Blink an LED in ASM

Compiled by Simeon Ramjit and Darnell Gracen

Created: 15.01.2018

Updated: 24.02.2018

Table of Contents

Table of Figures

Introduction

This document aims to give a detailed explanation of flashing an LED in ASM using the PIC16 microcontroller and an instruction-based delay. It will cover both the hardware and software aspects of the process and highlights certain considerations that must be made when creating a system using a microcontroller.

Hardware Aspect

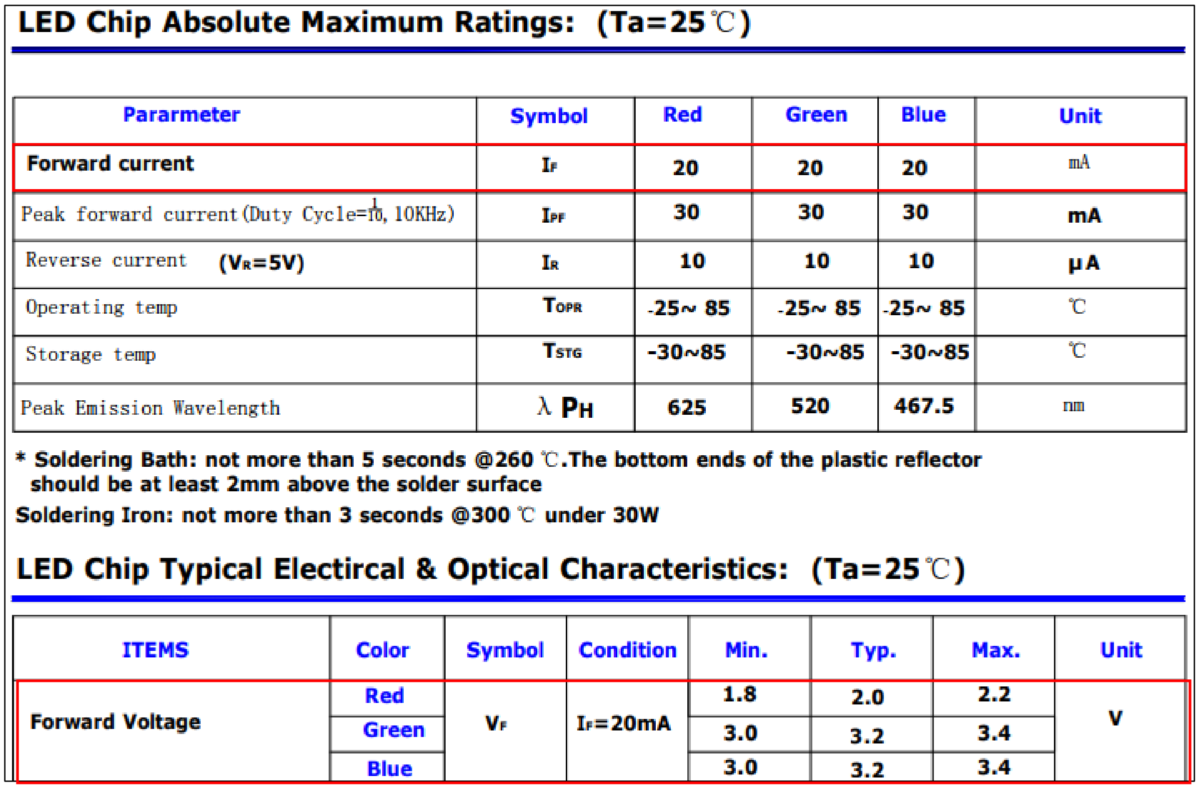
LED Resistor Calculation

LEDs or Light Emitting Diodes produce light by a process known as electroluminescence. To safely illuminate an LED, it must be placed in series with a resistor known as a current limiting resistor due to its function by virtue of its position in a circuit. These resistors are necessary because LEDs suffer from thermal runaway without them and in doing so, get destroyed much faster. The resistor limits the current that an LED can draw assuming a constant voltage source is used and operates it within specifications, but what are the specifications of an LED ? There are numerous aspects to an LED but for its operation the ones we are concerned with are:

- Forward Voltage (V_f)
- Forward Current (I_f)

These can be found in the datasheet of the LED as illustrated on the other page. If I_f is unknown, 5mm LEDs typically have a forward current of 20mA and a V_f of 3V.

The snippet below identifies the two parameters we need, the forward current and the forward voltage. However, there are three values for the forward voltage given, minimum, typical and maximum, so which do we use? The following justification can be made. If we use the typical forward voltage then the LED can withstand *minor* spikes and dips in the supply voltage. This is because a dip in the supply means less current through the LED but still more than the minimum forward current. A spike in the supply means more current through the LED but less than the maximum forward current. Therefore, choosing the 'typical' parameters ensures that for *minor* voltage spikes and dips, the LED stays on.



LED Chip Absolute Maximum Ratings: (Ta=25℃)

Pararmeter	Symbol	Red	Green	Blue	Unit
Forward current	I_F	20	20	20	mA
Peak forward current (Duty Cycle=1/10, 10KHz)	I_{PF}	30	30	30	mA
Reverse current (VV_R=5V)	I_R	10	10	10	μA
Operating temp	T_{OPR}	-25~ 85	-25~ 85	-25~ 85	℃
Storage temp	T_{STG}	-30~85	-30~85	-30~85	℃
Peak Emission Wavelength	λ PH	625	520	467.5	nm

* Soldering Bath: not more than 5 seconds @260 ℃.The bottom ends of the plastic reflector should be at least 2mm above the solder surface

Soldering Iron: not more than 3 seconds @300 ℃ under 30W

LED Chip Typical Electircal & Optical Characteristics: (Ta=25℃)

ITEMS	Color	Symbol	Condition	Min.	Typ.	Max.	Unit
Forward Voltage	Red	V_F	I_F=20mA	1.8	2.0	2.2	V
	Green			3.0	3.2	3.4	
	Blue			3.0	3.2	3.4	

Source: sparkfun.com/datasheets/Components/YSL-R596CR3G4B5C-C10.pdf

Figure 1:LED Datasheet Snippet

As with all diodes, there is a voltage drop associated with it i.e. the forward voltage or the voltage required to turn it on, the forward current is the also the current required for illumination. Both aspects must be met for the LED to light. To calculate the resistor needed, the following formula is used.

$$Resistor\ Value = \frac{Supply\ Voltage\ (V_S) - LED\ Forward\ Voltage\ (V_f)}{LED\ Foward\ Current\ (I_f)}$$

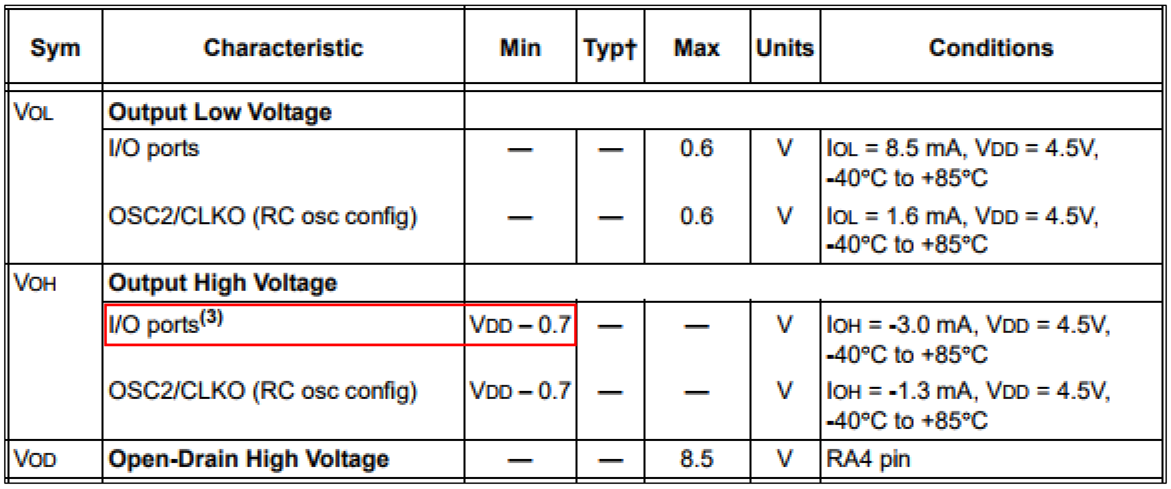
If the calculated value is not a standard value resistor then use a **higher value resistor** since using a lower value will allow a higher current flow that was not designed for. But how does placing a resistor in the circuit limit the current flow and protect the LED? In the formula above the numerator essentially gives the 'excess voltage' in the circuit. This voltage is then divided by the current required by the LED. This means that when current flows it would be a value such that the voltage drop across the resistor is the excess voltage in the circuit and the remaining voltage is the safe voltage for the LED to operate. It doesn't matter if the resistor is before or after the LED since they are in series with one another and the current through them will create the appropriate potential differences.

One last thing to consider is the power rating, now that you have the resistor and current value calculate the power rating of the resistor required for your application using the formula given below.

$$P_{Resistor} = {I_f}^2 R_{Current\ Limiting}$$

Microcontroller Considerations

Microcontrollers are mainly used to control other devices and rarely to source power. Therefore, when using it to power a device/system, the electrical characteristics of it must be considered to avoid damaging it. When sourcing current from the PIC16, a logic high is placed on a pin, however when sinking current, a logic low is placed. Source means that current flows from the PIC16 through the LED to ground and sinking current means that current flows from the supply through the LED, into the PIC and then to ground. Additionally, to sink and source current the pin should be configured as an output. Each pin has a maximum source and sink current (25mA – see pg 173) and voltage output as shown in the table below.

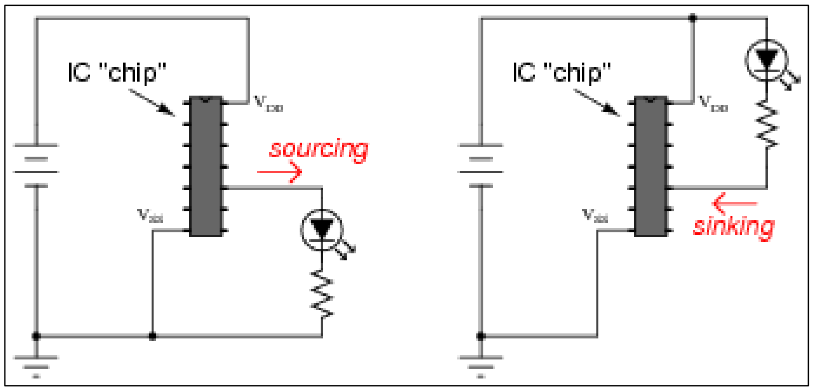


Sym	Characteristic	Min	Typ†	Max	Units	Conditions
V_{OL}	**Output Low Voltage**					
	I/O ports	—	—	0.6	V	I_{OL} = 8.5 mA, V_{DD} = 4.5V, -40°C to +85°C
	OSC2/CLKO (RC osc config)	—	—	0.6	V	I_{OL} = 1.6 mA, V_{DD} = 4.5V, -40°C to +85°C
V_{OH}	**Output High Voltage**					
	I/O ports[3]	V_{DD} – 0.7	—	—	V	I_{OH} = -3.0 mA, V_{DD} = 4.5V, -40°C to +85°C
	OSC2/CLKO (RC osc config)	V_{DD} – 0.7	—	—	V	I_{OH} = -1.3 mA, V_{DD} = 4.5V, -40°C to +85°C
V_{OD}	**Open-Drain High Voltage**	—	—	8.5	V	RA4 pin

Source: PIC16F87XA Data Sheet pg-179

Figure 2: Table snippet of output characteristics of PIC16877A

Looking at V_{OH}, the value stated is $V_{DD} – 0.7$, so assuming a supply voltage of 5V, the output high would be 4.3V, which is the value to be used in the LED calculation (V_S). If the load requires more current or a higher voltage to operate than what the MCU can supply, consider using a power transistor as a switch (TIP31C), don't forget the base resistor when doing this. It should be noted that RA4 is an open drain pin, essentially meaning that it cannot drive an output but it can still be configured as such, so to illuminate an LED on that pin, you would need to sink current.

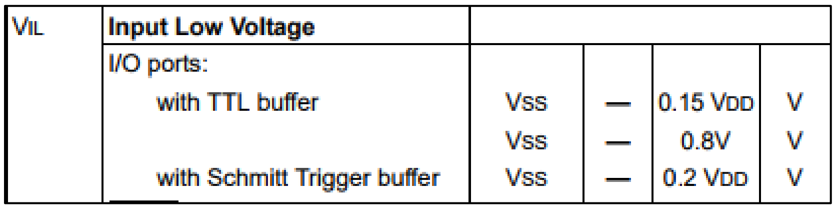


Source: allaboutcircuits.com/worksheets/regulated-power-sources/

Figure 3: Sourcing current (right) Sinking current (left)

Pull-up and Pull-Down resistors

These are used on input pins and are necessary to have a known logic level on a pin when an input signal is not present (prevents floating inputs). Pull-up resistors tie the pin to a positive voltage, i.e. some logic high. They make an input, active low. Pull-down resistors tie the pin to ground i.e. a logic low. They make an input, active low. Usually a 1kΩ – 10kΩ is used. There are only voltage thresholds and no current specifications for an input logic level. Therefore, we have to guarantee that those thresholds are met. Any pin can sink a maximum of 25mA but the intention of a PU/PD resistor is to have it draw the least amount of current possible. Let's use a current of 4mA for convenience of resistor value and assume the pin we're using has a TTL buffer and we need to calculate a pull-down resistor.



V_{IL}	**Input Low Voltage**				
	I/O ports:				
	with TTL buffer	V_{SS}	—	0.15 V_{DD}	V
		V_{SS}	—	0.8V	V
	with Schmitt Trigger buffer	V_{SS}	—	0.2 V_{DD}	V

Source: PIC16F87XA Data Sheet pg-178

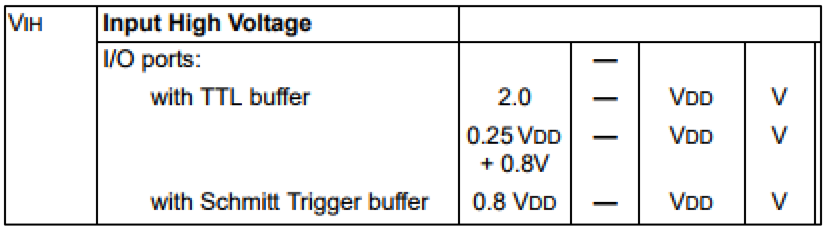
Figure 4: Input Low logic levels according to buffer

The *maximum* voltage level for an input low with a TTL buffer is 0.8V. Assuming a current flow of 4mA that gives a resistor value of:

$$\frac{5-0.8}{4}\times 10^{3} = 1050\Omega \cong 1k\Omega$$

For a pullup resistor the same process applies. The *minimum* voltage level for an input low with a TTL buffer is 2V. Assuming a current flow of 4mA that gives a resistor value of:

$$\frac{5-2}{4}\times 10^{3} = 750\Omega \cong 1k\Omega$$



V_{IH}	**Input High Voltage**				
	I/O ports:		—		
	with TTL buffer	2.0	—	V_{DD}	V
		$0.25 V_{DD} + 0.8V$	—	V_{DD}	V
	with Schmitt Trigger buffer	$0.8 V_{DD}$	—	V_{DD}	V

Source: PIC16F87XA Data Sheet pg-178

Figure 5: Input High Logic Levels according to buffer

To determine the type of buffer on the pin you're using, read the pin descriptions starting on page 8 and look at the note at the base of the page. That will indicate what buffer the pin according to its use in the program.

Switches

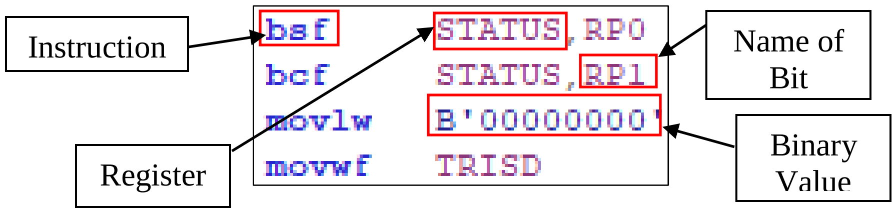
Using a switch is simply defining a pin as an input in the TRIS register and then using `btfss` or `btfsc` (check pg-161 for instruction description) and depending on the input you `goto` a specific routine. Then after that routine has completed you test the input again. On the hardware side, pullup/down the input and write your code to suit.

Software Aspect

This section will describe how the code is written as well as the process for developing the code. Before we get into the ASM just a small introduction if you've never written it before. Assembly language is MCU specific. A large part of it is manipulating registers which are blocks of memory capable of holding only 1 byte. Each hardware peripheral is in a specific 'bank' and is controlled by the bits in a register. There are four different types of registers, control, configuration, data and state. To flash an LED, we need to manipulate the TRIS (configuration) register (decides if pins are inputs or outputs) and the PORT (state) register (decides the logic value at that pin). To manipulate any register, we need to determine what bank the register is in, select that bank and then pass values from the program to the Working Register (WREG) and then move that to the peripheral register. All project related files can be found here.

Configuring Hardware Peripherals

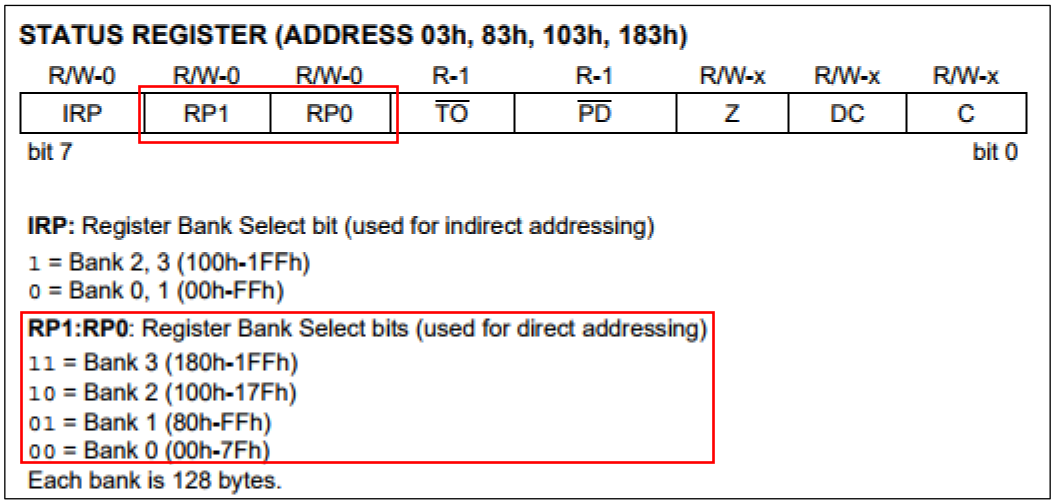
Let's assume the LED is on PORTD<7> (RD7 or PORTD pin 7) and we want to light the LED when the logic level is high, this means we're sourcing current so we configure RD7 as an output. This is done by 'clearing' TRISD<7> (i.e. explicitly making the 7th bit '0'). The same process would be followed for any other even RA4. The configuration is shown below.



```
bsf     STATUS,RP0
bcf     STATUS,RP1
movlw   B'00000000'
movwf   TRISD
```

Figure 6: Tris Register Configuration

Now what's happening in those four lines? Earlier we mentioned that registers are in banks (see pg 17 for complete bank listing) and to manipulate a register we need to ensure the PIC is in that bank. This is done by ensuring that certain bits in the ‘STATUS’ register are cleared (0- BCF) or set (1 - BSF) according to the register documentation. The memory mapping shows that the TRISD register is in bank one, hence the setting of RP0 and the clearing of RP1.



STATUS REGISTER (ADDRESS 03h, 83h, 103h, 183h)

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit 7							bit 0

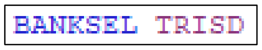
IRP: Register Bank Select bit (used for indirect addressing)
1 = Bank 2, 3 (100h-1FFh)
0 = Bank 0, 1 (00h-FFh)

RP1:RP0: Register Bank Select bits (used for direct addressing)
11 = Bank 3 (180h-1FFh)
10 = Bank 2 (100h-17Fh)
01 = Bank 1 (80h-FFh)
00 = Bank 0 (00h-7Fh)
Each bank is 128 bytes.

Source: PIC16F87XA Data Sheet pg-22

Figure 7: Snippet of STATUS register bit function description

An alternative to selecting the appropriate bank the following compiler directive can be used.



```
BANKSEL TRISD
```

Figure 8: Alternative way to do a bank select

After we selected the appropriate bank, we’re now ready to move a value into the register such that it makes TRISD<7> an output i.e. TRISD<7> must be 0, the rest of bits for the rest of pins can be left as 0 or 1, it does not matter in this case. This explains the third line where the binary value 00000000 is moved into the working register and then transferred from the working register to the TRISD register (4th line). The zero is in red because looking at figure 9 we see that to control TRISD<7> we need to manipulate the corresponding bit.

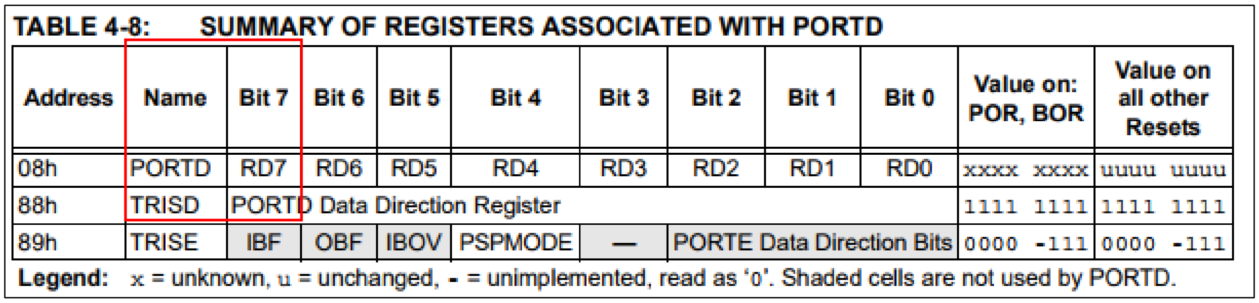


TABLE 4-8: SUMMARY OF REGISTERS ASSOCIATED WITH PORTD

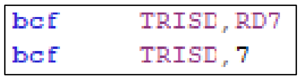
Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other Resets
08h	PORTD	RD7	RD6	RD5	RD4	RD3	RD2	RD1	RD0	xxxx xxxx	uuuu uuuu
88h	TRISD	PORTD Data Direction Register								1111 1111	1111 1111
89h	TRISE	IBF	OBF	IBOV	PSPMODE	—	PORTE Data Direction Bits			0000 -111	0000 -111

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by PORTD.

Source: PIC16F87XA Data Sheet pg-48

Figure 9:Bit Layout for registers associated with PORTD

An alternative to doing this after selecting the bank can be seeing in figure 10. Either of these two lines would work but then you run the risk of not knowing what state the rest of the pins are since you did not explicitly define them.

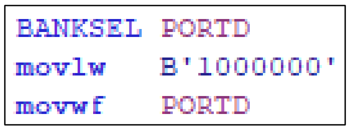


```
bcf      TRISD,RD7
bcf      TRISD,7
```

Figure 10: Bit Change Alternative

Changing LED state

When sourcing current, to turn the LED on a logic high must be placed on the pin with the LED. This is done by manipulating the PORTD register. So, to light the LED PORTD<7> must be high and to turn it off it must be low.



```
BANKSEL PORTD
movlw   B'1000000'
movwf   PORTD
```

Figure 11: PORTD manipulation

As we can see the same approach was used to write to the PORTD register, as with the TRIS register, with a logic high on bit 7 (see fig 9)

Instruction Based Delay

There are three ways to flash an LED an instruction-based delay, a timer generated interrupt and a PWM signal (more used for dimming). In this document however, we will only consider the instruction-based delay. From the manual:

"One instruction cycle consists of four oscillator periods; for an oscillator frequency of 4 MHz, this gives a normal instruction execution time of 1 µs. All instructions are executed within a single instruction cycle, unless a conditional test is true, or the program counter is changed as a result of an instruction. When this occurs, the execution takes two instruction cycles with the second cycle executed as a NOP."

What this essentially means is that 1 instruction takes 1 instruction cycle to execute and the time for one instruction cycle (T_{CY})(due to fetch, decode etc) is $\frac{\frac{1}{F_{OSC}}}{4}$ i.e. 1µs this means that we can use instructions as a delay by simply making the MCU execute a bunch of instructions that essentially do nothing but consume time. So how do we calculate this? Let's assume we want a delay frequency of 1Hz which is a delay time of 1s and the clock frequency is 4MHz.

$$1\ instruction = 1\mu s$$

$$x\ instructions = 1s$$

$$No. of\ instructions = \frac{1s}{1\mu s} = 1 \times 10^6\ instructions$$

That's the intuitive way, however there is a formula based on the same principle that gives the number of instructions.

$$No. of\ instructions = \frac{Required\ Time\ \times Oscillator\ Frequency}{4} = \frac{1s \times 4MHz}{4} = 1 \times 10^6$$

It's tempting to copy and paste Nop a million times but there's a relatively simpler way. We could load a register with the value 1 million and then keep subtracting 1 each time till we hit zero and that'll give us roughly the delay we're looking for, but a register can only hold 8 bits i.e. a maximum value of $2^8 - 1 = 255$. What we can do is create of bunch of nested loops that decrements registers and the amount of times that the decrement occurs will sum to the number of instruction cycles we need and therefore the delay. To get the number of loops and the value in the registers we'd go through the following process.

Loop Calculation

One register can hold a value of 255 and to decrement that value (`decfsz`) and form a loop (`goto`) would take three instruction cycles, i.e. 255*3 = 765us. So, with a loop of 765us we'd need to run that loop:

$$\frac{1 \times 10^6}{765} \cong 1307\ times$$

But that can't fit in one register so we'll keep dividing only by the maximum register value since we've already accounted the run-time, in seconds, of the first loop.

$$\frac{1307}{255} \cong 5\ times$$

Which is less than 255 which means we can fit it in one register. In total we'd have a delay time of $255 \times 3 \times 255 \times 5 = 975{,}375\mu s$ which is not 1s but we'll fine tune it later using the stopwatch.

What the calculation shows is that we need to decrement one register 255 times then do that 255 times again and run those two loops 5 times again to get a value close to one million, so we need three registers; one for each value. If you re-checked the calculations you'd end up with decimal numbers as the remainders but decimals can't be placed into registers, only integers.

Instruction Based Delay in ASM

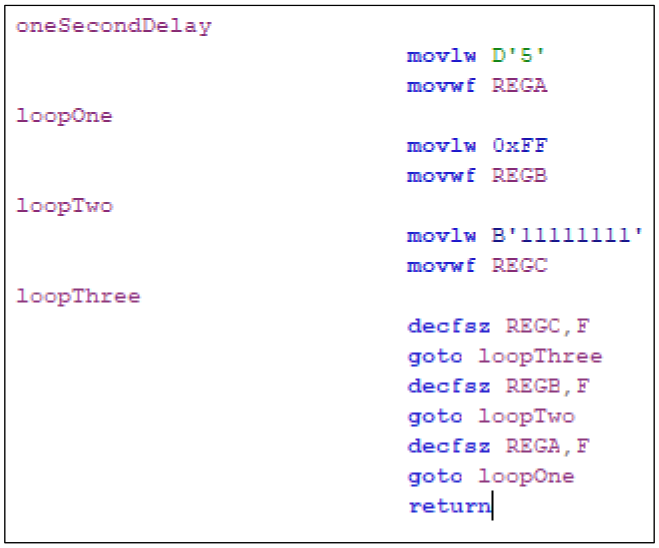

```
oneSecondDelay
                                movlw D'5'
                                movwf REGA
loopOne
                                movlw 0xFF
                                movwf REGB
loopTwo
                                movlw B'11111111'
                                movwf REGC
loopThree
                                decfsz REGC,F
                                goto loopThree
                                decfsz REGB,F
                                goto loopTwo
                                decfsz REGA,F
                                goto loopOne
                                return
```

Figure 12: 1s Delay Routine in ASM

You'll notice that in each of the lines with `movlw`, the representation changes from Decimal to Hexadecimal and then to Binary. This is an illustration of the way you can express the numbers you use in the compiler, use whichever is convenient to you.

MPLAB Debugging Tools

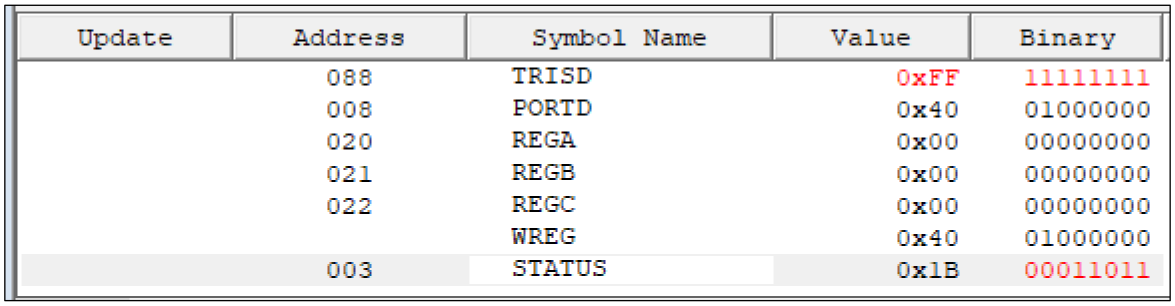
A large part of coding in ASM relies on the debugging tools provided by the MPLAB IDE. They include the following:

1. Watch Window – all registers and the value inside them are displayed here as the program runs. Useful to know if your code is manipulation registers as it should.
2. Stopwatch – times the execution of your code line by line, both in instruction cycles and seconds
3. Breakpoints – These can be used whenever you think your code is getting stuck in a loop or to see if code is executing at all. Put a break point in and if the program stops at that point you know the problem you think you having doesn't exist.

4. Stimulus Workbook – puts simulated values on certain peripherals as it would be in real life to test branch execution
5. Simulator – the aforementioned tools are available only when the simulator is active. Click Debugger > Select Tool > MPLAB SIM

Watch Window

Click 'View' on the toolbar and look down for 'Watch' this presents you with a blank screen. Double click on the field 'Symbol Name' and then type the name of the register you want to view while you *animate* through the program. Right clicking on the row with the column names allows you to add fields with the different number representations. Adding all the registers in the program will give a watch window as seen in figure 13.



Update	Address	Symbol Name	Value	Binary
	088	TRISD	0xFF	11111111
	008	PORTD	0x40	01000000
	020	REGA	0x00	00000000
	021	REGB	0x00	00000000
	022	REGC	0x00	00000000
		WREG	0x40	01000000
	003	STATUS	0x1B	00011011

Figure 13: Watch window before first step through

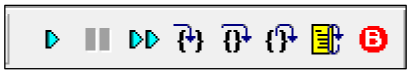
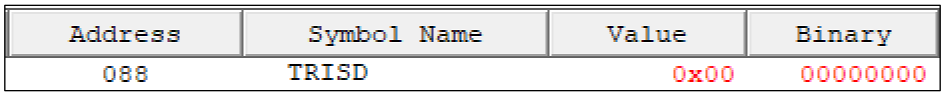


Figure 14: Simulator code-execution options

Look for the buttons in figure 14 when in the simulator and mouse over them to see their purpose. Using the **step into** button we can manually 'execute' one line of code at a time. If we step till just after we move the configuration value into TRISD we get figure 15.



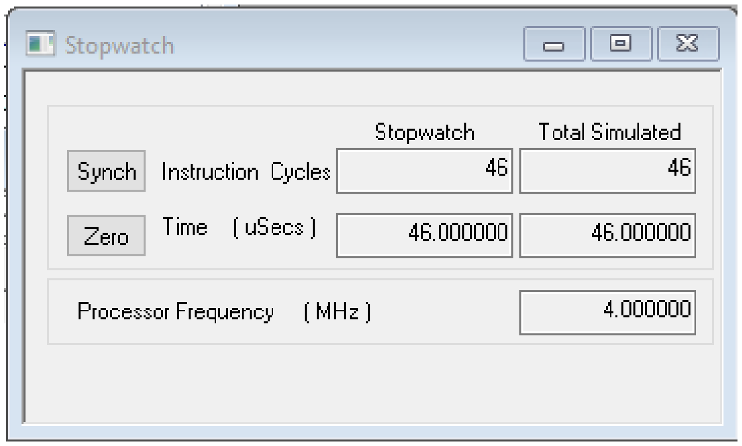
Address	Symbol Name	Value	Binary
088	TRISD	0x00	00000000

Figure 15: Value of TRISD register after moving in configuration value

With this small demonstration we can see how to use the watch window to determine whether or not registers have the value they should have as the program executes before burning it to the PIC.

Stopwatch

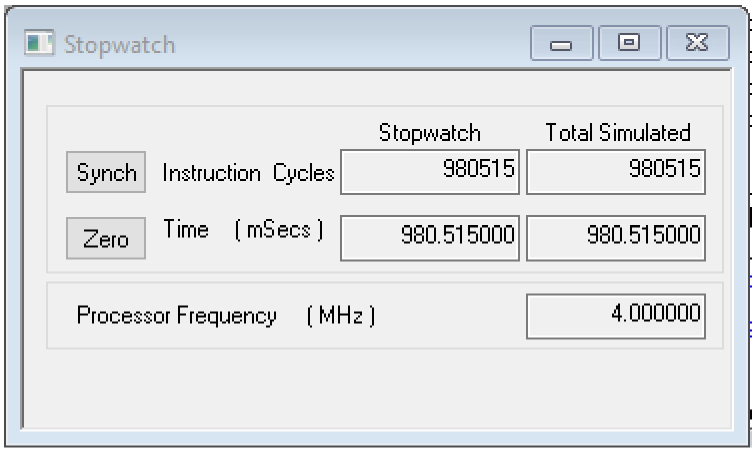
Click Debugger > Stopwatch. Change the oscillator frequency to 4MHz, Click Debugger > Settings. As mentioned previously this gives the execution time in seconds and instruction count. Let's execute till just before we call the delay routine.



	Stopwatch	Total Simulated
Instruction Cycles	46	46
Time (uSecs)	46.000000	46.000000
Processor Frequency (MHz)		4.000000

Figure 16: Execution time just before calling delay routine

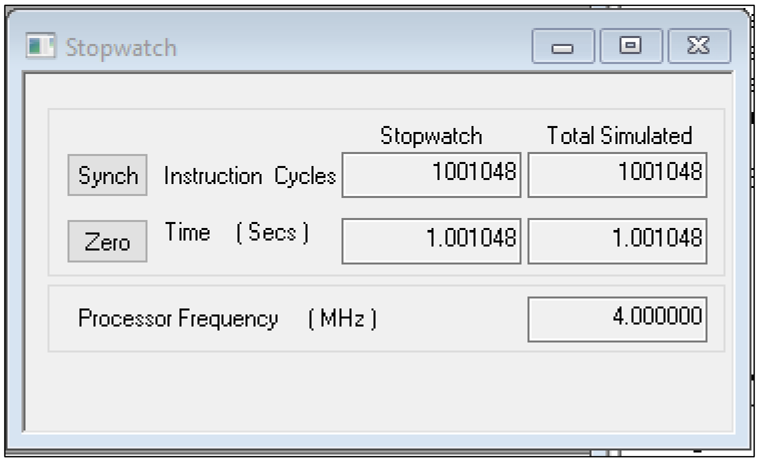
Now let's **step over** the delay routine to avoid having clicking a million times.



	Stopwatch	Total Simulated
Instruction Cycles	980515	980515
Time (mSecs)	980.515000	980.515000
Processor Frequency (MHz)		4.000000

Figure 17: Execution time after calling delay routine

Now we know our delay is almost what it should be and by adjusting the loop values it can be tailored to get closer to 1s.



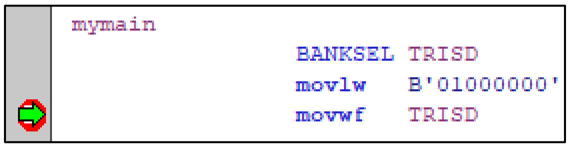
```
                         Stopwatch   Total Simulated
Instruction Cycles         1001048           1001048
Time  ( Secs )            1.001048          1.001048
Processor Frequency  ( MHz )                4.000000
```

Figure 18: Delay routine time after fine-fine tuning

The error won't be discernable to the human eye and if you need a delay that's 2s you could simply create a 1s delay and run it twice.

Breakpoints

Double click ***on the line*** that you want the breakpoint to be on and when you click **'Animate'** the program should run and stop at that point with the Program Counter or 'little green arrow' being on the breakpoint. If it doesn't reach the breakpoint then you know it's getting stuck in a loop or later on, if your code jumps into the Interrupt Service Routine.

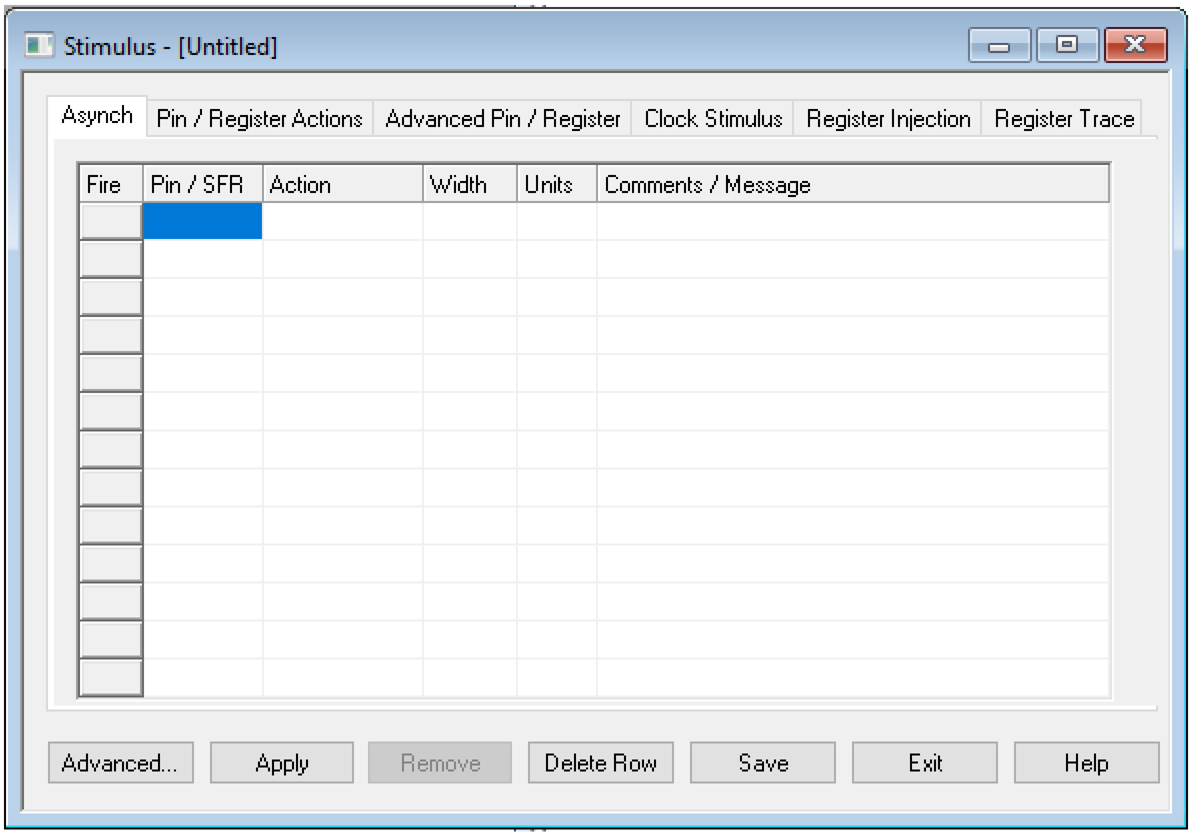


```
mymain
            BANKSEL TRISD
            movlw   B'01000000'
            movwf   TRISD
```

Figure 19: PC and breakpoint at the same point indicating good execution

Stimulus Workbook

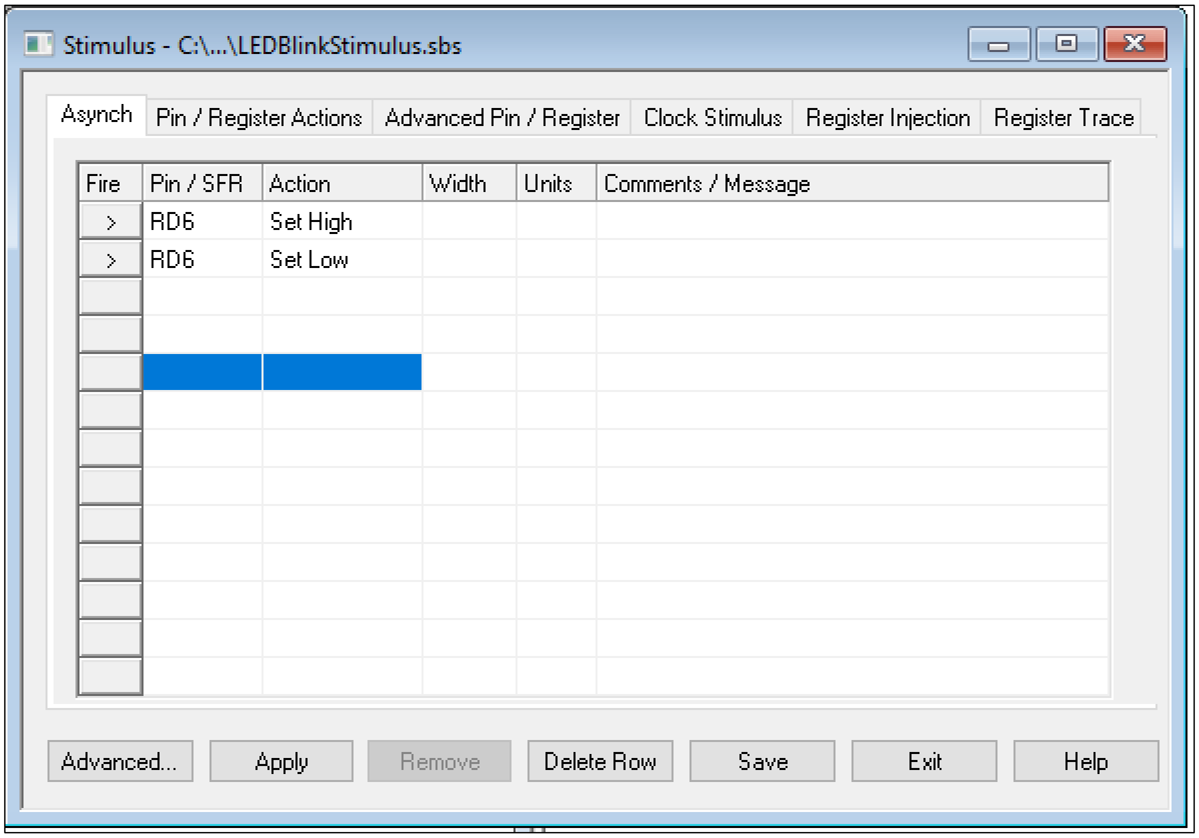
Click Debugger > Stimulus > New Workbook and you'll be greeted with figure 20. The workbook essentially acts as a signal on a pin. If you had a switch to control whether the LED was on or off then that would be a real, physical input that the program relies on. To test branch execution based on inputs we define stimulus in the workbook and fire it when testing.



Fire	Pin / SFR	Action	Width	Units	Comments / Message

Figure 20: Workbook before stimulus definition

To define a stimulus simply click a cell under ‘Pin/SFR’ and select the pin that would act as the input. And then to the right, select the corresponding under ‘Action’ and choose the action we want. Let’s assume we have a switch on PORTD<6>, the switch could either be on (input Logic High) or off (input Logic Low). The stimulus workbook can be seen in figure 21.

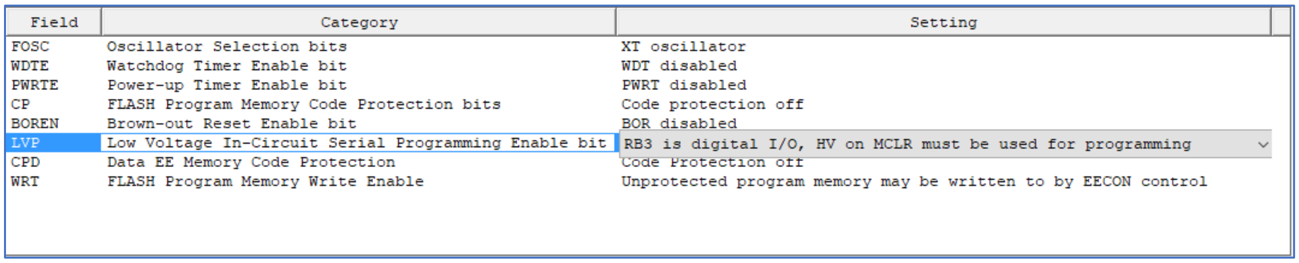


Fire	Pin / SFR	Action	Width	Units	Comments / Message
>	RD6	Set High			
>	RD6	Set Low			

Figure 21: Workbook after stimulus definition

Don’t forget to save it. To use the stimulus simply click the left-pointing arrow on the row with the action on the pin you want to perform and check the watch window to see if the value changed when you step again and then see if your branch executes as needed.

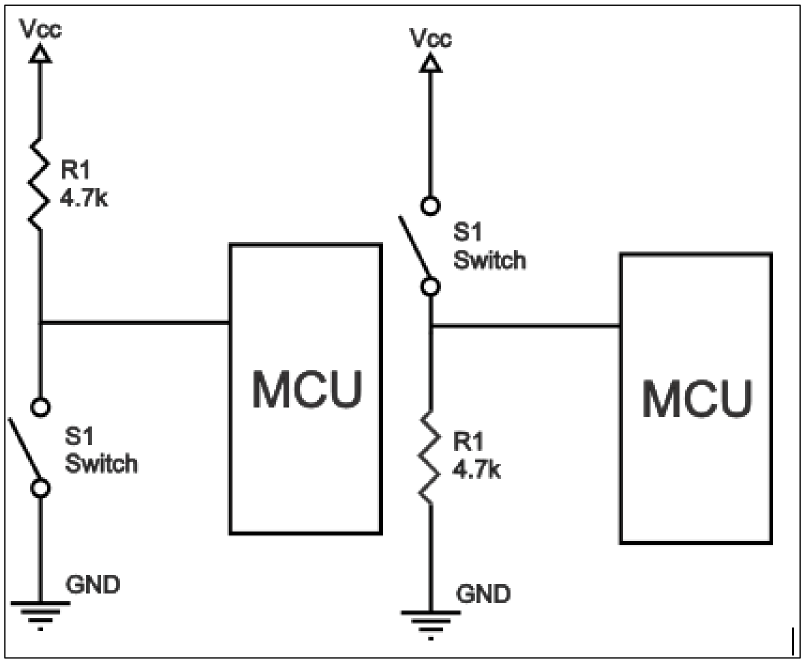
All project related files can be found here.



Field	Category	Setting
FOSC	Oscillator Selection bits	XT oscillator
WDTE	Watchdog Timer Enable bit	WDT disabled
PWRTE	Power-up Timer Enable bit	PWRT disabled
CP	FLASH Program Memory Code Protection bits	Code protection off
BOREN	Brown-out Reset Enable bit	BOR disabled
LVP	Low Voltage In-Circuit Serial Programming Enable bit	RB3 is digital I/O, HV on MCLR must be used for programming
CPD	Data EE Memory Code Protection	Code Protection off
WRT	FLASH Program Memory Write Enable	Unprotected program memory may be written to by EECON control

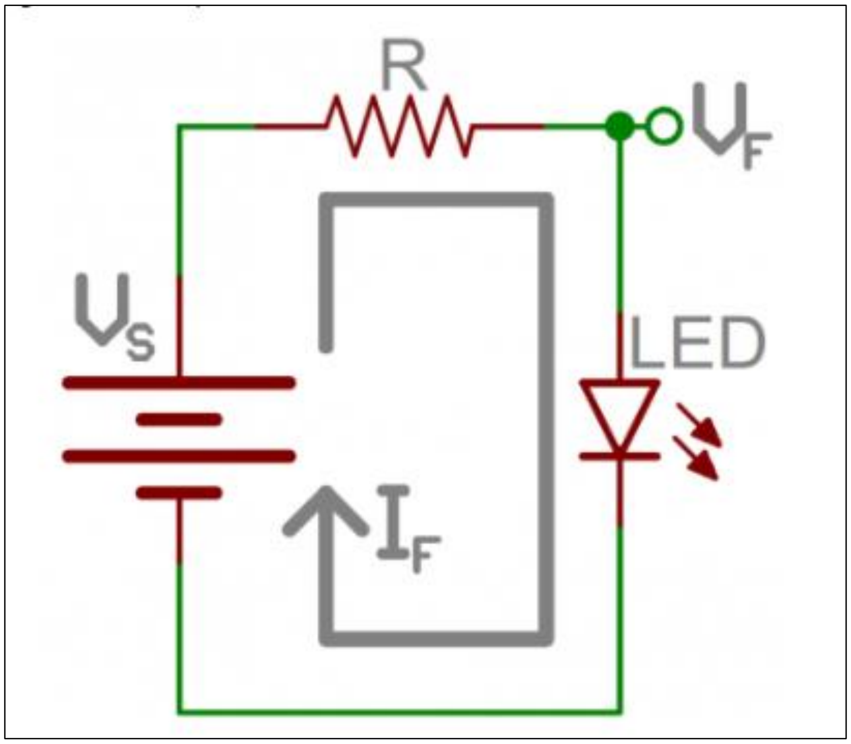
Figure 22: Appropriate configuration bits

Schematics



Source:.resistorguide.com/pull-up-resistor_pull-down-resistor/

Figure 23: Pull-up Resistor (Left) Pull-Down Resistor (Right)



Source: learn.sparkfun.com/tutorials/resistors/example-applications

Figure 24: Current Limiting Resistor

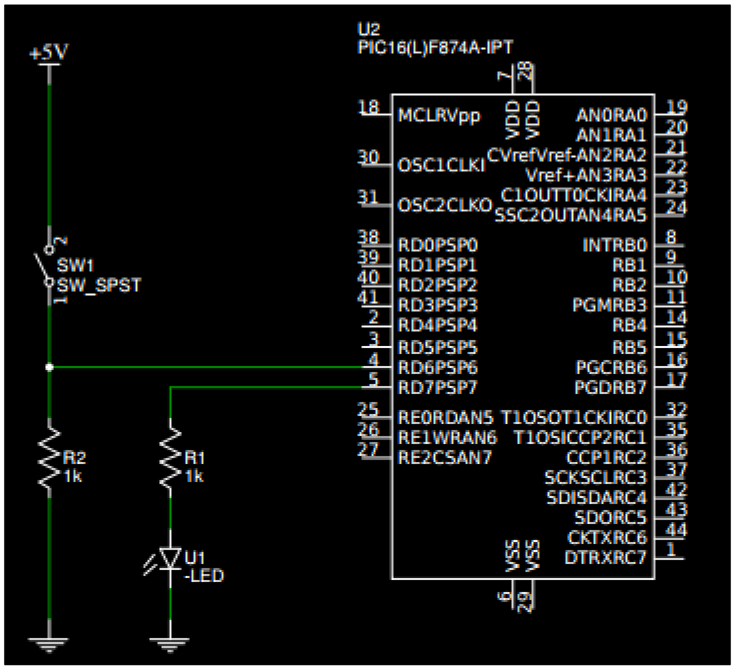


Figure 25: PIC16LF877A with switch and LED