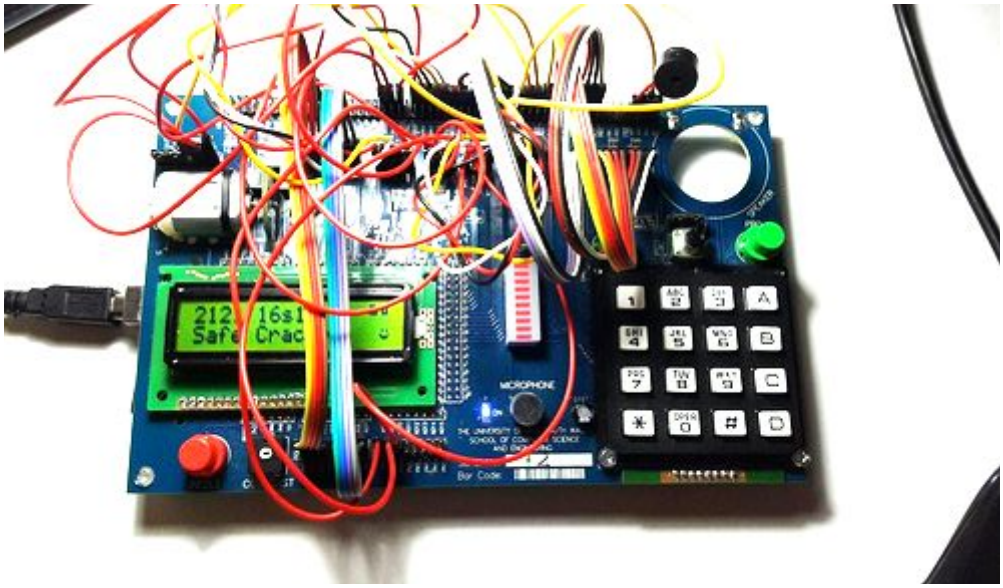
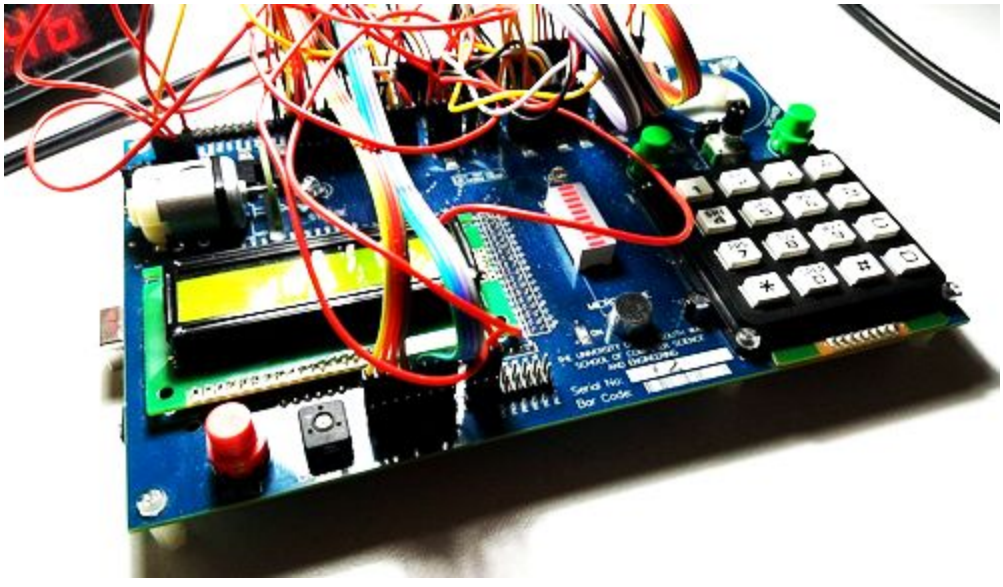
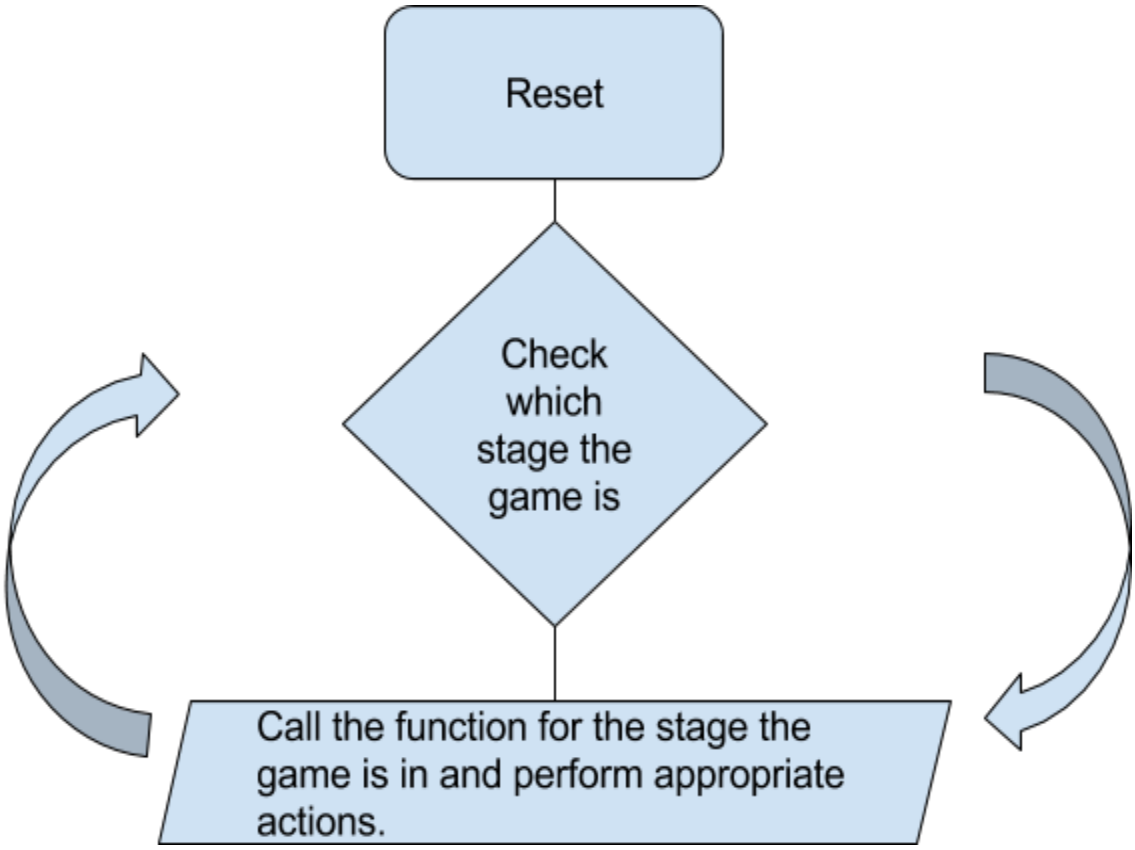


DESIGN MANUAL



SYSTEM FLOW CONTROL

Main Control



1. Our program starts off in RESET, which initialises the stack pointer, data segment variables, ports and port directions, the LCD and LCD backlight, the interrupts associated with the buttons and the settings for various timers. The interrupts that are enabled at this point are the interrupts associated with the buttons and the timer associated with the backlight. All other interrupts are enabled based on keypad and button input.

2. The program then goes into TIMER0, which is the timer that runs every 781 overflows (unscaled) to check what stage the game is in. Whilst the timer is continually check what stage the game is in, it will traverse through the stages and perform the function call of the stage where the game is in then return back to TIMER0 which will continue running throughout the game process.

Keypad Control

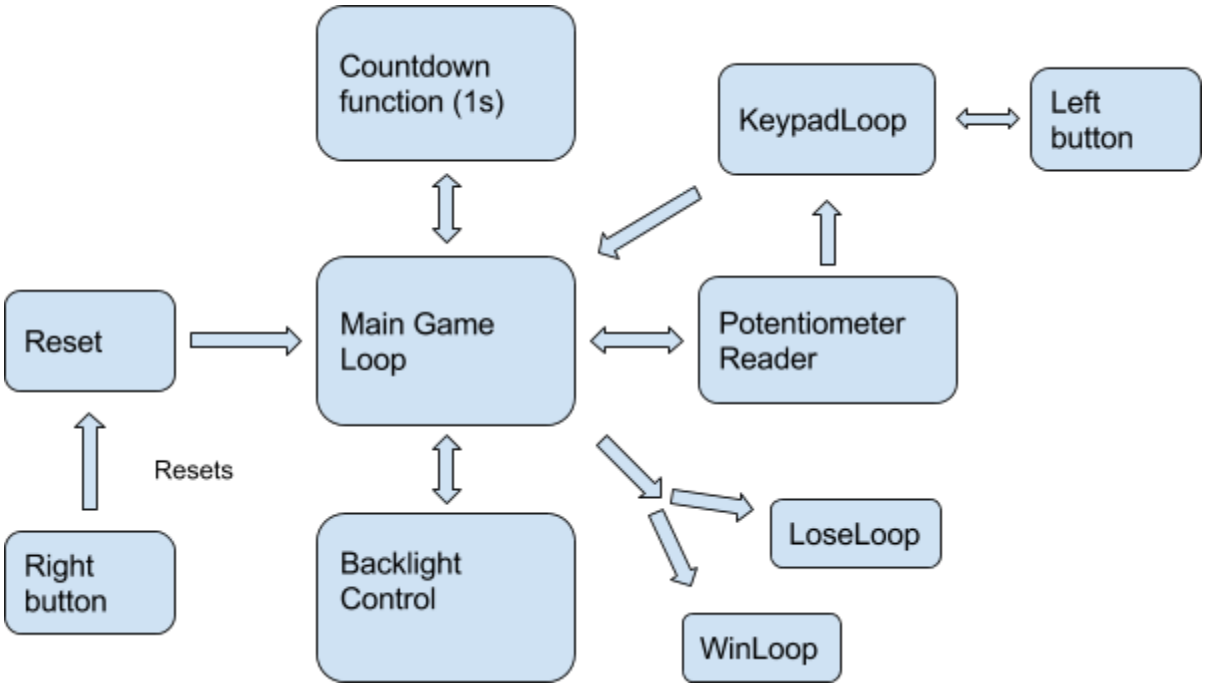
On the screens where the user is required to enter data or when entering data would yield any operation within the game, the program scans the keypad for input. The program then decides what to do with this input based on the current mode. We record the current mode in a register and that is initialised to entry mode. The stages of the game are allocated constants.

The Keypad is utilised for 4 purposes throughout the stages of the game. In order of occurrence:

1. START SCREEN:The keypad accepts the user's input of either A,B,C,D which is used to set the amount of seconds allocated for the countdown timer.
2. FIND POT POS SCREEN: The keypad accepts the user's input of any key which is compared with the random value generated by the timer.
3. ENTER CODE SCREEN: The keypad accepts the user's input of any key for the user to enter the code.
4. WIN/TIMEOUT SCREEN: The keypad accepts the user's input of any key to return the user to the start screen.

In addition, on the Start, Win and Timeout screens if a keypress is not made by the user within 5 seconds, the backlight is turned off.

Entire System Control



DATA STRUCTURES

Registers

Registers were used for storing values of temporary variables as well as loading, storing and manipulating the values of the data segment variables.

- Four registers are specifically defined for use as storing the current row and column when scanning for keypad presses, and others for holding the values of the row mask and column mask.

```
row = r16       ; keypad current row number
col = r17       ; keypad current column number
rmask = r18     ; keypad mask for current row during scan
cmask= r19      ; keypad mask for current column during scan
```

- Two registers are defined as temporary registers, used for holding temporary values such as when loading and storing data segment variables, as well as for passing values into functions.

```
temp = r20      ; temp variable
temp2 = r21     ; second temp variable
```

- Two registers were used for debouncing, one for PB1 and one for the keypad.

```
keyButtonPressed    = r7 ; an internal debounce flag for the keypad
debounce = r2; a flag debouncing PB1, (used after PB1 is clicked when the
game is won or lost - to ensure the game doesn't restart)
```

- Two registers were used to hold the next stage and the current stage of the game.

```
screenStage = r3      ; the current stage the game is on
screenStageFol = r4 ; a delayed version of screenstage (for checking when
                      it is desired that the initial stage code has already run)
```

- One register was used to hold the random code and another for the difficulty countdown.

```
difficultyCount = r24          ; a register holding the countdown value for the
                                 current difficulty
keypadCode = r22    ; the 'random' code being  searched for on the keypad
```

- One register (r16) was used for calculations in the macros.
- The rest of the registers were used for, timer counters (countdown timer, screen stage, game state).

```
counter = r5  ; a generic countdown register
running = r6  ; a flag representing if the backlight should be on indefinitely
                on the current screen
curRound = r23 ; a counter representing the current round (used for
                 addressing memory)
```

```
gameShouldReset = r25 ; a boolean flag indicating the game is going to
                        reset
```

Data Segment Variables

Data segment variables of 1-4 bytes in length were used for storing the values of important data. In cases where it was most appropriate to store data in memory, and not registers due to purpose or lack thereof, data memory was allocated.

GENERAL VARIABLES

```
gameloopTimer:      .byte 2    ; counts number of timer overflows for gameloop
counterTimer:       .byte 2    ; counts number of timer overflows for counter
keypadTimer:              .byte 2    ; counts number of timer overflows for
keypad
randomcode:         .byte max_num_rounds; stores the 'random' keypad items
highScores:         .byte 4  ; a byte array to store current highscores
```

BACKLIGHT VARIABLES

```
BacklightCounter:    .byte 2  ; counts timer overflows
BacklightTime:       .byte 1  ; counts number of seconds to trigger
                              backlight fade out
BacklightFadeCounter: .byte 1 ; used to pace the fade in process

BacklightFade:        .byte 1   ; flag indicating current backlight process -
                                stable/fade in/fade out
BacklightPWM:         .byte 1 ; current backlight brightness
```

SPEAKER VARIABLES

```
speakerCounter:       .byte 1 ; number of loops so far for the speaker timer
speakerCounterGoal: .byte 1 ; number of loops to do for the duration
```

Constants

Constants were set using .set and .equ and were using to help make the code more easily readable and maintainable. Constants were used for I/O Port Masks, states within the game (win,timeout, reset potentiometer, enter code) the debouncing time length for the timers, LCD backlight states (fading in, fading out, stable) and LCD instructions.

KEYPAD

```
.equ PORTLDIR              = 0xF0    ; PD7 - 4: output, PD3 - 0, input
.equ INITCOLMASK           = 0xEF    ; scan from the rightmost column,
```

```
        .equ INITROWMASK        = 0x01  ; scan from the top row
        .equ ROWMASK            = 0x0F  ; mask for the row
SPEAKER
        .equ speaker250         = 61    ; number of overflows for Timer4 to last
                                         250ms
        .equ speaker500         = 122   ; number of overflows for Timer4 to last
                                         500ms
        .equ speaker1000        = 244   ; number of overflows for Timer4 to last
                                         1000ms
GENERAL CONSTANTS
        .equ max_num_rounds     = 3     ; the number of rounds that will
                                         be played
POTENTIOMETER
        .equ counter_initial    = 3     ; the intial countdown value (3 seconds)
        .equ pot_pos_min        = 22    ; the minimum value on the pot (got from
                                         testing it)
        .equ pot_pos_max        = 1004  ; the maximum value on the pot (got
                                         from testing it)
STAGE OF THE GAME
        ;screen stage values representing the stage we are on
        .equ stage_start        = 0
        .equ stage_countdown    = 1
        .equ stage_pot_reset    = 2
        .equ stage_pot_find     = 3
        .equ stage_code_find    = 4
        .equ stage_code_enter   = 5
        .equ stage_win          = 6
        .equ stage_lose         = 7
```

Strings (CSEG)

Strings were stored for the use of writing to the screen using the *do_lcd_write_str* macro. A 1 between strings denotes a newline, and the final 0 represents end of the string pattern. Any other 0s before the final one represents padding to fit within a byte in CSEG memory. These “strings” are just byte arrays, but deserve their own section as they are ASCII characters and not just data values.

```
str_home_msg:       .db  "2121 16s1",      1,  "Safe Cracker",0,      0
str_keypadscan_msg: .db  "Position found!", 1,  "Scan for number",     0
```

```
str_findposition_msg:   .db  "Find POT POS",   1,  "Remaining:   ",   0
str_timeout_msg:        .db  "Game over",      1,  "You Lose!",       0
str_win_msg:            .db  "Game complete",  1,  "You Win!", 0,     0
str_reset_msg:          .db  "Reset POT to 0", 1,  "Remaining:   ",   0
str_countdown_msg:      .db  "2121 16s1",      1,  "Starting in ", 0, 0
str_entercode_msg:      .db  "Enter Code",     1,                     0
```

Byte Arrays (CSEG)

Byte arrays were stored in CSEG for the purpose of calling upon to store in LCD memory as custom characters. These are represented to memory the exact same way as strings, however these are not purposed to be ASCII values, and just made up 8 lines of a 5 pixel wide character.

```
lcd_char_smiley:    .db    0x00, 0x00, 0x0A, 0x00, 0x11, 0x0E, 0x00, 0x00
lcd_char_two:       .db    0x0E, 0x02, 0x04, 0x0F, 0x00, 0x00, 0x00, 0x00
lcd_char_one:       .db    0x0C, 0x04, 0x04, 0x0E, 0x00, 0x00, 0x00, 0x00
lcd_char_five:      .db    0x0E, 0x08, 0x06, 0x0E, 0x00, 0x00, 0x00, 0x00
lcd_char_six:       .db    0x0E, 0x08, 0x0E, 0x0E, 0x00, 0x00, 0x00, 0x00
lcd_char_zero:      .db    0x0E, 0x0A, 0x0A, 0x0E, 0x00, 0x00, 0x00, 0x00
```

EEPROM

The EEPROM was utilized to save the game state between uses of the system. This type of memory (Electrically Erasable Programmable Read-Only Memory) was used to store the most recently played difficulty, and high scores for each level of difficulty.

EEPROM Data Map:

0x0000	*Current Difficulty*
0x0001	*Initialize Flag*
0x0002	*Difficulty 20 Highscore*
0x0003	*Difficulty 15 Highscore*
0x0004	*Difficulty 10 Highscore*
0x0005	*Difficulty 6 Highscore*

When the board first switches on, it checks if the value 0xAA (a value chosen to be know as the *code* identifying that the EEPROM has been initialized before) is stored in EEPROM address 0x0001. If it is, it continues on to load the difficulty into the

difficultyCount register, and the high score values in the *highScores* portion of data memory.

Every time the user is presses one of the Alphabetic keys whilst on the home screen, the difficulty is changed, and stored in the difficultyCount register. When this occurs, the difficulty value is also stored in EEPROM in address 0x0000.

When the user reaches the *Find Code* screen, the system checks if the current countdown value (difficultyCount - counter) is greater than that of the currently stored value in the respective position in the *highScores* data memory block. If it is (representing a faster, or better, score), then the highScores data memory segment is updated and the score is written to the correct location in EEPROM.

MODULE SPECIFICATIONS

MODULE	FUNCTIONS	INPUT	OUTPUT
Buttons	**Initialise buttons**		**Trigger interrupt on falling edges Enable external interrupts**
Delay	**Sleep 1ms Sleep 5ms Sleep 20ms**		
LCD	**Initialise LCD**		**Initialise LCD Clear digits variable**
	Display START SCREEN	**PB1 - start the game Keypad- A/B/C/D to select difficulty**	**Clear screen then proceed to next screen Initialise difficulty**
	Display COUNTDOWN SCREEN		**Writes “2121 16s1\n Starting in 3...” Clear screen and display the**

			countdown from 3 seconds
	Display RESET POT SCREEN	**Temp register containing potentiometer reading**	**Writes "Reset POT to 0" on the top half of the screen and "Remaining: ?" on the bottom half of the screen.** **The countdown is written to the LCD where the "?" is with the time depending on the difficulty selected**
	Display FIND POT POS SCREEN	**Temp register containing potentiometer reading**	**Writes "Find POT Pos" on the top half of the screen and "Remaining: ?" on the bottom half of the screen.** **Same countdown continued before to be written to LCD at "?"**
	Display FIND CODE SCREEN		**Writes "Position Found!" on the top half of the screen and "Scan for number" on the bottom half.**
	Display ENTER CODE SCREEN	**Temp register containing code**	**Writes "Enter Code" to the top half of the screen.** **Then writes "*" for every code the user writes that is correct.**
	Clear CODE		**Clear the bottom half of the screen if the user enters the wrong code.**
	Display GAME COMPLETE SCREEN		**Write "Game complete" on the top half of the screen and "You Win!" on the bottom half.**
	Display TIMEOUT SCREEN		**Write "Game over" on the top half of the screen and "You Lose!" on the bottom half of the screen.**
LCD Backlight	**Initialise backlight and**		**Clears related variables** **Initialise PWM in Timer 3**

	timer		**Initialise OVF in Timer 2** **Enable interrupt in Timer 2**
	Enable backlight fade out		**Sets backlight fade state to fade out**
	Enable backlight fade in		**Clears related variables** **Sets backlight fade state to fade in**
Speaker	**Initialise timer sound**		
	Play timer sound		**250ms beep for every countdown**
	Initialise new round sound		**500ms beep**
	Play finished sound		**1000ms beep on arriving at the Win or Loss screen**
Motor	**Start Motor**	**Correct key press/hold at Find Code Screen**	**Motor spins**
	Stop Motor	**Correct key released at Find Code Screen**	**Motor stops spinning**
Keypad	**Initialise keypad timer**	**Registers that will be used**	**Registers that will be used, all now cleared**
	Initialise column scan		**A value in col, representing the column of a pressed button, if any**
	Delay scan		
	Initialise row scan		**A value in row, representing the row of a pressed button, if any**

	Initialise convert	**A key on the keypad has been pressed**	**Depending on the key that has been pressed. As outlined in button functions in the user manual**
Converter Function	**asciiconv**	**Temp Register**	**Converts the contents in the register to ascii equivalent then displays on LCD**
Macro	**LCD Writer**	**String in memory**	**Displays the string on the LCD**
	cpii	**Register & immediate value**	**Performs cpi with a register below r16**
	addi	**Register & immediate value**	**Performs adi with a register below r16**
	ldii	**Register & immediate value**	**Performs ldi with a register below r16**
	toggle	**Timer enable register, and a value to write**	**Toggles a timer on or off**
	writeTo EEPROM	**EEPROM address, and register containing value to write**	**Data stored in EEPROM**
	readTo EEPROM	**EEPROM address**	**Data read from EEPROM now stored in *temp* variable**
	speakerBeep For	**Time for speaker to beep for**	**Audible sound for the set duration, from the speaker**
	toggleStrobe		**Strobe light will toggle**
	cleanAllReg		**All registers are cleared**
	do_lcd_write_ str	**16 bit address containing CSEG**	**The string will then be written to the LCD**

		string	
	do_lcd_bottom		**LCD cursor will go to the bottom row**
	do_lcd_clear		**LCD will be cleared and cursor returned home**
	do_lcd_data	**A register containing data**	**The data stored in the passed in register will be written to the screen**
	do_lcd_show_custom	**The number of the custom character**	**The custom character that is represented by the passed in number will be written to the top right of the screen**
	do_lcd_store_custom	**The character number, and address to character in CSEG**	**The character will be stored to the desired character number in the LCD memory**
	enable_ADC		**The ADC will be enabled**
	disable_ADC		**The ADC will be disabled**
	do_lcd_data_i	**A raw data value (8 bit)**	**The raw data value will be written to the screen**
	do_lcd_command	**A raw data value (8 bit)**	**The 8 bit data value will be sent to the LCD control pins**
	clear_datamem	**16 bit data memory address**	**The address passed in, and its consecutive address will be cleared with a 0 in memory**

INTERRUPTS	**INPUT**	**OUTPUT**
Reset		**Initialises stack pointer** **Initialises data direction registers and clears port**

		registers **Initialises LCD** **Initialises turntable** **Initialises motor** **Initialises buttons** **Initialises backlight timer** **Initialises finished sounds** **Initialises a number of variables**
Right Button		**Returns to RESET**
Left Button		**Jumps to the Countdown screen if pressed on start screen** **Jumps to RESET screen if pressed on WIN/TIMEOUT screen**
Timer 0 Overflow	**Local counter stored in data segment**	**If local counter reaches 781 (0.s), check to see which stage the game is in and perform appropriate action.**
Timer 1 Overflow	**Local counter stored in data segment**	**If second is reached, checks which stage the game is in an displays the appropriate countdown for the stage.**
Timer 2 Overflow	**Local counter stored in data segment**	**Enable when keypad is required.** **Every 390 overflows (unscaled) check for any keypad presses through traversing the keypad.**

Timer 3 Overflow	**Local counter stored in data segment**	**If local counter reaches 15, backlight state is fade in and we have reached maximum brightness, set backlight state to stable** **If local counter reaches 15 and backlight state is fade in, increment OCR3BL** **If local counter reaches 15, backlight state is fade out and we have reached minimum brightness, set backlight state to stable** **If local counter reaches 15 and backlight state is fade out, decrement OCR3BL** **If mode is not running and local seconds counter reaches 5 (5s), enable backlight fade out.**
ADC Complete		**The ADC value which is used to determine the position of the pot for the reset pot screen, and the find pot screen.**

ALGORITHMS

Game Loop (TIMER0):

The game loop was run every 100ms, and every time it ran, a check was performed for the current game state. When the game state matched up with one of the conditional statements, then the respective code was executed, and the loop was able to restart.

```
gamelooptimer = gamelooptimer  + 1
if(gamelooptimer == 781) { // This means it has been 100ms
        gameloopTimer = 0
        if(screenstage == stage_countdown)
                countdownFunc()
                return
        else if(screenstage == stage_pot_reset)
                potResetFunc()
                return
        else if(screenstage == stage_pot_find)
                potFindFunc()
                return
        else if(screenstage == stage_code_find)
                codeFindFunc()
                return
        else if(screenstage == stage_code_ enter)
                codeEnterFunc()
                return
        else if(screenstage == stage_win)
                winFunc()
                return
        else if(screenstage == stage_lose)
                loseFunc()
                return
        else
                return
```

Initial Countdown Setup:

This code segment is only ran the FIRST time the game loop evaluates and calls the countdownFunc() function, on all further evaluations whilst the screenStage REMAINS as stage_countdown will not execute this code.

```
screenStage = stage_countdown
print(countdown_msg)
temp = 3
convertToAsciiAndPrint(temp) // The algorithm for this function is outlined below
print("...")
enableTimer1() // This timer actually performs the ‘‘countdown” on-screen
```

One-second timer (TIMER1)

This code segment effectively counts up a register every one second, and performs a different task depending on the current screen stage. If on the ‘stage_countdown’, the Loop will write out to the screen ‘3’, then after a second will replace this with ‘2’ etc. until at which point is has reached the countdown limit, and will sound the speaker and move on to the Reset Pot state. If in either the reset pot, or find pot state, the game will write out the time remaining to the screen whilst it is greater than 0, and will sound the speaker for 250ms on every counter decrement. If the counter reaches 0, the game is transition to the Timeout screen stage.

```
counttimer = counttimer + 1
if(counttimer == 30)  // This means it has been 1 second
        countimer = 0
        counter = counter + 1
        if(screenstage == stage_countdown)
                temp = counter_initial
                temp = temp - counter // Get the total countdown time remaining
                if(temp == 0)  // game has timed out
                        screenstage = stage_pot_reset
                        counter = 0
                        countimer = 0
                        SoundTheSpeaker(500ms)
                        RandomisePotLocation()
                        return
```

```
            convertToAsciiAndPrint(temp)
            return

        else if(screenstage == stage_stage_pot_reset OR
            screenstage == stage_stage_pot_find)

            temp = currentDifficulty
            temp = temp - counter
            if(temp == 0)
                screenstage = stage_lose
                return
        else
            SoundTheSpeaker(250ms)
            convertToAsciiAndPrint(temp)
            Return
```

Handle Backlight Fade and Display (TIMER3):

This segment counts up 30 overflows continually and checks to see which stage the backlight is in. If it is stable, reset and continue timer, if it is fadein check if the PWM is less than 0xFF if it is increment and write to output, if it is fadeout check if the PWM is greater than 0 if it is decrement and write to output. However if the the running register does not equal 1 then that means it is during gameplay and thus the backlight does not fade. If is is during gameplay begin a one second countdown, tally it and if 5 seconds are reached then begin fade out.

```
    backlightfadecounter = backlightfadecounter  + 1
        if(backlightfadecounter == 30)
            backlightfadecounter = 0
            if (backlightstate == BACKLIGHT_FADEIN)
                if(backlightPWM < 0xFF) {
                    backlightPWM++
                else
                    backlightstate = BACKLIGHT_STABLE
            else if (backlightstate == BACKLIGHT_FADEOUT)
                if(backlightpwm > 0)
```

```
                    backlightPWM = backlightPWM  - 1
                else
                    backlightstate = BACKLIGHT_STABLE

    if(mode!=RUNNING)
        backlightcounter++
        if(backlightcounter == 7812)
            backlightcounter = 0;
            backlightseconds++;
        if (backlightseconds == 5) {
            backlightseconds = 0;
            backlightstate = BACKLIGHT_FADEOUT;
```

Convert to ASCII (asciiconv):

This algorithm, given a number stored in the register temp, will make comparisons with the constant number of ten, to tally the number of times ten can be subtracted from it. Once this is no more, the remaining value left in temp is the decimal number of ones. This algorithm can be extended to work with the hundreds, thousands etc. columns, but only tens and ones were needed for the assignment.

```
    numTens = 0
    numOnes = 0

    while(temp >= 10) {
        numTens++
        temp = temp - 10
    }
        numOnes = temp
        print(numTens + '0')
        print(numOnes + '0')
```

Write String to LCD

This algorithm takes a string written and displays it to the LCD without requiring individual characters send to the LCD

```
do_lcd_clear = 0
zl = low(@0)
zh = high(@0)

while(r17!=0)
        if (r17 != 0) {
                do_lcd_bottom
        }
        do_lcd_data (r17)
```

Speaker

This algorithm will count up to 250ms and continually exclusively or a register then send it to the output ports to produce a sound from the speaker in the form of a square wave.

```
Input to PortB
temp2 = 1
EOR temp, temp2
Outpt to PortB
temp speakerCounter
temp++
speakerCounter = temp
temp2 = speakerCounterGoal
if(temp!= temp2) {
        return
}
Turn off Timer4
temp = 0
speakerCounter = temp
```

Get random data for POT position

This algorithm is used to retrieve “random” data for the purpose of randomizing the POT location in the RESET Pot and Find Pot screens. Also for the use in picking a “random” key for the keypad when in the findCode screen

Temp2 = Get current Timer3 Low 8 bits (This is random as it has been running since launch)
Store Temp2 as low 8 bits of POT location
Temp = Mask low 2 bits of Temp2
keyCode = Mask low 4 bits of Temp2
Store Temp as high lowest 2 bits of the higher byte of POT location
Use keyCode as the position of the "random" key in the keypad on findCode screen

Write DATA to EEPROM

This algorithm is used to write data to the EEPROM. It is stored as a macro to be called upon with a value representing the EEPROM address to store at, and a register containing the data desired to store.

If the last write hasn't completed yet
    Check again if the write has completed

Set 2 byte EEPROM address to write to in address register
Write to desired data to EEPROM data register
Write a logical one to EEMPE
Start EEPROM write by setting EEPE

return

Read DATA from EEPROM

This algorithm is used to read data from the EEPROM. It is stored as a macro to be called upon with a value representing the EEPROM address to read from. The read data is stored in the temp register.

If the last write hasn't completed yet
    Check again if the write has completed

Set 2 byte EEPROM address to write to in address register
Start eeprom read by writing EERE
Read data from Data Register into temp register

return