Debouncing and Stepper Motors

Associate Professor Sarath Kodagoda

Mr Kyle Alvarez

Timers and Interrupts

Timers allow for scheduling of processes with low overhead – interrupt driven

Interrupts are great for scheduling time dependent tasks using timer interrupts, trigger functions caused by an external event– through a port state change interrupt etc…

Start thinking about how you can use interrupts for your assignment

For now, we are gonna use them for debouncing and flashing a heartbeat LED

Feel free to go ahead and try achieving different frequencies on your HBLED or control multiple LEDs with different frequencies

Just remember, try and spend as little time as you can while in the ISR. At most, check and clear flags, increment/decrement counters, toggle some states. Definitely do not do any function calls or delays while inside the ISR

Also note that you cannot call the ISR directly like a normal function and it doesn't take or return any varibales. The program will only go into the ISR once certain conditions are fulfilled – such as a timer overflow. These interrupts have to be enabled. For a list of available interrupts, please read the relevant PIC datasheet.

Debouncing

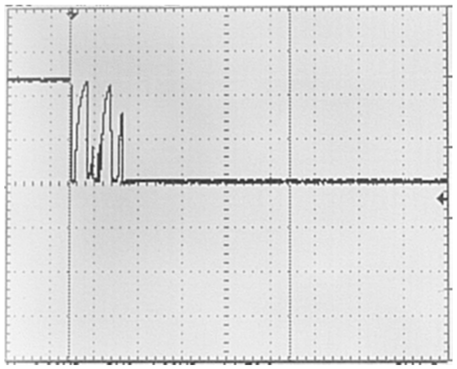
/* Bouncing is the tendency of any two metal contacts in an electronics device to generate multiple signals as the contacts close or open */

The amount of bouncing is dependent on the quality of the switch but it cannot be entirely removed as it is a physical phenomenon.

There are hardware and software solutions for debouncing, we'll be focussing on a software based approach. Hardware based approaches include Schmitt triggers, low pass RC circuits etc… all of which require hardware modifications to your kits. Software approaches are usually taken as they are essentially free (compared to physical hardware parts) and simple to implement.

/* How would we go about doing this? */
A simple way would be to do a quick delay (50ms maybe?) once a button has been detected to have been pressed. If that button is still pressed after the delay, then one can assume that the "bounce" of the switched has finished, transients are gone and to flag the button as pressed.

/* This is not a very good implementation of debouncing. Why? */
Checking the button state and delaying before checking again means causing a delay in the response of your program. i.e. your microcontroller can't do other tasks but wait. Usually you want to keep the use of delays to a minimum.



Debouncing

/* Instead, we can use an interrupt based approach to implement a debouncing subroutine and remove the use of delays*/

How do we do that?
What are we trying to achieve?

Think back to your Mechatronics 1 Major Project. How did you debounce the pushbuttons then? Can we use something similar or use a similar process (without using the CPLD)?

Let's try one approach. What if we keep a counter that counts how many times a button has been registered as pressed? We could check say, every 5ms and see if it's still high/pressed. If it's still high, increment the counter. If it's not, then the button is still bouncing, reset the count. We could then set an arbitrary value (sounds like a good place for defined constant) that once the counter is higher than that value (i.e. every time we checked the state of the button, it was high "x" times in a row, else it would have been reset) we can safely say that the bouncing has stopped and we can raise a flag to signify that the button is now considered "pressed".

How would we check every 5ms? (Hint: starts with "timer0" and ends in "interrupt"

If you have time, try and flowchart this process. The concept should make more sense once you start trying to draw it all down. Else, try and write down the pseudocode before going to the next slide. We want to emphasise understanding of the process rather than coding itself.

Debouncing - Pseudocode

```
#include <xc.h>
//Defines
#define _XTAL_FREQ 20000000                               // From previous exercise
#define TMR0_VAL 100                                      // “” “”
#define LED0 RB0                                          // Lets use PORT B pin 0 and connect it to an LED to show if the button has been pressed or not
#define PB0 !RB1                                          // We’ll NOT the pushbutton input to convert it to active high logic
#define DEBOUNCE_REQ_COUNT 10                             // Ehhh we can make this any reasonable number

volatile bit pb0Pressed = 0;                              // Bit flag to signal if the pushbutton is pressed
volatile bit pb0Released = 0;                             // Bit to signal if the pushbutton has been released
volatile unsigned char pb0DebounceCount = 0;              // Let this be the counter for seeing how many times the button is high in a row


//global variables
volatile unsigned char rtcCounter = 0;


void init(){                                              // Initialise it the same way you did last week. Make some Port B pin outputs, setup interrupts etc…
              }


void main(void){
              // Probably best to call the initialise function first
              while(1){                                   // Loop forever!
                            if(pb0Pressed){               // If the pressed flag has been raised,
                                                          // We can clear the flag
                                                          // And toggle the LED
                            }

              }
}
```

Debouncing - Pseudocode

```
/* Now the tricky part, time to write the ISR */
`
void interrupt isr(void){
        if (TMR0IF){                        // If the timer 0 flag interrupt has been raised, 1ms has passed
                // Clear the flag
                // Res-Set the timer0 value

                /* Debounce time */
                if (the pushbutton’s been pressed){
                        // Probably want to increase our debounce count
                        if (the debounce count is higher than the required count and if the pushbutton’s state is released) {
                                // Raise the pressed flag
                                // Clear the released flag
                        }
                }
                else {
                        // Reset the debounce count
                        // Probably want to raise the released flag to signify that the button hasn’t been pressed
                }
        }
}
```

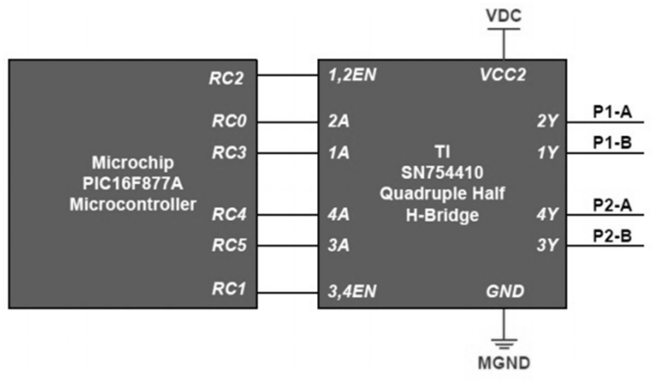
Stepper Motor

/* Diagram shows the interfacing scheme between the PIC, SN754410 and Unipolar Stepper Motor */

In order for the rotor to rotate, the phases must be periodically energised in a specific order, typically P1-A, P2-A, P1-B, P2-B, P1-A and so-on for clockwise rotation and the reverse order for counter-clockwise rotation.

The speed of the rotation is controlled by varying the time interval between each energising pulse.

Note that there is a minimum time required to energise the windings, try with your stepper motor to see what the limits are. e.g. try energising between the sequence with 1ms between them



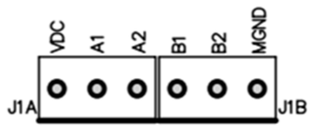
Stepper Motor

Jaycar YM2751 wiring colour scheme

Phase	Designator	Wire colour
Winding A Common (Centre Tap)	12V	Grey
Winding B Common (Centre Tap)	12V	Green
Winding A	A1	Yellow
Winding $\overline{A}$	A2	Blue
Winding B	B1	Brown
Winding $\overline{B}$	B2	Red

Minebea PM35L-048-CCD5 wiring colour scheme

Phase	Designator	Wire colour
Winding A Common (Centre Tap)	12V	Red
Winding B Common (Centre Tap)	12V	Red
Winding A	A1	Black
Winding $\overline{A}$	A2	Brown
Winding B	B1	Orange
Winding $\overline{B}$	B2	Yellow



Screw Terminals on the Stepper Motor Peripheral Module

State	PORTC 76543210	Phase	ENABLE	CTRL	Comment
1	00111100	A1	RC0	RC2	Single/Half
2	00101110	A1/B1	RC0/RC4	RC2/RC1	Half Only
3	00101011	B1	RC4	RC1	Single/Half
4	00100111	B1/A2	RC4/RC3	RC2/RC1	Half Only
5	00110101	A2	RC3	RC2	Single/Half
6	00010111	A2/B2	RC3/RC5	RC2/RC1	Half Only
7	00011011	B2	RC5	RC1	Single/Half
8	00011110	B2/A1	RC5/RC0	RC1/RC2	Half Only
Off	00111001	All windings deengerised			

Note: Odd states are for both single and half stepping, while the even state is only for half stepping

Stepper Motor Exercise

/* Some exercises to get you started with using the stepper motor*/

When initialising the stepper motor, energise it to a known winding. E.g. rifle through the states/steps from 0 to 7 or vice versa

Try and get the stepper motor to rotate indefinitely when powered on

Try and get the stepper motor to rotate indefinitely when a button is not pressed only to rotate in the opposite direction when the button is pressed

Try and get the stepper to move half steps when a button is pushed. Remember, your buttons should be debounced! Or if you want to see debouncing in action, try and see how many steps your stepper motor does on an undebounced button.

Try and get the stepper motor to rotate 180 degrees when a button is pressed

Stepper Motor Example

```
//Define Osillator  freq  20Mhz
//Define timer0 start count ->100
//Define stepping sequence
#define STEP0 0b00111100
#define STEP1 0b00101110
#define STEP2 0b00101011
#define STEP3 0b00100111
#define STEP4 0b00110101
#define STEP5 0b00010111
#define STEP6 0b00011011
#define STEP7 0b00011110

signed char cstep = 0;  //stores current step

void setup (void){
                  //Set PortC Output
                  //Setup other stuff etc… (like interrupts if you want to debounce etc…), PORTB for LEDs or pushbuttons
}

void move (char steps) { // Step parameter is how many steps you want to travel
                  //Loop number of steps     “  i.e. for(;steps!=0;--steps)  “”
                                    switch (cstep){
                                                      // Energise according to state, increase state count. If you’ve hit 7, wrap around back to 0
                                                      case 0:           PORTC = STEP1; cstep++; break;
                                                      // Insert the rest of the cases here
                                                      case 7:           PORTC = STEP0; cstep = 0; break;
                                                      // Default scenario – de-energise stuff
                                                      default: PORTC = 0x00; break;
                                    }
                                                      //10mS Delay
                  }
}


void main (void) {
                  // Call setup
                  // Call move, do stuff etc…

}
```