

Sam Harries - Assignment 2

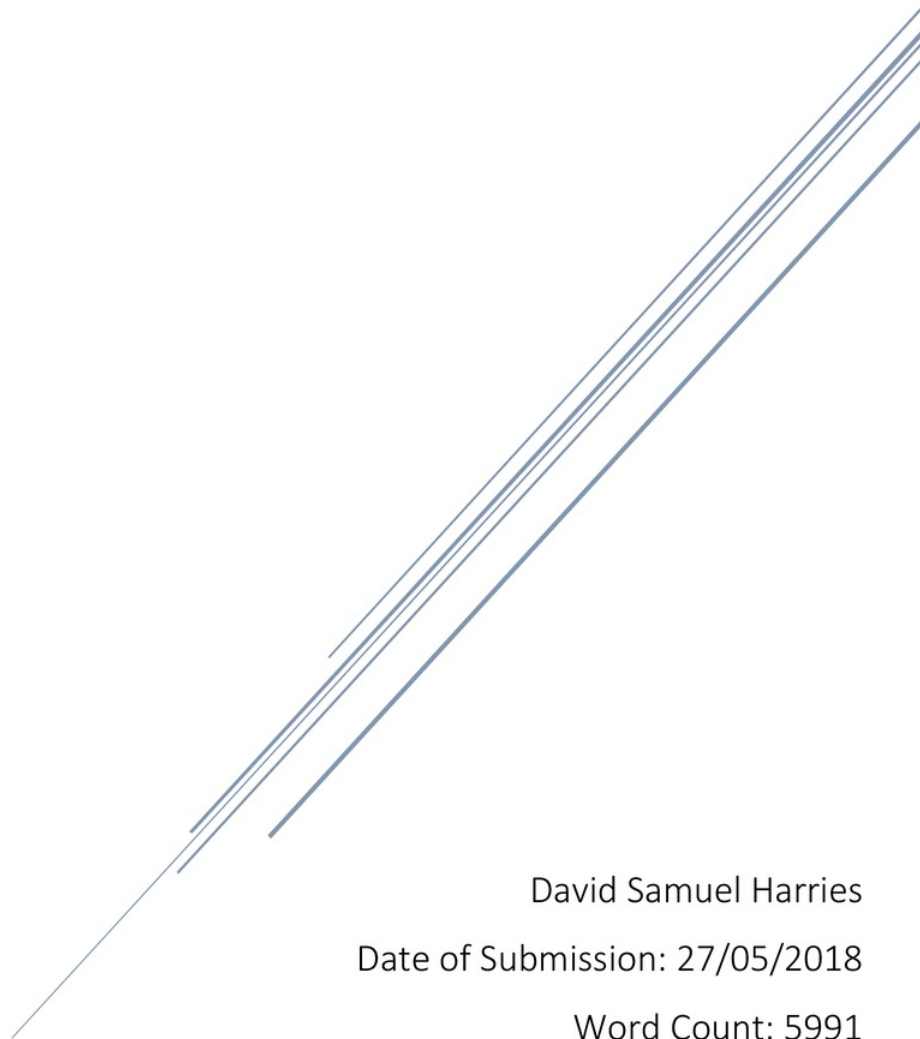
by David Harries

FILE	7309_DAVID_HARRIES_SAM_HARRIES_-_ASSIGNMENT_2_145116_1794478548.DOCX		
TIME SUBMITTED	27-MAY-2018 10:29PM (UTC+0100)	WORD COUNT	5991
SUBMISSION ID	89979658	CHARACTER COUNT	41983

HND Electrical and Electronic Engineering

PSDE5013

Assignment 2: DC Motor Controller



David Samuel Harries

Date of Submission: 27/05/2018

Word Count: 5991

TABLE OF CONTENTS

1	Introduction	5
2	System Hardware	6
2.1	Microcontroller	6
2.2	Keypad.....	7
2.3	LCD Screen	7
2.4	DC motor	8
2.5	Start / Stop Pushbuttons.....	8
2.6	Closed Loop Feedback.....	9
2.7	Development Board	9
3	System Design and Configuration	11
3.1	Timer 0	11
3.1.1	OPTION_REG	12
3.1.2	INTCON_REG	13
3.2	Timer 1	14
3.3	Timer 2	14
3.4	PWM	17
3.5	Ports.....	18
3.6	Other Registers	19
3.6.1	ADCON1	19
3.6.2	PIE1	20
3.6.3	PIE2	21
3.6.4	PIR1	22
3.6.5	PIR2	23
3.6.6	T1CON	24
3.6.7	Timer 2 Control Registers.....	25
3.6.8	CCP1CON	26
3.6.9	PWM and T2 Registers	27
4	Software Design and Explanation	28
4.1	Header Files, Function Prototypes and Global Variables.....	28
4.2	Interrupt Service Routine.....	28
4.3	Main Program	29
4.3.1	Configurations	29
4.3.2	While Super Loop.....	30
4.4	DisplaySpeed Function	33

4.5	DisplayValues Function	34
4.6	ScanKeypad Function	35
4.7	Delay Function	36
5	Testing and Analysis of Results	37
5.1	Simulated Test.....	37
5.1.1	Simulated Test 1 (100 rpm setpoint)	38
5.1.2	Simulated Test 2 (200 rpm setpoint)	38
5.1.3	Simulated Test 3 (300 rpm setpoint)	39
5.1.4	Simulated Test 4 (360 rpm setpoint)	39
5.2	Table of Simulated Results.....	40
5.3	Practical Test	40
5.3.1	Practical Test 1 (200 rpm setpoint)	42
5.3.2	Practical Test 2 (1000 rpm setpoint).....	43
5.3.3	Practical Test 3 (2000 rpm setpoint).....	44
5.3.4	Practical Test 4 (2350 rpm setpoint).....	45
5.4	Practical Test Summary.....	46
6	Discussion and Conclusions	47
7	Team Working.....	48
8	Appendices.....	49

List of Figures

Figure 1-1	5
Figure 2-1	6
Figure 2-2	7
Figure 2-3	7
Figure 2-4	8
Figure 2-5	9
Figure 2-6	9
Figure 3-1	11
Figure 3-2	12
Figure 3-3	13
Figure 3-4	14
Figure 3-5	15
Figure 3-6	15
Figure 3-7	16
Figure 3-8	17
Figure 3-9	17
Figure 3-10	18
Figure 3-11	19
Figure 3-12	20

Figure 3-13	21
Figure 3-14	22
Figure 3-15	23
Figure 3-16	24
Figure 3-17	25
Figure 3-18	25
Figure 3-19	26
Figure 3-20	27
Figure 4-1	28
Figure 4-2	29
Figure 4-3	30
Figure 4-4	31
Figure 4-5	33
Figure 4-6	34
Figure 4-7	36
Figure 5-1	37
Figure 5-2	38
Figure 5-3	38
Figure 5-4	39
Figure 5-5	39
Figure 5-6	40
Figure 5-7	41
Figure 5-8	41
Figure 5-9	42
Figure 5-10	42
Figure 5-11	43
Figure 5-12	43
Figure 5-13	44
Figure 5-14	44
Figure 5-15	45
Figure 5-16]	45
Figure 5-17	46

1 INTRODUCTION

This assignment will show the design of an embedded system using the configurable on-chip peripherals using a suitable embedded programming language such as C. The design must suit the user needs and fit the desired purpose. This is detailed in the system specification provided in the assignment brief.

An understanding of the problems associated with interfacing to external devices must be displayed. Solutions for these problems must also be provided and displayed in a practical system.

The system specification states that a microcontroller must be interfaced to a DC motor, incremental encoder, analogue sensing circuit, keypad, LCD display and two pushbuttons. Using the on-chip peripherals a PWM speed control system must be implemented to control the speed of the motor using speed feedback. A set-point for the speed must be able to be input via the keypad, speed feedback provided by the encoder and a PI controller must use this feedback to regulate the speed so that it meets the set-point. The LCD display must show the set-point, speed of the motor and the error deduced by the PI controller. The motor must be started / stopped through the use of the two pushbuttons. This system can be seen in **Figure 1-1**.

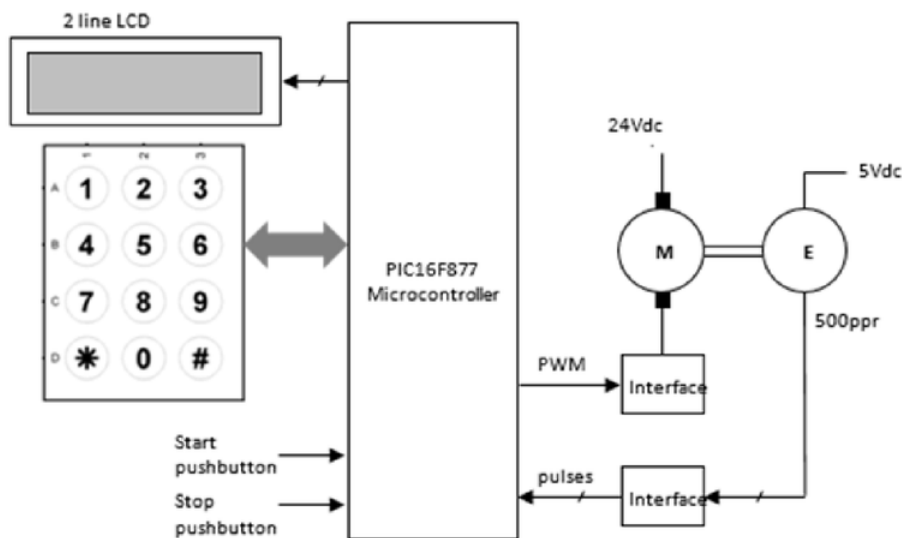


Figure 1-1

Once the system has been designed, it will be tested in a simulation to see if the program initially works. Once the simulation is proven to work, the system will be practically tested. All tests will be documented and can be found in **Section 5**. After testing, conclusions and recommendations will be made which can be found in **Section 6**.

*Good introduction.
The development
method using
prototyping should
have been included
here.*

2 SYSTEM HARDWARE

2.1 MICROCONTROLLER

The core element of this system is the microcontroller which is used to process all the information provided by the keypad, pushbuttons and encoder to control the DC motor. Therefore, selecting the right microcontroller for this system is important to achieve the requirements of the specification.

The PIC16F877 microcontroller is a suitable choice for this system as it has PWM (Pulse Width Modulation) capabilities, plenty of general purpose inputs and outputs, and the on-chip peripherals that will be required for this system such as timers.

The pinout diagram for the PIC16F877 microcontroller can be seen below and was obtained from the chip's datasheet.

40-Pin PDIP

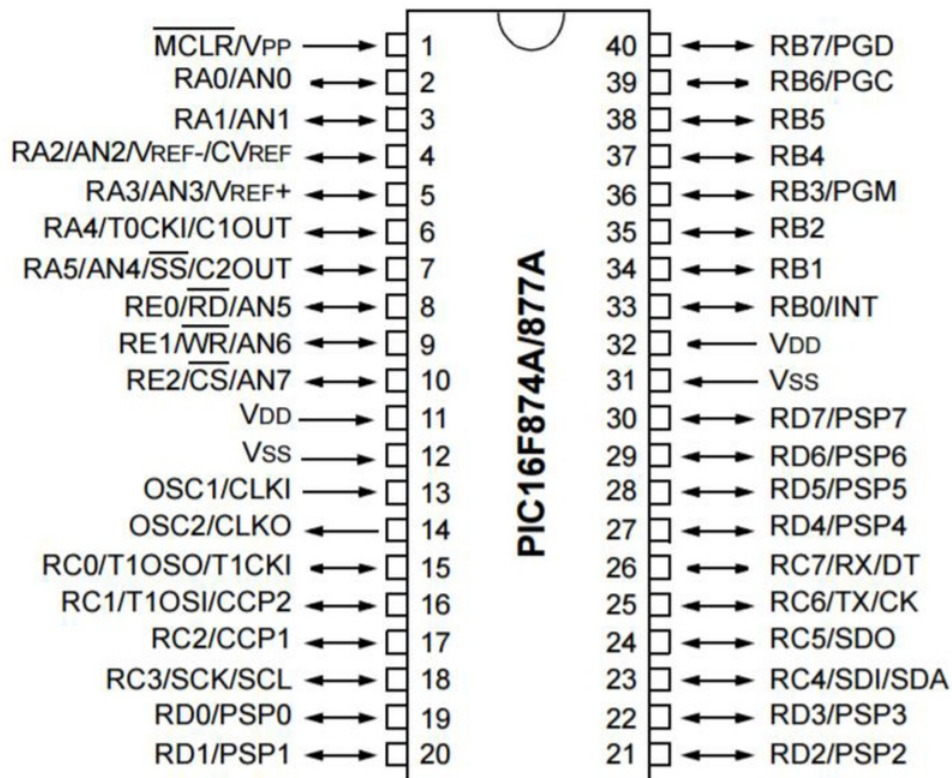


Figure 2-1

(PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)

2.2 KEYPAD

The keypad element of the system allows the user to input the set-point for the motor's speed so that the PI control algorithm can increase or decrease the output of the microcontroller. In order for this to be done a 12 button keypad must be interfaced to the microcontroller. This is done by multiplexing as the keypad being used is a 3x4 matrix input. This input requires the same pin on the microcontroller to share multiple button inputs on the keypad. Therefore, the inputs must be processed by a multiplex function within the microcontroller. An example setup of the keypad interface can be seen in **Figure 2-2**.

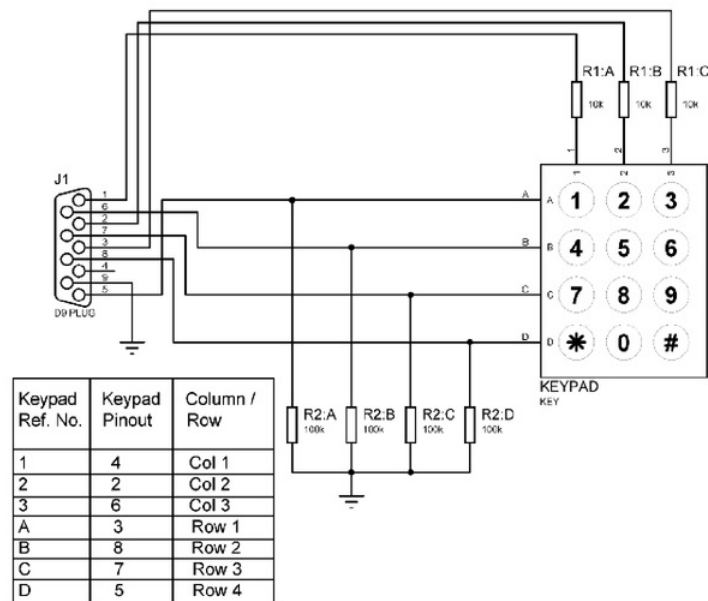


Figure 2-2

2.3 LCD SCREEN

According to the specification an LCD screen is required to display the set-point, current motor speed and the error. The LCD display will have 2 lines each 16 characters long. This will be enough room to fit all the required parameters.



Figure 2-3

The LCD will be connected to the micro controller using RB0 to RB5. In addition to these port B outputs, the LCD must be connected to a switch in the ON position to be operational.

2.4 DC MOTOR

The DC motor is the subject of the system and will be controlled by the microcontroller. The chosen motor will have an operating voltage of 24VDC. However, the PIC16F877 only has an output voltage of 5VDC. In order to allow a safe interaction between the motor and the microcontroller, the two voltages must be isolated through the use of a junction.

The motor's speed is going to be controlled by pulse width modulation (PWM). A MOSFET transistor will be used to determine the duty cycle of the motor by having the 5VDC output from the microcontroller activate the MOSFET's gate. When the MOSFET is active, the 24VDC will power the motor, and when it is not active, the 24VDC will power the motor. This junction keeps the 5VDC and the 24VDC voltages isolated from each other while allowing the microcontroller full control over the motor's speed. A flywheel diode will be used to dissipate the back EMF produced by the motor. This configuration can be seen in **Figure 2-4**.

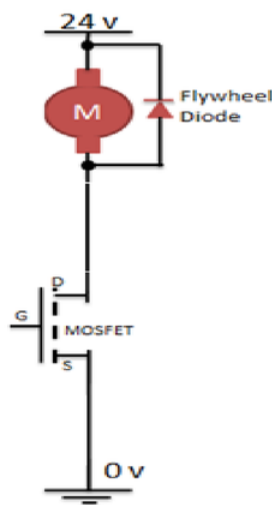


Figure 2-4

2.5 START / STOP PUSHBUTTONS

The pushbuttons in the specification are said to be used to start and stop the motor. These will be connected to port A of the controller and will be sourcing voltage. Port A will be configured as an input, meaning when one of the buttons is pressed, the pin needs to be at a logic HIGH or 1. When one of the buttons is pressed, their respective pin is set to a logic HIGH as the voltage can flow to it. When the button is released, the pin goes back to a logic LOW and any floating voltage is taken to ground through a set of pull-down resistors.

2.6 CLOSED LOOP FEEDBACK

The specification states that the motor must be controlled by a PI controller. In order for this to be done, speed feedback is required. An incremental encoder will be used to measure and send data to the controller. An encoder with an output of 500 PPR (Pulses per revolution) is selected.

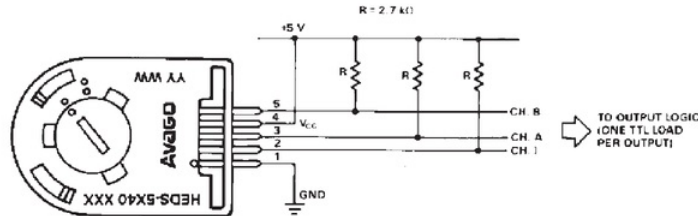


Figure 2-5

(HEDM-55xx/560x & HEDS-55xx/56xx Quick Assembly Two and Three Channel Optical Encoders, 2014)

Figure 2-5 shows a pin-out diagram for the encoder found from the datasheet. It shows how the encoder should be connected to Vcc, and the output channels. The outputs act as a voltage sink and have pull-up resistors connected to pins 2,3 and 5 to Vcc.

2.7 DEVELOPMENT BOARD

The development board is a key aspect of the design and testing of the initial system. The model of the development board to be used in this system is the HP488. The use of a development board allows for easy configuration and changes to the design, along with reusability for future developments to this system, or other projects.

Board layout

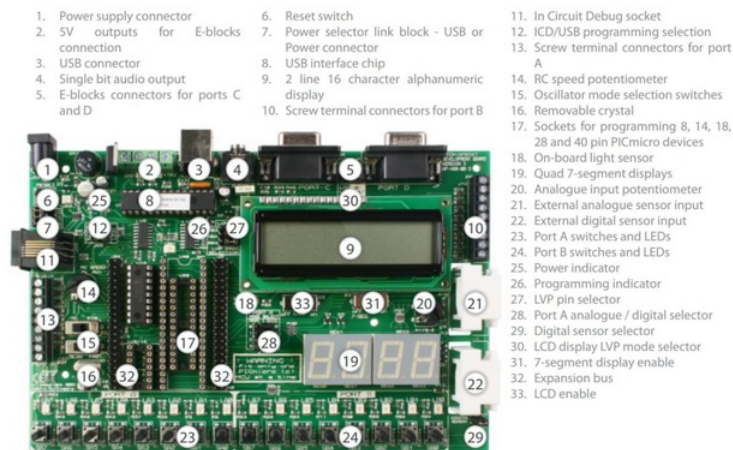


Figure 2-6

Figure 2-6 shows the HP488 along with descriptions of all the features it contains. As shown, the LCD display, pushbuttons and switches are located on the board already. It also shows multiple I/Os that will be used to interface with the PIC microcontroller.

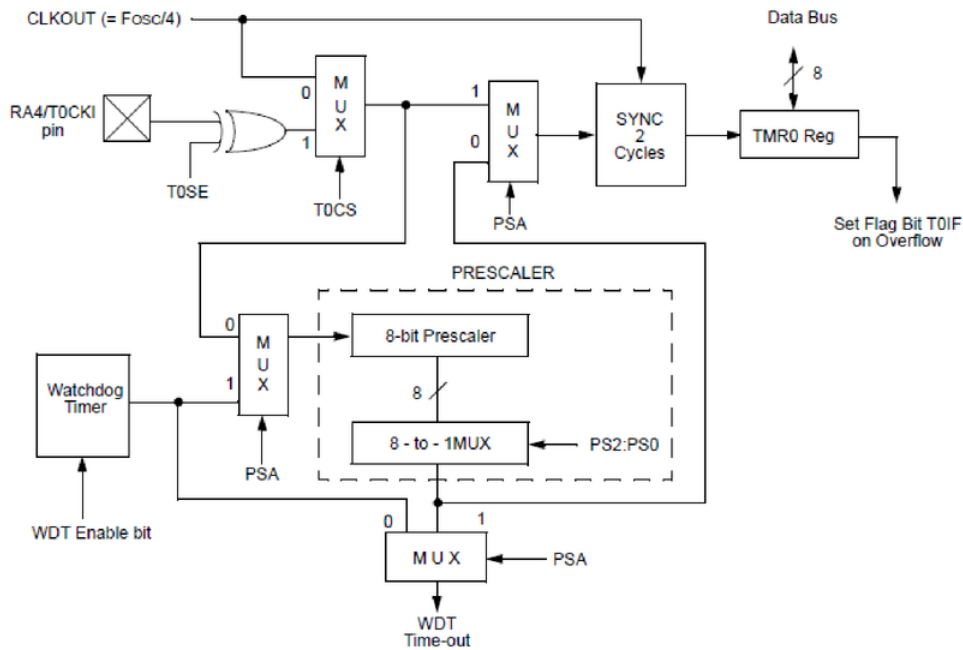
*Good description of
hardware needed for
the task.*

3 SYSTEM DESIGN AND CONFIGURATION

Before the program can be written, various on-chip peripherals must be configured through the use of the PIC16F877 datasheet.

3.1 TIMER 0

Timer 0 is an on-chip timer that will be used for the interrupt in the program. **Figure 3-1** shows the block diagram for Timer0 (TMR0).



Note: T0CS, T0SE, PSA, PS2:PS0 are (OPTION_REG<5:0>).

Figure 3-1

As shown, multiple registers must be set up in order for TMR0 to operate. These include OPTION_REG, INTCON_REG and TOIF overflow value. The configuration of these registers can be found in the following subsections.

TMR0 requires its own configuration so that the interrupt interval is set to 10ms. This can be seen in the following calculations:

The PIC16F877 has an internal clock source with a frequency of 19.6608MHz. This clock is labelled Fosc.

$$\text{Countable Clock Speed} = \frac{F_{osc}}{4} = 4915200\text{Hz}$$

$$\left(\frac{1}{4915200}\right) \times 256 = 52\mu\text{s}$$

$$\text{Max Prescaler} = 256$$

$$\frac{\frac{F_{osc}}{4}}{256} = \frac{4915200}{256} = 19200\text{Hz}$$

$$\text{Period} = \frac{1}{19200} \text{secs}$$

$$\text{Number of Counts every } \frac{1}{100} \text{secs} = 192 \text{ (Results in 10ms interrupt)}$$

$$\text{Preload TMR0} = 256 - 192 = 64$$

Therefore by adding the line TMR0 = 64, the timer gets preloaded with the value 64 which provides a 10ms interrupt.

3.1.1 OPTION_REG

"The OPTION_REG Register is a readable and writable register, which contains various control bits to configure the TMR0 prescaler/WDT postscaler (single assignable register known also as the prescaler), the external INT interrupt, TMR0 and the weak pull-ups on PORTB." (PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)

The configuration for OPTION_REG is provided by the datasheet in the form of descriptions for each bit in the configuration string. These can be seen below.

REGISTER 2-2: OPTION_REG REGISTER (ADDRESS 81h, 181h)

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
RBP _U	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	
bit 7								bit 0

- bit 7 **RBP_U**: PORTB Pull-up Enable bit
 1 = PORTB pull-ups are disabled
 0 = PORTB pull-ups are enabled by individual port latch values
- bit 6 **INTEDG**: Interrupt Edge Select bit
 1 = Interrupt on rising edge of RB0/INT pin
 0 = Interrupt on falling edge of RB0/INT pin
- bit 5 **T0CS**: TMR0 Clock Source Select bit
 1 = Transition on RA4/T0CKI pin
 0 = Internal instruction cycle clock (CLKO)
- bit 4 **T0SE**: TMR0 Source Edge Select bit
 1 = Increment on high-to-low transition on RA4/T0CKI pin
 0 = Increment on low-to-high transition on RA4/T0CKI pin
- bit 3 **PSA**: Prescaler Assignment bit
 1 = Prescaler is assigned to the WDT
 0 = Prescaler is assigned to the Timer0 module
- bit 2-0 **PS2:PS0**: Prescaler Rate Select bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

Figure 3-2

The ideal configuration for OPTION_REG for the system specified for this assignment is 0b10000111. This was produced by using the datasheet. Bit 7 is set to 1 as PORTB pull-up resistors are not required, and setting it to 1 disables them. Bit 6 determines whether the interrupt occurs on the rising or falling edge of the RB0 pin. For this system it is required to be on the falling edge, so this bit is set to 0. Bit 5 is used to select the clock source. The calculations for TMR0 configuration used the internal clock on the PIC. Therefore, this bit is set to 0 to select the internal clock. Bit 4 is used to select the source edge. This bit is set to 0 as an increment on low-to-high transition on R4 is required. Bit 3 is the prescaler assignment bit and is set to 0 as to assign the prescaler to the Timer 0 module. Bits 2-0 are used to select the prescaler rate. In this system a prescaler rate of 256 is required, meaning all three bits are set to 1. This results in a configuration string of 0b10000111.

3.1.2 INTCON_REG

"The OPTION_REG Register is a readable and writable register, which contains various control bits to configure the TMR0 prescaler/WDT postscaler (single assignable register known also as the prescaler), the external INT interrupt, TMR0 and the weak pull-ups on PORTB." (PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)

The configuration for INTCON_REG is provided by the datasheet in the form of descriptions for each bit in the configuration string. These can be seen below.

REGISTER 2-3: INTCON REGISTER (ADDRESS 0Bh, 8Bh, 10Bh, 18Bh)

	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF
	bit 7							bit 0
bit 7	GIE: Global Interrupt Enable bit 1 = Enables all unmasked interrupts 0 = Disables all interrupts							
bit 6	PEIE: Peripheral Interrupt Enable bit 1 = Enables all unmasked peripheral interrupts 0 = Disables all peripheral interrupts							
bit 5	TMR0IE: TMR0 Overflow Interrupt Enable bit 1 = Enables the TMR0 interrupt 0 = Disables the TMR0 interrupt							
bit 4	INTE: RB0/INT External Interrupt Enable bit 1 = Enables the RB0/INT external interrupt 0 = Disables the RB0/INT external interrupt							
bit 3	RBIE: RB Port Change Interrupt Enable bit 1 = Enables the RB port change interrupt 0 = Disables the RB port change interrupt							
bit 2	TMR0IF: TMR0 Overflow Interrupt Flag bit 1 = TMR0 register has overflowed (must be cleared in software) 0 = TMR0 register did not overflow							
bit 1	INTF: RB0/INT External Interrupt Flag bit 1 = The RB0/INT external interrupt occurred (must be cleared in software) 0 = The RB0/INT external interrupt did not occur							
bit 0	RBIF: RB Port Change Interrupt Flag bit 1 = At least one of the RB7:RB4 pins changed state; a mismatch condition will continue to set the bit. Reading PORTB will end the mismatch condition and allow the bit to be cleared (must be cleared in software). 0 = None of the RB7:RB4 pins have changed state							

Figure 3-3

(PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)

The configuration string required for INTCON_REG to operate as required is 0b11100000. Just as with the OPTION_REG, each bit represents an element of configuration. Bit 7 enables or disables all global interrupts. This system requires interrupts, therefore it is set to 1. Bit 6 is used as the peripheral interrupt enable bit, and is set to 1 to enable all unmasked peripheral interrupts. Bit 5 is used as an overflow interrupt enable bit. This is set to 1 to enable the TMR0 interrupt. Bit 4 is used to enable or disable external interrupts. This is not required and is therefore disabled by setting this bit to 0. Bit 3 is the RB port change interrupt enable bit. This is again not required and is therefore disabled by setting the bit to 0. Bit 2 is used as an overflow interrupt flag, and is set to 0 which means TMR0 did not overflow. Bit 1 is used as an external interrupt flag and is again set to 0. The final bit in the string, bit 0 is the RB port change interrupt flag bit. This is again set to 0 meaning that RB7:RB4 haven't changed states.

3.2 TIMER 1

Timer 1 is a 16 bit timer that will be used for this system to count pulses from the encoder. It has a maximum count of 65535 and uses two 8 bit registers to store the high byte and the low byte of the 16 bit count. TMR1H stores the high byte and TMR1L stores the low byte, both are readable and writable. **Figure 3-4** shows the Timer 1 block diagram found from the PIC16F877 datasheet.

TIMER1 BLOCK DIAGRAM

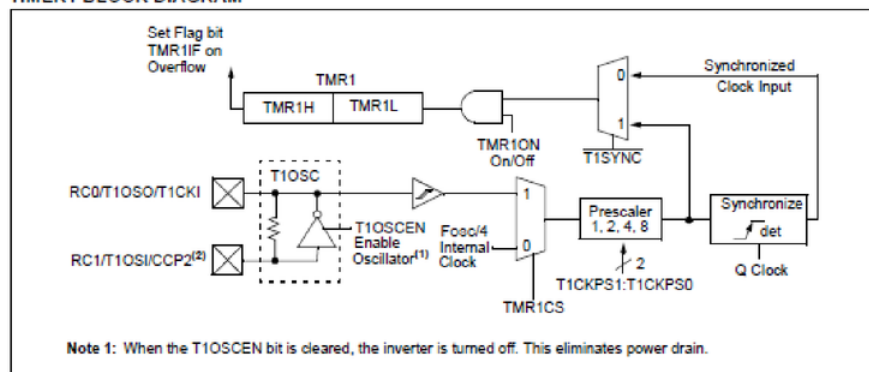


Figure 3-4

(PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)

The block diagram shows that the encoder can be connected to the microcontroller using the input pin RC0.

3.3 TIMER 2

Timer 2 is an 8 bit period register which gives a maximum value of 256. Timer 2 is set up using separate registers to Timer 0 and Timer 1. For this system Timer 2 will be used to interface the DC motor to the microcontroller through the use of PWM. Figure 3-5 shows the Timer 2 block diagram found from the PIC16F877 datasheet.

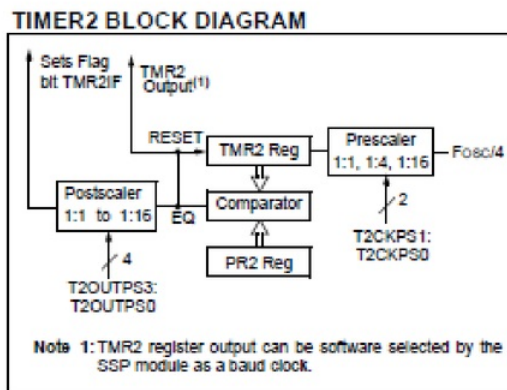


Figure 3-5

(PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)

The two registers that require configuration to set up Timer 2 are T2CON and CCP1CON. The configuration tables can be seen below for both registers.

REGISTER 7-1: T2CON: TIMER2 CONTROL REGISTER (ADDRESS 12h)

U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0
bit 7							bit 0

- bit 7 **Unimplemented:** Read as '0'
- bit 6-3 **TOUTPS3:TOUTPS0:** Timer2 Output Postscale Select bits
 0000 = 1:1 Postscale
 0001 = 1:2 Postscale
 0010 = 1:3 Postscale
 .
 .
 .
 1111 = 1:16 Postscale
- bit 2 **TMR2ON:** Timer2 On bit
 1 = Timer2 is on
 0 = Timer2 is off
- bit 1-0 **T2CKPS1:T2CKPS0:** Timer2 Clock Prescale Select bits
 00 = Prescaler is 1
 01 = Prescaler is 4
 1x = Prescaler is 16

Figure 3-6

REGISTER 8-1: CCP1CON REGISTER/CCP2CON REGISTER (ADDRESS: 17h/1Dh)

	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	—	—	CCPxX	CCPxY	CCPxM3	CCPxM2	CCPxM1	CCPxM0
	bit 7							bit 0
bit 7-6	Unimplemented: Read as '0'							
bit 5-4	CCPxX:CCPxY: PWM Least Significant bits							
	<u>Capture mode:</u> Unused							
	<u>Compare mode:</u> Unused							
	<u>PWM mode:</u> These bits are the two LSBs of the PWM duty cycle. The eight MSBs are found in CCPxL.							
bit 3-0	CCPxM3:CCPxM0: CCPx Mode Select bits							
	0000 = Capture/Compare/PWM disabled (resets CCPx module)							
	0100 = Capture mode, every falling edge							
	0101 = Capture mode, every rising edge							
	0110 = Capture mode, every 4th rising edge							
	0111 = Capture mode, every 16th rising edge							
	1000 = Compare mode, set output on match (CCPxIF bit is set)							
	1001 = Compare mode, clear output on match (CCPxIF bit is set)							
	1010 = Compare mode, generate software interrupt on match (CCPxIF bit is set, CCPx pin is unaffected)							
	1011 = Compare mode, trigger special event (CCPxIF bit is set, CCPx pin is unaffected); CCP1 resets TMR1; CCP2 resets TMR1 and starts an A/D conversion (if A/D module is enabled)							
	11xx = PWM mode							
Legend:								
R = Readable bit			W = Writable bit			U = Unimplemented bit, read as '0'		
- n = Value at POR			'1' = Bit is set			'0' = Bit is cleared		
						x = Bit is unknown		

Figure 3-7

T2CON is configured using the string 0b00000101. Bit 7 is unimplemented and is read as 0. Bit 6-3 represents the postscaler value. With a value of 0000 the postscaler is set to have a 1:1 ratio. Bit 2 enables Timer2 by setting it to 1. Bits 1 and 0 represent the prescaler. With these bits set to 01, the prescaler is set to 4.

CCP1CON is configured using the string 0b00001100. Using the datasheet it can be seen that bits 7 and 6 are unimplemented and are read as 0s. Bits 5-4 represent the PWM least significant bits. In order to set it to PWM mode, the bits are set to 00. Bits 3-0 are used to select the CCPx mode. When set to 1100, the PWM mode is selected.

3.4 PWM

In PWM mode, the CCPx pin produces up to a 10-bit resolution output. The simplified PWM diagram can be seen below.

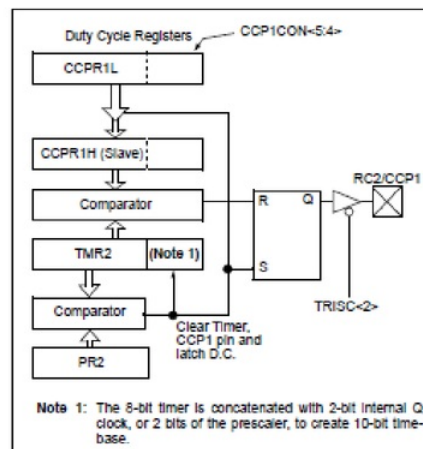


Figure 3-8

The 10 bit PWM resolution is explained well.

As shown, an 8-bit register is used to store CCPR1L, however it can be increased to a 10-bit register through the use of CCP1CON<5:4>. Two additional registers must be configured so that the PWM functions as required in this system. These registers are PIE2, and PIR2. PIE2 contains the individual enable bits for the CCP2 peripheral interrupt and is configured using the string PIE2 = 0b00000000. This was done using the register configuration table located on the PIC16F877 datasheet.

REGISTER 2-6: PIE2 REGISTER (ADDRESS 8Dh)

U-0	R/W-0	U-0	R/W-0	R/W-0	U-0	U-0	R/W-0
—	Reserved	—	EEIE	BCLIE	—	—	CCP2IE
bit 7							bit 0

- bit 7 **Unimplemented:** Read as '0'
- bit 6 **Reserved:** Always maintain this bit clear
- bit 5 **Unimplemented:** Read as '0'
- bit 4 **EEIE:** EEPROM Write Operation Interrupt Enable
1 = Enable EE Write Interrupt
0 = Disable EE Write Interrupt
- bit 3 **BCLIE:** Bus Collision Interrupt Enable
1 = Enable Bus Collision Interrupt
0 = Disable Bus Collision Interrupt
- bit 2-1 **Unimplemented:** Read as '0'
- bit 0 **CCP2IE:** CCP2 Interrupt Enable bit
1 = Enables the CCP2 interrupt
0 = Disables the CCP2 interrupt

Legend:

R = Readable bit W = Writable bit U = Unimplemented bit, read as '0'
 - n = Value at POR '1' = Bit is set '0' = Bit is cleared x = Bit is unknown

Figure 3-9

“The PIR2 register contains the flag bits for the CCP2 interrupt, the SSP bus collision interrupt and the EEPROM write operation interrupt.” (PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)

The PIR2 register is configured using the string PIR2 = 0b00000000 which was derived using the following configuration information.

REGISTER 2-7: PIR2 REGISTER (ADDRESS 0Dh)

	U-0	R/W-0	U-0	R/W-0	R/W-0	U-0	U-0	R/W-0
	—	Reserved	—	EEIF	BCLIF	—	—	CCP2IF
bit 7								bit 0
bit 7	Unimplemented: Read as '0'							
bit 6	Reserved: Always maintain this bit clear							
bit 5	Unimplemented: Read as '0'							
bit 4	EEIF: EEPROM Write Operation Interrupt Flag bit							
	1 = The write operation completed (must be cleared in software)							
	0 = The write operation is not complete or has not been started							
bit 3	BCLIF: Bus Collision Interrupt Flag bit							
	1 = A bus collision has occurred in the SSP, when configured for I2C Master mode							
	0 = No bus collision has occurred							
bit 2-1	Unimplemented: Read as '0'							
bit 0	CCP2IF: CCP2 Interrupt Flag bit							
	<u>Capture mode:</u>							
	1 = A TMR1 register capture occurred (must be cleared in software)							
	0 = No TMR1 register capture occurred							
	<u>Compare mode:</u>							
	1 = A TMR1 register compare match occurred (must be cleared in software)							
	0 = No TMR1 register compare match occurred							
	<u>PWM mode:</u>							
	Unused							

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
- n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

Figure 3-10

3.5 PORTS

The PIC16F877 has four different sets of ports; A, B, C and D. These ports can be used as either inputs or outputs depending on how they are configured. Port A is used for the pushbuttons which requires port A to be configured to be inputs. This is done using the string TRISA = 0xFF.

Port B is used for the LCD screen, which requires all the pins to be set to outputs. This is done using the string TRISB = 0x00.

Port C has one pin that will be used as the PWM output. All the other pins on this port are inputs. This is done using the string TRISC = 0b11111011 where RC2 is the output set to 0.

Port D is used to interface the microcontroller to the keypad. Half the pins are required to be outputs to do this, and the other half inputs. This is done using the string TRISD = 0b11110000, where RD0-RD3 are set to be outputs, and RD4-RD7 is set to inputs.

3.6 OTHER REGISTERS

This section will contain the configuration tables and information for any other registers being used within this system.

3.6.1 ADCON1

REGISTER 11-2: ADCON1 REGISTER (ADDRESS 9Fh)

U-0	U-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	—	—	—	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

bit 7 **ADFM**: A/D Result Format Select bit
 1 = Right justified. 6 Most Significant bits of ADRESH are read as '0'.
 0 = Left justified. 6 Least Significant bits of ADRESL are read as '0'.
 bit 6-4 **Unimplemented**: Read as '0'
 bit 3-0 **PCFG3:PCFG0**: A/D Port Configuration Control bits:

PCFG3: PCFG0	AN7 ⁽¹⁾ RE2	AN6 ⁽¹⁾ RE1	AN5 ⁽¹⁾ RE0	AN4 RA5	AN3 RA3	AN2 RA2	AN1 RA1	AN0 RA0	VREF+	VREF-	CHAN/ Refs ⁽²⁾
0000	A	A	A	A	A	A	A	A	VDD	VSS	8/0
0001	A	A	A	A	VREF+	A	A	A	RA3	VSS	7/1
0010	D	D	D	A	A	A	A	A	VDD	VSS	5/0
0011	D	D	D	A	VREF+	A	A	A	RA3	VSS	4/1
0100	D	D	D	D	A	D	A	A	VDD	VSS	3/0
0101	D	D	D	D	VREF+	D	A	A	RA3	VSS	2/1
011x	D	D	D	D	D	D	D	D	VDD	VSS	0/0
1000	A	A	A	A	VREF+	VREF-	A	A	RA3	RA2	6/2
1001	D	D	A	A	A	A	A	A	VDD	VSS	6/0
1010	D	D	A	A	VREF+	A	A	A	RA3	VSS	5/1
1011	D	D	A	A	VREF+	VREF-	A	A	RA3	RA2	4/2
1100	D	D	D	A	VREF+	VREF-	A	A	RA3	RA2	3/2
1101	D	D	D	D	VREF+	VREF-	A	A	RA3	RA2	2/2
1110	D	D	D	D	D	D	D	A	VDD	VSS	1/0
1111	D	D	D	D	VREF+	VREF-	D	A	RA3	RA2	1/2

A = Analog input D = Digital I/O

Note 1: These channels are not available on PIC16F873/876 devices.
Note 2: This column indicates the number of analog channels available as A/D inputs and the number of analog channels used as voltage reference inputs.

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
- n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

The ADRESH:ADRESL registers contain the 10-bit result of the A/D conversion. When the A/D conversion is complete, the result is loaded into this A/D result register pair, the GO/DONE bit (ADCON0<2>) is cleared and the A/D interrupt flag bit ADIF is set. The block diagram of the A/D module is shown in Figure 11-1.

After the A/D module has been configured as desired, the selected channel must be acquired before the conversion is started. The analog input channels must have their corresponding TRIS bits selected as inputs.

To determine sample time, see Section 11.1. After this acquisition time has elapsed, the A/D conversion can be started.

Figure 3-11

3.6.2 PIE1

PIE1: PERIPHERAL INTERRUPT ENABLE REGISTER 1

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
EEIE	ADIE	RCIE	C2IE	C1IE	OSFIE	TXIE	TMR1IE
bit 7							bit 0

Legend:

R = Readable bit

W = Writable bit

U = Unimplemented bit, read as '0'

-n = Value at POR

'1' = Bit is set

'0' = Bit is cleared

x = Bit is unknown

bit 7	EEIE: EE Write Complete Interrupt Enable bit 1 = Enables the EE write complete interrupt 0 = Disables the EE write complete interrupt
bit 6	ADIE: A/D Converter (ADC) Interrupt Enable bit 1 = Enables the ADC interrupt 0 = Disables the ADC interrupt
bit 5	RCIE: EUSART Receive Interrupt Enable bit 1 = Enables the EUSART receive interrupt 0 = Disables the EUSART receive interrupt
bit 4	C2IE: Comparator 2 Interrupt Enable bit 1 = Enables the Comparator C2 interrupt 0 = Disables the Comparator C2 interrupt
bit 3	C1IE: Comparator 1 Interrupt Enable bit 1 = Enables the Comparator C1 interrupt 0 = Disables the Comparator C1 interrupt
bit 2	OSFIE: Oscillator Fail Interrupt Enable bit 1 = Enables the oscillator fail interrupt 0 = Disables the oscillator fail interrupt
bit 1	TXIE: EUSART Transmit Interrupt Enable bit 1 = Enables the EUSART transmit interrupt 0 = Disables the EUSART transmit interrupt
bit 0	TMR1IE: Timer1 Overflow Interrupt Enable bit 1 = Enables the Timer1 overflow interrupt 0 = Disables the Timer1 overflow interrupt

PIE1 register bits are to enable the peripheral interrupts

Figure 3-12

3.6.3 PIE2

PIC16F882/883/884/886/887

2.2.2.5 PIE2 Register

The PIE2 register contains the interrupt enable bits, as shown in Register 2-5.

Note: Bit PEIE of the INTCON register must be set to enable any peripheral interrupt.

REGISTER 2-5: PIE2: PERIPHERAL INTERRUPT ENABLE REGISTER 2

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0
OSFIE	C2IE	C1IE	EEIE	BCIE	ULPWUE	—	CCP2IE
bit 7							bit 0

Legend:

R = Readable bit W = Writable bit U = Unimplemented bit, read as '0'
 -n = Value at POR '1' = Bit is set '0' = Bit is cleared x = Bit is unknown

- bit 7 **OSFIE:** Oscillator Fail Interrupt Enable bit
 1 = Enables oscillator fail interrupt
 0 = Disables oscillator fail interrupt
- bit 6 **C2IE:** Comparator C2 Interrupt Enable bit
 1 = Enables Comparator C2 interrupt
 0 = Disables Comparator C2 interrupt
- bit 5 **C1IE:** Comparator C1 Interrupt Enable bit
 1 = Enables Comparator C1 interrupt
 0 = Disables Comparator C1 interrupt
- bit 4 **EEIE:** EEPROM Write Operation Interrupt Enable bit
 1 = Enables EEPROM write operation interrupt
 0 = Disables EEPROM write operation interrupt
- bit 3 **BCIE:** Bus Collision Interrupt Enable bit
 1 = Enables Bus Collision interrupt
 0 = Disables Bus Collision interrupt
- bit 2 **ULPWUE:** Ultra Low-Power Wake-up Interrupt Enable bit
 1 = Enables Ultra Low-Power Wake-up interrupt
 0 = Disables Ultra Low-Power Wake-up interrupt
- bit 1 **Unimplemented:** Read as '0'
- bit 0 **CCP2IE:** CCP2 Interrupt Enable bit
 1 = Enables CCP2 interrupt
 0 = Disables CCP2 interrupt

Figure 3-13

3.6.4 PIR1

PIR1: PERIPHERAL INTERRUPT REQUEST REGISTER 1

R/W-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0	R-0	R/W-0
EEIF	ADIF	RCIF	C2IF	C1IF	OSFIF	TXIF	TMR1IF
bit 7							bit 0

Legend:

R = Readable bit W = Writable bit U = Unimplemented bit, read as '0'
 -n = Value at POR '1' = Bit is set '0' = Bit is cleared x = Bit is unknown

bit 7	EEIF: EEPROM Write Operation Interrupt Flag bit 1 = The write operation completed (must be cleared in software) 0 = The write operation has not completed or has not been started
bit 6	ADIF: A/D Converter Interrupt Flag bit 1 = A/D conversion complete (must be cleared in software) 0 = A/D conversion has not completed or has not been started
bit 5	RCIF: EUSART Receive Interrupt Flag bit 1 = The EUSART receive buffer is full (cleared by reading RCREG) 0 = The EUSART receive buffer is not full
bit 4	C2IF: Comparator C2 Interrupt Flag bit 1 = Comparator output (C2OUT bit) has changed (must be cleared in software) 0 = Comparator output (C2OUT bit) has not changed
bit 3	C1IF: Comparator C1 Interrupt Flag bit 1 = Comparator output (C1OUT bit) has changed (must be cleared in software) 0 = Comparator output (C1OUT bit) has not changed
bit 2	OSFIF: Oscillator Fail Interrupt Flag bit 1 = System oscillator failed, clock input has changed to INTOSC (must be cleared in software) 0 = System clock operating
bit 1	TXIF: EUSART Transmit Interrupt Flag bit 1 = The EUSART transmit buffer is empty (cleared by writing to TXREG) 0 = The EUSART transmit buffer is full
bit 0	TMR1IF: Timer1 Overflow Interrupt Flag bit 1 = The TMR1 register overflowed (must be cleared in software) 0 = The TMR1 register did not overflow

Figure 3-14

3.6.5 PIR2

PIR2 Register

The PIR2 register contains the interrupt flag bits.

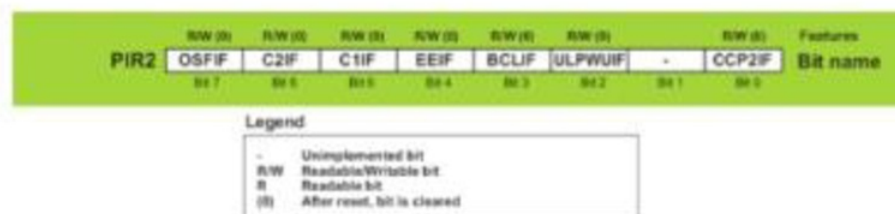


Fig. 2-18 PIR2 register

OSFIF - Oscillator Fail Interrupt Flag bit.

- 1 - System oscillator failed and clock input has changed to internal oscillator INTOSC. This bit must be cleared in software.
- 0 - System oscillator operates normally.

C2IF - Comparator C2 Interrupt Flag bit.

- 1 - Comparator C2 output has changed (bit C2OUT). This bit must be cleared in software.
- 0 - Comparator C2 output has not changed.

C1IF - Comparator C1 Interrupt Flag bit.

- 1 - Comparator C1 output has changed (bit C1OUT). This bit must be cleared in software.
- 0 - Comparator C1 output has not changed.

EEIF - EE Write Operation Interrupt Flag bit.

- 1 - EEPROM write completed. This bit must be cleared in software.
- 0 - EEPROM write is not completed or has not started.

BCLIF - Bus Collision Interrupt Flag bit.

- 1 - A bus collision has occurred in the MSSP when configured for I2C Master mode. This bit must be cleared in software.
- 0 - No bus collision has occurred.

ULPWUIF - Ultra Low-power Wake-up Interrupt Flag bit.

- 1 - Wake-up condition has occurred. This bit must be cleared in software.
- 0 - No Wake-up condition has occurred.

CCP2IF - CCP2 Interrupt Flag bit.

- 1 - CCP2 interrupt condition has occurred (unit for capturing, comparing and generating PWM signal). Depending on operating mode, capture or compare match has occurred. In both cases, the bit must be cleared in software. This bit is not used in PWM mode.
- 0 - No CCP2 interrupt condition has occurred.

Figure 3-15

3.6.6 T1CON

REGISTER 6-1: T1CON: TIMER1 CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
T1GINV ⁽¹⁾	TMR1GE ⁽²⁾	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON
bit 7							bit 0

Legend:

R = Readable bit W = Writable bit U = Unimplemented bit, read as '0'
 -n = Value at POR '1' = Bit is set '0' = Bit is cleared x = Bit is unknown

- bit 7 **T1GINV:** Timer1 Gate Invert bit⁽¹⁾
 1 = Timer1 gate is active-high (Timer1 counts when gate is high)
 0 = Timer1 gate is active-low (Timer1 counts when gate is low)
- bit 6 **TMR1GE:** Timer1 Gate Enable bit⁽²⁾
 If TMR1ON = 0:
 This bit is ignored
 If TMR1ON = 1:
 1 = Timer1 is on if Timer1 gate is not active
 0 = Timer1 is on
- bit 5-4 **T1CKPS<1:0>:** Timer1 Input Clock Prescale Select bits
 11 = 1:8 Prescale Value
 10 = 1:4 Prescale Value
 01 = 1:2 Prescale Value
 00 = 1:1 Prescale Value
- bit 3 **T1OSCEN:** LP Oscillator Enable Control bit
 If INTOSC without CLKOUT oscillator is active:
 1 = LP oscillator is enabled for Timer1 clock
 0 = LP oscillator is off
 Else:
 This bit is ignored. LP oscillator is disabled.
- bit 2 **T1SYNC:** Timer1 External Clock Input Synchronization Control bit
 TMR1CS = 1:
 1 = Do not synchronize external clock input
 0 = Synchronize external clock input
 TMR1CS = 0:
 This bit is ignored. Timer1 uses the internal clock
- bit 1 **TMR1CS:** Timer1 Clock Source Select bit
 1 = External clock from T1CK1 pin (on the rising edge)
 0 = Internal clock (Fosc/4)
- bit 0 **TMR1ON:** Timer1 On bit
 1 = Enables Timer1
 0 = Stops Timer1

Note 1: T1GINV bit inverts the Timer1 gate logic, regardless of source.

2: TMR1GE bit must be set to use either T1G pin or COUT, as selected by the T1GSS bit of the CMCON1 register, as a Timer1 gate source.

Figure 3-16

3.6.7 Timer 2 Control Registers

REGISTERS ASSOCIATED WITH TIMER2 AS A TIMER/COUNTER

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
0Bh, 8Bh, 10Bh, 18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	0000 000u
0Ch	PIR1	PSPIF ⁽¹⁾	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
8Ch	PIE1	PSPIE ⁽¹⁾	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
11h	TMR2	Timer2 Module's Register								0000 0000	0000 0000
12h	T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000	-000 0000
92h	PR2	Timer2 Period Register								1111 1111	1111 1111

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by the Timer2 module.

Note 1: Bits PSPIE and PSPIF are reserved on 28-pin devices; always maintain these bits clear.

Figure 3-17

The operation of timer2 is controlled by T2CON register. In below find the bit pattern of T2CON register.

T2CON: Timer2 Control Register

U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0
bit 7							bit 0

bit 7 Unimplemented: Read as '0'
 bit 6:3 TOUTPS3:TOUTPS0:
 Timer2 Output Postscale Select bits
 0000 = 1:1 Postscale
 0001 = 1:2 Postscale
 •
 •
 1111 = 1:16 Postscale

bit 2 TMR2ON: Timer2 On bit
 1 = Timer2 is on
 0 = Timer2 is off
 bit 1:0 T2CKPS1:T2CKPS0: Timer2 Clock Prescale Select bits
 00 = Prescaler is 1
 01 = Prescaler is 4
 1x = Prescaler is 16

Figure 3-18

3.6.8 CCP1CON

REGISTER 10-1: CCP1CON REGISTER (ADDRESS 17h)

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	CCP1X	CCP1Y	CCP1M3	CCP1M2	CCP1M1	CCP1M0
bit7							bit0

R = Readable bit
W = Writable bit
U = Unimplemented bit, read as '0'
-n = Value at POR reset

bit 7-6: **Unimplemented:** Read as '0'

bit 5-4: **CCP1X:CCP1Y:** PWM Least Significant bits
Capture Mode: Unused
Compare Mode: Unused
PWM Mode: These bits are the two LSBs of the PWM duty cycle. The eight MSBs are found in CCPRxL.

bit 3-0: **CCP1M3:CCP1M0:** CCPx Mode Select bits
0000 = Capture/Compare/PWM off (resets CCP1 module)
0100 = Capture mode, every falling edge
0101 = Capture mode, every rising edge
0110 = Capture mode, every 4th rising edge
0111 = Capture mode, every 16th rising edge
1000 = Compare mode, set output on match (CCP1IF bit is set)
1001 = Compare mode, clear output on match (CCP1IF bit is set)
1010 = Compare mode, generate software interrupt on match (CCP1IF bit is set, CCP1 pin is unaffected)
1011 = Compare mode, trigger special event (CCP1IF bit is set; CCP1 resets TMR1)
11xx = PWM mode

Figure 3-19

3.6.9 PWM and T2 Registers

REGISTERS ASSOCIATED WITH PWM AND TIMER2

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
0Bh, 8Bh, 10Bh, 18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	0000 000u
0Ch	PIR1	PSPIF ⁽¹⁾	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
0Dh	PIR2	—	—	—	—	—	—	—	CCP2IF	---- --0	---- --0
8Ch	PIE1	PSPIE ⁽¹⁾	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
8Dh	PIE2	—	—	—	—	—	—	—	CCP2IE	---- --0	---- --0
87h	TRISC	PORTC Data Direction Register								1111 1111	1111 1111
11h	TMR2	Timer2 Module's Register								0000 0000	0000 0000
92h	PR2	Timer2 Module's Period Register								1111 1111	1111 1111
12h	T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000	-000 0000
15h	CCPR1L	Capture/Compare/PWM Register 1 (LSB)								xxxx xxxx	uuuu uuuu
16h	CCPR1H	Capture/Compare/PWM Register 1 (MSB)								xxxx xxxx	uuuu uuuu
17h	CCP1CON	—	—	CCP1X	CCP1Y	CCP1M3	CCP1M2	CCP1M1	CCP1M0	--00 0000	--00 0000
1Bh	CCPR2L	Capture/Compare/PWM Register 2 (LSB)								xxxx xxxx	uuuu uuuu
1Ch	CCPR2H	Capture/Compare/PWM Register 2 (MSB)								xxxx xxxx	uuuu uuuu
1Dh	CCP2CON	—	—	CCP2X	CCP2Y	CCP2M3	CCP2M2	CCP2M1	CCP2M0	--00 0000	--00 0000

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by PWM and Timer2.

Note 1: Bits PSPIE and PSPIF are reserved on 28-pin devices; always maintain these bits clear.

Figure 3-20 Configuration
n
explanation
is very
good.

4 SOFTWARE DESIGN AND EXPLANATION

4.1 HEADER FILES, FUNCTION PROTOTYPES AND GLOBAL VARIABLES

The first section of code in the program includes the header files, function prototypes and global variables. The header files include pic.h and LCD.h. Pic.h is required as it contains the inner workings and libraries needed for the PIC16F877 to be used. LCD.h is included as it contains the pre-written functions that the LCD needs to operate. Using LCD.h is convenient as these functions do not have to be manually written again, which saves time.

Following the header files are the Function Prototypes. There are four prototypes included here; DisplaySpeed, DisplayValues, ScanKeypad and Delay. Each of these functions will be discussed in their own section later on. Each of the prototypes include the type of data they are bringing in and returning.

Finally, the global variables are declared. These variables can be accessed by any function within this program as they are global. Firstly, milliseconds, speed, fault, speed_low_byte, speed_high_byte and run are all declared as integers. They are also all initialised to have a value of 0. This means that when the system is started up, all these values have an initial value of 0.

Following these are the more important control variables, which are given a data type float as they require a higher degree of accuracy to have more control over the system. These variables are setpoint, error, output, prop_part, int_accum kp and ki. All of these are initialised to 0 once again, apart from Kp and Ki. Kp and Ki were set to 0.5 and 0.01 through tweaking to be ideal values for the practical build. This code can be seen in **Figure 4-1**.

```
//Header Files
#include <pic.h>
#include "LCD.h"

//Function Prototypes
void DisplaySpeed(int line, int value);
void DisplayValues(int line, int value1, int value2, int value3);
int ScanKeypad(void);
void Delay(int t);

//Global Variables
int milliseconds=0, speed=0, fault=0; //global variables
int speed_low_byte=0, speed_high_byte=0, run=0;
float setpoint=0, error=0, output=0, prop_part=0, int_accum = 0 ,Kp = 0.5, Ki = 0.1;
```

Figure 4-1

4.2 INTERRUPT SERVICE ROUTINE

The next section of code contains the interrupt service routine (ISR). The ISR is used to run periodically to accurately count pulses. The ISR doesn't run the risk of being delayed by other functions within the code as it interrupts whatever is currently being executed, to execute its own function. The code within the ISR can be seen in **Figure 4-2**.

*Good
explanati
on of
how the
ISR
operates
within the*

```

////////////////////////////////// ISR //////////////////////////////////////////
void interrupt ISR(void)                                     //there are 3 possible interrupt sources
{
    if(TOIF==1)                                             //timer 0 is timing, interrupt every 10ms
    {
        TOIF = 0;
        TMR0 = 64;
        milliseconds+=10;
        if((milliseconds==1000) && (fault==0))
        {
            speed_low_byte = TMR1L;
            speed_high_byte = TMR1H;
            TMR1L = 0;
            TMR1H = 0;
            milliseconds=0;
        }
    }
    if(TMR1IF==1)                                           //timer 1 is in event mode counting pulses
    {
        TMR1IF = 0;                                         //clear the interrupt flag
        fault = 1;
    }
    if(TMR2IF==1)                                           //timer 2 is for PWM
    {
        TMR2IF = 0;
    }
}

```

Figure 4-2

The code has 3 possible interrupt sources, TOIF, TMR1IF and TMR2IF. Since Timer 1 is set to event mode counting incoming pulses, If timer 1 overflows and the TMR1IF is set, fault is changed to 1 and the interrupt flag is cleared. This shows that a fault has occurred within the system and it isn't functioning correctly.

If timer 2 overflows and TMR2IF is set, then the interrupt flag is simply cleared. The majority of the code within the ISR involves Timer 0 and its interrupt flag. If TOIF is set, the flag gets reset and TMR0 is preloaded with 64 (For 10ms interrupts). The milliseconds global variable is incremented by 10 and then an if statement is reached. This if statement states that if milliseconds is equal to 1000 and fault is equal to 0, the following code will be executed. Therefore these lines of code will be executed every 1000ms.

The first two lines of code sets the global variables speed_low_byte and speed_high_byte to equal the low and high TMR1 registers respectively. The two registers TMR1L and TMR1H are then set to equal 0, along with the milliseconds variable.

*This is
explain
ed
clearly.*

4.3 MAIN PROGRAM

This section of code is split into two main sections; the configuration strings, and the while super loop.

4.3.1 Configurations

This section of code can be seen in **Figure 4-3**.

```

//////////////////////////////////// main program //////////////////////////////////////
void main(void)
{
    int enable_keypad=1, key;
    TRISA = 0xFF;

    ADCON1 = 0x8E;           //Configure ADCON1 = 0000 1110
    TRISB = 0x00;           //used for LCD display
    TRISC = 0x01;           //input RC0 used for counting with TMR1
    TRISD = 0b1110000;      //output RC2 used for PWM with TMR2

    PORTB = 0x00;
    LCD_Startup();

    PIE1 = 0b00000011;      //Peripheral interrupt enable register for TMR1 & TMR2
    PIE2 = 0;
    PIR1 = 0;               //clear Peripheral interrupts
    PIR2 = 0;               //clear Peripheral interrupts

    INTCON = 0b11100000;     //setup for TIMER 0 in timer mode
    OPTION_REG = 0b10000111;

    T1CON = 0b00000111;     //setup for TIMER 1 in event mode from RC0
    TMR1L = 0;
    TMR1H = 0;

    PR2 = 255;
    CCP1L = 0;              //0% DUTY CYCLE
    T2CON = 0b00000101;     //set up for Timer 2 in PWM mode using RC2
    CCP1CON = 0b00001100;   //set up for Timer 2 in PWM mode using RC2
                           //sets pwm mode and the extra 2 LSB's for PWM
                           //this is the alternative to 1023 / 4 for max PR2 = 255
                           //PWM is now (PR2 = 255)8 bits + CCP1CON 2 bits
                           //this gives you the 10 bits PWM
}

```

Figure 4-3

The first line of code within this section declares two local variables, the first enables the keypad through the use of the local variable `enable_keypad` being set to 1. The other is 'key' which will be used to store the value input by the user on the keypad.

Following this are the port configuration strings which were previously discussed in **Section 3.5**. `ADCON1` is also configured here with the string `ADCON1 = 0x8E` which sets the bits to 00001110. After the ports are configured the `LCD_Startup` function is called. This function is included in the `LCD.h` header file and is used to initialise the LCD screen so it is ready to be used.

As previously discussed, `PIE1` is set to `0b00000011` which enables the peripheral interrupt register for `TMR1` and `TMR2`. `PIE2` is cleared along with `PIR1` and `PIR2` by setting them to 0. `INTCON` and `OPTION_REG` registers are set to the previously discussed configurations.

`T1CON` is then configured to set up Timer 1 in event mode from `RC0`. The high and low `TMR1` registers are then set to equal 0. Finally, `PR2` is set to 255 which is the highest value, `CCP1L` is set to 0 to create a 0% duty cycle, `T2CON` and `CCP1CON` are then configured as previously discussed to set up timer 2 in PWM mode.

4.3.2 While Super Loop

The code within the while loop can be seen in **Figure 4-4**.

E
x
p
l
a
n
a
t
i
o
n
i
s
c
l
e
a
r
i
n
g

```

while(1)
{
    speed = (int)((float)((speed_low_byte + speed_high_byte*256)*6)/50);
    //speed = (int)((float)(speed_low_byte + speed_high_byte*256));
    //speed = speed * 3.1;
    //DisplaySpeed(0,speed);

    if((RA1==1)&&(RA2==0)) run = 1;
    if((RA1==0)&&(RA2==1)) run = 0;

    key = ScanKeypad();

    if((key!=-1) && (enable_keypad==1))
    {
        setpoint = setpoint * 10 + (float)key;
        enable_keypad=0;
    }

    // if((key=='#')&& (enable_keypad==1))
    if(key=='#')
    {
        setpoint = 0;
    }

    if(key==-1) enable_keypad=1;

    if(setpoint>2300) setpoint = 2300;
    if(setpoint<0) setpoint = 0;

    if(run==1)
    {
        error = setpoint - (float)speed;
        prop_part = error * Kp;
        int_accum += error * Ki;
        output = prop_part + int_accum;
    }
    else
    {
        error = 0;
        int_accum = 0;
        output = 0;
        RC2 = 0;
    }

    if(output < 0) output = 0;
    if(output > 1023) output = 1023;

    CCPRL1 = (int)output/4;

    DisplayValues(0, (int)setpoint, speed, (int)error);
    DisplayValues(1, (int)prop_part, (int)int_accum, (int)output);
}

```

Figure 4-4

This section of code is contained within a super loop initiated with the line `while(1)`. This means that the code will be executed over and over again until the system is powered down, only to be interrupted by the ISR. The first line of code within this loop is a calculation used to calculate the speed the motor is currently running at. The ISR obtains values from TMR1L and TMR1H and stores them within the global variables `speed_low_byte` and `speed_high_byte`. These values are physical parameters used in the calculation.

The next two lines are used to start or stop the motor depending on which pushbutton is pressed. The start button is connected to RA1, and the stop button is connected to RA2. If either of the if statements is found to be true, then `run` is set to 1 to start, or 0 to stop.

The following line is a function call which assigns the returned value from the `ScanKeypad` function to the local variable 'key'. This is then followed by a series of if statements which depend on the value of key. If key is equal not equal to -1 and the keypad is enabled, the input value is processed so that the value the user inputs can be displayed on the LCD screen. This is done by

having the current value stored within the setpoint variable multiplied by 10, then added to the most recent input value from the keypad. An example of this can be seen below for a value of 2468.

$$2 \text{ is input, setpoint} = (0 * 10) + 2 = 2$$

$$4 \text{ is input, setpoint} = (2 * 10) + 4 = 24$$

$$6 \text{ is input, setpoint} = (24 * 10) + 6 = 246$$

$$8 \text{ is input, setpoint} = (246 * 10) + 8 = 2468$$

Clear explanation of how the keypad encoder works for 4 figure values. worth noting there is a latch here which prevents debounce of the keypress.

The next if statement is executed if # is pressed. The code within this statement simply resets the value of setpoint to 0. This is followed by an if statement that checks to see if the value of key is equal to -1 and the keypad is enabled. If this is true, then the program checks to see if the value of setpoint is greater than 2300, or less than 0. If setpoint is greater than 2300, the value of setpoint is set to 2300. Likewise, if the value of setpoint is less than 0, it is set to equal 0. These two statements act as upper and lower limits to stop the user from inputting a setpoint that is too high, or too low which could be dangerous for the system.

The next section of code is contained within an if else statement. The first section is executed if run is equal to 1. This code acts as the PI controller algorithm. Firstly, the error is calculated by subtracting the speed from the setpoint. This error is then used to calculate the proportional part of the PI controller by multiplying the error by Kp. The integral part of the PI controller is then calculated by multiplying the error by Ki and accumulating it each time a calculation is done. Finally, the output of the PI controller is calculated by adding the proportional and integral parts together. This creates an output for the PWM.

If run is not equal to 1, the else statement is executed which sets the error, the integral accumulation, output and RC2 all to 0. This resets the PI controller if the stop button is pressed. An upper and lower limit for the output is then set to 1023 and 0 respectively to stop the output from going dangerously high or negative. CCPR1L is then set to equal the output/4 and two function calls for DisplayValues are declared. The first uses line 0 and is set to display the setpoint, speed and the error. The second function call uses line 1 and is set to display the proportional, integral accumulation and the output. Therefore, all the vital information for the controller and motor performance is displayed on the LCD as the specification requires.

The output / 4 was for testing and the final system used the 10 bit resolution. This should be stated here. The explanation of the code here is correct.

4.4 DISPLAYSPEED FUNCTION

The next section of code is the function description for DisplaySpeed. This function is used to store the speed in an array ready to be displayed on the LCD. The values for line and value are brought into this function to be used. **Figure 4-5** shows the code.

```
void DisplaySpeed(int line, int value)
{
    int units, tens, hunds, thous, temp;
    unsigned char string[17];

    thous = value /1000;
    hunds = value %1000/100;
    tens = value %1000%100/10;
    units = value %1000%100%10;

    string[0] = thous + 48;
    string[1] = hunds + 48;
    string[2] = tens + 48;
    string[3] = units + 48;
    string[4] = ' ';
    string[5] = 'R';
    string[6] = 'P';
    string[7] = 'M';
    string[8] = 0;

    DisplayString(line, string);    //send string data to LCD line 1
}
```

Figure 4-5

The first line of code declares five local variables of the datatype integer. These are units, tens, hunds, thous and temp. An array of 17 elements is then declared which will be used to store the units, tens, hundreds and thousands digits along with the letters RPM.

Following the declaration of the array are a series of calculations used to convert the number stored in the value variable into individual digits for units, tens, hundreds and thousands. These digits are then stored in the array elements 0-3. Each digit's variable is followed by a +48 to convert the value into its ASCII equivalent. The 8th element of the array contains a 0 to tell the program to end the array there.

This function ends with a DisplayString function call which sends the value in line, and the data stored within the array to the DisplayString function. This information is to be displayed on line 1 of the LCD, therefore when DisplaySpeed is called upon, the value of line is set to be 1.

4.5 DISPLAYVALUES FUNCTION

The next function definition is DisplayValues which can be seen in **Figure 4-6**. The function brings in the line, value1, value2 and value 3 variables to be used. The first line of code declares the same 5 local variables as the previous function. An array of 17 elements is once again declared in the next line. The rest of this function operates in the same way as DisplaySpeed, where the values stored within the value1, value2 and value3 variables are all converted into their individual digits, stored within the array and then the array is sent to the DisplayString function using line 1.

```
void DisplayValues(int line, int value1, int value2, int value3)
{
    int units, tens, hunds, thous, temp;
    unsigned char string[17];

    if(value1>=0)    string[0] = '+';
    else
    {
        string[0] = '-';
        value1 = value1*(-1);
    }

    thous = value1 /1000;
    hunds = value1 %1000/100;
    tens = value1 %1000%100/10;
    units = value1 %1000%100%10;

    string[1] = thous + 48;
    string[2] = hunds + 48;
    string[3] = tens + 48;
    string[4] = units + 48;

    if(value2>=0)    string[5] = '+';
    else
    {
        string[5] = '-';
        value2 = value2*(-1);
    }
    thous = value2 /1000;
    hunds = value2 %1000/100;
    tens = value2 %1000%100/10;
    units = value2 %1000%100%10;

    string[6] = thous + 48;
    string[7] = hunds + 48;
    string[8] = tens + 48;
    string[9] = units + 48;

    if(value3>=0)    string[10] = '+';
    else
    {
        string[10] = '-';
        value3 = value3*(-1);
    }
    thous = value3 /1000;
    hunds = value3 %1000/100;
    tens = value3 %1000%100/10;
    units = value3 %1000%100%10;

    string[11] = thous + 48;
    string[12] = hunds + 48;
    string[13] = tens + 48;
    string[14] = units + 48;
    string[15] = 0;

    DisplayString(line, string);    //send string data to LCD line 1
}
```

Figure 4-6

Expl
anat
ions
are

4.6 SCANKEYPAD FUNCTION

This function is used to scan the keypad so that when the user presses a key, it detects the correct value input. The function returns an integer in the form of the variable 'key'. Key is initially set to -1 as there is no -1 button on the keypad, allowing it to operate as a placeholder value.

This function determines which key is pressed by scanning each column in sequence by enabling and disabling the D ports. All three columns act on the same principle so only one needs to be described. For column 1, RD0 is set to output 1. This enables all the buttons on column 1 to be scanned. This is followed by a delay to increase stability. Following the delay are 4 if statements, each of which checks to see if a button has been pressed. RD4 represents key1, RD5 represents key4, RD6 represents key7 and RD7 represents key*. RD0 is then set back to 0, and the same process happens for column 2 where RD1 is enabled. Column 3 is scanned last when RD2 is enabled. This cycles until a button press has been detected and that value is returned.

```
int ScanKeypad(void)
{
    int key=-1;

    RD0 = 1; //scan column 1
    Delay(1);
    if(RD4 == 1) key = 1;
    if(RD5 == 1) key = 4;
    if(RD6 == 1) key = 7;
    if(RD7 == 1) key = '*';
    RD0 = 0; //disable column 1

    RD1 = 1; //scan column 2
    Delay(1);
    if(RD4 == 1) key = 2;
    if(RD5 == 1) key = 5;
    if(RD6 == 1) key = 8;
    if(RD7 == 1) key = 0;
    RD1 = 0; //disable column 2

    RD2 = 1; //scan column 3
    Delay(1);
    if(RD4 == 1) key = 3;
    if(RD5 == 1) key = 6;
    if(RD6 == 1) key = 9;
    if(RD7 == 1) key = '#';
    RD2 = 0; //disable column 3
    return key;
}
```

Good explanation that explains the function. Worth pointing out that this is called from the while loop and returns the key pressed.

4.7 DELAY FUNCTION

The final function description is the delay function. This can be seen in **Figure 4-7**.

```
void Delay(int t)
{
    int i;
    for(i=0;i<t;i++);
}
```

Figure 4-7

This function is used as a debouncer and to increase stability of the program. It takes a set value and increments a variable until that value is met. Thus, creating a delay for however many clock pulses it takes to reach the set value.

5 TESTING AND ANALYSIS OF RESULTS

5.1 SIMULATED TEST

The first tests for the program are simulations using Proteus. A simulated experiment can be seen in

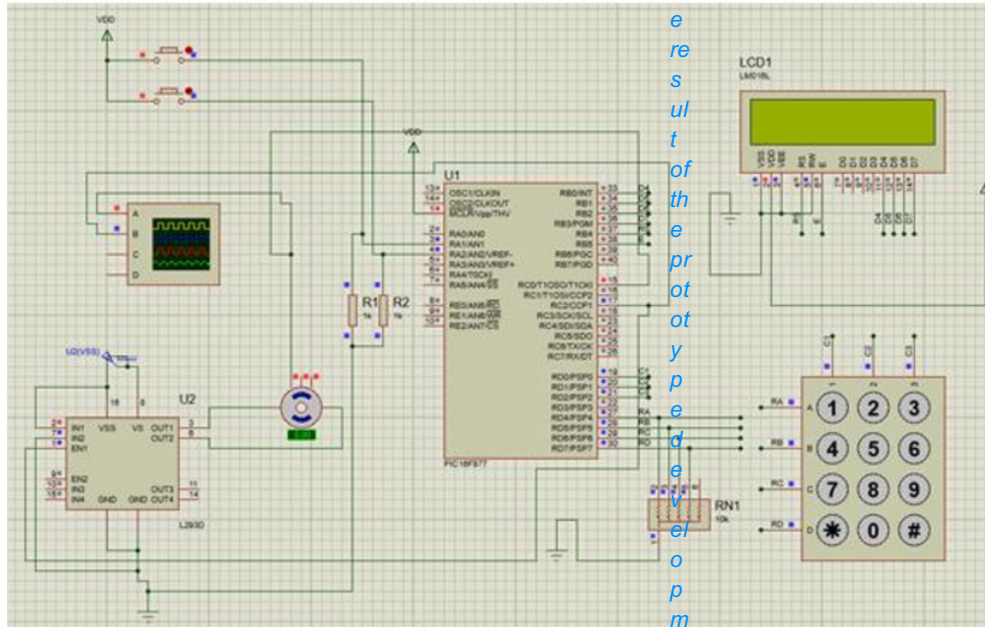


Figure 5-1

As shown, the start / stop push buttons are connected to port A, the LCD is connected to port B, port C is used for the PWM output to the motor, and port D is used to interface with the keypad. The PWM wave is needed to be tested to make sure that the program produces the right output to control the motor's speed. This is done by testing four set point values. These tests can be seen as follows, and all the results are documented in a table.

The following equation is used to calculate the frequency.

$$Frequency = \frac{1}{(division\ on\ scope) \times (timebase\ on\ scope)}$$

RPM is calculated by:

$$RPM = Frequency \times \left(\frac{60}{PPR}\right)$$

$$PPR(Pulses\ per\ revolution) = 100$$

5.1.1 Simulated Test 1 (100 rpm setpoint)

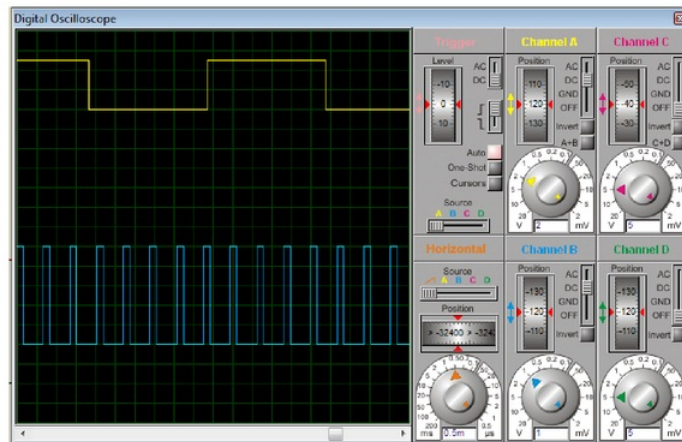


Figure 5-2

$$Frequency = \frac{1}{6} = 166.67$$

$$Calculated\ RPM = 166.7 \times \left(\frac{60}{100}\right) = 100\ rpm$$

$$Actual\ Speed = 99$$

5.1.2 Simulated Test 2 (200 rpm setpoint)

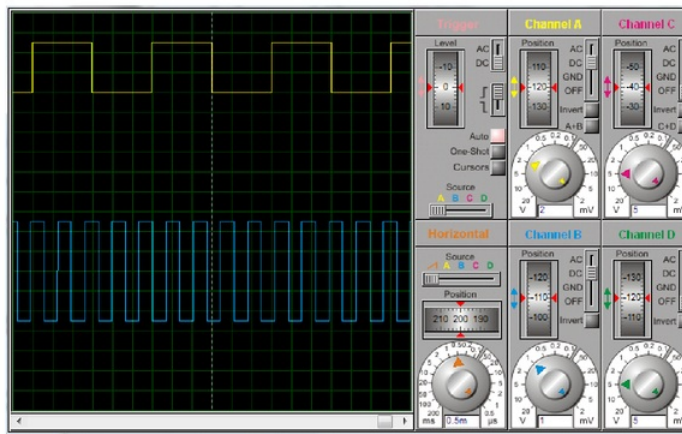


Figure 5-3

$$Frequency = \frac{1}{3} = 333.33Hz$$

$$\text{Calculated RPM} = 333.33 \times \left(\frac{60}{100}\right) = 200 \text{ rpm}$$

$$\text{Actual Speed} = 199 \text{ rpm}$$

5.1.3 Simulated Test 3 (300 rpm setpoint)

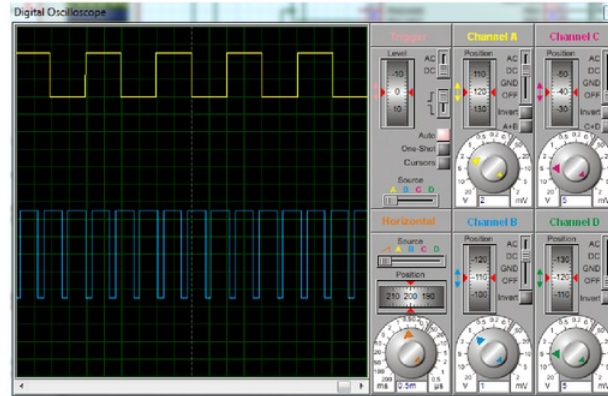


Figure 5-4

$$\text{Frequency} = \frac{1}{2} = 500\text{Hz}$$

$$\text{Calculated rpm} = 500 \times \left(\frac{60}{100}\right) = 300 \text{ rpm}$$

$$\text{Actual speed} = 298 \text{ rpm}$$

5.1.4 Simulated Test 4 (360 rpm setpoint)

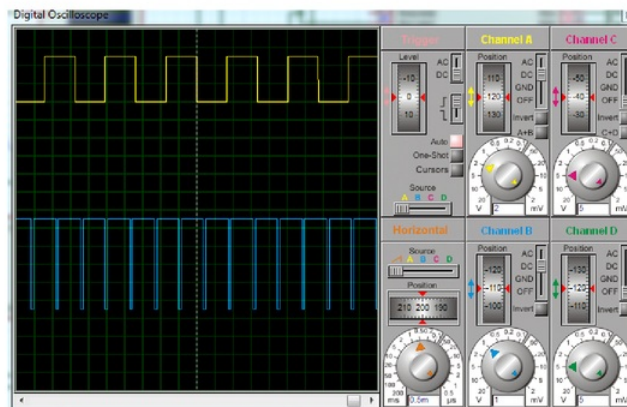


Figure 5-5

$$\text{Frequency} = \frac{1}{1.76\text{m}} = 568.18\text{Hz}$$

$$\text{Calculated rpm} = 568.18 \times \left(\frac{60}{100} \right) = 360 \text{ rpm}$$

$$\text{Actual speed} = 356 \text{ rpm}$$

The tests carried out are good.

5.2 TABLE OF SIMULATED RESULTS

Setpoint	Calculated RPM	Actual RPM
100	100	99
200	200	199
300	300	298
360	360	356

Table 1

These results from the four simulated tests proves that the PWM output from the program is accurate with a small % error. With the simulated test being a success, the practical build can be done.

5.3 PRACTICAL TEST

The practical test can be seen in **Figures 5-9**.

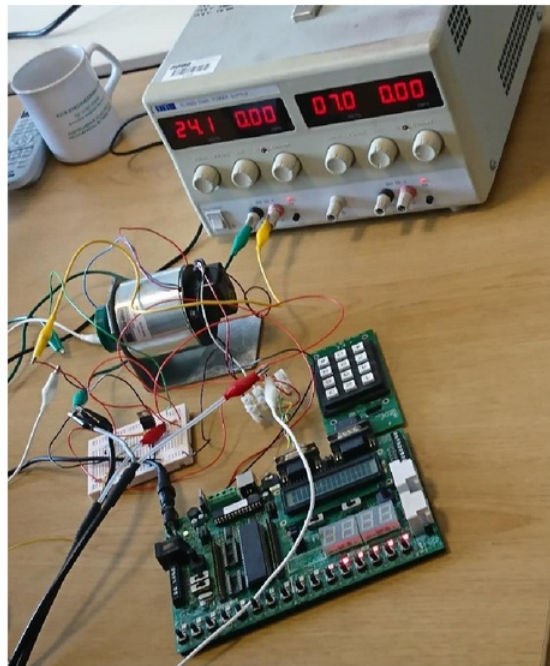


Figure 5-6

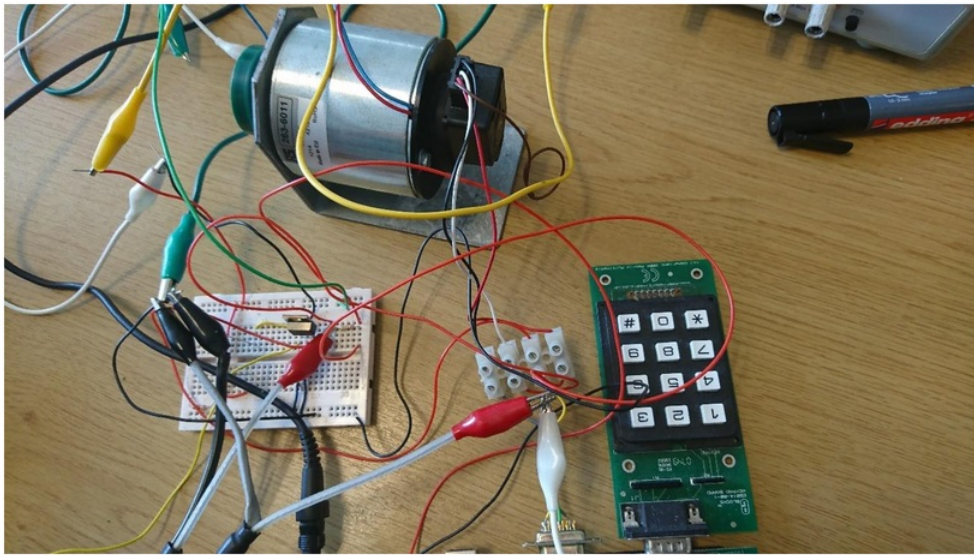


Figure 5-7

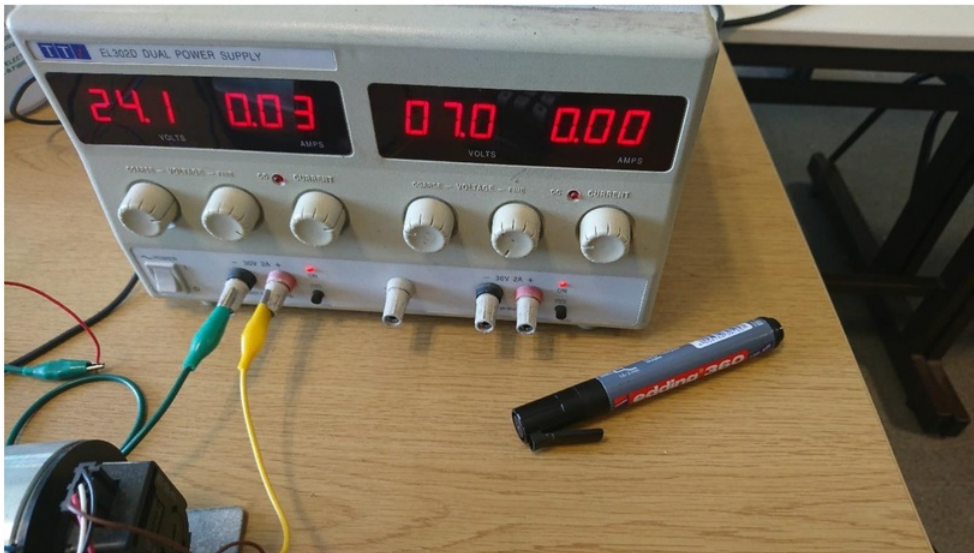


Figure 5-8

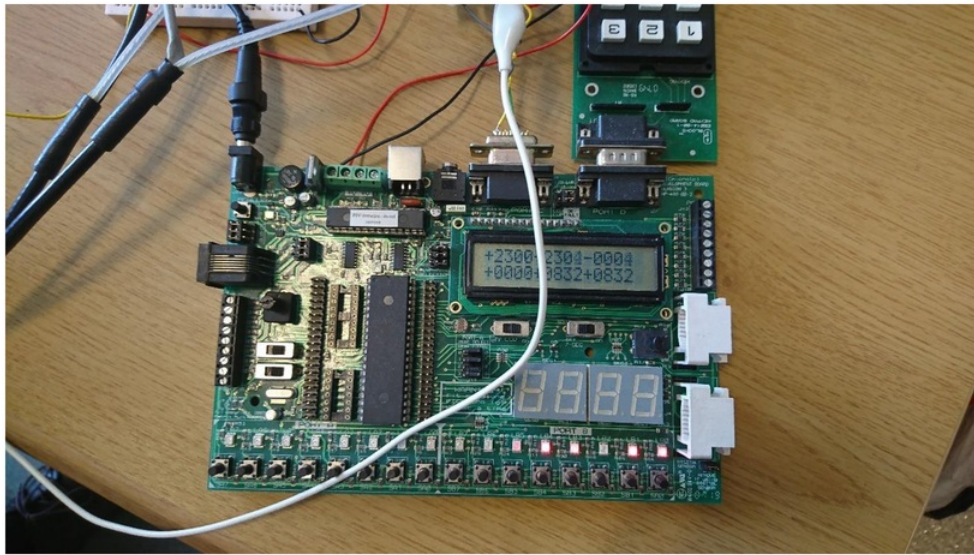


Figure 5-9

Once set up and programmed, the PWM and setpoint waveforms were measured using a Picoscope and documented for later comparison. The frequency, duty cycle and rpm were also calculated and documented to check that the waveform is output for the setpoint speed.

5.3.1 Practical Test 1 (200 rpm setpoint)

The Setpoint waveform for 200 rpm can be seen below.

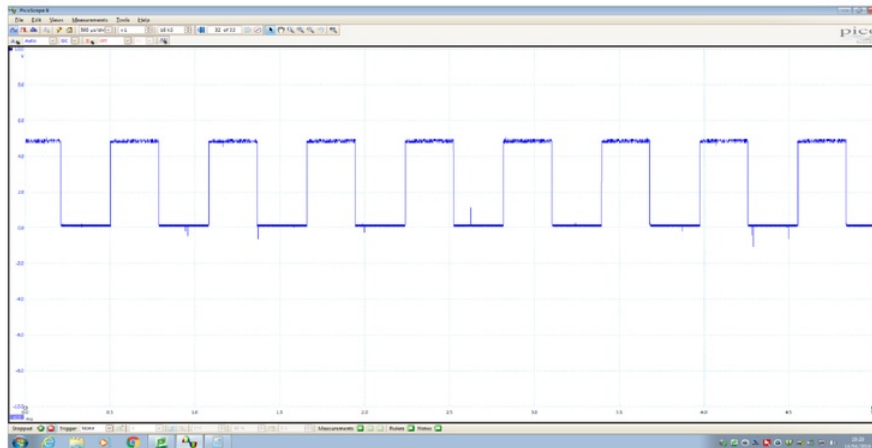


Figure 5-10

1.2 division at 0.5ms per division = 0.0006

$$\text{Frequency} = \frac{1}{0.0006} = 1666\text{Hz}$$

$$\text{Calculated rpm} = 1666 \times \left(\frac{60}{500}\right) = 200$$

The PWM waveform for 200 rpm can be seen below.

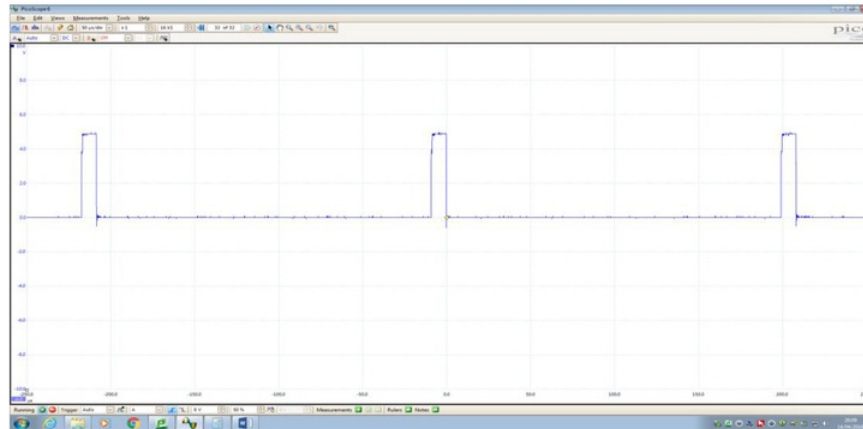


Figure 5-11

$$\text{Duty cycle} = 204 \times \left(\frac{100}{1023} \right) = 4.7\%$$

5.3.2 Practical Test 2 (1000 rpm setpoint)

The setpoint waveform for 1000rpm can be seen below.

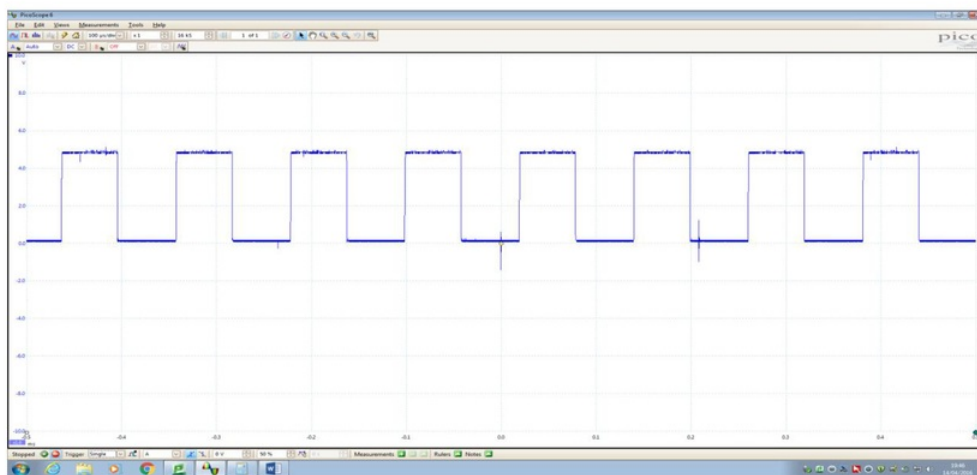


Figure 5-12

$$1.2 \text{ divisions at } 0.1\text{ms per division} = 0.00012$$

$$F = \frac{1}{0.00012} = 8333\text{Hz}$$

$$\text{Calculated rpm} = 8333 \times \left(\frac{60}{500} \right) = 999.6 \text{ rpm}$$

The PWM output waveform can be seen below.

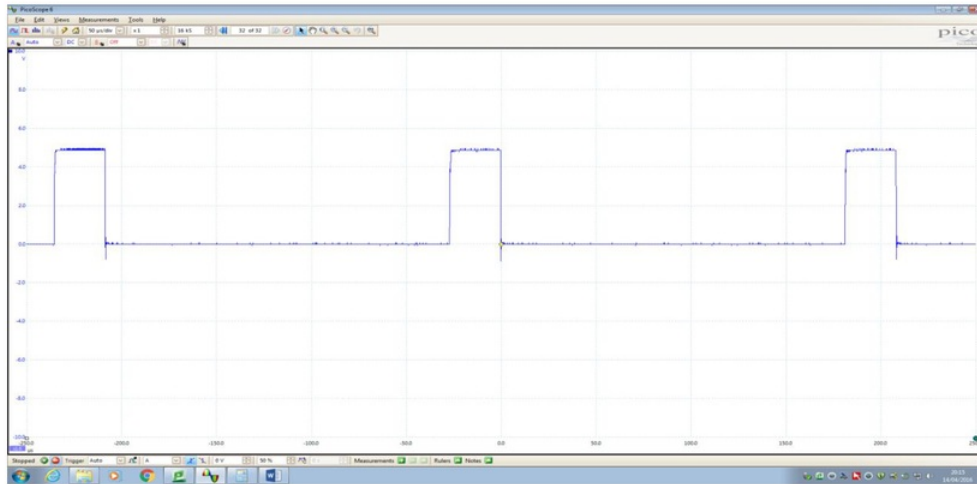


Figure 5-13

$$\text{Duty cycle} = 1003 \times \frac{100}{1023} = 13.3\%$$

5.3.3 Practical Test 3 (2000 rpm setpoint)

The setpoint output waveform can be seen below.

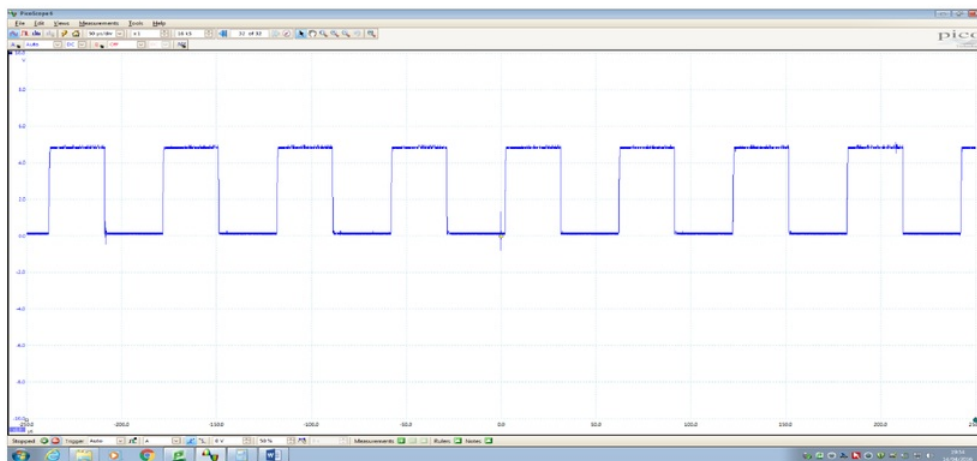


Figure 5-14

$$1.2 \text{ divisions at } 0.05\text{ms per division} = 0.00006$$

$$\text{Frequency} = \frac{1}{0.00006} = 16666\text{Hz}$$

$$\text{Calculated rpm} = 16666 \times \left(\frac{60}{500}\right) = 1999.2 \text{ rpm}$$

The PWM output waveform can be seen below.

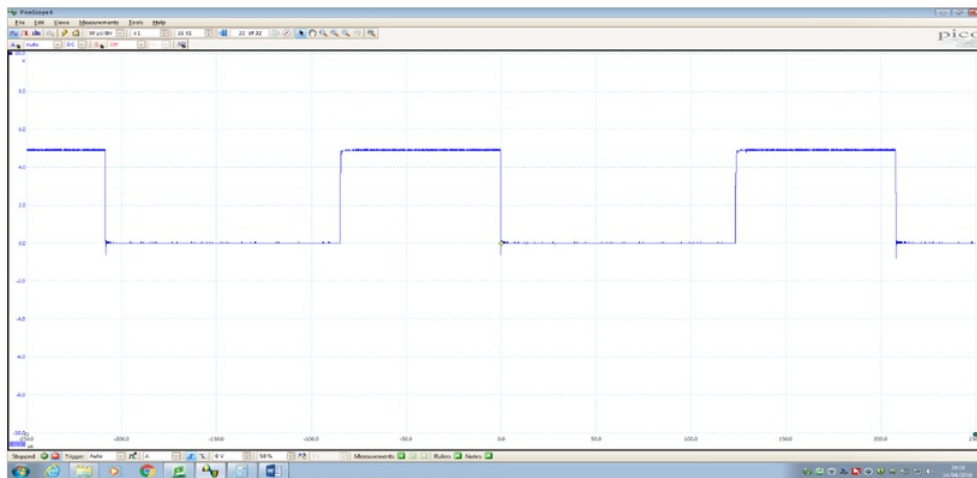


Figure 5-15

$$\text{Duty cycle} = 2001 \times \left(\frac{100}{1023} \right) = 41.4\%$$

5.3.4 Practical Test 4 (2350 rpm setpoint)

The setpoint output waveform can be seen below.

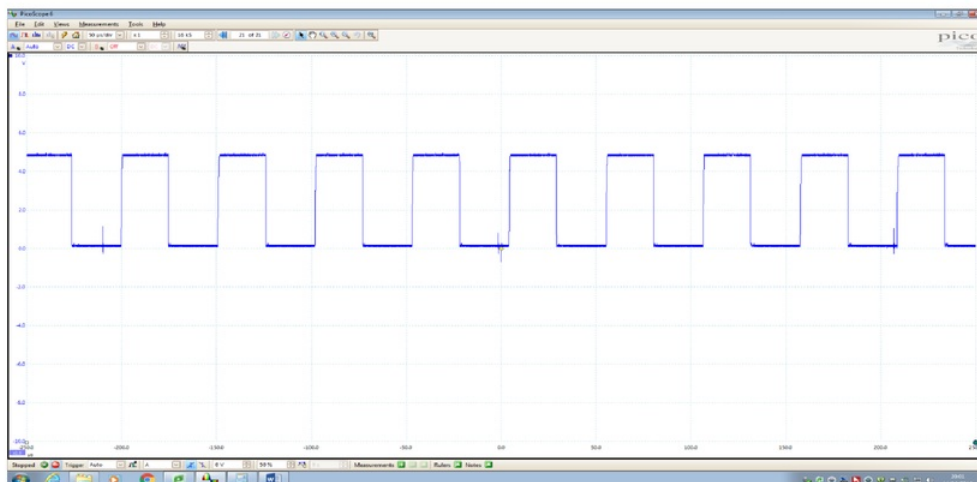


Figure 5-16]

$$1 \text{ division at } 0.05\text{ms per division} = 0.00005$$

$$\text{Frequency} = \frac{1}{0.00005} = 20000\text{Hz}$$

$$\text{Calculated rpm} = 20000 \times \left(\frac{60}{500} \right) = 2400 \text{ rpm}$$

The PWM output waveform can be seen below.

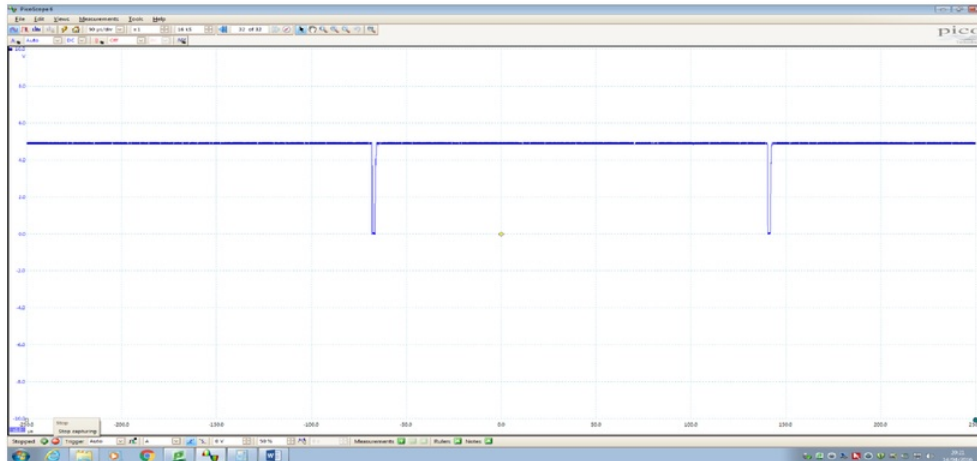


Figure 5-17

$$\text{Duty cycle} = 2349 \times \left(\frac{100}{1023} \right) = 99.6\%$$

Practical tests look good and the images of the setup give reference to the tests results.

5.4 PRACTICAL TEST SUMMARY

The practical tests and calculations have proved that the program produces an accurate PWM output so that the setpoint is met. The specification provided by the assignment brief was met and therefore this system is a success.

The test for system performance and with external disturbances should be included here.

6 DISCUSSION AND CONCLUSIONS

The specification for this assignment stated that a system must be created to control the speed of a DC motor by using a microcontroller. A setpoint was to be input through the use of a keypad, which would be displayed on an LCD display. This setpoint would be used to increase or decrease the speed of the DC motor until the setpoint was reached.

The motor was required to be started and stopped through the use of pushbuttons. These buttons proved to work in the practical test.

The speed, error, setpoint, proportional, integral accumulation and PWM output were required to be displayed on the LCD screen. This was achieved and proved within the practical test. The keypad interface operated as intended, and the PI controller calculated the error to a high degree of accuracy, so that the PWM output was able to accurately control the speed of the motor.

The use of a microcontroller along with a development board allowed for easy testing and design. Future alterations and improvements could easily be made to this system as the program can easily be altered and the microcontroller reprogrammed. A possible improvement to the system could be to improve the controller to have a derivative aspect, making it a PID controller.

Another possible change to the program would be to have the motor be able to alternate rotary direction.

The use of Proteus to simulate the system allows for testing before having to do a practical build. This saves time, and potentially money on development as changes to components can be done through a series of clicks rather than having to actually obtain replacement parts for a practical build. This allows for efficient design and testing.

All aims and objectives for this assignment were achieved by designing, creating and testing the specified system, therefore making this assignment a success.

*Concl
usion
makes
sense
with*

*some
good
points
made
for
recom
menda
tions
for
progre
ss.*

*The
discus
sion
could
have
includ
ed the
syste*

7 TEAM WORKING

This assignment was almost entirely done as an individual, from creating the code to simulated testing. However, to save time and be more efficient, the practical build was done as a group of 4 with the assistance of a lecturer. The testing of the practical build was also done as a group; however, a large variety of results were taken to increase diversity of our reports.

8 APPENDICES

PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers. (1998). 1st ed. [ebook] Chandler: Microchip Technology, pp.1,6,7,8,16. Available at:
<http://ww1.microchip.com/downloads/en/DeviceDoc/30292D.pdf> [Accessed 10 May 2018].

HEDM-55xx/560x & HEDS-55xx/56xx Quick Assembly Two and Three Channel Optical Encoders. (2014). 1st ed. [ebook] San Jose: Avago Technologies, pp.1,2. Available at:
<https://www.mouser.co.uk/ProductDetail/Broadcom-Avago/HEDS-5540A06?qs=RuhU64sK2%252buQS1Gokilejg==> [Accessed 12 May 2018].

Kernighan, B. and Ritchie, D. (2011). The C programming language. Englewood Cliffs, N.J.: Prentice-Hall.

Perryman, C. (2012). Microcontrollers. Delhi: Research World.

Sam Harries - Assignment 2

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

67/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

Text Comment. Good introduction. The development method using prototyping should have been included here.

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

Text Comment. Good description of hardware needed for the task.

PAGE 11

PAGE 12

Text Comment. Good explanation.

PAGE 13

PAGE 14

PAGE 15

PAGE 16

Text Comment. The 10 bit PWM resolution is explained well.

Text Comment. Configuration explanation is very good.

Text Comment. Good explanation of how the ISR operates within the system.

Text Comment. This is explained clearly.

Text Comment. Explanation is clear and makes sense.

Text Comment. Clear explanation of how the keypad encoder works for 4 figure values.

Text Comment. worth noting there is a latch here which prevents debounce of the keypress.

Text Comment. The output / 4 was for testing and the final system used the 10 bit resolution. This should be stated here. The explanation of the code here is correct.

Text Comment. Explanations are good here.

Text Comment. Good explanation that explains the function. Worth pointing out that this is called from the while loop and returns the key pressed.

Text Comment. This is the result of the prototype developments.

Text Comment. The tests carried out are good.

Text Comment. Practical tests look good and the images of the setup give reference to the tests results.

Text Comment. The test for system performance with external disturbances should be included here.

Text Comment. Conclusion makes sense with some good points made for recommendations for progress. The discussion could have included the system response to external disturbances.

GRADING FORM: CSG-RUBIC

DAVID HARRIES**0****RESEARCH-INFORMED LITERATURE** n/a**KNOWLEDGE AND UNDERSTANDING**

 The intro points out the task well. This should also include the development method of prototyping which is used to develop the final system. The prototypes include PWM driver, keypad encoder and optical encoder feedback input. 5.5/10 A good description of the hardware needed. 4/5 The design stages using prototypes have not been included and the development has jumped to the final design. However the stages are explained in the final design. The configuration is explained well. 13/20 The software design is explained well, but should be noted that the software was developed using prototype stages before combining it all to the final system. The code explanation is good. 6.5/10

29/40**ANALYSIS**

 Tests are good for CAD and development board. The system performance test with external

12/20

disturbance should have been included. 6.5/10
Conclusion makes sense and there are some good points made. The discussion should have included the system response with disturbances. 5.5/10

PRACTICAL APPLICATION AND DEP.



The only criticism is the lack of explanation of the prototyping stages. The development went well and some good tests results prove this. 6.5/10 Quality of work is good with good attention to detail in parts. 6.5/10

13/20

SKILLS FOR PROFESSIONAL PRAC.



The report is good and structured well. 6.5/10 Team working is good with good leadership skills. 6.5/10

13/20