FACULTY OF ELECTRONIC AND ELECTRICAL ENGINEERING

UNIVERSITY OF LEEDS

ELEC3662 - Embedded Systems Report

Alexander Bolton – 200938078
Thomas Preston – 200963806

Contents

Project Outline

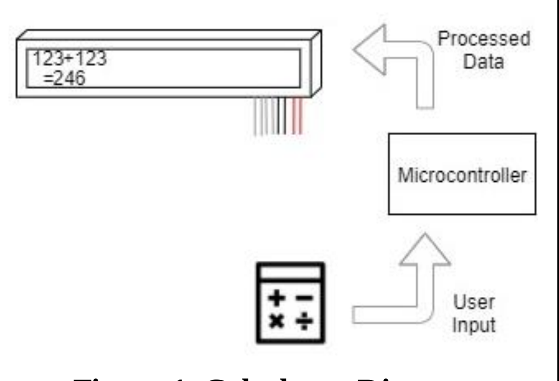


Figure 1. Calculator Diagram

The purpose of this project was to interface a microcontroller, the Tiva TM4C123, with external hardware components. This consisted of a 16 x 2 Liquid Crystal Display (LCD) and a 4 x 4 keypad. Utilising this hardware, and the programming language 'C'. The goal was to design and implement the software to act in conjunction with the interfaced hardware to create a calculator able to perform simple mathematical calculations. With the keypad acting as an input, the user inputs numbers and operators which were then fed into the microcontroller with the resultant output being displayed on the LCD screen, a diagram of which is shown in figure 1.

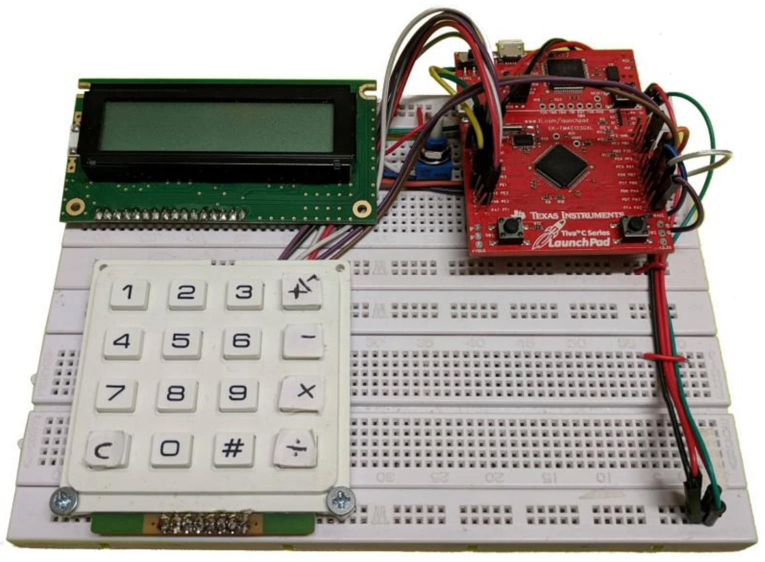


Figure 2. Calculator Picture

Circuit Diagram

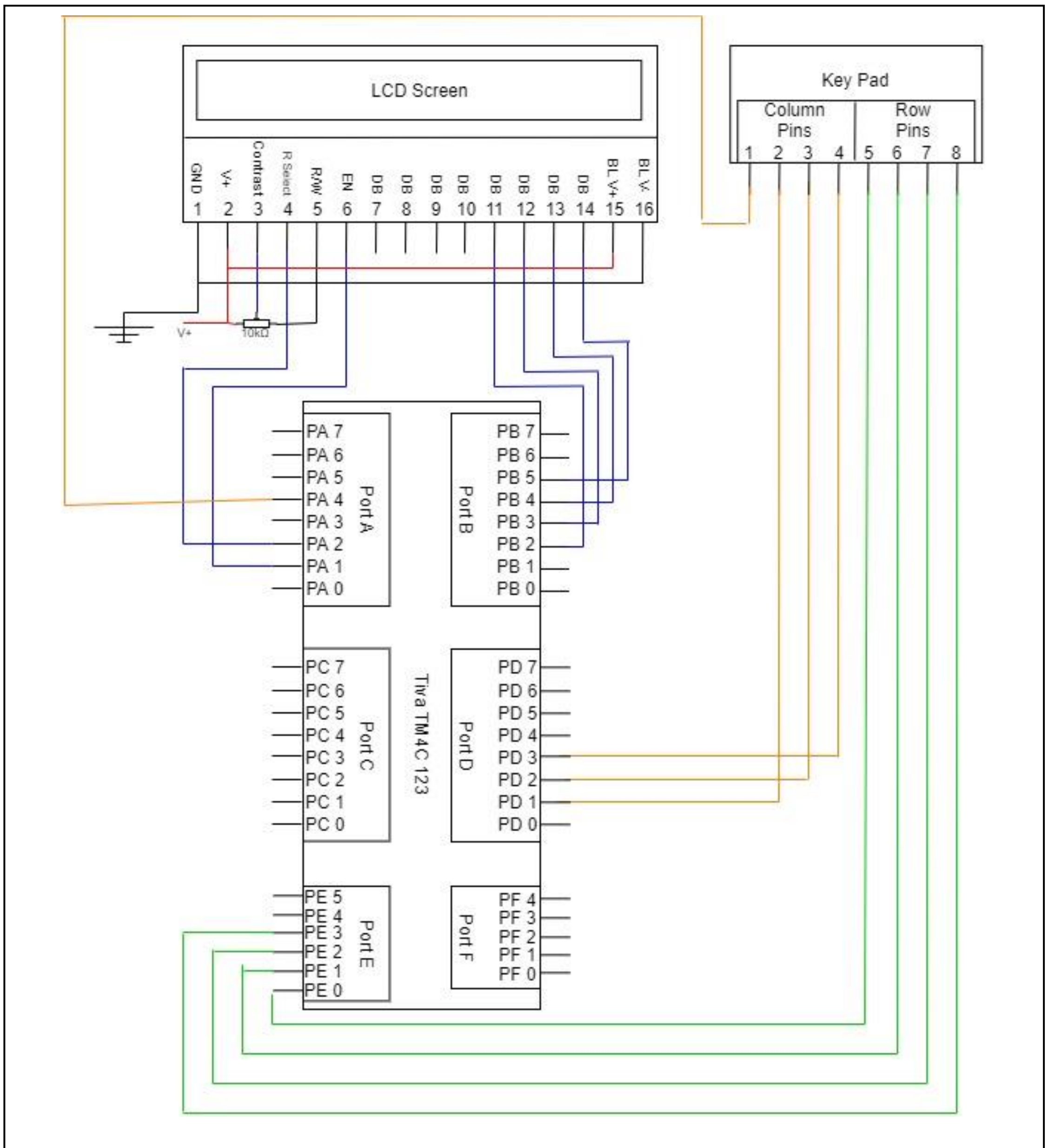


Figure 3. Circuit Diagram

Key Pad

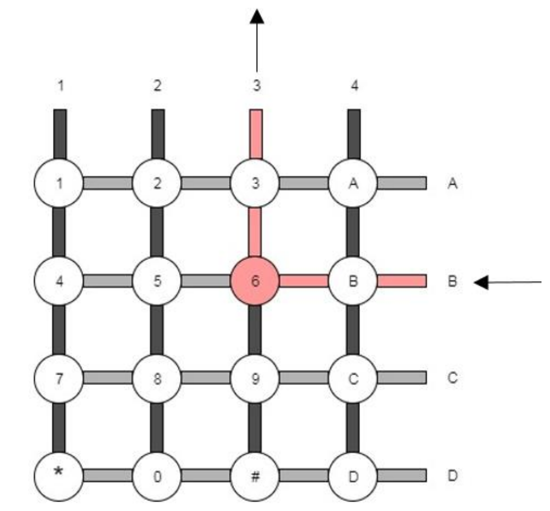


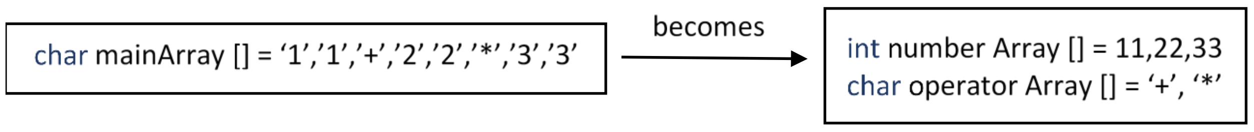
Figure 26. Keypad Diagram

The keypad consists of a four by four matrix of buttons. It operates by sending power through each row individually, when a button is pressed it forms a connection giving the voltage a path to follow, outputting through the columns [1], shown in **figure 24**. With each column and row being connected to an individual pin this allows the microcontroller to turn on each column sequentially in a loop, waiting for a response on the output, this polling of the keypad reduced the average power usage as it is not needed to be constantly on. It also allows for each individual key to have a unique address, which can be used within the program to assign symbols, we can see in **figure 26** that the symbol ‘6’ has the address B3.

We achieved this programmatically by establishing a matrix of symbols and operands to match the symbols on the keypad. This was done to sync the keypads physical button addressed with the code, so button A1 would be 1:1 in the matrix, and both correspond to symbol ‘1’.

By utilising ‘For loops’ the microcontroller can scan the keypad continuously, with one loop moving through the rows, A to B to C etc. Within this is a subsequent loop within this to scan the columns, A1 to A2 etc. Simply, it scans every column for each row. This is achieved by turning the output pin for each row high individually, and then checking the input pins one by one in quick succession. Once a button press has been detected the code matches this to the symbol matrix, returning the value to a global variable to be used within another library.

Calculations



```
char mainArray [] = ‘1’,’1’,’+’,’2’,’2’,’*’,’3’,’3’
```

```
int number Array [] = 11,22,33
char operator Array [] = ‘+’, ‘*’
```

Figure 27. Array Separation Example

Calculations were performed at the end of a user’s input phase, this was indicated by pressing the equals symbol. An interrupt was set up to detect this. To perform the required calculations the code was required to translate the pressed buttons to variables, achieved within the keypad library. These variables were then placed into an array. With each individual key press corresponding to a symbol that is subsequently stored individually within the array, meaning ’11 + 1’ would be stored as ‘1, 1, +, 1’. To perform calculations the main array is then manipulated splitting into two subset arrays, consisting of a number array and operand array. This is achieved by reading the array by checking if the stored value at each index is an integer, it does this until a non-integer (operand) is found, at which point it collates the previous indexes together to form one whole number while moving the operand to the separate array, shown in **figure 27**.

To perform the subsequent mathematical calculations inputted, while applying the rules of BIDMAS the code loops through the operand array, first counting the amount of operands then counting the amount of individual operands of each type. With this information, it can perform all the multiplications before any additions etc. Then the code takes the position of these operands within the array, and performs the desired calculation on the integers within the number array with corresponding index values, replacing those integers with the answer to the operation, before shifting the array to fill the gap. This process loops until there is only one value stored within the number array which is the calculations final answer.
A check is then performed on the answer before printing to the screen, to determine if it is negative. If so, the modulus of the answer is taken to attain a positive value. However, a flag is set which indicates to the print function to print a minus symbol before the numerical symbols.

LCD Screen

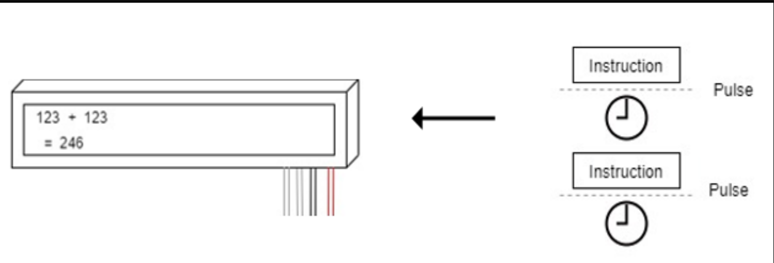
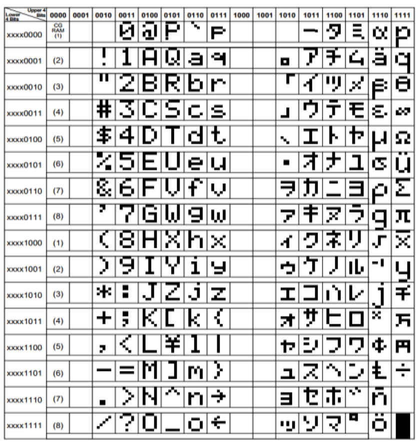


Figure 28. LCD Initialisation Diagram

The project guidelines stated that we must use the LCD in its four-bit mode variation, this allowed for a reduction in pin usage as only a nibble was being sent. However, because of this it was required to initialise the screen in a specific way, found within the screens datasheet. [2] The entailed pulsing precise information into the screen, meaning that five pins were set to high for a brief period before being returned to zero, this was achieved programmatically by a functioned called '*EnPulse'*. After which it was required to wait for a set amount of time, for the screen to register the instruction before another could be sent.

After initialisation, the screen could receive and display desired data. To display symbols on the screen, the screen had to be sent a specific pulse of information that corresponds with that symbol, predetermined by the manufacturer. A tabulated version of this was found within the datasheet and utilised. [2] Within the program this information was concatenated into a case machine, with a separate case to pulse the information into the screen for every symbol. The code would then determine which symbol was required and select the necessary case.

Within the screen library there were additional usability functions. These functions consisted of a curser go to function, which enabled the code to select a position of the curser using X and Y co-ordinates and an alternative to this referred to as *'newLine'* which was a more efficient, albeit less precise version that allowed for the code to alternate between the first and second lines allowing for faster printing. A clear screen function was implemented in a similar manner to the initialisation function, pulsing pins low, waiting and repeating these steps to initiate a clear command within the screen. As the symbol selection code was structured within a case machine, it allowed for the implementation of a *'printString'* function to be added. This function would cycle through a string, assigning each symbol to an array variable, then utilising the previously discussed symbol determination code to cycle through the array and case machine the string would be printed onto the screen.



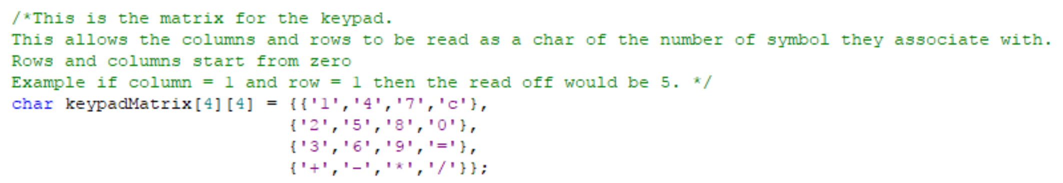
Upper 4 Bits / Lower 4 Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)			0	@	P	`	p				ー	タ	ミ	α	p
xxxx0001	(2)		!	1	A	Q	a	q			。	ア	チ	ム	ä	q
xxxx0010	(3)		"	2	B	R	b	r			「	イ	ツ	メ	β	θ
xxxx0011	(4)		#	3	C	S	c	s			」	ウ	テ	モ	ε	∞
xxxx0100	(5)		$	4	D	T	d	t			、	エ	ト	ヤ	μ	Ω
xxxx0101	(6)		%	5	E	U	e	u			・	オ	ナ	ユ	σ	ü
xxxx0110	(7)		&	6	F	V	f	v			ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111	(8)		'	7	G	W	g	w			ァ	キ	ヌ	ラ	g	π
xxxx1000	(1)		(	8	H	X	h	x			ィ	ク	ネ	リ	√	x̄
xxxx1001	(2)		)	9	I	Y	i	y			ゥ	ケ	ノ	ル	⁻¹	y
xxxx1010	(3)		*	:	J	Z	j	z			ェ	コ	ハ	レ	j	千
xxxx1011	(4)		+	;	K	[	k	{			ォ	サ	ヒ	ロ	×	万
xxxx1100	(5)		,	<	L	¥	l	|			ャ	シ	フ	ワ	¢	円
xxxx1101	(6)		-	=	M	]	m	}			ュ	ス	ヘ	ン	£	÷
xxxx1110	(7)		.	>	N	^	n	→			ョ	セ	ホ	゛	n̄	
xxxx1111	(8)		/	?	O	_	o	←			ッ	ソ	マ	゜	ö	█

Figure 29. Symbols Table [2]

ALEXANDER BOLTON - 200938078
THOMAS PRESTON - 200963806

Flow Charts

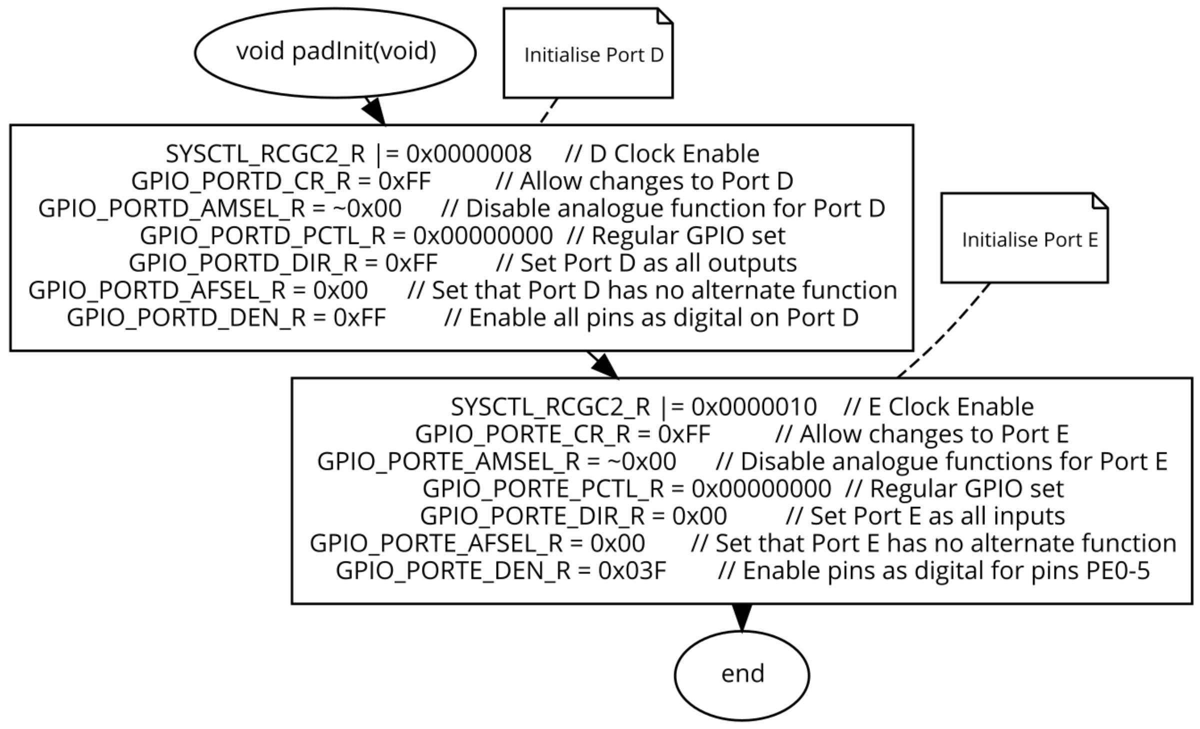
Keypad Functions



```
/*This is the matrix for the keypad.
This allows the columns and rows to be read as a char of the number of symbol they associate with.
Rows and columns start from zero
Example if column = 1 and row = 1 then the read off would be 5. */
char keypadMatrix[4][4] = {{'1','4','7','c'},
                           {'2','5','8','0'},
                           {'3','6','9','='},
                           {'+','-','*','/'}};
```

Figure 4. Keypad Library

Pad Initialisation



```
SYSCTL_RCGC2_R |= 0x0000008     // D Clock Enable
GPIO_PORTD_CR_R = 0xFF          // Allow changes to Port D
GPIO_PORTD_AMSEL_R = ~0x00      // Disable analogue function for Port D
GPIO_PORTD_PCTL_R = 0x00000000  // Regular GPIO set
GPIO_PORTD_DIR_R = 0xFF         // Set Port D as all outputs
GPIO_PORTD_AFSEL_R = 0x00       // Set that Port D has no alternate function
GPIO_PORTD_DEN_R = 0xFF         // Enable all pins as digital on Port D
```

```
SYSCTL_RCGC2_R |= 0x0000010     // E Clock Enable
GPIO_PORTE_CR_R = 0xFF          // Allow changes to Port E
GPIO_PORTE_AMSEL_R = ~0x00      // Disable analogue functions for Port E
GPIO_PORTE_PCTL_R = 0x00000000  // Regular GPIO set
GPIO_PORTE_DIR_R = 0x00         // Set Port E as all inputs
GPIO_PORTE_AFSEL_R = 0x00       // Set that Port E has no alternate function
GPIO_PORTE_DEN_R = 0x03F        // Enable pins as digital for pins PE0-5
```

Figure 5. Calculation Initialisation Function Flow Chart

Key Pressed Detection Function

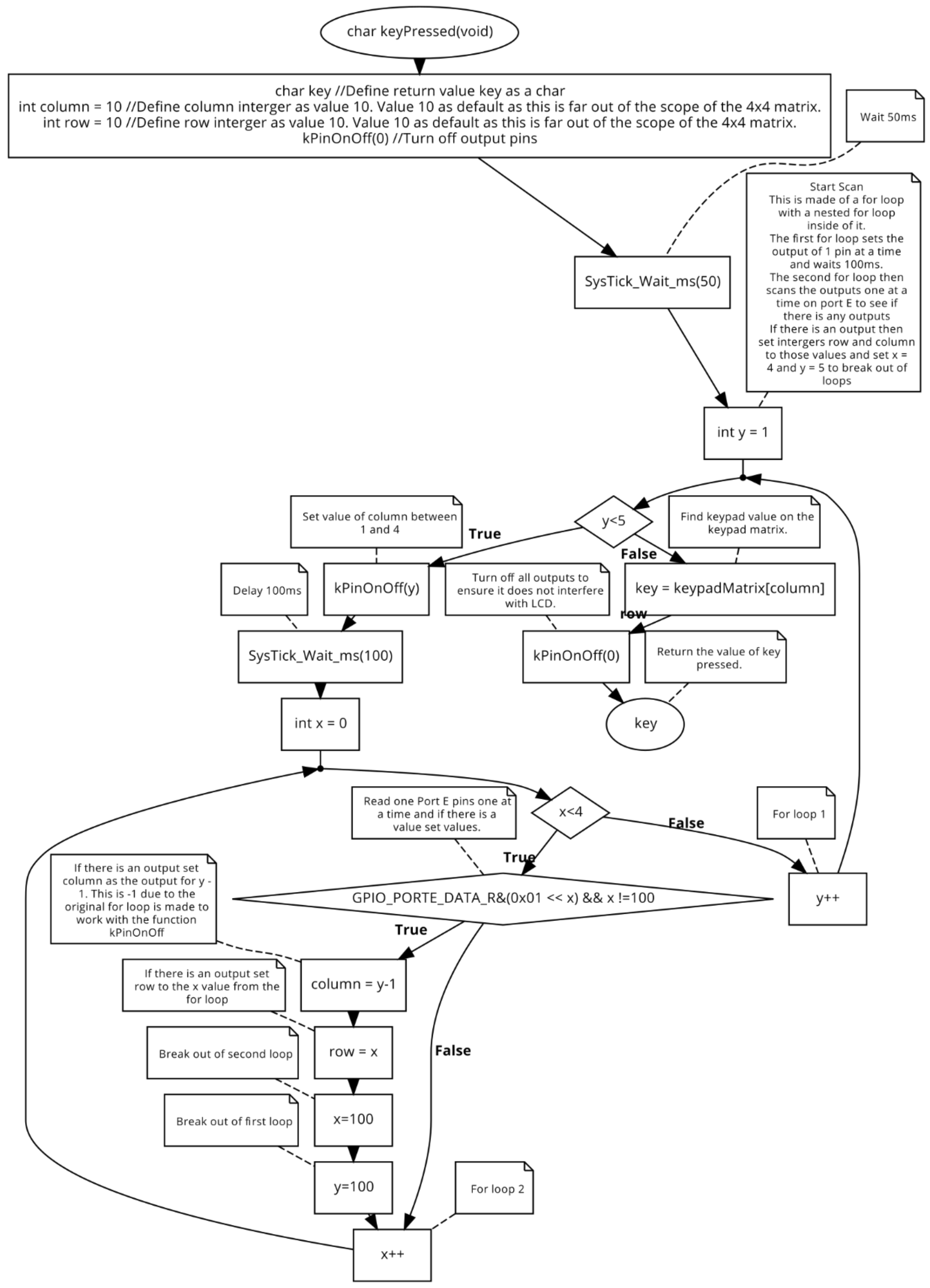

char keyPressed(void)
char key //Define return value key as a char
int column = 10 //Define column interger as value 10. Value 10 as default as this is far out of the scope of the 4x4 matrix.
int row = 10 //Define row interger as value 10. Value 10 as default as this is far out of the scope of the 4x4 matrix.
kPinOnOff(0) //Turn off output pins
Wait 50ms
SysTick_Wait_ms(50)
Start Scan This is made of a for loop with a nested for loop inside of it. The first for loop sets the output of 1 pin at a time and waits 100ms. The second for loop then scans the outputs one at a time on port E to see if there is any outputs If there is an output then set intergers row and column to those values and set x = 4 and y = 5 to break out of loops
int y = 1
y<5
True
False
Set value of column between 1 and 4
Find keypad value on the keypad matrix.
Delay 100ms
kPinOnOff(y)
Turn off all outputs to ensure it does not interfere with LCD.
key = keypadMatrix[column][row]
SysTick_Wait_ms(100)
kPinOnOff(0)
Return the value of key pressed.
key
int x = 0
Read one Port E pins one at a time and if there is a value set values.
x<4
False
For loop 1
True
If there is an output set column as the output for y - 1. This is -1 due to the original for loop is made to work with the function kPinOnOff
GPIO_PORTE_DATA_R&(0x01 << x) && x !=100
y++
True
If there is an output set row to the x value from the for loop
column = y-1
Break out of second loop
row = x
False
Break out of first loop
x=100
y=100
For loop 2
x++

Figure 6. Key Pressed Detection Function Flow Chart

Keypad Pin Toggle Function

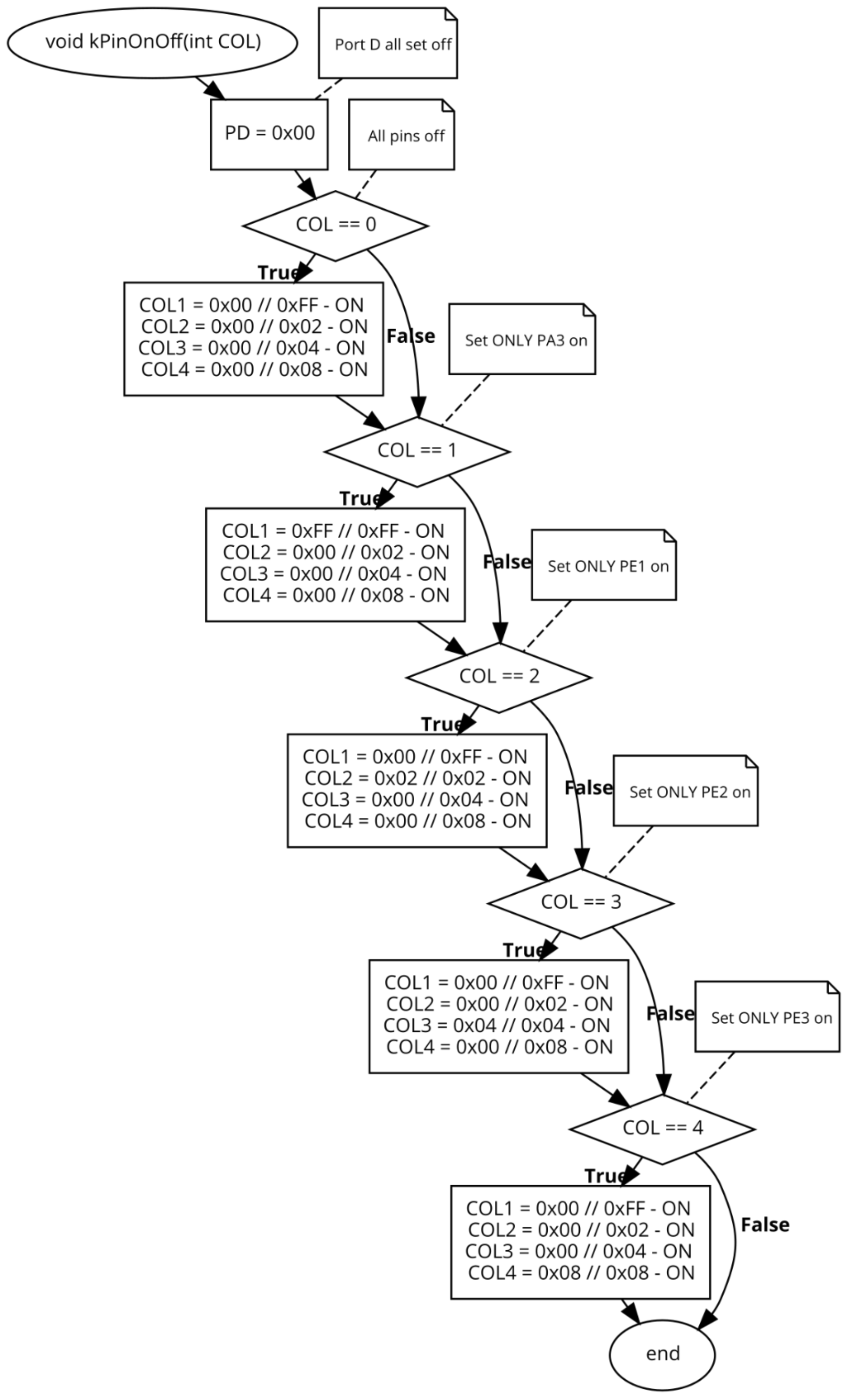
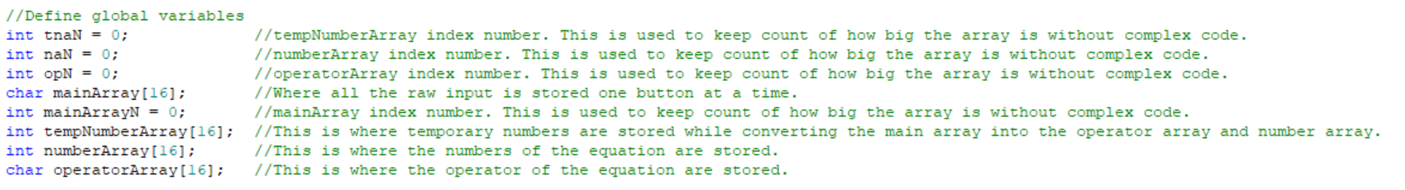


Figure 7. Keypad Toggle Function Flow Chart

Calculation Functions



```
//Define global variables
int tnaN = 0;              //tempNumberArray index number. This is used to keep count of how big the array is without complex code.
int naN = 0;               //numberArray index number. This is used to keep count of how big the array is without complex code.
int opN = 0;               //operatorArray index number. This is used to keep count of how big the array is without complex code.
char mainArray[16];        //Where all the raw input is stored one button at a time.
int mainArrayN = 0;        //mainArray index number. This is used to keep count of how big the array is without complex code.
int tempNumberArray[16];   //This is where temporary numbers are stored while converting the main array into the operator array and number array.
int numberArray[16];       //This is where the numbers of the equation are stored.
char operatorArray[16];    //This is where the operator of the equation are stored.
```

Figure 8. Calculation Library Variables

Calculator Initialisation Function

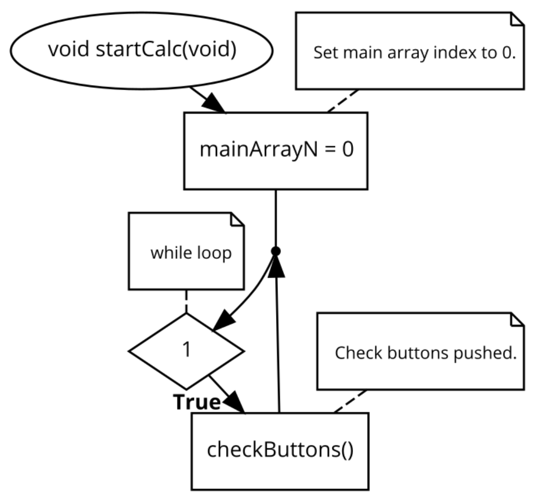


Figure 9. Calculator Initialisation Function Flow Chart

Check Button Press Function

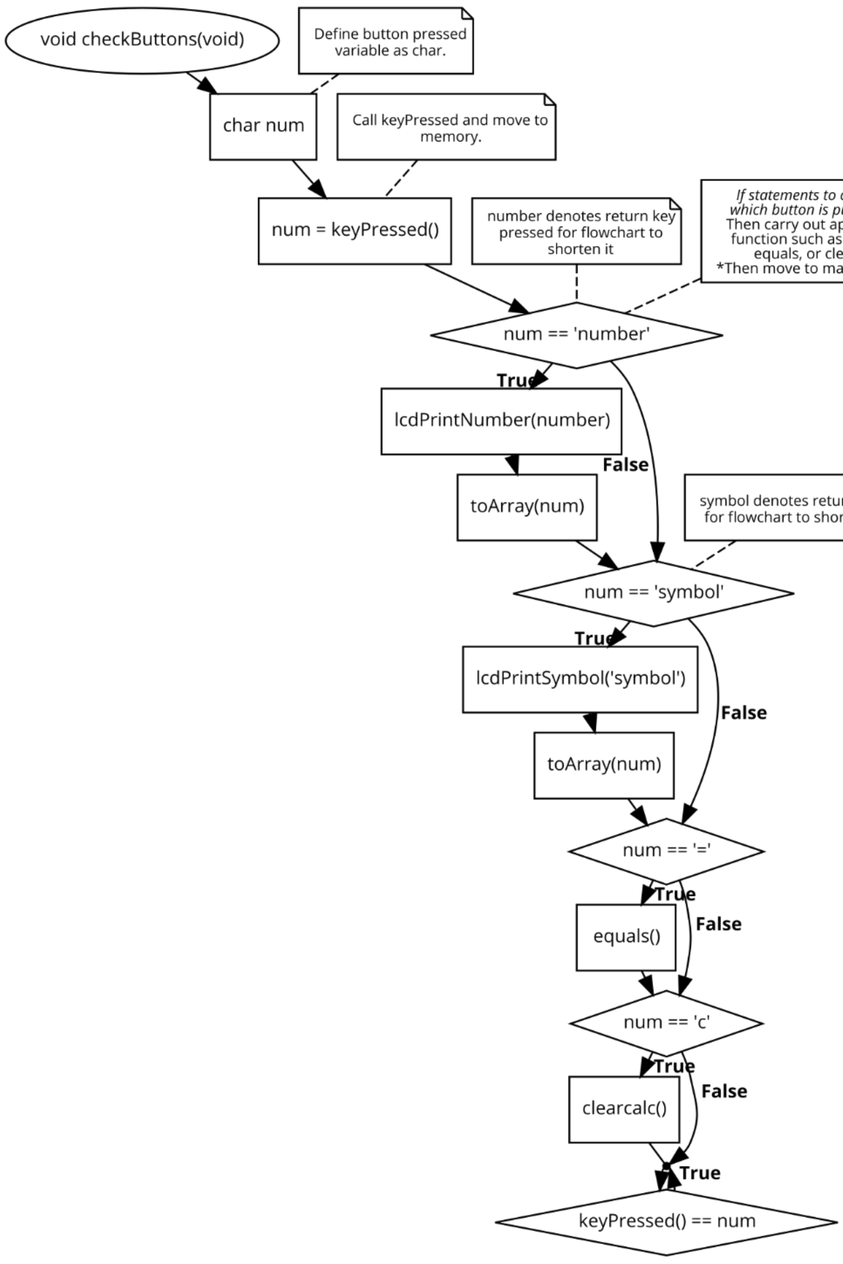

void checkButtons(void)
Define button pressed variable as char.
char num
Call keyPressed and move to memory.
num = keyPressed()
number denotes return key pressed for flowchart to shorten it
If statements to check which button is pushed. Then carry out appriate function such as print, equals, or clear *Then move to main array
num == 'number'
True
lcdPrintNumber(number)
False
toArray(num)
symbol denotes return key for flowchart to shorten it
num == 'symbol'
True
lcdPrintSymbol('symbol')
False
toArray(num)
num == '='
True
False
equals()
num == 'c'
True
False
clearcalc()
True
keyPressed() == num

Figure 10. Button Check Function Flow Chart

Equals Function

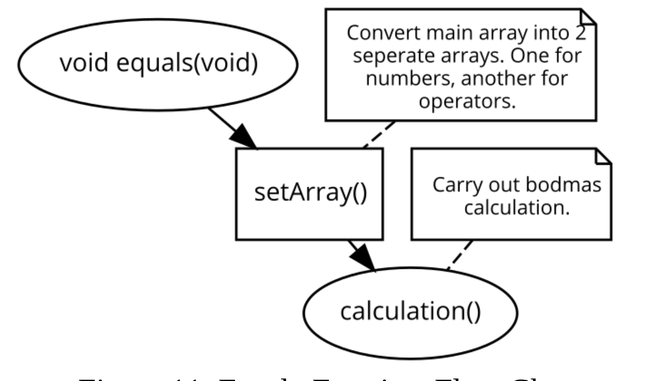

void equals(void)
Convert main array into 2 seperate arrays. One for numbers, another for operators.
setArray()
Carry out bodmas calculation.
calculation()

Figure 11. Equals Function Flow Chart

Array Building Function

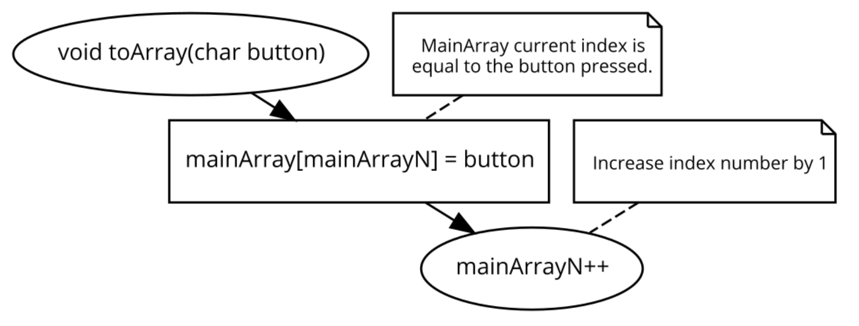

void toArray(char button)
MainArray current index is equal to the button pressed.
mainArray[mainArrayN] = button
Increase index number by 1
mainArrayN++

Figure 12. Array Building Function Flow Chart

Set Array Function

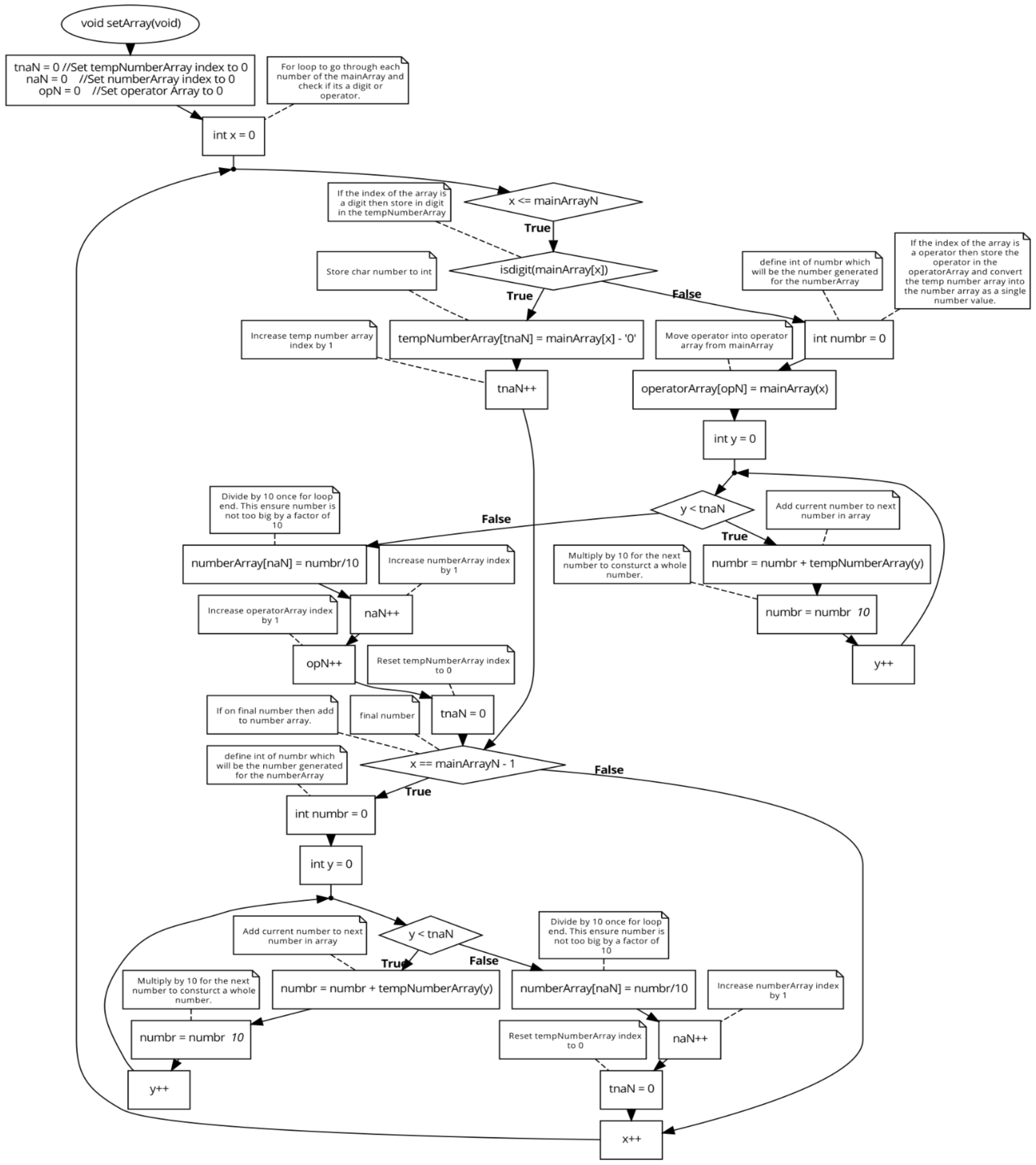

void setArray(void)
tnaN = 0 //Set tempNumberArray index to 0
naN = 0 //Set numberArray index to 0
opN = 0 //Set operator Array to 0
For loop to go through each number of the mainArray and check if its a digit or operator.
int x = 0
If the index of the array is a digit then store in digit in the tempNumberArray
x <= mainArrayN
True
If the index of the array is a operator then store the operator in the operatorArray and convert the temp number array into the number array as a single number value.
Store char number to int
isdigit(mainArray[x])
define int of numbr which will be the number generated for the numberArray
True
False
Increase temp number array index by 1
tempNumberArray[tnaN] = mainArray[x] - '0'
Move operator into operator array from mainArray
int numbr = 0
tnaN++
operatorArray[opN] = mainArray(x)
int y = 0
Divide by 10 once for loop end. This ensure number is not too big by a factor of 10
y < tnaN
Add current number to next number in array
False
True
numberArray[naN] = numbr/10
Increase numberArray index by 1
Multiply by 10 for the next number to consturct a whole number.
numbr = numbr + tempNumberArray(y)
Increase operatorArray index by 1
naN++
numbr = numbr 10
opN++
Reset tempNumberArray index to 0
y++
If on final number then add to number array.
final number
tnaN = 0
define int of numbr which will be the number generated for the numberArray
x == mainArrayN - 1
False
True
int numbr = 0
int y = 0
Add current number to next number in array
y < tnaN
Divide by 10 once for loop end. This ensure number is not too big by a factor of 10
True
False
Multiply by 10 for the next number to construct a whole number.
numbr = numbr + tempNumberArray(y)
numberArray[naN] = numbr/10
Increase numberArray index by 1
numbr = numbr 10
Reset tempNumberArray index to 0
naN++
y++
tnaN = 0
x++

Figure 13. Set Array Function Flowchart

Calculation Function

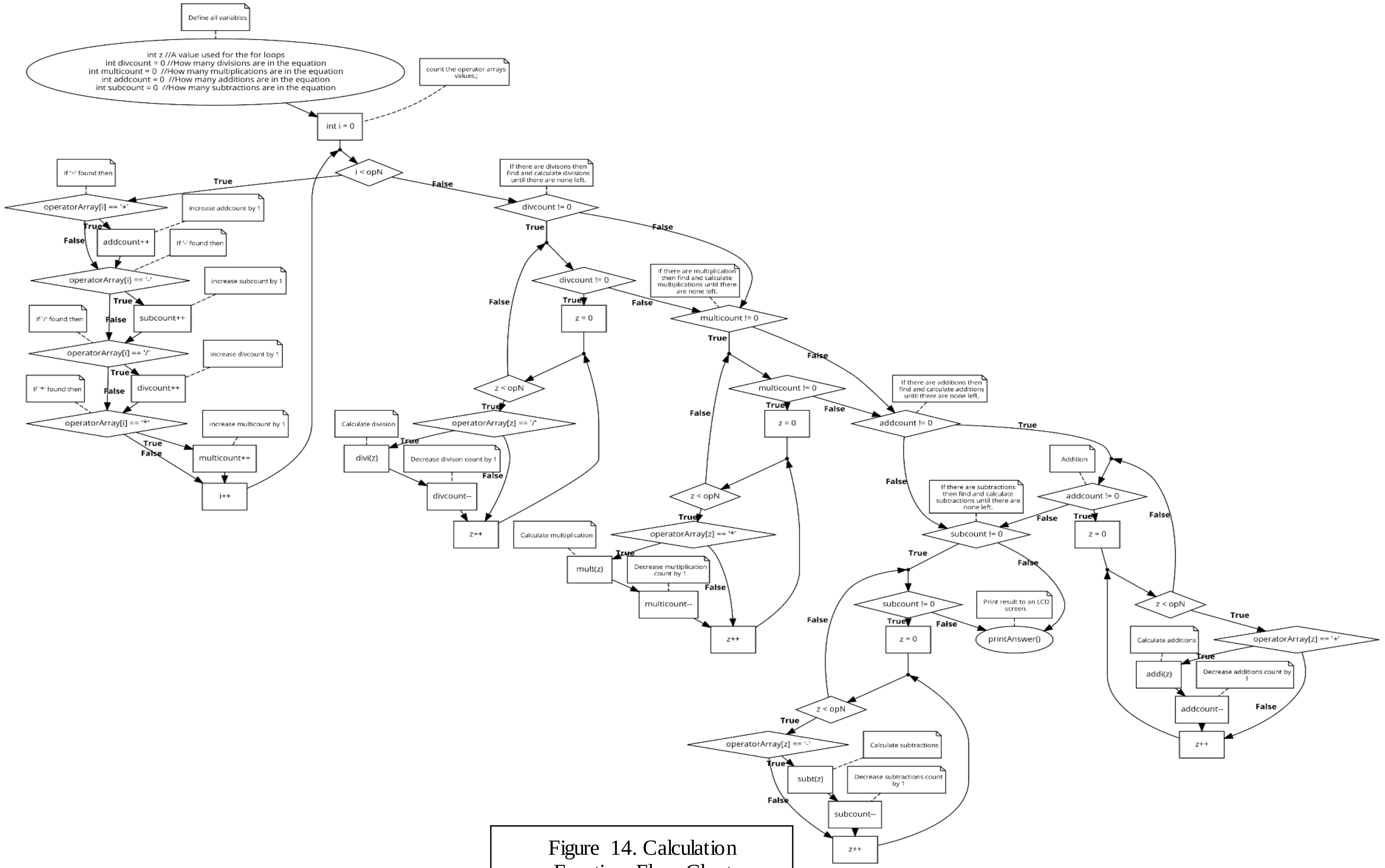

Define all variables
int z //A value used for the for loops
int divcount = 0 //How many divisions are in the equation
int multicount = 0 //How many multiplications are in the equation
int addcount = 0 //How many additions are in the equation
int subcount = 0 //How many subtractions are in the equation
count the operator arrays values.;
int i = 0
i < opN
True
False
If '+' found then
operatorArray[i] == '+'
increase addcount by 1
addcount++
If '-' found then
operatorArray[i] == '-'
increase subcount by 1
subcount++
If '/' found then
operatorArray[i] == '/'
increase divcount by 1
divcount++
If '*' found then
operatorArray[i] == '*'
increase multicount by 1
multicount++
i++
If there are divisons then find and calculate divisions until there are none left.
divcount != 0
divcount != 0
z = 0
z < opN
operatorArray[z] == '/'
Calculate division
divi(z)
Decrease divison count by 1
divcount--
z++
If there are multiplication then find and calculate multiplications until there are none left.
multicount != 0
multicount != 0
z = 0
z < opN
operatorArray[z] == '*'
Calculate multiplication
mult(z)
Decrease multiplication count by 1
multicount--
z++
If there are additions then find and calculate additions until there are none left.
addcount != 0
Addition
addcount != 0
z = 0
z < opN
operatorArray[z] == '+'
Calculate additions
addi(z)
Decrease additions count by 1
addcount--
z++
If there are subtractions then find and calculate subtractions until there are none left.
subcount != 0
subcount != 0
z = 0
z < opN
operatorArray[z] == '-'
Calculate subtractions
subt(z)
Decrease subtractions count by 1
subcount--
z++
Print result to an LCD screen.
printAnswer()

Figure 14. Calculation Function Flow Chart

Operator Functions (Divide, Multiply, Addition and Subtract)

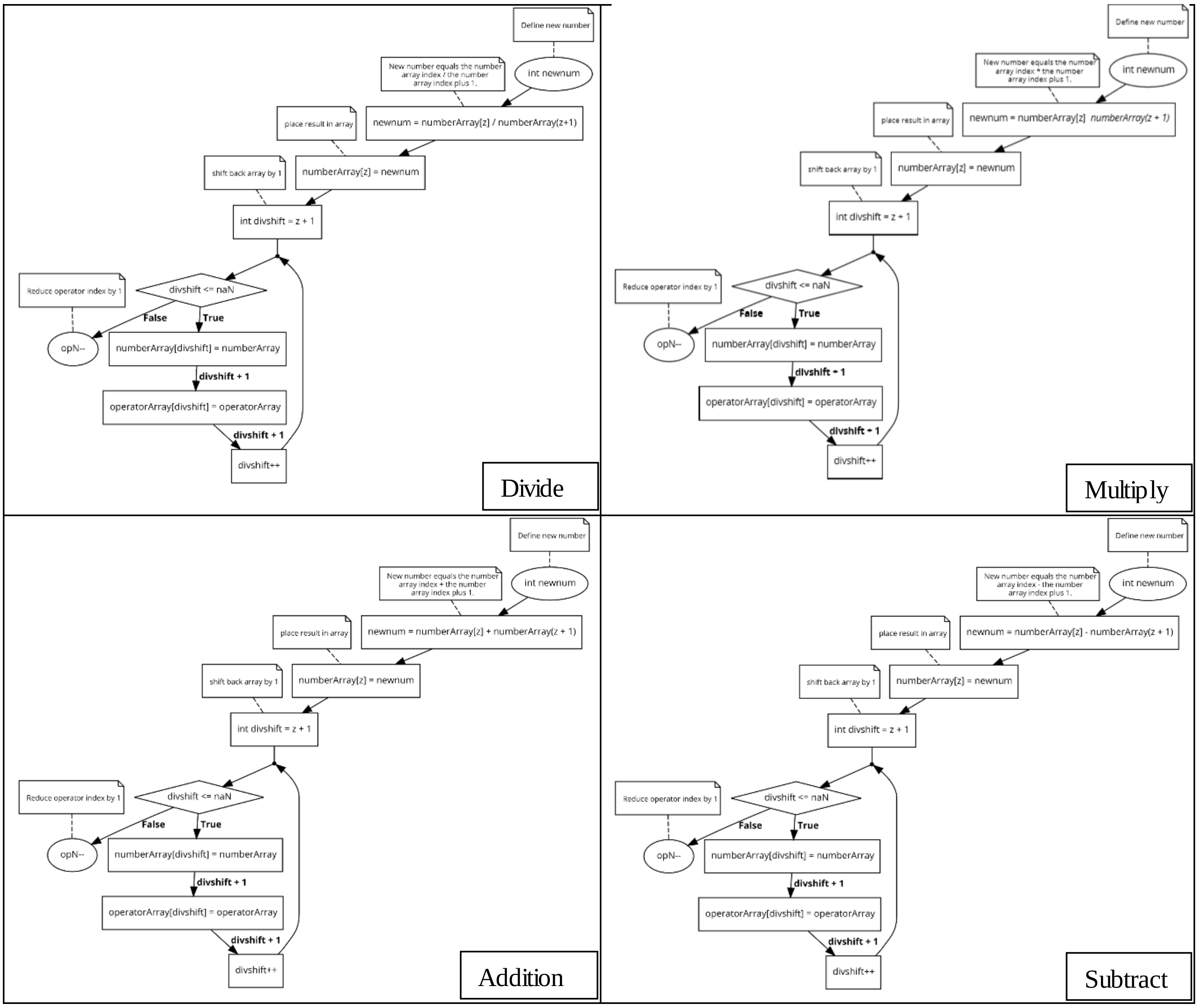

Define new number
int newnum
New number equals the number array index / the number array index plus 1.
newnum = numberArray[z] / numberArray(z+1)
place result in array
numberArray[z] = newnum
shift back array by 1
int divshift = z + 1
divshift <= naN
Reduce operator index by 1
False
True
opN--
numberArray[divshift] = numberArray
divshift + 1
operatorArray[divshift] = operatorArray
divshift + 1
divshift++
Divide
New number equals the number array index * the number array index plus 1.
newnum = numberArray[z] numberArray(z + 1)
Multiply
New number equals the number array index + the number array index plus 1.
newnum = numberArray[z] + numberArray(z + 1)
Addition
New number equals the number array index - the number array index plus 1.
newnum = numberArray[z] - numberArray(z + 1)
Subtract

Figure 15. Operand Functions Flow Charts

Print Answer Function

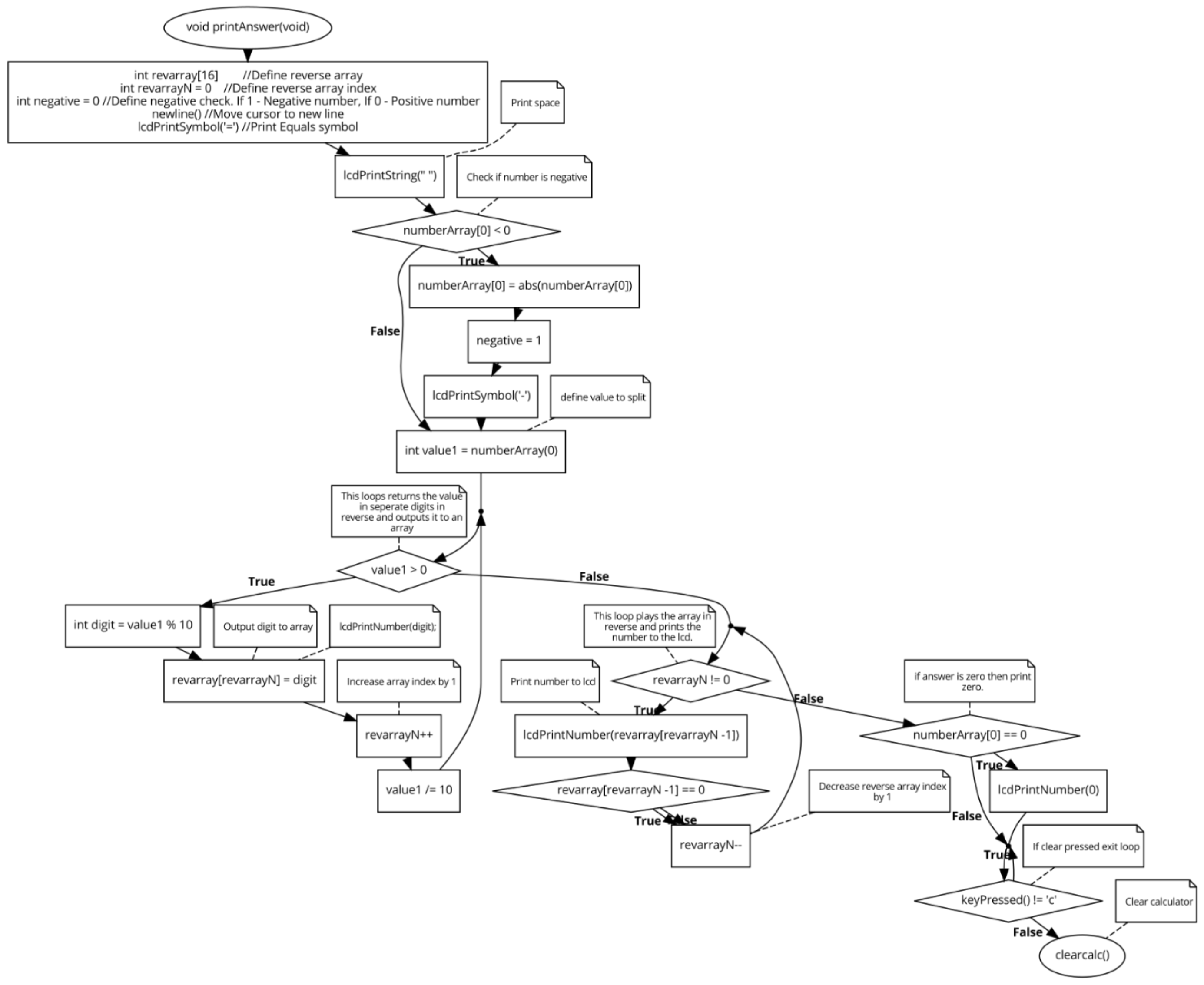

void printAnswer(void)
int revarray[16] //Define reverse array
int revarrayN = 0 //Define reverse array index
int negative = 0 //Define negative check. If 1 - Negative number, If 0 - Positive number
newline() //Move cursor to new line
lcdPrintSymbol('=') //Print Equals symbol
Print space
lcdPrintString(" ")
Check if number is negative
numberArray[0] < 0
True
False
numberArray[0] = abs(numberArray[0])
negative = 1
lcdPrintSymbol('-')
define value to split
int value1 = numberArray(0)
This loops returns the value in seperate digits in reverse and outputs it to an array
value1 > 0
True
False
int digit = value1 % 10
Output digit to array
lcdPrintNumber(digit);
revarray[revarrayN] = digit
Increase array index by 1
revarrayN++
value1 /= 10
This loop plays the array in reverse and prints the number to the lcd.
revarrayN != 0
Print number to lcd
True
False
lcdPrintNumber(revarray[revarrayN -1])
revarray[revarrayN -1] == 0
True
False
revarrayN--
Decrease reverse array index by 1
if answer is zero then print zero.
numberArray[0] == 0
True
False
lcdPrintNumber(0)
If clear pressed exit loop
True
keyPressed() != 'c'
False
Clear calculator
clearcalc()

Figure 16. Print Answer Function Flow Chart

Clear Screen Function

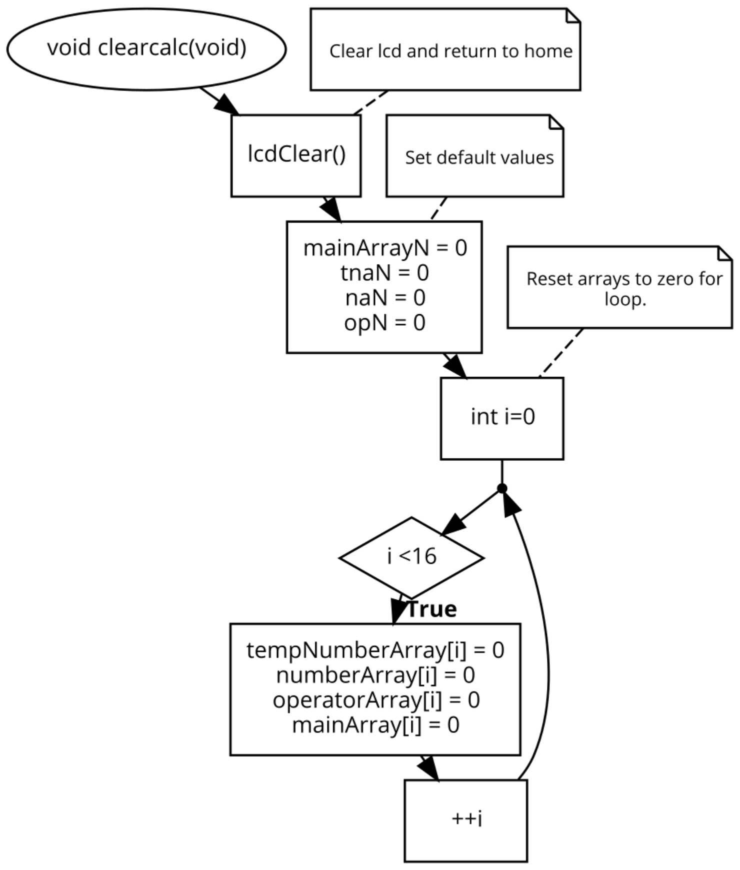
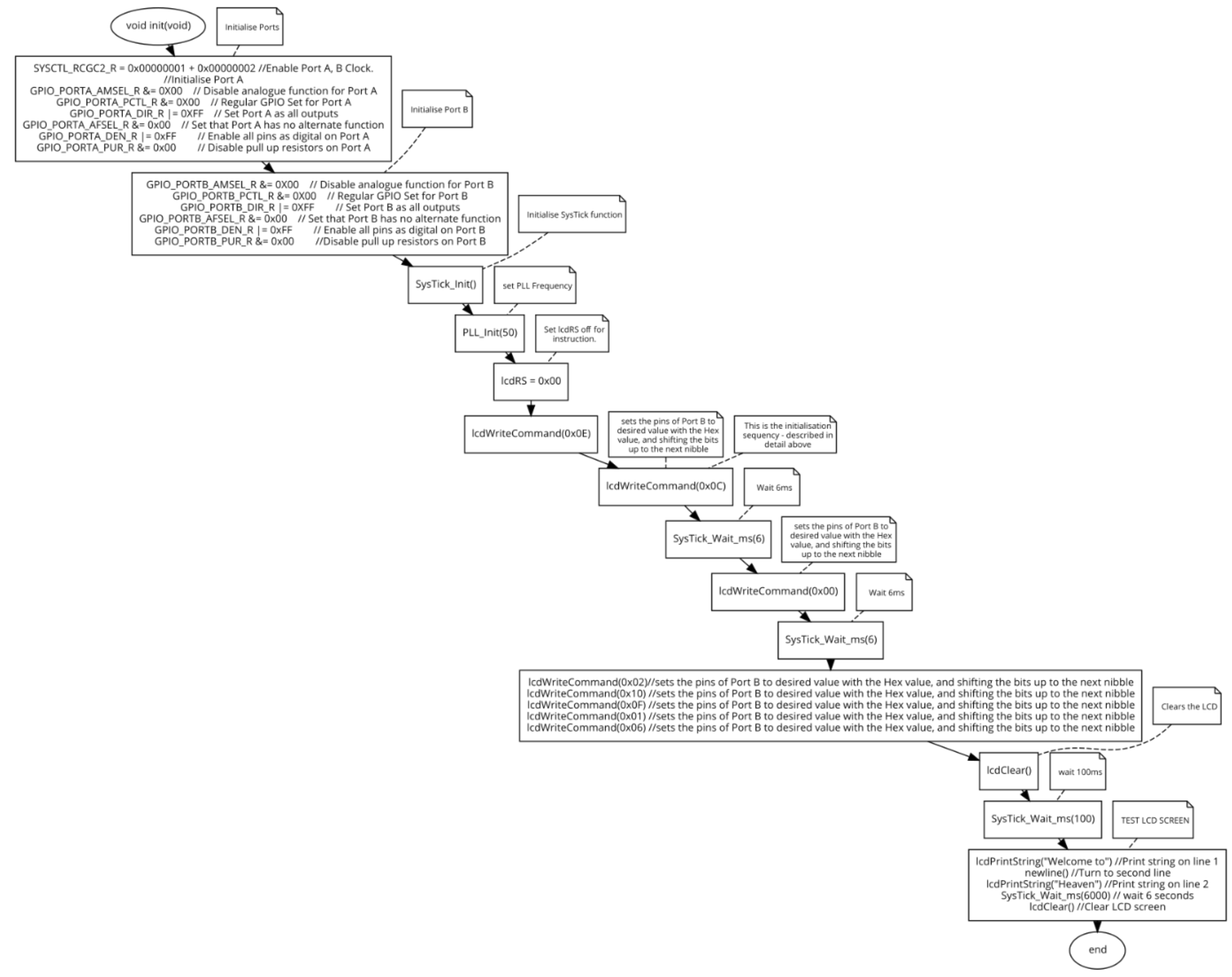


Figure 17. Clear Screen Function Flow Chart

Screen Functions

Screen Initialisation Function


void init(void)
Initialise Ports
Initialise Port B
Initialise SysTick function
SysTick_Init()
set PLL Frequency
PLL_Init(50)
Set lcdRS off for instruction.
lcdRS = 0x00
lcdWriteCommand(0x0E)
sets the pins of Port B to desired value with the Hex value, and shifting the bits up to the next nibble
This is the initialisation sequency - described in detail above
lcdWriteCommand(0x0C)
Wait 6ms
SysTick_Wait_ms(6)
sets the pins of Port B to desired value with the Hex value, and shifting the bits up to the next nibble
lcdWriteCommand(0x00)
Wait 6ms
SysTick_Wait_ms(6)
Clears the LCD
lcdClear()
wait 100ms
SysTick_Wait_ms(100)
TEST LCD SCREEN
end

```
SYSCTL_RCGC2_R = 0x00000001 + 0x00000002 //Enable Port A, B Clock.
//Initialise Port A
GPIO_PORTA_AMSEL_R &= 0X00    // Disable analogue function for Port A
GPIO_PORTA_PCTL_R &= 0X00    // Regular GPIO Set for Port A
GPIO_PORTA_DIR_R |= 0XFF    // Set Port A as all outputs
GPIO_PORTA_AFSEL_R &= 0x00    // Set that Port A has no alternate function
GPIO_PORTA_DEN_R |= 0xFF        // Enable all pins as digital on Port A
GPIO_PORTA_PUR_R &= 0x00        // Disable pull up resistors on Port A
```

```
GPIO_PORTB_AMSEL_R &= 0X00    // Disable analogue function for Port B
GPIO_PORTB_PCTL_R &= 0X00    // Regular GPIO Set for Port B
GPIO_PORTB_DIR_R |= 0XFF        // Set Port B as all outputs
GPIO_PORTB_AFSEL_R &= 0x00    // Set that Port B has no alternate function
GPIO_PORTB_DEN_R |= 0xFF        // Enable all pins as digital on Port B
GPIO_PORTB_PUR_R &= 0x00        //Disable pull up resistors on Port B
```

```
lcdWriteCommand(0x02)//sets the pins of Port B to desired value with the Hex value, and shifting the bits up to the next nibble
lcdWriteCommand(0x10) //sets the pins of Port B to desired value with the Hex value, and shifting the bits up to the next nibble
lcdWriteCommand(0x0F) //sets the pins of Port B to desired value with the Hex value, and shifting the bits up to the next nibble
lcdWriteCommand(0x01) //sets the pins of Port B to desired value with the Hex value, and shifting the bits up to the next nibble
lcdWriteCommand(0x06) //sets the pins of Port B to desired value with the Hex value, and shifting the bits up to the next nibble
```

```
lcdPrintString("Welcome to") //Print string on line 1
newline() //Turn to second line
lcdPrintString("Heaven") //Print string on line 2
SysTick_Wait_ms(6000) // wait 6 seconds
lcdClear() //Clear LCD screen
```

Figure 18. Screen initialisation Function Flow Chart

Write to Screen Function

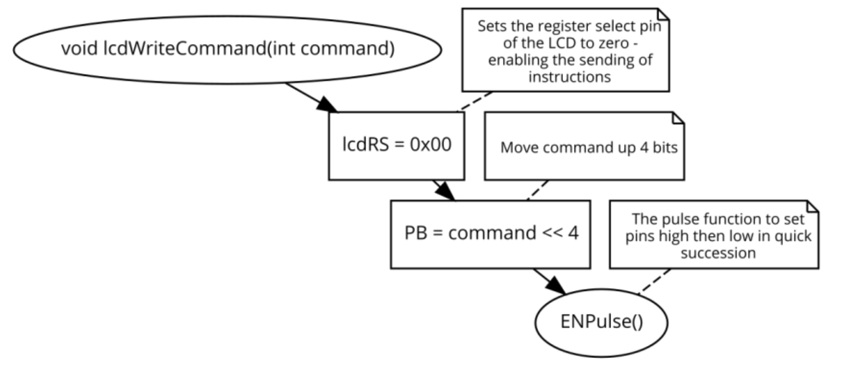

void lcdWriteCommand(int command)
Sets the register select pin of the LCD to zero - enabling the sending of instructions
lcdRS = 0x00
Move command up 4 bits
PB = command << 4
The pulse function to set pins high then low in quick succession
ENPulse()

Figure 19. Write to screen Function Flow Chart

Clear Screen Function

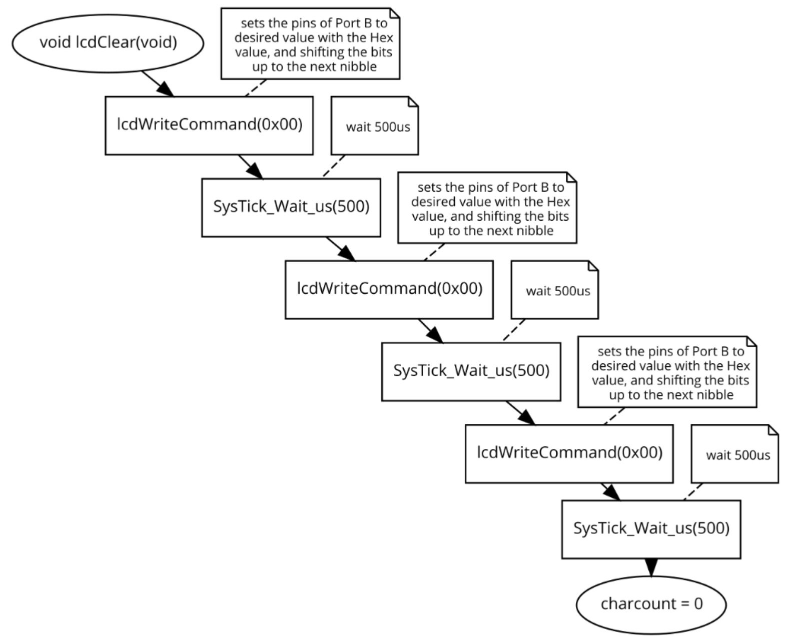

void lcdClear(void)
sets the pins of Port B to desired value with the Hex value, and shifting the bits up to the next nibble
lcdWriteCommand(0x00)
wait 500us
SysTick_Wait_us(500)
sets the pins of Port B to desired value with the Hex value, and shifting the bits up to the next nibble
lcdWriteCommand(0x00)
wait 500us
SysTick_Wait_us(500)
sets the pins of Port B to desired value with the Hex value, and shifting the bits up to the next nibble
lcdWriteCommand(0x00)
wait 500us
SysTick_Wait_us(500)
charcount = 0

Figure 20. LCD Clear Function Flow Chart

Information Pulse Function (ENPulse)

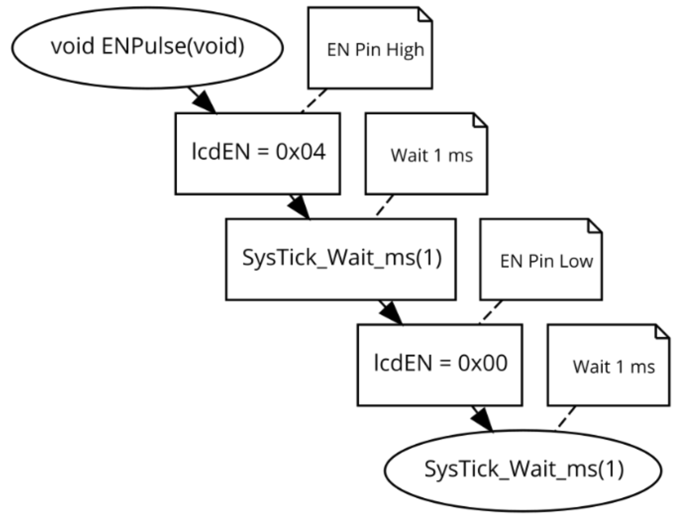


Figure 20. ENPulse Function Flow Chart

Screen Curser Go To Function

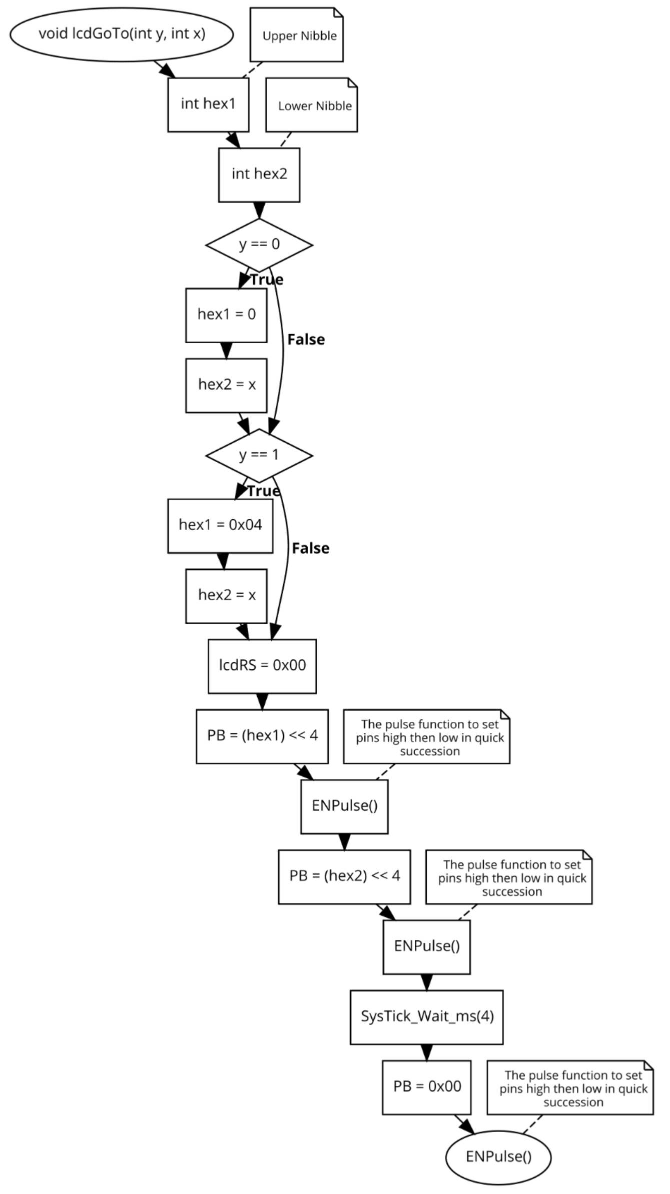

void lcdGoTo(int y, int x)
Upper Nibble
int hex1
Lower Nibble
int hex2
y == 0
True
False
hex1 = 0
hex2 = x
y == 1
True
False
hex1 = 0x04
hex2 = x
lcdRS = 0x00
PB = (hex1) << 4
The pulse function to set pins high then low in quick succession
ENPulse()
PB = (hex2) << 4
The pulse function to set pins high then low in quick succession
ENPulse()
SysTick_Wait_ms(4)
PB = 0x00
The pulse function to set pins high then low in quick succession
ENPulse()

Figure 21. Screen Curser Go To Function Flow Chart

Print Symbols to Screen Functions

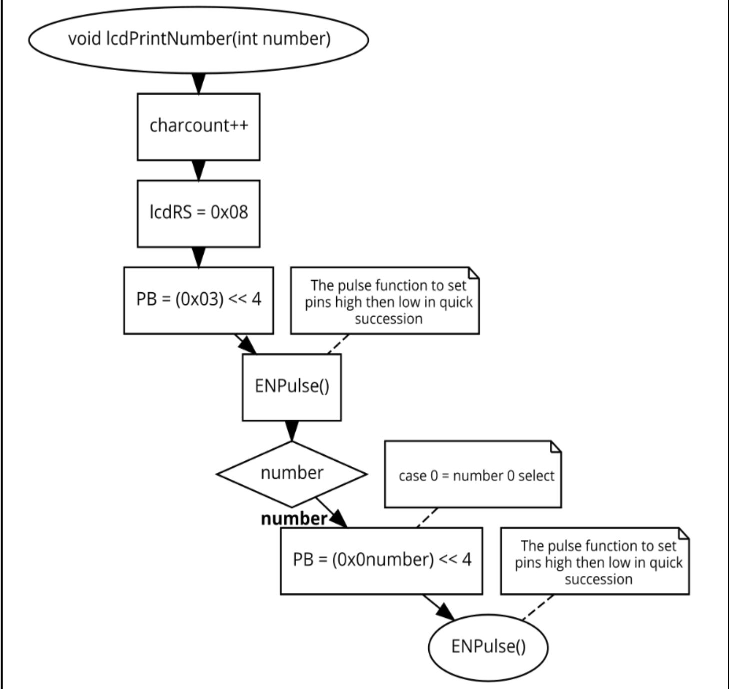

void lcdPrintNumber(int number)
charcount++
lcdRS = 0x08
PB = (0x03) << 4
The pulse function to set pins high then low in quick succession
ENPulse()
number
case 0 = number 0 select
number
PB = (0x0number) << 4
The pulse function to set pins high then low in quick succession
ENPulse()

Figure 22. Print Numerical Symbols to Screen Function Flow Chart

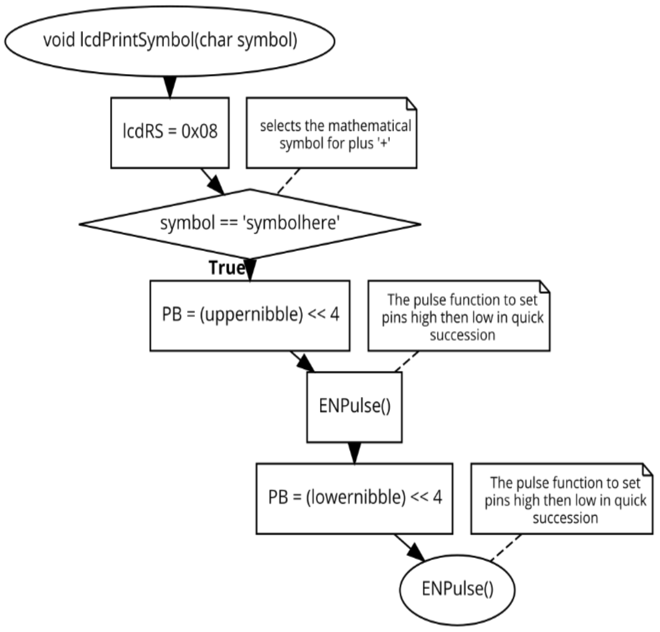

void lcdPrintSymbol(char symbol)
lcdRS = 0x08
selects the mathematical symbol for plus '+'
symbol == 'symbolhere'
True
PB = (uppernibble) << 4
The pulse function to set pins high then low in quick succession
ENPulse()
PB = (lowernibble) << 4
The pulse function to set pins high then low in quick succession
ENPulse()

Figure 23. Print Operand Symbols to Screen Function Flow Chart

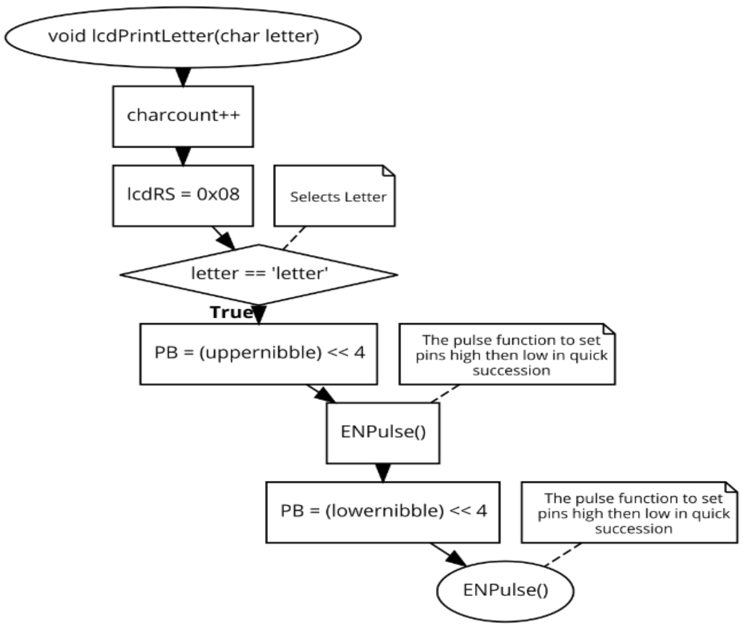

void lcdPrintLetter(char letter)
charcount++
lcdRS = 0x08
Selects Letter
letter == 'letter'
True
PB = (uppernibble) << 4
The pulse function to set pins high then low in quick succession
ENPulse()
PB = (lowernibble) << 4
The pulse function to set pins high then low in quick succession
ENPulse()

Figure 24. Print Letter Symbols to Screen Function Flow Chart

Print a Continuous String to the Screen Function

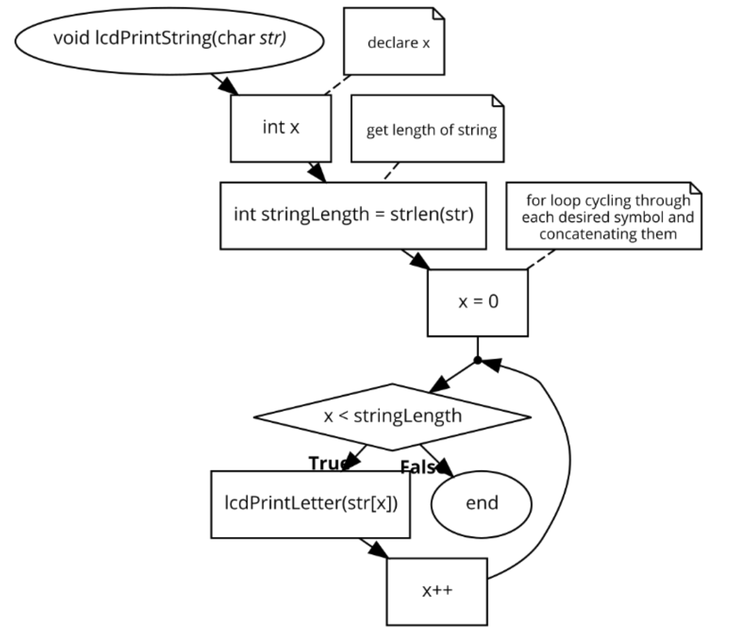

void lcdPrintString(char *str)*
declare x
int x
get length of string
int stringLength = strlen(str)
for loop cycling through each desired symbol and concatenating them
x = 0
x < stringLength
True
False
lcdPrintLetter(str[x])
end
x++

Figure 25. Print String to Screen Function Flow Chart

Testing & Issues

Several testing methods were implemented to ensure desired operation. A hardware approach was taken in the beginning, making use of an oscilloscope and multi-meter to probe the individual pins to guarantee that the code was operating as anticipated in terms of high and low timings, this was used heavily during the screen initialisation due to the specific wait lengths required. It was also useful for the identification of interference between the Keypad and Screen. Wherein when keys were pushed the screen would interpret the interference as transmitted data. The solution to this was to turn the keypad off before and LCD functions were carried out, isolating the two hardware components.

In conjunction with this, *'Print-F'* functions were used throughout the code. These are functions which display the value of variable as the code operators to the console, and they were used to check the code was operating as desired, by manipulating data as expected. However, an issue arose because the compiler did not interface with the console properly, the *'Print-F'* functions would cause it to crash. Our solution to this was to use an online compiler to build a simulation of the calculator which ran the code and the print functions to access the codes operation.

Several issues with the Keypad became apparent as the project progressed, one issue was the Microcontrollers Port D 0 not functioning, requiring the switch to Port A 4 (shown in the circuit diagram, figure 3). A Major issue was *'button bounce'* which is when one button was pressed the code would detect several key presses, intermittently interference would emulate other keys being pressed. To overcome this, delays were added to the code, in addition to breaks within the keypads For loops each time a button was pressed. This ensured that only one button press was detected at a time. Tuning was carried out on the delays as to not detract from usability.

Conclusion

The overall project was a success. The hardware provided and required functionality provided an interesting challenge allowing for the implementation of several innovative solutions. Having to use techniques to make the code more efficient because of the microcontrollers limitations forced us to adapt our techniques and develop our coding style. Despite frustration during the project at the issues that occurred, such as interference and the difficulty to initiate the screen, this forced us to utilise and bring together the full repertoire of the code and hardware debugging techniques we have learnt throughout our course, which otherwise would have remained separate.

In retrospect underestimating the scale of the project was our biggest hindrance although we could not have predicted the issues that would occur more time should have been set aside for the initial planning of the project. Opposed to jumping in straight away hoping to have it completed as quickly as possible. We would have used this to implement more complicated features, which would have involved employing additional hardware components.

References

[1] multicomp, “Keypad,” Farnell, 2012.
[2] Hitachi, Ltd, “HD44780U (LCD-II),” Chiyoda-ku, Tokyo 100-0004, Japan, 1998.
[3] Texas Instruments, “TM4C123GH6PM Microcontroller,” Texas Instruments, Dallas, Texas, 2013.