University of Huddersfield

Alarm Clock Design Report

Embedded systems

Jack Parkinson U1552044
4-28-2017

Contents

1. Introduction

1.1. Background

For the project this report is based on, an alarm clock with weather functionality was to be developed in the C programming language, based around the PIC16F877A MCU, using the Matrix Multimedia PIC development platform and MPLABX IDE. A design specification was provided, which detailed exactly how the alarm clock should operate and certain design requirements the program had to meet.

1.2. Aims and Objectives

The aim of this report is to explain the key design decisions made during the development of the alarm clock program. The main aim of the design approach taken for this project was a good end user experience meaning the alarm clock is responsive, keeps the time well and is intuitive to use. To achieve this, the program was design to:

- Implement interrupts wherever convenient
- Get in and out of the ISR as fast as possible
- Keep the use of delays to a minimum
- Priorities computation speed over RAM usage (store larger variables if it decreases computation time)

While the end user experience was the main focus, there were other considerations during the design process such as:

- Use local variables where convenient, to reduce RAM usage and increase data safety, increasing the programs overall efficiency and allowing room for additional functionality to be added in the future.
- Program is readable and well annotated, to make future maintenance and updates easier.
- Functions are reusable, to help reduce the development time of future projects.
- limiting code repetition to reduce programming memory usage, increasing the programs overall efficiency and allowing room for additional functionality to be added in the future.

2. Design Approach

2.1. Variables

The table below details and justifies any variable’s that where assigned data types for a very specific reason, any other variables used where just assigned to the smallest possible data type to reduce RAM usage.

Figure 2.1.1

Name	What’s it storing and why?	Type	Scope
elapsedTime	Stores the elapsed time in half seconds as this is the longest delay the Timer1 module can be set to that is devisable by a second. Storing half seconds reduces computation time vs storing seconds as the Timer1 delay doesn’t have to be cascaded. Storing half seconds doesn’t increase RAM usage as the same data type would be required for storing seconds. Also, converting half seconds to hours, mins and seconds doesn’t require any more computation time than doing the same conversion with seconds.	**unsigned long** used as this is the smallest data type that allows a decimal number upto 172800 (24 hours in half seconds)	A global variable set to static as it is only used with the main.c file
alarmTime	Stores the time the alarm is set to in half seconds as this allows it to be compared with the elapsedTime time without having to do any conversions, reducing computation time.	**unsigned long** used as this is the smallest data type that allows a decimal number upto 172800 (24 hours in half seconds)	A global variable set to static as it is only used with the main.c file
TMR0overflows	Stores the number of times the Timer0 module has overflowed (overflows every 0.05 seconds) for the flashing of LED D7 when the alarm is on. 0.05 seconds where stored vs 0.25 as it reduces computation	**unsigned short** used as this is the smallest data type that allows decimal number upto 404 (20 seconds in 0.05 seconds)	A global variable set to static as it is only used with the main.c file

	time as the Timer0 delay doesn't have to cascaded		
setTime	Stores the time the user has set in either the set time menu or set alarm menu in half seconds as both alarmTime and elapsedTime can then be set equal to this variable without do any conversions, reducing computation time	**unsigned long** used as this is the smallest data type that allows a decimal number upto 172800 (24 hours in half seconds)	A local variable initialised within each function it is used in, not set to static as it doesn't need to be used outside the function it is initialised in, reducing RAM usage.

After breaking down the tasks required for this project, it was decided that throughout the entire program, there is a need for sixteen Boolean variables. One of the first design decisions made was to use individual bits within two global unsigned char variables (modes1 and modes2) as Boolean variables to reduce RAM usage. This had the added benefit of reducing computation time due to bitwise operations.

Figure 2.1.2

modes1		
bit	name	description
0	tempMode	What units the temperature is to be displayed in **(1=°C, 0=°F)**
1	alarmOn	Whether the alarm is currently active **(1=on, 0=off)**
2	rhDisplayed	Whether relative humidity is to be displayed on the LCD **(1=display, 0=don't display)**
3	tempDisplayed	Whether temperature is to be displayed on the LCD **(1=display, 0=don't display)**
4	Snooze	Whether snooze is currently enabled **(1=enabled, 0=disabled)**
5	RBPressed	Whether RB7 was pressed **(1=pressed, 0=not pressed)**
6	alarmMode	Whether the alarm is currently enabled **(1=enabled, 0=disabled)**
7	configMode	Whether config mode is currently enabled (in the menu system) **(1=enabled, 0=disabled)**

Figure 2.1.3

modes2		
bit	name	description
0	RD0Pressed	Whether RD0 was pressed **(1=pressed, 0=not pressed)**
1	RD1Pressed	Whether RD1 was pressed **(1=pressed, 0=not pressed)**
2	RD2Pressed	Whether RD2 was pressed **(1=pressed, 0=not pressed)**
3	RD3Pressed	Whether RD3 was pressed **(1=pressed, 0=not pressed)**
4	RD4Pressed	Whether RD4 was pressed **(1=pressed, 0=not pressed)**
5	setTimeMenu	Whether the sub menu which allows the user to set the time is currently enabled **(1=enabled, 0=disabled)**
6	setAlarmMenu	Whether the sub menu which allows the user to set the alarm time is currently enabled **(1=enabled, 0=disabled)**
7	tempModeMenu	Whether the sub menu which allows the user to switch between °C and °F is currently enabled **(1=enabled, 0=disabled)**

To make the code more readable and easier to maintain, a custom bitwise macros library was created to check each bit with both modes variables and set each bit to either a 1 or 0.

Figure 2.1.4

Macro Name	Description
TEMP_MODE	Returns the value of the respective bit within modes1
ALARM_ON	
RH_DISPLAYED	
TEMP_DISPLAYED	
SNOOZE_ENABLED	
RB_PRESSED	
ALARM_MODE	
CONFIG_MODE	
RD0_PRESSED	Returns the value of the respective bit within modes2
RD1_PRESSED	
RD2_PRESSED	
RD3_PRESSED	
RD4_PRESSED	
TIME_MENU	
ALARM_MENU	
TEMP_MODE_MENU	
ENABLE_TEMP_MODE	Sets the value of the respective bit within modes1 to 1
ENABLE_ALARM_ON	
ENABLE_RH_DISPLAYED	
ENABLE_TEMP_DISPLAYED	
ENABLE_SNOOZE_ENABLED	
ENABLE_RB_PRESSED	
ENABLE_ALARM_MODE	
ENABLE_CONFIG_MODE	
ENABLE_RD0_PRESSED	Sets the value of the respective bit within modes2 to 1
ENABLE_RD1_PRESSED	
ENABLE_RD2_PRESSED	
ENABLE_RD3_PRESSED	
ENABLE_RD4_PRESSED	
ENABLE_TIME_MENU	
ENABLE_ALARM_MENU	
ENABLE_TEMP_MODE_MENU	
TEMP_MODE	Sets the value of the respective bit within modes1 to 0
DISABLE_ALARM_ON	
DISABLE_RH_DISPLAYED	
DISABLE_TEMP_DISPLAYED	
DISABLE_SNOOZE_ENABLED	
DISABLE_RB_PRESSED	
DISABLE_ALARM_MODE	
DISABLE_CONFIG_MODE	

DISABLE_RD0_PRESSED DISABLE_RD1_PRESSED DISABLE_RD2_PRESSED DISABLE_RD3_PRESSED DISABLE_RD4_PRESSED DISABLE_TIME_MENU DISABLE_ALARM_MENU DISABLE_TEMP_MODE_MENU	Sets the value of the respective bit within modes2 to 0

2.2. Code Structure

The program was comprised of the LCDdrive library provided with the design specification and four other custom libraries (Display, ADC, Menus and Bitwise Macros) and a main.c file to tie the whole project together. The call graphs below show each function within these files.

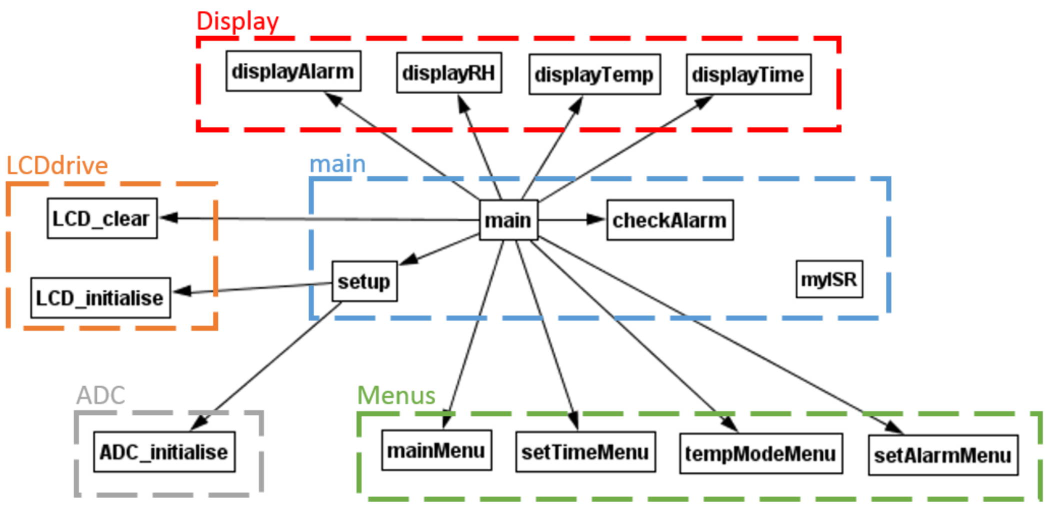


Figure 2.2.1

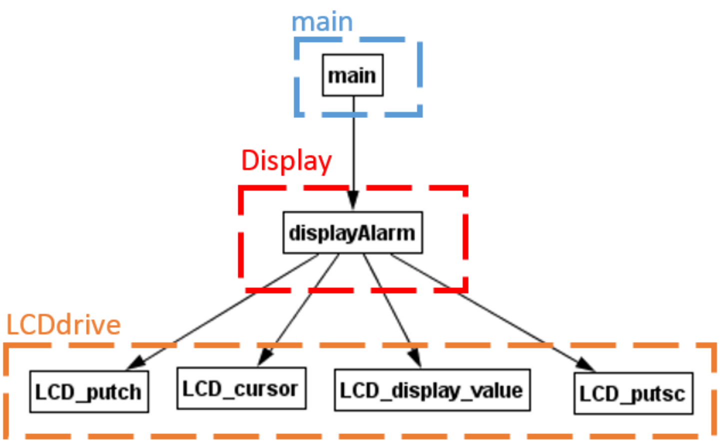


Figure 2.2.2

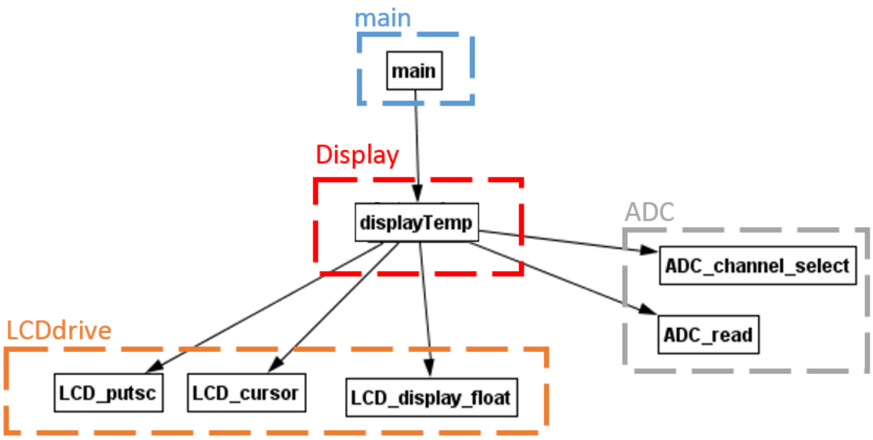


Figure 2.2.3

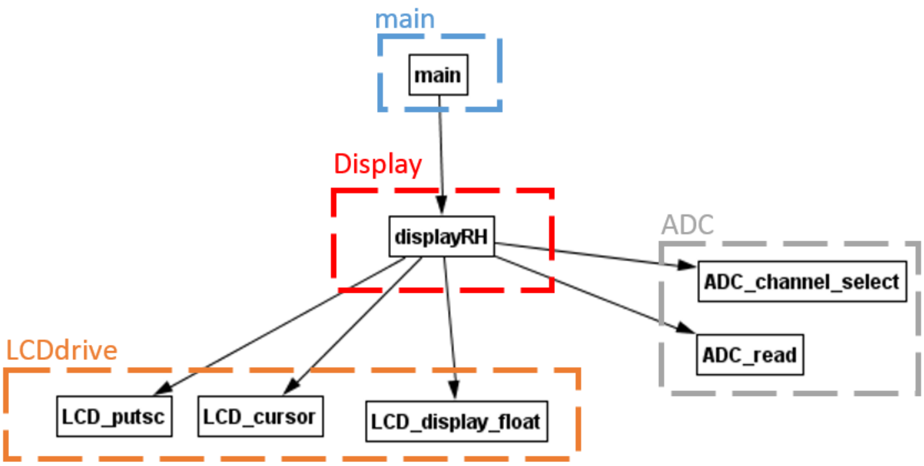


Figure 2.2.4

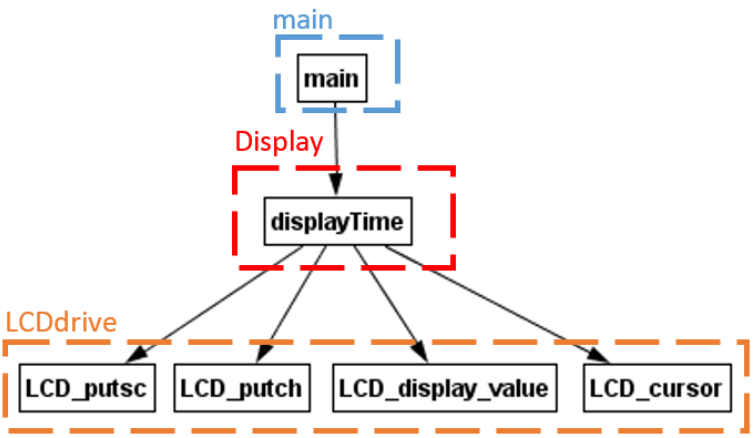


Figure 2.2.5

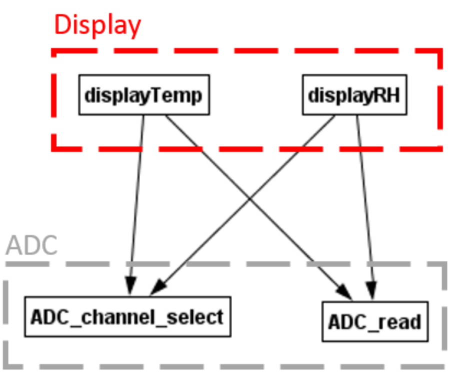


Figure 2.2.5

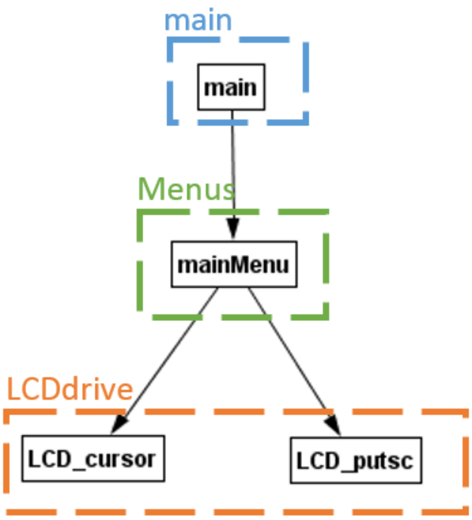


Figure 2.2.6

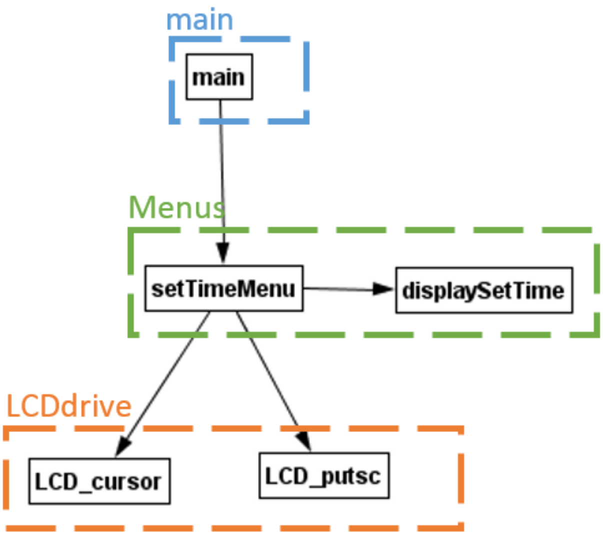


Figure 2.2.7

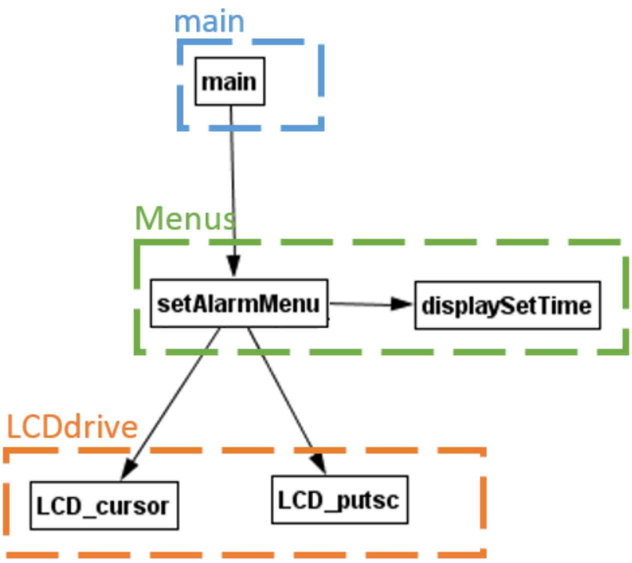


Figure 2.2.8

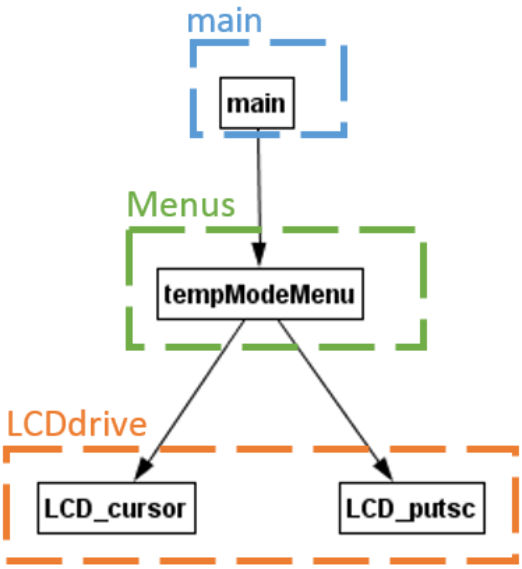


Figure 2.2.9

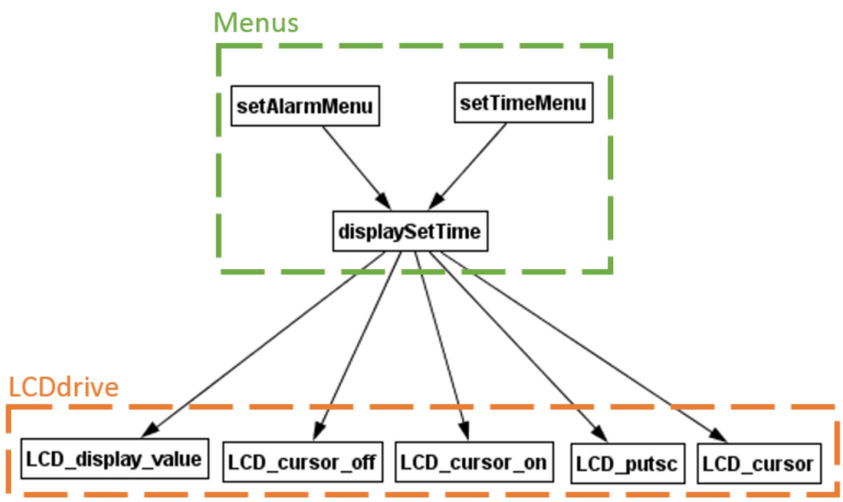
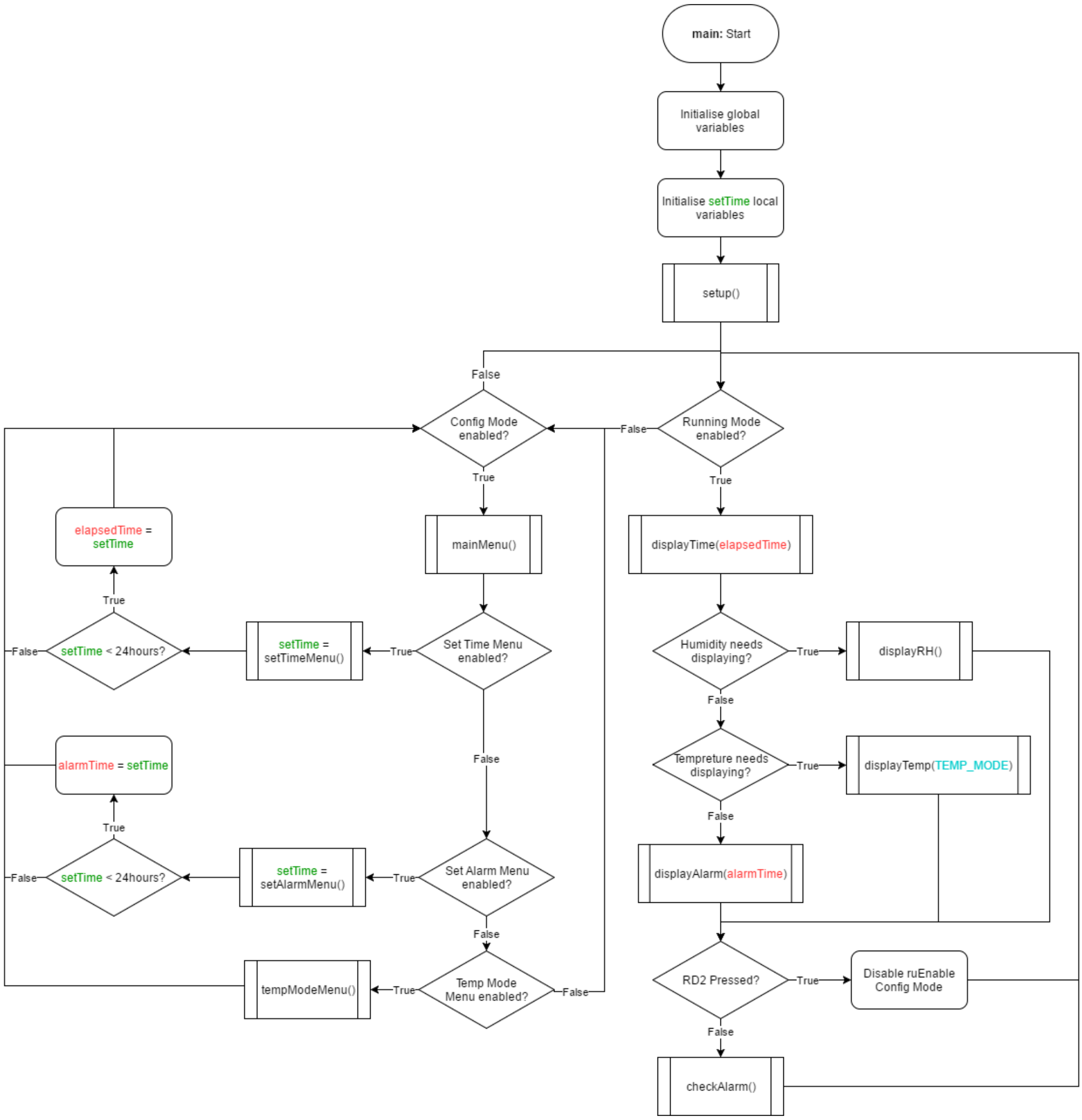
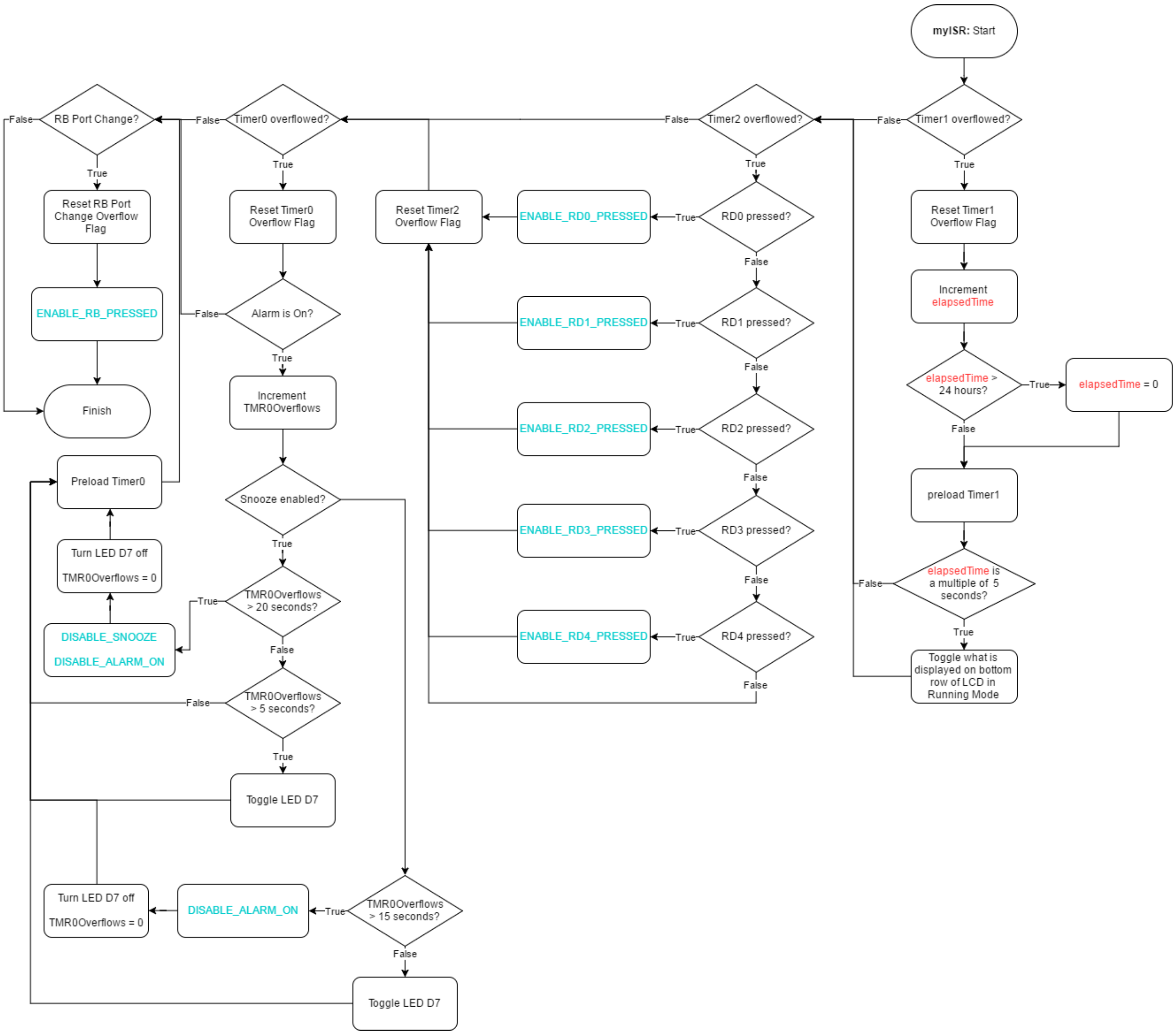


Figure 2.2.10

The flow charts below detail how the main.c file ties everything together.





myISR: Start
Timer1 overflowed?
Reset Timer1 Overflow Flag
Increment elapsedTime
elapsedTime > 24 hours?
elapsedTime = 0
preload Timer1
elapsedTime is a multiple of 5 seconds?
Toggle what is displayed on bottom row of LCD in Running Mode
Timer2 overflowed?
RD0 pressed?
ENABLE_RD0_PRESSED
RD1 pressed?
ENABLE_RD1_PRESSED
RD2 pressed?
ENABLE_RD2_PRESSED
RD3 pressed?
ENABLE_RD3_PRESSED
RD4 pressed?
ENABLE_RD4_PRESSED
Reset Timer2 Overflow Flag
Timer0 overflowed?
Reset Timer0 Overflow Flag
Alarm is On?
Increment TMR0Overflows
Snooze enabled?
TMR0Overflows > 20 seconds?
DISABLE_SNOOZE
DISABLE_ALARM_ON
Turn LED D7 off
TMR0Overflows = 0
Preload Timer0
TMR0Overflows > 5 seconds?
Toggle LED D7
TMR0Overflows > 15 seconds?
DISABLE_ALARM_ON
Turn LED D7 off
TMR0Overflows = 0
Toggle LED D7
RB Port Change?
Reset RB Port Change Overflow Flag
ENABLE_RB_PRESSED
Finish
True
False

3. Conclusion

Overall, the project can be considered a success as the design choice made did end up making the end user experience very good. The alarm clock was very responsive and RAM usage was around 37%. Most of the individual objectives were met as delays were only ever used as switch debounces meaning ever bit of the code, including the IRS was running very quickly. Variables were always assigned to the smallest possible type to reduce RAM usage. Local variables were used frequently to improve the reusability of function, reduce RAM usage and increase data safety. The end program was readable thanks to the use of a custom Macros library, making it easy to maintain and update in the future. The main areas for improvement would be splitting up the main.c file into a few more functions to make it a little more readable and redesigning some of the program so the LCD only updates when something has changed to improve the overall efficiency of my program.