Lesson 12
Input, Output and Macros

Overview

Introduction

In this lesson, we will write a small program to exercise our hardware, and we will look at how we can use macros to simplify our code.

In this section

Following is a list of topics in this section:

Organizing Program Flow

Introduction

When programming for embedded applications, we need to take into account that the program will be running continuously, and will be expected to respond to a variety of external events. To meet these demands, programs must be organized in a specific way.

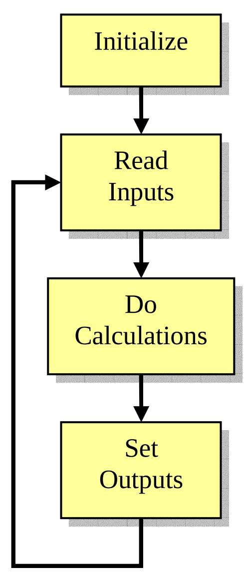
Race Conditions

One of the challenges in dealing with real time events is that they don't always happen when we expect them. If we aren't careful, inputs could change as we are executing our code, and the result can be very confusing.

Dealing with multiple events can be pretty confusing even when everything is confined to inside the computer. When we are connected to the outside world, the opportunities for conflicting events multiply greatly. This is even more true when we are controlling electronic circuits. When we are dealing with physical things, the outside world generally can't respond very quickly. Even though it may be slow compared to a modern PC, compared to, say, a water heater, the PIC is blindingly fast. There is nothing we can do to a physical device that doesn't take forever in PIC terms.

However, if we are controlling an electronic circuit, it's a different story. Transistors can respond in nanoseconds. The external circuitry, from the standpoint of our application, becomes an additional place we can encounter unexpected interactions.

Fortunately, most of these problems can be avoided by organizing the program in a particular way. By simply refraining from interacting with the outside while we are doing our logic, we can avoid most of the problems that are caused by unexpected changes:



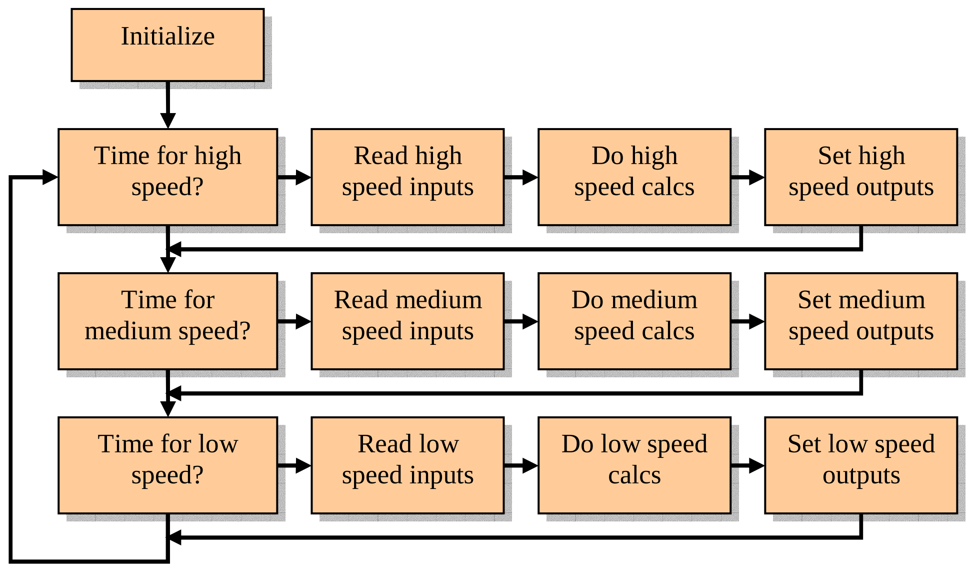
The key thing is not to allow the inputs to change while we are doing our calculations by simply ignoring them. We use the values read at the beginning of the cycle, then set the outputs all at once at the end. This leads to much more predictable behavior in our program.

Continued on next page

Organizing Program Flow, Continued

Multiple Frequencies

In the examples for this lesson, we will only be doing one time scale. But often the application needs to consider multiple time scales. For example, in a single PIC we may want to implement a keyer, which needs to respond in milliseconds, as well as control a VCO, which needs to be nudged very slowly to avoid phase noise, all the time maintaining a display which is updated on a different schedule. In that case, the model is extended to look something like the following:



Notice that we take care to only read and write those inputs and outputs that we absolutely need to change at the faster frequencies. Whenever possible, avoid making changes except at the lowest frequency. If changes are needed at a higher frequency, which is often the case, one must be careful to understand possible interactions.

Later in the course applications requiring multiple time domains will be examined.

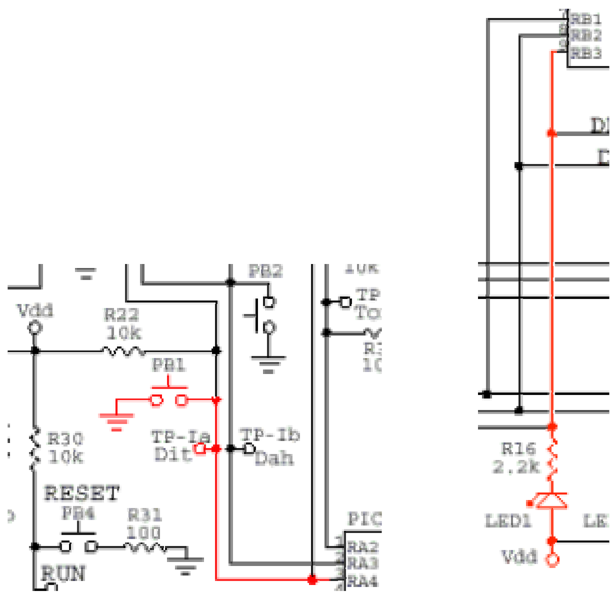
A Simple I/O Application

Introduction

To demonstrate how the recommended program layout is implemented, we will take the simplest possible example; read a pushbutton and cause an LED to track its position.

Understanding the ports

To understand what any program must do, it is necessary to examine the circuitry to which it is connected. In the current application, PB1 will be sensed and LED1 will be illuminated:



Notice that PB1 is connected to RA4 and LED1 is connected to RB3. Also significant is that PB1 is pulled up through R22, and thus will be high when open, and pulled to ground when pressed. Similarly, LED1 is connected the Vdd, and thus no current will flow if RB3 is high. Bringing RB3 low will cause current to flow through LED1.

Capturing this in the program

After setting up the same initial things that are always required to keep the assembler happy, we next want to capture what we learned from the schematic in our program:

```
;=================================================================
;      Manifest Constants
;=================================================================
LED1   equu           H'03'         ; PORTA bit number for LED
PB1    equ            H'04'         ; PORTB bit number for button
```

Now we can forget specifically what pins we are using and instead refer to the bit numbers in the ports by names that are easier to remember.

Continued on next page

A Simple I/O Application, Continued

Capturing this in the program (continued)

It will also be necessary to initialize the ports. In this particular application, all the inputs (all one of them) are on PORTA, and all the outputs on PORTB. To keep things readable and well documented, it is helpful to define constants for the settings for the TRIS bits as well:

```
;==============================================================
;      Manifest Constants
;==============================================================
LED1   equ    H'03'              ; PORTA bit number for LED
PB1    equ    H'04'              ; PORTB bit number for button
MASKA  equ    B'11111111'        ; PORTA all inputs
MASKB  equ    B'00000000'        ; PORTB all outputs
```

Defining File Register Storage

In this simple application, there will not be a huge number of calculations, so the requirements for file register storage are fairly minimal. However, since we want to read the inputs independent of our logic, we need storage for the inputs. Similarly, we need storage for the outputs that our calculations will determine, so that the outputs may be set in a later step:

```
;==============================================================
;      File register use
;==============================================================
              cblock       H'0c'
                     Buttons            ; Storage for inputs
                     LEDs               ; Storage for outputs
              endc

              goto         start
```

This approach of defining the constants then the storage is (hopefully) becoming terribly rote. The next step should also become habit.

Initialization

Now the I/O ports and file register storage must be initialized. There is no need to initialize the input storage since it will be fully determined each cycle, but the output storage will be manipulated bit by bit, so it is helpful to initialize that location:

```
;==============================================================
;  Mailine begins here -- Initialization
;==============================================================
start
       errorlevel   -302
       banksel      TRISA        ; Set PORTA to be all inputs
       movlw        MASKA        ; (somewhat redundant since
       movwf        TRISA        ;(reset does this anyway)
       banksel      TRISB
       movlw        MASKB        ; Set PORTB to be all outputs
       movwf        TRISB
       banksel      PORTB
       errorlevel   +302
       movlw        B'00001110'  ; Turn off all LEDs
       movwf        PORTB
       movlw        B'00001110'  ; Initialize LEDs to all off
       movwf        LEDs         ;
```

Notice that the banksel TRISB is really redundant, as is the initialization of PORTA. We have done it here to make our intent clear to the reader of the program.

Continued on next page

A Simple I/O Application, Continued

Reading the inputs

Referring back to the drawing on page two, the next thing to do is to read the inputs:

```
;=============================================================
; Main program loop here
;=============================================================

main

;-------------------------------------------------------------
;       Get inputs
;-------------------------------------------------------------
        movf          PORTA,W       ; Get the inputs from PORTA
        movwf         Buttons       ; Save them away
```

That was pretty simple.

Performing the calculations

Now take the results from reading the inputs, and set the output variable storage to reflect how we would like the outputs:

```
;-------------------------------------------------------------
;       Do Calculations
;-------------------------------------------------------------
        btfss         Buttons,PB1   ; Is PB1 pressed?
        goto          LEDon         ; Yes
        bsf           LEDs,LED1     ; No, turn off LED1
        goto          LEDoff        ; Skip over turn on LED
LEDon                               ; Output low = LED on
        bcf           LEDs,LED1     ; Yes, turn on LED1
LEDoff
```

Unfortunately, a bit of jumping around is required leading to a couple of extra labels that really don't add a lot to the readability of the code. Especially since the outputs are not really affected here, we could have simplified this code somewhat, but for now, we will leave it very explicit.

Notice that had we decided to shorten the code we could have always set the bit, then cleared it if necessary, eliminating one jump. Had we been working on the port directly, this could have caused a glitch in the output, but we are saved from that by preparing the outputs first.

Setting the outputs

The final step in our loop is to send the outputs to the external circuit:

```
;-------------------------------------------------------------
;       Set outputs
;-------------------------------------------------------------
        movf          LEDs,W        ; Pick up the output storage
        movwf         PORTB         ; And send it to the world

        goto          main          ; Play it again, Sam
```

Again, very simple. At this point, it would be good to assemble the program, program it into the PIC-EL, and test it.

Assembly Time Calculations

Introduction

Although the PIC itself is fairly limited, the assembler is quite competent. Frequently, it is helpful to do arithmetic within the assembler, especially arithmetic on addresses.

Related Constants

The assembler can perform common arithmetic operations on a constant almost anywhere a constant is required. Quite often an application will require several constants that are related. Rather than providing explicit values, simple arithmetic can be used making maintenance of the application simpler. As an example, suppose we are generating fixed speed Morse. We might have constants like:

```
DitTime    equ    D'18'
DahTime    equ    D'72'
```

If we made a change to our logic that required changing DitTime, we would also have to remember to change DahTime. We would probably not forget if these were the only two constants in the application, but we have already seen that the list of constants can be quite lengthy. The application would benefit from something like:

```
DitTime    equ    D'18'
DahTime    equ    3*DitTime
```

Remember, though, that this arithmetic is done at assembly time, not at execution time. This means that the result must be known when the program is assembled.

The current program counter operator

The assembler expression syntax includes all of the operators that are normally available for arithmetic and logical operations. In addition, there is a special symbol, the dollar sign, that stands for the current program counter.

It is important to recognize that, at assembly time, this is the address where the current instruction will be generated by the assembler. This can be a little confusing because, at execution time, the program counter will always be one higher than the location the original instruction occupied, because the program counter is incremented before the instruction is executed. But when performing address arithmetic, it is important to remember that everything must be known at assembly time.

Eliminating excessive labels

Often it will be necessary to generate labels for short jumps, as in our earlier example. These labels can clutter the program making it harder to read. It is helpful to reserve labels for somewhat more major events, but lots of unimportant labels can frustrate this. We can avoid those labels by calculating offsets from the current program counter, and using those as the target of our jump:

```
        btfss       Buttons,PB1   ; Is PB1 pressed?
        goto        $+3           ; Yes
        bsf         LEDs,LED1     ; No, turn off LED1
        goto        $+2
        bcf         LEDs,LED1     ; Yes, turn on LED1
```

We can think of the `goto $+3` as a “skip the next two instructions” instruction. `goto $+1`, of course, is essentially a two cycle `nop`.

Continued on next page

Assembly Time Calculations, Continued

Eliminating excessive labels (continued)

We can show that this is identical to our earlier examples by assembling the program both ways and examining the listing file. The listing file shows the program memory location in the left column and the code that is generated to store in that location in the second column.

First the original:

```
0012   1E0C        00065           btfss    Buttons,PB1    ; Is PB1 pressed?
0013   2816        00066           goto     LEDon          ; Yes
0014   158D        00067           bsf      LEDs,LED1      ; No, turn off LED1
0015   2817        00068           goto     LEDoff         ; Skip over turn on LED
0016               00069 LEDon                             ; Output low = LED on
0016   118D        00070           bcf      LEDs,LED1      ; Yes, turn on LED1
0017               00071 LEDoff
```

And then the new:

```
0012   1E0C        00065           btfss    Buttons,PB1    ; Is PB1 pressed?
0013   2816        00066           goto     $+3            ; Yes
0014   158D        00067           bsf      LEDs,LED1      ; No, turn off LED1
0015   2817        00068           goto     $+2
0016   118D        00069           bcf      LEDs,LED1      ; Yes, turn on LED1
```

Notice that the code is identical in both cases. The `goto LEDon` generates a `goto` location H’16’. Looking at the top listing, the symbol `LEDon` is at location H’16’, so this is what we would expect. In the lower listing, the `$+3` is also H’16’ because the instruction is at location H’13’.

Simple Macros

Introduction

We have already seen how assembler directives like `equ` can be used to substitute a symbol for a value. This is a very powerful way to help make our program more readable. In the above examples, we used LED1 to represent the bit number for the LED so that our code could use LED1 instead of 3. Next week if we come back to read the program, we will find it a lot easier to remember what LED1 meant than a 3.

The assembler provides a much more capable substitution mechanism called a macro. A macro is text that we want the assembler to substitute in our code. A macro, however, can cover multiple lines and can have substitutions within it.

Macro format

To define a macro, we use the following format:

Name `macro` *optional arguments*
    Stuff
    `endm`

We can have a list of arguments separated by commas. When we want to use the macro, we enter

Name matching list of arguments

And the assembler will replace that line with however many lines of "stuff" we defined in our macro.

Simple Example

Let's look at a very simple example. Suppose we find ourselves frequently clearing bits 1,2 and 5 of a cell. We could write a macro like:

```
Bitclr    macro    Location
          bcf      Location,1
          bcf      Location,2
          bcf      Location,5
          endm
```

Then, there might be code like:

```
cblock    H'20'
    Loc1
    Loc2
endc

Bitclr    Loc1
Bitclr    Loc2
```

The assembler would actually generate:

```
bcf      Loc1,1
bcf      Loc1,2
bcf      Loc1,5
bcf      Loc2,1
bcf      Loc2,2
bcf      Loc2,5
```

You can see how this can help not only reduce the work in doing repetitive things, but it can make the program somewhat more readable.

A Macro Example

Introduction

What if we wanted to extend our LED blinking program to do all three LEDs instead of just LED1. We could write a macro like:

```
;=============================================================
;      Macro definition
;=============================================================
ChkBut macro        Button,LED
       btfss        Buttons,Button       ; Is PB pressed?
       goto         $+3                  ; Yes
       bsf          LEDs,LED             ; No, turn off LED
       goto         $+2
       bcf          LEDs,LED             ; Yes, turn on LED
       endm
```

And then call it with:

```
       ChkBut       PB1,LED1
       ChkBut       PB2,LED2
       ChkBut       PB3,LED3
```

This allows us to do the same task over. Notice that in the case where we need to change the locations we manipulate, macros can have advantages over subroutines. There are ways of passing in variable locations and the like to subroutines, but if there are very many, it can get to be more complex than the problem we are trying to solve.

Memory Expansion

Notice, however, that the macro gets fully expanded before the code is generated, so while the source may be smaller, the actual code loaded into the PIC isn't:

```
                      -----
                      00081       ChkBut    PB1,LED1
0012   1E0C               M       btfss     Buttons,PB1    ; Is PB pressed?
0013   2816               M       goto      $+3            ; Yes
0014   158D               M       bsf       LEDs,LED1      ; No, turn off LED
0015   2817               M       goto      $+2
0016   118D               M       bcf       LEDs,LED1      ; Yes, turn on LED
                      00082       ChkBut    PB2,LED2
0017   1D8C               M       btfss     Buttons,PB2    ; Is PB pressed?
0018   281B               M       goto      $+3            ; Yes
0019   150D               M       bsf       LEDs,LED2      ; No, turn off LED
001A   281C               M       goto      $+2
001B   110D               M       bcf       LEDs,LED2      ; Yes, turn on LED
                      00083       ChkBut    PB3,LED3
001C   1D0C               M       btfss     Buttons,PB3    ; Is PB pressed?
001D   2820               M       goto      $+3            ; Yes
001E   148D               M       bsf       LEDs,LED3      ; No, turn off LED
001F   2821               M       goto      $+2
0020   108D               M       bcf       LEDs,LED3      ; Yes, turn on LED
```

Notice that the listing shows the letter 'M' to indicate lines that were added as a result of the macro expansion.

Wrap Up

Summary	We have looked at how to organize programs that will deal with the outside world, and we have reviewed how to do input and output. We have also examined the technique of performing arithmetic during the assembly, and used that to make writing macros a little simpler.
Coming Up	In the next lesson, we are going to look at timing loops and how we use them.