

Using and Programming The 7 Segment Display Prototype Circuit

Aaron Reason (2312456)

L4 Micros Peripherals and Interfaces - ACDA4004

Embedded Electronics Degree Apprenticeship

Table of Contents

Section 1: Circuit Recreation.....	4
Entire Circuit.....	4
Crystal Oscillator.....	5
PIC16F84A.....	6
Buttons.....	7
Display.....	8
Sounder.....	9
Section 2: Printing 0-F On The Display.....	10
Section 3: Coding Tasks.....	12
Task 1: Display 0-F At 0.2s Frequency.....	12
Task 2: Display "L" or "r".....	17
Task 3: All On or All Off.....	19
Task 4: Display 0-5-9.....	21
Task 5: Button Sound Producer.....	25
Task 6: Synthesiser PWM.....	28
References.....	37

Table of Figures

Figure 1: Entire Schematic.....	4
Figure 2: Crystal Oscillator and Capacitors.....	5
Figure 3: PIC16F84A In Proteus.....	6
Figure 4: Buttons In Proteus.....	7
Figure 5: LED Display.....	8
Figure 6: Sounder.....	9
Figure 7: Annotated Seven Segment Display.....	10
Figure 8: Task 4 Oscilloscope.....	26
Figure 9: PWM Signal.....	28
Figure 10: Middle Pulse Width.....	33
Figure 11: Max Pulse Width.....	33
Figure 12: Minimum Pulse width.....	33

Figure 1: Entire Schematic

Crystal Oscillator

The PIC16F84A can operate in four oscillator modes, for this circuit the XT (Crystal / Resonator) mode is being used (**Figure 2**). The crystal oscillator is connected to the OSC1 and OSC2 pins for clock input and output respectively. [1, p6]

The oscillator is set to 4MHz, which means the PIC is also configured to have a 4Mhz clock frequency in its properties to ensure everything operates as expected.

The 22pF values of the capacitors fall in line with the specification of tested ranges for this type and frequency of oscillator. These specific values were derived from the assignment board. [1, p22]

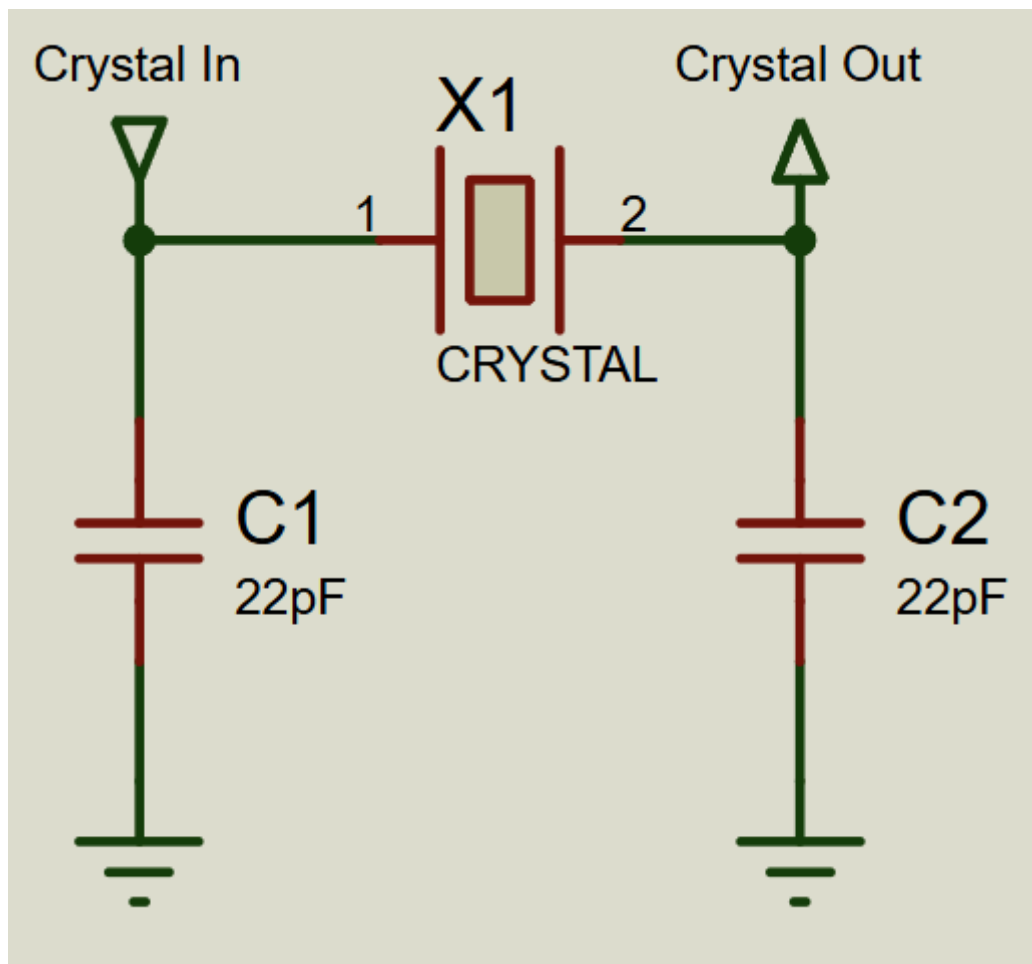


Figure 2: Crystal Oscillator and Capacitors

PIC16F84A

The microcontroller being used is the PIC16F84A (**Figure 3**). This PIC has a 5-bit and 8-bit input / output (I/O) port . These are labelled as PORTA and PORTB accordingly. PORTA and PORTB have data direction registers labelled TRISA and TRISB which control whether the port acts as an input or output. [1, pp15-16]

For this board, PORTA acts as an input port, being connected to a pair of buttons, while PORTB acts as an output port, being connected to the seven segment display and sounder. This setting is configured within the assembly source code by moving an 5-bit / 8-bit value to the *trisa* / *trisb* symbols, where each bit corresponds to input (1) or output (0). [1, pp15-16]

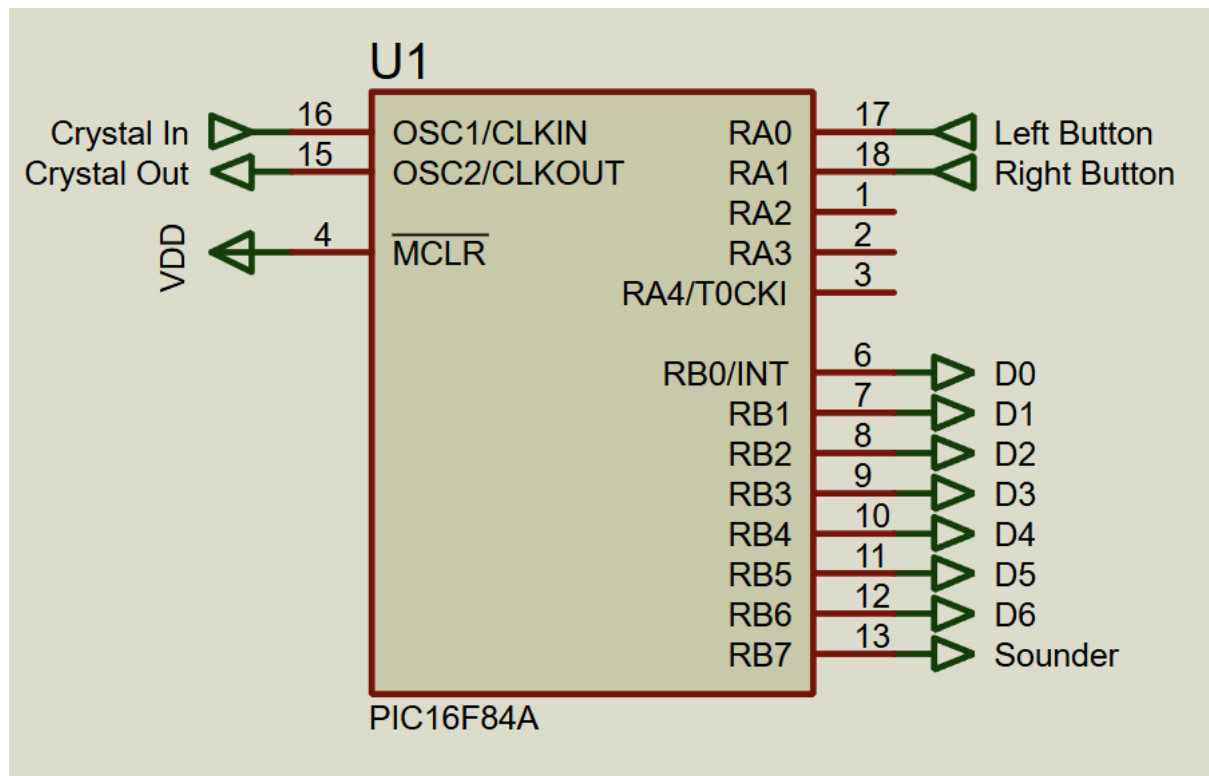


Figure 3: PIC16F84A In Proteus

Buttons

For external inputs to the system there are two buttons (**Figure 4**). These are connected to pins A1 and A2 and are used for controlling the sounder or what is displayed on the seven segment display.

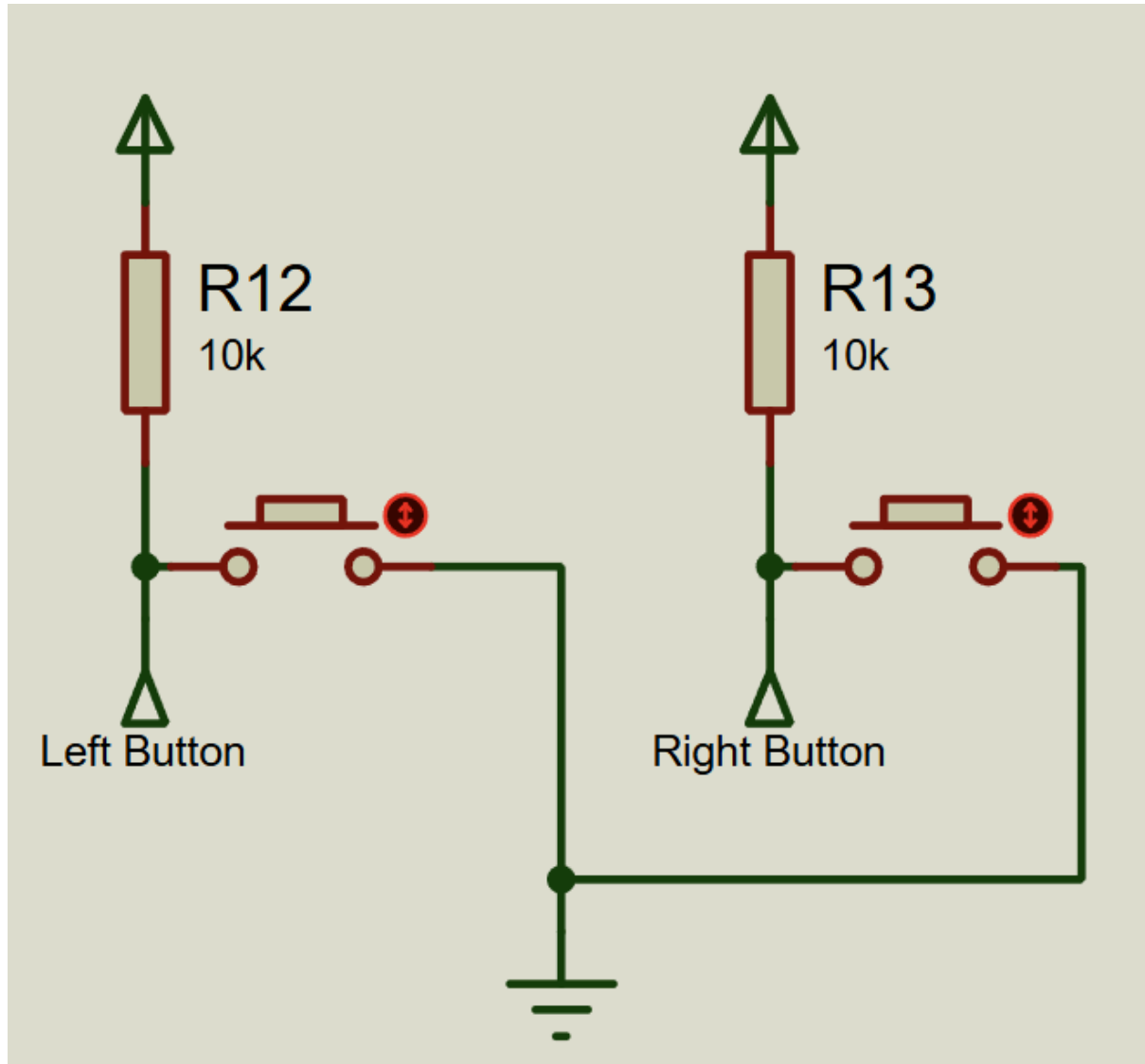


Figure 4: Buttons In Proteus

Display

The seven segment display (**Figure 5**) connects to the PIC microcontroller using 7 pins (B0-B6), each of these are tied to one of the seven individual segments on the display. When a corresponding pin is set low, the segment on the display is lit up. Outputting low causes the segment to light up due to the common anode configuration. [2]

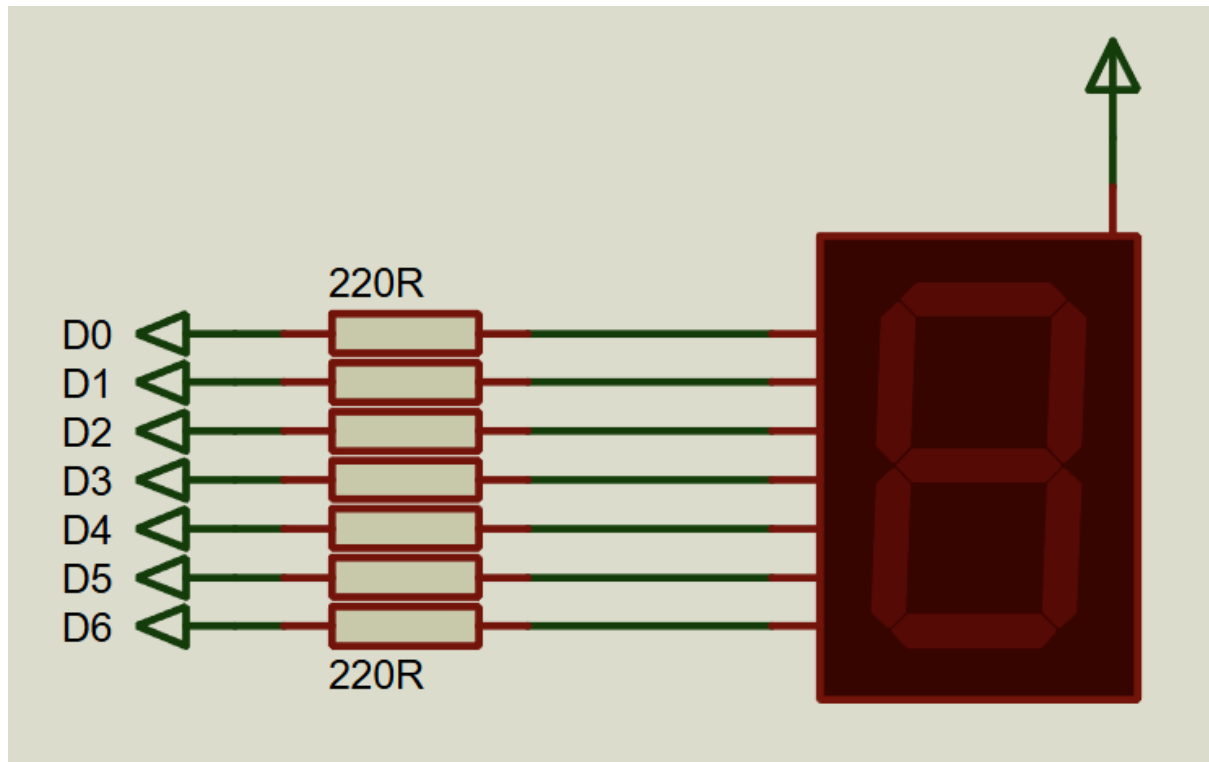


Figure 5: LED Display

Sounder

The piezoelectric sounder (**Figure 6**) can produce sound when an alternating voltage is applied to them, this sound is produced by the deformation and reformation of the material. To achieve this effect, the pin B7 can quickly be set high then low continuously.

For simulation sake, a digital oscilloscope has been attached to the incoming signal to the sounder. This allows monitoring of the signal to ensure any code is operating as expected and to provide a visual demonstration of the signal.

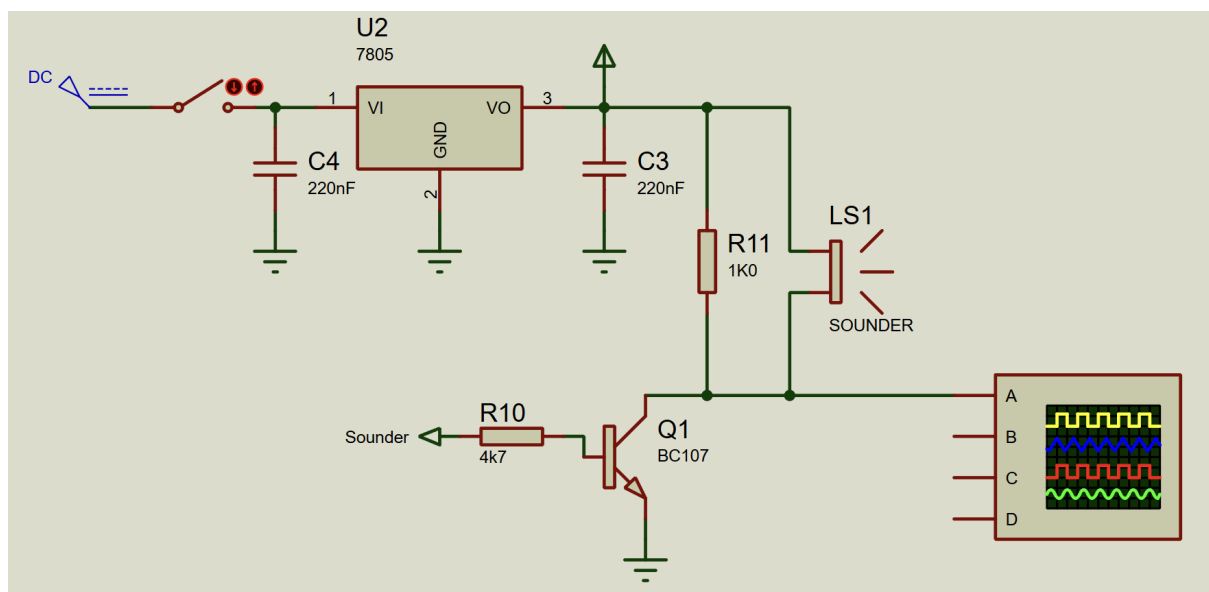


Figure 6: Sounder

Section 2: Printing 0-F On The Display

To print properly to the display, it must be known which pins are tied to which segment. Once this information is known, it will be as easy as visually inspecting the display to decide which pins should be set high (segment off) or low (segment on) depending on what needs to be displayed.

To figure out which pin corresponds to which segment, a quick brute-force approach was taken. The method for this was to output increasing powers of 2 to portB, this sets each pin low in ascending order and shows which pin links to which segment. **Figure 7** shows the results of which pin corresponds to which segment.

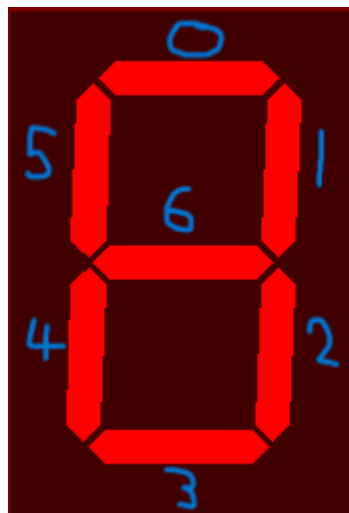


Figure 7: Annotated Seven Segment Display

To figure out what value to output to portB, segments that should be turned off need to be decided, then sum whichever segments we want off at a power of 2. For example to achieve the number "1", where segments 1 and 2 are on:

$$2^0 + 2^3 + 2^4 + 2^5 + 2^6 = 121$$

However, this must be outputted in a hexadecimal format, so 121 must be converted to hexadecimal, giving a value of 0x79 as the final output. A list of 0-F and their corresponding sums and hex values can be seen in **Table 1**. The desired patterns images have been taken from an external source. [3]

Table 1: Seven Segment Display Values

Desired Output	Segment Sum	Hex Value
	$2^6 = 64$	0x40
	$2^0 + 2^3 + 2^4 + 2^5 + 2^6 = 121$	0x79
	$2^2 + 2^5 = 36$	0x24
	$2^4 + 2^5 = 48$	0x30
	$2^0 + 2^3 + 2^4 = 25$	0x19
	$2^1 + 2^4 = 18$	0x12
	$2^1 = 2$	0x02
	$2^3 + 2^4 + 2^5 + 2^6 = 120$	0x78
	$0 = 0$	0x00
	$2^4 = 16$	0x10
	$2^3 = 8$	0x08
	$2^0 + 2^1 = 3$	0x03
	$2^1 + 2^2 + 2^6 = 70$	0x46
	$2^0 + 2^5 = 33$	0x21
	$2^1 + 2^2 = 6$	0x06
	$2^1 + 2^2 + 2^3 = 14$	0x0E

Section 3: Coding Tasks

Task 1: Display 0-F At 0.2s Frequency

This task requires the number 0-9 and letters A-F to be printed on the seven segment display at a 0.2ms frequency. For this, the values found in **Table 1** are used for printing the numbers and letters to the display. The pseudo-code for such a program is:

```
Initialisation code.....

x=0

Loop
  Delay for 200ms
  Print to portB offset x in Values
  Increment x by 1
  Check that x isn't too big, if so reset it

Go back to start of loop

Values x
  0
  1
  2.....
```

The main problem for this program is having a way of pausing for 200ms and printing to the display. Printing to the display has already been explored in **Section 2**, therefore only pausing must be investigated.

In the instruction set documentation it is specified that 1 instruction execution takes 4 clock cycles. Therefore with a 4Mhz clock: [4, p2]

$$\begin{aligned} \text{instructions/s} &= \frac{4,000,000}{4} = 1,000,000 \\ \therefore \text{time/operation} &= \frac{1}{1,000,000} = 1 \times 10^{-6} \text{s} \end{aligned}$$

$$1 \times 10^{-6} \text{s} = 1\mu\text{s}$$

Therefore, there is a $1\mu s$ instruction execution time. Knowing this, the amount of instructions required to generate a desired delay can be calculated. To begin with, how many microseconds are required is calculated:

$$\begin{aligned}1ms &= 1 \times 10^{-3}s \\1\mu s &= 1 \times 10^{-6}s \\1 \times 10^{-3} &= 1 \times 10^{-6} \times 1000\end{aligned}$$

$$\therefore 1ms = 1000\mu s$$

For a desired 200ms delay:

$$200 \times 1000 = 200,000$$

Showing 200,000 instruction cycles are needed in order to achieve a 200ms delay.

A 1ms delay subroutine can be created first, which can then be called 200 times to achieve a 200ms delay. For a 1ms delay, an equivalent of roughly 1000 instruction executions are needed. This is achievable through a simple counter system:

```
delay_1ms
    movlw 0xF9          ; 249

delay_loop
    addlw 0xFF          ; Decrease W by 1
    btfss status, Z     ; Is Z bit set?
    goto delay_loop     ; If not, redo the decrement

    return
```

This takes advantage of two's complement and the *addlw* instruction, by adding the two's complement of 255 it actually adds -1, allowing a decrement. With this subroutine a 1ms delay can be achieved.

To get the desired 200ms delay, this simply needs to be called 200 times:

```
delay_200ms
    movlw 0xC8
    movwf count
delay_1ms
    movlw 0xF9          ; 249

delay_loop
    addlw 0xFF          ; Decrease W by 1
    btfss status, Z     ; Is Z bit set?
    goto delay_loop     ; If not, redo the decrement

    decfsz count, 1
    goto delay_1ms

    return
```

Before completing the final program, a subroutine to provide the required hexadecimal value to be outputted to portB is needed. This can be achieved using a lookup table which uses the program counter to jump to a further part of the code. In this case, it will jump to a *retlw* instruction which stores the hexadecimal value in the working register which can then be used.

```
table addwf PCL
    retlw 0x40 ; 0
    retlw 0x79 ; 1
    retlw 0x24 ; 2
    retlw 0x30 ; 3
    retlw 0x19 ; 4
    retlw 0x12 ; 5
    retlw 0x02 ; 6
    retlw 0x78 ; 7
    retlw 0x00 ; 8
    retlw 0x10 ; 9
    retlw 0x08 ; A
    retlw 0x03 ; B
    retlw 0x46 ; C
    retlw 0x21 ; D
    retlw 0x06 ; E
    retlw 0x0E ; F
```

Finally, a piece of code is needed to prevent going over the limit, to do this the max value can be subtracted from the current value. If they are equal the zero bit will be set and the current value can be set back to 0 if so. For this the following assembly can be used:

```
movlw 0x10          ; Move max value to W
subwf current_value, 0 ; If its at max value, this will be 0
btfsc status, Z      ; Is z bit set? if so do next instruction
clrf current_value    ; Reset current_value if max value reached
```

Putting this all together, the task can be completed. The following source code demonstrates the complete task 1:

```
=====
; DEFINITIONS
=====

#include p16f84a.inc          ; Include register definition file

; The following definitions are for ease of accessing common memory addresses

; Define memory address for portB and trisb
portB equ 0x06
trisb equ 0x06

; Define memory address for portA and trisa
portA equ 0x05
trisa equ 0x05

status equ 0x03              ; Define memory address for status
RP0 equ 0x05                 ; Define bit position for memory bank selection bit
within status

count equ 0x0C
current_value equ 0x0D

=====
; RESET and INTERRUPT VECTORS
=====

; Reset Vector
RST    code 0x0
      goto Start

=====
; CODE SEGMENT
=====

PGM    code
Start
      bsf status,RP0          ; Select memory bank 1

      movlw 0x00
      movwf trisb             ; Set portB as output

      movlw 0x0f
      movwf trisa             ; set portA as input
```

```

        bcf status, RP0                ; Select memory bank 0

        movlw 0xff
        movwf portB                    ; Set portB high to turn off display
        clrf portA                      ; Set portA to 0
        clrf current_value

main_loop

        call delay_200ms

        movf current_value, W          ; Move current_value to W
        call table                      ; Retrieve the print value from the LUT
        movwf portB                    ; Print to display
        incf current_value, 1          ; Increment current_value

        movlw 0x10                     ; Move max value to W
        subwf current_value, 0         ; If its at max value, this will be 0
        btfsc status, Z                ; Is z bit set? if so do next instruction
        clrf current_value             ; Reset current_value if max value reached

        goto main_loop

table addwf PCL
        retlw 0x40 ; 0
        retlw 0x79 ; 1
        retlw 0x24 ; 2
        retlw 0x30 ; 3
        retlw 0x19 ; 4
        retlw 0x12 ; 5
        retlw 0x02 ; 6
        retlw 0x78 ; 7
        retlw 0x00 ; 8
        retlw 0x10 ; 9
        retlw 0x08 ; A
        retlw 0x03 ; B
        retlw 0x46 ; C
        retlw 0x21 ; D
        retlw 0x06 ; E
        retlw 0x0E ; F

delay_200ms
        movlw 0xC8
        movwf count

delay_1ms
        movlw 0xF9                    ; 249

delay_loop
        addlw 0xFF                    ; Decrease W by 1
        btfss status, Z               ; Is Z bit set?
        goto delay_loop               ; If not, redo the decrement

        decfsz count, 1
        goto delay_1ms

        return

;=====
END

```

Task 2: Display “L” or “r”

This task requires the use of the buttons to display either a “L” or an “r” on the seven segment display depending on whether the left or right button is pressed. The buttons are connected to pins A0 and A1, which have been configured as input ports.

To check if the buttons have been pressed, the corresponding bit values can be read to determine whether each button has been pressed or not. The *btfss* (bit test, skip if set) instruction can be used to check bit 0 and bit 1 of the *portA* symbol. If a 1 is read, a subroutine to handle the button press can be called to output to the display. The following pseudo-code demonstrates how this may be achieved:

```
Initialisation code.....
```

```
Loop
```

```
    Check bit 0 in portA
```

```
    If set
```

```
        Display “L”
```

```
    Check bit 1 in portA
```

```
    If set
```

```
        Display “r”
```

```
Go back to start of loop
```

Applying this into assembly:

```
=====
; DEFINITIONS
=====

#include p16f84a.inc           ; Include register definition file

; The following definitions are for ease of accessing common memory addresses

; Define memory address for portB and trisb
portB equ 0x06
trisb equ 0x06

; Define memory address for portA and trisa
portA equ 0x05
trisa equ 0x05

status equ 0x03                ; Define memory address for status
RP0 equ 0x05                   ; Define bit position for memory bank selection bit within status
```

```

;=====
; RESET and INTERRUPT VECTORS
;=====

; Reset Vector
RST    code    0x0
        goto    Start

;=====
; CODE SEGMENT
;=====

PGM     code

Start
        bsf     status,RP0                ; Select memory bank 1

        movlw   0x00
        movwf   trisb                    ; Set portB as output

        movlw   0x0f
        movwf   trisa                    ; set portA as input

        bcf     status, RP0              ; Select memory bank 0

        clrf    portB                    ; Set portB to 0
        clrf    portA                    ; Set portA to 0

main_loop

        ; Check if left button is pressed, if so, display an "L"
        btfss   portA, 0
        call    handle_button_press_L

        ; Check if right button is pressed, if so, display an "r"
        btfss   portA, 1
        call    handle_button_press_R

        goto    main_loop                ; Loop again

handle_button_press_L
        movlw   0                        ; 0 Corresponds to "L" in the lookup table
        call    lookup_table
        movwf   portB                    ; Print "L"
        return

handle_button_press_R
        movlw   1                        ; 0 Corresponds to "L" in the lookup table
        call    lookup_table
        movwf   portB                    ; Print "r"
        return

; Will return the offset stored in W
lookup_table addwf PCL
        retlw   0x47 ; L
        retlw   0x2f ; r

;=====
END

```

Task 3: All On or All Off

This task requires pretty much the exact same process as the last one, except instead of printing "L" or "r", the display is either fully on or fully off. The values required for this is simply **0xFF** (255) for all off, or **0x00** (0) for all on. The pseudo-code for such a program is:

```
Initialisation code.....

Loop

    Check bit 0 in portA
    If set
        Turn off display

    Check bit 1 in portA
    If set
        Turn on all segments on the display

Go back to start of loop
```

Applying this into assembly:

```
=====
; DEFINITIONS
=====

#include p16f84a.inc           ; Include register definition file

; The following definitions are for ease of accessing common memory addresses

; Define memory address for portB and trisb
portB equ 0x06
trisb equ 0x06

; Define memory address for portA and trisa
portA equ 0x05
trisa equ 0x05

status equ 0x03                ; Define memory address for status
RP0 equ 0x05                    ; Define bit position for memory bank selection bit within status

=====
; RESET and INTERRUPT VECTORS
=====

; Reset Vector
RST    code    0x0
        goto   Start
```

```

;=====
; CODE SEGMENT
;=====

PGM    code

Start
    bsf status,RP0                ; Select memory bank 1

    movlw 0x00
    movwf trisb                    ; Set portB as output

    movlw 0x0f
    movwf trisa                    ; set portA as input

    bcf status, RP0                ; Select memory bank 0

    clrf portB                     ; Set portB to 0
    clrf portA                     ; Set portA to 0

main_loop

    ; Check if left button is pressed, if so, turn off all segments
    btfss portA, 0
    call handle_button_press_L

    ; Check if right button is pressed, if so, light all segments
    btfss portA, 1
    call handle_button_press_R

    goto main_loop                ; Loop again

handle_button_press_L
    movlw 0                        ; 0 Corresponds to all off
    call lookup_table
    movwf portB                    ; All off
    return

handle_button_press_R
    movlw 1                        ; 0 Corresponds to all on
    call lookup_table
    movwf portB                    ; All On
    return

; Will return the offset stored in W
lookup_table addwf PCL
    retlw 0xFF ; All off
    retlw 0x00 ; All on

;=====
END

```

Task 4: Display 0-5-9

This task ties in problems that were solved in **Task 2** (handling button presses) and **Task 1** (display values in ascending order and upper bounds checking). The solution for each can be derived for this task. However, this task requires an additional level of complication, the value printed on the display is controlled by either button, going up to 9 or down to 0. The pseudo-code for this program is:

```
Initialisation code.....
```

```
x=5
```

```
Loop
```

```
    Print to portB offset x in Values
```

```
    Check if left button pressed
```

```
        If it is, decrement x by 1
```

```
        Check that x isn't too small, if it is clamp it to 0
```

```
    Check if right button is pressed
```

```
        If it is, increment x by 1
```

```
        Check that x isn't too big, if it is clamp it to 9
```

```
Go back to start of loop
```

```
Values x
```

```
    0
```

```
    1
```

```
    2.....
```

The challenge introduced here to be solved is the problem of lower bounds checking as an upper bound check has already been found previously. This can be done using only the *decfsz* instruction, as if we decrement to zero we can set our current value back to 1. This does require an extra value in the lookup table, which can also be used to indicate if something has gone wrong. The code for such a check is:

```
handle_button_press_L
    decfsz current_value, 1 ;Decrement current_value, continue if its not zero afterwards
    goto decrement_ok
    movlw 0x01                ; If current_value is zero, set it back to 1
    movwf current_value
```

```

decrement_ok
    movf current_value, W           ; Retrieve current value to display
    call lookup_table
    movwf portB

    call delay_250ms               ; Delays for button pressing
    call delay_250ms
    return

```

Pulling this all together provides the following final code:

```

;=====
; DEFINITIONS
;=====

#include p16f84a.inc           ; Include register definition file

; The following definitions are for ease of accessing common memory addresses

; Define memory address for portB and trisb
portB equ 0x06
trisb equ 0x06

; Define memory address for portA and trisa
portA equ 0x05
trisa equ 0x05

status equ 0x03               ; Define memory address for status
RP0 equ 0x05                  ; Define bit position for memory bank selection bit
within status

count equ 0x0C
current_value equ 0x0D

;=====
; RESET and INTERRUPT VECTORS
;=====

    ; Reset Vector
RST    code    0x0
        goto   Start

;=====
; CODE SEGMENT
;=====

PGM    code
Start
    bsf status,RP0           ; Select memory bank 1

    movlw 0x00
    movwf trisb              ; Set portB as output

    movlw 0x0f
    movwf trisa              ; set portA as input

    bcf status, RP0          ; Select memory bank 0

    movlw 0xff
    movwf portB              ; Set portB high to turn off display

```

```

        clrf portA                ; Set portA to 0

        movlw 6                  ; 6 as this has an extra value in the lookup_table
        movwf current_value

        call lookup_table
        movwf portB              ; All On

main_loop
        ; Check if left button is pressed, if so, turn off all segments
        btfss portA, 0
        call handle_button_press_L

        ; Check if right button is pressed, if so, light all segments
        btfss portA, 1
        call handle_button_press_R

        goto main_loop          ; Loop again

handle_button_press_L
        decfsz current_value, 1    ; Decrement current_value, continue if its not zero
afterwards
        goto decrement_ok

        movlw 0x01                ; If current_value is zero, set it back to 1
        movwf current_value

decrement_ok
        movf current_value, W      ; Retrieve current value to display
        call lookup_table
        movwf portB

        call delay_250ms          ; Delays for button pressing
        call delay_250ms
        return

handle_button_press_R

        call check_upper_bound     ; Subroutine to check its not too high
        movf current_value, W      ; Move the current value to W for printing to display

        call lookup_table          ; Get the value
        movwf portB               ; Print to display

        call delay_250ms          ; Delays for button pressing
        call delay_250ms
        return

check_upper_bound
        movf current_value, W      ; Move current_value to W
        sublw 0x0a                ; Sub LUT_MAX from w (negatives are non-zero)
        btfss status, Z           ; If the value is not 0, increment current_value
        incf current_value
        return

lookup_table addwf PCL
        retlw 0x100 ; THIS CASE SHOULD NOT GET MET
        retlw 0x40 ; 0
        retlw 0x79 ; 1
        retlw 0x24 ; 2
        retlw 0x30 ; 3

```

```

    retlw 0x19 ; 4
    retlw 0x12 ; 5
    retlw 0x02 ; 6
    retlw 0x78 ; 7
    retlw 0x00 ; 8
    retlw 0x10 ; 9

delay_250ms
    movlw 0xFF
    movwf count
delay_1ms
    movlw 0xF9          ; 249

delay_loop
    addlw 0xFF          ; Decrease W by 1
    btfss status, Z     ; Is Z bit set?
    goto delay_loop     ; If not, redo the decrement

    decfsz count, 1
    goto delay_1ms

    return

;=====
END

```

Task 5: Button Sound Producer

This task utilises the piezoelectric sounder as well as button holding. Piezoelectric sounders produce sound when an alternating voltage is applied to them. To produce this alternating effect, the pin the sounder is connected to needs to set high and low in a quick, alternating fashion. For this program the pseudo-code would look as such: [5]

```
Initialisation code.....

Loop
    Print to portB offset x in Values

    Check if a button is pressed
        Output 0 to portB
        Delay for a short period of time

        Output 0x80 to portB
        Delay for a short period of time

    Go back to start of loop
```

This task can derive and modify the delay subroutine for a higher frequency to produce a stable sound, the button handling can also be derived from earlier. The code for a subroutine that is called when either button is pressed that produces an alternating voltage is as follows:

```
handle_button_press
    movlw 0                ; Turn pin off
    movwf portB
    call delay_15ms        ; Delay for 15ms

    movlw 0x80             ; Turn pin on
    movwf portB
    call delay_15ms        ; Delay for 15ms
    return
```

This will produce an oscillation of high to low voltage, which will cause the piezo sounder to produce sound. The reading on the digital oscilloscope can be seen in **Figure 8**, this shows the oscillation from high to low is being sent to the sounder correctly:

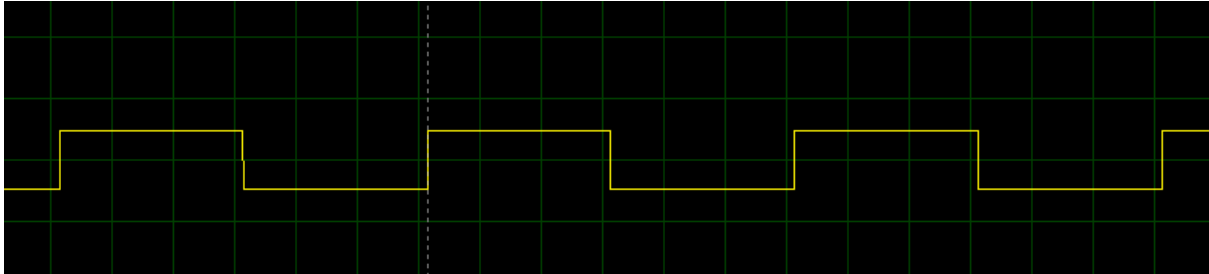


Figure 8: Task 4 Oscilloscope

Assembly for the entire program:

```

;=====
; DEFINITIONS
;=====

#include p16f84a.inc                ; Include register definition file

; The following definitions are for ease of accessing common memory addresses

; Define memory address for portB and trisb
portB equ 0x06
trisb equ 0x06

; Define memory address for portA and trisa
portA equ 0x05
trisa equ 0x05

status equ 0x03                    ; Define memory address for status
RP0 equ 0x05                        ; Define bit position for memory bank selection bit within status

count equ 0x0C

;=====
; RESET and INTERRUPT VECTORS
;=====

; Reset Vector
RST    code 0x0
        goto Start

;=====
; CODE SEGMENT
;=====

PGM    code

Start
        bsf status,RP0                ; Select memory bank 1

```

```

    movlw 0x00
    movwf trisb                ; Set portB as output

    movlw 0x0f
    movwf trisa                ; set portA as input

    bcf status, RP0           ; Select memory bank 0

    movlw 0xff
    movwf portB                ; Set portB high to turn off display
    clrf portA                ; Set portA to 0
    clrf count

main_loop

    ; Check if left button is pressed, if so, make a sound
    btfss portA, 0
    call handle_button_press

    ; Check if right button is pressed, if so, make a sound
    btfss portA, 1
    call handle_button_press

    goto main_loop            ; Loop again

handle_button_press
    movlw 0                    ; Turn pin off
    movwf portB
    call delay_15ms           ; Delay for 15ms

    movlw 0x80
    movwf portB                ; Turn pin on
    call delay_15ms           ; Delay for 15ms
    return

delay_15ms
    movlw 0xf
    movwf count
delay_1ms
    movlw 0xF9                ; 249

delay_loop
    addlw 0xFF                ; Decrease W by 1
    btfss status, Z           ; Is Z bit set?
    goto delay_loop           ; If not, redo the decrement

    decfsz count, 1
    goto delay_1ms

    return

;=====
END

```

Task 6: Synthesiser PWM

This task requires the wave to the sounder to produce a fixed frequency PWM signal and where the pulse width changes over time. This effect can be seen in **Figure 9**, which shows that despite the pulses width changing, the time between pulses remains the same. [6]

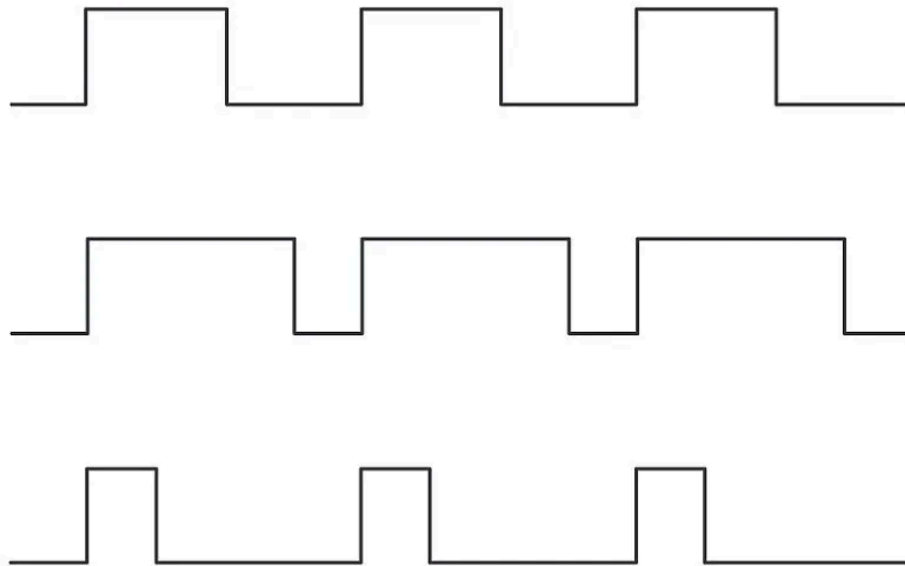


Figure 9: PWM Signal [6]

For this program, the pulse width will increase until it reaches a maximum, then it will decrease until it reaches a minimum.

To accomplish this two delays can be used that will increment / decrement in an opposing way as the time spent high is inversely proportional to the time spent low. A direction variable can also be used to check whether the pulse width should be increasing or decreasing. Additionally, a maximum and minimum delay variable can be used to make the assembly easier to read.

The pseudo-code for this looks like:

```
Initialisation code.....

Current_direction = 1

Max_delay = 248
Min_delay = 2

High_delay = 125
Low_delay = 125

Loop
    Output high to the sounder
    Delay for the value of High_delay

    Output low to the sounder
    Delay for the value of Low_delay

    If current_direction is increase high
        Increase high delay
        Decrease low delay

    If current_direction is decrease high
        Decrease high delay
        Increase low delay

    If high delay is at maximum or minimum
        Change direction to 1 or 0 depending

Go back to start of loop
```

Before anything a few key variables need to be made and initialised. These variables control the delay period that the sounder will be high / low for, the maximum value of the delays for easier code readability and a current direction variable to control whether the pulse width should increase or decrease:

```
high_delay equ 0x0D
low_delay equ 0x0E

delay_max equ 0x0F
delay_min equ 0x10
current_direction equ 0x11
```

```

.....
Start
    movlw 0x7D                ; 125
    movwf high_delay
    movwf low_delay

    movlw 0xF8
    movwf delay_max

                                ; 125 is midpoint
    movlw 0x2
    movwf delay_min

    movlw 0x1
    movwf current_direction
.....

```

Outputting to the sounder in the previous task used a fixed delay of 15ms, however this task requires a dynamic delay subroutine. To achieve this, the desired delay can simply be moved to the working register beforehand and this allows the following code to use it:

```

delay_custom_ms
    movwf count                ; Move desired delay period to W beforehand
delay_us
    movlw 0x19                ; roughly 100us
delay_loop
    addlw 0xFF                ; Decrease W by 1
    btfss status, Z           ; Is Z bit set?
    goto delay_loop           ; If not, redo the decrement
    decfsz count, 1
    goto delay_us
    return

```

Now an oscillating voltage can be produced with custom delays. The method to output to the sounder is the same as the previous task, instead this time the *high_delay* and *low_delay* variables will be used, as seen in the following code:

```

    movlw 0xff
    movwf portB                ; Output high to sounder
    movf high_delay, W
    call delay_custom_ms       ; Delay for the high part

    movlw 0x00
    movwf portB                ; Output low to sounder
    movf low_delay, W
    call delay_custom_ms       ; Delay for the low part

```

Next, to check whether the pulse width should be increasing or decreasing, the *current_direction* variable can be checked whether it is 1 or 0. By knowing which value it is, the direction can be determined. The *btfss* (bit check file, skip if set) and *btfsc* (bit check file, skip if clear) instructions can be used to determine whether the value is 1 or 0, and thus if the pulse width should increase or decrease:

```
    ; Check if direction is increasing or decreasing the high delay
    btfsc current_direction, 0
    call decrease_high_delay
    btfss current_direction, 0
    call increase_high_delay
```

The *decrease_high_delay* and *increase_high_delay* are subroutines which will increment and decrement each delay whilst using bound checking to ensure the value does not go over or under the maximum value.

```
decrease_high_delay
    ; Check if low delay is at max, if not increase it
    movf low_delay, W
    subwf delay_max, 0
    btfss status, Z
    incf low_delay

    ; Check if high delay is at min, if not decrease it
    movf delay_min, W
    subwf high_delay, 0
    btfss status, Z
    decf high_delay

    return

increase_high_delay
    ; Check if high delay is at max, if not increase it
    movf high_delay, W
    subwf delay_max, 0
    btfss status, Z
    incf high_delay

    ; Check if high delay is at min, if not decrease it
    movf delay_min, W
    subwf low_delay, 0
    btfss status, Z
    decf low_delay

    return
```

Finally, after incrementing the delay the value of one of the two delays should be checked if it is at a maximum, if so then the direction should be flipped. This is not done in the *increase_high_delay* and *decrease_high_delay* functions as there is a risk that the high_delay and low_delay would stray away from each other if they are not both completed each time the subroutine is called. To check the direction, the same bounds check as before can be used:

```
check_direction
    ; If high is at the maximum value, change direction
    movf high_delay, W
    subwf delay_max, 0
    btfsc status, Z
    incf current_direction, 1

    ; If high is at the minimum value, change direction
    movf delay_min, W
    subwf high_delay, 0
    btfsc status, Z
    clrf current_direction

    return
```

The complete main loop operates like the pseudo-code described and uses each of the subroutines that were just defined, which ends up looking like this in assembly:

```
main_loop

    movlw 0xff
    movwf portB                ; Output high to sounder
    movf high_delay, W
    call delay_custom_ms        ; Delay for the high part

    movlw 0x00
    movwf portB                ; Output low to sounder
    movf low_delay, W
    call delay_custom_ms        ; Delay for the low part

    ; Check if direction is increasing or decreasing the high delay
    btfsc current_direction, 0
    call decrease_high_delay
    btfss current_direction, 0
    call increase_high_delay

    call check_direction        ; Check if max or min delay has been reached

    goto main_loop             ; Loop again
```

When using the following code, the pulse width will start at a middle value, then the pulse width will increase until it reaches its maximum value and then decrease back down to the minimum. This successfully completes the task, and each the middle, maximum and minimum pulse width read on the oscilloscope can be seen in **Figure 10**, **Figure 11**, and **Figure 12**.

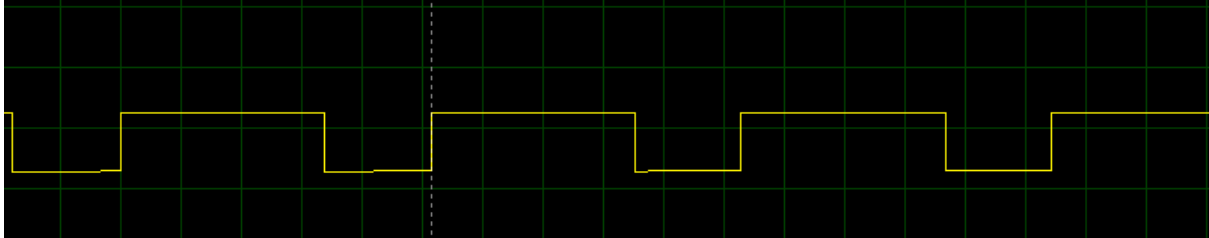


Figure 10: Middle Pulse Width

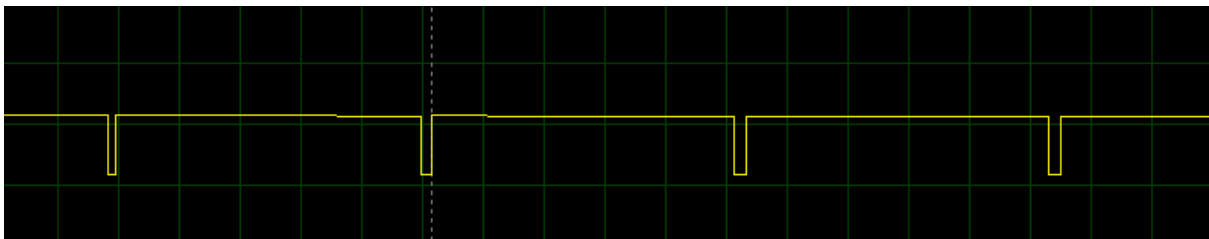


Figure 11: Max Pulse Width

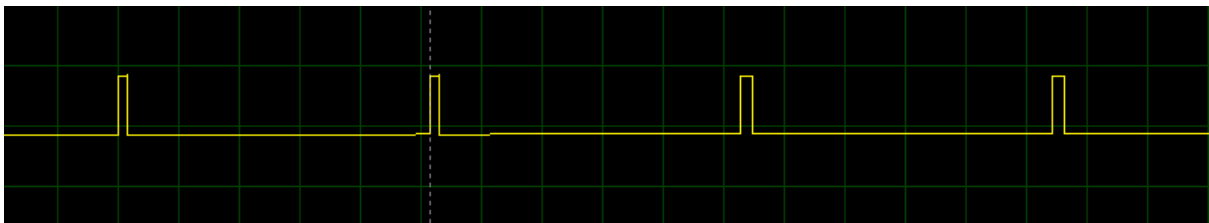


Figure 12: Minimum Pulse width

The complete assembly code for this task is as follows:

```
=====
; DEFINITIONS
=====

#include p16f84a.inc          ; Include register definition file

; The following definitions are for ease of accessing common memory addresses

; Define memory address for portB and trisb
portB equ 0x06
trisb equ 0x06

; Define memory address for portA and trisa
portA equ 0x05
trisa equ 0x05
```

```

status equ 0x03                ; Define memory address for status
RP0 equ 0x05                   ; Define bit position for memory bank selection bit
within status

count equ 0x0C
high_delay equ 0x0D
low_delay equ 0x0E

delay_max equ 0x0F
delay_min equ 0x10
current_direction equ 0x11

;=====
; RESET and INTERRUPT VECTORS
;=====

; Reset Vector
RST    code 0x0
      goto Start

;=====
; CODE SEGMENT
;=====

PGM    code

Start
      bsf status,RP0            ; Select memory bank 1

      movlw 0x00
      movwf trisb               ; Set portB as output

      movlw 0x0f
      movwf trisa               ; set portA as input

      bcf status, RP0          ; Select memory bank 0

      movlw 0xff
      movwf portB               ; Set portB high to turn off display
      clrf portA                ; Set portA to 0
      clrf count

      movlw 0x7D                ; 125
      movwf high_delay
      movwf low_delay

      movlw 0xF8
      movwf delay_max
      ; 125 is midpoint
      movlw 0x2
      movwf delay_min

      movlw 0x1
      movwf current_direction

main_loop
      movlw 0xff
      movwf portB               ; Output high to sounder
      movf high_delay, W
      call delay_custom_ms      ; Delay for the high part

      movlw 0x00
      movwf portB               ; Output low to sounder

```

```

movf low_delay, W
call delay_custom_ms          ; Delay for the low part

; Check if direction is increasing or decreasing the high delay
btfsc current_direction, 0
call decrease_high_delay
btfss current_direction, 0
call increase_high_delay

call check_direction          ; Check if max or min delay has been reached

goto main_loop                ; Loop again

```

check_direction

```

; If high is at the maximum value, change direction
movf high_delay, W
subwf delay_max, 0
btfsc status, Z
incf current_direction, 1

; If high is at the minimum value, change direction
movf delay_min, W
subwf high_delay, 0
btfsc status, Z
clrf current_direction

return

```

decrease_high_delay

```

; Check if low delay is at max, if not increase it
movf low_delay, W
subwf delay_max, 0
btfss status, Z
incf low_delay

; Check if high delay is at min, if not decrease it
movf delay_min, W
subwf high_delay, 0
btfss status, Z
decf high_delay

return

```

increase_high_delay

```

; Check if high delay is at max, if not increase it
movf high_delay, W
subwf delay_max, 0
btfss status, Z
incf high_delay

; Check if high delay is at min, if not decrease it
movf delay_min, W
subwf low_delay, 0
btfss status, Z
decf low_delay

return

```

```

delay_custom_ms
    movwf count          ; Move desire delay period to W beforehand
delay_us
    movlw 0x19           ; roughly 100us

delay_loop
    addlw 0xFF           ; Decrease W by 1
    btfss status, Z      ; Is Z bit set?
    goto delay_loop      ; If not, redo the decrement

    decfsz count, 1
    goto delay_us

    return

;=====
END

```

References

- [1] Microchip, "PIC16F84A Data Sheet"
<https://ww1.microchip.com/downloads/en/devicedoc/35007b.pdf>
(accessed 12th January, 2025)
- [2] Electronics Tutorials, "7 Segment Display"
<https://www.electronics-tutorials.ws/blog/7-segment-display-tutorial.html> (accessed January 12, 2025)
- [3] Wikipedia, "Seven Segment Display"
https://en.wikipedia.org/wiki/Seven-segment_display (accessed 12th January, 2025)
- [4] Microchip, "Section 29. Instruction Set"
<https://ww1.microchip.com/downloads/en/DeviceDoc/31029a.pdf>
(accessed January 12, 2025)
- [5] RS Online, "How Do Piezo Buzzers Work?"
<https://uk.rs-online.com/web/content/discovery/ideas-and-advice/how-do-piezo-buzzers-work> (accessed January 12, 2025)
- [6] CircuitBread, "What is a PWM signal?"
<https://www.circuitbread.com/ee-faq/what-is-a-pwm-signal> (accessed January 13, 2025)