this warning. Now our code would look like:

```
movf        PORTB,W     ; Read PORTB
movwf       Bank0       ; Save the value
banksel     TRISB       ; Switch to Bank 1
errorlevel  -302        ; Turn off warning
movf        TRISB,W     ; Read PORTB again
movwf       Bank1       ; Save the value again
banksel     PORTB       ; Back to Bank 0
errorlevel  +302        ; Turn warning back on
```

We can use `errorlevel` to suppress any warning that we wish. Generally, it is helpful documentation that we recognized the warning and really intended to do what we did.

Using the Outputs

Introduction

The 16F84A I/O pins are all set to be inputs at power up. However, each can be individually configured to be an output. In this section, we will see how to use those outputs.

The TRIS registers

In the previous example, we noticed that the TRISB register was all ones at power up. TRIS stands for **tri-s**tate. A '1' in any bit of either of the TRIS registers means that the corresponding pin is set to be tri-stated; that is, an input. The TRISA and TRISB registers match, bit for bit, with the PORTA and PORTB registers.

If bit 1 of TRISB is set to be a 1, then bit 1 of PORTB is an input. Similarly, if bit 1 of TRISB is set to be a 0, then bit 1 of PORTB is an output. When we store a value into that bit, the corresponding PIC pin will assert the voltage corresponding to that logic state.

Another small example

This time, make a Lesson8c, and into Lesson8c.asm put the same starting and ending code, but for the main part of the code, try this:

```
        bcf           PORTB,1       ; Try to change PORTB
        bsf           PORTB,1
        banksel       TRISB         ; Switch to Bank 1
        errorlevel    -302          ; Turn off warning
        bcf           TRISB,1       ; Make bit 1 output
        banksel       PORTB         ; Back to Bank 0
        errorlevel    +302          ; Turn warning back on
        bcf           PORTB,1       ; Try to change PORTB again
        bsf           PORTB,1
```

We will try to change PORTB, bit 1. Then we will set that pin to be an output, and we will try it again.

When you single step through this, watch PORTB in the file register display. As soon as we make it an output, the bit gets set. That's because the port remembered what we tried to set it to, even though it wasn't an output yet.

In this example we initialized only one bit. Typically, we would initialize all the bits of the port at one time by storing a value into the TRIS register rather than setting an individual bit.

A final example

Shortly, we will be starting to deal with real hardware. Let's write a little program to read button 1 of the PIC-EL and light LED1 if PB1 is pressed. Let's call this Lesson8d.

First thing, it would be helpful to define constants for the LED and pushbutton:

```
PB1         equ   D'4'        ; Pushbutton 1 on PORTA
LED1        equ   D'3'        ; LED 1 on PORTB
```

This way we don't need to keep remembering what pin is attached to what device (although we still need to remember which port).

Continued on next page

Using the Outputs, Continued

A final example (continued)

Now, we need to set up the ports. I like to use binary constants for this so I can visualize all the pins, but there is no rule that says we couldn't use decimal or hex numbers here. While we're at it, let's set up all the PIC-EL pins:

```
movlw       B'00000000'  ; All bits as output
banksel     TRISB        ; Select Bank 1
errorlevel  -302         ; Supress warning
movwf       TRISB        ; Set tristate mode
banksel     PORTB        ; Select Bank 0
errorlevel  +302         ; Re-enable warning

movlw       B'00011011'  ; Spkr out, others in
banksel     TRISA        ; Select Bank 1
errorlevel  -302         ; Suppress error
movwf       TRISA        ; Set mode for the pins
banksel     PORTA        ; Select Bank 0
errorlevel  +302         ; Restore message
```

It would have been more efficient to do the bank selection once, and then do the stores, and then select bank 0. However, this step-by-step approach is a little more obvious when you come back in a few months and wonder what you did.

Now it turns out that our loop is pretty simple. The buttons are wired so that zero volts means that they are pressed. The LEDs are wired so that a false output (zero volts or the PIC sinking current) causes them to light. So what we want to do is to make the LED bit 0 if the button bit is 0.

```
Loop
      btfsc  PORTA,PB1    ; Button pressed?
      goto   Up           ; No, go to button up
      bcf    PORTB,LED1   ; Yes, turn on LED
      goto   Loop         ; Do it again
Up
      bsf    PORTB,LED1   ; Turn off LED
      goto   Loop         ; Play it again, Sam
```

As you single-step through this, you should be able to set up a stimulus to represent the pushbutton and watch the LED result in `PORTB`.

Homework Assignment

Extend the program from Lesson8d to have all three LEDs track the positions of the three pushbuttons. You will need to refer to the PIC-EL schematic to determine what bits are responsible for these devices.

Simulate this program thoroughly; in Lesson 11 we will be testing it on the real hardware!

Wrap Up

Summary	We have looked at how the PIC can get input from and send output to the outside world. In the course of doing that, we needed to learn how the memory banking of the PIC16F84A works.
Coming Up	Next lesson we will look at some of the assembler directives that we haven't seen yet. There are several that are somewhat optional, but can make our life a little easier.