

Sam Harries Assignment 1 - Final Version

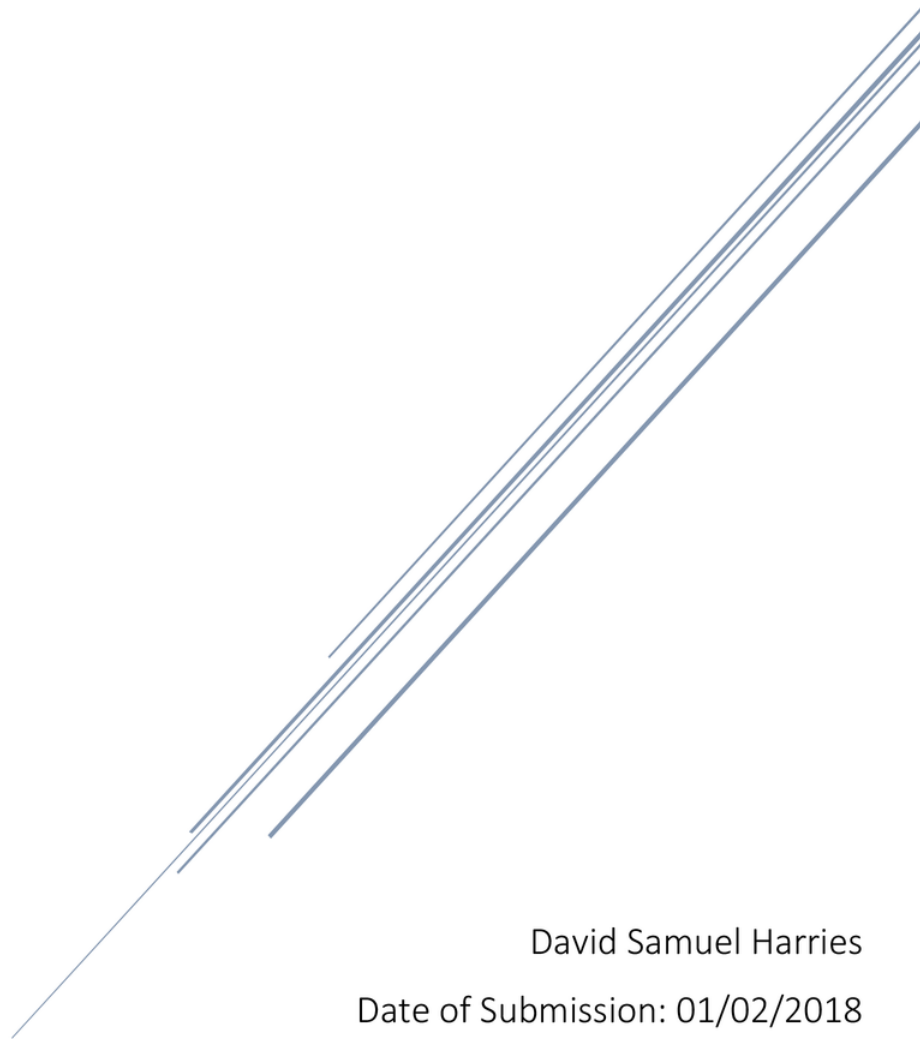
by David Harries

FILE	7309_DAVID_HARRIES_SAM_HARRIES_ASSIGNMENT_1_-_FINAL_VERSION_139109_1130522794.DOCX (11M)		
TIME SUBMITTED	01-FEB-2018 09:11PM (UTC+0000)	WORD COUNT	9347
SUBMISSION ID	82683874	CHARACTER COUNT	58224

HND Electrical and Electronic Engineering

PSDE5013: Configuration and Programming of Embedded
Systems

Assignment 1: Calendar Alarm Clock



David Samuel Harries

Date of Submission: 01/02/2018

Word Count: 9347

TABLE OF CONTENTS

1	Introduction	5
2	Research.....	7
2.1	PIC 16F877 Microcontroller	7
2.2	HP488 Development Board.....	10
3	Design.....	12
3.1	Interrupt Timer	13
3.2	Clock Program	18
3.3	Pre-processors, Function Prototypes and Global Variables.....	21
3.4	Interrupt ISR.....	23
3.5	Main Program	24
3.5.1	Configuration	24
3.5.2	Super-loop and Alarm Conditions	25
3.5.3	Case 0	26
3.5.4	Case 1 – Setting Hour	27
3.5.5	Case 2 – Setting Minute	27
3.5.6	Case 3 – Setting Day	28
3.5.7	Case 4 – Setting Month	28
3.5.8	Case 5 – Setting Date	29
3.5.9	Case 6 – Setting Year.....	30
3.5.10	Case 7 – Saving Values	30
3.5.11	Case 8 – Setting Alarm	31
3.5.12	Case 9 – Setting Alarm Hours.....	32
3.5.13	Case 10 – Setting Alarm Minutes	32
3.6	TimeDisplay and SetTimeDisplay Functions	33
3.7	Delay Function	35
3.8	StateDisplay Function	35
3.9	DateDisplay and SetDateDisplay Functions	35
3.10	AlarmTimeDisplay and SetAlarmTimeDisplay Functions	39
4	Testing and Analysis of Results	40
4.1	Timer Simulation Tests.....	40
4.2	Alarm Clock Simulation Tests.....	41
4.3	Practical Tests for the Alarm Clock	52
5	Conclusion and Recommendations	55
6	Appendices.....	56

Table of Figures

Figure 1.a.....	5
Figure 1.b.....	5
Figure 1.c.....	6
Figure 2.a – PIC 16F84 Pinout Diagram (PIC16F84 18-pin Enhanced FLASH/EEPROM 8-bit Microcontroller Datasheet, 2001)	7
Figure 2.b – PIC 16F877 Pinout Diagram (PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)	7
Figure 2.c – PIC 16F84 Internal Architecture (PIC16F84 18-pin Enhanced FLASH/EEPROM 8-bit Microcontroller Datasheet, 2001)	8
Figure 2.d – PIC 16F877 Internal Architecture (PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)	9
Figure 2.e	10
Figure 2.f	11
Figure 3.a.....	12
Figure 3.b - PIC 16F877 Internal Architecture (PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)	13
Figure 3.c - PIC 16F877 Internal Architecture (PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)	14
Figure 3.d - PIC 16F877 Internal Architecture (PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)	15
Figure 3.e - PIC 16F877 Internal Architecture (PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)	15
Figure 3.f	17
Figure 3.g.....	17
Figure 3.h	18
Figure 3.i	20
Figure 3.j	21
Figure 3.k.....	23
Figure 3.l	24
Figure 3.m	25
Figure 3.n	26
Figure 3.o	27
Figure 3.p	27
Figure 3.q	28
Figure 3.r	29
Figure 3.s.....	29
Figure 3.t	30
Figure 3.u	31
Figure 3.v.....	31
Figure 3.w.....	32
Figure 3.x.....	32
Figure 3.y.....	33
Figure 3.z.....	34
Figure 3.aa.....	35
Figure 3.bb	35

Figure 3.cc	36
Figure 3.dd	36
Figure 3.ee	37
Figure 3.ff	37
Figure 3.gg.....	38
Figure 3.hh	38
Figure 3.ii.....	39
Figure 4.a.....	40
Figure 4.b	40
Figure 4.c.....	41
Figure 4.d	41
Figure 4.e	44
Figure 4.f	44
Figure 4.g.....	44
Figure 4.h	44
Figure 4.i	44
Figure 4.j	45
Figure 4.k.....	46
Figure 4.l	47
Figure 4.m	47
Figure 4.n	48
Figure 4.o	48
Figure 4.p	48
Figure 4.q	48
Figure 4.r	48
Figure 4.s.....	48
Figure 4.t	49
Figure 4.u	49
Figure 4.v.....	49
Figure 4.w.....	49
Figure 4.x.....	49
Figure 4.y.....	49
Figure 4.z.....	50
Figure 4.aa.....	50
Figure 4.bb	50
Figure 4.cc	50
Figure 4.dd	51
Figure 4.ee	51
Figure 4.ff	52
Figure 4.gg.....	53
Figure 4.hh	53
Figure 4.ii.....	54

Table of Tables

Table 1.....	14
Table 2.....	16
Table 3.....	16

1 INTRODUCTION

The aims and objectives of this assignment are to prove an understanding of the design and development of an embedded system based on a microcontroller using configurable on-chip peripherals.

The system that will be created in this assignment is a calendar alarm clock and must follow the specification stated in the assignment brief. The specification states:

- The date and time must be set after pressing button 2. The user should then be able to increment the hours with each press of button 3. Another press of button 2 should enable the user to increment the minutes with button 3. Each consecutive press of button 2 should then enable the user to change the day, date, month and year. If button 1 is pressed in any of these states, then the display must return to its original state while also saving any changed values. **Figure 1.a** shows an example of this display.



Figure 1.a

- An alarm function must also be part of the system. This means that when button 1 is pressed, the user must be able to set a specific time, which when reached sets off an alarm. The alarm must be enabled through the use of button 3, which will also act as a deactivation button. While the alarm is being configured the word ALARM must also be shown on the display showing the user that the alarm is being configured. **Figure 1.b** shows an example display of the system in alarm configuration mode.



Figure 1.b

Figure 1.c shows the system in alarm configuration mode. The “!” symbol represents the alarm being on.



Figure 1.c

The system will be designed using a PIC 16F877 microcontroller which is found on the HP488 development board. These will both be discussed in ***Section 2.***

2 RESEARCH

2.1 PIC 16F877 MICROCONTROLLER

The PIC 16F877 is a powerful RISC CPU which has a high performance and has an internal clock of 20MHz. RISC stands for reduced instruction set computer which is a microprocessor designed to perform a smaller number of operations than a typical computer so that it can execute these operations at a higher speed.

A similar microprocessor is the PIC 16F84, however the PIC 16F877 is much more powerful. This can be seen by comparing the specifications found on their respective datasheets.

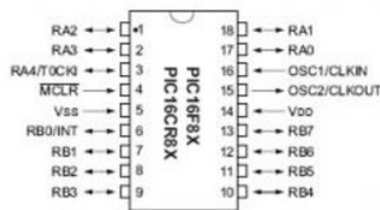


Figure 2.a – PIC 16F84 Pinout Diagram (PIC16F84 18-pin Enhanced FLASH/EEPROM 8-bit Microcontroller Datasheet, 2001)

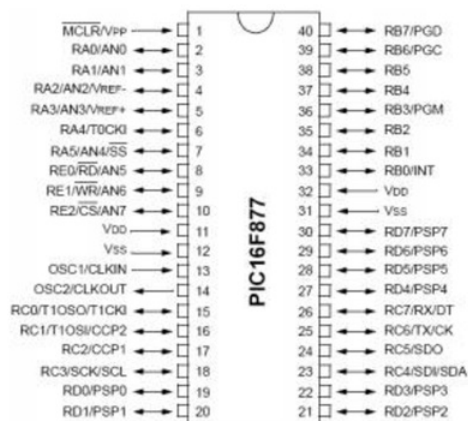


Figure 2.b – PIC 16F877 Pinout Diagram (PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)

Just by looking at the two pinout diagrams it can be seen that the PIC 16F877 has 40 pins in comparison to the PIC16F84's 18. This enables the PIC 16F877 to have more inputs and outputs. The majority of these pins are I/O ports. The PIC 16F877 has a total of five different I/O ports whereas the PIC 16F84 only has two I/O ports.

A second comparison that can be made by looking at the internal architecture of the two microcontrollers. These can be seen on the following pages.

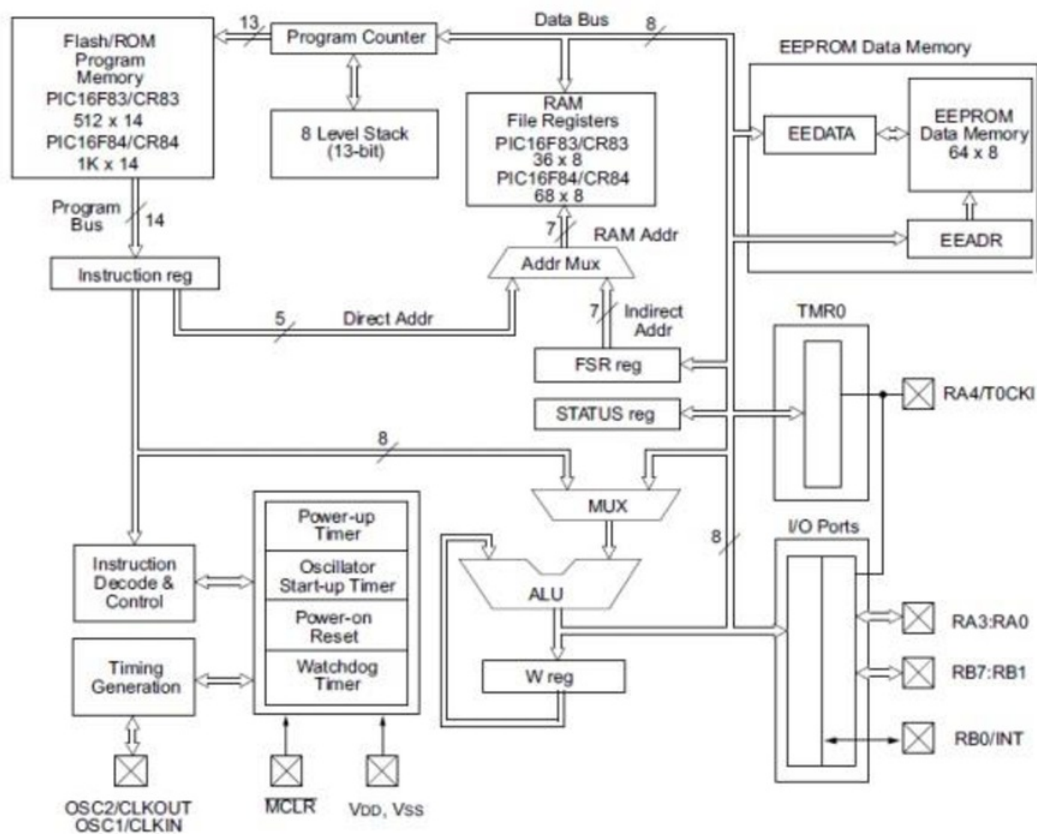


Figure 2.c – PIC 16F84 Internal Architecture (PIC16F84 18-pin Enhanced FLASH/EEPROM 8-bit Microcontroller Datasheet, 2001)

Device	Program FLASH	Data Memory	Data EEPROM
PIC16F874	4K	192 Bytes	128 Bytes
PIC16F877	8K	368 Bytes	256 Bytes

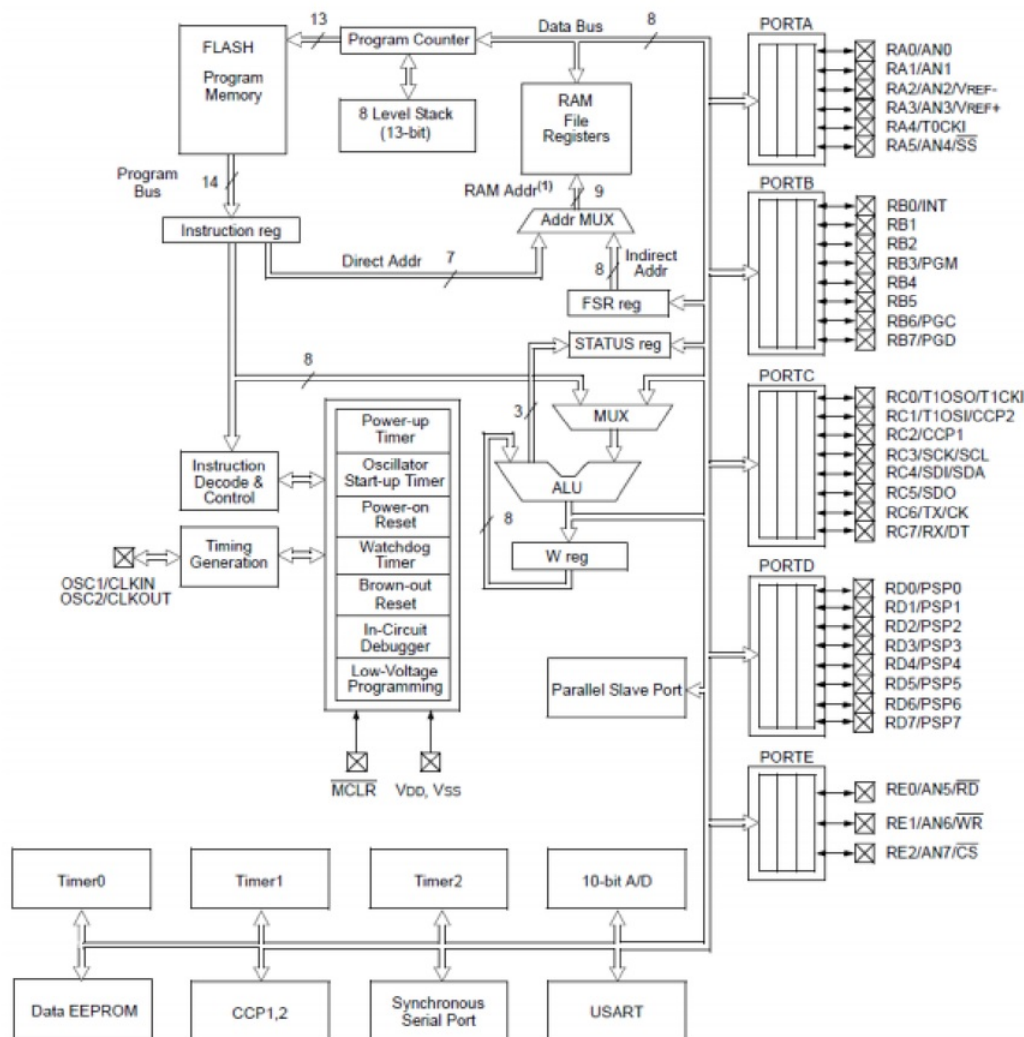


Figure 2.d – PIC 16F877 Internal Architecture (PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)

The PIC 16F877's internal architecture is much more complex than that of the PIC 16F84. As seen in the **Figure 2.d**, the data memory of the PIC 16F877 is 368 bytes compared to the PIC 16F84's 192 bytes. The data EEPROM and the Program FLASH of the PIC 16F877 is double that of the PIC 16F84's. There are far more devices within the PIC 16F877 along with the extra I/O ports.

In both chips each device is connected to the internal bus system and can be programmed by the CPU. The PIC 16F877 has a number of additional modules within its internal architecture which include a 8 channel ADC module (10 bit), a CCP module, a synchronous serial port, a PSP module and a USART module. In addition to these modules the PIC 16F877 has three timer modules

whereas the PIC 16F84 only has one. All of these factors make the PIC 16F877 a very suitable chip to use for this system.

2.2 HP488 DEVELOPMENT BOARD

The HP488 development board will be used to test the system once it's been designed. The board can be seen in **Figure 2.e**.

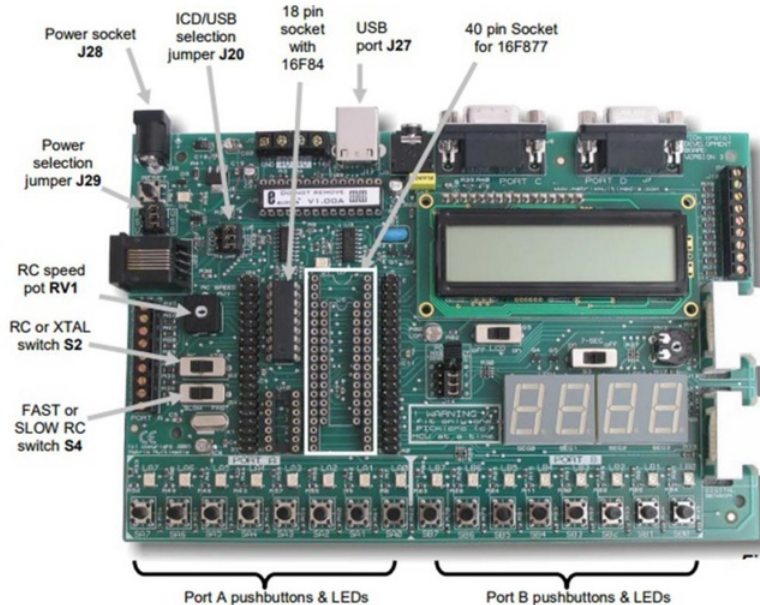


Figure 2.e

The HP488 board has a number of on board components. These include:

- 4*7 Segment Displays
- 2 Line LCD Display
- 16 LEDs
- 16 Push Buttons
- LDR for Analogue Input
- Potentiometer for Analogue Input
- USB Port
- ICD2 Socket
- Crystal Oscillator
- Reset Button

The PIC 16F877 chip can be seen in **Figure 2.e** and is connected to the peripherals listed above. For the system specified in the assignment brief, three buttons are required along with the 2 line LCD display. These requirements are fulfilled by this development board so therefore it is suitable hardware to test the system on. **Figure 2.f** shows a diagram from the HP488 datasheet which shows all of the external peripherals on the board.

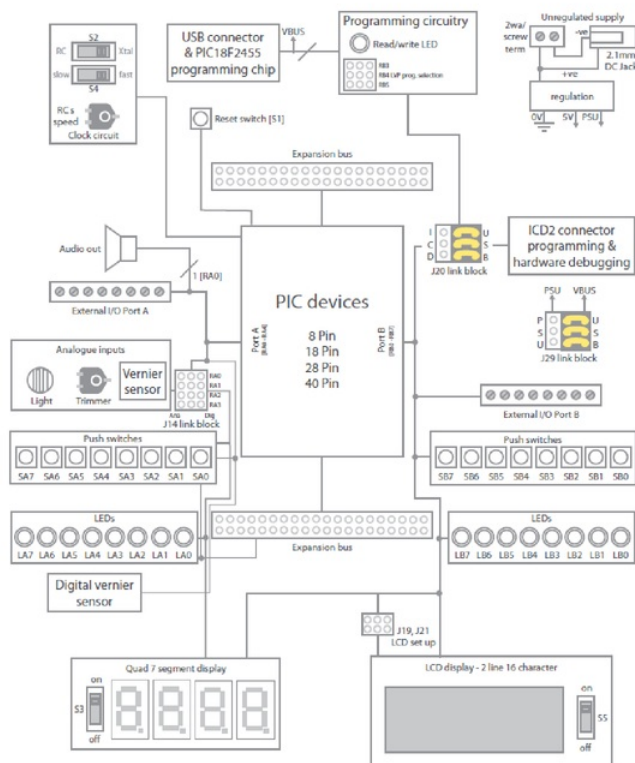


Figure 2.f

Real
son
able
rese
arch
and

goo
d
com
pari
son
of
MC
U's
with
limit
ations.

3 DESIGN

In order to design the program that will operate the microcontroller a block diagram must be created. This diagram will detail how the states are navigated between and what each state of the program does.

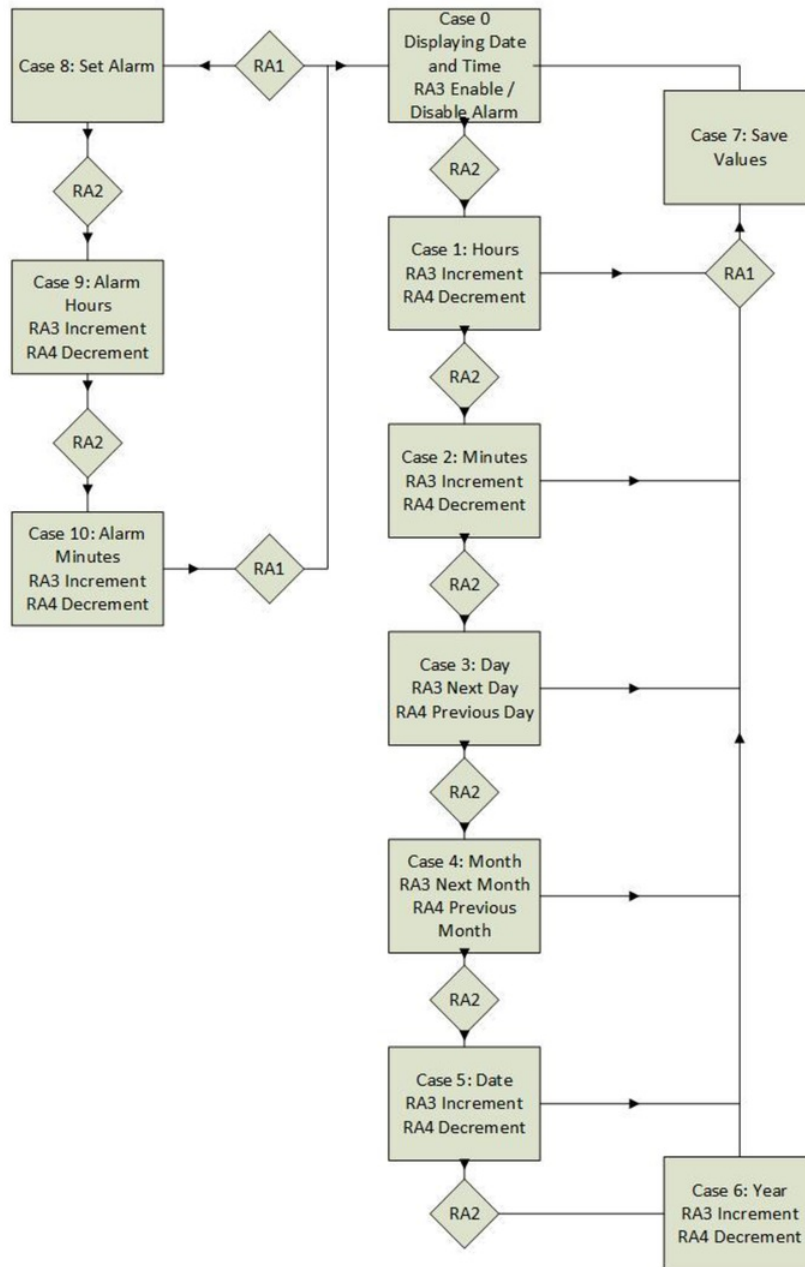


Figure 3.a

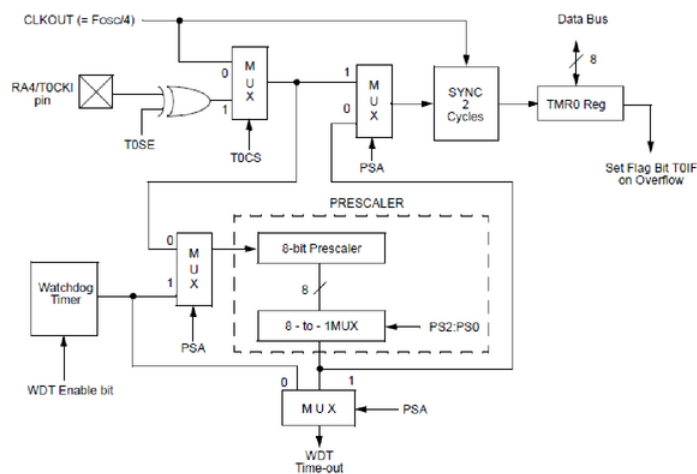
Good
diagram.

Alongside these cases, an interrupt must also be made for the clock to operate. This interrupt will use a timer which will count each second. Every minute, an increment will be added onto the minutes on the clock. When the minutes exceed 59, an increment will be added onto the hours and the minutes will be reset back to 0. When the hours exceed 23, the clock will move on to the next day, changing the day's name, and the date. For each month, when the date reaches 28 (February or 29 for a leap year), 30 (September, April, June, and November) or 31 (January, March, May, July, August, October and December), the month will change and the date will be reset back to 0. Finally, when December ends, the month will loop back around to January and an increment will be added onto the year.

3.1 INTERRUPT TIMER

In order for the alarm clock to maintain an accurate time, an ISR (Interrupt Service Routine) must be made. ISR stands for "Interrupt Service Routine." "An ISR (also called an interrupt handler) is a software process invoked by an interrupt request from a hardware device. It handles the request and sends it to the CPU, interrupting the active process. When the ISR is complete, the process is resumed." (Techterms.com, 2018). This is an ideal method of keeping the timer accurate as the interrupt will always execute its code regardless of what the program is currently doing.

The timer for this circuit must be placed within the ISR so that whenever the criteria for the interrupt to be executed are met, it will update the time on the clock. There are multiple ways of creating an interrupt for the PIC 16F877. For this system TIMER0 will be used for the ISR.



Good explanation.

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

Figure 3.b - PIC 16F877 Internal Architecture (PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)

Figure 3.b shows the block diagram for TIMER0 found on the PIC 16F877 datasheet. To get TIMER0 to operate, two registers must be set up. These registers are INTCON, and OPTION_REG. To set up the OPTION_REG register, the datasheet must be referred to.

	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	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7	bit 0							
bit 7	RBP : 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 (CLKOUT)							
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				
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.c - PIC 16F877 Internal Architecture (PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)

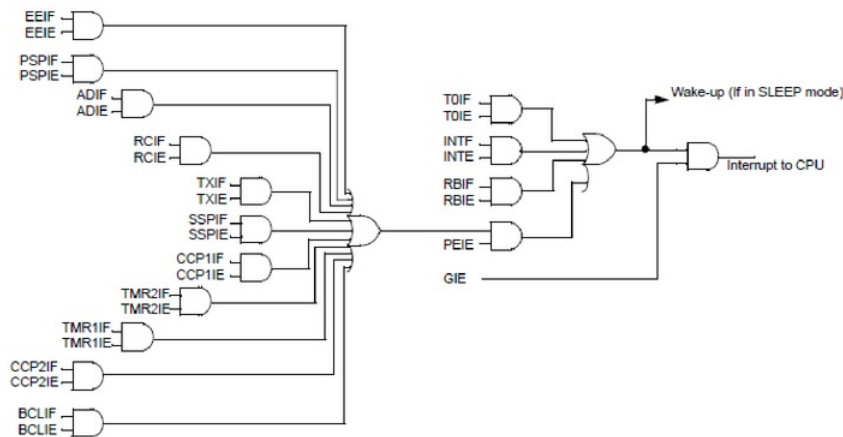
A line of code written in C such as `OPTION_REG = 0x87` would configure `OPTION_REG` in such a way that can be seen in the following table.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
RBP	INT	TOCS	TOSE	PSA	PS2	PS1	PS0
1	0	0	0	0	1	1	1

Table 1

Table 1 shows that each bit in has been assigned a binary value from the line of code previously stated. Using the datasheet it can be seen that `RBP` (Bit 7) determines whether pull-up resistors are enabled or disabled. Since it has been assigned 1, pull-up resistors have been disabled. `INTEDG` (Bit 6) determines whether the interrupt happens on the rising edge or the falling edge of the clock pulse. Since it has been assigned 0 it means that the interrupt will operate on the falling edge of the clock pulse. `TOCS` (Bit 5) determines the clock source. Since the 5th bit is 0, the register is told to operate using the internal instruction clock (CLKOUT). `TOSE` (Bit 4) determines whether the interrupt should increment on high-to-low transition or low-to-high transition on RA4/T0CKI pin. Being assigned to 0 makes it operate on a low-to-high transition. `PSA` (Bit 3) is the prescaler assignment bit. When 0 the prescaler is assigned to the `TIMER0` module which is what is needed for the timer to operate as an interrupt in this system. The final 3 bits (`PS2`, `PS1` and `PS0`) determine the prescaler rate. Using the table shown in Figure 3.c it can be seen that when assigned to 111 respectively, the `TMR0` prescaler rate is 1:256.

The `INTCON` register must also be configured in order for `TIMER0` to act as an interrupt. The diagram shown in **Figure 3.d** shows the interrupt logic diagram for the `INTCON` register. This shows the logic behind what criteria must be met for the CPU to be interrupted. **Figure 3.e** shows what each Bit represents.



The following table shows which devices have which interrupts.

Device	TOIF	INTF	RBIF	PSPIF	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	EEIF	BCLIF	CCP2IF
PIC16F876/873	Yes	Yes	Yes	—	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
PIC16F877/874	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Figure 3.d - PIC 16F877 Internal Architecture (PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)

	R/W-U	R/W-U	R/W-U	R/W-U	R/W-U	R/W-U	R/W-x	R/W-x
	GIE	PEIE	TOIE	INTE	RBIE	TOIF	INTF	RBIF
bit 7								
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	TOIE: 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	TOIF: 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							
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.e - PIC 16F877 Internal Architecture (PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, 1998)

Using the table shown above, when TOIE (TIMER0 Overflow Enable) along with GIE (Global Interrupt Enable) the CPU will be interrupted whenever a math overflow from TOIF (TIMER0 Overflow Interrupt Flag). This means that when TIMER0 produces a count of over 256 (Overflow) TOIF is set to a logic 1. TOIE is already set to 1 as previously stated, which means that the AND gate

sends a 1 through to the 3 input OR gate seen above. This then sends a 1 to the final AND gate. Since GIE was set to a logic 1 earlier, the AND gate has a 1 on both inputs and sends an interrupt to the CPU.

Using the datasheet, a configuration for INTCON can be made that will allow TIMERO to operate as an interrupt. **Table 2** shows the configuration.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
GIE	PEIE	TOIE	INTE	RBIE	TOIF	INTF	RBIF
1	0	1	0	0	0	0	0

Table 2

The configuration used in **Table 2** is INTCON = 0xA0; which has everything except GIE and TOIE set to 0.

With the two registers set up, the timing calculations can be done and the code can be written. The PIC 16F877 microcontroller has an internal oscillation frequency of 19.668MHz. With 4 pulses per instruction:

$$F_{osc} = \frac{1966800}{4} = 4915200Hz$$

The system overflows at 256, therefore the value needs to be 256 or under. In order to get an overflow of 10ms, the prescaler is used. With the prescaler set to the max value of 1:256, the value remains <256. This can be seen from **Table 3**.

BIT Value	TMRO Rate	49152 / TMRO Rate	< 256 Yes/No
000	1 : 2	24576	No
001	1 : 4	12288	No
010	1 : 8	6144	No
011	1 : 16	3072	No
100	1 : 32	1536	No
101	1 : 64	768	No
110	1 : 128	384	No
111	1 : 256	193	Yes

Table 3

To calculate the instruction cycle time of 4915200Hz, the inverse of Fosc must be taken:

$$Instruction\ Cycle\ Time = \frac{1}{4915200} = 0.2\mu s$$

$$Timer\ Increment\ Time = 0.2\mu s \times 256prescale = 51.2\mu s$$

$$Count\ for\ 10ms = \frac{10ms}{51.2\mu s} = 195$$

$$Accounting\ for\ error = 195 - 2 = 193$$

$$Preload\ Count: 256 - 193 = 63$$

The preload count value means that 63 needs to be preloaded into TMRO in order to get an interrupt of 10ms. With these calculations complete, the code can be written. The code can be seen in **Figure 3.f**

Good
explanation of
config.

```

#include <pic.h>

int milli=0, secs=0; //Must be declared global for function prototype variables

void interrupt ISR (void)
{
    if(TOIF==1)
    {
        TOIF = 0; //Reset interrupt flag, set to zero
        TMR0 = 63; //Reload preload time value.
        //Increment milliseconds by 10 to increase the value of the timer in steps of 10
        //If milli equals 100 increment sec by 1, clear milli = 0 and toggle RBO
        if(milli==100)
        {
            secs ++; //Increment seconds by 1 goes here
            milli = 0; //Reset milli to zero
            RBO = ~RBO; // Toggle RBO pin
        }
    }
}

void main(void) //Main Function
{
    TRISB = 0x00; //Configure port B = 0000 0000
    OPTION_REG = 0x87;
    INTCON = 0x00; //Start super-loop
    while(1)
    {
    }
}

```

Figure 3.f

As **Figure 3.f** shows, the source file for the pic is included in the first line. Following this two global variables are declared with the data type int. These are milli and secs which are both assigned to 0. The ISR is then declared with the name interrupt ISR. The first line of the interrupt is an if statement which states that if TOIF is equal to 1, the following code is executed. Within this if statement, TOIF is reset to 0 and TMR0 is preloaded with the calculated value of 63. The following lines are commented out as they are used to convert the 10ms interrupt into a 1 second interrupt. They work in the respect that milli is incremented in steps of 10 to increase the value of the timer in steps of 10. Then if milli is equal to 100, secs is incremented by 1. This works it takes 100, 10ms cycles to create 1 second. RBO is then toggled which will be used to see the 10ms, and 1 second clock pulse. The interrupt is then ended.

Following the interrupt is the main program which configures the pins of the PIC 16F877 chip. The first line is TRISB = 0x00; which configures the B ports to 0000 0000. This makes them all outputs. Following this are the register configurations derived earlier. Finally, a super-loop is created so that the program doesn't end. The super-loop is empty since the only thing this program does is create a clock pulse using the ISR.

With the program complete, it can be implemented into Proteus. **Figure 3.g** shows the circuit used to test the interrupt.

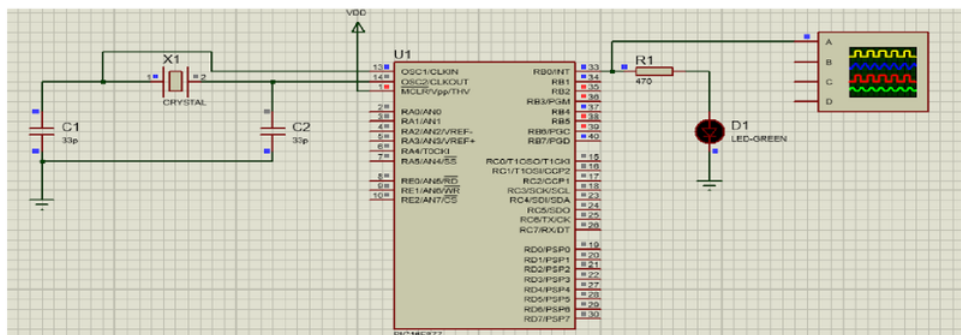


Figure 3.g

The testing for the timer can be found in **Section 4.1**.

3.2 CLOCK PROGRAM

With the timer created, the configurations for the registers and preload value can be used inside the clock program to accurately tell the time. The program is broken down into three files; main.c, LCD.c and LCD.h. LCD.h is the header file that includes function prototypes that are used in the main.c and LCD.c files. The code for LCD.h can be seen in **Figure 3.h**.

```
/*  
Project:- LCD Display  
File:- LCD.h  
  
Description:- These functions are compatible with Dave Scammell's  
*/  
  
#define CTRL_WR 0  
#define LCD_HOME_L1 0  
#define LCD_LINE1 0  
  
void LCD_Delay(int t);  
void LCD_Command(unsigned char cmd);  
void LCD_Data(unsigned char data);  
void LCD_Cursor(unsigned char x, unsigned char line);  
void LCD_Startup(void);  
void LCD_write(int ctrl, int posn);  
void DisplayString(int line, char * str);
```

Figure 3.h

The functions seen in LCD.h can be found in LCD.c. The code for LCD.c can be seen on the following pages.

```

/*
Project:- LCD Display
File:- LCD.c

Description:- These functions are compatible with Dave Scammell's

*/
#include <pic.h>
#include "LCD.h"

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

void LCD_Command(unsigned char cmd)
{
    unsigned char msn, lsn, output;

    RB4 = 0; //tell LCD we are sending a command
    msn = (cmd & 0xF0) / 16; //load most significant nibble
    output = (PORTB & 0xF0) | msn;
    PORTB = output; //send most significant nibble
    LCD_Delay(2);
    RB5 = 1; //enable LCD
    LCD_Delay(2);
    RB5 = 0;

    lsn = cmd & 0x0F;
    output = (PORTB & 0xF0) | lsn;
    PORTB = output;
    LCD_Delay(2);
    RB5 = 1;
    LCD_Delay(2);
    RB5 = 0;
    LCD_Delay(250);
}

void LCD_Data(unsigned char data)
{
    unsigned char msn, lsn, output;

    RB4 = 1; //tell LCD we are sending data
    msn = (data & 0xF0) / 16; //load most significant nibble
    output = (PORTB & 0xF0) | msn;
    PORTB = output; //send most significant nibble
    LCD_Delay(2);
    RB5 = 1; //enable LCD
    LCD_Delay(2);
    RB5 = 0;

    lsn = data & 0x0F;
    output = (PORTB & 0xF0) | lsn;
    PORTB = output;
    LCD_Delay(2);
    RB5 = 1;
}

```

```

    PORTB = output;
    LCD_Delay(2);
    RB5 = 1;
    LCD_Delay(2);
    RB5 = 0;
    LCD_Delay(250);
    RB4 = 0;
}

void LCD_Cursor(unsigned char x, unsigned char line)
{
    unsigned char y;
    if(line==0)
    {
        y = 0x80;
    }
    else
    {
        y = 0xC0;
    }
    LCD_Command(x + y);
}

void LCD_Startup(void)
{
    LCD_Delay(10000);
    LCD_Command(0x33);
    LCD_Command(0x32);
    LCD_Command(0x2C);
    LCD_Command(0x06);
    LCD_Command(0x0C);
    LCD_Command(0x01);
    LCD_Delay(5000);
    LCD_Command(0x02);
    LCD_Delay(5000);
}

void LCD_write(int ctrl, int posn)           //for compatibility with Dave Scammell
{
    LCD_Cursor(0,0);
}

void DisplayString(int line, char * str)     //for compatibility with Dave Scammell
{
    unsigned char i=0;

    LCD_Cursor(0,line);

    while(str[i]!=0)
    {
        LCD_Data(str[i]);
        i++;
    }
}

```

Figure 3.1

LCD.c contains the function definitions for the prototypes declared in LCD.h. The first function is LCD_Delay. This function is simply used as a delay in other functions. It works by using a local variable i and incrementing it continuously until it reaches the value of t. The value of t determines how long the delay lasts.

The next function in LCD.c is LCD_Command. This function is used to configure the ports on the PIC 16F877 and enables the LCD display. The output RB4 is used to tell the LCD that a command

is being sent to it. The output RB5 is used to enable LCD. Following this function is the LCD_Data function. This function is used to send data to the LCD. It uses RB4 and RB5 for the same functions.

The next function is LCD_Cursor. This function determines where each line lies on the Y axis on the LCD display. If line is equal to 0, then the line lies on the coordinates y= 0x80. Else, the line will reside on the coordinate y=0xC0. The following function is LCD_Startup which is used to configure the LCD display when the system is started up. Following this function is LCD_Write which is used as a function call for LCD_Cursor with the inputs 0 for ctrl and 0 for posn.

The final function is DisplayString which is used to actually determine while line the data is sent to on the LCD display. This function is used in the main code to send the time to the first line, and the day, date, month and year to the second line.

With the LCD.c and LCD.h files obtained, the main program can be written. The main.c is further broken down into various functions. These can be seen listed below:

- TimeDisplay
- SetTimeDisplay
- SetDateDisplay
- StateDisplay
- DateDisplay
- Delay
- AlarmTimeDisplay
- SetAlarmTimeDisplay

Along with these functions, an ISR is created in order to keep the time accurate. These will all be discussed in the following sections.

3.3 PRE-PROCESSORS, FUNCTION PROTOTYPES AND GLOBAL VARIABLES

Figure 3.j shows the Pre-processor declarations, Function Prototypes and global Variables.

```
//HEADER FILES
#include <pic.h>
#include "LCD.h"

//FUNCTION PROTOTYPES
void TimeDisplay(int line);
void SetTimeDisplay(int line);
void SetDateDisplay(int line);
void StateDisplay(int line);
void DateDisplay(int line);
void Delay(int t);
void AlarmTimeDisplay(int line);
void SetAlarmTimeDisplay(int line);

//GLOBAL VARIABLES FOR INITIAL VALUES OF TIME AND DATE
int milli=0, secs=0, mins=50, hours=23, state=0, days=1, month=12,
day_number=1, year=2018, alarm_h=0, alarm_m=0, alarm=0,

//GLOBAL VARIABLES FOR TEMPORARY VALUES TO SET TIME AND DATE
hours_temp=0, mins_temp=0, state_temp=0, day_temp=0, month_temp=0, days_temp=0,
year_temp=0, alarm_hours_temp=0, alarm_mins_temp=0;
```

Figure 3.j

The pre-processor declarations shown in **Figure 3.j** are the pic.h file and the LCD.h file. The pic.h file includes the internal architecture and operation of the PIC 16F877 and the LCD.h file contains the function prototypes used within the main.c file and the LCD.c file that control the LCD display.

Following the pre-processor declarations are the function prototypes. These declare what functions will be used within the program along with what data they will have as an input and output. The prototypes with void before their name return no data. However, the all of the functions have inputs. All the inputs are of the data type int and all except Delay take in the variable 'line'. This is because each of these functions requires an input value from 'line' to determine which line on the LCD display will be used. The Delay function has an input 't' which simply determines how long the delay lasts.

The final part of this section is the global variables. These have been split into two separate groups. The first group of global variables are the variables used to store the Time, Date and Alarm information. These include the milliseconds, seconds, minutes, hours, state, days, month, day_number (used to display the name of the day), year, alarm hours, alarm minutes and alarm enabler. Each of these variables is assigned an initial value so that when the system is turned on, each variable will have a starting point. Most of the variables are set to 0, however some are assigned numbers such as mins and hours in order to show a specific time whenever the system is turned on. The year is also set to 2018 so that the user doesn't have to increment 2018 times just to reach the current year.

The next set of global variables is used to store temporary values for the Time, Date and Alarm information. These are used in the functions that set the time, date and alarm. All of these variables are assigned the value 0 since they will only be used to change the time, date and to set the alarm.

Good explanation

3.4 INTERRUPT ISR

Following the pre-processor declarations, function prototypes and global variables is the ISR. This code can be seen in **Figure 3.k**.

```
//INTERRUPT
void interrupt ISR (void)
{
    if(TOIF==1)
    {
        TOIF = 0;                // reset interrupt flag, set to zero
        TMR0 = 255;              // reload preload time value.
        milli += 1;              // increment milliseconds
        if(milli==100)           // if milli equals 100
        {
            secs ++;             // increment seconds by 1
            milli = 0;           // reset milli to zero
            if(secs==60)         // if secs equals 60
            {
                mins ++;         // increment mins by 1
                secs = 0;         // set secs to zero
                if(mins==60)      // if mins equals 60
                {
                    hours ++;    // increment hours by 1
                    mins = 0;    // set mins to zero
                    if(hours==24) // if hours equals 24
                    {
                        days ++; // increment days by 1
                        day_number ++; // increment day_number by 1
                        hours = 0; // set hours to 0.
                        if(((month==1)||(month==3)||(month==5)||(month==7)||(month==8)||(month==10)||(month==12)) && (days>31))
                        { // if the month is equal to 1,3,5,7,8,10,12 AND days is greater than 31
                            days = 1; // set days to 1
                            month ++; // increment month by 1
                        } // end if
                        if((month==2) && (days>28) && (year%4 != 0))
                        { // if the month is equal to 2 AND days is greater than 28 AND the year is not divisible by 4
                            days = 1; // set days to 1
                            month ++; // increment month by 1
                        } // end if
                        if((month==2) && (days>28) && (year%4 == 0))
                        { // if the month is equal to 2 AND days is greater than 28 AND the year is a multiple of 4
                            days = 1; // set days to 1
                            month ++; // increment month by 1
                        } // end if
                        if(((month==4)||(month==6)||(month==9)||(month==11)) && (days>30))
                        { // if the month is equal to 2 AND days is greater than 29 AND the year is not a multiple of 4
                            days = 1; // set days to 1
                            month ++; // increment month by 1
                        } // end if
                        if(month>12) // if month greater than 12
                        {
                            year ++; // increment year by 1
                            month = 1; // set month to 1
                        }
                        if(day_number==7) day_number=0; // if day_number equals 7 then day_number=zero
                    } // end if
                } // end if
            } // end if
        } // end if
    } // end if
}
```

Figure 3.k

As previously stated, the interrupt is used to 'interrupt' the program to change the time regardless of what the program is currently executing. This allows the program to maintain an accurate time. This code uses the information gathered from the timer designed earlier to create an interrupt. The function definition shows that the interrupt returns no values and doesn't import any values with the use of the keyword void before and after the interrupt name.

The first line within the interrupt is an if statement that checks to see if TOIF is set to 1. If it is then it will execute the following code. TOIF becomes 1 whenever TIMER0 has an overflow. This will happen every 10 ms as shown in the tests for the timer found in **Section 4**. The code within this if statement resets TOIF back to 0, the preload time value is reloaded, and the global variable milli is incremented. This is once again followed by an if statement that states if milli is equal to 100, then the code following will be executed. This converts the 10ms clock pulse into a 1 second timer.

The next section of code increments 1 second, and resets milli back to 0. After 60 seconds have been incremented the next if statement becomes true. This executes the next section of code which increments minutes by 1 every 60 seconds. This pattern cascades from minutes to hours, hours to days (and day_number), and days to months. However, for the months there are multiple if statements.

These if statements are used to determine how many days should be incremented before the month is changed. Each month is represented with a number, 1 being January to 12 being December. The first if statement states that if the month is equal to 1,3,5,7,8,10,12 and days is greater than 31 the month is incremented and the days variable is reset back to 1. The second if statement states that if the month is equal to 2, days is greater than 28 and the year is not divisible by 4, days is reset to 1 and the month is incremented. This statement checks whether the month is February and the year is not a leap year. Therefore, the amount of days that should be in this month is 28. The next if statement is exactly the same except it increments when it is a leap year and the days are equal to 29. The final if statement that determines when to increment the month states that if the month is equal to 4, 6, 9 or 11 and days is greater than 30, the month will be incremented. This if statement represents the months that have 30 days.

The final increment is for the year when the month is greater than 12. The month is then looped back around to 1 (Jan) and a year is added on. Following this is the reset for the day_number. When day_number is equal to 7, it resets back to 0. This loops the days of the week continuously. The interrupt is then ended by closing each set of brackets.

3.5 MAIN PROGRAM

The next section of the code is the main program. This code contains the switch case statement that was illustrated earlier in **Section 3** in **Figure 3.a**. It also contains the configuration for the OPTION_REG and INTCON registers that enable the interrupt, the configuration for the I/O ports A, B and C.

The functions previously stated in the prototypes are called from within the main program. However they will be discussed in more detail within their respective sections. The code for the main program will be broken up into subsections for each case.

3.5.1 Configuration

The start of the main program contains the configurations for the I/O ports, ADCON1, OPTION_REG and INTCON registers, and TMRO. These can be seen in

```
void main(void)
{
    int time;

    TRISA = 0x1E;           // configure port A = 0001 1110
    TRISB = 0x00;           // configure port B = 0000 0000
    TRISC = 0x00;           // configure port C = 0000 0000
    ADCON1 = 0x07;          // configure ADCON1 = 0000 0111
    OPTION_REG = 0x87;      // configure OPTION_REG = 1000 0111
    INTCON = 0xA0;          // configure INTCON = 1010 0000
    TMR0=255;              // pre-loading TMRO
}
```

Figure 3.1

The first line is a declaration for a local variable of data type int named 'time'. This is followed by three lines which configure the I/O ports A, B and C using hexadecimal codes.

- TRISA = 0x1E; configures port A = 0001 1110
- TRISB = 0x00; configures port B = 0000 0000
- TRISC = 0x00; configures port C = 0000 0000

For port A, RA0 is assigned the least significant bit (LSB) which is 0. This pin will be used as an output for the buzzer. To configure this pin as an output it is assigned a 0. RA1 to RA4 will be used as inputs. The next four bits are all assigned 1 to make these pins outputs.

For port B, all of the pins act as outputs so RB0 to RB7 are all assigned 0 on their respective bits. Port C is not used in this system so these could be set to anything, or removed from the code. Following the I/O configuration is the register configurations that were derived previously.

ADCON1 is configured to enable the digital I/O on the PIC 16F877.

3.5.2 Super-loop and Alarm Conditions

The next part of the main program is the start of the super-loop, and alarm conditions. This can be seen in

```

LCD_Startup();           // function header, runs function in LCD.c
while(1)                 // start super-loop
{
    if(alarm==1)          // if alarm equals 1
    {
        if((hours==alarm_h)&&(mins==alarm_m)) // if the set alarm time is equal to the actual time
        {
            RA0 = 1;      // set output RA0 to 1
        }
    }
    if(alarm==0)          // if alarm equals zero
    {
        RA0 = 0;         // set output RA0 to zero
    }
}

```

Figure 3.m

The code shown in **Figure 3.m** starts with a function call for LCD_Startup which is found in LCD.c. This starts up and enables the LCD display to be configured. Following this is a while statement which starts the super-loop. This loop will continue to cycle until the system is powered down, and is interrupted by the ISR when the interrupt conditions are met.

The first section of code within the super-loop creates the conditions in which the alarm activates. This is done by using three if statements. The first sees if the global variable 'alarm' is equal to 1. If true, the program executes the next if statement which checks to see if the global variable hours is equal to alarm_h and mins is equal to alarm_m. Alarm_h and Alarm_m are set to a specific time in the function SetAlarmTimeDisplay. When the time is equal to the alarm time, the next code is executed. This sets RA0 to 1. When RA0 is set to 1 the pin acts as a source which will be connected to a buzzer which will act as an alarm.

The final if statement checks to see if alarm is equal to 0. If this is the case then the alarm does not go off and RA0 is set to 0. The following switch case statement uses the block diagram in **Figure 3.a** to design the states.

3.5.3 Case 0

The next section of code is the switch case statement which contains 10 cases. The first of which is case 0. This can be seen in *Figure 3.n*.

```
switch(state) // controls flow depending on state value
{
    case 0: // case 0, run if state equals 0
        TimeDisplay(0); // function header, runs TimeDisplay function prototype
        DateDisplay(1);

        month_temp = month; // sets month_temp to equal month
        day_temp = day_number; // sets day_temp to equal day_number
        hours_temp = hours; // sets hours_temp to equal hours
        mins_temp = mins; // sets mins_temp to equal mins
        days_temp = days; // sets days_temp to equal days
        year_temp = year; // sets year_temp to equal year

        if(RA1) state = 8; // if RA1 is equal to 1 set state to equal 8
        if(RA2) state = 1; // if RA2 is equal to 1 set state to equal 1
        if(RA3==1) // if RA3 is equal to 1
        {
            if(alarm==0) // if alarm equals zero
            {
                alarm = 1; // sets alarm to equal 1
            }
            else // else command, if alarm is not equal to zero
            {
                alarm = 0; // sets alarm to zero
            }
        }

        break; // this breaks out from each case
}
```

Figure 3.n

The code starts with a switch case statement named 'state'. Each case within this statement is given a state number which corresponds to the case. A switch case statement acts by executing code within the case that's currently selected. Cases are selected by changing the global variable 'state' to whatever number case is required. Case 0 is begun when state is equal to 0, and two functions are called. TimeDisplay is a function that displays the time on line 0 of the LCD display, whereas DateDisplay displays the date on line 1.

Following this, are a set of assignments. These can be seen in the code and are used to set the temp_variables the same values as the 'real' variables. This is done so that when the time and date is being set, the time is equal to the current time so it is easier to make small alterations rather than having to change everything.

The next part of this case involves the inputs on port A. There are three if statements, the first states that if RA1 is equal to 1, then state is set to 8. This changes the case to case 8 which is used to set the alarm time. The second if statement states that if RA2 is equal to 1, state is set to 1. This changes the case to case 1 which is used to set the hour. The final if statement checks to see if RA3 is equal to 1. If true, the program checks to see if alarm is equal to 0. If true alarm is then set to 1. If alarm is equal to 1 alarm is set to 0. This series of if statements acts as a toggle for the alarm to be enabled and disabled.

The case is then ended through the use of a 'break'. Every case ends this way.

Good
explanati
on.

3.5.4 Case 1 – Setting Hour

Case 1 contains the code that allows the user to increment and decrement the hour variable.

Figure 3.o shows the code.

```
case 1:                                     // case 1, run if state equals 1
    SetTimeDisplay(0);                     // function header, runs TimeDisplay function prototype

    if(RA1) state = 7;                     // if RA1 is equal to 1 set state to equal 7
    if(RA2) state = 2;                     // if RA2 is equal to 1 set state to equal 2
    if(RA3)                                // if RA3 is equal to 1
    {
        hours_temp++;                      // increment hours_temp by 1
        if(hours_temp>23) hours_temp=0;    // if hours_temp is greater than 23 set to zero
    }
    if(RA4)                                // if RA4 = 1
    {
        while(RA4);                        // while RA4 is equal to 1
        Delay(100);                        // function call for Delay
        hours_temp--;                      // decrement hours_temp by 1
        if(hours_temp<0) hours_temp=23;    // if hours_temp is greater than 23 set to zero
    }
    break;                                 // this breaks out from each case
```

Figure 3.o

This case starts with a function call for SetTimeDisplay which displays the time as its being set by the user on line 0. Following this is the same navigation of if statements, except when RA2 is equal to 1 the state is then set to 2 which is used to set the minutes.

The next two if statements act as an increment and decrement for the hours. When RA3 is equal to 1 the hours_temp variable is incremented. It can be incremented up to 23, and once exceeded, will loop around back to 0 to represent the 24 hours in a day.

When RA4 is equal to 1, the hours_temp variable will be decremented instead. When taken below 0, it will loop back around to 23. A while statement is used as a de-bouncer in the decrement if statement to ensure it only decrements once. This uses the Delay function to ensure the input doesn't bounce and create multiple decrements.

The specification for this system only stated that there should be an increment option. However, adding a decrement option adds a much nicer quality of life improvement to the specified system without over complicating the code.

3.5.5 Case 2 – Setting Minute

Case 2 includes the code that allows the user to increment and decrement the minute.

```
case 2:                                     // case 2, run if state equals 2
    SetTimeDisplay(0);                     // function header, runs SetTimeDisplay function prototype

    if(RA1) state = 7;                     // if RA1 is equal to 1 set state to equal 7
    if(RA2) state = 3;                     // if RA2 is equal to 1 set state to equal 3
    if(RA3)                                // if RA3 is equal to 1 step in
    {
        mins_temp++;                      // increment mins_temp by 1
        if(mins_temp>59) mins_temp=0;    // if mins_temp is greater than 59 set to zero
    }
    if(RA4)                                // if RA4 is = 1
    {
        while(RA4);                        // while RA4 is equal to 1
        Delay(100);                        // function call for Delay
        mins_temp--;                      // decrement mins_temp by 1
        if(mins_temp<0) mins_temp=59;    // if mins_temp is greater than 59 set to zero
    }
    break;                                 // this breaks out from each case
```

Figure 3.p

As **Figure 3.p** shows the structure of this case is identical to case 1. The only changes are that RA2 now changes the state to state 3, and the minutes are incremented and decremented instead of the hours through the use of the inputs RA3 and RA4 respectively.

When the minutes are incremented past 59, the minutes loop back around to 0. This represents the 60 minutes in an hour. Likewise, when the minutes are decremented past 0 it loops around to 59. The same de-bounce while loop is used in the decrement code to stop multiple decrements.

3.5.6 Case 3 – Setting Day

Case 3 contains the code that allows the user to change the day name. This can be seen in **Figure 3.q**.

```
case 3:                                     // case 3, run if state equals 3
    SetDateDisplay(1);

    if(RA1) state = 7;                     // if RA1 is equal to 1 set state to equal 7
    if(RA2) state = 4;                     // if RA2 is equal to 1 set state to equal 4
    if(RA3)                                // if RA3 is equal to 1 step in
    {
        day_temp++;                       // increment month_temp by 1
        if(day_temp>6) day_temp=0;        // if day_temp is greater than 59 set to zero
    }
    if(RA4)                                // if RA4 is = 1
    {
        while(RA4);                       // while RA4 is equal to 1
        Delay(100);                       // function call for Delay
        day_temp--;                       // decrement day_temp by 1
        if(day_temp<0) day_temp=6;        // if mins_temp is greater than 59 set to zero
    }
    break;                                 // this breaks out from each case
```

Figure 3.q

Once again the structure is the same. However, instead of calling the function SetTimeDisplay, the function SetDateDisplay is called. This displays the date as its being set by the user on the LCD display on line 1.

When RA2 is equal to 1, the state is changed to 4. Just like the previous cases, when RA3 is equal to 1, the day_temp variable is incremented, and when RA4 is equal to 1 it is decremented. Since there are only 7 days, the variable loops when incremented above 6 or below 0.

This value of 0 to 6 represents the 7 days where 0 is equal to Sunday and 6 is equal to Saturday. These are used to display the day which will be shown later in the DateDisplay and SetDateDisplay functions.

3.5.7 Case 4 – Setting Month

Case 4 contains the code that allows the user to change the month. This can be seen in **Figure 3.r**.

```

case 4:                                     // case 4, run if state equals 4
SetDateDisplay(1);
if(RA1) state = 7;                         // if RA1 is equal to 1 set state to equal 7
if(RA2) state = 5;                         // if RA2 is equal to 1 set state to equal 5
if(RA3)
{
    month_temp++;                          // increment month_temp by 1
    if(month_temp>12) month_temp=1;        // if month_temp is greater than 59 set to zero
}
if(RA4)
{
    while(RA4);                            // while RA4 is equal to 1
    Delay(100);                            // function call for Delay
    month_temp--;                          // decrement month temp by 1
    if(month_temp<1) month_temp=12;        // if month_temp is greater than 59 set to zero
}
break;                                     // this breaks out from each case

```

Figure 3.r

The structure of this case is identical to that of Case 3. Once again the only differences are that RA2 changes the state to state 5, RA3 increments the month_temp variable and RA4 decrements the month_temp variable.

Like the day_temp variable, the month_temp variable holds an int value which represents each month, 1 being January and 12 being December. When incremented above 12, the value loops back to 1, and when decremented lower than 1 it loops back to 12.

This integer value is then converted into the name of the month in the DateDisplay and SetDateDisplay functions.

3.5.8 Case 5 – Setting Date

Case 5 contains the code that allows the user to change the date in increments or decrements.

```

case 5:                                     // case 5, run if state equals 5
SetDateDisplay(1);
if(RA1) state = 7;                         // if RA1 is equal to 1 set state to equal 7
if(RA2) state = 6;                         // if RA2 is equal to 1 set state to equal 6
if(RA3)
{
    days_temp++;                           // increment days temp by 1
    if(((month_temp==1)||(month_temp==3)||(month_temp==5)||(month_temp==7)||(month_temp==9)||(month_temp==11)||(month_temp==13))&&(days_temp>31))
    {
        days_temp = 1;                     // if the month is equal to 1,3,5,7,9,11,13 AND days_temp is greater than 31
        // set days to 1
    }
    // end if
    if((month_temp==2)&&(days_temp>28)&&(year_temp%4!=0))
    {
        days_temp = 1;                     // if the month is equal to 2 AND days_temp is greater than 28 AND the year is not divisible by 4
        // set days to 1
    }
    // end if
    if((month_temp==2)&&(days_temp>28)&&(year_temp%4==0))
    {
        days_temp = 1;                     // if the month is equal to 2 AND days_temp is greater than 28 AND the year is a multiple of 4
        // set days to 1
    }
    // end if
    if(((month_temp==4)||(month_temp==6)||(month_temp==8)||(month_temp==10)||(month_temp==12))&&(days_temp>30))
    {
        days_temp = 1;                     // if the month is equal to 4,6,8,10,12 AND days is greater than 30 AND the year is not a multiple of 4
        // set days to 1
    }
    // end if
}
// end if RA3
if(RA4)
{
    while(RA4);                            // if RA4 is equal to 1
    Delay(100);                            // function call for Delay
    days_temp--;
    if(((month_temp==1)||(month_temp==3)||(month_temp==5)||(month_temp==7)||(month_temp==9)||(month_temp==11)||(month_temp==13))&&(days_temp<1))
    {
        days_temp = 31;                     // if the month is equal to 1,3,5,7,9,11,13 AND days_temp is greater than 31
        // set days to 1
    }
    // end if
    if((month_temp==2)&&(days_temp<1)&&(year_temp%4!=0))
    {
        days_temp = 28;                     // if the month is equal to 2 AND days_temp is greater than 28 AND the year is not divisible by 4
        // set days to 1
    }
    // end if
    if((month_temp==2)&&(days_temp<1)&&(year_temp%4==0))
    {
        days_temp = 29;                     // if the month is equal to 2 AND days_temp is greater than 28 AND the year is a multiple of 4
        // set days to 1
    }
    // end if
    if(((month_temp==4)||(month_temp==6)||(month_temp==8)||(month_temp==10)||(month_temp==12))&&(days_temp<1))
    {
        days_temp = 30;                     // if the month is equal to 4 AND days is greater than 30 AND the year is not a multiple of 4
        // set days to 1
    }
    // end if
}
// end if RA4
break;                                     // this breaks out from each case

```

Figure 3.s

This case starts off the same as the previous case with the function call for SetDateDisplay, and the navigation options with RA1 and RA2. RA1 still takes the program to case 7, however RA2 takes the program to case 6.

Following these is the first if statement which is used to increment the date when RA3 is set to 1. This case was purposely put after selecting the month as the month is crucial to determine what date the user should be able to set. For example, the user should not be able to put the date past 28 in the month of February unless it's on a leap year. In that case the user should be able to set the date to the 29th, but no higher.

The code within this if statement looks at what month is set, and restricts the user from selecting a non-suitable date. This works similarly to how the interrupt determines when to change month. When the date limit for a specific month is reached, the date is set back to 1 by each individual if statement.

The second main if statement does exactly the same except for decrementing the date. When the date is taken below 1, it is set to either 28, 29, 30 or 31 depending on what month and year is set.

3.5.9 Case 6 – Setting Year

Case 6 contains the code that allows the user to change the year. This can be seen in **Figure 3.t** below.

```

case 6:                                // case 6, run if state equals 6
  SetDateDisplay(1);
  if(RA1) state = 7;                   // if RA1 is equal to 1 set state to equal 7
  if(RA3)                               // if RA3 is equal to 1
  {
    year_temp++;                       // increment year_temp by 1
  }
  if(RA4)                               // if RA4 = 1
  {
    while(RA4);
    Delay(100);
    year_temp--;                       // decrement year_temp by 1
  }
  break;                               // this breaks out from each case

```

Figure 3.t

This case is rather simple. The structure resembles that of cases 1-4, however since this is the last step required for setting the date, RA2 no longer has a function and has been removed from the code. By putting a 1 on RA3 the year is incremented, and by putting a 1 on RA4 the year is decremented. Once the user reaches this case, the only option for them to leave is by pressing RA1 which takes the program to case 7.

3.5.10 Case 7 – Saving Values

Case 7 contains the code that allows the program to retain the values the user has selected in the previous cases. This can be seen in **Figure 3.u**.

```

case 7:                                     // case 7, run if state equals 7
    TimeDisplay(0);                         // function header, runs TimeDisplay function prototype
    DateDisplay(1);

    hours = hours_temp;                    // sets hours to equal hours_temp
    mins = mins_temp;                     // sets mins to equal mins_temp
    day_number = day_temp;                 // sets day_number to equal day_temp
    month = month_temp;                   // sets month to equal month_temp
    days = days_temp;                     // sets days to equal days_temp
    year = year_temp;                     // sets year to equal year_temp
    alarm_h = alarm_hours_temp;            // sets alarm_h to equal alarm_hours_temp
    alarm_m = alarm_mins_temp;             // sets alarm_m to equal alarm_mins_temp

    Delay(100);                            // function header, runs Delay function prototype
    state = 0;                             // sets state to equal zero
    break;                                 // this breaks out from each case

```

Figure 3.u

As shown above, the case starts with the function calls TimeDisplay and DateDisplay. These two functions are once again used to display the actual time and date on lines 0 and 1 respectively. Following this are a series of assigning strings which are used to assign the 'real' global variables the values stored within the 'temp' global variables which were selected by the user.

By doing this, the program stores the selected time and date within the 'real' global variables which are then used in the functions TimeDisplay and DateDisplay. This is the final step of setting the time and date as now the clock displays the selected time, and the interrupt continues the clock accurately from there on.

After these strings, the delay function call is used to create a small delay before setting the state back to 0 which in turn changes the case from 7, to 0.

3.5.11 Case 8 – Setting Alarm

This state contains the code that starts the configuration of the alarm. The code can be seen in **Figure 3.v**.

```

case 8:                                     // case 8, run if state equals 8
    DateDisplay(1);                         // function header, runs Display alarm and Date function
    AlarmTimeDisplay(0);                   // sets alarm_hours_temp to equal alarm_h
    alarm_hours_temp = alarm_h;             // sets alarm_mins_temp to equal alarm_m
    alarm_mins_temp = alarm_m;

    if(RA1) state = 0;                     // if RA1 is equal to 1 set state to equal 0
    if(RA2) state = 9;                     // if RA2 is equal to 1 set state to equal 9

    break;                                 // this breaks out from each case

```

Figure 3.v

This state begins with two function calls; DateDisplay and AlarmTimeDisplay. DateDisplay is used here to allow the clock to continue displaying the date as it isn't relevant to setting the alarm. AlarmTimeDisplay is a function used to display the time the alarm is currently set to. When the clock is first powered up the default alarm is 00:00. Following these function calls are two assigning strings which are used to set alarm_hours_temp and alarm_mins_temp to alarm_h and alarm_m respectively. This puts any previously set alarm values within the temp variables so that any existing alarms can easily be altered.

The case is then continued with two if statements used to navigate through the alarm configuration cases. Firstly, when RA1 is set to 1, the program returns to state 0. When RA2 is pressed the program continues on to case 9.

3.5.12 Case 9 – Setting Alarm Hours

This case contains the code that allows the user to set the hour the alarm is meant to go off. The code can be seen in **Figure 3.w**.

```
case 9:                                     // case 9, run if state equals 9
    SetAlarmTimeDisplay(0);                // function header, runs SetAlarmTimeDisplay function prototype
    if(RA1) state = 7;                     // if RA1 is equal to 1 set state to equal 7
    if(RA2) state = 10;                    // if RA2 is equal to 1 set state to equal 10
    if(RA3)                                // if RA3 is equal to 1 step in
    {
        alarm_hours_temp++;                // increment alarm_hours_temp by 1
        if(alarm_hours_temp>23) alarm_hours_temp=0; // if alarm_hours_temp is greater than 23 set to 0
    }
    if(RA4)
    {
        alarm_hours_temp--;                // decrement alarm_hours_temp by 1
        if(alarm_hours_temp<0) alarm_hours_temp=23; // if alarm_hours_temp is less than 0 set to 23
    }
    break;                                 // this breaks out from each case
```

Figure 3.w

This case starts off with a new function call; SetAlarmTimeDisplay. This function is used to display the alarm as its being set by the user. Following the function call are four if statements. The first if statement states that if RA1 is equal to 1, the state is changed to 7. This allows the user to save any changes as case 7 transfers the values stored in the 'temp' variables to the 'real' variables. If RA2 is set to 1, the state is changed to 10 which makes the program proceed to case 10.

The next if statement states that if RA3 is set to 1, the alarm_hours_temp variable is incremented. Just like when setting the hours, if the variable is incremented past 23, it loops back around to 0. The final if statement decrements alarm_hours_temp when RA4 is equal to 1. The case is then ended.

3.5.13 Case 10 – Setting Alarm Minutes

Case 10 is the final case in the main program. Its purpose is the same as case 9 except instead of setting the hour, it sets the minutes. The code can be seen

```
case 10:                                     // case 10, run if state equals 10
    SetAlarmTimeDisplay(0);                // function header, runs SetAlarmTimeDisplay function prototype
    if(RA1) state = 7;                     // if RA1 is equal to 1 set state to equal 7
    if(RA2) state = 8;                     // if RA2 is equal to 1 set state to equal 8
    if(RA3)                                // if RA3 is equal to 1
    {
        alarm_mins_temp++;                // increment alarm_mins_temp by 1
        if(alarm_mins_temp>59) alarm_mins_temp=0; // if alarm_mins_temp is greater than 59 set to 0
    }
    if(RA4)
    {
        alarm_mins_temp--;                // decrement alarm_mins_temp by 1
        if(alarm_mins_temp<0) alarm_mins_temp=59; // if alarm_mins_temp is less than 0 set to 59
    }
    break;                                 // this breaks out from each case
}
while(RA1 || RA2 || RA3);                 // while RA1 or RA2 or RA3 are active do nothing
```

Figure 3.x

The major differences here are that when RA2 is pressed, the program returns to state 8 which means none of the alarm settings are saved, and when RA3 or RA4 are set to 1, the alarm_mins_temp variable is incremented or decremented respectively.

When the user has selected the desired alarm, it is saved by setting RA1 to 1 which takes the program to case 7 where the temp values are transferred to the real values.

*States are explained
clearly.*

The final part of the main program is a single line which states; while RA1 or RA2 or RA3 are set to 1, the program should do nothing. This allows the user time to configure the time and alarm due to the microprocessors fast clock speed.

3.6 TIMEDISPLAY AND SETTIMEDISPLAY FUNCTIONS

The TimeDisplay function is used to display the time constantly while the clock is active. The only time this function isn't displaying data on the LCD screen is when the user is setting the time or the alarm. SetTimeDisplay is the function which displays data on the LCD when the user is doing so. TimeDisplay uses the values stored within the 'real' global variables to display the time, whereas SetTimeDisplay uses the 'temp' global variables. These two functions both share similar structure where the major differences are the global variables used. The code for both these functions can be found in **Figure 3.y** and **Figure 3.z**.

```

//////////////////////////////////////display time function//////////////////////////////////////
void TimeDisplay(int line)
3{
    char time_string[16] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; // local string declared with 16

    if(alarm==1) // if alarm is equal to 1 step in
    {
        time_string[0] = (char) hours / 10 + 48; // displays the first hour digit. 48 equals the ascii code for 0
        time_string[1] = (char) hours % 10 + 48; // displays the second hour digit. 48 is ascii for 0
        time_string[2] = ':'; // displays the : digit.
        time_string[3] = (char) mins / 10 + 48; // displays the first minute digit. 48 is ascii 0
        time_string[4] = (char) mins % 10 + 48; // displays the second minute digit.
        time_string[5] = ' '; // space
        time_string[6] = ' '; // space
        time_string[7] = ' '; // space
        time_string[8] = ' '; // space
        time_string[9] = ' '; // space
        time_string[10] = ' '; // space
        time_string[11] = ' '; // space
        time_string[12] = ' '; // space
        time_string[13] = ' '; // space
        time_string[14] = (char) 33; // ascii code 33 = !
        time_string[15] = 0; // ends the string by placing 0 in the last string
    }
    if(alarm==0) // if alarm is equal to 0
    {
        time_string[0] = (char) hours / 10 + 48; // displays the first hour digit, 48 is ascii 0
        time_string[1] = (char) hours % 10 + 48; // displays the second hour digit
        time_string[2] = ':'; // displays the : digit
        time_string[3] = (char) mins / 10 + 48; // displays the first minute digit
        time_string[4] = (char) mins % 10 + 48; // displays the second minute digit
        time_string[5] = ' '; // space
        time_string[6] = ' '; // space
        time_string[7] = ' '; // space
        time_string[8] = ' '; // space
        time_string[9] = ' '; // space
        time_string[10] = ' '; // space
        time_string[11] = ' '; // space
        time_string[12] = ' '; // space
        time_string[13] = ' '; // space
        time_string[14] = ' '; // space
        time_string[15] = 0; // ends the string by placing 0 in the last string
    }
    DisplayString(line, time_string); // sends the string to the LCD on line 0
}

```

Figure 3.y

```

//////////////////////////////////////display set time function//////////////////////////////////////
void SetTimeDisplay(int line)
{
    char time_string[16] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; //Local string declared

    if(alarm==1) // if alarm is equal to 1
    {
        time_string[0] = (char) hours_temp / 10 + 48; // displays the first hour digit
        time_string[1] = (char) hours_temp % 10 + 48; // displays the second hour digit.
        time_string[2] = ':'; // displays the : digit
        time_string[3] = (char) mins_temp / 10 + 48; // displays the first minute
        time_string[4] = (char) mins_temp % 10 + 48; // displays the second minute digit
        time_string[5] = ' '; // space
        time_string[6] = ' '; // space
        time_string[7] = ' '; // space
        time_string[8] = ' '; // space
        time_string[9] = ' '; // space
        time_string[10] = ' '; // space
        time_string[11] = ' '; // space
        time_string[12] = ' '; // space
        time_string[13] = ' '; // space
        time_string[14] = (char) 33; // ascii code 33 = !
        time_string[15] = 0; // Ends the string by placing 0 in the last string
    }
    if(alarm==0) // if alarm is equal to 0
    {
        time_string[0] = (char) hours_temp / 10 + 48; // displays the first hour digit
        time_string[1] = (char) hours_temp % 10 + 48; // displays the second hour digit
        time_string[2] = ':'; // displays the : digit
        time_string[3] = (char) mins_temp / 10 + 48; // displays the first minute digit
        time_string[4] = (char) mins_temp % 10 + 48; // displays the second minute
        time_string[5] = ' '; // space
        time_string[6] = ' '; // space
        time_string[7] = ' '; // space
        time_string[8] = ' '; // space
        time_string[9] = ' '; // space
        time_string[10] = ' '; // space
        time_string[11] = ' '; // space
        time_string[12] = ' '; // space
        time_string[13] = ' '; // space
        time_string[14] = ' '; // space
        time_string[15] = 0; // ends the string by placing 0 in the last string
    }
    DisplayString(line, time_string); // sends the string to the LCD on line 0
}

```

Figure 3.2

Just by looking at the two figures it can be seen that their structures are the same. Each function brings in the line variable of data type int in order to determine which line of the LCD they should send data to. Following the prototype definition is a local array named time_string which is defined to store the data type char. The array in both functions has 16 elements and is initially filled with 0s. Following the array declaration are two if statements. The first is used if the variable alarm is equal to 1 and the second is used if it's set to 0. (Alarm is toggled between 1 and 0 by the input RA3 when the program is operating in case 0.). The first five elements (0 to 4) in both functions are used to display the time. TimeDisplay uses the global variables hours and mins to do this whereas SetTimeDisplay uses hours_temp and mins_temp. The 15th element is the only other element that contains a visible character when the alarm is set. This element contains the ascii code 33 which represents an exclamation mark. This is used to display to the user that the alarm is toggled on. If alarm is set to 0 the element contains a space to show nothing.

After both if statements, another function call is declared. This function call is DisplayString which can be found in LCD.c. This function uses the variable line to determine which line to display the data on. When both TimeDisplay and SetTimeDisplay are called they have a 0 integer as an input which is used to tell DisplayString to display the data on line 0 of the LCD. The function call also takes the data stored within the array time_string in order to send that data to the LCD display.

*Good
explanation.*

3.7 DELAY FUNCTION

This function is used as a small delay during various points of the program this function can be seen in **Figure 3.aa**.

```
////////////////////////////////////delay function////////////////////////////////////
void Delay(int t)                                     // start of function
{
    int i;                                           // local variable i
    for(i=0;i<t;i++);                               // increment i as long as i is smaller than t
}
```

Figure 3.aa

The Delay function consists of a local variable i which is then incremented continuously via a for statement until it reaches the variable t. The function then ends. This takes time for the microprocessor to do and acts as a time delay. By increasing the value of t, the delay also increases.

3.8 STATEDISPLAY FUNCTION

This function is used to display the current state that the program is in to help with design and testing. This code can be seen in

```
////////////////////////////////////display state function////////////////////////////////////
void StateDisplay(int line)
{
    char state_string[8] = {0, 0, 0, 0, 0, 0, 0, 0}; // local array for the state number with 8 elements
    state_string[0] = ' ';                          // space
    state_string[1] = ' ';                          // space
    state_string[2] = ' ';                          // space
    state_string[3] = ' ';                          // space
    state_string[4] = ' ';                          // space
    state_string[5] = ' ';                          // space
    state_string[6] = (char) state + 48;             // state number
    state_string[7] = 0;                             // ends the string by placing a 0 in the last string
    DisplayString(line, state_string);               // function call for DisplayString in LCD.c
}
```

Figure 3.bb

This function once again brings in the integer line to determine what line it will send data to. Inside the function is a local array which has 8 elements named state_string. This array stores char data types and only uses element 6. Within this element is the global variable state which enables this function to display which state the program is currently operating in.

Just like TimeDisplay, the function is ended with the DisplayString function call for the same reasons as previously stated.

3.9 DATEDISPLAY AND SETDATEDISPLAY FUNCTIONS

DateDisplay and SetDateDisplay operate on the same principles that TimeDisplay and SetTimeDisplay operate on. However, these functions are much longer as they display the day, date, month and year. Small sections of code can be found below.

```

////////////////////////////////////display date function////////////////////////////////////
void DateDisplay(int line)

char day_number_string[5] = {0, 0, 0, 0, 0};
char month_string[5] = {0, 0, 0, 0, 0}; // local string declared with 5 elements
char date_string[16] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; // local string declared with 16 elements

if(day_number==0)
{
    day_number_string[0] = (char) 83; // char S
    day_number_string[1] = (char) 85; // char U
    day_number_string[2] = (char) 78; // char N
    day_number_string[3] = ' '; // creates space
    day_number_string[4] = 0; // ends the string by placing 0 in the last string
}
if(day_number==1)
{
    day_number_string[0] = (char) 77; // char M
    day_number_string[1] = (char) 79; // char O
    day_number_string[2] = (char) 78; // char N
    day_number_string[3] = ' '; // creates space
    day_number_string[4] = 0; // ends the string by placing 0 in the last string
}
if(day_number==2)
{
    day_number_string[0] = (char) 84; // char T
    day_number_string[1] = (char) 85; // char U
    day_number_string[2] = (char) 69; // char E
    day_number_string[3] = ' '; // creates space
    day_number_string[4] = 0; // ends the string by placing 0 in the last string
}
if(day_number==3)
{
    day_number_string[0] = (char) 87; // char W
    day_number_string[1] = (char) 69; // char E
    day_number_string[2] = (char) 68; // char D
    day_number_string[3] = ' '; // creates space
    day_number_string[4] = 0; // ends the string by placing 0 in the last string
}
if(day_number==4)

```

Figure 3.cc

```

}

if(month==1)
{
    month_string[0] = (char) 74; // JAN
    month_string[1] = (char) 65;
    month_string[2] = (char) 78;
    month_string[3] = ' ';
    month_string[4] = 0;
}
if(month==2)
{
    month_string[0] = (char) 70; // FEB
    month_string[1] = (char) 69;
    month_string[2] = (char) 66;
    month_string[3] = ' ';
    month_string[4] = 0;
}
if(month==3)
{
    month_string[0] = (char) 77; // MAR
    month_string[1] = (char) 65;
    month_string[2] = (char) 82;
    month_string[3] = ' ';
    month_string[4] = 0;
}
if(month==4)
{
    month_string[0] = (char) 65; // APR
    month_string[1] = (char) 80;
    month_string[2] = (char) 82;
    month_string[3] = ' ';
    month_string[4] = 0;
}
if(month==5)
{
    month_string[0] = (char) 77; // MAY
    month_string[1] = (char) 65;
    month_string[2] = (char) 89;
    month_string[3] = ' ';
    month_string[4] = 0;
}

```

Figure 3.dd

```

}
if(month==12)
{
    month_string[0] = (char) 68;    // DEC
    month_string[1] = (char) 69;
    month_string[2] = (char) 67;
    month_string[3] = ' ';
    month_string[4] = 0;
}

date_string[0] = day_number_string[0];    // day letter 1
date_string[1] = day_number_string[1];    // day letter 2
date_string[2] = day_number_string[2];    // day letter 3
date_string[3] = day_number_string[3];    // space
date_string[4] = month_string[0];        // month letter 1
date_string[5] = month_string[1];        // month letter 2
date_string[6] = month_string[2];        // month letter 3
date_string[7] = month_string[3];        // space
date_string[8] = days / 10 + 48;          // day 10's
date_string[9] = days % 10 + 48;          // day units
date_string[10] = ' ';                    // space
date_string[11] = year / 1000 + 48;       // year 1000's
date_string[12] = year % 1000 / 100 + 48; // year 100's
date_string[13] = year % 100 / 10 + 48;   // year 10's
date_string[14] = year % 10 + 48;         // year units
date_string[15] = 0;                      // ends the string by placing 0 in the last string

DisplayString(line, date_string);          // sends the string to the LCD on line 1
}

```

Figure 3.ee

The above figures show that this function utilizes three local arrays. The first is day_number_string which contains 5 elements. This array is used to store the name of each day. The names are made by having three ascii codes stored within the array to spell the start of each day. This can be seen in **Figure 3.cc**. By using the day_number variable the name of each day is produced.

The second array is used to store the name of each month using the same principle. The array is named month_string and has 5 elements again. It uses the series of if statements and the number of the month to determine which month should be stored within the local array. This can be seen in **Figure 3.dd**.

The final part of this function uses the third array. This array is named date_string which has a total of 16 elements. This array stores the data from day_number_string and month_string along with the days and year variable in order to create a string of data that can be displayed on the LCD.

This array is then taken by the function DisplayString to be displayed on line 1 of the LCD display. The function DisplaySetDate operates exactly the same except it uses the temp variables to determine what is shown on the LCD rather than the real values. This can be seen below.

```

void SetDateDisplay(int line)
{
    char month_string[5] = {0, 0, 0, 0, 0};
    char day_number_string[5] = {0, 0, 0, 0, 0};
    char date_string[16] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; // local string declared with 16 elements

    if(day_temp==0)
    {
        day_number_string[0] = (char) 83;    // char S
        day_number_string[1] = (char) 85;    // char U
        day_number_string[2] = (char) 78;    // char N
        day_number_string[3] = ' ';          // space
        day_number_string[4] = 0;            // ends the string by placing 0 in the last string
    }
    if(day_temp==1)
    {
        day_number_string[0] = (char) 77;    // char M
        day_number_string[1] = (char) 79;    // char O
        day_number_string[2] = (char) 78;    // char N
        day_number_string[3] = ' ';          // space
        day_number_string[4] = 0;            // ends the string by placing 0 in the last string
    }
}

```

Figure 3.ff

```

if(month_temp==1)
{
    month_string[0] = (char) 74;    //JAN
    month_string[1] = (char) 65;
    month_string[2] = (char) 78;
    month_string[3] = ' ';
    month_string[4] = 0;
}
if(month_temp==2)
{
    month_string[0] = (char) 70;    //FEB
    month_string[1] = (char) 69;
    month_string[2] = (char) 66;
    month_string[3] = ' ';
    month_string[4] = 0;
}
if(month_temp==3)
{
    month_string[0] = (char) 77;    //MAR
    month_string[1] = (char) 65;
    month_string[2] = (char) 82;
    month_string[3] = ' ';
    month_string[4] = 0;
}
if(month_temp==4)
{
    month_string[0] = (char) 65;    //APR
    month_string[1] = (char) 80;
    month_string[2] = (char) 82;
    month_string[3] = ' ';
    month_string[4] = 0;
}

```

Figure 3.gg

```

}
date_string[0] = day_number_string[0];    // day letter 1
date_string[1] = day_number_string[1];    // day letter 1
date_string[2] = day_number_string[2];    // day letter 1
date_string[3] = day_number_string[3];    // space
date_string[4] = month_string[0];        // month letter 1
date_string[5] = month_string[1];        // month letter 2
date_string[6] = month_string[2];        // month letter 3
date_string[7] = month_string[3];        // space
date_string[8] = days_temp / 10 + 48;    // day 10's
date_string[9] = days_temp % 10 + 48;    // day units
date_string[10] = ' ';                  // space
date_string[11] = year_temp / 1000 + 48; // year 1000's
date_string[12] = year_temp % 1000 / 100 + 48; // year 100's
date_string[13] = year_temp % 100 / 10 + 48; // year 10's
date_string[14] = year_temp % 10 + 48;   // year units
date_string[15] = 0;                    // ends the string by placing 0 in the last string

DisplayString(line, date_string);        // sends the string to the LCD on line 1
}

```

Figure 3.hh

The ascii codes used for these functions were derived using an ascii table found at <http://www.asciitable.com/>.

3.10 ALARMTIMEDISPLAY AND SETALARMTIMEDISPLAY FUNCTIONS

These functions are used to display the time when the alarm is being set. AlarmTimeDisplay is used in case 8 where the user initially presses RA1. When the user wants to change the alarm_hours_temp value and the alarm_mins_temp value SetAlarmTimeDisplay is used. The code for these two functions is once again almost identical and can be seen in **Figure 3.ii**.

```
////////////////////////////////////display alarm time function////////////////////////////////////
void AlarmTimeDisplay(int line)
{
    char alarm_string[16] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; //local string declared with
                                                                    //16 elements
    alarm_string[0] = (char) alarm_h / 10 + 48; // displays the first hour digit.
    alarm_string[1] = (char) alarm_h % 10 + 48; // displays the second hour digit.
    alarm_string[2] = ':'; // displays the : digit.
    alarm_string[3] = (char) alarm_m / 10 + 48; // displays the first minute digit.
    alarm_string[4] = (char) alarm_m % 10 + 48; // displays the second minute digit.
    alarm_string[5] = ' '; // space
    alarm_string[6] = ' '; // space
    alarm_string[7] = ' '; // space
    alarm_string[8] = ' '; // space
    alarm_string[9] = ' '; // space
    alarm_string[10] = (char) 65; // ascii code 65 = A
    alarm_string[11] = (char) 76; // ascii code 76 = L
    alarm_string[12] = (char) 65; // ascii code 65 = A
    alarm_string[13] = (char) 82; // ascii code 82 = R
    alarm_string[14] = (char) 77; // ascii code 77 = M
    alarm_string[15] = 0; // ends the string by placing 0 in the last string

    DisplayString(line, alarm_string); // sends the string to the LCD on line 0
}

////////////////////////////////////display set alarm time function////////////////////////////////////
void SetAlarmTimeDisplay(int line)
{
    char alarm_string[16] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; //Local string with 16 elements

    alarm_string[0] = (char) alarm_hours_temp / 10 + 48; //Displays the first hour digit.
    alarm_string[1] = (char) alarm_hours_temp % 10 + 48; //Displays the second hour digit.
    alarm_string[2] = ':'; // displays the : digit.
    alarm_string[3] = (char) alarm_mins_temp / 10 + 48; // displays the first minute digit.
    alarm_string[4] = (char) alarm_mins_temp % 10 + 48; // displays the second minute digit.
    alarm_string[5] = ' '; // space
    alarm_string[6] = ' '; // space
    alarm_string[7] = ' '; // space
    alarm_string[8] = ' '; // space
    alarm_string[9] = ' '; // space
    alarm_string[10] = (char) 65; // ascii code 65 = A
    alarm_string[11] = (char) 76; // ascii code 76 = L
    alarm_string[12] = (char) 65; // ascii code 65 = A
    alarm_string[13] = (char) 82; // ascii code 82 = R
    alarm_string[14] = (char) 77; // ascii code 77 = M
    alarm_string[15] = 0; // ends the string by placing 0 in the last string

    DisplayString(line, alarm_string); // sends the string to the LCD on line 0
}
```

Figure 3.ii

These functions operate once again on the same principle that TimeDisplay operates upon. They use a local array named alarm_string with 16 elements. Elements 0 to 4 contain the alarm hours and minutes and elements 10 to 14 contain ascii codes that spell out ALARM. This displays when the alarm is being set. Once again DisplayString is used to send the array data to the LCD display. For SetAlarmTimeDisplay the variables used are the temp alarm variables.

These functions were the last lines of code for the program to operate. With the program complete simulation and testing can be done. This can be seen in **Section 4**.

*Program code is
explained well in this
section.*

4 TESTING AND ANALYSIS OF RESULTS

4.1 TIMER SIMULATION TESTS

The first thing to be tested is the timer circuit to ensure that a 1 second pulse was created. This is done by simulating a timer circuit in Proteus. The circuit can be seen in **Figure 4.a**.

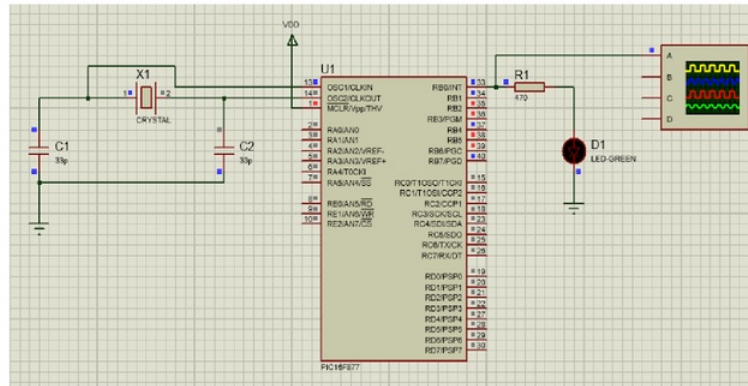


Figure 4.a

This circuit is designed with an external oscillator circuit connected to the CLKIN and CLKOUT pins of the PIC. Connected to RBO is a pull up resistor connected in series with an LED which is then connected to ground. This acts as an indicator for how fast the clock is pulsing. Also connected to RBO is a scope which will show the output waveform. The first pulse to be tested is the 10ms pulse. The test can be seen in **Figure 4.b**.

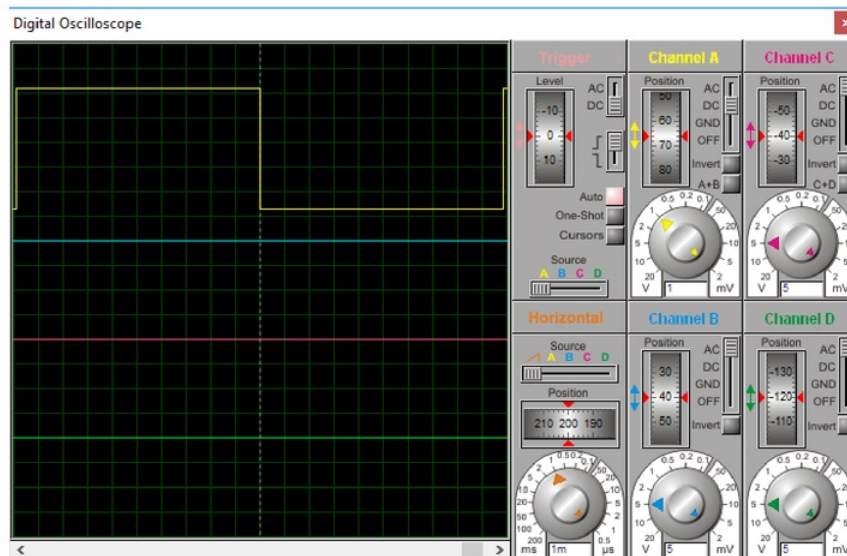


Figure 4.b

The circuit uses three push buttons as inputs for RA1, RA2, RA3 and RA4. Each of these push buttons are connected to their respective pins while also being connected in parallel to pull down resistors. These resistors are there to stop floating voltages from staying on the input pins. These pull down resistors are all connected to ground. When one of the push buttons is pressed the 5V supply voltage gets applied to the input pins putting them to a logic 1.

RA0 is the only output on port A, and is connected to a pull up resistor which is then connected to an LED. This LED is used to simulate the alarm. RA0 acts as a source which means that a voltage is only applied to this pin when it is set to a logic 1.

Port B is connected to the LCD screen in order to send the data from the PIC. Both the PIC and the LCD screen are powered by a 5V supply.

In order to test the alarm clock in Proteus a testing table must be created. This table will contain the criteria being tested, the expected result, a pass / fail column and a figure reference for each test. To make testing easier, within each display function the state number has also been implemented so that the case can always be seen. Each test will have a figure to reference to which can be found after the table.

Test	Expected Result	Pass / Fail	Figure Ref.
Case 0 Displaying Time and Date	Case 0 Displayed Time and Date Displayed	Pass	Figure 4.e
Case 1 - Increment Hours	Case 1 Displayed Hours Incremented	Pass	Figure 4.f
Case 1 - Decrement Hours	Case 1 Displayed Hours Decrement	Pass	Figure 4.g
Case 2 - Increment Minutes	Case 2 Displayed Minutes Incremented	Pass	Figure 4.h
Case 2 - Decrement Minutes	Case 2 Displayed Minutes Decrement	Pass	Figure 4.i
Case 3 - Days + and -	Case 3 Displayed Each Day Being Cycled Through	Pass	Figure 4.j
Case 4 - Months + and -	Case 4 Displayed Each Month Being Cycled Through	Pass	Figure 4.k Figure 4.l
Case 5 - Increment date	Case 5 Displayed Date Incremented	Pass	Figure 4.m
Case 5 - Decrement Date	Case 5 Displayed Date Decrement	Pass	Figure 4.n
February 28 days	February have 28 day limit	Pass	Figure 4.o Figure 4.p
February Leap Year 29 Days	February date limit increased to 29 on a leap year	Pass	Figure 4.q
Case 6 - Increment Year	Case 6 Displayed Year Incremented	Pass	Figure 4.r
Case 6 - Decrement Year	Case 6 Displayed Year Decrement	Pass	Figure 4.s
Back to Case 0 - Saved Vales	Case 0 Displayed Time Values Saved	Pass	Figure 4.t
Case 8 - Alarm Function	Case 8 Displayed 00:00 and ALARM Displayed	Pass	Figure 4.u

Case 9 - Increment Alarm Hours	Case 9 Displayed Alarm Hour Incremented	Pass	Figure 4.v
Case 9 – Decrement Alarm Hours	Case 9 Displayed Alarm Hours Decremented	Pass	Figure 4.w
Case 10 – Increment Alarm Minutes	Case 10 Displayed Alarm Minutes incremented	Pass	Figure 4.w
Case 10 – Decrement Alarm Minutes	Case 10 Displayed Alarm Minutes Decremented	Pass	Figure 4.w
Return to Case 8	Case 8 Displayed Alarm Hours and Minutes reset to 00:00	Pass	Figure 4.x
Alarm time being set 01:01	01:01 displayed	Pass	Figure 4.y
Save Alarm Time	Case 0 Displayed Then Case 8 Displayed along with 01:01	Pass	Figure 4.z Figure 4.aa
Alarm Enabled with RA3	Case 0 Displayed ! symbol displayed	Pass	Figure 4.bb
Alarm Activates at 01:01	LED Lights up at 01:01	Pass	Figure 4.cc
Alarm Turns off with another press of RA3	LED Turns off	Pass	Figure 4.dd
Alarm doesn't activate if it isn't enabled	LED does not activate at 01:01	Pass	Figure 4.ee

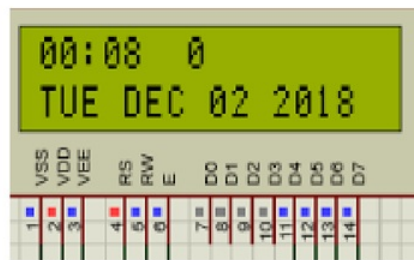


Figure 4.e



Figure 4.f



Figure 4.g

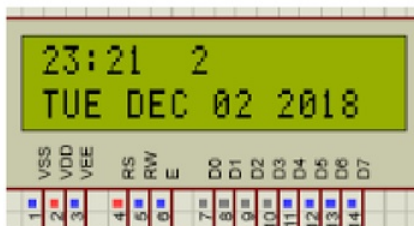


Figure 4.h

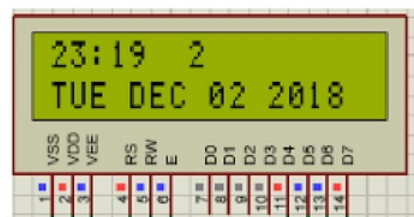


Figure 4.i



Figure 4.j



Figure 4.k

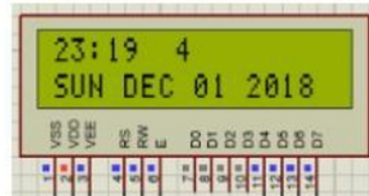
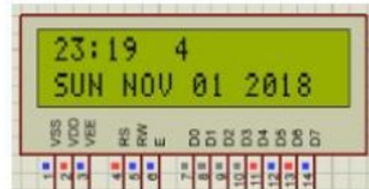
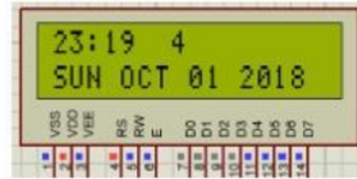


Figure 4.l

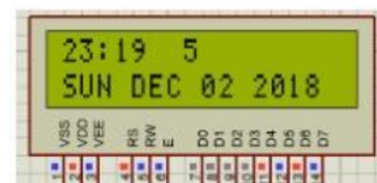


Figure 4.m

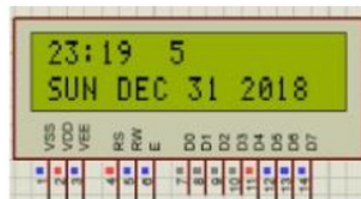


Figure 4.n

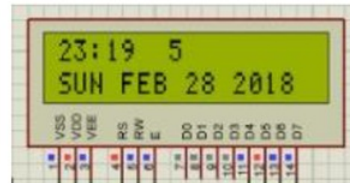


Figure 4.o

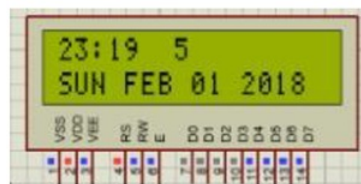


Figure 4.p

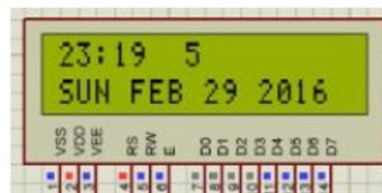


Figure 4.q

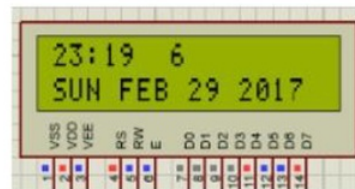


Figure 4.r

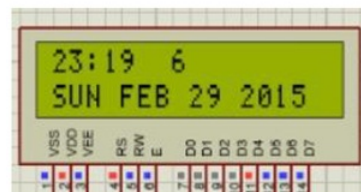


Figure 4.s

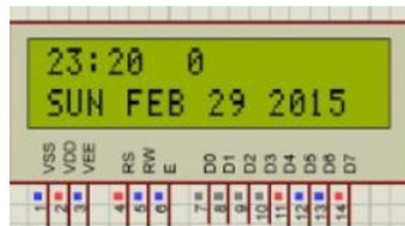


Figure 4.t



Figure 4.u

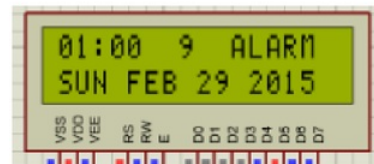


Figure 4.v

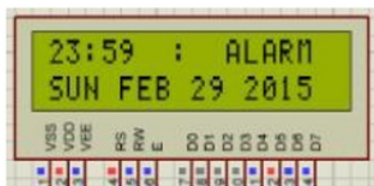


Figure 4.w



Figure 4.x



Figure 4.y

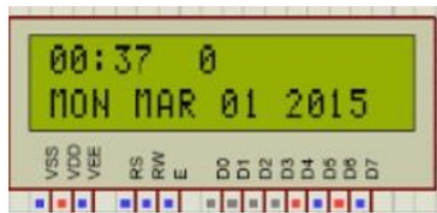


Figure 4.z



Figure 4.aa

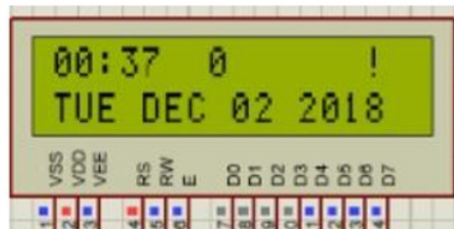


Figure 4.bb

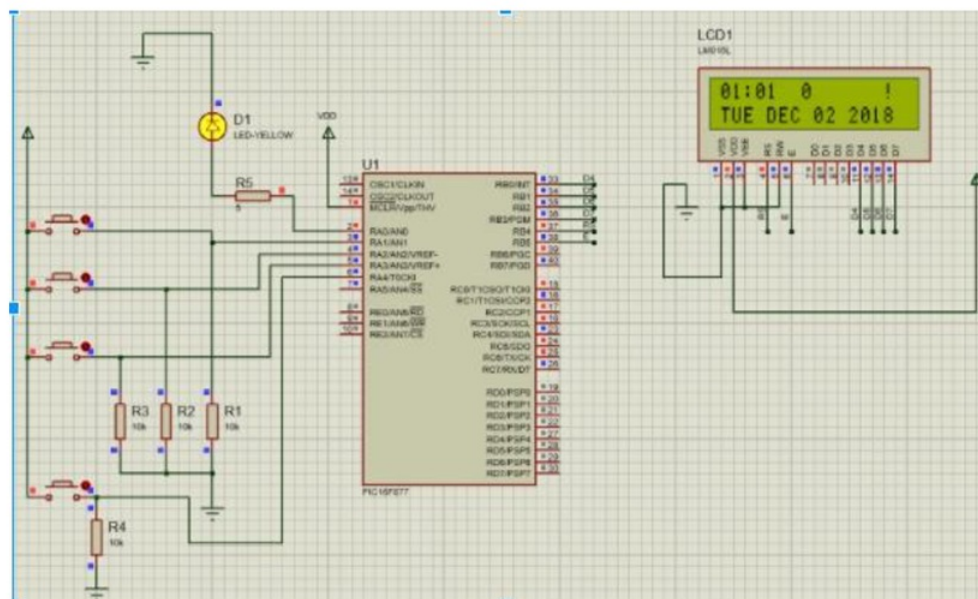
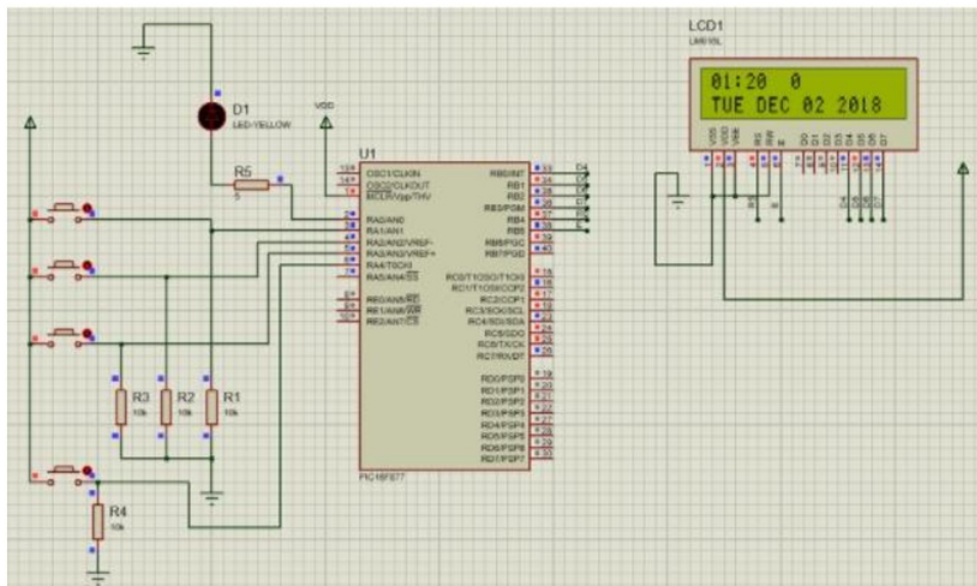


Figure 4.cc



4.3 PRACTICAL TESTS FOR THE ALARM CLOCK

In order to test the alarm clock on the development board, the program must be downloaded to the PIC 16F877. Once done the following tests will be done.

- Checking the display
- Setting the Alarm
- Enabling the Alarm

These tests can be seen in the following figures.

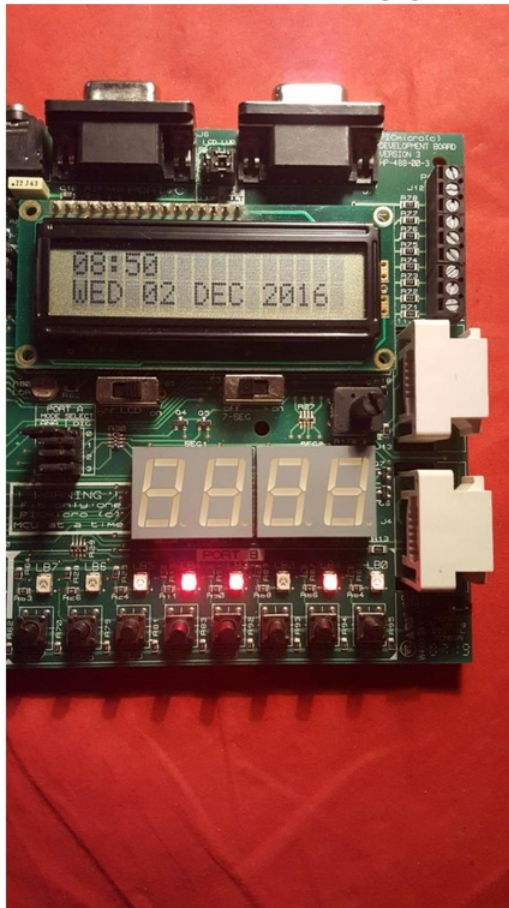


Figure 4.4f

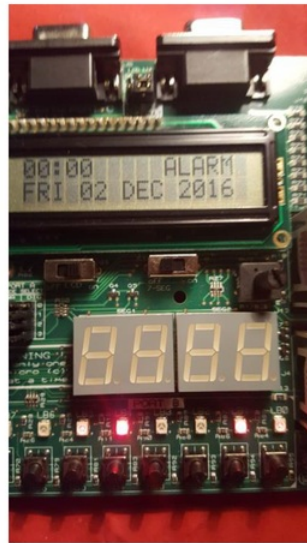


Figure 4.gg



Figure 4.hh

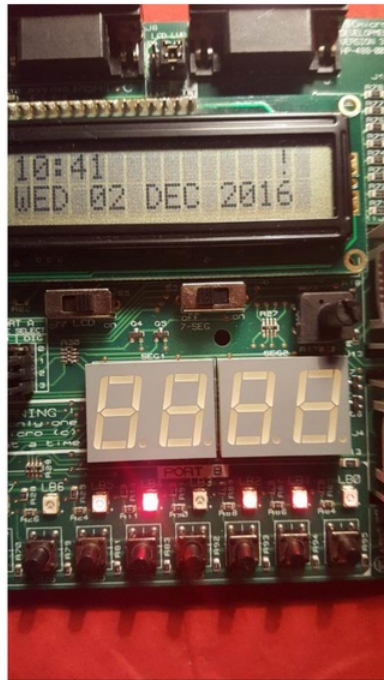


Figure 4.ii

These photos prove that the program works on the development board making the design a success.

5 CONCLUSION AND RECOMMENDATIONS

To conclude, microcontrollers are very versatile and suitable solutions to many problems. They are more affordable than PLCs and computers but can still handle many operations. They are ideal at focusing on repetitive tasks however and can get overburdened if given too many tasks to execute.

The use of Proteus as a development and design tool is invaluable as it has a user friendly interface, click and drag functionality for components, a very large library of components and is extremely compatible with boards such as the HP488. Another, great thing about Proteus is being able to see when an input or an output is at a logic high. This allows for trouble shooting and a greater understanding of how a system works.

When it came to the design of the program, the datasheet for the PIC 16F877 was a valuable tool. The information obtained from the datasheet enabled the ISR to be designed which was a key component to the alarm clock system.

The design process went very smoothly as when tested the entire system worked as the specification requested. Therefore the development was a success. However, there are improvements that could be made to the system. Firstly, a blink function could be added so that when a specific value is being edited by the user, it blinks on and off. This way the user can see what they are changing without having to have the DisplayState function tell them what case they are in. Another improvement would be to include de-bouncing while loops on more of the decrement functions as sometimes the push button would cause multiple decrements. Finally, a snooze function could be added that allows the user to disable the alarm for 5 minutes, after which the alarm will reactivate.

*Good
conclusion.*

6 APPENDICES

PIC16F87X 28/40-Pin 8-Bit CMOS FLASH Microcontrollers. (1998). 1st ed. [ebook] Dallas: Microchip, pp.1-9, 47, 129. Available at: <http://ww1.microchip.com/downloads/en/DeviceDoc/30292D.pdf> [Accessed 5 Jan. 2018].

PIC16F84 18-pin Enhanced FLASH/EEPROM 8-bit Microcontroller Datasheet. (2001). 1st ed. [ebook] Dallas: Microchip, pp.1-4. Available at: <http://ww1.microchip.com/downloads/en/DeviceDoc/35007b.pdf> [Accessed 11 Jan. 2018].

Techterms.com. (2018). *ISR (Interrupt Service Routine) Definition*. [online] Available at: <https://techterms.com/definition/isr> [Accessed 16 Jan. 2018].

Asciitable.com. (2018). *Ascii Table - ASCII character codes and html, octal, hex and decimal chart conversion*. [online] Available at: <http://www.asciitable.com/> [Accessed 17 Jan. 2018].

Sh
oul
d
ha
ve
title
of
Re
fer
en
ce
list
or
Bib
lio
gra
ph
y.

Sam Harries Assignment 1 - Final Version

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

75/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

Text Comment. Reasonable research and good comparison of MCU's with limitations.

PAGE 13

Text Comment. Good diagram.

PAGE 14

Text Comment. Good explanation.

PAGE 15

PAGE 16

PAGE 17

Text Comment. Good explanation of config.

PAGE 18

Text Comment. Good to see the prototype explained here.

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

Text Comment. Good explanation

PAGE 24

Text Comment. this is explained well.

PAGE 25

PAGE 26

PAGE 27

Text Comment. Good explanation.

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

Text Comment. States are explained clearly.

PAGE 34

PAGE 35

Text Comment. Good explanation.

PAGE 36

PAGE 37

PAGE 38

PAGE 39

PAGE 40

Text Comment. Program code is explained well in this section.

PAGE 41

PAGE 42

PAGE 43

PAGE 44

PAGE 45

PAGE 46

PAGE 47

PAGE 48

PAGE 49

PAGE 50

PAGE 51

PAGE 52

PAGE 53

PAGE 54

PAGE 55

PAGE 56

Text Comment. Good conclusion.

PAGE 57

Text Comment. Should have title of Reference list or Bibliography.

RESEARCH-INFORMED LITERATURE

 The research is good and with a comparison of MCU's. 10/15 Refs could be more academic oriented. 5/10

15/25

KNOWLEDGE AND UNDERSTANDING

 Intro is good 3/5 System design and config is explained well. 8.5/10 Software design and explanation is very good. 8.5/10

20/25

ANALYSIS

 Testing is good and explained clearly. 8/10
Conclusion is good. 8/10

16/20

PRACTICAL APPLICATION AND DEP.

 Software practical design and application is good. 8.5/10 Testing is thorough. 8.5/10

17/20

SKILLS FOR PROFESSIONAL PRAC.

 Report is very good 3.5/5 Team working is good. 3.5/5

7/10