Reading Switches

In this chapter, were going to be looking at ways of reading switches.

We will start with just lighting a LED when a switch is pressed, and move on to create a simple pedestrian crossing system.

Switch types

Switches basically come in two types; momentary or push switches; used for a doorbell, QWERTY keyboard switch, numeric/HEX/telephone keypad switch, and latching or toggle type used for light on/off, operating mode selection, or DIL DIP switch.

PIC Inputs

As we discussed previously, all the PICs I/O pins can be used as digital inputs and PIC's can only do one thing at a time. If you're lucky, you can devote all the PICs time to just waiting for a switch to be pressed, however it's much more likely that the PIC has lots of other things to do (driving LEDs, handling communications etc.), and so can only check every so often for the switch activation. If the switch is of the momentary type, it's possible the input pulse is so fast that the PIC may miss it.

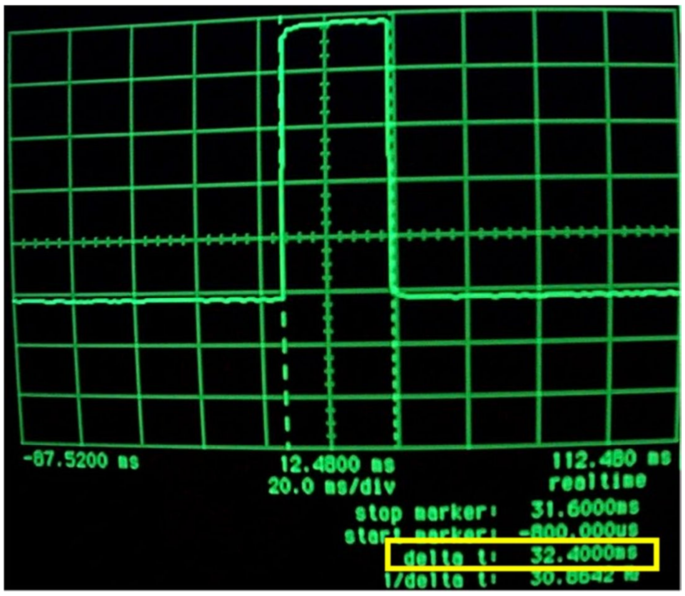

Fig 1. Trace showing a fast push button press.

Fig 1 shows a trace of a push button switch being pressed. The fastest I could do it was around 32ms so any PIC application that uses this particular switch would need to be able to cater for this switch. 32ms is pretty quick, but the PIC is faster.

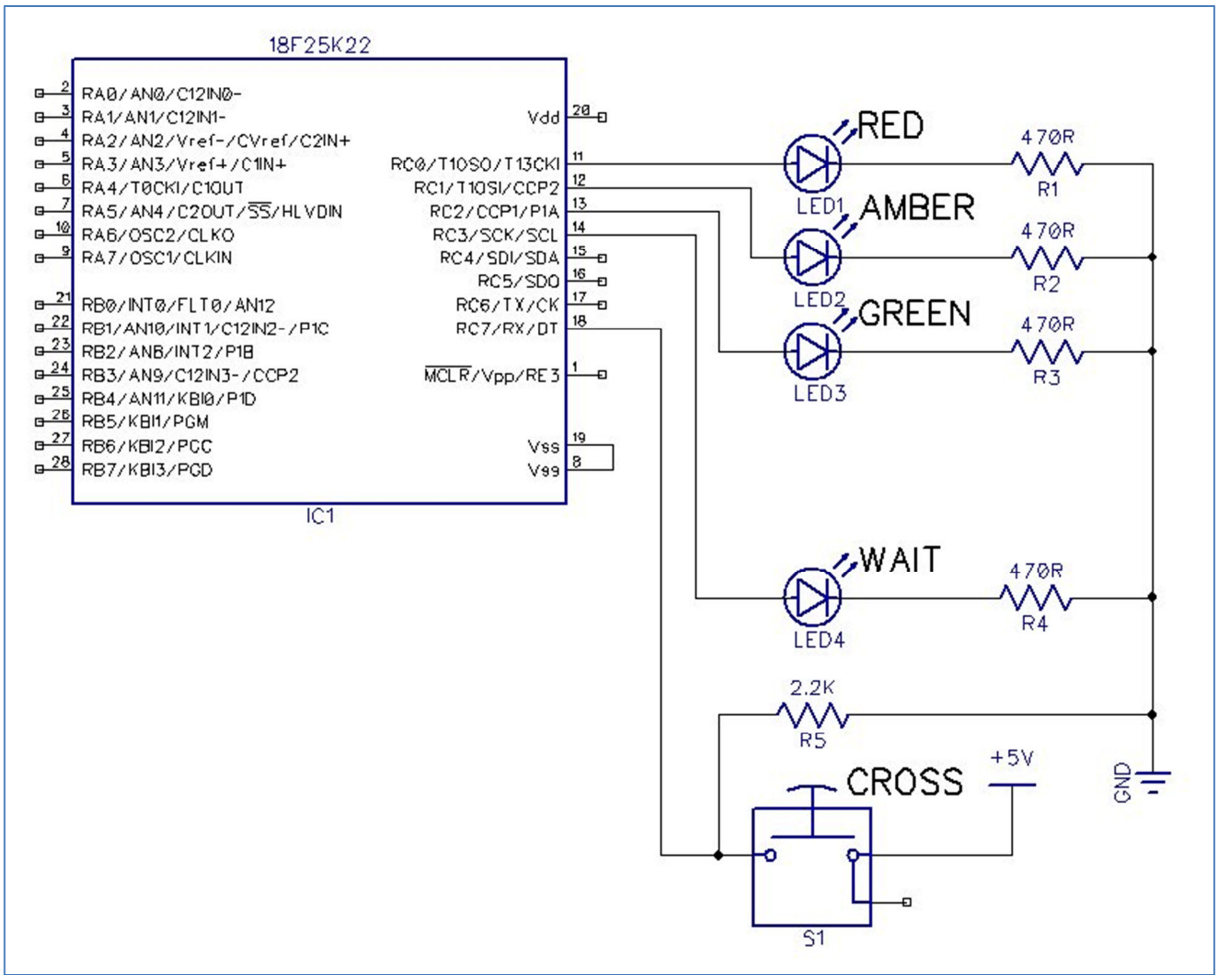

Fig 2. Circuit 4a - A simple circuit for a pedestrian crossing simulator.

We're going to use Circuit4a as a basis for the experiments in this chapter.

The first program will be just to capture the push-button S1 being pressed and to illuminate LED4.

```
'-- Configure I/O ports and pins here
    Output PORTC          ' Make ALL of PORTC an output

    Symbol LED_RED       PORTC.0
    Symbol LED_AMBER     PORTC.1
    Symbol LED_GREEN     PORTC.2
    Symbol LED_WAIT      PORTC.3
    Symbol SW_CROSS      PORTC.7
    Input SW_CROSS


'-- Start of main program
    ' Make sure that all the LEDs are off as we don't know how the PIC will power up
    Low LED_RED
    Low LED_AMBER
    Low LED_GREEN
    Low LED_WAIT
```

```
DoAgain:
    If SW_CROSS = 0 Then GoTo DoAgain

    ' The pushbutton has been pressed
    High LED_WAIT
    DelayMS 1000
    Low LED_WAIT


    GoTo DoAgain
```

As before, lines highlighted in yellow are BASIC instructions that we've not come across before. The first one is:

```
     Input SW_CROSS
```

You will notice that just before this line, we create a SYMBOL called SW_CROSS and assigned it to PORTC.7. Input SW_CROSS tells the compiler to make pin PORTC.7 a digital input.

The next line is one of the most powerful and useful instructions in probably any programming language:

```
     IF
```

This forces the program to ask a question, and then act on the result.
So in this case, if SW_CROSS = 0 then re-start program execution from label DoAgain.

The GOTO instruction when used within the IF statement is actually optional.
`If SW_CROSS = 0 Then DoAgain` and `If SW_CROSS = 0 Then GoTo DoAgain` do exactly the same thing, generate exactly the same machine code, so you can use either with no penalty on program space usage, or performance.

Remember that digital I/O pins can only have a value of 0 (LOW) or 1 (HIGH). The circuit in Fig2 has a 2.2K resistor on pin PORTC.7 (aka SW_CROSS) pulling it to 0v. If the button isn't pressed the pin will affectively be at 0v.

*** DO NOT ALLOW PINS TO FLOAT. ***
This is a great way to have a PIC misbehave. When using pins for digital input, don't allow them to float; this means that somehow, you should force them to be a logic 0, or logic 1. When used with switches using a resistor to tie the pin to a logic level is ideal.

If SW_CROSS doesn't equal 0 (so by default it must equal 1), then the PIC will not execute the GOTO DoAgain part of the statement and will drop down to the line below.
Since the push-button switch pulls this pin to +5v when pressed (which is a LOGIC 1 / HIGH), the only way the test can fail is if the push button is pressed.

Before we try this code out though, it's worth a little further discussion.

This code is actually bad programming and will probably end up causing problems as the program grows in complexity. It's also not very efficient.

The code can be rewritten like this:

```
DoAgain:
    If SW_CROSS = 1 Then
        ' The pushbutton has been pressed
        High LED_WAIT
        DelayMS 1000
        Low LED_WAIT
    End If

    GoTo DoAgain
```

The IF has been changed to read IF SW_CROSS = 1 (so now, has the push button been pressed) rather than the previous, has the push button NOT been pressed.
If it has been pressed, then the indented code is executed. Since BASIC doesn't mind how you format or layout your code, the indentation is just for ease of reading by you; the compiler doesn't really care how you lay your code out. To mark the end of the IF statement, we have a new statement:

`End If`
This tells the compiler that everything between the IF, and the END IF should only be executed IF the condition (SW_CROSS = 1) is true. If the condition is not true, skip to the line after the END IF and continue on from there.

So, we've change the condition from =0, to =1, indented some code and added an extra line for the END IF, and this is supposed to be more efficient?
Well, compiling this code requires one less instruction than the previous version; and one instruction less means less instructions to execute, so it requires less program space and will execute faster. I also personally think it's more intuitive and easier to read.
However, the big benefit comes when you need to test multiple switches.
After the END IF, it would be simple to insert another block of code to handle a second, third or tenth switch. It would be a lot harder with the original code version.

So, the code above will keep looking for the push button to be pressed and when it detects a press, will illuminate a LED for 1 second, turn the LED off, and start waiting for button to be pressed again.

No matter how fast you press the button, you won't be able to press it fast enough for the PIC to miss it… or will you ?

Technobabble

Assuming you've not adjusted any of the PIC configuration registers from those specified in chapter 2, then the PIC is affectively running at 64MHz. However, this PIC requires four clock pulses or cycles per instruction; so it's actually really running at 16MHz, but even so, that equates to 16,000,000 machine code instructions per second, or 16 MIPS.

A quick look at the assembler that the compiler produces shows that to check the input of the port, perform the IF and the GOTO takes around four machine code instructions, which means that the PIC can do this entire process around 4,000,000 times per second; which is 0.00025ms or 0.25us (micro-seconds).
The fastest I could push a push button was 32ms, so the PIC can read the switch around 128,000 times faster than I can press it.

Actually it's a little bit slower than this as the PIC has to read a couple of extra things from memory before it can execute the GOTO but it's still faster than you are.

When fast… just isn't fast enough

At some point your program could get so large and complex that the user can indeed press the push button and your program won't be able to detect it.

The following code is a slight enhancement on the code above. As well as illuminating the LED when the push button is pressed, it toggles LED_RED on and off and has an additional 250ms delay. This delay simulates a busy program and LED_RED should flash around four times per second.

```
DoAgain:
    Toggle LED_RED

    If SW_CROSS = 1 Then
        ' The pushbutton has been pressed
        High LED_WAIT
        DelayMS 1000
        Low LED_WAIT
    End If

    DelayMS 250

    GoTo DoAgain
```

Using this program, it's often possible for the PIC to miss the push button press if you are quick with the press, but not always. This apparent randomness can cause all sorts of problems and wasted time. It could for example look like a possible faulty switch or lose connection in the hardware. A long press is usually detected ok. There are of course ways around this and these will be discussed at a later point.

Simple pedestrian crossing

The pedestrian crossing allows pedestrians to safely cross the road at busy junctions and intersections by temporarily stopping the flow of traffic. Unlike traditional traffic lights, the traffic is allowed to flow freely through the junction until a pedestrian wishes to cross the road at which point they press a button on a panel and this illuminates a “wait” indicator. After a predetermined about of time, a set of traffic light type indicators change, forcing the traffic to stop to allow the pedestrian to cross. After a brief interval (usually around 2 seconds shorter than the average pedestrian can sprint across a busy highway). The traffic lights start to change back to allow the traffic to flow again.

The code here is for a very simple crossing.

```
'-- Configure I/O ports and pins here
    Output PORTC         ' Make ALL of PORTC an output

    Symbol LED_RED       PORTC.0
    Symbol LED_AMBER     PORTC.1
    Symbol LED_GREEN     PORTC.2
    Symbol LED_WAIT      PORTC.3
    Symbol SW_CROSS      PORTC.7
    Input SW_CROSS


'-- Start of main program
    ' Make sure that all the LEDs are in their correct startup settings
    Low LED_RED
    Low LED_AMBER
    Low LED_WAIT
    High LED_GREEN   ' Traffic allowed to flow freely

DoAgain:
    If SW_CROSS = 1 Then
        ' The pushbutton has been pressed
        High LED_WAIT  ' We light the LED to give visual feed-back that they have pressed the
                       ' button. Also, just like a real crossing, no matter how often, or hard
                       ' you press the button it won't make any difference. You now wait !!

        DelayMS 5000 ' Make pedestrian wait for 5 seconds
        Low LED_WAIT ' Clear the visual indication LED

        ' Start to change the sequence of the traffic lights
        ' Green Off, Amber On
        Low LED_GREEN
        High LED_AMBER
        DelayMS 1000

        ' Amber Off, Red On
        Low LED_AMBER
        High LED_RED

        DelayMS 10000 ' Traffic stopped, pedestrian can now cross

        ' Times up... change the lights back to green via the proper sequence
        High LED_AMBER ' Remember the RED is still lit, so now we have RED + AMBER
        DelayMS 1000

        ' Back to normal traffic - Amber Off, Red Off, Green On.
        Low LED_AMBER
        Low LED_RED
        High LED_GREEN   ' Traffic allowed to flow freely again
    End If

    GoTo DoAgain
```

This is really just an amalgamation of the previous traffic lights program, and the push button code from above and should require no further explanation.

As a programming exercise, add two additional LEDs to the circuit, one red and one green and use these two LEDs to indicate to the pedestrian whether it is safe or not to cross the road.

Toggle switches

Toggle switches are simpler in many ways to handle programmatically as they tend to be set to a position and then left for a period of time; usually seconds or longer as opposed to milliseconds.

This has been a brief introduction to connecting switches to the PIC. In subsequent chapters we will look at interfacing to keypads and issues surrounding switch “contact bounce”.