CONTENTS

1. System Flow Control **2**

2. Data Structures **2**

2.1 Status Register 3

2.1.1 Purpose 3

2.1.2 Structure 3

2.1.2.1 Mode Settings 4

2.1.3 Write To/Read From 4

2.2 LCD Queue 5

2.2.1 Element Description 6

2.2.2 Adding to Queue 6

2.2.3 Removing from Queue 6

2.2.4 Reading from Queue 6

2.3 Turntable String 7

2.3.1 Format 7

2.3.2 Iteration/Interpretation 7

3. Algorithms **8**

3.1 Queue 8

3.1.1 Appending 8

3.1.2 Removing 9

3.2 Turntable 10

3.3 Keypad 10

3.4 Ascii Conversion 11

3.4.1 Minutes and Seconds 11

4. Module Specification **12**

4.1 Timer0 – Keypad polling 12

4.2 Timer1 14

4.2.1 Beep timings on microwave finish 14

4.2.2 Turntable orientation updates 15

4.2.3 Motor state and timer updates 16

4.3 Timer2 17

4.3.1 LCD pipelining 17

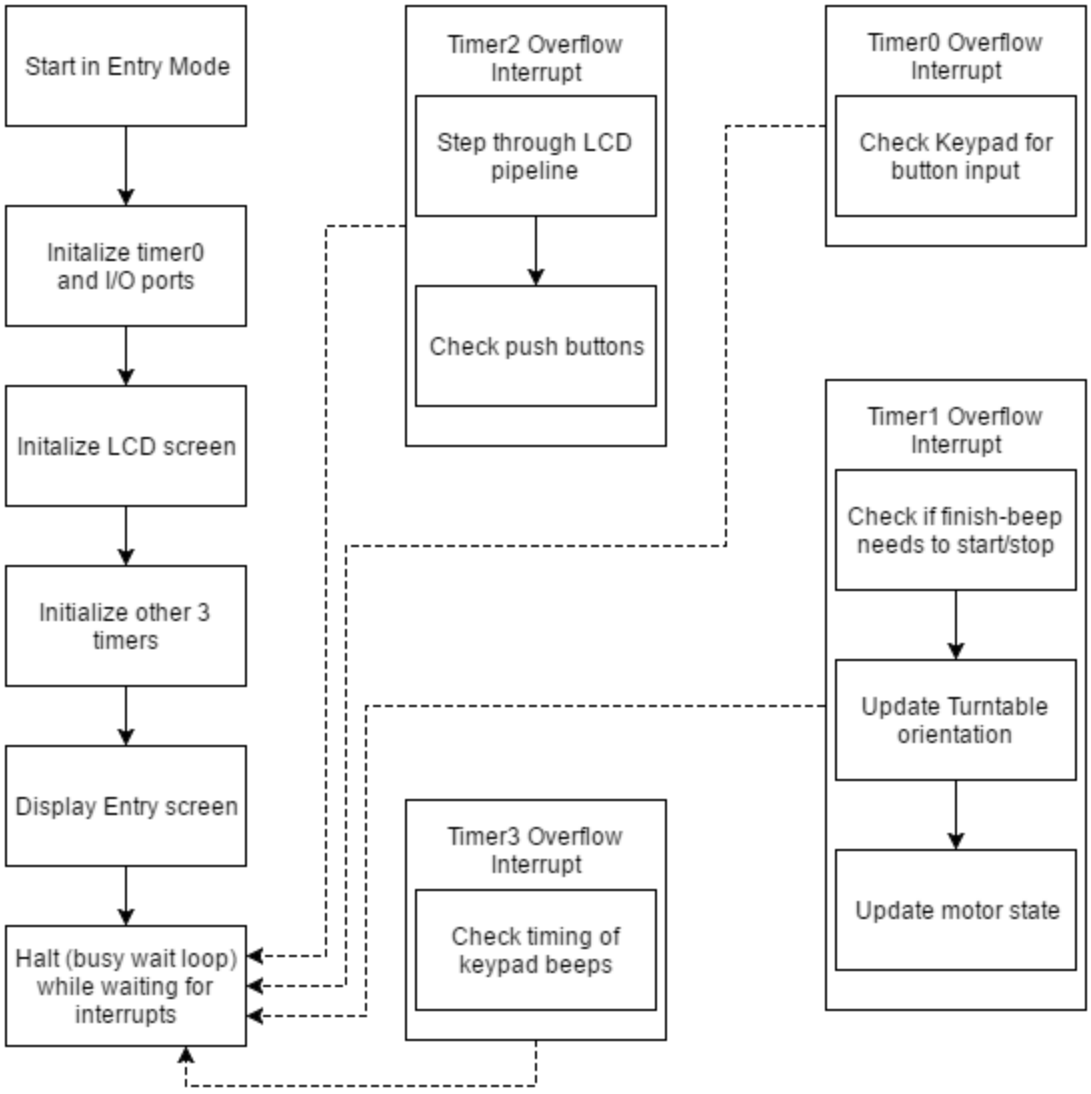
4.3.2 Push Buttons state check 18

4.3.2.1 Debouncing Method 19

4.4 Timer3 – Beep timings on button press 19

1. System Flow Control

The general structure of the code is fairly simple; after the initial startup period, the code runs entirely off interrupts, as seen in the figure below. It should be noted that some timer interrupts handle more than one module. More detail about each of the modules can be found in part 4. As we were not able to get the speaker working properly, we instead used the second topmost LED (which was not used in the assignment specification) as an analog for the speaker. When it turns on, the microwave is 'beeping'. This will be referred to as the 'beep LED' in the rest of the report.



2. Data Structures

2.1 Status Register

2.1.1 Purpose

The status register is used to store the current state of operation. It is a global register stored in r29 at all times. Comparisons with the status register allow for conditional execution. Since the emulator runs solely on interrupts, a global state was needed to be maintained.

Pseudocode Example:

```
if_in_mode 'D' ;door is open
Do not accept input from keypad
```

2.1.2 Structure

The high 6 bits of the status register have specified interpretations. The low 2 bits are reserved.

- O - Old state flag
    - This flag is altered when opening the door in order to know which state to return to once the door is closed again.
    - If the door is opened (PB1 pressed) in either Running or Paused modes, the flag is cleared. This signifies that, on close, the microwave will enter the Paused state. Otherwise, the flag is set and, on door close, the emulator is set to entry mode.
- K - Keypad Press Flag
    - A flag to prevent multiple inputs from the same keypad press.
    - If it is set, then no subsequent keypad presses will be accepted. The K flag will be cleared if a keypad scan is completed and no key presses are detected.
- D - Direction Flag
    - Represents the direction of turntable rotation. 0 represents clockwise and 1 anticlockwise.
    - Flipped each time the microwave starts running from entry mode. (Will not flip if paused and then resumed)
- Mn - Mode Setting
    - Stores the current execution state of the emulator. Allows for comparison and logic execution (Primarily using if_in_mode macro)
    - See 2.1.2.1 Mode Settings for possible combinations. Permutations not listed are reserved.

2.1.2.1 Mode Settings

M2	M1	M0	Mode
0	0	0	Running (R)
0	0	1	Entry (E)
0	1	0	Paused (P)
0	1	1	RESERVED
1	0	0	Finished (F)
1	0	1	RESERVED
1	1	0	Power Entry (X)
1	1	1	Door Open (D)

2.1.3 Write To/Read From

Upon a state change, the global status register must be updated. A macro (changeMode) to change the mode (the most common operation on the status register) was developed. Given a mode key (a single upper case letter - See 2.1.2.1) it will change the mode bits in the status register. Writing such as this was done at any time a user action should cause a state change.

Most logic depends on reads from the status register. A macro (if_in_mode) was developed to read the mode and can be used as a precursor to a branch command (brne/breq).

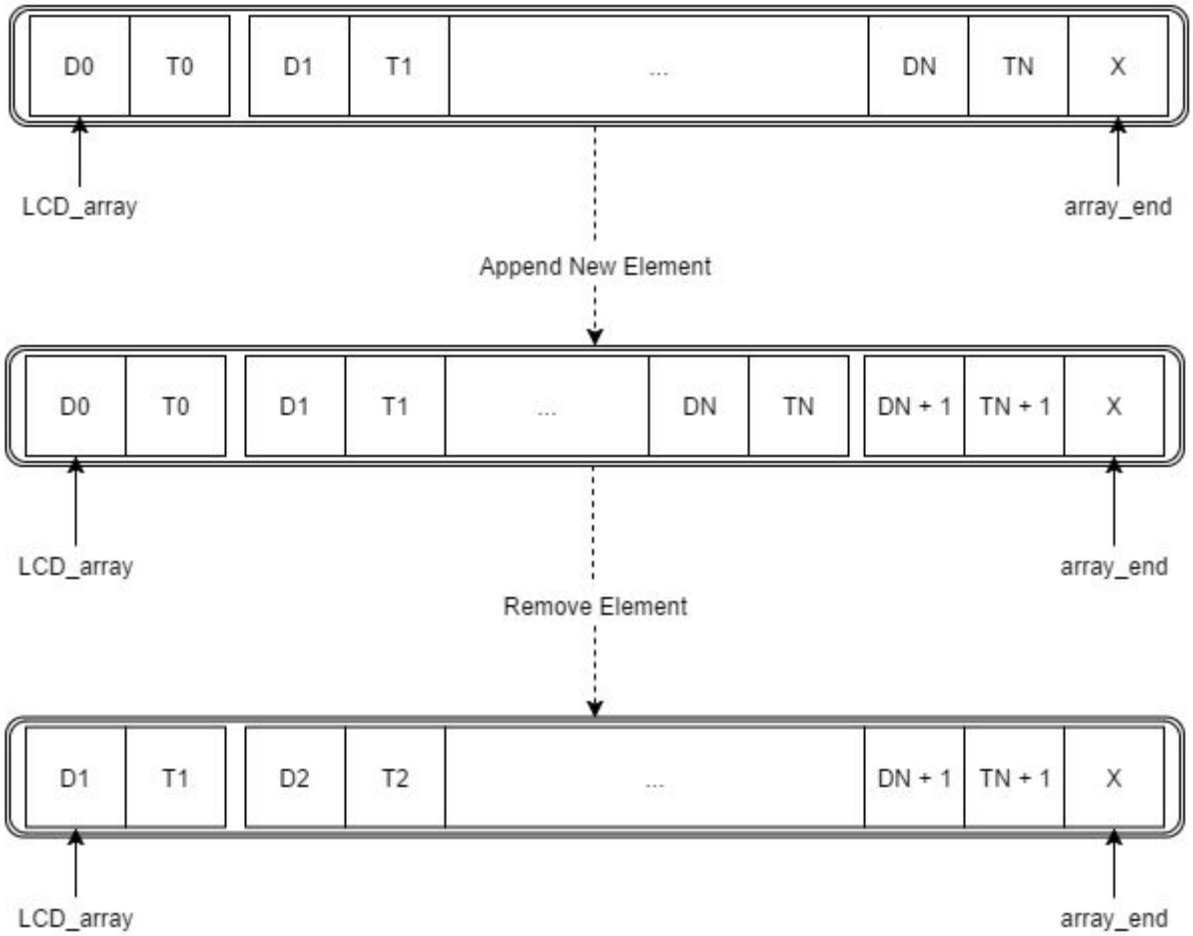
Implementation of pseudocode example from last page:

```
if_in_mode 'D' ;door is open
breq do_nothing
```

2.2 LCD Queue

A 5000 byte queue was used to store LCD commands prior to execution so the commands could be executed while other timings were being processed. The size was overestimated to allow for significant amount of input and since the mega2560 has a large data memory space (256kB). Realistically, this limit should never be reached and even if you were to try to reach it it might be physically impossible (without some help).

A pointer to the memory space after the final element in the queue was stored in the variable array_end.



2.2.1 Element Description

Each element of the queue consisted of two items. The command/data (Dn in Figure section 2.2) and a type description that differentiates the previous item as a command or data (Tn in Figure section 2.2) as Data and Commands have slightly different handling by the LCD.

Second Item meanings:

- 0 - Data
- 1 - Command
- 2 - The element has been processed and is now waiting for the LCD to be ready for the next element in the queue (For information on waiting loop see 4.1 LCD Module Description)

2.2.2 Adding to Queue

Standard queue operations were used to append new elements. Each new element was placed at the end of the queue, increasing the length by one and updating where in memory the next element will be inserted. Three macros; mov_lcd_data_a, do_lcd_data_a and do_lcd_command_a were developed to handle appending to the queue. All three are essentially the same, the only difference being how elements are identified (the second item of the element) and how data should be passed to the register. Two data macros were developed to support both passing the data as a literal (e.g ‘A’ - do_lcd_data_a) and passed within a register (e.g r16 - mov_lcd_data_a).

2.2.3 Removing from Queue

Elements are only removed from the queue when in the first position. When an element is removed, every other element is moved forward one position and the queue length reduced by one. A macro, queue_pop, was written to handle this process. If the queue is empty, this operation will have no effect. An element is only removed from the queue after it has been handled by the LCD (this will be covered in more depth in 4.1 LCD Module Description).

2.2.4 Reading from Queue

The only element ever read from the queue is that in the one in the first position. It is therefore possible to simply use the LCD_array (the label given to the start of the queue) and LCD_array+1 as the addresses to read from. After 10 interrupts (~1.3ms), if no LCD handling is taking place, an attempt to read the queue is made, resulting in one of two cases:

1. The end of queue pointer points to the start of the queue - The queue is empty
    a. Do no queue reading
2. The queue is not empty
    a. Begin the process of handling an LCD command or LCD data

After a queue read is complete, the element is removed and the queue updated.

2.3 Turntable String

The turntable has to make 3 full revolutions per minute (each character change represented an eighth of a revolution, or 2.5 seconds for each character change) while the microwave is running and persist through other modes, changing direction on finish or cancellation of operation.

2.3.1 Format

The turntable was implemented as the string ‘-/|``|/-’ in program memory. The backtick (‘`’) character was used as an alternative to the backslash (‘\’) character but for clarity the ‘\’ character will be used in this manual.

2.3.2 Iteration/Interpretation

The operation of the turntable was implemented in the overflow interrupt for timer1. A counter was made in data memory (num_overflow_int) which counted the number of interrupts and would trigger after it had reached 78 (roughly 2.5 seconds).

The Z pointer was used to point to the current character that the turntable was on, incrementing each time it had to display a new character and resetting back to the first character in the string once it had gone past the fourth character. To implement the turntable changing direction upon the microwave being stopped or the microwave finishing, the D flag on the status register would be examined and Z temporarily incremented by 4 as the turntable character is displayed to the screen (where the order of characters is reversed).

3. Algorithms

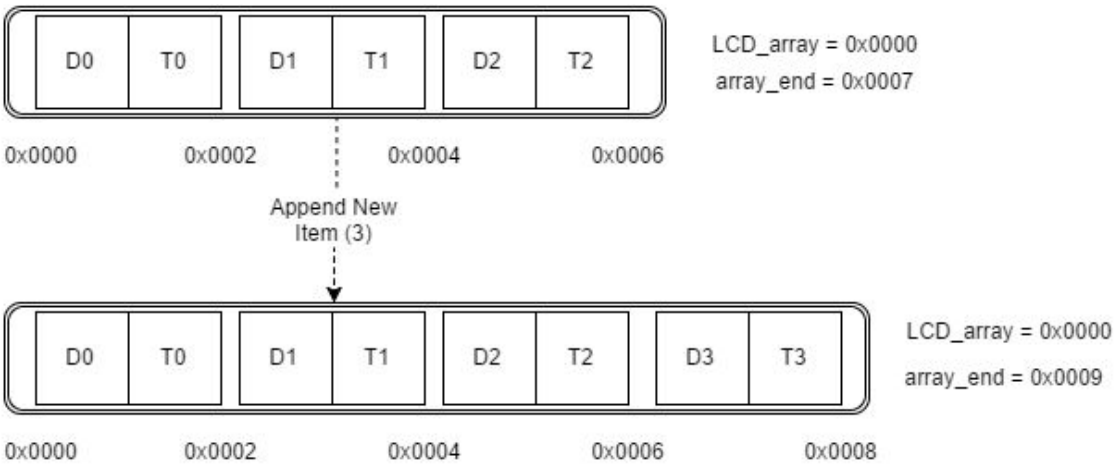
3.1 Queue

3.1.1 Appending

Appending to the queue is simple pointer arithmetic. The array_end pointer (to the memory address after the final element) is used as the address to store the new element. The array_end pointer is initially loaded into AVR's X pointer. The data/command is written first with a post increment of X, followed by the type, with another post increment. The new X pointer value is written out as the new value for array_end. This results in array_end being increased by 2 and once again pointing to the end of the queue.

Pseudocode:

```
Load array_end into x                ; x = array_end
Write data/command and increment x   ; x = array_end + 1
Write type and increment x           ; x = array_end + 2
Write x into array end               ; array_end=x=array_end+2
```

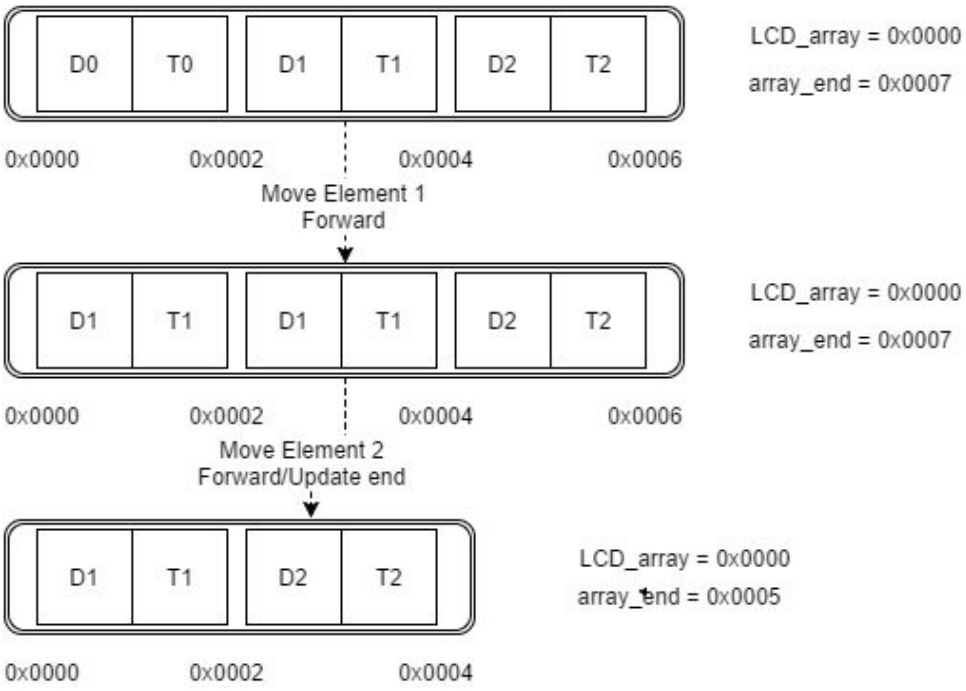


3.1.2 Removing

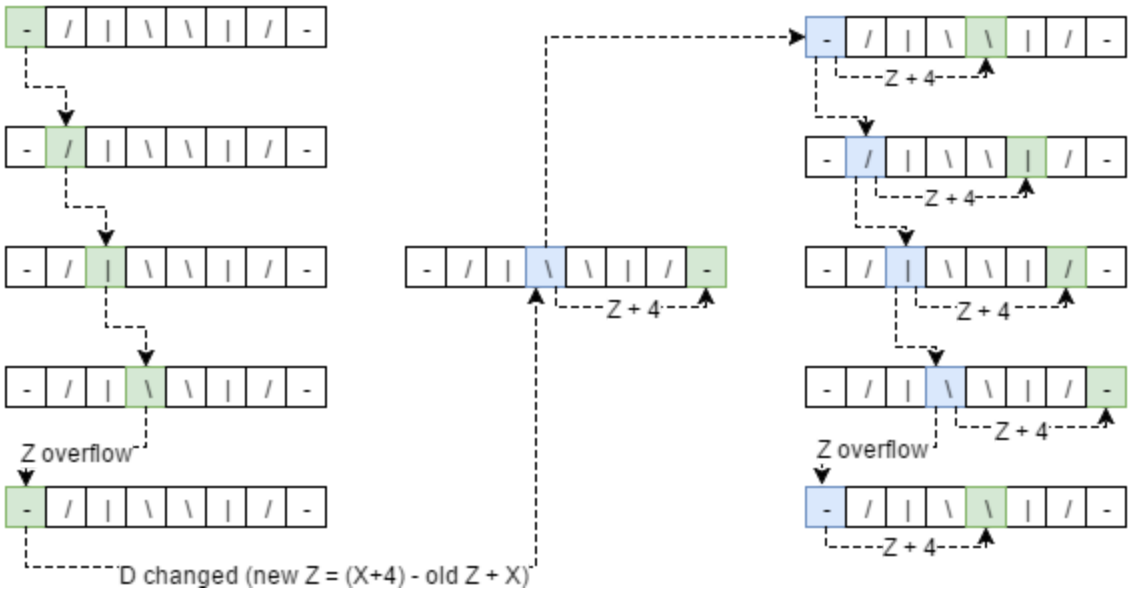
The algorithm for removing elements from the queue is a loop that shifts every element forward by one and decrements the array_end pointer two positions to the new end of the queue. Two pointers are used to iterate through the queue. The algorithm is modelled on a classic queue data structure remove.

Pseudocode:

```
Load Y with the start of array
Load Z with end of array
if Y == Z:
      Queue is empty. END
else if Y+2 == Z:
      One element in array
      Store Y into array_end. END
else:
      More than one element in the array
      Increment Y to the element ahead            ;Y = Y+2
      Load this element into X
      Decrement Y back to position it was in       ;Y = Y-3
      Load position pointed to by Y with values in X
      incrementing Y each time                          ;Y = Y+2
      Jump back to the else if above and repeat
```



3.2 Turntable



The turntable algorithm is fairly straightforward. The above diagram describes what happens to the turntable as it changes state 5 times before the D flag in the status register is changed (changing the direction of rotation of the turntable), the green square being the character being displayed by the LCD. This causes Z to point to a new location corresponding to the formula (where X is the address of the first character in the turntable string):

```
new Z = (X+4) - old Z + X
```

Now that D has been changed, when it comes time for the turntable character to display, the program will briefly add 4 to Z and display the character pointed to by that string before taking 4 away from Z again.

3.3 Keypad

The keypad was implemented in a way that was very similar to the example given in the lectures. Some pythonic pseudocode is provided below:

```
for columns 0 to 3:
    if button in column is pressed:
        for row in column:
            if button is pressed:
                break
```

For the numbers 1-9, those are determined by the formula:

```
num = row*3 + col
```

The others are identified on a case-by-case basis.

3.4 Ascii Conversion

The conversion from binary numbers to text characters is primarily done via a macro (bin_to_txt). This macro takes in a binary representation of a 2-digit number in one register and prints out the decimal representation of that number on the LCD wherever the cursor is.

It does this by repeatedly subtracting 10 from the input number (incrementing r19 each time this happens) until the input number is less than 10. Then it prints r19 before printing the remaining single-digit number. If passed a single-digit number, r19 will be 0 when printed, giving us the leading zeros we want.

3.4.1 Minutes and Seconds

The timer was split into two different registers—one for minutes and one for seconds. While the seconds register allows display of seconds up to 99, the system will endeavour to keep the seconds part under 60. When a new keypress is detected, the program resolves the new values for the minutes and seconds registers using these steps (in the label ‘convert_continue’):

1. If seconds = 0 and the minutes is less than 10, just add the pressed value to the seconds register.
2. Otherwise, check the minutes register.
3. If minutes = 0, multiply seconds by 10 and add the pressed value to the seconds register.
4. Else, if minutes >= 10, it means that we ignore the pressed value.
5. Finally, we multiply minutes by 10, count the number of 10’s in the seconds register, add those two values together, multiply seconds by 10 and add the pressed value to the seconds register.

4. Module Specification

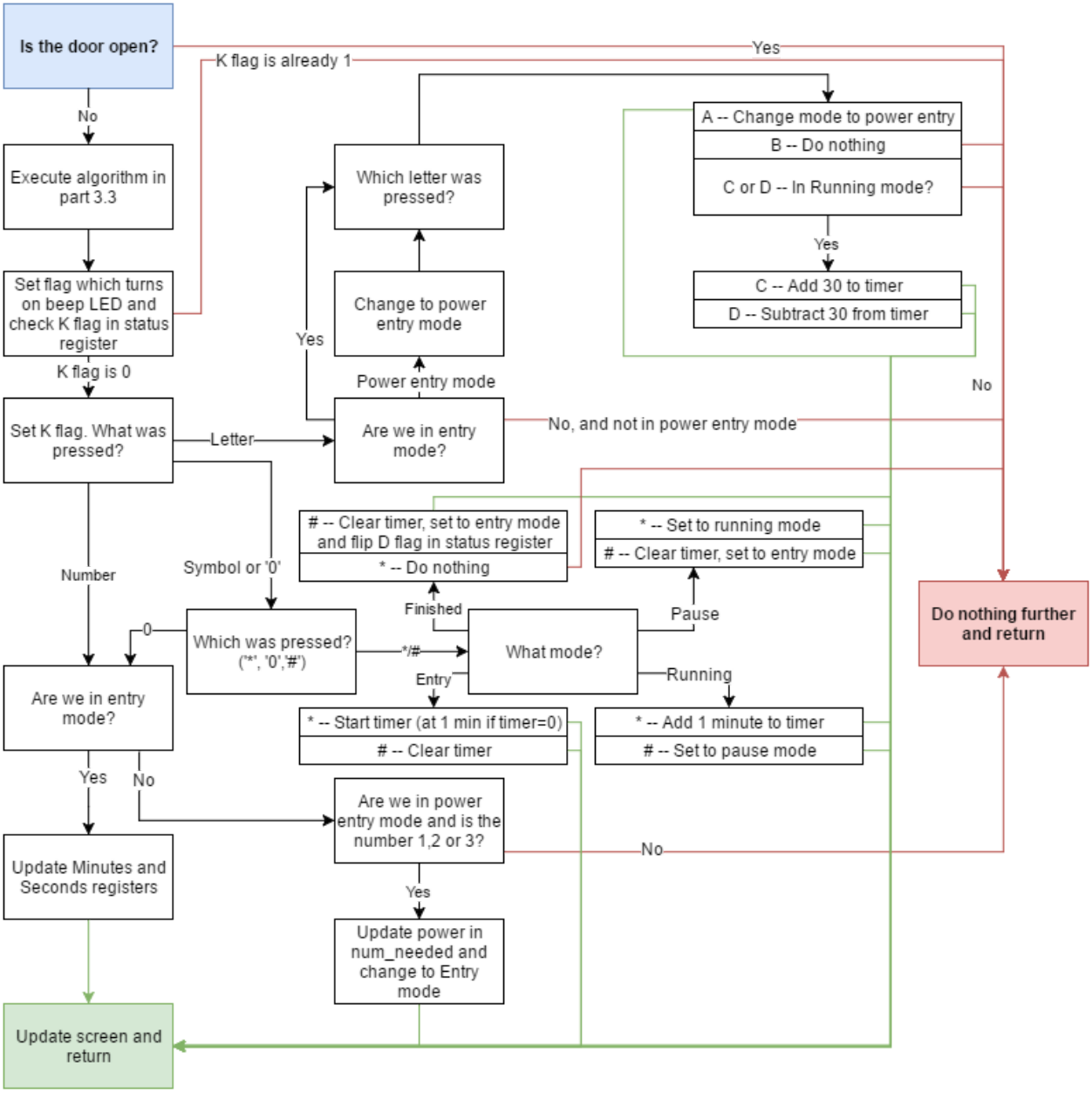
4.1 Timer0 – Keypad polling

After executing the algorithm detailed in part 3.3, the program then changes its behaviour depending on what state the program is in. If the door is open then all inputs are ignored. A detailed flowchart detailing the decision making process is below. It is fairly complicated, but the general idea is that the keypad figures out a broad category that it is dealing with (numbers, symbols or letters) before sorting those out individually based on the mode that the emulator is in.

Input Variables:

- Status register
    - K flag -- used for debouncing
    - Mode -- used to find out what mode the processor was in
- num_needed
    - The number of interrupts (in timer1) that we keep the motor on every second
- Minutes and Seconds -- used to store the timer value

A flowchart detailing the keypad decision tree is on the next page.


Is the door open?
Yes
K flag is already 1
No
Execute algorithm in part 3.3
Which letter was pressed?
A -- Change mode to power entry
B -- Do nothing
C or D -- In Running mode?
Yes
Set flag which turns on beep LED and check K flag in status register
Change to power entry mode
C -- Add 30 to timer
D -- Subtract 30 from timer
Yes
K flag is 0
Power entry mode
No
Set K flag. What was pressed?
Letter
Are we in entry mode?
No, and not in power entry mode
-- Clear timer, set to entry mode and flip D flag in status register
* -- Do nothing
* -- Set to running mode
# -- Clear timer, set to entry mode
Number
Symbol or '0'
Finished
Pause
Do nothing further and return
0
Which was pressed? ('*', '0','#')
*/#
What mode?
Entry
Running
Are we in entry mode?
* -- Start timer (at 1 min if timer=0)
# -- Clear timer
* -- Add 1 minute to timer
# -- Set to pause mode
Yes
No
Are we in power entry mode and is the number 1,2 or 3?
Update Minutes and Seconds registers
No
Yes
Update power in num_needed and change to Entry mode
Update screen and return

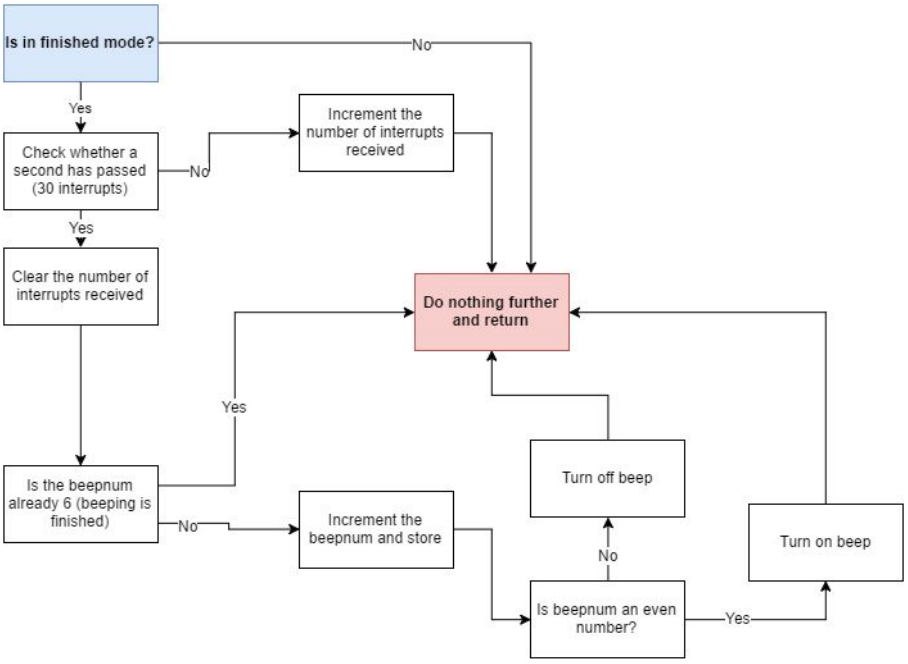
4.2 Timer1

4.2.1 Beep timings on microwave finish

The beep on microwave finish was an extended aim. On timer1 interrupt, the emulator checks whether it is in finished state and, if so handles the beeping process which would already have been commenced. The first beep is started as soon as the microwave enters finished mode. This means that after one second (30 timer1 interrupts) the beep is turned off.

Input Variables:

- num_overflow_int
    - Contains the number of timer1 interrupts for counting seconds
- beepnum
    - The position of the cycle so far. 3 beeps means 6 positions.
    - Odd numbers being a signal to beep, even to not.

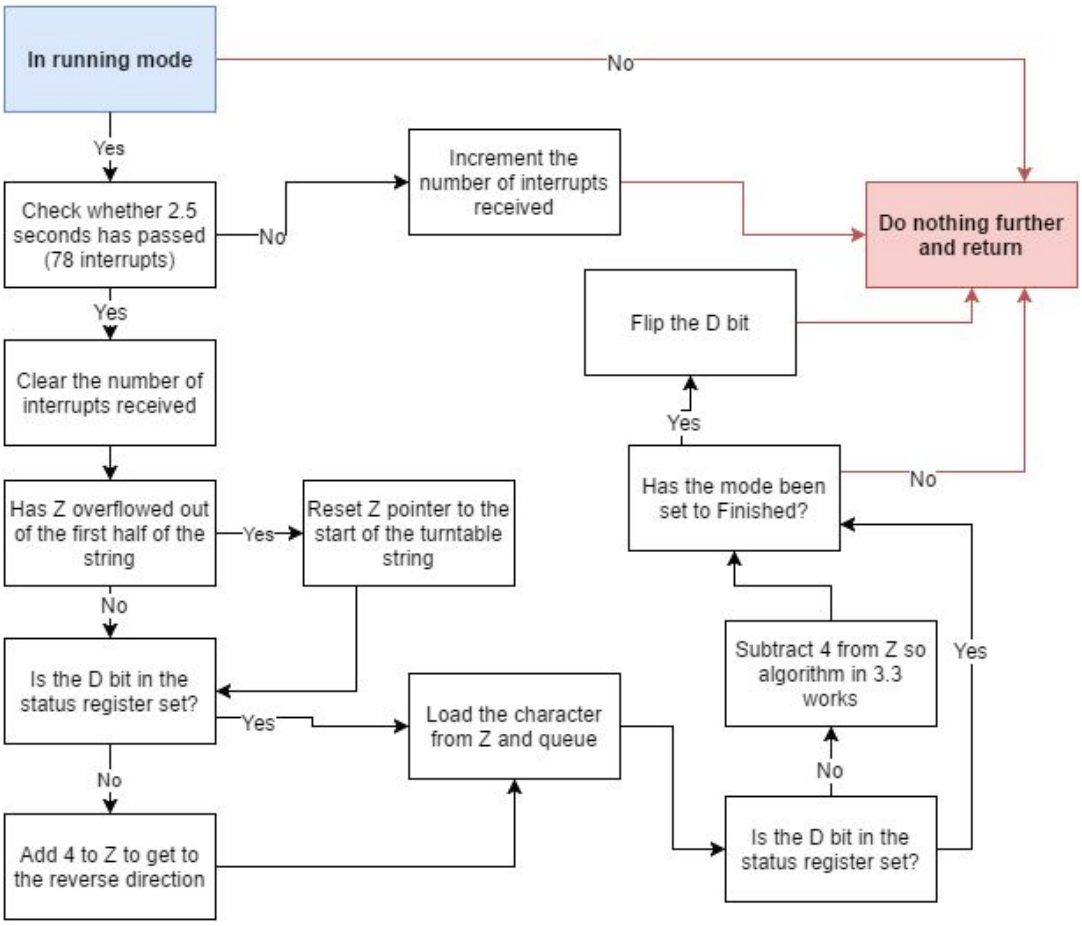


4.2.2 Turntable orientation updates

The turntable updates at approximately every 2.5 seconds (78 interrupts of timer1). The updates are done by the the algorithm explored in 3.2, simply iterating through a predefined string, appending each character to the LCD queue for display. This module will take in the D bit in the status register and the Z pointer (which points to the current position of the turntable) and turntable_seconds.

Input Variables:

- D bit in Status
    - Determines whether rotation is clockwise or anticlockwise
- Z pointer
    - Gives the current position of the turntable by pointing to a character in the turntable string
- turntable_seconds
    - Maintains the number of timer1 interrupts for the turntable

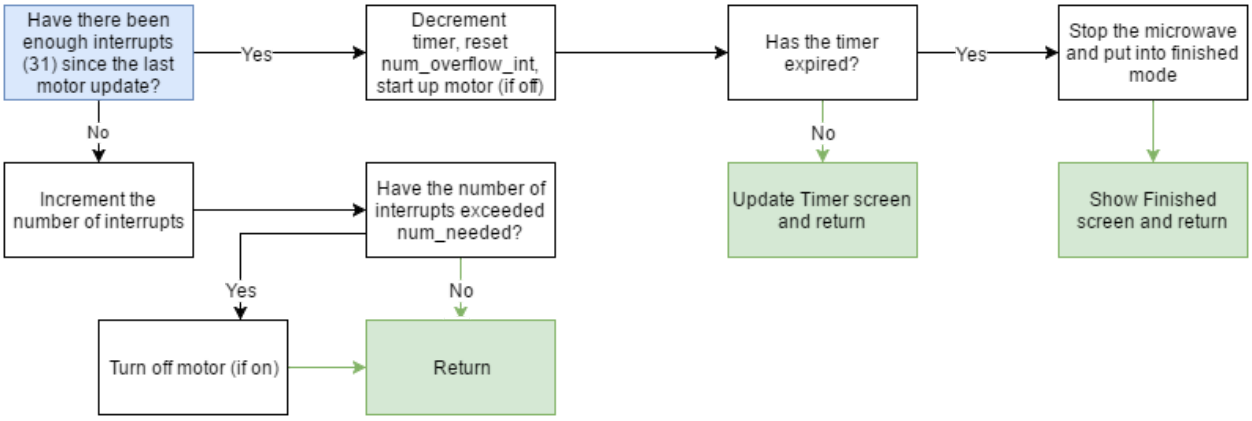


4.2.3 Motor state and timer updates

This module handles the turning on and off of the magnetron. It is only ever handled during running mode (determined using the status register). The time at which the motor is turned off depends on the power level (1, 2 or 3) which is represented in the num_needed variable. As this variable increases, the duty cycle of the motor will also increase. When the number of interrupts meets the num_needed variable, the motor will be switched off. The power level is changed in the keypad module (see section 4.1).

Input Variables:

- num_overflow_int
    - Keeps count of interrupts and every second (31 interrupts) it resets
- num_needed
    - The number of interrupts (in timer1) that we keep the motor on every second



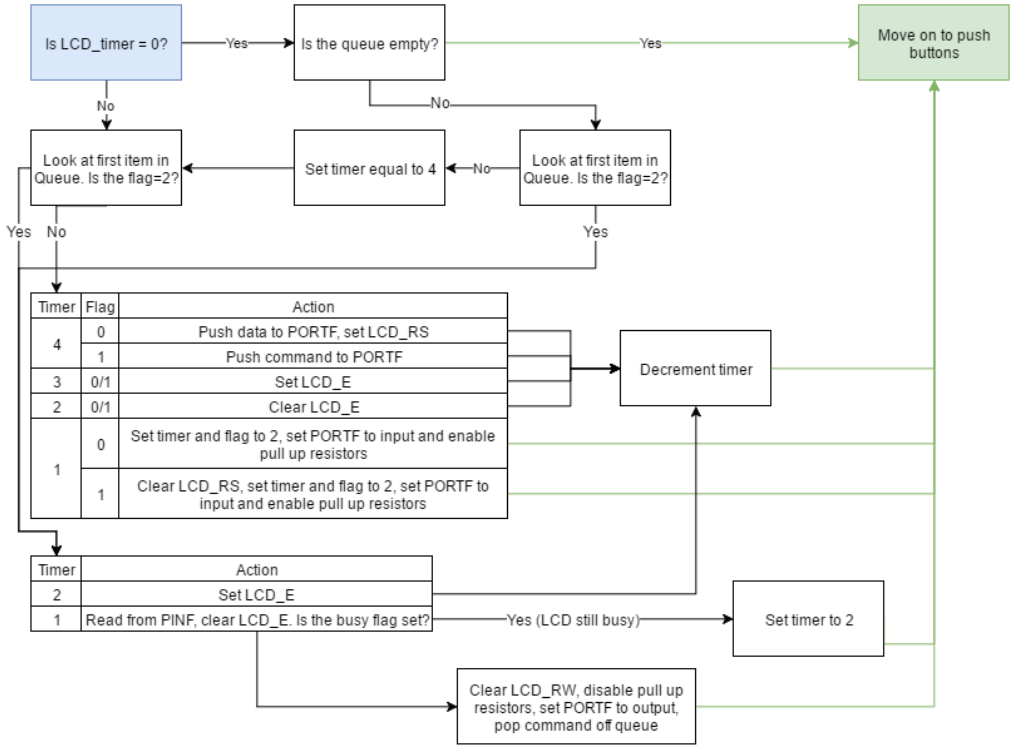
4.3 Timer2

4.3.1 LCD pipelining

In an effort to eliminate LCD processing time from interfering with other interrupts or inputs, we decided to do away with the standard LCD command structure, which put the entire processor on hold while waiting for the LCD to catch up. With this pipeline in place, the processor can continue dealing with other interrupts while the LCD is handling the commands fed to it. Handling the LCD is done in 4 stages, both in this pipelined approach and the standard busy waiting loop. The timer input variable is used to identify which stage of processing the current command is at. As commands are processed completely, they are popped off the front of the queue using the algorithm described in section 3.1.2. A command will not be popped until the LCD busy flag is cleared, indicating it is ready for the next command.

Input Variables:

- Command Array
    - First element is the argument passed to the LCD (8-bit variable)
    - Second element is the flag
        - Set to 0 if it is data
        - Set to 1 if it is a command
        - Set to 2 if it has been processed and is waiting
- Timer
    - Every ~1ms it counts down and further progresses the pipeline

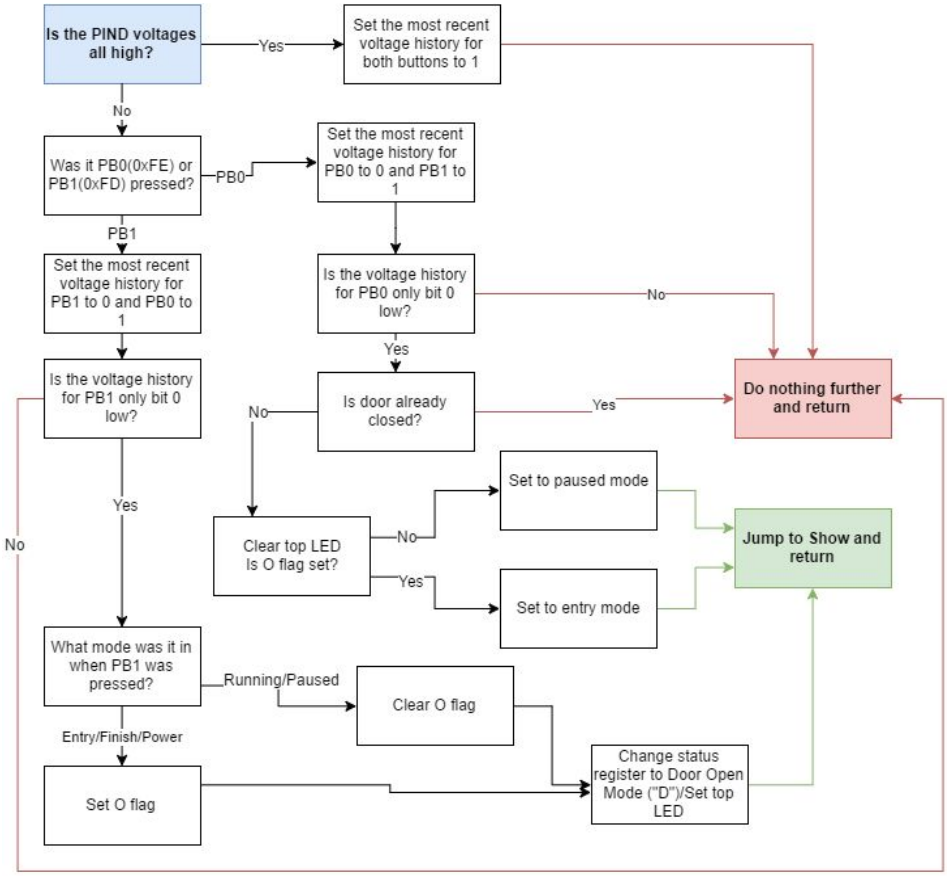


4.3.2 Push Buttons state check

The push button module is responsible for opening the door of the emulator and, when closing the door, returning it to a suitable mode. If the door is opened in either running or pause mode, it will be reset back to pause mode. Otherwise it will be reset to entry mode. To achieve this functionality, the Push Button module takes in the status register and pays particular notice to the O (old state) flag. This is the only module which reads and writes to this flag.

Input Variables:

- Status Register
    - Used to set the O flag if opening the door
    - Used to read the O flag if closing the door
- VoltagesSeen0
    - Voltage History of PB0
- VoltagesSeen1
    - Voltage History of PB1
- PIND
    - Input voltages of the button ports



4.3.2.1 Debouncing Method

Effectively, a shift register is used for debouncing both push buttons. An input voltage history is stored in data memory. This voltage history is updated every 10 interrupts of timer2.

The voltages from the pins in PORTD (the push button port) are read and the history updated accordingly. If the voltage on the pin corresponding to PB0 is low, then a 0 is shifted into bit 0 of the voltage history byte. If the voltage is high, a 1 is shifted into bit 0. A button press is only acknowledged if it is the first instance of a low voltage in the corresponding buttons history (ie if the voltage history is 0xFE).

4.4 Timer3 – Beep timings on button press

This is the simplest interrupt. Whenever a button is pressed by the keypad, the ‘untouched’ flag is enabled, the beep LED is turned on and the timer is reset (this happens in the keypad section). When the timer interrupts 2 times after being reset, ~250ms will have passed and the beep LED will be turned off.